Program Construction and Verification Components Based on Kleene Algebra

Victor B. F. Gomes and Georg Struth

March 17, 2025

Abstract

Variants of Kleene algebra support program construction and verification by algebraic reasoning. This entry provides a verification component for Hoare logic based on Kleene algebra with tests, verification components for weakest preconditions and strongest postconditions based on Kleene algebra with domain and a component for step-wise refinement based on refinement Kleene algebra with tests. In addition to these components for the partial correctness of while programs, a verification component for total correctness based on divergence Kleene algebras and one for (partial correctness) of recursive programs based on domain quantales are provided. Finally we have integrated memory models for programs with pointers and a program trace semantics into the weakest precondition component.

Contents

1 Introductory Remarks

These Isabelle theories provide program construction and verification components for simpe while programs based on variants of Kleene algebra with tests and Kleene algebra with domain, as well as a component for parameterless recursive programs based on domain quantales. The general approach consists in using the algebras for deriving verification conditions for the control flow of programs. They are linked by formal soundness proofs with denotational program semantics of the store and data domain—here predominantly with a relational semantics. Assignment laws can then be derived in this semantics. Program construction and verification tasks are performed within the concrete semantics as well; structured syntax for programs could easily be added, but is not provided at the moment.

All components are correct by construction relative to Isabelle's small trustworthy core, as our soundness proofs make the axiomatic extensions provided by the algebras consistent with respect to it.

The main components are integrated into previous AFP entries for Kleene algebras [3], Kleene algebras with tests [1] and Kleene algebras with domain [5]. As an overview and perhaps for educational purposes, we have also added two standalone components based on Hoare logic and weakest (liberal) preconditions that use only Isabelle's main libraries.

Background information on the general approach and the first main component, which is based on Kleene algebra with tests, can be found in [2]. An introduction to Kleene algebra with domain is given in [4]; a paper describing the corresponding verification component in detail is in preparation.

We are planning to add further components and expand and restructure the existing ones in the future. We would like to invite anyone interested in the algebraic approach to collaborate with us on these and contribute to this project.

2 Two Standalone Components

theory *VC-KAT-scratch*
  imports *Main*
begin

2.1 Component Based on Kleene Algebra with Tests

This component supports the verification and step-wise refinement of simple while programs in a partial correctness setting.

2.1.1 KAT: Definition and Basic Properties

notation *times* (**infixl** ‹·› *70*)

class *plus-ord* = *plus* + *ord* +
  assumes *less-eq-def*: $x \leq y \longleftrightarrow x + y = y$
  and *less-def*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$

class *dioid* = *semiring* + *one* + *zero* + *plus-ord* +
  assumes *add-idem* [*simp*]: $x + x = x$
  and *mult-onel* [*simp*]: $1 \cdot x = x$
  and *mult-oner* [*simp*]: $x \cdot 1 = x$
  and *add-zerol* [*simp*]: $0 + x = x$
  and *annil* [*simp*]: $0 \cdot x = 0$
  and *annir* [*simp*]: $x \cdot 0 = 0$

begin

subclass *monoid-mult*
  by (*standard*, *simp-all*)

subclass *order*
  apply (*standard*, *simp-all add*: *less-def less-eq-def add-commute*)

apply *force*
by (*metis add-assoc*)

lemma *mult-isol*: $x \leq y \Longrightarrow z \cdot x \leq z \cdot y$
by (*metis distrib-left less-eq-def*)

lemma *mult-isor*: $x \leq y \Longrightarrow x \cdot z \leq y \cdot z$
by (*metis distrib-right less-eq-def*)

lemma *add-iso*: $x \leq y \Longrightarrow x + z \leq y + z$
by (*metis* (*no-types*, *lifting*) *abel-semigroup.commute add.abel-semigroup-axioms add.semigroup-axioms add-idem less-eq-def semigroup.assoc*)

lemma *add-lub*: $x + y \leq z \longleftrightarrow x \leq z \wedge y \leq z$
by (*metis add-assoc add.left-commute add-idem less-eq-def*)

end

class *kleene-algebra* = *dioid* +
fixes *star* :: $'a \Rightarrow 'a$ (‹-*› [*101*] *100*)
assumes *star-unfoldl*: $1 + x \cdot x^\star \leq x^\star$
and *star-unfoldr*: $1 + x^\star \cdot x \leq x^\star$
and *star-inductl*: $z + x \cdot y \leq y \Longrightarrow x^\star \cdot z \leq y$
and *star-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot x^\star \leq y$

begin

lemma *star-sim*: $x \cdot y \leq z \cdot x \Longrightarrow x \cdot y^\star \leq z^\star \cdot x$
proof −
assume $x \cdot y \leq z \cdot x$
hence $x + z^\star \cdot x \cdot y \leq x + z^\star \cdot z \cdot x$
by (*metis add-lub distrib-left eq-refl less-eq-def mult-assoc*)
also have ... $\leq z^\star \cdot x$
using *add-lub mult-isor star-unfoldr* **by** *fastforce*
finally show *?thesis*
by (*simp add*: *star-inductr*)
qed

end

class *kat* = *kleene-algebra* +
fixes *at* :: $'a \Rightarrow 'a$
assumes *test-one* [*simp*]: *at* (*at* *1*) = *1*
and *test-mult* [*simp*]: *at* (*at* (*at* (*at x*) · *at* (*at y*))) = *at* (*at y*) · *at* (*at x*)
and *test-mult-comp* [*simp*]: *at x* · *at* (*at x*) = *0*
and *test-de-morgan*: *at x* + *at y* = *at* (*at* (*at x*) · *at* (*at y*))

begin

definition *t-op* :: $'a \Rightarrow 'a$ (‹*t-*› [*100*] *101*) **where**
$t\ x = at\ (at\ x)$

lemma *t-n* [*simp*]: $t\ (at\ x) = at\ x$
by (*metis add-idem test-de-morgan test-mult t-op-def*)

lemma *t-comm*: $t\ x \cdot t\ y = t\ y \cdot t\ x$
by (*metis add-commute test-de-morgan test-mult t-op-def*)

lemma *t-idem* [*simp*]: $t\ x \cdot t\ x = t\ x$
by (*metis add-idem test-de-morgan test-mult t-op-def*)

lemma *t-mult-closed* [*simp*]: $t\ (t\ x \cdot t\ y) = t\ x \cdot t\ y$
using *t-comm t-op-def* **by** *auto*

2.1.2 Propositional Hoare Logic

definition *H* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$H\ p\ x\ q \longleftrightarrow t\ p \cdot x \leq x \cdot t\ q$

definition *if-then-else* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹*if - then - else - fi*› [*64*,*64*,*64*] *63*)
where
if p then x else y fi $= t\ p \cdot x + at\ p \cdot y$

definition *while* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹*while - do - od*› [*64*,*64*] *63*) **where**
while p do x od $= (t\ p \cdot x)^\star \cdot at\ p$

definition *while-inv* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹*while - inv - do - od*› [*64*,*64*,*64*] *63*)
where
while p inv i do x od = *while p do x od*

lemma *H-skip*: $H\ p\ 1\ p$
by (*simp add*: *H-def*)

lemma *H-cons*: $t\ p \leq t\ p' \Longrightarrow t\ q' \leq t\ q \Longrightarrow H\ p'\ x\ q' \Longrightarrow H\ p\ x\ q$
by (*meson H-def mult-isol mult-isor order.trans*)

lemma *H-seq*: $H\ r\ y\ q \Longrightarrow H\ p\ x\ r \Longrightarrow H\ p\ (x \cdot y)\ q$
proof −
assume *h1*: $H\ p\ x\ r$ **and** *h2*: $H\ r\ y\ q$
hence *h3*: $t\ p \cdot x \leq x \cdot t\ r$ **and** *h4*: $t\ r \cdot y \leq y \cdot t\ q$
using *H-def* **apply** *blast* **using** *H-def h2* **by** *blast*
hence $t\ p \cdot x \cdot y \leq x \cdot t\ r \cdot y$
using *mult-isor* **by** *blast*
also have ... $\leq x \cdot y \cdot t\ q$
by (*simp add*: *h4 mult-isol mult-assoc*)
finally show *?thesis*
by (*simp add*: *H-def mult-assoc*)
qed

lemma *H-cond*: $H\ (t\ p \cdot t\ r)\ x\ q \Longrightarrow H\ (t\ p \cdot at\ r)\ y\ q \Longrightarrow H\ p\ (\mathit{if}\ r\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi})\ q$
proof –
  assume *h1*: $H\ (t\ p \cdot t\ r)\ x\ q$ **and** *h2*: $H\ (t\ p \cdot at\ r)\ y\ q$
  hence *h3*: $t\ r \cdot t\ p \cdot t\ r \cdot x \leq t\ r \cdot x \cdot t\ q$ **and** *h4*: $at\ r \cdot t\ p \cdot at\ r \cdot y \leq at\ r \cdot y \cdot t\ q$
    by (*simp add*: *H-def mult-isol mult-assoc*, *metis H-def h2 mult-isol mult-assoc t-mult-closed t-n*)
  hence *h5*: $t\ p \cdot t\ r \cdot x \leq t\ r \cdot x \cdot t\ q$ **and** *h6*: $t\ p \cdot at\ r \cdot y \leq at\ r \cdot y \cdot t\ q$
    by (*simp add*: *mult-assoc t-comm*, *metis h4 mult-assoc t-comm t-idem t-n*)
  have $t\ p \cdot (t\ r \cdot x + at\ r \cdot y) = t\ p \cdot t\ r \cdot x + t\ p \cdot at\ r \cdot y$
    by (*simp add*: *distrib-left mult-assoc*)
  also have ... $\leq t\ r \cdot x \cdot t\ q + t\ p \cdot at\ r \cdot y$
    using *h5 add-iso* **by** *blast*
  also have ... $\leq t\ r \cdot x \cdot t\ q + at\ r \cdot y \cdot t\ q$
    by (*simp add*: *add-commute h6 add-iso*)
  finally show *?thesis*
    by (*simp add*: *H-def if-then-else-def distrib-right*)
qed

lemma *H-loop*: $H\ (t\ p \cdot t\ r)\ x\ p \Longrightarrow H\ p\ (\mathit{while}\ r\ \mathit{do}\ x\ \mathit{od})\ (t\ p \cdot at\ r)$
proof –
  assume $H\ (t\ p \cdot t\ r)\ x\ p$
  hence $t\ r \cdot t\ p \cdot t\ r \cdot x \leq t\ r \cdot x \cdot t\ p$
    by (*metis H-def distrib-left less-eq-def mult-assoc t-mult-closed*)
  hence $t\ p \cdot t\ r \cdot x \leq t\ r \cdot x \cdot t\ p$
    by (*simp add*: *mult-assoc t-comm*)
  hence $t\ p \cdot (t\ r \cdot x)^\star \cdot at\ r \leq (t\ r \cdot x)^\star \cdot t\ p \cdot at\ r$
    by (*metis mult-isor star-sim mult-assoc*)
  hence $t\ p \cdot (t\ r \cdot x)^\star \cdot at\ r \leq (t\ r \cdot x)^\star \cdot at\ r \cdot t\ p \cdot at\ r$
    by (*metis mult-assoc t-comm t-idem t-n*)
  thus *?thesis*
    by (*metis H-def mult-assoc t-mult-closed t-n while-def*)
qed

lemma *H-while-inv*: $t\ p \leq t\ i \Longrightarrow t\ i \cdot at\ r \leq t\ q \Longrightarrow H\ (t\ i \cdot t\ r)\ x\ i \Longrightarrow H\ p\ (\mathit{while}\ r\ \mathit{inv}\ i\ \mathit{do}\ x\ \mathit{od})\ q$
  by (*metis H-cons H-loop t-mult-closed t-n while-inv-def*)

end

2.1.3 Soundness and Relation KAT

notation *relcomp* (**infixl** ‹;› *70*)

interpretation *rel-d*: *dioid Id* {} (∪) (;) (⊆) (⊂)
  by (*standard*, *auto*)

lemma (**in** *dioid*) *power-inductl*: $z + x \cdot y \leq y \Longrightarrow x \mathbin{\hat{}} i \cdot z \leq y$
  apply (*induct i*; *clarsimp simp add*: *add-lub*)
  by (*metis local.dual-order.trans local.mult-isol mult-assoc*)

lemma (**in** *dioid*) *power-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot x \mathbin{\hat{}} i \leq y$
  apply (*induct i*; *clarsimp simp add*: *add-lub*)
  proof −
    fix i
    assume $z \cdot x \mathbin{\hat{}} i \leq y$ $z \leq y$ $y \cdot x \leq y$
    hence $(z \cdot x \mathbin{\hat{}} i) \cdot x \leq y$
      using *local.dual-order.trans local.mult-isor* **by** *blast*
    thus $z \cdot (x \cdot x \mathbin{\hat{}} i) \leq y$
      by (*simp add*: *mult-assoc local.power-commutes*)
  qed

lemma *power-is-relpow*: *rel-d.power* X i $= X \mathbin{\hat{}\hat{}} i$
  by (*induct i*, *simp-all add*: *relpow-commute*)

lemma *rel-star-def*: $X\hat{}* = (\bigcup i.$ *rel-d.power* X $i)$
  by (*simp add*: *power-is-relpow rtrancl-is-UN-relpow*)

lemma *rel-star-contl*: X ; $Y\hat{}* = (\bigcup i.$ X ; *rel-d.power* Y $i)$
  by (*simp add*: *rel-star-def relcomp-UNION-distrib*)

lemma *rel-star-contr*: $X\hat{}*$; $Y = (\bigcup i.$ (*rel-d.power* X i) ; $Y)$
  by (*simp add*: *rel-star-def relcomp-UNION-distrib2*)

definition *rel-at* :: *'a rel* ⇒ *'a rel* **where**
  rel-at X = *Id* $\cap - X$

interpretation *rel-kat*: *kat Id* {} (∪) (;) (⊆) (⊂) *rtrancl rel-at*
  apply *standard*
  apply *auto*[*2*]
  by (*auto simp*: *rel-star-contr rel-d.power-inductl rel-star-contl SUP-least rel-d.power-inductr rel-at-def*)

2.1.4 Embedding Predicates in Relations

type-synonym *'a pred* = *'a* ⇒ *bool*

abbreviation *p2r* :: *'a pred* ⇒ *'a rel* (‹⌈-⌉›) **where**
  $\lceil P \rceil \equiv \{(s,s) \mid s.\ P\ s\}$

lemma *t-p2r* [*simp*]: *rel-kat.t-op* $\lceil P \rceil = \lceil P \rceil$
  by (*auto simp add*: *rel-kat.t-op-def rel-at-def*)

lemma *p2r-neg-hom* [*simp*]: *rel-at* $\lceil P \rceil = \lceil \lambda s.\ \neg\ P\ s \rceil$
  by (*auto simp*: *rel-at-def*)

lemma *p2r-conj-hom* [*simp*]: $\lceil P\rceil \cap \lceil Q\rceil = \lceil\lambda s.\ \ P\ s \wedge\ Q\ s\rceil$
  by *auto*

lemma *p2r-conj-hom-var* [*simp*]: $\lceil P\rceil\ ;\ \lceil Q\rceil = \lceil\lambda s.\ P\ s \wedge\ Q\ s\rceil$
  by *auto*

lemma *p2r-disj-hom* [*simp*]: $\lceil P\rceil \cup \lceil Q\rceil = \lceil\lambda s.\ P\ s \vee\ Q\ s\rceil$
  by *auto*

lemma *impl-prop* [*simp*]: $\lceil P\rceil \subseteq \lceil Q\rceil \longleftrightarrow (\forall\, s.\ P\ s \longrightarrow\ \ Q\ s)$
  by *auto*

2.1.5 Store and Assignment

type-synonym $'a\ store = string\ \Rightarrow\ 'a$

definition *gets* :: $string \Rightarrow ('a\ store \Rightarrow 'a) \Rightarrow 'a\ store\ rel$ (‹- ::= -› [*70*, *65*] *61*)
where
  $v ::= e = \{(s,\ s(v := e\ s))\ |s.\ True\}$

lemma *H-assign*: *rel-kat.H* $\lceil\lambda s.\ P\ (s\ (v := e\ s))\rceil\ (v ::= e)\ \lceil P\rceil$
  by (*auto simp*: *gets-def rel-kat.H-def rel-kat.t-op-def rel-at-def*)

lemma *H-assign-var*: $(\forall\, s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow$ *rel-kat.H* $\lceil P\rceil\ (v ::= e)$ $\lceil Q\rceil$
  by (*auto simp*: *gets-def rel-kat.H-def rel-kat.t-op-def rel-at-def*)

abbreviation *H-sugar* :: $'a\ pred \Rightarrow 'a\ rel \Rightarrow 'a\ pred \Rightarrow bool$ (‹*PRE* - - *POST* -› [*64*,*64*,*64*] *63*) **where**
  PRE P X POST Q ≡ *rel-kat.H* $\lceil P\rceil\ X\ \lceil Q\rceil$

abbreviation *if-then-else-sugar* :: $'a\ pred \Rightarrow 'a\ rel \Rightarrow 'a\ rel \Rightarrow 'a\ rel$ (‹*IF* - *THEN* - *ELSE* - *FI*› [*64*,*64*,*64*] *63*) **where**
  IF P THEN X ELSE Y FI ≡ *rel-kat.if-then-else* $\lceil P\rceil\ X\ Y$

abbreviation *while-inv-sugar* :: $'a\ pred \Rightarrow 'a\ pred \Rightarrow 'a\ rel \Rightarrow 'a\ rel$ (‹*WHILE* - *INV* - *DO* - *OD*› [*64*,*64*,*64*] *63*) **where**
  WHILE P INV I DO X OD ≡ *rel-kat.while-inv* $\lceil P\rceil\ \lceil I\rceil\ X$

2.1.6 Verification Example

lemma *euclid*:
  PRE $(\lambda s{::}nat\ store.\ s\ ''x'' = x \wedge s\ ''y'' = y)$
   $(WHILE\ (\lambda s.\ s\ ''y'' \neq 0)\ INV\ (\lambda s.\ gcd\ (s\ ''x'')\ (s\ ''y'') = gcd\ x\ y)$
    DO
     $(''z'' ::= (\lambda s.\ s\ ''y''));$
     $(''y'' ::= (\lambda s.\ s\ ''x''\ mod\ s\ ''y''));$
     $(''x'' ::= (\lambda s.\ s\ ''z''))$
    OD)
   POST $(\lambda s.\ s\ ''x'' = gcd\ x\ y)$

apply (*rule rel-kat.H-while-inv*, *simp-all*, *clarsimp*)
apply (*intro rel-kat.H-seq*)
apply (*subst H-assign*, *simp*)+
apply (*rule H-assign-var*)
using *gcd-red-nat* **by** *auto*

2.1.7 Definition of Refinement KAT

class *rkat* = *kat* +
fixes $R :: 'a \Rightarrow 'a \Rightarrow 'a$
assumes *R1*: $H\ p\ (R\ p\ q)\ q$
and *R2*: $H\ p\ x\ q \Longrightarrow x \leq R\ p\ q$

begin

2.1.8 Propositional Refinement Calculus

lemma *R-skip*: $1 \leq R\ p\ p$
by (*simp add*: *H-skip R2*)

lemma *R-cons*: $t\ p \leq t\ p' \Longrightarrow t\ q' \leq t\ q \Longrightarrow R\ p'\ q' \leq R\ p\ q$
by (*simp add*: *H-cons R2 R1*)

lemma *R-seq*: $(R\ p\ r) \cdot (R\ r\ q) \leq R\ p\ q$
using *H-seq R2 R1* **by** *blast*

lemma *R-cond*: *if v then* $(R\ (t\ v \cdot t\ p)\ q)$ *else* $(R\ (at\ v \cdot t\ p)\ q)$ *fi* $\leq R\ p\ q$
by (*metis H-cond R1 R2 t-comm t-n*)

lemma *R-loop*: *while q do* $(R\ (t\ p \cdot t\ q)\ p)$ *od* $\leq R\ p\ (t\ p \cdot at\ q)$
by (*simp add*: *H-loop R2 R1*)

end

2.1.9 Soundness and Relation RKAT

definition *rel-R* :: $'a\ rel \Rightarrow 'a\ rel \Rightarrow 'a\ rel$ **where**
rel-R $P\ Q = \bigcup\{X.\ rel\text{-}kat.H\ P\ X\ Q\}$

interpretation *rel-rkat*: *rkat Id* {} (∪) (;) (⊆) (⊂) *rtrancl rel-at rel-R*
by (*standard*, *auto simp*: *rel-R-def rel-kat.H-def rel-kat.t-op-def rel-at-def*)

2.1.10 Assignment Laws

lemma *R-assign*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow (v ::= e) \subseteq rel\text{-}R\ \lceil P \rceil\ \lceil Q \rceil$
by (*simp add*: *H-assign-var rel-rkat.R2*)

lemma *R-assignr*: $(\forall s.\ Q'\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow (rel\text{-}R\ \lceil P \rceil\ \lceil Q' \rceil)\ ;\ (v ::= e) \subseteq rel\text{-}R\ \lceil P \rceil\ \lceil Q \rceil$
proof −

assume *a1*: $\forall s.\ Q'\ s \longrightarrow Q\ (s(v := e\ s))$
have $\forall p\ pa\ cs\ f.\ \exists fa.\ (p\ fa \vee cs ::= f \subseteq rel\text{-}R\ \lceil p \rceil\ \lceil pa \rceil) \wedge (\neg\ pa\ (fa(cs := f\ fa::'a)) \vee cs ::= f \subseteq rel\text{-}R\ \lceil p \rceil\ \lceil pa \rceil)$
using *R-assign* **by** *blast*
hence $v ::= e \subseteq rel\text{-}R\ \lceil Q' \rceil\ \lceil Q \rceil$
using *a1* **by** *blast*
thus *?thesis*
by (*meson dual-order.trans rel-d.mult-isol rel-rkat.R-seq*)
qed

lemma *R-assignl*: $(\forall s.\ P\ s \longrightarrow P'\ (s\ (v := e\ s))) \Longrightarrow (v ::= e)\ ;\ (rel\text{-}R\ \lceil P' \rceil\ \lceil Q \rceil) \subseteq rel\text{-}R\ \lceil P \rceil\ \lceil Q \rceil$
proof −
assume *a1*: $\forall s.\ P\ s \longrightarrow P'\ (s(v := e\ s))$
have $\forall p\ pa\ cs\ f.\ \exists fa.\ (p\ fa \vee cs ::= f \subseteq rel\text{-}R\ \lceil p \rceil\ \lceil pa \rceil) \wedge (\neg\ pa\ (fa(cs := f\ fa::'a)) \vee cs ::= f \subseteq rel\text{-}R\ \lceil p \rceil\ \lceil pa \rceil)$
using *R-assign* **by** *blast*
then have $v ::= e \subseteq rel\text{-}R\ \lceil P \rceil\ \lceil P' \rceil$
using *a1* **by** *blast*
then show *?thesis*
by (*meson dual-order.trans rel-d.mult-isor rel-rkat.R-seq*)
qed

2.1.11 Refinement Example

lemma *var-swap-ref1*:
$rel\text{-}R\ \lceil \lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
$\supseteq (''z'' ::= (\lambda s.\ s\ ''x''));\ rel\text{-}R\ \lceil \lambda s.\ s\ ''z'' = a \wedge s\ ''y'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref2*:
$rel\text{-}R\ \lceil \lambda s.\ s\ ''z'' = a \wedge s\ ''y'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
$\supseteq (''x'' ::= (\lambda s.\ s\ ''y''));\ rel\text{-}R\ \lceil \lambda s.\ s\ ''z'' = a \wedge s\ ''x'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref3*:
$rel\text{-}R\ \lceil \lambda s.\ s\ ''z'' = a \wedge s\ ''x'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
$\supseteq (''y'' ::= (\lambda s.\ s\ ''z''));\ rel\text{-}R\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref-var*:
$rel\text{-}R\ \lceil \lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
$\supseteq (''z'' ::= (\lambda s.\ s\ ''x''));\ (''x'' ::= (\lambda s.\ s\ ''y''));\ (''y'' ::= (\lambda s.\ s\ ''z''))$
using *var-swap-ref1 var-swap-ref2 var-swap-ref3 rel-rkat.R-skip* **by** *fastforce*

end

2.2 Component Based on Kleene Algebra with Domain

This component supports the verification and step-wise refinement of simple while programs in a partial correctness setting.

theory *VC-KAD-scratch*
  imports *Main*
begin

2.2.1 KAD: Definitions and Basic Properties

notation *times* (**infixl** ‹·› *70*)

class *plus-ord* = *plus* + *ord* +
  assumes *less-eq-def*: $x \leq y \longleftrightarrow x + y = y$
  and *less-def*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$

class *dioid* = *semiring* + *one* + *zero* + *plus-ord* +
  assumes *add-idem* [*simp*]: $x + x = x$
  and *mult-onel* [*simp*]: $1 \cdot x = x$
  and *mult-oner* [*simp*]: $x \cdot 1 = x$
  and *add-zerol* [*simp*]: $0 + x = x$
  and *annil* [*simp*]: $0 \cdot x = 0$
  and *annir* [*simp*]: $x \cdot 0 = 0$

begin

subclass *monoid-mult*
  by (*standard*, *simp-all*)

subclass *order*
  by (*standard*, *simp-all add*: *less-def less-eq-def add-commute*, *auto*, *metis add-assoc*)

lemma *mult-isor*: $x \leq y \Longrightarrow x \cdot z \leq y \cdot z$
  by (*metis distrib-right less-eq-def*)

lemma *mult-isol*: $x \leq y \Longrightarrow z \cdot x \leq z \cdot y$
  by (*metis distrib-left less-eq-def*)

lemma *add-iso*: $x \leq y \Longrightarrow z + x \leq z + y$
  by (*metis add.semigroup-axioms add-idem less-eq-def semigroup.assoc*)

lemma *add-ub*: $x \leq x + y$
  by (*metis add-assoc add-idem less-eq-def*)

lemma *add-lub*: $x + y \leq z \longleftrightarrow x \leq z \wedge y \leq z$
  by (*metis add-assoc add-ub add.left-commute less-eq-def*)

end

class *kleene-algebra* = *dioid* +
  fixes *star* :: $'a \Rightarrow 'a$ (‹-*› [*101*] *100*)
  assumes *star-unfoldl*: $1 + x \cdot x^\star \leq x^\star$
  and *star-unfoldr*: $1 + x^\star \cdot x \leq x^\star$
  and *star-inductl*: $z + x \cdot y \leq y \Longrightarrow x^\star \cdot z \leq y$
  and *star-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot x^\star \leq y$

begin

lemma *star-sim*: $x \cdot y \leq z \cdot x \Longrightarrow x \cdot y^\star \leq z^\star \cdot x$
proof −
  assume $x \cdot y \leq z \cdot x$
  hence $x + z^\star \cdot x \cdot y \leq x + z^\star \cdot z \cdot x$
    by (*metis add-lub distrib-left eq-refl less-eq-def mult-assoc*)
  also have ... $\leq z^\star \cdot x$
    using *add-lub mult-isor star-unfoldr* **by** *fastforce*
  finally show *?thesis*
    by (*simp add*: *star-inductr*)
qed

end

class *antidomain-kleene-algebra* = *kleene-algebra* +
  fixes *ad* :: $'a \Rightarrow 'a$ (‹*ad*›)
  assumes *as1* [*simp*]: $ad\ x \cdot x = 0$
  and *as2* [*simp*]: $ad\ (x \cdot y) + ad\ (x \cdot ad\ (ad\ y)) = ad\ (x \cdot ad\ (ad\ y))$
  and *as3* [*simp*]: $ad\ (ad\ x) + ad\ x = 1$

begin

definition *dom-op* :: $'a \Rightarrow 'a$ (‹*d*›) **where**
  $d\ x = ad\ (ad\ x)$

lemma *a-subid-aux*: $ad\ x \cdot y \leq y$
  by (*metis add-commute add-ub as3 mult-1-left mult-isor*)

lemma *d1-a* [*simp*]: $d\ x \cdot x = x$
  by (*metis add-commute dom-op-def add-zerol as1 as3 distrib-right mult-1-left*)

lemma *a-mul-d* [*simp*]: $ad\ x \cdot d\ x = 0$
  by (*metis add-commute dom-op-def add-zerol as1 as2 distrib-right mult-1-left*)

lemma *a-d-closed* [*simp*]: $d\ (ad\ x) = ad\ x$
  by (*metis d1-a dom-op-def add-zerol as1 as3 distrib-left mult-1-right*)

lemma *a-idem* [*simp*]: $ad\ x \cdot ad\ x = ad\ x$
  by (*metis a-d-closed d1-a*)

lemma *meet-ord*: $ad\ x \leq ad\ y \longleftrightarrow ad\ x \cdot ad\ y = ad\ x$

by (*metis a-d-closed a-subid-aux d1-a order.antisym mult-1-right mult-isol*)

lemma *d-wloc*: $x \cdot y = 0 \longleftrightarrow x \cdot d\ y = 0$
by (*metis a-subid-aux d1-a dom-op-def add-ub order.antisym as1 as2 mult-1-right mult-assoc*)

lemma *gla-1*: $ad\ x \cdot y = 0 \Longrightarrow ad\ x \leq ad\ y$
by (*metis a-subid-aux d-wloc dom-op-def add-zerol as3 distrib-left mult-1-right*)

lemma *a2-eq* [*simp*]: $ad\ (x \cdot d\ y) = ad\ (x \cdot y)$
by (*metis a-mul-d d1-a dom-op-def gla-1 add-ub order.antisym as1 as2 mult-assoc*)

lemma *a-supdist*: $ad\ (x + y) \leq ad\ x$
by (*metis add-commute gla-1 add-ub add-zerol as1 distrib-left less-eq-def*)

lemma *a-antitone*: $x \leq y \Longrightarrow ad\ y \leq ad\ x$
by (*metis a-supdist less-eq-def*)

lemma *a-comm*: $ad\ x \cdot ad\ y = ad\ y \cdot ad\ x$
proof −
{ fix *x y*
have $ad\ x \cdot ad\ y = d\ (ad\ x \cdot ad\ y) \cdot ad\ x \cdot ad\ y$
by (*simp add*: *mult-assoc*)
also have ... $\leq d\ (ad\ y) \cdot ad\ x$
by (*metis a-antitone a-d-closed a-subid-aux mult-oner a-subid-aux dom-op-def mult-isol mult-isor meet-ord*)
finally have $ad\ x \cdot ad\ y \leq ad\ y \cdot ad\ x$
by *simp* **}**
thus *?thesis*
by (*simp add*: *order.antisym*)
qed

lemma *a-closed* [*simp*]: $d\ (ad\ x \cdot ad\ y) = ad\ x \cdot ad\ y$
proof −
have *f1*: $\bigwedge x\ y.\ ad\ x \leq ad\ (ad\ y \cdot x)$
by (*simp add*: *a-antitone a-subid-aux*)
have $\bigwedge x\ y.\ d\ (ad\ x \cdot y) \leq ad\ x$
by (*metis a2-eq a-antitone a-comm a-d-closed dom-op-def f1*)
hence $\bigwedge x\ y.\ d\ (ad\ x \cdot y) \cdot y = ad\ x \cdot y$
by (*metis d1-a dom-op-def meet-ord mult-assoc*)
thus *?thesis*
by (*metis a-comm a-idem dom-op-def*)
qed

lemma *a-exp* [*simp*]: $ad\ (ad\ x \cdot y) = d\ x + ad\ y$
proof (*rule order.antisym*)
have $ad\ (ad\ x \cdot y) \cdot ad\ x \cdot d\ y = 0$
using *d-wloc mult-assoc* **by** *fastforce*
hence *a*: $ad\ (ad\ x \cdot y) \cdot d\ y \leq d\ x$

by (*metis a-closed a-comm dom-op-def gla-1 mult-assoc*)
have $ad\ (ad\ x \cdot y) = ad\ (ad\ x \cdot y) \cdot d\ y + ad\ (ad\ x \cdot y) \cdot ad\ y$
by (*metis dom-op-def as3 distrib-left mult-oner*)
also have ... $\leq d\ x + ad\ (ad\ x \cdot y) \cdot ad\ y$
using *a add-lub dual-order.trans* **by** *blast*
finally show $ad\ (ad\ x \cdot y) \leq d\ x + ad\ y$
by (*metis a-antitone a-comm a-subid-aux meet-ord*)
next
have $ad\ y \leq ad\ (ad\ x \cdot y)$
by (*simp add: a-antitone a-subid-aux*)
thus $d\ x + ad\ y \leq ad\ (ad\ x \cdot y)$
by (*metis a2-eq a-antitone a-comm a-subid-aux dom-op-def add-lub*)
qed

lemma *d1-sum-var*: $x + y \leq (d\ x + d\ y) \cdot (x + y)$
proof −
have $x + y = d\ x \cdot x + d\ y \cdot y$
by *simp*
also have ... $\leq (d\ x + d\ y) \cdot x + (d\ x + d\ y) \cdot y$
by (*metis add-commute add-lub add-ub combine-common-factor*)
finally show *?thesis*
by (*simp add: distrib-left*)
qed

lemma *a4*: $ad\ (x + y) = ad\ x \cdot ad\ y$
proof (*rule order.antisym*)
show $ad\ (x + y) \leq ad\ x \cdot ad\ y$
by (*metis a-supdist add-commute mult-isor meet-ord*)
hence $ad\ x \cdot ad\ y = ad\ x \cdot ad\ y + ad\ (x + y)$
using *less-eq-def add-commute* **by** *simp*
also have ... $= ad\ (ad\ (ad\ x \cdot ad\ y) \cdot (x + y))$
by (*metis a-closed a-exp*)
finally show $ad\ x \cdot ad\ y \leq ad\ (x + y)$
using *a-antitone d1-sum-var dom-op-def* **by** *auto*
qed

lemma *kat-prop*: $d\ x \cdot y \leq y \cdot d\ z \longleftrightarrow d\ x \cdot y \cdot ad\ z = 0$
proof
show $d\ x \cdot y \leq y \cdot d\ z \Longrightarrow d\ x \cdot y \cdot ad\ z = 0$
by (*metis add-commute dom-op-def add-zerol annir as1 less-eq-def mult-isor mult-assoc*)
next
assume *h*: $d\ x \cdot y \cdot ad\ z = 0$
hence $d\ x \cdot y = d\ x \cdot y \cdot d\ z + d\ x \cdot y \cdot ad\ z$
by (*metis dom-op-def as3 distrib-left mult-1-right*)
thus $d\ x \cdot y \leq y \cdot d\ z$
by (*metis a-subid-aux add-commute dom-op-def h add-zerol mult-assoc*)
qed

lemma *shunt*: $ad\ x \leq ad\ y + ad\ z \longleftrightarrow ad\ x \cdot d\ y \leq ad\ z$
proof
  assume $ad\ x \leq ad\ y + ad\ z$
  hence $ad\ x \cdot d\ y \leq ad\ y \cdot d\ y + ad\ z \cdot d\ y$
    by (*metis distrib-right mult-isor*)
  thus $ad\ x \cdot d\ y \leq ad\ z$
   by (*metis a-closed a-d-closed a-exp a-mul-d a-supdist dom-op-def dual-order.trans less-eq-def*)
next
  assume *h*: $ad\ x \cdot d\ y \leq ad\ z$
  have $ad\ x = ad\ x \cdot ad\ y + ad\ x \cdot d\ y$
    by (*metis add-commute dom-op-def as3 distrib-left mult-1-right*)
  also have ... $\leq ad\ x \cdot ad\ y + ad\ z$
    using *h add-lub dual-order.trans* **by** *blast*
  also have ... $\leq ad\ y + ad\ z$
    by (*metis a-subid-aux add-commute add-lub add-ub dual-order.trans*)
  finally show $ad\ x \leq ad\ y + ad\ z$
    by *simp*
qed

2.2.2 wp Calculus

definition *if-then-else* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹*if - then - else - fi*› [*64*,*64*,*64*] *63*)
where
  if p then x else y fi $= d\ p \cdot x + ad\ p \cdot y$

definition *while* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹*while - do - od*› [*64*,*64*] *63*) **where**
  while p do x od $= (d\ p \cdot x)^{\star} \cdot ad\ p$

definition *while-inv* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹*while - inv - do - od*› [*64*,*64*,*64*] *63*)
where
  while p inv i do x od = *while p do x od*

definition *wp* :: $'a \Rightarrow 'a \Rightarrow 'a$ **where**
  $wp\ x\ p = ad\ (x \cdot ad\ p)$

lemma *demod*: $d\ p \leq wp\ x\ q \longleftrightarrow d\ p \cdot x \leq x \cdot d\ q$
  by (*metis as1 dom-op-def gla-1 kat-prop meet-ord mult-assoc wp-def*)

lemma *wp-weaken*: $wp\ x\ p \leq wp\ (x \cdot ad\ q)\ (d\ p \cdot ad\ q)$
  by (*metis a4 a-antitone a-d-closed a-mul-d dom-op-def gla-1 mult-isol mult-assoc wp-def*)

lemma *wp-seq* [*simp*]: $wp\ (x \cdot y)\ q = wp\ x\ (wp\ y\ q)$
  using *a2-eq dom-op-def mult-assoc wp-def* **by** *auto*

lemma *wp-seq-var*: $p \leq wp\ x\ r \Longrightarrow r \leq wp\ y\ q \Longrightarrow p \leq wp\ (x \cdot y)\ q$
proof −
  assume *a1*: $p \leq wp\ x\ r$

assume *a2*: $r \leq wp\ y\ q$
have $\forall z.\ \neg\ wp\ x\ r \leq z \vee p \leq z$
using *a1 dual-order.trans* **by** *blast*
then show *?thesis*
using *a2 a-antitone mult-isol wp-def wp-seq* **by** *auto*
qed

lemma *wp-cond-var* [*simp*]: $wp\ (if\ p\ then\ x\ else\ y\ fi)\ q = (ad\ p + wp\ x\ q) \cdot (d\ p + wp\ y\ q)$
using *a4 a-d-closed dom-op-def if-then-else-def distrib-right mult-assoc wp-def* **by** *auto*

lemma *wp-cond-aux1* [*simp*]: $d\ p \cdot wp\ (if\ p\ then\ x\ else\ y\ fi)\ q = d\ p \cdot wp\ x\ q$
proof –
have $d\ p \cdot wp\ (if\ p\ then\ x\ else\ y\ fi)\ q = ad\ (ad\ p) \cdot (ad\ p + wp\ x\ q) \cdot (d\ p + wp\ y\ q)$
using *dom-op-def mult.semigroup-axioms semigroup.assoc wp-cond-var* **by** *fastforce*
also have ... $= wp\ x\ q \cdot d\ p \cdot (d\ p + d\ (wp\ y\ q))$
using *a-comm a-d-closed dom-op-def distrib-left wp-def* **by** *auto*
also have ... $= wp\ x\ q \cdot d\ p$
by (*metis a-exp dom-op-def add-ub meet-ord mult-assoc*)
finally show *?thesis*
by (*simp add*: *a-comm dom-op-def wp-def*)
qed

lemma *wp-cond-aux2* [*simp*]: $ad\ p \cdot wp\ (if\ p\ then\ x\ else\ y\ fi)\ q = ad\ p \cdot wp\ y\ q$
by (*metis* (*no-types*) *abel-semigroup.commute if-then-else-def a-d-closed add.abel-semigroup-axioms dom-op-def wp-cond-aux1*)

lemma *wp-cond* [*simp*]: $wp\ (if\ p\ then\ x\ else\ y\ fi)\ q = (d\ p \cdot wp\ x\ q) + (ad\ p \cdot wp\ y\ q)$
by (*metis as3 distrib-right dom-op-def mult-1-left wp-cond-aux2 wp-cond-aux1*)

lemma *wp-star-induct-var*: $d\ q \leq wp\ x\ q \Longrightarrow d\ q \leq wp\ (x^\star)\ q$
using *demod star-sim* **by** *blast*

lemma *wp-while*: $d\ p \cdot d\ r \leq wp\ x\ p \Longrightarrow d\ p \leq wp\ (while\ r\ do\ x\ od)\ (d\ p \cdot ad\ r)$
proof –
assume $d\ p \cdot d\ r \leq wp\ x\ p$
hence $d\ p \leq wp\ (d\ r \cdot x)\ p$
using *dom-op-def mult.semigroup-axioms semigroup.assoc shunt wp-def* **by** *fastforce*
hence $d\ p \leq wp\ ((d\ r \cdot x)^\star)\ p$
using *wp-star-induct-var* **by** *blast*
thus *?thesis*
by (*simp add*: *while-def*) (*use local.dual-order.trans wp-weaken* **in** *fastforce*)
qed

lemma *wp-while-inv*: $d\ p \leq d\ i \Longrightarrow d\ i \cdot ad\ r \leq d\ q \Longrightarrow d\ i \cdot d\ r \leq wp\ x\ i \Longrightarrow d\ p \leq wp\ (while\ r\ inv\ i\ do\ x\ od)\ q$
proof −
  assume *a1*: $d\ p \leq d\ i$ **and** *a2*: $d\ i \cdot ad\ r \leq d\ q$ **and** $d\ i \cdot d\ r \leq wp\ x\ i$
  hence $d\ i \leq wp\ (while\ r\ inv\ i\ do\ x\ od)\ (d\ i \cdot ad\ r)$
    by (*simp add*: *while-inv-def wp-while*)
  also have ... $\leq$ $wp\ (while\ r\ inv\ i\ do\ x\ od)\ q$
    by (*metis a2 a-antitone a-d-closed dom-op-def mult-isol wp-def*)
  finally show *?thesis*
    using *a1 dual-order.trans* **by** *blast*
qed

lemma *wp-while-inv-break*: $d\ p \leq wp\ y\ i \Longrightarrow d\ i \cdot ad\ r \leq d\ q \Longrightarrow d\ i \cdot d\ r \leq wp\ x\ i \Longrightarrow d\ p \leq wp\ (y \cdot (while\ r\ inv\ i\ do\ x\ od))\ q$
  by (*metis dom-op-def eq-refl mult-1-left mult-1-right wp-def wp-seq wp-seq-var wp-while-inv*)

end

2.2.3 Soundness and Relation KAD

notation *relcomp* (**infixl** ‹;› *70*)

interpretation *rel-d*: *dioid Id* {} (∪) (;) (⊆) (⊂)
  by (*standard*, *auto*)

lemma (**in** *dioid*) *pow-inductl*: $z + x \cdot y \leq y \Longrightarrow x\ \hat{}\ i \cdot z \leq y$
  apply (*induct i*; *clarsimp simp add*: *add-lub*)
  by (*metis local.dual-order.trans local.mult-isol mult-assoc*)

lemma (**in** *dioid*) *pow-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot x\ \hat{}\ i \leq y$
  apply (*induct i*; *clarsimp simp add*: *add-lub*)
  proof −
    fix *i*
    assume $z \cdot x\ \hat{}\ i \leq y\ z \leq y\ y \cdot x \leq y$
    hence $(z \cdot x\ \hat{}\ i) \cdot x \leq y$
      using *local.dual-order.trans local.mult-isor* **by** *blast*
    thus $z \cdot (x \cdot x\ \hat{}\ i) \leq y$
      by (*simp add*: *mult-assoc local.power-commutes*)
  qed

lemma *power-is-relpow*: $rel\text{-}d.power\ X\ i = X\ \hat{}\hat{}\ i$
  by (*induct i*, *simp-all add*: *relpow-commute*)

lemma *rel-star-def*: $X\hat{}{*} = (\bigcup i.\ rel\text{-}d.power\ X\ i)$
  by (*simp add*: *power-is-relpow rtrancl-is-UN-relpow*)

lemma *rel-star-contl*: $X\ ;\ Y\hat{}{*} = (\bigcup i.\ X\ ;\ rel\text{-}d.power\ Y\ i)$
  by (*simp add*: *rel-star-def relcomp-UNION-distrib*)

lemma *rel-star-contr*: $X\widehat{\ }* \ ; \ Y = (\bigcup i. \ (rel\text{-}d.power\ X\ i) \ ; \ Y)$
by (*simp add*: *rel-star-def relcomp-UNION-distrib2*)

definition *rel-ad* :: *'a rel* $\Rightarrow$ *'a rel* **where**
rel-ad $R = \{(x,x) \mid x. \ \neg\ (\exists\, y. \ (x,y) \in R)\}$

interpretation *rel-aka*: *antidomain-kleene-algebra Id* {} ($\cup$) (;) ($\subseteq$) ($\subset$) *rtrancl rel-ad*
apply *standard*
apply *auto*[*2*]
by (*auto simp*: *rel-star-contr rel-d.pow-inductl rel-star-contl SUP-least rel-d.pow-inductr rel-ad-def*)

2.2.4 Embedding Predicates in Relations

type-synonym *'a pred* = *'a* $\Rightarrow$ *bool*

abbreviation *p2r* :: *'a pred* $\Rightarrow$ *'a rel* (‹⌈-⌉›) **where**
$\lceil P \rceil \equiv \{(s,s) \mid s. \ P\ s\}$

lemma *d-p2r* [*simp*]: *rel-aka.dom-op* $\lceil P \rceil = \lceil P \rceil$
by (*auto simp*: *rel-aka.dom-op-def rel-ad-def*)

lemma *p2r-neg-hom* [*simp*]: *rel-ad* $\lceil P \rceil = \lceil \lambda s. \ \neg P\ s \rceil$
by (*auto simp*: *rel-ad-def*)

lemma *p2r-conj-hom* [*simp*]: $\lceil P \rceil \cap \lceil Q \rceil = \lceil \lambda s. \ P\ s \wedge Q\ s \rceil$
by *auto*

lemma *p2r-conj-hom-var* [*simp*]: $\lceil P \rceil \ ; \ \lceil Q \rceil = \lceil \lambda s. \ P\ s \wedge Q\ s \rceil$
by *auto*

lemma *p2r-disj-hom* [*simp*]: $\lceil P \rceil \cup \lceil Q \rceil = \lceil \lambda s. \ P\ s \vee Q\ s \rceil$
by *auto*

2.2.5 Store and Assignment

type-synonym *'a store* = *string* $\Rightarrow$ *'a*

definition *gets* :: *string* $\Rightarrow$ (*'a store* $\Rightarrow$ *'a*) $\Rightarrow$ *'a store rel* (‹- ::= -› [*70*, *65*] *61*)
where
$v ::= e = \{(s,s\ (v := e\ s)) \mid s. \ True\}$

lemma *wp-assign* [*simp*]: *rel-aka.wp* $(v ::= e)\ \lceil Q \rceil = \lceil \lambda s. \ Q\ (s\ (v := e\ s)) \rceil$
by (*auto simp*: *rel-aka.wp-def gets-def rel-ad-def*)

abbreviation *spec-sugar* :: *'a pred* $\Rightarrow$ *'a rel* $\Rightarrow$ *'a pred* $\Rightarrow$ *bool* (‹*PRE* - - *POST* -› [*64*,*64*,*64*] *63*) **where**
PRE P X POST Q $\equiv$ *rel-aka.dom-op* $\lceil P \rceil \subseteq$ *rel-aka.wp X* $\lceil Q \rceil$

abbreviation *if-then-else-sugar* :: $'a$ *pred* ⇒ $'a$ *rel* ⇒ $'a$ *rel* ⇒ $'a$ *rel* (‹*IF - THEN - ELSE - FI*› [*64*,*64*,*64*] *63*) **where**
IF P THEN X ELSE Y FI ≡ *rel-aka.if-then-else* ⌈*P*⌉ *X Y*

abbreviation *while-inv-sugar* :: $'a$ *pred* ⇒ $'a$ *pred* ⇒ $'a$ *rel* ⇒ $'a$ *rel* (‹*WHILE - INV - DO - OD*› [*64*,*64*,*64*] *63*) **where**
WHILE P INV I DO X OD ≡ *rel-aka.while-inv* ⌈*P*⌉ ⌈*I*⌉ *X*

2.2.6 Verification Example

lemma *euclid*:
PRE (λ*s*::*nat store*. *s* ''*x*'' = *x* ∧ *s* ''*y*'' = *y*)
(*WHILE* (λ*s*. *s* ''*y*'' ≠ *0*) *INV* (λ*s*. *gcd* (*s* ''*x*'') (*s* ''*y*'') = *gcd x y*)
DO
(''*z*'' ::= (λ*s*. *s* ''*y*''));
(''*y*'' ::= (λ*s*. *s* ''*x*'' *mod s* ''*y*''));
(''*x*'' ::= (λ*s*. *s* ''*z*''))
OD)
POST (λ*s*. *s* ''*x*'' = *gcd x y*)
apply (*rule rel-aka.wp-while-inv*, *simp-all*) **using** *gcd-red-nat* **by** *auto*

context *antidomain-kleene-algebra*
begin

2.2.7 Propositional Hoare Logic

definition *H* :: $'a$ ⇒ $'a$ ⇒ $'a$ ⇒ *bool* **where**
H p x q ⟷ *d p* ≤ *wp x q*

lemma *H-skip*: *H p 1 p*
by (*simp add*: *H-def dom-op-def wp-def*)

lemma *H-cons*: *d p* ≤ *d p*′ ⟹ *d q*′ ≤ *d q* ⟹ *H p*′ *x q*′ ⟹ *H p x q*
by (*meson H-def demod mult-isol order-trans*)

lemma *H-seq*: *H p x r* ⟹ *H r y q* ⟹ *H p* (*x* · *y*) *q*
by (*metis H-def a-d-closed demod dom-op-def wp-seq-var*)

lemma *H-cond*: *H* (*d p* · *d r*) *x q* ⟹ *H* (*d p* · *ad r*) *y q* ⟹ *H p* (*if r then x else y fi*) *q*
proof −
assume *h1*: *H* (*d p* · *d r*) *x q* **and** *h2*: *H* (*d p* · *ad r*) *y q*
hence *h3*: *d p* · *d r* ≤ *wp x q* **and** *h4*: *d p* · *ad r* ≤ *wp y q*
using *H-def a-closed dom-op-def* **apply** *auto*[*1*]
using *H-def h2 a-closed dom-op-def* **by** *auto*
hence *h5*: *d p* ≤ *ad r* + *wp x q* **and** *h6*: *d p* ≤ *d r* + *wp y q*
apply (*simp add*: *dom-op-def shunt wp-def*)
using *h4 a-d-closed dom-op-def shunt wp-def* **by** *auto*
hence *d p* ≤ *d p* · (*d r* + *wp y q*)

by (*metis a-idem distrib-left dom-op-def less-eq-def*)
also have ... $\leq$ (*ad r* + *wp x q*) $\cdot$ (*d r* + *wp y q*)
by (*simp add*: *h5 mult-isor*)
finally show *?thesis*
by (*simp add*: *H-def*)
qed

lemma *H-loop*: *H* (*d p* $\cdot$ *d r*) *x p* $\Longrightarrow$ *H p* (*while r do x od*) (*d p* $\cdot$ *ad r*)
by (*metis* (*full-types*) *H-def a-closed dom-op-def wp-while*)

lemma *H-while-inv*: *d p* $\leq$ *d i* $\Longrightarrow$ *d i* $\cdot$ *ad r* $\leq$ *d q* $\Longrightarrow$ *H* (*d i* $\cdot$ *d r*) *x i* $\Longrightarrow$ *H p* (*while r inv i do x od*) *q*
using *H-def a-closed dom-op-def wp-while-inv* **by** *auto*

end

2.2.8 Definition of Refinement KAD

class *rkad* = *antidomain-kleene-algebra* +
fixes *R* :: $'a \Rightarrow 'a \Rightarrow 'a$
assumes *R-def*: *x* $\leq$ *R p q* $\longleftrightarrow$ *d p* $\leq$ *wp x q*

begin

2.2.9 Propositional Refinement Calculus

lemma *HR*: *H p x q* $\longleftrightarrow$ *x* $\leq$ *R p q*
by (*simp add*: *H-def R-def*)

lemma *wp-R1*: *d p* $\leq$ *wp* (*R p q*) *q*
using *R-def* **by** *blast*

lemma *wp-R2*: *x* $\leq$ *R* (*wp x q*) *q*
by (*simp add*: *R-def wp-def*)

lemma *wp-R3*: *d p* $\leq$ *wp x q* $\Longrightarrow$ *x* $\leq$ *R p q*
by (*simp add*: *R-def*)

lemma *H-R1*: *H p* (*R p q*) *q*
by (*simp add*: *HR*)

lemma *H-R2*: *H p x q* $\Longrightarrow$ *x* $\leq$ *R p q*
by (*simp add*: *HR*)

lemma *R-skip*: *1* $\leq$ *R p p*
by (*simp add*: *H-R2 H-skip*)

lemma *R-cons*: *d p* $\leq$ *d p'* $\Longrightarrow$ *d q'* $\leq$ *d q* $\Longrightarrow$ *R p' q'* $\leq$ *R p q*
by (*simp add*: *H-R1 H-R2 H-cons*)

lemma *R-seq*: $(R\ p\ r) \cdot (R\ r\ q) \leq R\ p\ q$
using *H-R1 H-R2 H-seq* **by** *blast*

lemma *R-cond*: *if* v *then* $(R\ (d\ v \cdot d\ p)\ q)$ *else* $(R\ (ad\ v \cdot d\ p)\ q)$ *fi* $\leq R\ p\ q$
by (*simp add*: *H-R1 H-R2 H-cond a-comm dom-op-def*)

lemma *R-loop*: *while* q *do* $(R\ (d\ p \cdot d\ q)\ p)$ *od* $\leq R\ p\ (d\ p \cdot ad\ q)$
by (*simp add*: *H-R1 H-R2 H-loop*)

end

2.2.10 Soundness and Relation RKAD

definition *rel-R* :: $'a\ rel \Rightarrow 'a\ rel \Rightarrow 'a\ rel$ **where**
rel-R $P\ Q = \bigcup\{X.\ rel\text{-}aka.dom\text{-}op\ P \subseteq rel\text{-}aka.wp\ X\ Q\}$

interpretation *rel-rkad*: *rkad Id* $\{\}$ $(\cup)$ $(;)$ $(\subseteq)$ $(\subset)$ *rtrancl rel-ad rel-R*
by (*standard*, *auto simp*: *rel-R-def rel-aka.dom-op-def rel-ad-def rel-aka.wp-def*, *blast*)

2.2.11 Assignment Laws

lemma *R-assign*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow (v ::= e) \subseteq rel\text{-}R\ \lceil P\rceil\ \lceil Q\rceil$
by (*auto simp*: *rel-rkad.R-def*)

lemma *H-assign-var*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow rel\text{-}aka.H\ \lceil P\rceil\ (v ::= e)\ \lceil Q\rceil$
by (*auto simp*: *rel-aka.H-def rel-aka.dom-op-def rel-ad-def gets-def rel-aka.wp-def*)

lemma *R-assignr*: $(\forall s.\ Q'\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow (rel\text{-}R\ \lceil P\rceil\ \lceil Q'\rceil)\ ;\ (v ::= e) \subseteq rel\text{-}R\ \lceil P\rceil\ \lceil Q\rceil$
apply (*subst rel-rkad.HR*[*symmetric*], *rule rel-aka.H-seq*) **defer**
by (*erule H-assign-var*, *simp add*: *rel-rkad.H-R1*)

lemma *R-assignl*: $(\forall s.\ P\ s \longrightarrow P'\ (s\ (v := e\ s))) \Longrightarrow (v ::= e)\ ;\ (rel\text{-}R\ \lceil P'\rceil\ \lceil Q\rceil) \subseteq rel\text{-}R\ \lceil P\rceil\ \lceil Q\rceil$
by (*subst rel-rkad.HR*[*symmetric*], *rule rel-aka.H-seq*, *erule H-assign-var*, *simp add*: *rel-rkad.H-R1*)

2.2.12 Refinement Example

lemma *var-swap-ref1*:
rel-R $\lceil\lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
$\supseteq (''z'' ::= (\lambda s.\ s\ ''x''));\ rel\text{-}R\ \lceil\lambda s.\ s\ ''z'' = a \wedge s\ ''y'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref2*:
rel-R $\lceil\lambda s.\ s\ ''z'' = a \wedge s\ ''y'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$

$\supseteq$ (''x'' ::= ($\lambda s.$ s ''y'')); *rel-R* $\lceil \lambda s.$ s ''z'' = a $\wedge$ s ''x'' = b$\rceil$ $\lceil \lambda s.$ s ''x'' = b $\wedge$ s ''y'' = a$\rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref3*:
rel-R $\lceil \lambda s.$ s ''z'' = a $\wedge$ s ''x'' = b$\rceil$ $\lceil \lambda s.$ s ''x'' = b $\wedge$ s ''y'' = a$\rceil$
$\supseteq$ (''y'' ::= ($\lambda s.$ s ''z'')); *rel-R* $\lceil \lambda s.$ s ''x'' = b $\wedge$ s ''y'' = a$\rceil$ $\lceil \lambda s.$ s ''x'' = b $\wedge$ s ''y'' = a$\rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref-var*:
rel-R $\lceil \lambda s.$ s ''x'' = a $\wedge$ s ''y'' = b$\rceil$ $\lceil \lambda s.$ s ''x'' = b $\wedge$ s ''y'' = a$\rceil$
$\supseteq$ (''z'' ::= ($\lambda s.$ s ''x'')); (''x'' ::= ($\lambda s.$ s ''y'')); (''y'' ::= ($\lambda s.$ s ''z''))
using *var-swap-ref1 var-swap-ref2 var-swap-ref3 rel-rkad.R-skip* **by** *fastforce*

end

3 Isomorphisms Between Predicates, Sets and Relations

theory *P2S2R*
imports *Main*

begin

notation *relcomp* (**infixl** ‹;› *70*)
notation *inf* (**infixl** ‹$\sqcap$› *70*)
notation *sup* (**infixl** ‹$\sqcup$› *65*)
notation *Id-on* (‹*s2r*›)
notation *Domain* (‹*r2s*›)
notation *Collect* (‹*p2s*›)

definition *rel-n* :: *'a rel* $\Rightarrow$ *'a rel* **where**
rel-n $\equiv$ ($\lambda X.$ *Id* $\cap$ $-$ *X*)

lemma *subid-meet*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow R \cap S = R ; S$
by *blast*

3.1 Isomorphism Between Sets and Relations

lemma *srs*: *r2s* $\circ$ *s2r* = *id*
by *auto*

lemma *rsr*: $R \subseteq Id \Longrightarrow$ *s2r* (*r2s R*) = *R*
by (*auto simp*: *Id-def Id-on-def Domain-def*)

lemma *s2r-inj*: *inj s2r*
by (*metis Domain-Id-on injI*)

lemma *r2s-inj*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2s\ R = r2s\ S \Longrightarrow R = S$
  by (*metis rsr*)

lemma *s2r-surj*: $\forall R \subseteq Id.\ \exists A.\ R = s2r\ A$
  using *rsr* **by** *auto*

lemma *r2s-surj*: $\forall A.\ \exists R \subseteq Id.\ A = r2s\ R$
  by (*metis Domain-Id-on Id-onE pair-in-Id-conv subsetI*)

lemma *s2r-union-hom*: $s2r\ (A \cup B) = s2r\ A \cup s2r\ B$
  by (*simp add*: *Id-on-def*)

lemma *s2r-inter-hom*: $s2r\ (A \cap B) = s2r\ A \cap s2r\ B$
  by (*auto simp*: *Id-on-def*)

lemma *s2r-inter-hom-var*: $s2r\ (A \cap B) = s2r\ A\ ;\ s2r\ B$
  by (*auto simp*: *Id-on-def*)

lemma *s2r-compl-hom*: $s2r\ (-\ A) = rel\text{-}n\ (s2r\ A)$
  by (*auto simp add*: *rel-n-def*)

lemma *r2s-union-hom*: $r2s\ (R \cup S) = r2s\ R \cup r2s\ S$
  by *auto*

lemma *r2s-inter-hom*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2s\ (R \cap S) = r2s\ R \cap r2s\ S$
  by *auto*

lemma *r2s-inter-hom-var*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2s\ (R\ ;\ S) = r2s\ R \cap r2s\ S$
  by (*metis r2s-inter-hom subid-meet*)

lemma *r2s-ad-hom*: $R \subseteq Id \Longrightarrow r2s\ (rel\text{-}n\ R) = -\ r2s\ R$
  by (*metis r2s-surj rsr s2r-compl-hom*)

3.2 Isomorphism Between Predicates and Sets

type-synonym $'a\ pred = 'a \Rightarrow bool$

definition *s2p* :: $'a\ set \Rightarrow 'a\ pred$ **where**
  $s2p\ S = (\lambda x.\ x \in S)$

lemma *sps* [*simp*]: $s2p \circ p2s = id$
  by (*intro ext*, *simp add*: *s2p-def*)

lemma *psp* [*simp*]: $p2s \circ s2p = id$
  by (*intro ext*, *simp add*: *s2p-def*)

lemma *s2p-bij*: *bij s2p*
  using *o-bij psp sps* **by** *blast*

lemma *p2s-bij*: *bij p2s*
  using *o-bij psp sps* **by** *blast*

lemma *s2p-compl-hom*: $s2p\ (-\ A) = -\ (s2p\ A)$
  by (*metis Collect-mem-eq comp-eq-dest-lhs id-apply sps uminus-set-def*)

lemma *s2p-inter-hom*: $s2p\ (A \cap B) = (s2p\ A) \sqcap (s2p\ B)$
  by (*metis Collect-mem-eq comp-eq-dest-lhs id-apply inf-set-def sps*)

lemma *s2p-union-hom*: $s2p\ (A \cup B) = (s2p\ A) \sqcup (s2p\ B)$
  by (*auto simp*: *s2p-def*)

lemma *p2s-neg-hom*: $p2s\ (-\ P) = -\ (p2s\ P)$
  by *fastforce*

lemma *p2s-conj-hom*: $p2s\ (P \sqcap Q) = p2s\ P \cap p2s\ Q$
  by *blast*

lemma *p2s-disj-hom*: $p2s\ (P \sqcup Q) = p2s\ P \cup p2s\ Q$
  by *blast*

3.3 Isomorphism Between Predicates and Relations

definition *p2r* :: *'a pred* ⇒ *'a rel* **where**
  $p2r\ P = \{(s,s)\ |s.\ P\ s\}$

definition *r2p* :: *'a rel* ⇒ *'a pred* **where**
  $r2p\ R = (\lambda x.\ x \in Domain\ R)$

lemma *p2r-subid*: $p2r\ P \subseteq Id$
  by (*simp add*: *p2r-def subset-eq*)

lemma *p2s2r*: $p2r = s2r \circ p2s$
proof (*intro ext*)
  fix *P* :: *'a pred*
  have $\{(a,\ a)\ |a.\ P\ a\} = \{(b,\ a).\ b = a \wedge P\ b\}$
    by *blast*
  thus $p2r\ P = (s2r \circ p2s)\ P$
    by (*simp add*: *Id-on-def' p2r-def*)
qed

lemma *r2s2p*: $r2p = s2p \circ r2s$
  by (*intro ext*, *simp add*: *r2p-def s2p-def*)

lemma *prp* [*simp*]: $r2p \circ p2r = id$
  by (*intro ext*, *simp add*: *p2s2r r2p-def*)

lemma *rpr*: $R \subseteq Id \Longrightarrow p2r\ (r2p\ R) = R$

by (*metis comp-apply id-apply p2s2r psp r2s2p rsr*)

lemma *p2r-inj*: *inj p2r*
by (*metis comp-eq-dest-lhs id-apply injI prp*)

lemma *r2p-inj*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2p\ R = r2p\ S \Longrightarrow R = S$
by (*metis rpr*)

lemma *p2r-surj*: $\forall\ R \subseteq Id.\ \exists P.\ R = p2r\ P$
using *rpr* **by** *auto*

lemma *r2p-surj*: $\forall P.\ \exists R \subseteq Id.\ P = r2p\ R$
by (*metis comp-apply id-apply p2r-subid prp*)

lemma *p2r-neg-hom*: $p2r\ (-\ P) = rel\text{-}n\ (p2r\ P)$
by (*simp add*: *p2s2r p2s-neg-hom s2r-compl-hom*)

lemma *p2r-conj-hom* [*simp*]: $p2r\ P \cap p2r\ Q = p2r\ (P \sqcap Q)$
by (*simp add*: *p2s2r p2s-conj-hom s2r-inter-hom*)

lemma *p2r-conj-hom-var* [*simp*]: $p2r\ P\ ;\ p2r\ Q = p2r\ (P \sqcap Q)$
by (*simp add*: *p2s2r p2s-conj-hom s2r-inter-hom-var*)

lemma *p2r-id-neg* [*simp*]: $Id \cap -\ p2r\ p = p2r\ (-p)$
by (*auto simp*: *p2r-def*)

lemma [*simp*]: $p2r\ bot = \{\}$
by (*auto simp*: *p2r-def*)

lemma *p2r-disj-hom* [*simp*]: $p2r\ P \cup p2r\ Q = p2r\ (P \sqcup Q)$
by (*simp add*: *p2s2r p2s-disj-hom s2r-union-hom*)

lemma *r2p-ad-hom*: $R \subseteq Id \Longrightarrow r2p\ (rel\text{-}n\ R) = -\ (r2p\ R)$
by (*simp add*: *r2s2p r2s-ad-hom s2p-compl-hom*)

lemma *r2p-inter-hom*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2p\ (R \cap S) = (r2p\ R) \sqcap (r2p\ S)$
by (*simp add*: *r2s2p r2s-inter-hom s2p-inter-hom*)

lemma *r2p-inter-hom-var*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2p\ (R\ ;\ S) = (r2p\ R) \sqcap (r2p\ S)$
by (*simp add*: *r2s2p r2s-inter-hom-var s2p-inter-hom*)

lemma *rel-to-pred-union-hom*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2p\ (R \cup S) = (r2p\ R) \sqcup (r2p\ S)$
by (*simp add*: *Domain-Un-eq r2s2p s2p-union-hom*)

end

4 Components Based on Kleene Algebra with Tests

4.1 Verification Component

This component supports the verification of simple while programs in a partial correctness setting.

theory *VC-KAT*
imports ../*P2S2R*
KAT-and-DRA.PHL-KAT
KAT-and-DRA.KAT-Models

begin

This first part changes some of the facts from the AFP KAT theories. It should be added to KAT in the next AFP version. Currently these facts provide an interface between the KAT theories and the verification component.

no-notation *if-then-else* (‹*if - then - else - fi*› [*64*,*64*,*64*] *63*)
no-notation *while* (‹*while - do - od*› [*64*,*64*] *63*)
unbundle *no floor-ceiling-syntax*

notation *relcomp* (**infixl** ‹;› *70*)
notation *p2r* (‹⌈-⌉›)

context *kat*
begin

4.1.1 Definitions of Hoare Triple

definition $H :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$H\ p\ x\ q \longleftrightarrow t\ p \cdot x \leq x \cdot t\ q$

lemma *H-var1*: $H\ p\ x\ q \longleftrightarrow t\ p \cdot x \cdot n\ q = 0$
by (*metis H-def n-kat-3 t-n-closed*)

lemma *H-var2*: $H\ p\ x\ q \longleftrightarrow t\ p \cdot x = t\ p \cdot x \cdot t\ q$
by (*simp add*: *H-def n-kat-2*)

4.1.2 Syntax for Conditionals and Loops

definition *ifthenelse* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹*if - then - else - fi*› [*64*,*64*,*64*] *63*)
where
if p then x else y fi $= (t\ p \cdot x + n\ p \cdot y)$

definition *while* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹*while - do - od*› [*64*,*64*] *63*) **where**
while b do x od $= (t\ b \cdot x)^\star \cdot n\ b$

definition *while-inv* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹*while - inv - do - od*› [*64*,*64*,*64*] *63*)
where

while p inv i do x od = *while p do x od*

4.1.3 Propositional Hoare Logic

lemma *H-skip*: $H\ p\ 1\ p$
by (*simp add*: *H-def*)

lemma *H-cons-1*: $t\ p \leq t\ p' \Longrightarrow H\ p'\ x\ q \Longrightarrow H\ p\ x\ q$
using *H-def phl-cons1* **by** *blast*

lemma *H-cons-2*: $t\ q' \leq t\ q \Longrightarrow H\ p\ x\ q' \Longrightarrow H\ p\ x\ q$
using *H-def phl-cons2* **by** *blast*

lemma *H-cons*: $t\ p \leq t\ p' \Longrightarrow t\ q' \leq t\ q \Longrightarrow H\ p'\ x\ q' \Longrightarrow H\ p\ x\ q$
by (*simp add*: *H-cons-1 H-cons-2*)

lemma *H-seq-swap*: $H\ p\ x\ r \Longrightarrow H\ r\ y\ q \Longrightarrow H\ p\ (x \cdot y)\ q$
by (*simp add*: *H-def phl-seq*)

lemma *H-seq*: $H\ r\ y\ q \Longrightarrow H\ p\ x\ r \Longrightarrow H\ p\ (x \cdot y)\ q$
by (*simp add*: *H-seq-swap*)

lemma *H-exp1*: $H\ (t\ p \cdot t\ r)\ x\ q \Longrightarrow H\ p\ (t\ r \cdot x)\ q$
using *H-def n-de-morgan-var2 phl.ht-at-phl-export1* **by** *auto*

lemma *H-exp2*: $H\ p\ x\ q \Longrightarrow H\ p\ (x \cdot t\ r)\ (t\ q \cdot t\ r)$
by (*metis H-def phl.ht-at-phl-export2 test-mult*)

lemma *H-cond-iff*: $H\ p\ (\mathit{if}\ r\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi})\ q \longleftrightarrow H\ (t\ p \cdot t\ r)\ x\ q \wedge H\ (t\ p \cdot n\ r)\ y\ q$
proof −
have $H\ p\ (\mathit{if}\ r\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi})\ q \longleftrightarrow t\ p \cdot (t\ r \cdot x + n\ r \cdot y) \cdot n\ q = 0$
by (*simp add*: *H-var1 ifthenelse-def*)
also have ... $\longleftrightarrow t\ p \cdot t\ r \cdot x \cdot n\ q + t\ p \cdot n\ r \cdot y \cdot n\ q = 0$
by (*simp add*: *distrib-left mult-assoc*)
also have ... $\longleftrightarrow t\ p \cdot t\ r \cdot x \cdot n\ q = 0 \wedge t\ p \cdot n\ r \cdot y \cdot n\ q = 0$
by (*metis add-0-left no-trivial-inverse*)
finally show *?thesis*
by (*metis H-var1 test-mult*)
qed

lemma *H-cond*: $H\ (t\ p \cdot t\ r)\ x\ q \Longrightarrow H\ (t\ p \cdot n\ r)\ y\ q \Longrightarrow H\ p\ (\mathit{if}\ r\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi})\ q$
by (*simp add*: *H-cond-iff*)

lemma *H-loop*: $H\ (t\ p \cdot t\ r)\ x\ p \Longrightarrow H\ p\ (\mathit{while}\ r\ \mathit{do}\ x\ \mathit{od})\ (t\ p \cdot n\ r)$
proof −
assume *a1*: $H\ (t\ p \cdot t\ r)\ x\ p$
have $t\ (t\ p \cdot n\ r) = n\ r \cdot t\ p \cdot n\ r$

using *n-preserve test-mult* **by** *presburger*
then show *?thesis*
using *a1 H-def H-exp1 conway.phl.it-simr phl-export2 while-def* **by** *auto*
qed

lemma *H-while-inv*: $t\ p \le t\ i \Longrightarrow t\ i \cdot n\ r \le t\ q \Longrightarrow H\ (t\ i \cdot t\ r)\ x\ i \Longrightarrow H\ p$ $(while\ r\ inv\ i\ do\ x\ od)\ q$
by (*metis H-cons H-loop test-mult while-inv-def*)

Finally we prove a frame rule.

lemma *H-frame*: $H\ p\ x\ p \Longrightarrow H\ q\ x\ r \Longrightarrow H\ (t\ p \cdot t\ q)\ x\ (t\ p \cdot t\ r)$
proof −
assume $H\ p\ x\ p$ **and** a: $H\ q\ x\ r$
hence b: $t\ p \cdot x \le x \cdot t\ p$ **and** $t\ q \cdot x \le x \cdot t\ r$
using *H-def* **apply** *blast* **using** *H-def a* **by** *blast*
hence $t\ p \cdot t\ q \cdot x \le t\ p \cdot x \cdot t\ r$
by (*simp add*: *mult-assoc mult-isol*)
also have ... $\le x \cdot t\ p \cdot t\ r$
by (*simp add*: *b mult-isor*)
finally show *?thesis*
by (*metis H-def mult-assoc test-mult*)
qed

end

4.1.4 Store and Assignment

The proper verification component starts here.

type-synonym *'a store = string ⇒ 'a*

lemma *t-p2r* [*simp*]: *rel-dioid-tests.t* $\lceil P \rceil = \lceil P \rceil$
by (*auto simp*: *p2r-def*)

lemma *impl-prop* [*simp*]: $\lceil P \rceil \subseteq \lceil Q \rceil \longleftrightarrow (\forall s.\ P\ s \longrightarrow\ Q\ s)$
by (*auto simp*: *p2r-def*)

lemma *Id-simp* [*simp*]: $Id \cap (-\ Id \cup X) = Id \cap X$
by *auto*

lemma *Id-p2r* [*simp*]: $Id \cap \lceil P \rceil = \lceil P \rceil$
by (*auto simp*: *Id-def p2r-def*)

lemma *Id-p2r-simp* [*simp*]: $Id \cap (-\ Id \cup \lceil P \rceil) = \lceil P \rceil$
by *simp*

Next we derive the assignment command and assignment rules.

definition *gets* :: *string ⇒ ('a store ⇒ 'a) ⇒ 'a store rel* (‹- ::= -› [*70*, *65*] *61*)
where

$v ::= e = \{(s,s\ (v := e\ s))\ |s.\ True\}$

lemma *H-assign-prop*: $\lceil\lambda s.\ P\ (s\ (v := e\ s))\rceil\ ;\ (v ::= e) = (v ::= e)\ ;\ \lceil P\rceil$
by (*auto simp*: *p2r-def gets-def*)

lemma *H-assign*: *rel-kat.H* $\lceil\lambda s.\ P\ (s\ (v := e\ s))\rceil\ (v ::= e)\ \lceil P\rceil$
by (*auto simp add*: *rel-kat.H-def gets-def p2r-def*)

lemma *H-assign-var*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow$ *rel-kat.H* $\lceil P\rceil\ (v ::= e)\ \lceil Q\rceil$
by (*auto simp*: *p2r-def gets-def rel-kat.H-def*)

lemma *H-assign-iff* [*simp*]: *rel-kat.H* $\lceil P\rceil\ (v ::= e)\ \lceil Q\rceil \longleftrightarrow (\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s)))$
by (*auto simp*: *p2r-def gets-def rel-kat.H-def*)

lemma *H-assign-floyd*: *rel-kat.H* $\lceil P\rceil\ (v ::= e)\ \lceil\lambda s.\ \exists w.\ s\ v = e\ (s(v := w)) \wedge P\ (s(v := w))\rceil$
by (*rule H-assign-var*, *metis fun-upd-same fun-upd-triv fun-upd-upd*)

4.1.5 Simplified Hoare Rules

lemma *sH-cons-1*: $\forall s.\ P\ s \longrightarrow P'\ s \Longrightarrow$ *rel-kat.H* $\lceil P'\rceil\ X\ \lceil Q\rceil \Longrightarrow$ *rel-kat.H* $\lceil P\rceil\ X\ \lceil Q\rceil$
by (*rule rel-kat.H-cons-1*, *auto simp only*: *p2r-def*)

lemma *sH-cons-2*: $\forall s.\ Q'\ s \longrightarrow Q\ s \Longrightarrow$ *rel-kat.H* $\lceil P\rceil\ X\ \lceil Q'\rceil \Longrightarrow$ *rel-kat.H* $\lceil P\rceil\ X\ \lceil Q\rceil$
by (*rule rel-kat.H-cons-2*, *auto simp only*: *p2r-def*)

lemma *sH-cons*: $\forall s.\ P\ s \longrightarrow P'\ s \Longrightarrow \forall s.\ Q'\ s \longrightarrow Q\ s \Longrightarrow$ *rel-kat.H* $\lceil P'\rceil\ X\ \lceil Q'\rceil \Longrightarrow$ *rel-kat.H* $\lceil P\rceil\ X\ \lceil Q\rceil$
by (*simp add*: *sH-cons-1 sH-cons-2*)

lemma *sH-cond*: *rel-kat.H* $\lceil P \sqcap T\rceil\ X\ \lceil Q\rceil \Longrightarrow$ *rel-kat.H* $\lceil P \sqcap -\ T\rceil\ Y\ \lceil Q\rceil \Longrightarrow$ *rel-kat.H* $\lceil P\rceil$ (*rel-kat.ifthenelse* $\lceil T\rceil\ X\ Y)\ \lceil Q\rceil$
by (*rule rel-kat.H-cond*, *auto simp add*: *rel-kat.H-def p2r-def*, *blast+*)

lemma *sH-cond-iff*: *rel-kat.H* $\lceil P\rceil$ (*rel-kat.ifthenelse* $\lceil T\rceil\ X\ Y)\ \lceil Q\rceil \longleftrightarrow$ (*rel-kat.H* $\lceil P \sqcap T\rceil\ X\ \lceil Q\rceil \wedge$ *rel-kat.H* $\lceil P \sqcap -\ T\rceil\ Y\ \lceil Q\rceil$)
by (*simp add*: *rel-kat.H-cond-iff*)

lemma *sH-while-inv*: $\forall s.\ P\ s \longrightarrow I\ s \Longrightarrow \forall s.\ I\ s \wedge \neg\ R\ s \longrightarrow Q\ s \Longrightarrow$ *rel-kat.H* $\lceil I \sqcap R\rceil\ X\ \lceil I\rceil$
$\Longrightarrow$ *rel-kat.H* $\lceil P\rceil$ (*rel-kat.while-inv* $\lceil R\rceil\ \lceil I\rceil\ X)\ \lceil Q\rceil$
by (*rule rel-kat.H-while-inv*, *auto simp*: *p2r-def rel-kat.H-def*, *fastforce*)

lemma *sH-H*: *rel-kat.H* $\lceil P\rceil\ X\ \lceil Q\rceil \longleftrightarrow (\forall s\ s'.\ P\ s \longrightarrow (s,s') \in X \longrightarrow Q\ s')$
by (*simp add*: *rel-kat.H-def*, *auto simp add*: *p2r-def*)

Finally we provide additional syntax for specifications and commands.

abbreviation *H-sugar* :: $'a$ *pred* $\Rightarrow$ $'a$ *rel* $\Rightarrow$ $'a$ *pred* $\Rightarrow$ *bool* (‹*PRE - - POST -*› [*64*,*64*,*64*] *63*) **where**
PRE P X POST Q $\equiv$ *rel-kat.H* $\lceil P \rceil$ *X* $\lceil Q \rceil$

abbreviation *if-then-else-sugar* :: $'a$ *pred* $\Rightarrow$ $'a$ *rel* $\Rightarrow$ $'a$ *rel* $\Rightarrow$ $'a$ *rel* (‹*IF - THEN - ELSE - FI*› [*64*,*64*,*64*] *63*) **where**
IF P THEN X ELSE Y FI $\equiv$ *rel-kat.ifthenelse* $\lceil P \rceil$ *X Y*

abbreviation *while-sugar* :: $'a$ *pred* $\Rightarrow$ $'a$ *rel* $\Rightarrow$ $'a$ *rel* (‹*WHILE - DO - OD*› [*64*,*64*] *63*) **where**
WHILE P DO X OD $\equiv$ *rel-kat.while* $\lceil P \rceil$ *X*

abbreviation *while-inv-sugar* :: $'a$ *pred* $\Rightarrow$ $'a$ *pred* $\Rightarrow$ $'a$ *rel* $\Rightarrow$ $'a$ *rel* (‹*WHILE - INV - DO - OD*› [*64*,*64*,*64*] *63*) **where**
WHILE P INV I DO X OD $\equiv$ *rel-kat.while-inv* $\lceil P \rceil$ $\lceil I \rceil$ *X*

lemma *H-cond-iff2*[*simp*]: *PRE p* (*IF r THEN x ELSE y FI*) *POST q* $\longleftrightarrow$ (*PRE* ($p \sqcap r$) *x POST q*) $\wedge$ (*PRE* ($p \sqcap - r$) *y POST q*)
by (*simp add*: *rel-kat.H-cond-iff*)

end

4.1.6 Verification Examples

theory *VC-KAT-Examples*
imports *VC-KAT*
begin

lemma *euclid*:
PRE (λs::*nat store*. $s\ ''x'' = x \wedge s\ ''y'' = y$)
(*WHILE* ($\lambda s.\ s\ ''y'' \neq 0$) *INV* ($\lambda s.\ gcd\ (s\ ''x'')\ (s\ ''y'') = gcd\ x\ y$)
DO
($''z''$ ::= ($\lambda s.\ s\ ''y''$));
($''y''$ ::= ($\lambda s.\ s\ ''x''$ *mod* $s\ ''y''$));
($''x''$ ::= ($\lambda s.\ s\ ''z''$))
OD)
POST ($\lambda s.\ s\ ''x'' = gcd\ x\ y$)
apply (*rule sH-while-inv*)
apply *simp-all*
apply *force*
apply (*intro rel-kat.H-seq*)
apply (*subst H-assign*, *simp*)+
apply (*intro H-assign-var*)
using *gcd-red-nat* **by** *auto*

lemma *maximum*:
PRE (λs:: *nat store*. *True*)
(*IF* ($\lambda s.\ s\ ''x'' \geq s\ ''y''$)

THEN (*''z''* ::= (λ*s*. *s* *''x''*))
ELSE (*''z''* ::= (λ*s*. *s* *''y''*))
FI)
POST (λ*s*. *s* *''z''* = *max* (*s* *''x''*) (*s* *''y''*))
by *auto*

lemma *integer-division*:
PRE (λ*s*::*nat store*. *s* *''x''* ≥ *0*)
(*''q''* ::= (λ*s*. *0*));
(*''r''* ::= (λ*s*. *s* *''x''*));
(*WHILE* (λ*s*. *s* *''y''* ≤ *s* *''r''*) *INV* (λ*s*. *s* *''x''* = *s* *''q''* ∗ *s* *''y''* + *s* *''r''* ∧ *s* *''r''* ≥ *0*)
DO
(*''q''* ::= (λ*s*. *s* *''q''* + *1*));
(*''r''* ::= (λ*s*. *s* *''r''* − *s* *''y''*))
OD)
POST (λ*s*. *s* *''x''* = *s* *''q''* ∗ *s* *''y''* + *s* *''r''* ∧ *s* *''r''* ≥ *0* ∧ *s* *''r''* < *s* *''y''*)
apply (*intro rel-kat.H-seq*, *subst sH-while-inv*, *simp-all*)
apply *auto*[*1*]
apply (*intro rel-kat.H-seq*)
by (*subst H-assign*, *simp-all*)+

lemma *imp-reverse*:
PRE (λ*s*:: *'a list store*. *s* *''x''* = *X*)
(*''y''* ::= (λ*s*. []));
(*WHILE* (λ*s*. *s* *''x''* ≠ []) *INV* (λ*s*. *rev* (*s* *''x''*) @ *s* *''y''* = *rev* *X*)
DO
(*''y''* ::= (λ*s*. *hd* (*s* *''x''*) # *s* *''y''*));
(*''x''* ::= (λ*s*. *tl* (*s* *''x''*)))
OD)
POST (λ*s*. *s* *''y''*= *rev* *X*)
apply (*intro rel-kat.H-seq*, *rule sH-while-inv*)
apply *auto*[*2*]
apply (*rule rel-kat.H-seq*, *rule H-assign-var*)
apply *auto*[*1*]
apply (*rule H-assign-var*)
apply (*clarsimp*, *metis append.simps*(*1*) *append.simps*(*2*) *append-assoc hd-Cons-tl rev.simps*(*2*))
by *simp*

end

4.1.7 Verification Examples with Automated VCG

theory *VC-KAT-Examples2*
imports *VC-KAT HOL−Eisbach.Eisbach*
begin

The following simple tactic for verification condition generation has been implemented with the Eisbach proof methods language.

named-theorems *hl-intro*

declare *sH-while-inv* [*hl-intro*]
rel-kat.H-seq [*hl-intro*]
H-assign-var [*hl-intro*]
rel-kat.H-cond [*hl-intro*]

method *hoare* = (*rule hl-intro*; *hoare?*)

lemma *euclid*:
$PRE\ (\lambda s{::}nat\ store.\ s\ ''x'' = x \wedge s\ ''y'' = y)$
$(WHILE\ (\lambda s.\ s\ ''y'' \neq 0)\ INV\ (\lambda s.\ gcd\ (s\ ''x'')\ (s\ ''y'') = gcd\ x\ y)$
DO
$(''z'' ::= (\lambda s.\ s\ ''y''));$
$(''y'' ::= (\lambda s.\ s\ ''x''\ mod\ s\ ''y''));$
$(''x'' ::= (\lambda s.\ s\ ''z''))$
$OD)$
$POST\ (\lambda s.\ s\ ''x'' = gcd\ x\ y)$
apply *hoare*
using *gcd-red-nat* **by** *auto*

lemma *integer-division*:
$PRE\ (\lambda s{::}nat\ store.\ s\ ''x'' \geq 0)$
$(''q'' ::= (\lambda s.\ 0));$
$(''r'' ::= (\lambda s.\ s\ ''x''));$
$(WHILE\ (\lambda s.\ s\ ''y'' \leq s\ ''r'')\ INV\ (\lambda s.\ s\ ''x'' = s\ ''q'' * s\ ''y'' + s\ ''r'' \wedge s$
$''r'' \geq 0)$
DO
$(''q'' ::= (\lambda s.\ s\ ''q'' + 1));$
$(''r'' ::= (\lambda s.\ s\ ''r'' - s\ ''y''))$
$OD)$
$POST\ (\lambda s.\ s\ ''x'' = s\ ''q'' * s\ ''y'' + s\ ''r'' \wedge s\ ''r'' \geq 0 \wedge s\ ''r'' < s\ ''y'')$
by *hoare auto*

lemma *imp-reverse*:
$PRE\ (\lambda s{::}\ 'a\ list\ store.\ s\ ''x'' = X)$
$(''y'' ::= (\lambda s.\ []));$
$(WHILE\ (\lambda s.\ s\ ''x'' \neq [])\ INV\ (\lambda s.\ rev\ (s\ ''x'')\ @\ s\ ''y'' = rev\ X)$
DO
$(''y'' ::= (\lambda s.\ hd\ (s\ ''x'')\ \#\ s\ ''y''));$
$(''x'' ::= (\lambda s.\ tl\ (s\ ''x'')))$
$OD)$
$POST\ (\lambda s.\ s\ ''y''= rev\ X\ )$
apply *hoare*
apply *auto*[*3*]
apply (*clarsimp*, *metis* (*no-types*, *lifting*) *Cons-eq-appendI append-eq-append-conv2*
hd-Cons-tl rev.simps(*2*) *self-append-conv*)
by *simp*

end

4.2 Refinement Component

theory *RKAT*
imports *AVC-KAT/VC-KAT*

begin

4.2.1 RKAT: Definition and Basic Properties

A refinement KAT is a KAT expanded by Morgan's specification statement.

class *rkat* = *kat* +
fixes $R :: 'a \Rightarrow 'a \Rightarrow 'a$
assumes *spec-def*: $x \leq R\ p\ q \longleftrightarrow H\ p\ x\ q$

begin

lemma *R1*: $H\ p\ (R\ p\ q)\ q$
using *spec-def* **by** *blast*

lemma *R2*: $H\ p\ x\ q \Longrightarrow x \leq R\ p\ q$
by (*simp add*: *spec-def*)

4.2.2 Propositional Refinement Calculus

lemma *R-skip*: $1 \leq R\ p\ p$
proof −
have $H\ p\ 1\ p$
by (*simp add*: *H-skip*)
thus *?thesis*
by (*rule R2*)
qed

lemma *R-cons*: $t\ p \leq t\ p' \Longrightarrow t\ q' \leq t\ q \Longrightarrow R\ p'\ q' \leq R\ p\ q$
proof −
assume *h1*: $t\ p \leq t\ p'$ **and** *h2*: $t\ q' \leq t\ q$
have $H\ p'\ (R\ p'\ q')\ q'$
by (*simp add*: *R1*)
hence $H\ p\ (R\ p'\ q')\ q$
using *h1 h2 H-cons-1 H-cons-2* **by** *blast*
thus *?thesis*
by (*rule R2*)
qed

lemma *R-seq*: $(R\ p\ r) \cdot (R\ r\ q) \leq R\ p\ q$
proof −
have $H\ p\ (R\ p\ r)\ r$ **and** $H\ r\ (R\ r\ q)\ q$
by (*simp add*: *R1*)+

hence *H p* ((*R p r*) · (*R r q*)) *q*
  by (*rule H-seq-swap*)
thus *?thesis*
  by (*rule R2*)
qed

lemma *R-cond*: *if v then* (*R* (*t v* · *t p*) *q*) *else* (*R* (*n v* · *t p*) *q*) *fi* ≤ *R p q*
proof −
have *H* (*t v* · *t p*) (*R* (*t v* · *t p*) *q*) *q* **and** *H* (*n v* · *t p*) (*R* (*n v* · *t p*) *q*) *q*
  by (*simp add*: *R1*)+
hence *H p* (*if v then* (*R* (*t v* · *t p*) *q*) *else* (*R* (*n v* · *t p*) *q*) *fi*) *q*
  by (*simp add*: *H-cond n-mult-comm*)
thus *?thesis*
  by (*rule R2*)
qed

lemma *R-loop*: *while q do* (*R* (*t p* · *t q*) *p*) *od* ≤ *R p* (*t p* · *n q*)
proof −
have *H* (*t p* · *t q*) (*R* (*t p* · *t q*) *p*) *p*
  by (*simp-all add*: *R1*)
hence *H p* (*while q do* (*R* (*t p* · *t q*) *p*) *od*) (*t p* · *n q*)
  by (*simp add*: *H-loop*)
thus *?thesis*
  by (*rule R2*)
qed

lemma *R-zero-one*: *x* ≤ *R 0 1*
proof −
have *H 0 x 1*
  by (*simp add*: *H-def*)
thus *?thesis*
  by (*rule R2*)
qed

lemma *R-one-zero*: *R 1 0* = *0*
proof −
have *H 1* (*R 1 0*) *0*
  by (*simp add*: *R1*)
thus *?thesis*
  by (*simp add*: *H-def join.le-bot*)
qed

end

end

4.2.3 Models of Refinement KAT

theory *RKAT-Models*

imports *RKAT*

begin

So far only the relational model is developed.

definition *rel-R* :: $'a$ *rel* $\Rightarrow$ $'a$ *rel* $\Rightarrow$ $'a$ *rel* **where**
rel-R P $Q = \bigcup\{X.$ *rel-kat.H* P X $Q\}$

interpretation *rel-rkat*: *rkat* $(\cup)$ $(;)$ *Id* $\{\}$ $(\subseteq)$ $(\subset)$ *rtrancl* $(\lambda X.$ *Id* $\cap$ $-$ $X)$ *rel-R*
by (*standard*, *auto simp*: *rel-R-def rel-kat.H-def*)

end

theory *VC-RKAT*
imports ../*RKAT-Models*

begin

This component supports the step-wise refinement of simple while programs in a partial correctness setting.

4.2.4 Assignment Laws

The store model is taken from KAT

lemma *R-assign*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow (v ::= e) \subseteq$ *rel-R* $\lceil P\rceil$ $\lceil Q\rceil$
proof $-$
assume $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s)))$
hence *rel-kat.H* $\lceil P\rceil$ $(v ::= e)$ $\lceil Q\rceil$
by (*rule H-assign-var*)
thus *?thesis*
by (*rule rel-rkat.R2*)
qed

lemma *R-assignr*: $(\forall s.\ Q'\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow$ (*rel-R* $\lceil P\rceil$ $\lceil Q'\rceil$) ; $(v ::= e) \subseteq$ *rel-R* $\lceil P\rceil$ $\lceil Q\rceil$
by (*metis H-assign-var rel-kat.H-seq rel-rkat.R1 rel-rkat.R2*)

lemma *R-assignl*: $(\forall s.\ P\ s \longrightarrow P'\ (s\ (v := e\ s))) \Longrightarrow (v ::= e)$; (*rel-R* $\lceil P'\rceil$ $\lceil Q\rceil$) $\subseteq$ *rel-R* $\lceil P\rceil$ $\lceil Q\rceil$
by (*metis H-assign-var rel-kat.H-seq rel-rkat.R1 rel-rkat.R2*)

4.2.5 Simplified Refinement Laws

lemma *R-cons*: $(\forall s.\ P\ s \longrightarrow P'\ s) \Longrightarrow (\forall s.\ Q'\ s \longrightarrow Q\ s) \Longrightarrow$ *rel-R* $\lceil P'\rceil$ $\lceil Q'\rceil$ $\subseteq$ *rel-R* $\lceil P\rceil$ $\lceil Q\rceil$
by (*simp add*: *rel-rkat.R1 rel-rkat.R2 sH-cons-1 sH-cons-2*)

lemma *if-then-else-ref*: $X \subseteq X' \Longrightarrow Y \subseteq Y' \Longrightarrow IF\ P\ THEN\ X\ ELSE\ Y\ FI \subseteq IF\ P\ THEN\ X'\ ELSE\ Y'\ FI$
by (*auto simp*: *rel-kat.ifthenelse-def*)

lemma *while-ref*: $X \subseteq X' \Longrightarrow WHILE\ P\ DO\ X\ OD \subseteq WHILE\ P\ DO\ X'\ OD$
by (*simp add*: *rel-kat.while-def rel-dioid.mult-isol rel-dioid.mult-isor rel-kleene-algebra.star-iso*)

end

4.2.6 Refinement Examples

theory *VC-RKAT-Examples*
imports *VC-RKAT*
begin

Currently there is only one example, and no tactic for automating refinement proofs is provided.

lemma *var-swap-ref1*:
$rel\text{-}R\ \lceil\lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
$\supseteq (''z'' ::= (\lambda s.\ s\ ''x''));\ rel\text{-}R\ \lceil\lambda s.\ s\ ''z'' = a \wedge s\ ''y'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref2*:
$rel\text{-}R\ \lceil\lambda s.\ s\ ''z'' = a \wedge s\ ''y'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
$\supseteq (''x'' ::= (\lambda s.\ s\ ''y''));\ rel\text{-}R\ \lceil\lambda s.\ s\ ''z'' = a \wedge s\ ''x'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref3*:
$rel\text{-}R\ \lceil\lambda s.\ s\ ''z'' = a \wedge s\ ''x'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
$\supseteq (''y'' ::= (\lambda s.\ s\ ''z''));\ rel\text{-}R\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
by (*rule R-assignl*, *auto*)

lemma *var-swap-ref-var*:
$rel\text{-}R\ \lceil\lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b\rceil\ \lceil\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a\rceil$
$\supseteq (''z'' ::= (\lambda s.\ s\ ''x''));\ (''x'' ::= (\lambda s.\ s\ ''y''));\ (''y'' ::= (\lambda s.\ s\ ''z''))$
using *var-swap-ref1 var-swap-ref2 var-swap-ref3 rel-rkat.R-skip* **by** *fastforce*

end

5 Components Based on Kleene Algebra with Domain

theory *VC-KAD*

imports *KAD.Modal-Kleene-Algebra-Models ../P2S2R*

begin

5.1 Verification Component for Backward Reasoning

This component supports the verification of simple while programs in a partial correctness setting.

unbundle *no floor-ceiling-syntax*

notation *p2r* (‹⌈-⌉›)
notation *r2p* (‹⌊-⌋›)

context *antidomain-kleene-algebra*
begin

5.1.1 Additional Facts for KAD

lemma *fbox-shunt*: $d\ p \cdot d\ q \leq |x]\ t \longleftrightarrow d\ p \leq ad\ q + |x]\ t$
by (*metis a-6 a-antitone′ a-loc add-commute addual.ars-r-def am-d-def da-shunt2 fbox-def*)

5.1.2 Syntax for Conditionals and Loops

definition *cond* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹*if - then - else - fi*› [*64*,*64*,*64*] *63*) **where**
if p then x else y fi $= d\ p \cdot x + ad\ p \cdot y$

definition *while* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹*while - do - od*› [*64*,*64*] *63*) **where**
while p do x od $= (d\ p \cdot x)^\star \cdot ad\ p$

definition *whilei* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹*while - inv - do - od*› [*64*,*64*,*64*] *63*) **where**
while p inv i do x od = *while p do x od*

5.1.3 Basic Weakest (Liberal) Precondition Calculus

In the setting of Kleene algebra with domain, the wlp operator is the forward modal box operator of antidomain Kleene algebra.

lemma *fbox-export1*: $ad\ p + |x]\ q = |d\ p \cdot x]\ q$
using *a-d-add-closure addual.ars-r-def fbox-def fbox-mult* **by** *auto*

lemma *fbox-export2*: $|x]\ p \leq |x \cdot ad\ q]\ (d\ p \cdot ad\ q)$
proof −
{**fix** *t*
have $d\ t \cdot x \leq x \cdot d\ p \Longrightarrow d\ t \cdot x \cdot ad\ q \leq x \cdot ad\ q \cdot d\ p \cdot ad\ q$
by (*metis* (*full-types*) *a-comm-var a-mult-idem ads-d-def am2 ds.ddual.mult-assoc phl-export2*)
hence $d\ t \leq |x]\ p \Longrightarrow d\ t \leq |x \cdot ad\ q]\ (d\ p \cdot ad\ q)$

by (*metis a-closure′ addual.ars-r-def ans-d-def dka.dsg3 ds.ddual.mult-assoc fbox-def fbox-demodalisation3*)
}
thus *?thesis*
by (*metis a-closure′ addual.ars-r-def ans-d-def fbox-def order-refl*)
qed

lemma *fbox-export3*: $|x \cdot ad\ p]\ q = |x]\ (d\ p + d\ q)$
using *a-de-morgan-var-3 ds.ddual.mult-assoc fbox-def* **by** *auto*

lemma *fbox-seq* [*simp*]: $|x \cdot y]\ q = |x]\ |y]\ q$
by (*simp add*: *fbox-mult*)

lemma *fbox-seq-var*: $p' \leq |y]\ q \Longrightarrow p \leq |x]\ p' \Longrightarrow p \leq |x \cdot y]\ q$
proof −
assume *h1*: $p \leq |x]\ p'$ **and** *h2*: $p' \leq |y]\ q$
hence $|x]\ p' \leq |x]\ |y]\ q$
by (*simp add*: *dka.dom-iso fbox-iso*)
thus *?thesis*
by (*metis h1 dual-order.trans fbox-seq*)
qed

lemma *fbox-cond-var* [*simp*]: $|\mathit{if}\ p\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}]\ q = (ad\ p + |x]\ q) \cdot (d\ p + |y]\ q)$
using *cond-def a-closure′ ads-d-def ans-d-def fbox-add2 fbox-export1* **by** *auto*

lemma *fbox-cond-aux1* [*simp*]: $d\ p \cdot |\mathit{if}\ p\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}]\ q = d\ p \cdot |x]\ q$
proof −
have $d\ p \cdot |\mathit{if}\ p\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}]\ q = d\ p \cdot |x]\ q \cdot (d\ p + d\ (\ |y]\ q))$
using *a-d-add-closure addual.ars-r-def ds.ddual.mult-assoc fbox-def fbox-cond-var* **by** *auto*
thus *?thesis*
by (*metis addual.ars-r-def am2 dka.dns5 ds.ddual.mult-assoc fbox-def*)
qed

lemma *fbox-cond-aux2* [*simp*]: $ad\ p \cdot |\mathit{if}\ p\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}]\ q = ad\ p \cdot |y]\ q$
by (*metis cond-def a-closure′ add-commute addual.ars-r-def ans-d-def fbox-cond-aux1*)

lemma *fbox-cond* [*simp*]: $|\mathit{if}\ p\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}]\ q = (d\ p \cdot |x]\ q) + (ad\ p \cdot |y]\ q)$
proof −
have $|\mathit{if}\ p\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}]\ q = (d\ p + ad\ p) \cdot |\mathit{if}\ p\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}]\ q$
by (*simp add*: *addual.ars-r-def*)
thus *?thesis*
by (*metis distrib-right′ fbox-cond-aux1 fbox-cond-aux2*)
qed

lemma *fbox-cond-var2* [*simp*]: $|\mathit{if}\ p\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}]\ q = \mathit{if}\ p\ \mathit{then}\ |x]\ q\ \mathit{else}\ |y]\ q\ \mathit{fi}$
using *cond-def fbox-cond* **by** *auto*

lemma *fbox-while-unfold*: $|while\ t\ do\ x\ od]\ p = (d\ t + d\ p) \cdot (ad\ t + |x]\ |while\ t\ do\ x\ od]\ p)$
by (*metis fbox-export1 fbox-export3 dka.dom-add-closed fbox-star-unfold-var while-def*)

lemma *fbox-while-var1*: $d\ t \cdot |while\ t\ do\ x\ od]\ p = d\ t \cdot |x]\ |while\ t\ do\ x\ od]\ p$
by (*metis fbox-while-unfold a-mult-add ads-d-def dka.dns5 ds.ddual.mult-assoc join.sup-commute*)

lemma *fbox-while-var2*: $ad\ t \cdot |while\ t\ do\ x\ od]\ p \leq d\ p$
proof −
have $ad\ t \cdot |while\ t\ do\ x\ od]\ p = ad\ t \cdot (d\ t + d\ p) \cdot (ad\ t + |x]\ |while\ t\ do\ x\ od]\ p)$
by (*metis fbox-while-unfold ds.ddual.mult-assoc*)
also have ... $=$ $ad\ t \cdot d\ p \cdot (ad\ t + |x]\ |while\ t\ do\ x\ od]\ p)$
by (*metis a-de-morgan-var-3 addual.ars-r-def dka.dom-add-closed s4*)
also have ... $\leq d\ p \cdot (ad\ t + |x]\ |while\ t\ do\ x\ od]\ p)$
using *a-subid-aux1′ mult-isor* **by** *blast*
finally show *?thesis*
by (*metis a-de-morgan-var-3 a-mult-idem addual.ars-r-def ans4 dka.dsr5 dpdz.domain-1′′ dual-order.trans fbox-def*)
qed

lemma *fbox-while*: $d\ p \cdot d\ t \leq |x]\ p \Longrightarrow d\ p \leq |while\ t\ do\ x\ od]\ (d\ p \cdot ad\ t)$
proof −
assume $d\ p \cdot d\ t \leq |x]\ p$
hence $d\ p \leq |d\ t \cdot x]\ p$
by (*simp add*: *fbox-export1 fbox-shunt*)
hence $d\ p \leq |(d\ t \cdot x)^\star]\ p$
by (*simp add*: *fbox-star-induct-var*)
thus *?thesis*
using *order-trans while-def fbox-export2* **by** *presburger*
qed

lemma *fbox-while-var*: $d\ p = |d\ t \cdot x]\ p \Longrightarrow d\ p \leq |while\ t\ do\ x\ od]\ (d\ p \cdot ad\ t)$
by (*metis eq-refl fbox-export1 fbox-shunt fbox-while*)

lemma *fbox-whilei*: $d\ p \leq d\ i \Longrightarrow d\ i \cdot ad\ t \leq d\ q \Longrightarrow d\ i \cdot d\ t \leq |x]\ i \Longrightarrow d\ p \leq |while\ t\ inv\ i\ do\ x\ od]\ q$
proof −
assume *a1*: $d\ p \leq d\ i$ **and** *a2*: $d\ i \cdot ad\ t \leq d\ q$ **and** $d\ i \cdot d\ t \leq |x]\ i$
hence $d\ i \leq |while\ t\ inv\ i\ do\ x\ od]\ (d\ i \cdot ad\ t)$
by (*simp add*: *fbox-while whilei-def*)
also have ... $\leq |while\ t\ inv\ i\ do\ x\ od]\ q$
using *a2 dka.dom-iso fbox-iso* **by** *fastforce*
finally show *?thesis*
using *a1 dual-order.trans* **by** *blast*
qed

The next rule is used for dealing with while loops after a series of sequential steps.

lemma *fbox-whilei-break*: $d\ p \leq |y]\ i \Longrightarrow d\ i \cdot ad\ t \leq d\ q \Longrightarrow d\ i \cdot d\ t \leq |x]\ i \Longrightarrow d\ p \leq |y \cdot (while\ t\ inv\ i\ do\ x\ od)]\ q$
  apply (*rule fbox-seq-var*, *rule fbox-whilei*, *simp-all*, *blast*)
  using *fbox-simp* **by** *auto*

Finally we derive a frame rule.

lemma *fbox-frame*: $d\ p \cdot x \leq x \cdot d\ p \Longrightarrow d\ q \leq |x]\ t \Longrightarrow d\ p \cdot d\ q \leq |x]\ (d\ p \cdot d\ t)$
  using *dual.mult-isol-var fbox-add1 fbox-demodalisation3 fbox-simp* **by** *auto*

lemma *fbox-frame-var*: $d\ p \leq |x]\ p \Longrightarrow d\ q \leq |x]\ t \Longrightarrow d\ p \cdot d\ q \leq |x]\ (d\ p \cdot d\ t)$
  using *fbox-frame fbox-demodalisation3 fbox-simp* **by** *auto*

end

5.1.4 Store and Assignment

type-synonym $'a\ store = string \Rightarrow 'a$

notation *rel-antidomain-kleene-algebra.fbox* (‹*wp*›)
and *rel-antidomain-kleene-algebra.fdia* (‹*relfdia*›)

definition *gets* :: $string \Rightarrow ('a\ store \Rightarrow 'a) \Rightarrow 'a\ store\ rel$ (‹- ::= -› [70, 65] 61)
where
  $v ::= e = \{(s,s\ (v := e\ s))\ |s.\ True\}$

lemma *assign-prop*: $\lceil \lambda s.\ P\ (s\ (v := e\ s)) \rceil\ ;\ (v ::= e) = (v ::= e)\ ;\ \lceil P \rceil$
  by (*auto simp add*: *p2r-def gets-def*)

lemma *wp-assign* [*simp*]: $wp\ (v ::= e)\ \lceil Q \rceil = \lceil \lambda s.\ Q\ (s\ (v := e\ s)) \rceil$
  by (*auto simp*: *rel-antidomain-kleene-algebra.fbox-def gets-def rel-ad-def p2r-def*)

lemma *wp-assign-var* [*simp*]: $\lfloor wp\ (v ::= e)\ \lceil Q \rceil \rfloor = (\lambda s.\ Q\ (s\ (v := e\ s)))$
  by (*simp*, *auto simp*: *r2p-def p2r-def*)

lemma *wp-assign-det*: $wp\ (v ::= e)\ \lceil Q \rceil = relfdia\ (v ::= e)\ \lceil Q \rceil$
 by (*auto simp add*: *rel-antidomain-kleene-algebra.fbox-def rel-antidomain-kleene-algebra.fdia-def gets-def p2r-def rel-ad-def*, *fast*)

5.1.5 Simplifications

notation *rel-antidomain-kleene-algebra.ads-d* (‹*rdom*›)

abbreviation *spec-sugar* :: $'a\ pred \Rightarrow 'a\ rel \Rightarrow 'a\ pred \Rightarrow bool$ (‹*PRE - - POST* -› [64,64,64] 63) **where**
  $PRE\ P\ X\ POST\ Q \equiv rdom\ \lceil P \rceil \subseteq wp\ X\ \lceil Q \rceil$

abbreviation *cond-sugar* :: $'a$ *pred* ⇒ $'a$ *rel* ⇒ $'a$ *rel* ⇒ $'a$ *rel* (‹*IF - THEN - ELSE - FI*› [*64*,*64*,*64*] *63*) **where**
IF P THEN X ELSE Y FI ≡ *rel-antidomain-kleene-algebra.cond* ⌈*P*⌉ *X Y*

abbreviation *whilei-sugar* :: $'a$ *pred* ⇒ $'a$ *pred* ⇒ $'a$ *rel* ⇒ $'a$ *rel* (‹*WHILE - INV - DO - OD*› [*64*,*64*,*64*] *63*) **where**
WHILE P INV I DO X OD ≡ *rel-antidomain-kleene-algebra.whilei* ⌈*P*⌉ ⌈*I*⌉ *X*

lemma *d-p2r* [*simp*]: *rdom* ⌈*P*⌉ = ⌈*P*⌉
by (*simp add*: *p2r-def rel-antidomain-kleene-algebra.ads-d-def rel-ad-def*)

lemma *p2r-conj-hom-var-symm* [*simp*]: ⌈*P*⌉ ; ⌈*Q*⌉ = ⌈*P* ⊓ *Q*⌉
by (*simp add*: *p2r-conj-hom-var*)

lemma *p2r-neg-hom* [*simp*]: *rel-ad* ⌈*P*⌉ = ⌈− *P*⌉
by (*simp add*: *rel-ad-def p2r-def*)

lemma *wp-trafo*: ⌊*wp X* ⌈*Q*⌉⌋ = (λ*s*. ∀ *s*′. (*s*,*s*′) ∈ *X* ⟶ *Q s*′)
by (*auto simp*: *rel-antidomain-kleene-algebra.fbox-def rel-ad-def p2r-def r2p-def*)

lemma *wp-trafo-var*: ⌊*wp X* ⌈*Q*⌉⌋ *s* = (∀ *s*′. (*s*,*s*′) ∈ *X* ⟶ *Q s*′)
by (*simp add*: *wp-trafo*)

lemma *wp-simp*: *rdom* ⌈⌊*wp X Q*⌋⌉ = *wp X Q*
by (*metis d-p2r rel-antidomain-kleene-algebra.a-subid*′ *rel-antidomain-kleene-algebra.addual.bbox-def rpr*)

lemma *wp-simp-var* [*simp*]: *wp* ⌈*P*⌉ ⌈*Q*⌉ = ⌈− *P* ⊔ *Q*⌉
by (*simp add*: *rel-antidomain-kleene-algebra.fbox-def*)

lemma *impl-prop* [*simp*]: ⌈*P*⌉ ⊆ ⌈*Q*⌉ ⟷ (∀ *s*. *P s* ⟶ *Q s*)
by (*auto simp*: *p2r-def*)

lemma *p2r-eq-prop* [*simp*]: ⌈*P*⌉ = ⌈*Q*⌉ ⟷ (∀ *s*. *P s* ⟷ *Q s*)
by (*auto simp*: *p2r-def*)

lemma *impl-prop-var* [*simp*]: *rdom* ⌈*P*⌉ ⊆ *rdom* ⌈*Q*⌉ ⟷ (∀ *s*. *P s* ⟶ *Q s*)
by *simp*

lemma *p2r-eq-prop-var* [*simp*]: *rdom* ⌈*P*⌉ = *rdom* ⌈*Q*⌉ ⟷ (∀ *s*. *P s* ⟷ *Q s*)
by *simp*

lemma *wp-whilei*: (∀ *s*. *P s* ⟶ *I s*) ⟹ (∀ *s*. (*I* ⊓ −*T*) *s* ⟶ *Q s*) ⟹ (∀ *s*. (*I* ⊓ *T*) *s* ⟶ ⌊*wp X* ⌈*I*⌉⌋ *s*)
⟹ (∀ *s*. *P s* ⟶ ⌊*wp* (*WHILE T INV I DO X OD*) ⌈*Q*⌉⌋ *s*)
apply (*simp only*: *impl-prop-var*[*symmetric*] *wp-simp*)
by (*rule rel-antidomain-kleene-algebra.fbox-whilei*, *simp-all*, *simp-all add*: *p2r-def*)

end

5.1.6 Verification Examples

theory *VC-KAD-Examples*
imports *VC-KAD*

begin

lemma *euclid*:
  PRE (λs::*nat store*. s ''x'' = x ∧ s ''y'' = y)
   (*WHILE* (λs. s ''y'' ≠ *0*) *INV* (λs. *gcd* (s ''x'') (s ''y'') = *gcd* x y)
    DO
     (''z'' ::= (λs. s ''y''));
     (''y'' ::= (λs. s ''x'' *mod* s ''y''));
     (''x'' ::= (λs. s ''z''))
    OD)
   POST (λs. s ''x'' = *gcd* x y)
  by (*rule rel-antidomain-kleene-algebra.fbox-whilei*, *auto simp*: *gcd-non-0-nat*)

lemma *euclid-diff*:
   PRE (λs::*nat store*. s ''x'' = x ∧ s ''y'' = y ∧ x > *0* ∧ y > *0*)
    (*WHILE* (λs. s ''x''≠ s ''y'') *INV* (λs. *gcd* (s ''x'') (s ''y'') = *gcd* x y)
     DO
       (*IF* (λs. s ''x'' > s ''y'')
        THEN (''x'' ::= (λs. s ''x'' − s ''y''))
        ELSE (''y'' ::= (λs. s ''y'' − s ''x''))
        FI)
    OD)
    POST (λs. s ''x'' = *gcd* x y)
  apply (*rule rel-antidomain-kleene-algebra.fbox-whilei*, *simp-all*)
  apply *auto*[*1*]
  by (*metis gcd.commute gcd-diff1-nat le-cases nat-less-le*)

lemma *varible-swap*:
  PRE (λs. s ''x'' = a ∧ s ''y'' = b)
    (''z'' ::= (λs. s ''x''));
    (''x'' ::= (λs. s ''y''));
    (''y'' ::= (λs. s ''z''))
   POST (λs. s ''x'' = b ∧ s ''y'' = a)
  by *simp*

lemma *maximum*:
  PRE (λs:: *nat store*. *True*)
   (*IF* (λs. s ''x'' ≥ s ''y'')
    THEN (''z'' ::= (λs. s ''x''))
    ELSE (''z'' ::= (λs. s ''y''))
    FI)
   POST (λs. s ''z'' = *max* (s ''x'') (s ''y''))
  by *auto*

lemma *integer-division*:

```
  PRE (λs::nat store. x ≥ 0)
   (''q'' ::= (λs. 0));
   (''r'' ::= (λs. x));
   (WHILE (λs. y ≤ s ''r'') INV (λs. x = s ''q'' * y + s ''r'' ∧ s ''r'' ≥ 0)
    DO
     (''q'' ::= (λs. s ''q'' + 1));
     (''r'' ::= (λs. s ''r'' − y))
     OD)
   POST (λs. x = s ''q'' * y + s ''r'' ∧ s ''r'' ≥ 0 ∧ s ''r'' < y)
  by (rule rel-antidomain-kleene-algebra.fbox-whilei-break, simp-all)
```

lemma *factorial*:

```
  PRE (λs::nat store. True)
   (''x'' ::= (λs. 0));
   (''y'' ::= (λs. 1));
   (WHILE (λs. s ''x'' ≠ x0) INV (λs. s ''y'' = fact (s ''x''))
   DO
     (''x'' ::= (λs. s ''x'' + 1));
     (''y'' ::= (λs. s ''y'' · s ''x''))
   OD)
   POST (λs. s ''y'' = fact x0)
  by (rule rel-antidomain-kleene-algebra.fbox-whilei-break, simp-all)
```

lemma *my-power*:

```
  PRE (λs::nat store. True)
   (''i'' ::= (λs. 0));
   (''y'' ::= (λs. 1));
   (WHILE (λs. s ''i'' < n) INV (λs. s ''y'' = x ^ (s ''i'') ∧ s ''i'' ≤ n)
     DO
       (''y'' ::= (λs. (s ''y'') * x));
       (''i'' ::= (λs. s ''i'' + 1))
     OD)
   POST (λs. s ''y'' = x ^ n)
  by (rule rel-antidomain-kleene-algebra.fbox-whilei-break, auto)
```

lemma *imp-reverse*:

```
  PRE (λs:: 'a list store. s ''x'' = X)
   (''y'' ::= (λs. []));
   (WHILE (λs. s ''x'' ≠ []) INV (λs. rev (s ''x'') @ s ''y'' = rev X)
    DO
     (''y'' ::= (λs. hd (s ''x'') # s ''y''));
     (''x'' ::= (λs. tl (s ''x'')))
    OD)
   POST (λs. s ''y''= rev X )
  apply (rule rel-antidomain-kleene-algebra.fbox-whilei-break, simp-all)
  apply auto[1]
  by (safe, metis append.simps append-assoc hd-Cons-tl rev.simps(2))
```

end

5.1.7 Verification Examples with Automated VCG

theory *VC-KAD-Examples2*
imports *VC-KAD HOL−Eisbach.Eisbach*
begin

We have provide a simple tactic in the Eisbach proof method language. Additional simplification steps are sometimes needed to bring the resulting verification conditions into shape for first-order automated theorem proving.

named-theorems *ht*

```
declare rel-antidomain-kleene-algebra.fbox-whilei [ht]
  rel-antidomain-kleene-algebra.fbox-seq-var [ht]
  subset-refl[ht]

method hoare = (rule ht; hoare?)

lemma euclid2:
  PRE (λs::nat store. s ''x'' = x ∧ s ''y'' = y)
   (WHILE (λs. s ''y'' ≠ 0) INV (λs. gcd (s ''x'') (s ''y'') = gcd x y)
    DO
     (''z'' ::= (λs. s ''y''));
     (''y'' ::= (λs. s ''x'' mod s ''y''));
     (''x'' ::= (λs. s ''z''))
    OD)
   POST (λs. s ''x'' = gcd x y)
  apply hoare
  using gcd-red-nat by auto

lemma euclid-diff2:
   PRE (λs::nat store. s ''x'' = x ∧ s ''y'' = y ∧ x > 0 ∧ y > 0)
    (WHILE (λs. s ''x''≠ s ''y'') INV (λs. gcd (s ''x'') (s ''y'') = gcd x y)
     DO
       (IF (λs. s ''x'' >  s ''y'')
        THEN (''x'' ::= (λs. s ''x'' − s ''y''))
        ELSE (''y'' ::= (λs. s ''y'' − s ''x''))
        FI)
    OD)
    POST (λs. s ''x'' = gcd x y)
  apply (hoare, simp-all)
  apply auto[1]
  by (metis gcd.commute gcd-diff1-nat le-cases nat-less-le)

lemma integer-division2:
  PRE (λs::nat store. x ≥ 0)
    (''q'' ::= (λs. 0));
    (''r'' ::= (λs. x));
    (WHILE (λs. y ≤ s ''r'') INV (λs. x = s ''q'' ∗ y + s ''r'' ∧ s ''r'' ≥ 0)
     DO
```

$(''q'' ::= (\lambda s.\ s\ ''q'' + 1));$
$(''r'' ::= (\lambda s.\ s\ ''r'' - y))$
OD)
POST $(\lambda s.\ x = s\ ''q'' * y + s\ ''r'' \wedge s\ ''r'' \geq 0 \wedge s\ ''r'' < y)$
by *hoare simp-all*

lemma *factorial2*:
PRE (λs::*nat store. True*)
$(''x'' ::= (\lambda s.\ 0));$
$(''y'' ::= (\lambda s.\ 1));$
(*WHILE* $(\lambda s.\ s\ ''x'' \neq x0)$ *INV* $(\lambda s.\ s\ ''y'' = fact\ (s\ ''x''))$
DO
$(''x'' ::= (\lambda s.\ s\ ''x'' + 1));$
$(''y'' ::= (\lambda s.\ s\ ''y'' \cdot s\ ''x''))$
OD)
POST $(\lambda s.\ s\ ''y'' = fact\ x0)$
by *hoare simp-all*

lemma *my-power2*:
PRE (λs::*nat store. True*)
$(''i'' ::= (\lambda s.\ 0));$
$(''y'' ::= (\lambda s.\ 1));$
(*WHILE* $(\lambda s.\ s\ ''i'' < n)$ *INV* $(\lambda s.\ s\ ''y'' = x\ \hat{}\ (s\ ''i'') \wedge s\ ''i'' \leq n)$
DO
$(''y'' ::= (\lambda s.\ (s\ ''y'') * x));$
$(''i'' ::= (\lambda s.\ s\ ''i'' + 1))$
OD)
POST $(\lambda s.\ s\ ''y'' = x\ \hat{}\ n)$
by *hoare auto*

lemma *imp-reverse2*:
PRE (λs:: *'a list store.* $s\ ''x'' = X$)
$(''y'' ::= (\lambda s.\ []));$
(*WHILE* $(\lambda s.\ s\ ''x'' \neq [])$ *INV* $(\lambda s.\ rev\ (s\ ''x'')\ @\ s\ ''y'' = rev\ X)$
DO
$(''y'' ::= (\lambda s.\ hd\ (s\ ''x'')\ \#\ s\ ''y''));$
$(''x'' ::= (\lambda s.\ tl\ (s\ ''x'')))$
OD)
POST $(\lambda s.\ s\ ''y''= rev\ X\ )$
apply (*hoare*, *simp-all*)
apply *auto*[*1*]
by (*clarsimp*, *metis append.simps append-assoc hd-Cons-tl rev.simps*(*2*))

end

5.2 Verification Component for Forward Reasoning

theory *VC-KAD-dual*
imports *VC-KAD*

begin

context *modal-kleene-algebra*
begin

This component supports the verification of simple while programs in a partial correctness setting.

5.2.1 Basic Strongest Postcondition Calculus

In modal Kleene algebra, strongest postconditions are backward diamond operators. These are linked with forward boxes aka weakest preconditions by a Galois connection. This duality has been implemented in the AFP entry for Kleene algebra with domain and is picked up automatically in the following proofs.

lemma *r-ad* [*simp*]: $r\ (ad\ p) = ad\ p$
  using *a-closure addual.ars-r-def am-d-def domrangefix* **by** *auto*

lemma *bdia-export1*: $\langle x|\ (r\ p \cdot r\ t) = \langle r\ t \cdot x|\ p$
  by (*metis ardual.ads-d-def ardual.ds.ddual.rsr2 ardual.ds.fdia-mult bdia-def*)

lemma *bdia-export2*: $r\ p \cdot \langle x|\ q = \langle x \cdot r\ p|\ q$
  using *ardual.ads-d-def ardual.am2 ardual.fdia-export-2 bdia-def* **by** *auto*

lemma *bdia-seq* [*simp*]: $\langle x \cdot y|\ q = \langle y|\ \langle x|\ q$
  by (*simp add*: *ardual.ds.fdia-mult*)

lemma *bdia-seq-var*: $\langle x|\ p \le p' \Longrightarrow \langle y|\ p' \le q \Longrightarrow \langle x \cdot y|\ p \le q$
 by (*metis ardual.ds.fd-subdist-1 ardual.ds.fdia-mult dual-order.trans join.sup-absorb2*)

lemma *bdia-cond-var* [*simp*]: $\langle if\ p\ then\ x\ else\ y\ fi|\ q = \langle x|\ (d\ p \cdot r\ q) + \langle y|\ (ad\ p \cdot r\ q)$
  by (*metis* (*no-types*, *lifting*) *bdia-export1 a4' a-de-morgan a-de-morgan-var-3 a-subid-aux1' ardual.ds.fdia-add2 dka.dnso1 dka.dsg4 domrange dpdz.dnso1 cond-def join.sup.absorb-iff1 rangedom*)

lemma *bdia-while*: $\langle x|\ (d\ t \cdot r\ p) \le r\ p \Longrightarrow \langle while\ t\ do\ x\ od|\ p \le r\ p \cdot ad\ t$
proof −
  assume $\langle x|\ (d\ t \cdot r\ p) \le r\ p$
  hence $\langle d\ t \cdot x|\ p \le r\ p$
    by (*metis bdia-export1 dka.dsg4 domrange rangedom*)
  hence $\langle (d\ t \cdot x)^\star|\ p \le r\ p$
    by (*meson ardual.fdemodalisation22 ardual.kat-2-equiv-opp star-sim1*)
  hence $r\ (ad\ t) \cdot \langle (d\ t \cdot x)^\star|\ p \le r\ p \cdot ad\ t$
    by (*metis ardual.dpdz.dsg4 ars-r-def mult-isol r-ad*)
  thus *?thesis*
    by (*metis bdia-export2 while-def r-ad*)
qed

lemma *bdia-whilei*: $r\ p \leq r\ i \Longrightarrow r\ i \cdot ad\ t \leq r\ q \Longrightarrow \langle x|\ (d\ t \cdot r\ i) \leq r\ i \Longrightarrow$ $\langle$*while t inv i do x od*$|\ p \leq r\ q$
proof −
  assume *a1*: $r\ p \leq r\ i$ **and** *a2*: $r\ i \cdot ad\ t \leq r\ q$ **and** $\langle x|\ (d\ t \cdot r\ i) \leq r\ i$
  hence $\langle$*while t inv i do x od*$|\ i \leq r\ i \cdot ad\ t$
    by (*simp add*: *bdia-while whilei-def*)
  hence $\langle$*while t inv i do x od*$|\ i \leq r\ q$
    using *a2 dual-order.trans* **by** *blast*
  hence $r\ i \leq |$*while t inv i do x od*$]\ r\ q$
    using *ars-r-def box-diamond-galois-1 domrange* **by** *fastforce*
  hence $r\ p \leq |$*while t inv i do x od*$]\ r\ q$
    using *a1 dual-order.trans* **by** *blast*
  thus *?thesis*
    using *ars-r-def box-diamond-galois-1 domrange* **by** *fastforce*
qed

lemma *bdia-whilei-break*: $\langle y|\ p \leq r\ i \Longrightarrow r\ i \cdot ad\ t \leq r\ q \Longrightarrow \langle x|\ (d\ t \cdot r\ i) \leq r\ i$ $\Longrightarrow \langle y \cdot ($*while t inv i do x od*$)|\ p \leq r\ q$
  using *bdia-whilei ardual.ads-d-def ardual.ds.fdia-mult bdia-def* **by** *auto*

end

5.2.2 Floyd's Assignment Rule

lemma *bdia-assign* [*simp*]: *rel-antirange-kleene-algebra.bdia* $(v ::= e)\ \lceil P \rceil = \lceil \lambda s.$ $\exists\, w.\ s\ v = e\ (s(v := w)) \wedge P\ (s(v{:=}w)) \rceil$
  apply (*simp add*: *rel-antirange-kleene-algebra.bdia-def gets-def p2r-def rel-ar-def*)
  apply *safe*
  by (*metis fun-upd-apply fun-upd-triv fun-upd-upd*, *fastforce*)

lemma *d-p2r* [*simp*]: *rel-antirange-kleene-algebra.ars-r* $\lceil P \rceil = \lceil P \rceil$
  by (*simp add*: *p2r-def rel-antirange-kleene-algebra.ars-r-def rel-ar-def*)

abbreviation *fspec-sugar* :: *'a pred* ⇒ *'a rel* ⇒ *'a pred* ⇒ *bool* (‹*FPRE - - POST -*› [*64*,*64*,*64*] *63*) **where**
  FPRE P X POST Q ≡ *rel-antirange-kleene-algebra.bdia X* $\lceil P \rceil \subseteq$ *rel-antirange-kleene-algebra.ars-r* $\lceil Q \rceil$

end

5.2.3 Verification Examples

theory *VC-KAD-dual-Examples*
imports *VC-KAD-dual*

begin

The proofs are essentially the same as with forward boxes.

lemma *euclid*:

```
  FPRE (λs::nat store. s ''x'' = x ∧ s ''y'' = y)
   (WHILE (λs. s ''y'' ≠ 0) INV (λs. gcd (s ''x'') (s ''y'') = gcd x y)
    DO
     (''z'' ::= (λs. s ''y''));
     (''y'' ::= (λs. s ''x'' mod s ''y''));
     (''x'' ::= (λs. s ''z''))
    OD)
   POST (λs. s ''x'' = gcd x y)
  by (rule rel-modal-kleene-algebra.bdia-whilei, auto simp: gcd-non-0-nat)

lemma euclid-diff:
   FPRE (λs::nat store. s ''x'' = x ∧ s ''y'' = y ∧ x > 0 ∧ y > 0)
    (WHILE (λs. s ''x''≠ s ''y'') INV (λs. gcd (s ''x'') (s ''y'') = gcd x y)
     DO
        (IF (λs. s ''x'' >  s ''y'')
         THEN (''x'' ::= (λs. s ''x'' − s ''y''))
         ELSE (''y'' ::= (λs. s ''y'' − s ''x''))
         FI)
    OD)
    POST (λs. s ''x'' = gcd x y)
  apply (rule rel-modal-kleene-algebra.bdia-whilei, simp-all)
  apply auto[1]
  by (metis gcd.commute gcd-diff1-nat le-cases nat-less-le)

lemma varible-swap:
  FPRE (λs. s ''x'' = a ∧ s ''y'' = b)
    (''z'' ::= (λs. s ''x''));
    (''x'' ::= (λs. s ''y''));
    (''y'' ::= (λs. s ''z''))
   POST (λs. s ''x'' = b ∧ s ''y'' = a)
  by simp

lemma maximum:
  FPRE (λs:: nat store. True)
   (IF (λs. s ''x'' ≥ s ''y'')
    THEN (''z'' ::= (λs. s ''x''))
    ELSE (''z'' ::= (λs. s ''y''))
    FI)
   POST (λs. s ''z'' = max (s ''x'') (s ''y''))
  by auto

lemma integer-division:
  FPRE (λs::nat store. x ≥ 0)
    (''q'' ::= (λs. 0));
    (''r'' ::= (λs. x));
    (WHILE (λs. y ≤ s ''r'') INV (λs. x = s ''q'' ∗ y + s ''r'' ∧ s ''r'' ≥ 0)
     DO
      (''q'' ::= (λs. s ''q'' + 1));
      (''r'' ::= (λs. s ''r'' − y))
```

```
    OD)
  POST (λs. x = s ''q'' * y + s ''r'' ∧ s ''r'' ≥ 0 ∧ s ''r'' < y)
  by (rule rel-modal-kleene-algebra.bdia-whilei-break, simp-all, auto simp: p2r-def)

lemma factorial:
  FPRE (λs::nat store. True)
  (''x'' ::= (λs. 0));
  (''y'' ::= (λs. 1));
  (WHILE (λs. s ''x'' ≠ x0) INV (λs. s ''y'' = fact (s ''x''))
  DO
    (''x'' ::= (λs. s ''x'' + 1));
    (''y'' ::= (λs. s ''y'' · s ''x''))
  OD)
  POST (λs. s ''y'' = fact x0)
  by (rule rel-modal-kleene-algebra.bdia-whilei-break, simp-all, auto simp: p2r-def)

lemma my-power:
  FPRE (λs::nat store. True)
  (''i'' ::= (λs. 0));
  (''y'' ::= (λs. 1));
  (WHILE (λs. s ''i'' < n) INV (λs. s ''y'' = x ^ (s ''i'') ∧ s ''i'' ≤ n)
    DO
      (''y'' ::= (λs. (s ''y'') * x));
      (''i'' ::= (λs. s ''i'' + 1))
    OD)
  POST (λs. s ''y'' = x ^ n)
 by (rule rel-modal-kleene-algebra.bdia-whilei-break, simp-all, auto simp add: p2r-def)

lemma imp-reverse:
  FPRE (λs:: 'a list store. s ''x'' = X)
  (''y'' ::= (λs. []));
  (WHILE (λs. s ''x'' ≠ []) INV (λs. rev (s ''x'') @ s ''y'' = rev X)
   DO
    (''y'' ::= (λs. hd (s ''x'') # s ''y''));
    (''x'' ::= (λs. tl (s ''x'')))
   OD)
  POST (λs. s ''y''= rev X )
  apply (rule rel-modal-kleene-algebra.bdia-whilei-break, simp-all)
  apply auto[1]
  by (safe, metis append.simps append-assoc hd-Cons-tl rev.simps(2))

end
```

5.3 Verification Component for Total Correctness

theory *VC-KAD-wf*

imports *VC-KAD KAD.Modal-Kleene-Algebra-Applications*

begin

This component supports the verification of simple while programs in a total correctness setting.

5.3.1 Relation Divergence Kleene Algebras

Divergence Kleene algebras have been formalised in the AFP entry for Kleene algebra with domain. The nabla or divergence operation models those states of a relation from which infinitely ascending chains may start.

definition *rel-nabla* :: *'a rel* ⇒ *'a rel* **where**
rel-nabla X = $\bigcup$ {*P*. *P* ⊆ *relfdia X P*}

definition *rel-nabla-bin* :: *'a rel* ⇒ *'a rel* ⇒ *'a rel* **where**
rel-nabla-bin X Q = $\bigcup$ {*P*. *P* ⊆ *relfdia X P* ∪ *rdom Q*}

lemma *rel-nabla-d-closed* [*simp*]: *rdom* (*rel-nabla x*) = *rel-nabla x*
by (*auto simp*: *rel-nabla-def rel-antidomain-kleene-algebra.fdia-def rel-antidomain-kleene-algebra.ads-d-def rel-ad-def*)

lemma *rel-nabla-bin-d-closed* [*simp*]: *rdom* (*rel-nabla-bin x q*) = *rel-nabla-bin x q*
by (*auto simp*: *rel-nabla-bin-def rel-antidomain-kleene-algebra.fdia-def rel-antidomain-kleene-algebra.ads-d-def rel-ad-def*)

lemma *rel-nabla-unfold*: *rel-nabla X* ⊆ *relfdia X* (*rel-nabla X*)
by (*simp add*: *rel-nabla-def rel-ad-def rel-antidomain-kleene-algebra.fdia-def*, *blast*)

lemma *rel-nabla-bin-unfold*: *rel-nabla-bin X Q* ⊆ *relfdia X* (*rel-nabla-bin X Q*) ∪ *rdom Q*
by (*simp add*: *rel-nabla-bin-def rel-ad-def rel-antidomain-kleene-algebra.fdia-def*, *blast*)

lemma *rel-nabla-coinduct-var*: *P* ⊆ *relfdia X P* ⟹ *P* ⊆ *rel-nabla X*
by (*simp add*: *rel-nabla-def rel-antidomain-kleene-algebra.fdia-def rel-ad-def*, *blast*)

lemma *rel-nabla-bin-coinduct*: *P* ⊆ *relfdia X P* ∪ *rdom Q* ⟹ *P* ⊆ *rel-nabla-bin X Q*
by (*simp add*: *rel-nabla-bin-def rel-antidomain-kleene-algebra.fdia-def rel-ad-def*, *blast*)

The two fusion lemmas are, in fact, hard-coded fixpoint fusion proofs. They might be replaced by more generic fusion proofs eventually.

lemma *nabla-fusion1*: *rel-nabla X* ∪ *relfdia* (X^*) *Q* ⊆ *rel-nabla-bin X Q*
proof −
have *rel-nabla X* ∪ *relfdia* (X^*) *Q* ⊆ *relfdia X* (*rel-nabla X*) ∪ *relfdia X* (*relfdia* (X^*) *Q*) ∪ *rdom Q*
by (*metis* (*no-types*, *lifting*) *Un-mono inf-sup-aci*(*6*) *order-refl rel-antidomain-kleene-algebra.dka.fdia-star-un rel-nabla-unfold sup.commute*)

also have ... = *relfdia X* (*rel-nabla X* ∪ *relfdia* (*X**) *Q*) ∪ *rdom Q*
by (*simp add*: *rel-antidomain-kleene-algebra.dka.fdia-add1*)
finally show *?thesis*
using *rel-nabla-bin-coinduct* **by** *blast*
qed

lemma *rel-ad-inter-seq*: *rel-ad X* ∩ *rel-ad Y* = *rel-ad X* ; *rel-ad Y*
by (*auto simp*: *rel-ad-def*)

lemma *fusion2-aux2*: *rdom* (*rel-nabla-bin X Q*) ⊆ *rdom* (*rel-nabla-bin X Q* ∩ *rel-ad* (*relfdia* (*X**) *Q*) ∪ *relfdia* (*X**) *Q*)
apply (*auto simp*: *rel-antidomain-kleene-algebra.ads-d-def rel-ad-def*)
by (*metis pair-in-Id-conv r-into-rtrancl rel-antidomain-kleene-algebra.a-one rel-antidomain-kleene-algebra.a-s*
rel-antidomain-kleene-algebra.addual.ars-r-def rel-antidomain-kleene-algebra.dka.dns1''
rel-antidomain-kleene-algebra.dpdz.dom-one rel-antidomain-kleene-algebra.ds.ddual.rsr5
rel-antidomain-kleene-algebra.dual.conway.dagger-unfoldr-eq rel-antidomain-kleene-algebra.dual.tc-eq
rel-nabla-bin-d-closed)

lemma *nabla-fusion2*: *rel-nabla-bin X Q* ⊆ *rel-nabla X* ∪ *relfdia* (*X**) *Q*
proof −
have *rel-nabla-bin X Q* ∩ *rel-ad* (*relfdia* (*X**) *Q*) ⊆ (*relfdia X* (*rel-nabla-bin X Q*) ∪ *rdom Q*) ∩ *rel-ad* (*relfdia* (*X**) *Q*)
by (*meson Int-mono equalityD1 rel-nabla-bin-unfold*)
also have ... ⊆ (*relfdia X* (*rel-nabla-bin X Q* ∩ *rel-ad* (*relfdia* (*X**) *Q*) ∪ *relfdia* (*X**) *Q*) ∪ *rdom Q*) ∩ *rel-ad* (*relfdia* (*X**) *Q*)
using *fusion2-aux2 rel-antidomain-kleene-algebra.dka.fd-iso1* **by** *blast*
also have ... = (*relfdia X* (*rel-nabla-bin X Q* ∩ *rel-ad* (*relfdia* (*X**) *Q*)) ∪ *relfdia X* (*relfdia* (*X**) *Q*) ∪ *rdom Q*) ∩ *rel-ad* (*relfdia* (*X**) *Q*)
by (*simp add*: *rel-antidomain-kleene-algebra.dka.fdia-add1*)
also have ... = (*relfdia X* (*rel-nabla-bin X Q* ∩ *rel-ad* (*relfdia* (*X**) *Q*)) ∪ *relfdia* (*X**) *Q*) ∩ *rel-ad* (*relfdia* (*X**) *Q*)
using *rel-antidomain-kleene-algebra.dka.fdia-star-unfold-var* **by** *blast*
finally have *rel-nabla-bin X Q* ∩ *rel-ad* (*relfdia* (*X**) *Q*) ⊆ *relfdia X* ((*rel-nabla-bin X Q*) ∩ *rel-ad* (*relfdia* (*X**) *Q*))
by (*metis* (*no-types*, *lifting*) *inf-commute order-trans-rules*(*23*) *rel-ad-inter-seq*
rel-antidomain-kleene-algebra.a-mult-add rel-antidomain-kleene-algebra.a-subid-aux1'
rel-antidomain-kleene-algebra.addual.bdia-def rel-antidomain-kleene-algebra.ds.ddual.rsr5)
hence *rdom* (*rel-nabla-bin X Q*) ; *rel-ad* (*relfdia* (*X**) *Q*) ⊆ *rdom* (*rel-nabla X*)
by (*metis rel-ad-inter-seq rel-antidomain-kleene-algebra.addual.ars-r-def rel-nabla-bin-d-closed*
rel-nabla-coinduct-var rel-nabla-d-closed)
thus *?thesis*
by (*metis rel-antidomain-kleene-algebra.addual.ars-r-def rel-antidomain-kleene-algebra.addual.bdia-def*
rel-antidomain-kleene-algebra.d-a-galois1 rel-antidomain-kleene-algebra.dpdz.domain-invol
rel-nabla-bin-d-closed rel-nabla-d-closed)
qed

lemma *rel-nabla-coinduct*: *P* ⊆ *relfdia X P* ∪ *rdom Q* ⟹ *P* ⊆ *rel-nabla X* ∪ *relfdia* (*rtrancl X*) *Q*
by (*meson nabla-fusion2 order-trans rel-nabla-bin-coinduct*)

interpretation *rel-fdivka*: *fdivergence-kleene-algebra rel-ad* $(\cup)$ $(;)$ *Id* $\{\}$ $(\subseteq)$ $(\subset)$ *rtrancl rel-nabla*
proof
  fix $x\ y\ z$:: *'a rel*
  show *rdom* (*rel-nabla* x) = *rel-nabla* x
    by *simp*
  show *rel-nabla* $x \subseteq$ *relfdia* x (*rel-nabla* x)
    by (*simp add*: *rel-nabla-unfold*)
  show *rdom* $y \subseteq$ *relfdia* $x\ y \cup$ *rdom* $z \Longrightarrow$ *rdom* $y \subseteq$ *rel-nabla* $x \cup$ *relfdia* $(x^*)\ z$
    by (*simp add*: *rel-nabla-coinduct*)
qed

5.3.2 Meta-Equational Loop Rule

context *fdivergence-kleene-algebra*
begin

The rule below is inspired by Arden' rule from language theory. It can be used in total correctness proofs.

lemma *fdia-arden*: $\nabla x = 0 \Longrightarrow d\ p \leq d\ q + |x\rangle\ p \Longrightarrow d\ p \leq |x^\star\rangle\ q$
proof −
  assume *a1*: $\nabla x =$ *zero-class.zero*
  assume $d\ p \leq d\ q + |x\rangle\ p$
  then have *ad* (*ad* p) $\leq$ *zero-class.zero* + *ad* (*ad* $(x^\star \cdot q)$)
    using *a1 add-commute ads-d-def dka.fd-def nabla-coinduction* **by** *force*
  then show *?thesis*
    by (*simp add*: *ads-d-def dka.fd-def*)
qed

lemma *fdia-arden-eq*: $\nabla x = 0 \Longrightarrow d\ p = d\ q + |x\rangle\ p \Longrightarrow d\ p = |x^\star\rangle\ q$
  by (*simp add*: *fdia-arden dka.fdia-star-induct-eq order.eq-iff*)

lemma *fdia-arden-iff*: $\nabla x = 0 \Longrightarrow (d\ p = d\ q + |x\rangle\ p \longleftrightarrow d\ p = |x^\star\rangle\ q)$
  by (*metis fdia-arden-eq dka.fdia-d-simp dka.fdia-star-unfold-var*)

lemma $|x^\star]\ p \leq |x]\ p$
  by (*simp add*: *fbox-antitone-var*)

lemma *fbox-arden*: $\nabla x = 0 \Longrightarrow d\ q \cdot |x]\ p \leq d\ p \Longrightarrow |x^\star]\ q \leq d\ p$
proof −
  assume *h1*: $\nabla x = 0$ **and** $d\ q \cdot |x]\ p \leq d\ p$
  hence *ad* $p \leq$ *ad* $(d\ q \cdot |x]\ p)$
   by (*metis a-antitone' a-subid addual.ars-r-def dpdz.domain-subid dual-order.trans*)
  hence *ad* $p \leq$ *ad* $q + |x\rangle$ *ad* p
    by (*simp add*: *a-6 addual.bbox-def ds.fd-def*)
  hence *ad* $p \leq |x^\star\rangle$ *ad* q
    by (*metis fdia-arden h1 a-4 ads-d-def dpdz.dsg1 fdia-def meet-ord-def*)
  thus *?thesis*

by (*metis a-antitone′ ads-d-def fbox-simp fdia-fbox-de-morgan-2*)
qed

lemma *fbox-arden-eq*: $\nabla x = 0 \Longrightarrow d\ q \cdot |x]\ p = d\ p \Longrightarrow |x^\star]\ q = d\ p$
by (*simp add*: *fbox-arden order.antisym fbox-star-induct-eq*)

lemma *fbox-arden-iff*: $\nabla x = 0 \Longrightarrow (d\ p = d\ q \cdot |x]\ p \longleftrightarrow d\ p = |x^\star]\ q)$
by (*metis fbox-arden-eq fbox-simp fbox-star-unfold-var*)

lemma *fbox-arden-while-iff*: $\nabla\ (d\ t \cdot x) = 0 \Longrightarrow (d\ p = (d\ t + d\ q) \cdot |d\ t \cdot x]\ p \longleftrightarrow d\ p = |\mathit{while}\ t\ \mathit{do}\ x\ \mathit{od}]\ q)$
by (*metis fbox-arden-iff dka.dom-add-closed fbox-export3 while-def*)

lemma *fbox-arden-whilei*: $\nabla\ (d\ t \cdot x) = 0 \Longrightarrow (d\ i = (d\ t + d\ q) \cdot |d\ t \cdot x]\ i \Longrightarrow d\ i = |\mathit{while}\ t\ \mathit{inv}\ i\ \mathit{do}\ x\ \mathit{od}]\ q)$
using *fbox-arden-while-iff whilei-def* **by** *auto*

lemma *fbox-arden-whilei-iff*: $\nabla\ (d\ t \cdot x) = 0 \Longrightarrow (d\ i = (d\ t + d\ q) \cdot |d\ t \cdot x]\ i \longleftrightarrow d\ i = |\mathit{while}\ t\ \mathit{inv}\ i\ \mathit{do}\ x\ \mathit{od}]\ q)$
using *fbox-arden-while-iff whilei-def* **by** *auto*

5.3.3 Noethericity and Absence of Divergence

Noetherian elements have been defined in the AFP entry for Kleene algebra with domain. First we show that noethericity and absence of divergence coincide. Then we turn to the relational model and show that noetherian relations model terminating programs.

lemma *noether-nabla*: $\mathit{Noetherian}\ x \Longrightarrow \nabla\ x = 0$
by (*metis nabla-closure nabla-unfold noetherian-alt*)

lemma *nabla-noether-iff*: $\mathit{Noetherian}\ x \longleftrightarrow \nabla\ x = 0$
using *nabla-noether noether-nabla* **by** *blast*

lemma *nabla-preloeb-iff*: $\nabla\ x = 0 \longleftrightarrow \mathit{PreLoebian}\ x$
using *Noetherian-iff-PreLoebian nabla-noether noether-nabla* **by** *blast*

end

lemma *rel-nabla-prop*: $\mathit{rel\text{-}nabla}\ R = \{\} \longleftrightarrow (\forall P.\ P \subseteq \mathit{relfdia}\ R\ P \longrightarrow P = \{\})$
by (*metis bot.extremum-uniqueI rel-nabla-coinduct-var rel-nabla-unfold*)

lemma *fdia-rel-im1*: $\mathit{s2r}\ ((\mathit{converse}\ R)\ ``\ P) = \mathit{relfdia}\ R\ (\mathit{s2r}\ P)$
by (*auto simp*: *Id-on-def rel-antidomain-kleene-algebra.ads-d-def rel-ad-def rel-antidomain-kleene-algebra.fdia-Image-def converse-def*)

lemma *fdia-rel-im2*: $\mathit{s2r}\ ((\mathit{converse}\ R)\ ``\ (\mathit{r2s}\ (\mathit{rdom}\ P))) = \mathit{relfdia}\ R\ P$
by (*simp add*: *fdia-rel-im1 rsr*)

lemma *wf-nabla-aux*: $(P \subseteq (converse\ R)$ `` $P \longrightarrow P = \{\}) \longleftrightarrow (s2r\ P \subseteq relfdia\ R\ (s2r\ P) \longrightarrow s2r\ P = \{\})$
apply (*standard*, *metis Domain-Id-on Domain-mono Id-on-empty fdia-rel-im1*)
using *fdia-rel-im1* **by** *fastforce*

A relation is noeterian if its converse is wellfounded. Hence a relation is noetherian if and only if its divergence is empty. In the relational program semantics, noetherian programs terminate.

lemma *wf-nabla*: $wf\ (converse\ R) \longleftrightarrow rel\text{-}nabla\ R = \{\}$
by (*metis* (*no-types*, *lifting*) *fdia-rel-im2 rel-fdivka.nabla-unfold-eq rel-nabla-prop rel-nabla-unfold wfE-pf wfI-pf wf-nabla-aux*)

end

5.3.4 Verification Examples

theory *VC-KAD-wf-Examples*
imports *VC-KAD-wf*
begin

The example should be taken with a grain of salt. More work is needed to make the while rule cooperate with simplification.

lemma *euclid*:
$rel\text{-}nabla\ ($
$\lceil \lambda s::nat\ store.\ 0 < s\ ''y'' \rceil\ ;$
$((''z'' ::= (\lambda s.\ s\ ''y''))\ ;$
$(''y'' ::= (\lambda s.\ s\ ''x''\ mod\ s\ ''y''))\ ;$
$(''x'' ::= (\lambda s.\ s\ ''z''))))$
$= \{\}$
$\Longrightarrow$
$PRE\ (\lambda s::nat\ store.\ s\ ''x'' = x \wedge s\ ''y'' = y)$
$(WHILE\ (\lambda s.\ s\ ''y'' \neq 0)\ INV\ (\lambda s.\ gcd\ (s\ ''x'')\ (s\ ''y'') = gcd\ x\ y)$
DO
$(''z'' ::= (\lambda s.\ s\ ''y''));$
$(''y'' ::= (\lambda s.\ s\ ''x''\ mod\ s\ ''y''));$
$(''x'' ::= (\lambda s.\ s\ ''z''))$
$OD)$
$POST\ (\lambda s.\ s\ ''x'' = gcd\ x\ y)$
apply (*subst rel-fdivka.fbox-arden-whilei*[*symmetric*], *simp-all*)
using *gcd-red-nat gr0I* **by** *force*

The termination assumption is now explicit in the verification proof. Here it is left untouched. Means beyond these components are required for discharging it.

end

5.4 Two Extensions

5.4.1 KAD Component with Trace Semantics

theory *Path-Model-Example*
  imports *VC-KAD HOL−Eisbach.Eisbach*
begin

This component supports the verification of simple while programs in a partial correctness setting based on a program trace semantics.

Program traces are modelled as non-empty paths or state sequences. The non-empty path model of Kleene algebra is taken from the AFP entry for Kleene algebra. Here we show that sets of paths form antidomain Kleene Algebras.

definition *pp-a* :: *'a ppath set* ⇒ *'a ppath set* **where**
  pp-a X = {(*Node u*) |*u*. ¬ (∃ *v* ∈ *X*. *u* = *pp-first v*)}

interpretation *ppath-aka*: *antidomain-kleene-algebra pp-a* (∪) *pp-prod pp-one* {} (⊆) (⊂) *pp-star*
  apply *standard*
  apply (*clarsimp simp add*: *pp-prod-def pp-a-def*)
  apply (*simp add*: *pp-prod-def pp-a-def*, *safe*, *metis pp-first.simps*(*1*) *pp-first-pp-fusion*)
  by (*auto simp*: *pp-a-def pp-one-def*)

A verification component can then be built with little effort, by and large reusing parts of the relational components that are generic with respect to the store.

definition *pp-gets* :: *string* ⇒ (*'a store* ⇒ *'a*) ⇒ *'a store ppath set* (‹- ::= -› [*70*, *65*] *61*) **where**
  v ::= *e* = {*Cons s* (*Node* (*s* (*v* := *e s*))) | *s*. *True*}

definition *p2pp* :: *'a pred* ⇒ *'a ppath set* **where**
  p2pp P = {*Node s* |*s*. *P s*}

lemma *pp-a-neg* [*simp*]: *pp-a* (*p2pp Q*) = *p2pp* (−*Q*)
  by (*force simp add*: *pp-a-def p2pp-def*)

lemma *ppath-assign* [*simp*]: *ppath-aka.fbox* (*v* ::= *e*) (*p2pp Q*) = *p2pp* (λ*s*. *Q* (*s*(*v* := *e s*)))
  by (*force simp*: *ppath-aka.fbox-def pp-a-def p2pp-def pp-prod-def pp-gets-def*)

no-notation *spec-sugar* (‹*PRE* - - *POST* -› [*64*,*64*,*64*] *63*)
  and *relcomp* (**infixl** ‹;› *70*)
  and *cond-sugar* (‹*IF* - *THEN* - *ELSE* - *FI*› [*64*,*64*,*64*] *63*)
  and *whilei-sugar* (‹*WHILE* - *INV* - *DO* - *OD*› [*64*,*64*,*64*] *63*)
  and *gets* (‹- ::= -› [*70*, *65*] *61*)
  and *rel-antidomain-kleene-algebra.fbox* (‹*wp*›)
  and *rel-antidomain-kleene-algebra.ads-d* (‹*rdom*›)

and *p2r* (‹⌈-⌉›)

notation *ppath-aka.fbox* (‹*wp*›)
and *ppath-aka.ads-d* (‹*rdom*›)
and *p2pp* (‹⌈-⌉›)
and *pp-prod* (**infixl** ‹;› *70*)

abbreviation *spec-sugar* :: *'a pred* ⇒ *'a ppath set* ⇒ *'a pred* ⇒ *bool* (‹*PRE* - - *POST* -› [*64*,*64*,*64*] *63*) **where**
PRE P X POST Q ≡ *rdom* ⌈*P*⌉ ⊆ *wp X* ⌈*Q*⌉

abbreviation *cond-sugar* :: *'a pred* ⇒ *'a ppath set* ⇒ *'a ppath set* ⇒ *'a ppath set* (‹*IF* - *THEN* - *ELSE* - *FI*› [*64*,*64*,*64*] *63*) **where**
IF P THEN X ELSE Y FI ≡ *ppath-aka.cond* ⌈*P*⌉ *X Y*

abbreviation *whilei-sugar* :: *'a pred* ⇒ *'a pred* ⇒ *'a ppath set* ⇒ *'a ppath set* (‹*WHILE* - *INV* - *DO* - *OD*› [*64*,*64*,*64*] *63*) **where**
WHILE P INV I DO X OD ≡ *ppath-aka.whilei* ⌈*P*⌉ ⌈*I*⌉ *X*

lemma [*simp*]: *p2pp P* ∪ *p2pp Q* = *p2pp* (*P* ⊔ *Q*)
by (*force simp*: *p2pp-def*)

lemma [*simp*]: *p2pp P*; *p2pp Q* = *p2pp* (*P* ⊓ *Q*)
by (*force simp*: *p2pp-def pp-prod-def*)

lemma [*intro*!]: *P* ≤ *Q* ⟹ ⌈*P*⌉ ⊆ ⌈*Q*⌉
by (*force simp*: *p2pp-def*)

lemma [*simp*]: *rdom* ⌈*P*⌉ = ⌈*P*⌉
by (*simp add*: *ppath-aka.addual.ars-r-def*)

lemma *euclid*:
PRE (λ*s*::*nat store*. *s* ''*x*'' = *x* ∧ *s* ''*y*'' = *y*)
(*WHILE* (λ*s*. *s* ''*y*'' ≠ *0*) *INV* (λ*s*. *gcd* (*s* ''*x*'') (*s* ''*y*'') = *gcd x y*)
DO
(''*z*'' ::= (λ*s*. *s* ''*y*''));
(''*y*'' ::= (λ*s*. *s* ''*x*'' *mod s* ''*y*''));
(''*x*'' ::= (λ*s*. *s* ''*z*''))
OD)
POST (λ*s*. *s* ''*x*'' = *gcd x y*)
by (*rule ppath-aka.fbox-whilei*, *simp-all*, *auto simp*: *p2pp-def rel-ad-def gcd-non-0-nat*)

lemma *euclid-diff*:
PRE (λ*s*::*nat store*. *s* ''*x*'' = *x* ∧ *s* ''*y*'' = *y* ∧ *x* > *0* ∧ *y* > *0*)
(*WHILE* (λ*s*. *s* ''*x*''≠ *s* ''*y*'') *INV* (λ*s*. *gcd* (*s* ''*x*'') (*s* ''*y*'') = *gcd x y*)
DO
(*IF* (λ*s*. *s* ''*x*'' > *s* ''*y*'')
THEN (''*x*'' ::= (λ*s*. *s* ''*x*'' − *s* ''*y*''))
ELSE (''*y*'' ::= (λ*s*. *s* ''*y*'' − *s* ''*x*''))

FI)
OD)
POST ($\lambda s.\ s\ ''x'' = gcd\ x\ y$)
apply (*rule ppath-aka.fbox-whilei*, *simp-all*)
apply (*simp-all add*: *p2pp-def*)
apply *auto*[*2*]
by (*safe*, *metis gcd.commute gcd-diff1-nat le-cases nat-less-le*)

lemma *varible-swap*:
PRE ($\lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b$)
(''z'' ::= ($\lambda s.\ s\ ''x''$));
(''x'' ::= ($\lambda s.\ s\ ''y''$));
(''y'' ::= ($\lambda s.\ s\ ''z''$))
POST ($\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a$)
by *auto*

lemma *maximum*:
PRE (λs:: *nat store*. *True*)
(*IF* ($\lambda s.\ s\ ''x'' \geq s\ ''y''$)
THEN (''z'' ::= ($\lambda s.\ s\ ''x''$))
ELSE (''z'' ::= ($\lambda s.\ s\ ''y''$))
FI)
POST ($\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y'')$)
by *auto*

lemma *integer-division*:
PRE (λs::*nat store*. $x \geq 0$)
(''q'' ::= ($\lambda s.\ 0$));
(''r'' ::= ($\lambda s.\ x$));
(*WHILE* ($\lambda s.\ y \leq s\ ''r''$) *INV* ($\lambda s.\ x = s\ ''q'' * y + s\ ''r'' \wedge s\ ''r'' \geq 0$)
DO
(''q'' ::= ($\lambda s.\ s\ ''q'' + 1$));
(''r'' ::= ($\lambda s.\ s\ ''r'' - y$))
OD)
POST ($\lambda s.\ x = s\ ''q'' * y + s\ ''r'' \wedge s\ ''r'' \geq 0 \wedge s\ ''r'' < y$)
by (*rule ppath-aka.fbox-whilei-break*, *auto*)

We now reconsider these examples with an Eisbach tactic.

named-theorems *ht*

declare *ppath-aka.fbox-whilei* [*ht*]
ppath-aka.fbox-seq-var [*ht*]
subset-refl[*ht*]

method *hoare* = (*rule ht*; *hoare?*)

lemma *euclid2*:
PRE (λs::*nat store*. $s\ ''x'' = x \wedge s\ ''y'' = y$)
(*WHILE* ($\lambda s.\ s\ ''y'' \neq 0$) *INV* ($\lambda s.\ gcd\ (s\ ''x'')\ (s\ ''y'') = gcd\ x\ y$)

DO
(*''z''* ::= (*λs. s ''y''*));
(*''y''* ::= (*λs. s ''x'' mod s ''y''*));
(*''x''* ::= (*λs. s ''z''*))
OD)
POST (*λs. s ''x'' = gcd x y*)
apply *hoare*
using *gcd-red-nat* **by** *auto*

lemma *euclid-diff2*:
PRE (*λs::nat store. s ''x'' = x ∧ s ''y'' = y ∧ x > 0 ∧ y > 0*)
(*WHILE* (*λs. s ''x''≠ s ''y''*) *INV* (*λs. gcd* (*s ''x''*) (*s ''y''*) = *gcd x y*)
DO
(*IF* (*λs. s ''x'' > s ''y''*)
THEN (*''x''* ::= (*λs. s ''x'' − s ''y''*))
ELSE (*''y''* ::= (*λs. s ''y'' − s ''x''*))
FI)
OD)
POST (*λs. s ''x'' = gcd x y*)
by (*hoare*; *clarsimp*; *metis gcd.commute gcd-diff1-nat le-cases nat-less-le*)

lemma *varible-swap2*:
PRE (*λs. s ''x'' = a ∧ s ''y'' = b*)
(*''z''* ::= (*λs. s ''x''*));
(*''x''* ::= (*λs. s ''y''*));
(*''y''* ::= (*λs. s ''z''*))
POST (*λs. s ''x'' = b ∧ s ''y'' = a*)
by *clarsimp*

lemma *maximum2*:
PRE (*λs:: nat store. True*)
(*IF* (*λs. s ''x'' ≥ s ''y''*)
THEN (*''z''* ::= (*λs. s ''x''*))
ELSE (*''z''* ::= (*λs. s ''y''*))
FI)
POST (*λs. s ''z'' = max* (*s ''x''*) (*s ''y''*))
by *auto*

lemma *integer-division2*:
PRE (*λs::nat store. x ≥ 0*)
(*''q''* ::= (*λs. 0*));
(*''r''* ::= (*λs. x*));
(*WHILE* (*λs. y ≤ s ''r''*) *INV* (*λs. x = s ''q'' ∗ y + s ''r'' ∧ s ''r'' ≥ 0*)
DO
(*''q''* ::= (*λs. s ''q'' + 1*));
(*''r''* ::= (*λs. s ''r'' − y*))
OD)
POST (*λs. x = s ''q'' ∗ y + s ''r'' ∧ s ''r'' ≥ 0 ∧ s ''r'' < y*)
by *hoare auto*

```
lemma my-power2:
  PRE (λs::nat store. True)
   (''i'' ::= (λs. 0));
   (''y'' ::= (λs. 1));
   (WHILE (λs. s ''i'' < n) INV (λs. s ''y'' = x ^ (s ''i'') ∧ s ''i'' ≤ n)
     DO
       (''y'' ::= (λs. (s ''y'') * x));
       (''i'' ::= (λs. s ''i'' + 1))
     OD)
   POST (λs. s ''y'' = x ^ n)
  by hoare auto

lemma imp-reverse2:
  PRE (λs:: 'a list store. s ''x'' = X)
   (''y'' ::= (λs. []));
   (WHILE (λs. s ''x'' ≠ []) INV (λs. rev (s ''x'') @ s ''y'' = rev X)
    DO
     (''y'' ::= (λs. hd (s ''x'') # s ''y''));
     (''x'' ::= (λs. tl (s ''x'')))
    OD)
   POST (λs. s ''y''= rev X )
  apply hoare
  apply auto
  apply (metis append.simps append-assoc hd-Cons-tl rev.simps(2))
done

end
```

5.4.2 KAD Component for Pointer Programs

```
theory Pointer-Examples
  imports VC-KAD-Examples2 HOL−Hoare.Heap

begin
```

This component supports the verification of simple while programs with pointers in a partial correctness setting.

All we do here is integrating Nipkow's implementation of pointers and heaps.

```
type-synonym 'a state = string ⇒ ('a ref + ('a ⇒ 'a ref))

lemma list-reversal:
  PRE (λs :: 'a state. List (projr (s ''h'')) (projl (s ''p'')) Ps
        ∧ List (projr (s ''h'')) (projl (s ''q'')) Qs
        ∧ set Ps ∩ set Qs = {})
    (WHILE (λs. projl (s ''p'') ≠ Null)
     INV (λs. ∃ps qs. List (projr (s ''h'')) (projl (s ''p'')) ps
              ∧ List (projr (s ''h'')) (projl (s ''q'')) qs
```

$\wedge$ *set ps* $\cap$ *set qs* = {} $\wedge$ *rev ps* @ *qs* = *rev Ps* @ *Qs*)
DO
(''r'' ::= (λ*s*. *s* ''p''));
(''p'' ::= (λ*s*. *Inl* (*projr* (*s* ''h'') (*addr* (*projl* (*s* ''p''))))));
(''h'' ::= (λ*s*. *Inr* ((*projr* (*s* ''h''))(*addr* (*projl* (*s* ''r'')) := *projl* (*s* ''q'')))));
(''q'' ::= (λ*s*. *s* ''r''))
OD)
POST (λ*s*. *List* (*projr* (*s* ''h'')) (*projl* (*s* ''q'')) (*rev Ps* @ *Qs*))
apply *hoare*
apply *auto*[*2*]
by (*clarsimp*, *fastforce intro*: *notin-List-update*[*THEN iffD2*])

end

6 Bringing KAT Components into Scope of KAD

theory *KAD-is-KAT*
imports *KAD.Antidomain-Semiring*
KAT-and-DRA.KAT
AVC-KAD/VC-KAD
AVC-KAT/VC-KAT

begin

context *antidomain-kleene-algebra*
begin

Every Kleene algebra with domain is a Kleene algebra with tests. This fact should eventually move into the AFP KAD entry.

sublocale *kat* (+) ($\cdot$) *1 0* ($\leq$) (<) *star antidomain-op*
apply *standard*
apply *simp*
using *a-d-mult-closure am-d-def* **apply** *auto*[*1*]
using *dpdz.dom-weakly-local* **apply** *auto*[*1*]
using *a-d-add-closure a-de-morgan* **by** *presburger*

The next statement links the wp operator with the Hoare triple.

lemma *H-kat-to-kad*: $H\ p\ x\ q \longleftrightarrow d\ p \leq |x]\ (d\ q)$
using *H-def addual.ars-r-def fbox-demodalisation3* **by** *auto*

end

lemma *H-eq*: $P \subseteq Id \Longrightarrow Q \subseteq Id \Longrightarrow$ *rel-kat.H P X Q* = *rel-antidomain-kleene-algebra.H P X Q*
apply (*simp add*: *rel-kat.H-def rel-antidomain-kleene-algebra.H-def*)
apply (*subgoal-tac rel-antidomain-kleene-algebra.t P* = *Id* $\cap$ *P*)
apply (*subgoal-tac rel-antidomain-kleene-algebra.t Q* = *Id* $\cap$ *Q*)
apply *simp*

apply (*auto simp*: *rel-ad-def*)
done

no-notation *VC-KAD.spec-sugar* (‹*PRE* - - *POST* -› [*64*,*64*,*64*] *63*)
and *VC-KAD.cond-sugar* (‹*IF* - *THEN* - *ELSE* - *FI*› [*64*,*64*,*64*] *63*)
and *VC-KAD.gets* (‹- ::= -› [*70*, *65*] *61*)

Next we provide some syntactic sugar.

lemma *H-from-kat*: *PRE p x POST q* = ($\lceil p \rceil \leq$ (*rel-antidomain-kleene-algebra.fbox x*) $\lceil q \rceil$)
apply (*subst H-eq*)
apply (*clarsimp simp add*: *p2r-def*)
apply (*clarsimp simp add*: *p2r-def*)
apply (*subst rel-antidomain-kleene-algebra.H-kat-to-kad*)
apply (*subgoal-tac rel-antidomain-kleene-algebra.ads-d* $\lceil p \rceil = \lceil p \rceil$)
apply (*subgoal-tac rel-antidomain-kleene-algebra.ads-d* $\lceil q \rceil = \lceil q \rceil$)
apply *simp*
apply (*auto simp*: *rel-antidomain-kleene-algebra.ads-d-def rel-ad-def p2r-def*)
done

lemma *cond-iff*: *rel-kat.ifthenelse* $\lceil P \rceil$ *X Y* = *rel-antidomain-kleene-algebra.cond* $\lceil P \rceil$ *X Y*
by (*auto simp*: *rel-kat.ifthenelse-def rel-antidomain-kleene-algebra.cond-def*)

lemma *gets-iff*: *v* ::= *e* = *VC-KAD.gets v e*
by (*auto simp*: *VC-KAT.gets-def VC-KAD.gets-def*)

Finally we present two examples to test the integration.

lemma *maximum*:
PRE (λs:: *nat store*. *True*)
(*IF* (λs. *s* ''*x*'' $\geq$ *s* ''*y*'')
THEN (''*z*'' ::= (λs. *s* ''*x*''))
ELSE (''*z*'' ::= (λs. *s* ''*y*''))
FI)
POST (λs. *s* ''*z*'' = *max* (*s* ''*x*'') (*s* ''*y*''))
by (*simp only*: *sH-cond-iff H-assign-iff*, *auto*)

lemma *maximum2*:
PRE (λs:: *nat store*. *True*)
(*IF* (λs. *s* ''*x*'' $\geq$ *s* ''*y*'')
THEN (''*z*'' ::= (λs. *s* ''*x*''))
ELSE (''*z*'' ::= (λs. *s* ''*y*''))
FI)
POST (λs. *s* ''*z*'' = *max* (*s* ''*x*'') (*s* ''*y*''))
apply (*subst H-from-kat*)
apply (*subst cond-iff*)
apply (*subst gets-iff*)
apply (*subst gets-iff*)
by *auto*

end

7 Component for Recursive Programs

theory *Domain-Quantale*
  imports *KAD.Modal-Kleene-Algebra*

begin

This component supports the verification and step-wise refinement of recursive programs in a partial correctness setting.

notation
  times (**infixl** ‹·› *70*) **and**
  bot (‹⊥›) **and**
  top (‹⊤›) **and**
  inf (**infixl** ‹⊓› *65*) **and**
  sup (**infixl** ‹⊔› *65*)

7.1 Lattice-Ordered Monoids with Domain

class *bd-lattice-ordered-monoid* = *bounded-lattice* + *distrib-lattice* + *monoid-mult* +
  assumes *left-distrib*: $x \cdot (y \sqcup z) = x \cdot y \sqcup x \cdot z$
  and *right-distrib*: $(x \sqcup y) \cdot z = x \cdot z \sqcup y \cdot z$
  and *bot-annil* [*simp*]: $\bot \cdot x = \bot$
  and *bot-annir* [*simp*]: $x \cdot \bot = \bot$

begin

sublocale *semiring-one-zero* (⊔) (·) *1 bot*
  by (*standard*, *auto simp*: *sup.assoc sup.commute sup-left-commute left-distrib right-distrib sup-absorb1*)

sublocale *dioid-one-zero* (⊔) (·) *1 bot* (≤) (<)
  by (*standard*, *simp add*: *le-iff-sup*, *auto*)

end

no-notation *ads-d* (‹*d*›)
  and *ars-r* (‹*r*›)
  and *antirange-op* (‹*ar* -› [*999*] *1000*)

class *domain-bdlo-monoid* = *bd-lattice-ordered-monoid* +
  assumes *rdv*: $(z \sqcap x \cdot top) \cdot y = z \cdot y \sqcap x \cdot top$

begin

definition $d\ x = 1 \sqcap x \cdot \top$

sublocale *ds*: *domain-semiring* $(\sqcup)$ $(\cdot)$ *1* $\bot$ *d* $(\leq)$ $(<)$
proof *standard*
  fix x y
  show $x \sqcup d\ x \cdot x = d\ x \cdot x$
   by (*metis d-def inf-sup-absorb left-distrib mult-1-left mult-1-right rdv sup.absorb-iff1 sup.idem sup.left-commute top-greatest*)
  show $d\ (x \cdot d\ y) = d\ (x \cdot y)$
    by (*simp add*: *d-def inf-absorb2 rdv mult-assoc*)
  show $d\ x \sqcup 1 = 1$
    by (*simp add*: *d-def sup.commute*)
  show $d\ bot = bot$
    by (*simp add*: *d-def inf.absorb1 inf.commute*)
  show $d\ (x \sqcup y) = d\ x \sqcup d\ y$
    by (*simp add*: *d-def inf-sup-distrib1*)
qed

end

7.2 Boolean Monoids with Domain

class *boolean-monoid* = *boolean-algebra* + *monoid-mult* +
  assumes *left-distrib'*: $x \cdot (y \sqcup z) = x \cdot y \sqcup x \cdot z$
  and *right-distrib'*: $(x \sqcup y) \cdot z = x \cdot z \sqcup y \cdot z$
  and *bot-annil'* [*simp*]: $\bot \cdot x = \bot$
  and *bot-annir'* [*simp*]: $x \cdot \bot = \bot$

begin

subclass *bd-lattice-ordered-monoid*
  by (*standard*, *simp-all add*: *left-distrib' right-distrib'*)

lemma *inf-bot-iff-le*: $x \sqcap y = \bot \longleftrightarrow x \leq -y$
  by (*metis le-iff-inf inf-sup-distrib1 inf-top-right sup-bot.left-neutral sup-compl-top compl-inf-bot inf.assoc inf-bot-right*)

end

class *domain-boolean-monoid* = *boolean-monoid* +
  assumes *rdv'*: $(z \sqcap x \cdot \top) \cdot y = z \cdot y \sqcap x \cdot \top$

begin

sublocale *dblo*: *domain-bdlo-monoid* *1* $(\cdot)$ $(\sqcap)$ $(\leq)$ $(<)$ $(\sqcup)$ $\bot$ $\top$
  by (*standard*, *simp add*: *rdv'*)

definition $a\ x = 1 \sqcap -(dblo.d\ x)$

lemma *a-d-iff*: $a\ x = 1 \sqcap -(x \cdot \top)$

by (*clarsimp simp*: *a-def dblo.d-def inf-sup-distrib1*)

lemma *topr*: $-(x \cdot \top) \cdot \top = -(x \cdot \top)$
proof (*rule order.antisym*)
  show $-(x \cdot \top) \le -(x \cdot \top) \cdot \top$
    by (*metis mult-isol-var mult-oner order-refl top-greatest*)
  have $-(x \cdot \top) \sqcap (x \cdot \top) = \bot$
    by *simp*
  hence $(-(x \cdot \top) \sqcap (x \cdot \top)) \cdot \top = \bot$
    by *simp*
  hence $-(x \cdot \top) \cdot \top \sqcap (x \cdot \top) = \bot$
    by (*metis rdv'*)
  thus $-(x \cdot \top) \cdot \top \le -(x \cdot \top)$
    by (*simp add*: *inf-bot-iff-le*)
qed

lemma *dd-a*: *dblo.d* $x = a\ (a\ x)$
  by (*metis a-d-iff dblo.d-def double-compl inf-top.left-neutral mult-1-left rdv' topr*)

lemma *ad-a* [*simp*]: a (*dblo.d* x) $= a\ x$
  by (*simp add*: *a-def*)

lemma *da-a* [*simp*]: *dblo.d* $(a\ x) = a\ x$
  using *ad-a dd-a* **by** *auto*

lemma *a1* [*simp*]: $a\ x \cdot x = \bot$
proof −
  have $a\ x \cdot x \cdot \top = \bot$
    by (*metis a-d-iff inf-compl-bot mult-1-left rdv' topr*)
  then show *?thesis*
    by (*metis* (*no-types*) *dblo.d-def dblo.ds.domain-very-strict inf-bot-right*)
qed

lemma *a2* [*simp*]: $a\ (x \cdot y) \sqcup a\ (x \cdot a\ (a\ y)) = a\ (x \cdot a\ (a\ y))$
  by (*metis a-def dblo.ds.dsr2 dd-a sup.idem*)

lemma *a3* [*simp*]: $a\ (a\ x) \sqcup a\ x = 1$
  by (*metis a-def da-a inf.commute sup.commute sup-compl-top sup-inf-absorb sup-inf-distrib1*)

subclass *domain-bdlo-monoid* **..**

The next statement shows that every boolean monoid with domain is an antidomain semiring. In this setting the domain operation has been defined explicitly.

sublocale *ad*: *antidomain-semiring* $a\ (\sqcup)\ (\cdot)\ 1\ \bot\ (\le)\ (<)$
  rewrites *ad-eq*: *ad.ads-d* $x = d\ x$
proof −
  show *class.antidomain-semiring* $a\ (\sqcup)\ (\cdot)\ 1\ \bot\ (\le)\ (<)$

    by (*standard*, *simp-all*)
  then interpret *ad*: *antidomain-semiring a* (⊔) (·) *1* ⊥ (≤) (<) **.**
  show *ad.ads-d x* = *d x*
    by (*simp add*: *ad.ads-d-def dd-a*)
qed

end

7.3 Boolean Monoids with Range

class *range-boolean-monoid* = *boolean-monoid* +
  assumes *ldv'*: $y \cdot (z \sqcap \top \cdot x) = y \cdot z \sqcap \top \cdot x$

begin

definition $r\ x = 1 \sqcap \top \cdot x$

definition $ar\ x = 1 \sqcap -(r\ x)$

lemma *ar-r-iff*: $ar\ x = 1 \sqcap -(\top \cdot x)$
  by (*simp add*: *ar-def inf-sup-distrib1 r-def*)

lemma *topl*: $\top \cdot (-(\top \cdot x)) = -(\top \cdot x)$
proof (*rule order.antisym*)
  show $\top \cdot - (\top \cdot x) \leq - (\top \cdot x)$
    by (*metis bot-annir' compl-inf-bot inf-bot-iff-le ldv'*)
  show $- (\top \cdot x) \leq \top \cdot - (\top \cdot x)$
    by (*metis inf-le2 inf-top.right-neutral mult-1-left mult-isor*)
qed

lemma *r-ar*: $r\ x = ar\ (ar\ x)$
  by (*metis ar-r-iff double-compl inf.commute inf-top.right-neutral ldv' mult-1-right r-def topl*)

lemma *ar-ar* [*simp*]: $ar\ (r\ x) = ar\ x$
  by (*simp add*: *ar-def ldv' r-def*)

lemma *rar-ar* [*simp*]: $r\ (ar\ x) = ar\ x$
  using *r-ar ar-ar* **by** *force*

lemma *ar1* [*simp*]: $x \cdot ar\ x = \bot$
proof −
  have $\top \cdot x \cdot ar\ x = \bot$
    by (*metis ar-r-iff inf-compl-bot ldv' mult-oner topl*)
  then show *?thesis*
    by (*metis inf-bot-iff-le inf-le2 inf-top.right-neutral mult-1-left mult-isor mult-oner topl*)
qed

lemma *ars*: $r\ (r\ x \cdot y) = r\ (x \cdot y)$
by (*metis inf.commute inf-top.right-neutral ldv′ mult-oner mult-assoc r-def*)

lemma *ar2* [*simp*]: $ar\ (x \cdot y) \sqcup ar\ (ar\ (ar\ x) \cdot y) = ar\ (ar\ (ar\ x) \cdot y)$
by (*metis ar-def ars r-ar sup.idem*)

lemma *ar3* [*simp*]: $ar\ (ar\ x) \sqcup ar\ x = 1$
by (*metis ar-def rar-ar inf.commute sup.commute sup-compl-top sup-inf-absorb sup-inf-distrib1*)

sublocale *ar*: *antirange-semiring* $(\sqcup)$ $(\cdot)$ 1 $\bot$ *ar* $(\leq)$ $(<)$
rewrites *ar-eq*: $ar.ars\text{-}r\ x = r\ x$
proof −
show *class.antirange-semiring* $(\sqcup)$ $(\cdot)$ 1 $\bot$ *ar* $(\leq)$ $(<)$
by (*standard*, *simp-all*)
then interpret *ar*: *antirange-semiring* $(\sqcup)$ $(\cdot)$ 1 $\bot$ *ar* $(\leq)$ $(<)$ **.**
show $ar.ars\text{-}r\ x = r\ x$
by (*simp add*: *ar.ars-r-def r-ar*)
qed

end

7.4 Quantales

This part will eventually move into an AFP quantale entry.

class *quantale* = *complete-lattice* + *monoid-mult* +
assumes *Sup-distr*: $Sup\ X \cdot y = Sup\ \{z.\ \exists x \in X.\ z = x \cdot y\}$
and *Sup-distl*: $x \cdot Sup\ Y = Sup\ \{z.\ \exists y \in Y.\ z = x \cdot y\}$

begin

lemma *bot-annil″* [*simp*]: $\bot \cdot x = \bot$
using *Sup-distr*[**where** $X=\{\}$] **by** *auto*

lemma *bot-annirr″* [*simp*]: $x \cdot \bot = \bot$
using *Sup-distl*[**where** $Y=\{\}$] **by** *auto*

lemma *sup-distl*: $x \cdot (y \sqcup z) = x \cdot y \sqcup x \cdot z$
using *Sup-distl*[**where** $Y=\{y, z\}$] **by** (*fastforce intro*!: *Sup-eqI*)

lemma *sup-distr*: $(x \sqcup y) \cdot z = x \cdot z \sqcup y \cdot z$
using *Sup-distr*[**where** $X=\{x, y\}$] **by** (*fastforce intro*!: *Sup-eqI*)

sublocale *semiring-one-zero* $(\sqcup)$ $(\cdot)$ 1 $\bot$
by (*standard*, *auto simp*: *sup.assoc sup.commute sup-left-commute sup-distl sup-distr*)

sublocale *dioid-one-zero* $(\sqcup)$ $(\cdot)$ 1 $\bot$ $(\leq)$ $(<)$
by (*standard*, *simp add*: *le-iff-sup*, *auto*)

lemma *Sup-sup-pred*: $x \sqcup Sup\{y.\ P\ y\} = Sup\{y.\ y = x \vee P\ y\}$
  apply (*rule order.antisym*)
  apply (*simp add*: *Collect-mono Sup-subset-mono Sup-upper*)
  using *Sup-least Sup-upper le-supI2* **by** *fastforce*

definition *star* :: $'a \Rightarrow 'a$ **where**
  $star\ x = (SUP\ i.\ x\ \hat{}\ i)$

lemma *star-def-var1*: $star\ x = Sup\{y.\ \exists\, i.\ y = x\ \hat{}\ i\}$
  by (*simp add*: *star-def full-SetCompr-eq*)

lemma *star-def-var2*: $star\ x = Sup\{x\ \hat{}\ i\ |i.\ True\}$
  by (*simp add*: *star-def full-SetCompr-eq*)

lemma *star-unfoldl'* [*simp*]: $1 \sqcup x \cdot (star\ x) = star\ x$
proof −
  have $1 \sqcup x \cdot (star\ x) = x\ \hat{}\ 0 \sqcup x \cdot Sup\{y.\ \exists\, i.\ y = x\ \hat{}\ i\}$
    by (*simp add*: *star-def-var1*)
  also have ... $= x\ \hat{}\ 0 \sqcup Sup\{y.\ \exists\, i.\ y = x\ \hat{}\ (i + 1)\}$
    by (*simp add*: *Sup-distl*, *metis*)
  also have ... $= Sup\{y.\ y = x\ \hat{}\ 0 \vee (\exists\, i.\ y = x\ \hat{}\ (i + 1))\}$
    using *Sup-sup-pred* **by** *simp*
  also have ... $= Sup\{y.\ \exists\, i.\ y = x\ \hat{}\ i\}$
    by (*metis Suc-eq-plus1 power.power.power-Suc power.power-eq-if*)
  finally show *?thesis*
    by (*simp add*: *star-def-var1*)
qed

lemma *star-unfoldr'* [*simp*]: $1 \sqcup (star\ x) \cdot x = star\ x$
proof −
  have $1 \sqcup (star\ x) \cdot x = x\ \hat{}\ 0 \sqcup Sup\{y.\ \exists\, i.\ y = x\ \hat{}\ i\} \cdot x$
    by (*simp add*: *star-def-var1*)
  also have ... $= x\ \hat{}\ 0 \sqcup Sup\{y.\ \exists\, i.\ y = x\ \hat{}\ i \cdot x\}$
    by (*simp add*: *Sup-distr*, *metis*)
  also have ... $= x\ \hat{}\ 0 \sqcup Sup\{y.\ \exists\, i.\ y = x\ \hat{}\ (i + 1)\}$
    using *power-Suc2* **by** *simp*
  also have ... $= Sup\{y.\ y = x\ \hat{}\ 0 \vee (\exists\, i.\ y = x\ \hat{}\ (i + 1))\}$
    using *Sup-sup-pred* **by** *simp*
  also have ... $= Sup\{y.\ \exists\, i.\ y = x\ \hat{}\ i\}$
    by (*metis Suc-eq-plus1 power.power.power-Suc power.power-eq-if*)
  finally show *?thesis*
    by (*simp add*: *star-def-var1*)
qed

lemma (**in** *dioid-one-zero*) *power-inductl*: $z + x \cdot y \le y \Longrightarrow (x\ \hat{}\ n) \cdot z \le y$
proof (*induct n*)
  case *0* **show** *?case*
    using *0.prems* **by** *simp*

case *Suc* **thus** *?case*
  by (*simp*, *metis mult.assoc mult-isol order-trans*)
qed

lemma (**in** *dioid-one-zero*) *power-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot (x \mathbin{\char`\^} n) \leq y$
proof (*induct n*)
  case *0* **show** *?case*
    using *0.prems* **by** *auto*
  case *Suc*
  {
    fix *n*
    assume $z + y \cdot x \leq y \Longrightarrow z \cdot x \mathbin{\char`\^} n \leq y$
      and $z + y \cdot x \leq y$
    hence $z \cdot x \mathbin{\char`\^} n \leq y$
      by *simp*
    also have $z \cdot x \mathbin{\char`\^} \mathit{Suc}\ n = z \cdot x \cdot x \mathbin{\char`\^} n$
      by (*metis mult.assoc power-Suc*)
    moreover have ... $= (z \cdot x \mathbin{\char`\^} n) \cdot x$
      by (*metis mult.assoc power-commutes*)
    moreover have ... $\leq y \cdot x$
      by (*metis calculation*(*1*) *mult-isor*)
    moreover have ... $\leq y$
      using ‹$z + y \cdot x \leq y$› **by** *simp*
    ultimately have $z \cdot x \mathbin{\char`\^} \mathit{Suc}\ n \leq y$ **by** *simp*
  }
  thus *?case*
    by (*metis Suc*)
qed

lemma *star-inductl′*: $z \sqcup x \cdot y \leq y \Longrightarrow (\mathit{star}\ x) \cdot z \leq y$
proof −
  assume $z \sqcup x \cdot y \leq y$
  hence $\forall i.\ x \mathbin{\char`\^} i \cdot z \leq y$
    by (*simp add*: *power-inductl*)
  hence $\mathit{Sup}\{w.\ \exists i.\ w = x \mathbin{\char`\^} i \cdot z\} \leq y$
    by (*intro Sup-least*, *fast*)
  hence $\mathit{Sup}\{w.\ \exists i.\ w = x \mathbin{\char`\^} i\} \cdot z \leq y$
    using *Sup-distr Sup-le-iff* **by** *auto*
  thus $(\mathit{star}\ x) \cdot z \leq y$
    by (*simp add*: *star-def-var1*)
qed

lemma *star-inductr′*: $z \sqcup y \cdot x \leq y \Longrightarrow z \cdot (\mathit{star}\ x) \leq y$
proof −
  assume $z \sqcup y \cdot x \leq y$
  hence $\forall i.\ z \cdot x \mathbin{\char`\^} i \leq y$
    by (*simp add*: *power-inductr*)
  hence $\mathit{Sup}\{w.\ \exists i.\ w = z \cdot x \mathbin{\char`\^} i\} \leq y$
    by (*intro Sup-least*, *fast*)

hence $z \cdot Sup\{w.\ \exists\, i.\ w = x \mathbin{\widehat{}} i\} \leq y$
using *Sup-distl Sup-le-iff* **by** *auto*
thus $z \cdot (star\ x) \leq y$
by (*simp add*: *star-def-var1*)
qed

sublocale *ka*: *kleene-algebra* $(\sqcup)$ $(\cdot)$ 1 $\bot$ $(\leq)$ $(<)$ *star*
by *standard* (*simp-all add*: *star-inductl′ star-inductr′*)

end

Distributive quantales are often assumed to satisfy infinite distributivity laws between joins and meets, but finite ones suffice for our purposes.

class *distributive-quantale* = *quantale* + *distrib-lattice*

begin

subclass *bd-lattice-ordered-monoid*
by (*standard*, *simp-all add*: *distrib-left*)

lemma $(1 \sqcap x \cdot \top) \cdot x = x$

oops

end

7.5 Domain Quantales

class *domain-quantale* = *distributive-quantale* +
assumes *rdv′′*: $(z \sqcap x \cdot \top) \cdot y = z \cdot y \sqcap x \cdot \top$

begin

subclass *domain-bdlo-monoid*
by (*standard*, *simp add*: *rdv′′*)

end

class *range-quantale* = *distributive-quantale* +
assumes *ldv′′*: $y \cdot (z \sqcap \top \cdot x) = y \cdot z \sqcap \top \cdot x$

class *boolean-quantale* = *quantale* + *complete-boolean-algebra*

begin

subclass *boolean-monoid*
by (*standard*, *simp-all add*: *sup-distl*)

lemma $(1 \sqcap x \cdot \top) \cdot x = x$

oops

lemma $(1 \sqcap -(x \cdot \top)) \cdot x = \bot$

oops

end

7.6 Boolean Domain Quantales

class *domain-boolean-quantale* = *domain-quantale* + *boolean-quantale*

begin

subclass *domain-boolean-monoid*
  by (*standard*, *simp add*: *rdv''*)

lemma *fbox-eq*: $ad.fbox\ x\ q = Sup\{d\ p\ |p.\ d\ p \cdot x \leq x \cdot d\ q\}$
  apply (*rule Sup-eqI*[*symmetric*])
  apply *clarsimp*
  using *ad.fbox-demodalisation3 ad.fbox-simp* **apply** *auto*[*1*]
  apply *clarsimp*
  by (*metis ad.fbox-def ad.fbox-demodalisation3 ad.fbox-simp da-a eq-refl*)

lemma *fdia-eq*: $ad.fdia\ x\ p = Inf\{d\ q\ |q.\ x \cdot d\ p \leq d\ q \cdot x\}$
  apply (*rule Inf-eqI*[*symmetric*])
  apply *clarsimp*
  using *ds.fdemodalisation2* **apply** *auto*[*1*]
  apply *clarsimp*
  by (*metis ad.fd-eq-fdia ad.fdia-def da-a double-compl ds.fdemodalisation2 inf-bot-iff-le inf-compl-bot*)

The specification statement can be defined explicitly.

definition $R :: 'a \Rightarrow 'a \Rightarrow 'a$ **where**
  $R\ p\ q \equiv Sup\{x.\ (d\ p) \cdot x \leq x \cdot d\ q\}$

lemma $x \leq R\ p\ q \Longrightarrow d\ p \leq ad.fbox\ x\ (d\ q)$
proof (*simp add*: *R-def ad.kat-1-equiv ad.kat-2-equiv*)
  assume $x \leq Sup\{x.\ d\ p \cdot x \cdot a\ q = \bot\}$
  hence $d\ p \cdot x \cdot a\ q \leq d\ p \cdot Sup\{x.\ d\ p \cdot x \cdot a\ q = \bot\} \cdot a\ q$
    using *mult-double-iso* **by** *blast*
  also have ... $= Sup\{d\ p \cdot x \cdot a\ q\ |x.\ d\ p \cdot x \cdot a\ q = \bot\}$
    apply (*subst Sup-distl*)
    apply (*subst Sup-distr*)
    apply *clarsimp*
    by *metis*
  also have ... $= \bot$
    by (*auto simp*: *Sup-eqI*)

finally show *?thesis*
using *ad.fbox-demodalisation3 ad.kat-3 ad.kat-4 le-bot* **by** *blast*
qed

lemma $d\ p \leq ad.fbox\ x\ (d\ q) \Longrightarrow x \leq R\ p\ q$
apply (*simp add*: *R-def*)
apply (*rule Sup-upper*)
apply *simp*
using *ad.fbox-demodalisation3 ad.fbox-simp* **apply** *auto*[*1*]
done

end

7.7 Relational Model of Boolean Domain Quantales

interpretation *rel-dbq*: *domain-boolean-quantale*
‹(−)› *uminus* ‹(∩)› ‹(⊆)› ‹(⊂)› ‹(∪)› ‹{}› *UNIV* ‹⋂› ‹⋃› *Id* ‹(*O*)›
by *standard auto*

7.8 Modal Boolean Quantales

class *range-boolean-quantale* = *range-quantale* + *boolean-quantale*

begin

subclass *range-boolean-monoid*
by (*standard*, *simp add*: *ldv''*)

lemma *fbox-eq*: $ar.bbox\ x\ (r\ q) = Sup\{r\ p\ |p.\ x \cdot r\ p \leq (r\ q) \cdot x\}$
apply (*rule Sup-eqI*[*symmetric*])
apply *clarsimp*
using *ar.ardual.fbox-demodalisation3 ar.ardual.fbox-simp* **apply** *auto*[*1*]
apply *clarsimp*
by (*metis ar.ardual.fbox-def ar.ardual.fbox-demodalisation3 eq-refl rar-ar*)

lemma *fdia-eq*: $ar.bdia\ x\ (r\ p) = Inf\{r\ q\ |q.\ (r\ p) \cdot x \leq x \cdot r\ q\}$
apply (*rule Inf-eqI*[*symmetric*])
apply *clarsimp*
using *ar.ars-r-def ar.ardual.fdemodalisation22 ar.ardual.kat-3-equiv-opp ar.ardual.kat-4-equiv-opp*
apply *auto*[*1*]
apply *clarsimp*
using *ar.bdia-def ar.ardual.ds.fdemodalisation2 r-ar* **by** *fastforce*

end

class *modal-boolean-quantale* = *domain-boolean-quantale* + *range-boolean-quantale*
+
assumes *domrange'* [*simp*]: $d\ (r\ x) = r\ x$
and *rangedom'* [*simp*]: $r\ (d\ x) = d\ x$

begin

sublocale *mka*: *modal-kleene-algebra* $(\sqcup)$ $(\cdot)$ *1* $\bot$ $(\leq)$ $(<)$ *star a ar*
  by *standard* (*simp-all add*: *ar-eq ad-eq*)

end

no-notation *fbox* (‹(|-] -)› [*61*,*81*] *82*)
  and *antidomain-semiringl-class.fbox* (‹(|-] -)› [*61*,*81*] *82*)

notation *ad.fbox* (‹(|-] -)› [*61*,*81*] *82*)

7.9 Recursion Rule

lemma *recursion*: *mono* $(f :: 'a \Rightarrow 'a :: \textit{domain-boolean-quantale}) \Longrightarrow$
  $(\bigwedge x.\ d\ p \leq |x]\ d\ q \Longrightarrow d\ p \leq |f\ x]\ d\ q) \Longrightarrow\ d\ p \leq |\mathit{lfp}\ f]\ d\ q$
  apply (*erule lfp-ordinal-induct* [**where** *f*=*f*], *simp*)
  by (*auto simp*: *ad.addual.ardual.fbox-demodalisation3 Sup-distr Sup-distl intro*: *Sup-mono*)

We have already tested this rule in the context of test quantales [2], which is based on a formalisation of quantales that is currently not in the AFP. The two theories will be merged as soon as the quantale is available in the AFP.

end

References

[1] A. Armstrong, V. B. F. Gomes, and G. Struth. Kleene algebra with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2014.

[2] A. Armstrong, V. B. F. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2016.

[3] A. Armstrong, G. Struth, and T. Weber. Kleene algebra. *Archive of Formal Proofs*, 2013.

[4] J. Desharnais and G. Struth. Internal axioms for domain semirings. *Science of Computer Programming*, 76(3):181–203, 2011.

[5] V. B. F. Gomes, W. Guttman, P. Höfner, G. Struth, and T. Weber. Kleene algebra with domain. *Archive of Formal Proofs*, 2016.