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 **3**

2 Two Standalone Components **4**
2.1 Component Based on Kleene Algebra with Tests 4
2.1.1 KAT: Definition and Basic Properties 4
2.1.2 Propositional Hoare Logic 6
2.1.3 Soundness and Relation KAT 6
2.1.4 Embedding Predicates in Relations 7
2.1.5 Store and Assignment 8
2.1.6 Verification Example 8
2.1.7 Definition of Refinement KAT 8
2.1.8 Propositional Refinement Calculus 9
2.1.9 Soundness and Relation RKAT 9
2.1.10 Assignment Laws 9
2.1.11 Refinement Example 9
2.2 Component Based on Kleene Algebra with Domain 10
2.2.1 KAD: Definitions and Basic Properties 10

2.2.2 wp Calculus . . . 13
2.2.3 Soundness and Relation KAD . . . 14
2.2.4 Embedding Predicates in Relations . . . 14
2.2.5 Store and Assignment . . . 15
2.2.6 Verification Example . . . 15
2.2.7 Propositional Hoare Logic . . . 16
2.2.8 Definition of Refinement KAD . . . 16
2.2.9 Propositional Refinement Calculus . . . 16
2.2.10 Soundness and Relation RKAD . . . 17
2.2.11 Assignment Laws . . . 17
2.2.12 Refinement Example . . . 17

3 Isomorphisms Between Predicates, Sets and Relations 18
3.1 Isomorphism Between Sets and Relations . . . 18
3.2 Isomorphism Between Predicates and Sets . . . 19
3.3 Isomorphism Between Predicates and Relations . . . 20

4 Components Based on Kleene Algebra with Tests 22
4.1 Verification Component . . . 22
4.1.1 Definitions of Hoare Triple . . . 22
4.1.2 Syntax for Conditionals and Loops . . . 22
4.1.3 Propositional Hoare Logic . . . 23
4.1.4 Store and Assignment . . . 24
4.1.5 Simplified Hoare Rules . . . 24
4.1.6 Verification Examples . . . 26
4.1.7 Verification Examples with Automated VCG . . . 27
4.2 Refinement Component . . . 28
4.2.1 RKAT: Definition and Basic Properties . . . 28
4.2.2 Propositional Refinement Calculus . . . 28
4.2.3 Models of Refinement KAT . . . 29
4.2.4 Assignment Laws . . . 29
4.2.5 Simplified Refinement Laws . . . 30
4.2.6 Refinement Examples . . . 30

5 Components Based on Kleene Algebra with Domain 31
5.1 Verification Component for Backward Reasoning . . . 31
5.1.1 Additional Facts for KAD . . . 31
5.1.2 Syntax for Conditionals and Loops . . . 31
5.1.3 Basic Weakest (Liberal) Precondition Calculus . . . 31
5.1.4 Store and Assignment . . . 33
5.1.5 Simplifications . . . 33
5.1.6 Verification Examples . . . 34
5.1.7 Verification Examples with Automated VCG . . . 36
5.2 Verification Component for Forward Reasoning . . . 38

5.2.1 Basic Strongest Postcondition Calculus 38
5.2.2 Floyd's Assignment Rule 39
5.2.3 Verification Examples 40
5.3 Verification Component for Total Correctness 42
5.3.1 Relation Divergence Kleene Algebras 42
5.3.2 Meta-Equational Loop Rule 43
5.3.3 Noethericity and Absence of Divergence 44
5.3.4 Verification Examples 45
5.4 Two Extensions 45
5.4.1 KAD Component with Trace Semantics 45
5.4.2 KAD Component for Pointer Programs 50

6 Bringing KAT Components into Scope of KAD **50**

7 Component for Recursive Programs **52**
7.1 Lattice-Ordered Monoids with Domain 52
7.2 Boolean Monoids with Domain 53
7.3 Boolean Monoids with Range 54
7.4 Quantales 55
7.5 Domain Quantales 56
7.6 Boolean Domain Quantales 57
7.7 Relational Model of Boolean Domain Quantales 58
7.8 Modal Boolean Quantales 58
7.9 Recursion Rule 58

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 do-

main [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*
  ⟨*proof*⟩

subclass *order*
  ⟨*proof*⟩

lemma *mult-isol*: $x \leq y \Longrightarrow z \cdot x \leq z \cdot y$
⟨*proof*⟩

lemma *mult-isor*: $x \leq y \Longrightarrow x \cdot z \leq y \cdot z$
⟨*proof*⟩

lemma *add-iso*: $x \leq y \Longrightarrow x + z \leq y + z$
⟨*proof*⟩

lemma *add-lub*: $x + y \leq z \longleftrightarrow x \leq z \wedge y \leq z$
⟨*proof*⟩

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*⟩

end

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

begin

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

lemma *t-n* [*simp*]: $t\ (at\ x) = at\ x$
⟨*proof*⟩

lemma *t-comm*: $t\ x \cdot t\ y = t\ y \cdot t\ x$
⟨*proof*⟩

lemma *t-idem* [*simp*]: $t\ x \cdot t\ x = t\ x$
⟨*proof*⟩

lemma *t-mult-closed* [*simp*]: $t\ (t\ x \cdot t\ y) = t\ x \cdot t\ y$
⟨*proof*⟩

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*: $H\ p\ 1\ p$
⟨*proof*⟩

lemma *H-cons*: $t\ p \leq t\ p' \Longrightarrow t\ q' \leq t\ q \Longrightarrow H\ p'\ x\ q' \Longrightarrow H\ p\ x\ q$
⟨*proof*⟩

lemma *H-seq*: $H\ r\ y\ q \Longrightarrow H\ p\ x\ r \Longrightarrow H\ p\ (x \cdot y)\ q$
⟨*proof*⟩

lemma *H-cond*: $H\ (t\ p \cdot t\ r)\ x\ q \Longrightarrow H\ (t\ p \cdot at\ r)\ y\ q \Longrightarrow H\ p$ (*if r then x else y fi*) *q*
⟨*proof*⟩

lemma *H-loop*: $H\ (t\ p \cdot t\ r)\ x\ p \Longrightarrow H\ p$ (*while r do x od*) $(t\ p \cdot at\ r)$
⟨*proof*⟩

lemma *H-while-inv*: $t\ p \leq t\ i \Longrightarrow t\ i \cdot at\ r \leq t\ q \Longrightarrow H\ (t\ i \cdot t\ r)\ x\ i \Longrightarrow H\ p$ (*while r inv i do x od*) *q*
⟨*proof*⟩

end

2.1.3 Soundness and Relation KAT

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

interpretation *rel-d*: *dioid Id* {} (∪) (;) (⊆) (⊂)
⟨*proof*⟩

lemma (**in** *dioid*) *power-inductl*: $z + x \cdot y \leq y \Longrightarrow x\ \hat{}\ i \cdot z \leq y$

⟨*proof*⟩

lemma (**in** *dioid*) *power-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot x \hat{} i \leq y$
⟨*proof*⟩

lemma *power-is-relpow*: *rel-d.power X i* = $X \hat{\hat{}} i$
⟨*proof*⟩

lemma *rel-star-def*: $X\hat{}* = (\bigcup i.$ *rel-d.power X i*$)$
⟨*proof*⟩

lemma *rel-star-contl*: X ; $Y\hat{}* = (\bigcup i.$ X ; *rel-d.power Y i*$)$
⟨*proof*⟩

lemma *rel-star-contr*: $X\hat{}*$; $Y = (\bigcup i.$ (*rel-d.power X i*) ; $Y)$
⟨*proof*⟩

definition *rel-at* :: *'a rel* ⇒ *'a rel* **where**
rel-at X = *Id* ∩ − *X*

interpretation *rel-kat*: *kat Id* {} (∪) (;) (⊆) (⊂) *rtrancl rel-at*
⟨*proof*⟩

2.1.4 Embedding Predicates in Relations

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

abbreviation *p2r* :: *'a pred* ⇒ *'a rel* (‹⌈-⌉›) **where**
⌈*P*⌉ ≡ {(*s*,*s*) |*s*. *P s*}

lemma *t-p2r* [*simp*]: *rel-kat.t-op* ⌈*P*⌉ = ⌈*P*⌉
⟨*proof*⟩

lemma *p2r-neg-hom* [*simp*]: *rel-at* ⌈*P*⌉ = ⌈λ*s*. ¬ *P s*⌉
⟨*proof*⟩

lemma *p2r-conj-hom* [*simp*]: ⌈*P*⌉ ∩ ⌈*Q*⌉ = ⌈λ*s*. *P s* ∧ *Q s*⌉
⟨*proof*⟩

lemma *p2r-conj-hom-var* [*simp*]: ⌈*P*⌉ ; ⌈*Q*⌉ = ⌈λ*s*. *P s* ∧ *Q s*⌉
⟨*proof*⟩

lemma *p2r-disj-hom* [*simp*]: ⌈*P*⌉ ∪ ⌈*Q*⌉ = ⌈λ*s*. *P s* ∨ *Q s*⌉
⟨*proof*⟩

lemma *impl-prop* [*simp*]: ⌈*P*⌉ ⊆ ⌈*Q*⌉ ⟷ (∀ *s*. *P s* ⟶ *Q s*)
⟨*proof*⟩

2.1.5 Store and Assignment

type-synonym *'a store = string ⇒ 'a*

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: rel-kat.H ⌈λs. P (s (v := e s))⌉ (v ::= e) ⌈P⌉*
⟨*proof*⟩

lemma *H-assign-var: (∀ s. P s ⟶ Q (s (v := e s))) ⟹ rel-kat.H ⌈P⌉ (v ::= e) ⌈Q⌉*
⟨*proof*⟩

abbreviation *H-sugar :: 'a pred ⇒ 'a rel ⇒ 'a pred ⇒ bool (‹PRE - - POST -› [64,64,64] 63)* **where**
PRE P X POST Q ≡ rel-kat.H ⌈P⌉ X ⌈Q⌉

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-kat.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-kat.while-inv ⌈P⌉ ⌈I⌉ X

2.1.6 Verification Example

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)
⟨*proof*⟩

2.1.7 Definition of Refinement KAT

class *rkat = kat +*
fixes *R :: 'a ⇒ 'a ⇒ 'a*
assumes *R1: H p (R p q) q*
and *R2: H p x q ⟹ x ≤ R p q*

begin

2.1.8 Propositional Refinement Calculus

lemma *R-skip*: $1 \leq R\ p\ p$
⟨*proof*⟩

lemma *R-cons*: $t\ p \leq t\ p' \Longrightarrow t\ q' \leq t\ q \Longrightarrow R\ p'\ q' \leq R\ p\ q$
⟨*proof*⟩

lemma *R-seq*: $(R\ p\ r) \cdot (R\ r\ q) \leq R\ p\ q$
⟨*proof*⟩

lemma *R-cond*: *if* v *then* $(R\ (t\ v \cdot t\ p)\ q)$ *else* $(R\ (at\ v \cdot t\ p)\ q)$ *fi* $\leq R\ p\ q$
⟨*proof*⟩

lemma *R-loop*: *while* q *do* $(R\ (t\ p \cdot t\ q)\ p)$ *od* $\leq R\ p\ (t\ p \cdot at\ q)$
⟨*proof*⟩

end

2.1.9 Soundness and Relation RKAT

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

interpretation *rel-rkat*: *rkat Id* $\{\}$ $(\cup)$ $(;)$ $(\subseteq)$ $(\subset)$ *rtrancl rel-at rel-R*
⟨*proof*⟩

2.1.10 Assignment Laws

lemma *R-assign*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow (v ::= e) \subseteq rel\text{-}R\ \lceil P \rceil\ \lceil Q \rceil$
⟨*proof*⟩

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*⟩

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*⟩

2.1.11 Refinement Example

lemma *var-swap-ref1*:
$rel\text{-}R\ \lceil \lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
$\supseteq (''z'' ::= (\lambda s.\ s\ ''x''));\ rel\text{-}R\ \lceil \lambda s.\ s\ ''z'' = a \wedge s\ ''y'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
⟨*proof*⟩

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-R* $\lceil\lambda s.\ s$ ''z'' $= a \wedge s$ ''x'' $= b\rceil$ $\lceil\lambda s.\ s$ ''x'' $= b \wedge s$ ''y'' $= a\rceil$
⟨*proof*⟩

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'' $= a\rceil$
⟨*proof*⟩

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''))
⟨*proof*⟩

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*
⟨*proof*⟩

subclass *order*
⟨*proof*⟩

lemma *mult-isor*: $x \leq y \Longrightarrow x \cdot z \leq y \cdot z$
⟨*proof*⟩

lemma *mult-isol*: $x \leq y \Longrightarrow z \cdot x \leq z \cdot y$
⟨*proof*⟩

lemma *add-iso*: $x \leq y \Longrightarrow z + x \leq z + y$
⟨*proof*⟩

lemma *add-ub*: $x \leq x + y$
⟨*proof*⟩

lemma *add-lub*: $x + y \leq z \longleftrightarrow x \leq z \wedge y \leq z$
⟨*proof*⟩

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*⟩

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*: $ad\ x \cdot y \leq y$
⟨*proof*⟩

lemma *d1-a* [*simp*]: $d\ x \cdot x = x$
⟨*proof*⟩

lemma *a-mul-d* [*simp*]: $ad\ x \cdot d\ x = 0$

⟨*proof*⟩

lemma *a-d-closed* [*simp*]: $d\ (ad\ x) = ad\ x$
⟨*proof*⟩

lemma *a-idem* [*simp*]: $ad\ x \cdot ad\ x = ad\ x$
⟨*proof*⟩

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

lemma *d-wloc*: $x \cdot y = 0 \longleftrightarrow x \cdot d\ y = 0$
⟨*proof*⟩

lemma *gla-1*: $ad\ x \cdot y = 0 \Longrightarrow ad\ x \leq ad\ y$
⟨*proof*⟩

lemma *a2-eq* [*simp*]: $ad\ (x \cdot d\ y) = ad\ (x \cdot y)$
⟨*proof*⟩

lemma *a-supdist*: $ad\ (x + y) \leq ad\ x$
⟨*proof*⟩

lemma *a-antitone*: $x \leq y \Longrightarrow ad\ y \leq ad\ x$
⟨*proof*⟩

lemma *a-comm*: $ad\ x \cdot ad\ y = ad\ y \cdot ad\ x$
⟨*proof*⟩

lemma *a-closed* [*simp*]: $d\ (ad\ x \cdot ad\ y) = ad\ x \cdot ad\ y$
⟨*proof*⟩

lemma *a-exp* [*simp*]: $ad\ (ad\ x \cdot y) = d\ x + ad\ y$
⟨*proof*⟩

lemma *d1-sum-var*: $x + y \leq (d\ x + d\ y) \cdot (x + y)$
⟨*proof*⟩

lemma *a4*: $ad\ (x + y) = ad\ x \cdot ad\ y$
⟨*proof*⟩

lemma *kat-prop*: $d\ x \cdot y \leq y \cdot d\ z \longleftrightarrow d\ x \cdot y \cdot ad\ z = 0$
⟨*proof*⟩

lemma *shunt*: $ad\ x \leq ad\ y + ad\ z \longleftrightarrow ad\ x \cdot d\ y \leq ad\ z$
⟨*proof*⟩

2.2.2 wp Calculus

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

definition *while* :: $'a \Rightarrow 'a \Rightarrow 'a$ ($\langle$*while - do - od*$\rangle$ [*64*,*64*] *63*) **where**
while p do x od = (*d p* · *x*)* · *ad p*

definition *while-inv* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ ($\langle$*while - inv - do - od*$\rangle$ [*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* · *ad p*)

lemma *demod*: *d p* ≤ *wp x q* ⟷ *d p* · *x* ≤ *x* · *d q*
⟨*proof*⟩

lemma *wp-weaken*: *wp x p* ≤ *wp* (*x* · *ad q*) (*d p* · *ad q*)
⟨*proof*⟩

lemma *wp-seq* [*simp*]: *wp* (*x* · *y*) *q* = *wp x* (*wp y q*)
⟨*proof*⟩

lemma *wp-seq-var*: *p* ≤ *wp x r* ⟹ *r* ≤ *wp y q* ⟹ *p* ≤ *wp* (*x* · *y*) *q*
⟨*proof*⟩

lemma *wp-cond-var* [*simp*]: *wp* (*if p then x else y fi*) *q* = (*ad p* + *wp x q*) · (*d p* + *wp y q*)
⟨*proof*⟩

lemma *wp-cond-aux1* [*simp*]: *d p* · *wp* (*if p then x else y fi*) *q* = *d p* · *wp x q*
⟨*proof*⟩

lemma *wp-cond-aux2* [*simp*]: *ad p* · *wp* (*if p then x else y fi*) *q* = *ad p* · *wp y q*
⟨*proof*⟩

lemma *wp-cond* [*simp*]: *wp* (*if p then x else y fi*) *q* = (*d p* · *wp x q*) + (*ad p* · *wp y q*)
⟨*proof*⟩

lemma *wp-star-induct-var*: *d q* ≤ *wp x q* ⟹ *d q* ≤ *wp* (*x**) *q*
⟨*proof*⟩

lemma *wp-while*: *d p* · *d r* ≤ *wp x p* ⟹ *d p* ≤ *wp* (*while r do x od*) (*d p* · *ad r*)
⟨*proof*⟩

lemma *wp-while-inv*: *d p* ≤ *d i* ⟹ *d i* · *ad r* ≤ *d q* ⟹ *d i* · *d r* ≤ *wp x i* ⟹ *d p* ≤ *wp* (*while r inv i do x od*) *q*

⟨*proof*⟩

lemma *wp-while-inv-break*: $d\ p \leq wp\ y\ i \Longrightarrow d\ i \cdot ad\ r \leq d\ q \Longrightarrow d\ i \cdot d\ r \leq wp\ x\ i \Longrightarrow d\ p \leq wp\ (y \cdot (while\ r\ inv\ i\ do\ x\ od))\ q$
⟨*proof*⟩

end

2.2.3 Soundness and Relation KAD

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

interpretation *rel-d*: *dioid Id* {} (∪) (;) (⊆) (⊂)
⟨*proof*⟩

lemma (**in** *dioid*) *pow-inductl*: $z + x \cdot y \leq y \Longrightarrow x\ \hat{}\ i \cdot z \leq y$
⟨*proof*⟩

lemma (**in** *dioid*) *pow-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot x\ \hat{}\ i \leq y$
⟨*proof*⟩

lemma *power-is-relpow*: *rel-d.power X i* = $X\ \hat{}\hat{}\ i$
⟨*proof*⟩

lemma *rel-star-def*: $X\hat{}* = (\bigcup i.\ rel\text{-}d.power\ X\ i)$
⟨*proof*⟩

lemma *rel-star-contl*: $X\ ;\ Y\hat{}* = (\bigcup i.\ X\ ;\ rel\text{-}d.power\ Y\ i)$
⟨*proof*⟩

lemma *rel-star-contr*: $X\hat{}*\ ;\ Y = (\bigcup i.\ (rel\text{-}d.power\ X\ i)\ ;\ Y)$
⟨*proof*⟩

definition *rel-ad* :: *'a rel* ⇒ *'a rel* **where**
rel-ad R = {(*x*,*x*) | *x*. ¬ (∃ *y*. (*x*,*y*) ∈ *R*)}

interpretation *rel-aka*: *antidomain-kleene-algebra Id* {} (∪) (;) (⊆) (⊂) *rtrancl rel-ad*
⟨*proof*⟩

2.2.4 Embedding Predicates in Relations

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

abbreviation *p2r* :: *'a pred* ⇒ *'a rel* (‹⌈-⌉›) **where**
⌈*P*⌉ ≡ {(*s*,*s*) |*s*. *P s*}

lemma *d-p2r* [*simp*]: *rel-aka.dom-op* ⌈*P*⌉ = ⌈*P*⌉
⟨*proof*⟩

lemma *p2r-neg-hom* [*simp*]: *rel-ad* ⌈*P*⌉ = ⌈λ*s*. ¬*P s*⌉
⟨*proof*⟩

lemma *p2r-conj-hom* [*simp*]: ⌈*P*⌉ ∩ ⌈*Q*⌉ = ⌈λ*s*. *P s* ∧ *Q s*⌉
⟨*proof*⟩

lemma *p2r-conj-hom-var* [*simp*]: ⌈*P*⌉ ; ⌈*Q*⌉ = ⌈λ*s*. *P s* ∧ *Q s*⌉
⟨*proof*⟩

lemma *p2r-disj-hom* [*simp*]: ⌈*P*⌉ ∪ ⌈*Q*⌉ = ⌈λ*s*. *P s* ∨ *Q s*⌉
⟨*proof*⟩

2.2.5 Store and Assignment

type-synonym *'a store* = *string* ⇒ *'a*

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

lemma *wp-assign* [*simp*]: *rel-aka.wp* (*v* ::= *e*) ⌈*Q*⌉ = ⌈λ*s*. *Q* (*s* (*v* := *e s*))⌉
⟨*proof*⟩

abbreviation *spec-sugar* :: *'a pred* ⇒ *'a rel* ⇒ *'a pred* ⇒ *bool* (‹*PRE* - - *POST* -› [*64*,*64*,*64*] *63*) **where**
PRE P X POST Q ≡ *rel-aka.dom-op* ⌈*P*⌉ ⊆ *rel-aka.wp X* ⌈*Q*⌉

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*)
⟨*proof*⟩

context *antidomain-kleene-algebra*
begin

2.2.7 Propositional Hoare Logic

definition $H :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$H\ p\ x\ q \longleftrightarrow d\ p \leq wp\ x\ q$

lemma *H-skip*: $H\ p\ 1\ p$
⟨*proof*⟩

lemma *H-cons*: $d\ p \leq d\ p' \Longrightarrow d\ q' \leq d\ q \Longrightarrow H\ p'\ x\ q' \Longrightarrow H\ p\ x\ q$
⟨*proof*⟩

lemma *H-seq*: $H\ p\ x\ r \Longrightarrow H\ r\ y\ q \Longrightarrow H\ p\ (x \cdot y)\ q$
⟨*proof*⟩

lemma *H-cond*: $H\ (d\ p \cdot d\ r)\ x\ q \Longrightarrow H\ (d\ p \cdot ad\ r)\ y\ q \Longrightarrow H\ p\ (\mathit{if}\ r\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi})\ q$
⟨*proof*⟩

lemma *H-loop*: $H\ (d\ p \cdot d\ r)\ x\ p \Longrightarrow H\ p\ (\mathit{while}\ r\ \mathit{do}\ x\ \mathit{od})\ (d\ p \cdot ad\ r)$
⟨*proof*⟩

lemma *H-while-inv*: $d\ p \leq d\ i \Longrightarrow d\ i \cdot ad\ r \leq d\ q \Longrightarrow H\ (d\ i \cdot d\ r)\ x\ i \Longrightarrow H\ p\ (\mathit{while}\ r\ \mathit{inv}\ i\ \mathit{do}\ x\ \mathit{od})\ q$
⟨*proof*⟩

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*: $H\ p\ x\ q \longleftrightarrow x \leq R\ p\ q$
⟨*proof*⟩

lemma *wp-R1*: $d\ p \leq wp\ (R\ p\ q)\ q$
⟨*proof*⟩

lemma *wp-R2*: $x \leq R\ (wp\ x\ q)\ q$
⟨*proof*⟩

lemma *wp-R3*: $d\ p \leq wp\ x\ q \Longrightarrow x \leq R\ p\ q$
⟨*proof*⟩

lemma *H-R1*: $H\ p\ (R\ p\ q)\ q$

⟨*proof*⟩

lemma *H-R2*: $H\ p\ x\ q \Longrightarrow x \leq R\ p\ q$
⟨*proof*⟩

lemma *R-skip*: $1 \leq R\ p\ p$
⟨*proof*⟩

lemma *R-cons*: $d\ p \leq d\ p' \Longrightarrow d\ q' \leq d\ q \Longrightarrow R\ p'\ q' \leq R\ p\ q$
⟨*proof*⟩

lemma *R-seq*: $(R\ p\ r) \cdot (R\ r\ q) \leq R\ p\ q$
⟨*proof*⟩

lemma *R-cond*: *if* v *then* $(R\ (d\ v \cdot d\ p)\ q)$ *else* $(R\ (ad\ v \cdot d\ p)\ q)$ *fi* $\leq R\ p\ q$
⟨*proof*⟩

lemma *R-loop*: *while* q *do* $(R\ (d\ p \cdot d\ q)\ p)$ *od* $\leq R\ p\ (d\ p \cdot ad\ q)$
⟨*proof*⟩

end

2.2.10 Soundness and Relation RKAD

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

interpretation *rel-rkad*: *rkad Id* {} (∪) (;) (⊆) (⊂) *rtrancl rel-ad rel-R*
⟨*proof*⟩

2.2.11 Assignment Laws

lemma *R-assign*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow (v ::= e) \subseteq rel\text{-}R\ \lceil P \rceil\ \lceil Q \rceil$
⟨*proof*⟩

lemma *H-assign-var*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow rel\text{-}aka.H\ \lceil P \rceil\ (v ::= e)\ \lceil Q \rceil$
⟨*proof*⟩

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*⟩

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*⟩

2.2.12 Refinement Example

lemma *var-swap-ref1*:

rel-R ⌈λ*s*. *s* ″*x*″ = *a* ∧ *s* ″*y*″ = *b*⌉ ⌈λ*s*. *s* ″*x*″ = *b* ∧ *s* ″*y*″ = *a*⌉
⊇ (″*z*″ ::= (λ*s*. *s* ″*x*″)); *rel-R* ⌈λ*s*. *s* ″*z*″ = *a* ∧ *s* ″*y*″ = *b*⌉ ⌈λ*s*. *s* ″*x*″ = *b* ∧ *s* ″*y*″ = *a*⌉
⟨*proof*⟩

lemma *var-swap-ref2*:
rel-R ⌈λ*s*. *s* ″*z*″ = *a* ∧ *s* ″*y*″ = *b*⌉ ⌈λ*s*. *s* ″*x*″ = *b* ∧ *s* ″*y*″ = *a*⌉
⊇ (″*x*″ ::= (λ*s*. *s* ″*y*″)); *rel-R* ⌈λ*s*. *s* ″*z*″ = *a* ∧ *s* ″*x*″ = *b*⌉ ⌈λ*s*. *s* ″*x*″ = *b* ∧ *s* ″*y*″ = *a*⌉
⟨*proof*⟩

lemma *var-swap-ref3*:
rel-R ⌈λ*s*. *s* ″*z*″ = *a* ∧ *s* ″*x*″ = *b*⌉ ⌈λ*s*. *s* ″*x*″ = *b* ∧ *s* ″*y*″ = *a*⌉
⊇ (″*y*″ ::= (λ*s*. *s* ″*z*″)); *rel-R* ⌈λ*s*. *s* ″*x*″ = *b* ∧ *s* ″*y*″ = *a*⌉ ⌈λ*s*. *s* ″*x*″ = *b* ∧ *s* ″*y*″ = *a*⌉
⟨*proof*⟩

lemma *var-swap-ref-var*:
rel-R ⌈λ*s*. *s* ″*x*″ = *a* ∧ *s* ″*y*″ = *b*⌉ ⌈λ*s*. *s* ″*x*″ = *b* ∧ *s* ″*y*″ = *a*⌉
⊇ (″*z*″ ::= (λ*s*. *s* ″*x*″)); (″*x*″ ::= (λ*s*. *s* ″*y*″)); (″*y*″ ::= (λ*s*. *s* ″*z*″))
⟨*proof*⟩

end

3 Isomorphisms Between Predicates, Sets and Relations

theory *P2S2R*
imports *Main*

begin

notation *relcomp* (**infixl** ‹;› *70*)
notation *inf* (**infixl** ‹⊓› *70*)
notation *sup* (**infixl** ‹⊔› *65*)
notation *Id-on* (‹*s2r*›)
notation *Domain* (‹*r2s*›)
notation *Collect* (‹*p2s*›)

definition *rel-n* :: ′*a rel* ⇒ ′*a rel* **where**
rel-n ≡ (λ*X*. *Id* ∩ − *X*)

lemma *subid-meet*: *R* ⊆ *Id* ⟹ *S* ⊆ *Id* ⟹ *R* ∩ *S* = *R* ; *S*
⟨*proof*⟩

3.1 Isomorphism Between Sets and Relations

lemma *srs*: *r2s* ∘ *s2r* = *id*

⟨*proof*⟩

lemma *rsr*: $R \subseteq Id \Longrightarrow s2r\ (r2s\ R) = R$
⟨*proof*⟩

lemma *s2r-inj*: *inj s2r*
⟨*proof*⟩

lemma *r2s-inj*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2s\ R = r2s\ S \Longrightarrow R = S$
⟨*proof*⟩

lemma *s2r-surj*: $\forall\, R \subseteq Id.\ \exists\, A.\ R = s2r\ A$
⟨*proof*⟩

lemma *r2s-surj*: $\forall\, A.\ \exists\, R \subseteq Id.\ A = r2s\ R$
⟨*proof*⟩

lemma *s2r-union-hom*: $s2r\ (A \cup B) = s2r\ A \cup s2r\ B$
⟨*proof*⟩

lemma *s2r-inter-hom*: $s2r\ (A \cap B) = s2r\ A \cap s2r\ B$
⟨*proof*⟩

lemma *s2r-inter-hom-var*: $s2r\ (A \cap B) = s2r\ A\ ;\ s2r\ B$
⟨*proof*⟩

lemma *s2r-compl-hom*: $s2r\ (-\ A) = rel\text{-}n\ (s2r\ A)$
⟨*proof*⟩

lemma *r2s-union-hom*: $r2s\ (R \cup S) = r2s\ R \cup r2s\ S$
⟨*proof*⟩

lemma *r2s-inter-hom*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2s\ (R \cap S) = r2s\ R \cap r2s\ S$
⟨*proof*⟩

lemma *r2s-inter-hom-var*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2s\ (R\ ;\ S) = r2s\ R \cap r2s\ S$
⟨*proof*⟩

lemma *r2s-ad-hom*: $R \subseteq Id \Longrightarrow r2s\ (rel\text{-}n\ R) = -\ r2s\ R$
⟨*proof*⟩

3.2 Isomorphism Between Predicates and Sets

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

definition *s2p* :: *'a set* ⇒ *'a pred* **where**
s2p S = (λ*x*. *x* ∈ *S*)

lemma *sps* [*simp*]: *s2p* ∘ *p2s* = *id*

⟨*proof*⟩

lemma *psp* [*simp*]: *p2s* ∘ *s2p* = *id*
⟨*proof*⟩

lemma *s2p-bij*: *bij s2p*
⟨*proof*⟩

lemma *p2s-bij*: *bij p2s*
⟨*proof*⟩

lemma *s2p-compl-hom*: *s2p* (− *A*) = − (*s2p A*)
⟨*proof*⟩

lemma *s2p-inter-hom*: *s2p* (*A* ∩ *B*) = (*s2p A*) ⊓ (*s2p B*)
⟨*proof*⟩

lemma *s2p-union-hom*: *s2p* (*A* ∪ *B*) = (*s2p A*) ⊔ (*s2p B*)
⟨*proof*⟩

lemma *p2s-neg-hom*: *p2s* (− *P*) = − (*p2s P*)
⟨*proof*⟩

lemma *p2s-conj-hom*: *p2s* (*P* ⊓ *Q*) = *p2s P* ∩ *p2s Q*
⟨*proof*⟩

lemma *p2s-disj-hom*: *p2s* (*P* ⊔ *Q*) = *p2s P* ∪ *p2s Q*
⟨*proof*⟩

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 = (λ*x*. *x* ∈ *Domain R*)

lemma *p2r-subid*: *p2r P* ⊆ *Id*
⟨*proof*⟩

lemma *p2s2r*: *p2r* = *s2r* ∘ *p2s*
⟨*proof*⟩

lemma *r2s2p*: *r2p* = *s2p* ∘ *r2s*
⟨*proof*⟩

lemma *prp* [*simp*]: *r2p* ∘ *p2r* = *id*
⟨*proof*⟩

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

lemma *p2r-inj*: *inj p2r*
⟨*proof*⟩

lemma *r2p-inj*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2p\ R = r2p\ S \Longrightarrow R = S$
⟨*proof*⟩

lemma *p2r-surj*: $\forall\ R \subseteq Id.\ \exists P.\ R = p2r\ P$
⟨*proof*⟩

lemma *r2p-surj*: $\forall P.\ \exists R \subseteq Id.\ P = r2p\ R$
⟨*proof*⟩

lemma *p2r-neg-hom*: $p2r\ (-\ P) = rel\text{-}n\ (p2r\ P)$
⟨*proof*⟩

lemma *p2r-conj-hom* [*simp*]: $p2r\ P \cap p2r\ Q = p2r\ (P \sqcap Q)$
⟨*proof*⟩

lemma *p2r-conj-hom-var* [*simp*]: $p2r\ P\ ;\ p2r\ Q = p2r\ (P \sqcap Q)$
⟨*proof*⟩

lemma *p2r-id-neg* [*simp*]: $Id \cap -\ p2r\ p = p2r\ (-p)$
⟨*proof*⟩

lemma [*simp*]: $p2r\ bot = \{\}$
⟨*proof*⟩

lemma *p2r-disj-hom* [*simp*]: $p2r\ P \cup p2r\ Q = p2r\ (P \sqcup Q)$
⟨*proof*⟩

lemma *r2p-ad-hom*: $R \subseteq Id \Longrightarrow r2p\ (rel\text{-}n\ R) = -\ (r2p\ R)$
⟨*proof*⟩

lemma *r2p-inter-hom*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2p\ (R \cap S) = (r2p\ R) \sqcap (r2p\ S)$
⟨*proof*⟩

lemma *r2p-inter-hom-var*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2p\ (R\ ;\ S) = (r2p\ R) \sqcap (r2p\ S)$
⟨*proof*⟩

lemma *rel-to-pred-union-hom*: $R \subseteq Id \Longrightarrow S \subseteq Id \Longrightarrow r2p\ (R \cup S) = (r2p\ R) \sqcup (r2p\ S)$
⟨*proof*⟩

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*: $H\ p\ x\ q \longleftrightarrow t\ p \cdot x \cdot n\ q = 0$
⟨*proof*⟩

lemma *H-var2*: $H\ p\ x\ q \longleftrightarrow t\ p \cdot x = t\ p \cdot x \cdot t\ q$
⟨*proof*⟩

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*: $H\ p\ 1\ p$
⟨*proof*⟩

lemma *H-cons-1*: $t\ p \leq t\ p' \Longrightarrow H\ p'\ x\ q \Longrightarrow H\ p\ x\ q$
⟨*proof*⟩

lemma *H-cons-2*: $t\ q' \leq t\ q \Longrightarrow H\ p\ x\ q' \Longrightarrow H\ p\ x\ q$
⟨*proof*⟩

lemma *H-cons*: $t\ p \leq t\ p' \Longrightarrow t\ q' \leq t\ q \Longrightarrow H\ p'\ x\ q' \Longrightarrow H\ p\ x\ q$
⟨*proof*⟩

lemma *H-seq-swap*: $H\ p\ x\ r \Longrightarrow H\ r\ y\ q \Longrightarrow H\ p\ (x \cdot y)\ q$
⟨*proof*⟩

lemma *H-seq*: $H\ r\ y\ q \Longrightarrow H\ p\ x\ r \Longrightarrow H\ p\ (x \cdot y)\ q$
⟨*proof*⟩

lemma *H-exp1*: $H\ (t\ p \cdot t\ r)\ x\ q \Longrightarrow H\ p\ (t\ r \cdot x)\ q$
⟨*proof*⟩

lemma *H-exp2*: $H\ p\ x\ q \Longrightarrow H\ p\ (x \cdot t\ r)\ (t\ q \cdot t\ r)$
⟨*proof*⟩

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*⟩

lemma *H-cond*: $H\ (t\ p \cdot t\ r)\ x\ q \Longrightarrow H\ (t\ p \cdot n\ r)\ y\ q \Longrightarrow H\ p\ (\mathit{if}\ r\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi})\ q$
⟨*proof*⟩

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*⟩

lemma *H-while-inv*: $t\ p \leq t\ i \Longrightarrow t\ i \cdot n\ r \leq t\ q \Longrightarrow H\ (t\ i \cdot t\ r)\ x\ i \Longrightarrow H\ p\ (\mathit{while}\ r\ \mathit{inv}\ i\ \mathit{do}\ x\ \mathit{od})\ q$
⟨*proof*⟩

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*⟩

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 ⌈P⌉ = ⌈P⌉*
⟨*proof*⟩

lemma *impl-prop* [*simp*]: *⌈P⌉ ⊆ ⌈Q⌉ ⟷ (∀ s. P s ⟶ Q s)*
⟨*proof*⟩

lemma *Id-simp* [*simp*]: *Id ∩ (− Id ∪ X) = Id ∩ X*
⟨*proof*⟩

lemma *Id-p2r* [*simp*]: *Id ∩ ⌈P⌉ = ⌈P⌉*
⟨*proof*⟩

lemma *Id-p2r-simp* [*simp*]: *Id ∩ (− Id ∪ ⌈P⌉) = ⌈P⌉*
⟨*proof*⟩

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*: *⌈λs. P (s (v := e s))⌉ ; (v ::= e) = (v ::= e) ; ⌈P⌉*
⟨*proof*⟩

lemma *H-assign*: *rel-kat.H ⌈λs. P (s (v := e s))⌉ (v ::= e) ⌈P⌉*
⟨*proof*⟩

lemma *H-assign-var*: *(∀ s. P s ⟶ Q (s (v := e s))) ⟹ rel-kat.H ⌈P⌉ (v ::= e) ⌈Q⌉*
⟨*proof*⟩

lemma *H-assign-iff* [*simp*]: *rel-kat.H ⌈P⌉ (v ::= e) ⌈Q⌉ ⟷ (∀ s. P s ⟶ Q (s (v := e s)))*
⟨*proof*⟩

lemma *H-assign-floyd*: *rel-kat.H ⌈P⌉ (v ::= e) ⌈λs. ∃ w. s v = e (s(v := w)) ∧ P (s(v := w))⌉*
⟨*proof*⟩

4.1.5 Simplified Hoare Rules

lemma *sH-cons-1*: *∀ s. P s ⟶ P' s ⟹ rel-kat.H ⌈P'⌉ X ⌈Q⌉ ⟹ rel-kat.H ⌈P⌉ X ⌈Q⌉*
⟨*proof*⟩

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* $\lceil P \rceil\ X\ \lceil Q \rceil$
⟨*proof*⟩

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* $\lceil P \rceil\ X\ \lceil Q \rceil$
⟨*proof*⟩

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* $\lceil T \rceil\ X\ Y$) $\lceil Q \rceil$
⟨*proof*⟩

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* $\lceil P \sqcap -\ T \rceil\ Y\ \lceil Q \rceil$)
⟨*proof*⟩

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* $\lceil R \rceil\ \lceil I \rceil\ X$) $\lceil Q \rceil$
⟨*proof*⟩

lemma *sH-H*: *rel-kat.H* $\lceil P \rceil\ X\ \lceil Q \rceil \longleftrightarrow (\forall s\ s'.\ P\ s \longrightarrow (s,s') \in X \longrightarrow Q\ s')$
⟨*proof*⟩

Finally we provide additional syntax for specifications and commands.

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

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

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-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* ⟷ (*PRE* (*p* ⊓ *r*) *x POST q*) ∧ (*PRE* (*p* ⊓ − *r*) *y POST q*)
⟨*proof*⟩

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* (λs. $s\ ''y'' \neq 0$) *INV* (λs. *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 (λs. $s\ ''x'' = gcd\ x\ y$)
⟨*proof*⟩

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

lemma *integer-division*:
PRE (λs::*nat store*. $s\ ''x'' \geq 0$)
($''q'' ::= (\lambda s.\ 0)$);
($''r'' ::= (\lambda s.\ s\ ''x'')$);
(*WHILE* (λs. $s\ ''y'' \leq s\ ''r''$) *INV* (λs. $s\ ''x'' = s\ ''q'' * s\ ''y'' + s\ ''r'' \wedge s\ ''r'' \geq 0$)
DO
($''q'' ::= (\lambda s.\ s\ ''q'' + 1)$);
($''r'' ::= (\lambda s.\ s\ ''r'' - s\ ''y'')$)
OD)
POST (λs. $s\ ''x'' = s\ ''q'' * s\ ''y'' + s\ ''r'' \wedge s\ ''r'' \geq 0 \wedge s\ ''r'' < s\ ''y''$)
⟨*proof*⟩

lemma *imp-reverse*:
PRE (λs:: *'a list store*. $s\ ''x'' = X$)
($''y'' ::= (\lambda s.\ [])$);
(*WHILE* (λs. $s\ ''x'' \neq []$) *INV* (λs. *rev* ($s\ ''x''$) @ $s\ ''y'' = rev\ X$)
DO
($''y'' ::= (\lambda s.\ hd\ (s\ ''x'')\ \#\ s\ ''y'')$);
($''x'' ::= (\lambda s.\ tl\ (s\ ''x''))$)
OD)
POST (λs. $s\ ''y'' = rev\ X$)
⟨*proof*⟩

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 (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*)
⟨*proof*⟩

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

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''*));

$(''x'' ::= (\lambda s.\ tl\ (s\ ''x'')))$
$OD)$
$POST\ (\lambda s.\ s\ ''y'' = rev\ X\ )$
⟨*proof*⟩

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*: $H\ p\ (R\ p\ q)\ q$
⟨*proof*⟩

lemma *R2*: $H\ p\ x\ q \Longrightarrow x \leq R\ p\ q$
⟨*proof*⟩

4.2.2 Propositional Refinement Calculus

lemma *R-skip*: $1 \leq R\ p\ p$
⟨*proof*⟩

lemma *R-cons*: $t\ p \leq t\ p' \Longrightarrow t\ q' \leq t\ q \Longrightarrow R\ p'\ q' \leq R\ p\ q$
⟨*proof*⟩

lemma *R-seq*: $(R\ p\ r) \cdot (R\ r\ q) \leq R\ p\ q$
⟨*proof*⟩

lemma *R-cond*: *if v then* $(R\ (t\ v \cdot t\ p)\ q)$ *else* $(R\ (n\ v \cdot t\ p)\ q)$ *fi* $\leq R\ p\ q$
⟨*proof*⟩

lemma *R-loop*: *while q do* $(R\ (t\ p \cdot t\ q)\ p)$ *od* $\leq R\ p\ (t\ p \cdot n\ q)$
⟨*proof*⟩

lemma *R-zero-one*: $x \leq R\ 0\ 1$
⟨*proof*⟩

lemma *R-one-zero*: $R\ 1\ 0 = 0$
⟨*proof*⟩

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**
$rel\text{-}R\ P\ Q = \bigcup \{X.\ rel\text{-}kat.H\ P\ X\ Q\}$

interpretation *rel-rkat*: *rkat* $(\cup)$ $(;)$ *Id* $\{\}$ $(\subseteq)$ $(\subset)$ *rtrancl* $(\lambda X.\ Id \cap -\ X)$ *rel-R*
⟨*proof*⟩

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*: $(\forall s.\ P\ s \longrightarrow Q\ (s\ (v := e\ s))) \Longrightarrow (v ::= e) \subseteq rel\text{-}R\ \lceil P \rceil\ \lceil Q \rceil$
⟨*proof*⟩

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*⟩

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*⟩

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* $\lceil P \rceil\ \lceil Q \rceil$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *while-ref*: $X \subseteq X' \Longrightarrow$ *WHILE P DO X OD* $\subseteq$ *WHILE P DO X' OD*
⟨*proof*⟩

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-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-R* $\lceil \lambda s.\ s\ ''z'' = a \wedge s\ ''y'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
⟨*proof*⟩

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* $\lceil \lambda s.\ s\ ''z'' = a \wedge s\ ''x'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
⟨*proof*⟩

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* $\lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
⟨*proof*⟩

lemma *var-swap-ref-var*:
rel-R $\lceil \lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b \rceil\ \lceil \lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a \rceil$
$\supseteq (''z'' ::= (\lambda s.\ s\ ''x''))$; $(''x'' ::= (\lambda s.\ s\ ''y''))$; $(''y'' ::= (\lambda s.\ s\ ''z''))$
⟨*proof*⟩

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*: $d\ p \cdot d\ q \leq |x]\ t \longleftrightarrow d\ p \leq ad\ q + |x]\ t$
⟨*proof*⟩

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*: $ad\ p + |x]\ q = |d\ p \cdot x]\ q$
⟨*proof*⟩

lemma *fbox-export2*: $|x]\ p \leq |x \cdot ad\ q]\ (d\ p \cdot ad\ q)$
⟨*proof*⟩

lemma *fbox-export3*: $|x \cdot ad\ p]\ q = |x]\ (d\ p + d\ q)$
⟨*proof*⟩

lemma *fbox-seq* [*simp*]: $|x \cdot y]\ q = |x]\ |y]\ q$
⟨*proof*⟩

lemma *fbox-seq-var*: $p' \leq |y]\ q \Longrightarrow p \leq |x]\ p' \Longrightarrow p \leq |x \cdot y]\ q$
⟨*proof*⟩

lemma *fbox-cond-var* [*simp*]: $|if\ p\ then\ x\ else\ y\ fi]\ q = (ad\ p + |x]\ q) \cdot (d\ p + |y]\ q)$
⟨*proof*⟩

lemma *fbox-cond-aux1* [*simp*]: $d\ p \cdot |if\ p\ then\ x\ else\ y\ fi]\ q = d\ p \cdot |x]\ q$
⟨*proof*⟩

lemma *fbox-cond-aux2* [*simp*]: $ad\ p \cdot |if\ p\ then\ x\ else\ y\ fi]\ q = ad\ p \cdot |y]\ q$
⟨*proof*⟩

lemma *fbox-cond* [*simp*]: $|if\ p\ then\ x\ else\ y\ fi]\ q = (d\ p \cdot |x]\ q) + (ad\ p \cdot |y]\ q)$
⟨*proof*⟩

lemma *fbox-cond-var2* [*simp*]: $|if\ p\ then\ x\ else\ y\ fi]\ q = if\ p\ then\ |x]\ q\ else\ |y]\ q\ fi$
⟨*proof*⟩

lemma *fbox-while-unfold*: $|while\ t\ do\ x\ od]\ p = (d\ t + d\ p) \cdot (ad\ t + |x]\ |while\ t\ do\ x\ od]\ p)$
⟨*proof*⟩

lemma *fbox-while-var1*: $d\ t \cdot |while\ t\ do\ x\ od]\ p = d\ t \cdot |x]\ |while\ t\ do\ x\ od]\ p$
⟨*proof*⟩

lemma *fbox-while-var2*: $ad\ t \cdot |while\ t\ do\ x\ od]\ p \leq d\ p$
⟨*proof*⟩

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*⟩

lemma *fbox-while-var*: $d\ p = |d\ t \cdot x]\ p \Longrightarrow d\ p \leq |while\ t\ do\ x\ od]\ (d\ p \cdot ad\ t)$
⟨*proof*⟩

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*⟩

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

lemma *fbox-whilei-break*: $d\ p \leq |y]\ i \Longrightarrow d\ i \cdot ad\ t \leq d\ q \Longrightarrow d\ i \cdot d\ t \leq |x]\ i \Longrightarrow d\ p \leq |y \cdot (while\ t\ inv\ i\ do\ x\ od)]\ q$

⟨*proof*⟩

Finally we derive a frame rule.

lemma *fbox-frame*: $d\ p \cdot x \leq x \cdot d\ p \Longrightarrow d\ q \leq |x]\ t \Longrightarrow d\ p \cdot d\ q \leq |x]\ (d\ p \cdot d\ t)$
⟨*proof*⟩

lemma *fbox-frame-var*: $d\ p \leq |x]\ p \Longrightarrow d\ q \leq |x]\ t \Longrightarrow d\ p \cdot d\ q \leq |x]\ (d\ p \cdot d\ t)$
⟨*proof*⟩

end

5.1.4 Store and Assignment

type-synonym *'a store* = *string* ⇒ *'a*

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

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

lemma *assign-prop*: $\lceil \lambda s.\ P\ (s\ (v := e\ s))\rceil\ ;\ (v ::= e) = (v ::= e)\ ;\ \lceil P\rceil$
⟨*proof*⟩

lemma *wp-assign* [*simp*]: *wp* $(v ::= e)\ \lceil Q\rceil = \lceil \lambda s.\ Q\ (s\ (v := e\ s))\rceil$
⟨*proof*⟩

lemma *wp-assign-var* [*simp*]: $\lfloor wp\ (v ::= e)\ \lceil Q\rceil\rfloor = (\lambda s.\ Q\ (s\ (v := e\ s)))$
⟨*proof*⟩

lemma *wp-assign-det*: *wp* $(v ::= e)\ \lceil Q\rceil$ = *relfdia* $(v ::= e)\ \lceil Q\rceil$
⟨*proof*⟩

5.1.5 Simplifications

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

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

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* $\lceil P\rceil\ \lceil I\rceil\ X$

lemma *d-p2r* [*simp*]: *rdom* $\lceil P \rceil = \lceil P \rceil$
⟨*proof*⟩

lemma *p2r-conj-hom-var-symm* [*simp*]: $\lceil P \rceil \ ; \lceil Q \rceil = \lceil P \sqcap Q \rceil$
⟨*proof*⟩

lemma *p2r-neg-hom* [*simp*]: *rel-ad* $\lceil P \rceil = \lceil - P \rceil$
⟨*proof*⟩

lemma *wp-trafo*: $\lfloor wp\ X\ \lceil Q \rceil \rfloor = (\lambda s.\ \forall\, s'.\ (s,s') \in X \longrightarrow Q\ s')$
⟨*proof*⟩

lemma *wp-trafo-var*: $\lfloor wp\ X\ \lceil Q \rceil \rfloor\ s = (\forall\, s'.\ (s,s') \in X \longrightarrow Q\ s')$
⟨*proof*⟩

lemma *wp-simp*: *rdom* $\lceil \lfloor wp\ X\ Q \rfloor \rceil = wp\ X\ Q$
⟨*proof*⟩

lemma *wp-simp-var* [*simp*]: $wp\ \lceil P \rceil\ \lceil Q \rceil = \lceil - P \sqcup Q \rceil$
⟨*proof*⟩

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

lemma *p2r-eq-prop* [*simp*]: $\lceil P \rceil = \lceil Q \rceil \longleftrightarrow (\forall\, s.\ P\ s \longleftrightarrow\ Q\ s)$
⟨*proof*⟩

lemma *impl-prop-var* [*simp*]: *rdom* $\lceil P \rceil \subseteq$ *rdom* $\lceil Q \rceil \longleftrightarrow (\forall\, s.\ P\ s \longrightarrow\ Q\ s)$
⟨*proof*⟩

lemma *p2r-eq-prop-var* [*simp*]: *rdom* $\lceil P \rceil =$ *rdom* $\lceil Q \rceil \longleftrightarrow (\forall\, s.\ P\ s \longleftrightarrow\ Q\ s)$
⟨*proof*⟩

lemma *wp-whilei*: $(\forall\, s.\ P\ s \longrightarrow I\ s) \Longrightarrow (\forall\, s.\ (I \sqcap -T)\ s \longrightarrow Q\ s) \Longrightarrow (\forall\, s.\ (I \sqcap T)\ s \longrightarrow \lfloor wp\ X\ \lceil I \rceil \rfloor\ s)$
$\Longrightarrow (\forall\, s.\ P\ s \longrightarrow \lfloor wp\ (WHILE\ T\ INV\ I\ DO\ X\ OD)\ \lceil Q \rceil \rfloor\ s)$
⟨*proof*⟩

end

5.1.6 Verification Examples

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

begin

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
$(''z'' ::= (\lambda s.\ s\ ''y''));$
$(''y'' ::= (\lambda s.\ s\ ''x''\ mod\ s\ ''y''));$
$(''x'' ::= (\lambda s.\ s\ ''z''))$
$OD)$
$POST\ (\lambda s.\ s\ ''x'' = gcd\ x\ y)$
$\langle proof \rangle$

lemma *euclid-diff*:
$PRE\ (\lambda s{::}nat\ store.\ s\ ''x'' = x \wedge s\ ''y'' = y \wedge x > 0 \wedge y > 0)$
$(WHILE\ (\lambda s.\ s\ ''x'' \neq s\ ''y'')\ INV\ (\lambda s.\ gcd\ (s\ ''x'')\ (s\ ''y'') = gcd\ x\ y)$
DO
$(IF\ (\lambda s.\ s\ ''x'' > s\ ''y'')$
$THEN\ (''x'' ::= (\lambda s.\ s\ ''x'' - s\ ''y''))$
$ELSE\ (''y'' ::= (\lambda s.\ s\ ''y'' - s\ ''x''))$
$FI)$
$OD)$
$POST\ (\lambda s.\ s\ ''x'' = gcd\ x\ y)$
$\langle proof \rangle$

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

lemma *maximum*:
$PRE\ (\lambda s{::}\ nat\ store.\ True)$
$(IF\ (\lambda s.\ s\ ''x'' \geq s\ ''y'')$
$THEN\ (''z'' ::= (\lambda s.\ s\ ''x''))$
$ELSE\ (''z'' ::= (\lambda s.\ s\ ''y''))$
$FI)$
$POST\ (\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y''))$
$\langle proof \rangle$

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

⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *factorial2*:
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)
  ⟨proof⟩

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)
  ⟨proof⟩

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 )
  ⟨proof⟩

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*]: $r\ (ad\ p) = ad\ p$
⟨*proof*⟩

lemma *bdia-export1*: $\langle x|\ (r\ p \cdot r\ t) = \langle r\ t \cdot x|\ p$
⟨*proof*⟩

lemma *bdia-export2*: $r\ p \cdot \langle x|\ q = \langle x \cdot r\ p|\ q$
⟨*proof*⟩

lemma *bdia-seq* [*simp*]: $\langle x \cdot y|\ q = \langle y|\ \langle x|\ q$
⟨*proof*⟩

lemma *bdia-seq-var*: $\langle x|\ p \leq p' \Longrightarrow \langle y|\ p' \leq q \Longrightarrow \langle x \cdot y|\ p \leq q$
⟨*proof*⟩

lemma *bdia-cond-var* [*simp*]: $\langle$*if p then x else y fi*$|\ q = \langle x|\ (d\ p \cdot r\ q) + \langle y|\ (ad\ p$
$\cdot\ r\ q)$
⟨*proof*⟩

lemma *bdia-while*: $\langle x|\ (d\ t \cdot r\ p) \leq r\ p \Longrightarrow \langle$*while t do x od*$|\ p \leq r\ p \cdot ad\ t$
⟨*proof*⟩

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*⟩

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*$)|\ p \leq r\ q$
⟨*proof*⟩

end

5.2.2 Floyd's Assignment Rule

lemma *bdia-assign* [*simp*]: *rel-antirange-kleene-algebra.bdia* $(v ::= e)\ \lceil P \rceil = \lceil \lambda s.$
$\exists w.\ s\ v = e\ (s(v := w)) \wedge P\ (s(v{:=}w))\rceil$
⟨*proof*⟩

lemma *d-p2r* [*simp*]: *rel-antirange-kleene-algebra.ars-r* $\lceil P \rceil = \lceil P \rceil$
⟨*proof*⟩

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

lemma *euclid-diff*:
$FPRE\ (\lambda s{::}nat\ store.\ s\ ''x'' = x \wedge s\ ''y'' = y \wedge x > 0 \wedge y > 0)$
$(WHILE\ (\lambda s.\ s\ ''x'' \neq s\ ''y'')\ INV\ (\lambda s.\ gcd\ (s\ ''x'')\ (s\ ''y'') = gcd\ x\ y)$
DO
$(IF\ (\lambda s.\ s\ ''x'' > s\ ''y'')$
$THEN\ (''x'' ::= (\lambda s.\ s\ ''x'' - s\ ''y''))$
$ELSE\ (''y'' ::= (\lambda s.\ s\ ''y'' - s\ ''x''))$
$FI)$
$OD)$
$POST\ (\lambda s.\ s\ ''x'' = gcd\ x\ y)$
⟨*proof*⟩

lemma *varible-swap*:
$FPRE\ (\lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b)$
$(''z'' ::= (\lambda s.\ s\ ''x''));$
$(''x'' ::= (\lambda s.\ s\ ''y''));$
$(''y'' ::= (\lambda s.\ s\ ''z''))$
$POST\ (\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a)$
⟨*proof*⟩

lemma *maximum*:
$FPRE\ (\lambda s{::}\ nat\ store.\ True)$
$(IF\ (\lambda s.\ s\ ''x'' \geq s\ ''y'')$
$THEN\ (''z'' ::= (\lambda s.\ s\ ''x''))$
$ELSE\ (''z'' ::= (\lambda s.\ s\ ''y''))$
$FI)$
$POST\ (\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y''))$
⟨*proof*⟩

lemma *integer-division*:
$FPRE\ (\lambda s{::}nat\ store.\ x \geq 0)$

$(''q'' ::= (\lambda s.\ 0));$
$(''r'' ::= (\lambda s.\ x));$
$(WHILE\ (\lambda s.\ y \leq s\ ''r'')\ INV\ (\lambda s.\ x = s\ ''q'' * y + s\ ''r'' \wedge s\ ''r'' \geq 0)$
DO
$(''q'' ::= (\lambda s.\ s\ ''q'' + 1));$
$(''r'' ::= (\lambda s.\ s\ ''r'' - y))$
$OD)$
$POST\ (\lambda s.\ x = s\ ''q'' * y + s\ ''r'' \wedge s\ ''r'' \geq 0 \wedge s\ ''r'' < y)$
⟨*proof*⟩

lemma *factorial*:
$FPRE\ (\lambda s{::}nat\ store.\ True)$
$(''x'' ::= (\lambda s.\ 0));$
$(''y'' ::= (\lambda s.\ 1));$
$(WHILE\ (\lambda s.\ s\ ''x'' \neq x0)\ INV\ (\lambda s.\ s\ ''y'' = fact\ (s\ ''x''))$
DO
$(''x'' ::= (\lambda s.\ s\ ''x'' + 1));$
$(''y'' ::= (\lambda s.\ s\ ''y'' \cdot s\ ''x''))$
$OD)$
$POST\ (\lambda s.\ s\ ''y'' = fact\ x0)$
⟨*proof*⟩

lemma *my-power*:
$FPRE\ (\lambda s{::}nat\ store.\ True)$
$(''i'' ::= (\lambda s.\ 0));$
$(''y'' ::= (\lambda s.\ 1));$
$(WHILE\ (\lambda s.\ s\ ''i'' < n)\ INV\ (\lambda s.\ s\ ''y'' = x \hat{}\ (s\ ''i'') \wedge s\ ''i'' \leq n)$
DO
$(''y'' ::= (\lambda s.\ (s\ ''y'') * x));$
$(''i'' ::= (\lambda s.\ s\ ''i'' + 1))$
$OD)$
$POST\ (\lambda s.\ s\ ''y'' = x \hat{}\ n)$
⟨*proof*⟩

lemma *imp-reverse*:
$FPRE\ (\lambda s{::}\ 'a\ list\ store.\ s\ ''x'' = X)$
$(''y'' ::= (\lambda s.\ []));$
$(WHILE\ (\lambda s.\ s\ ''x'' \neq [])\ INV\ (\lambda s.\ rev\ (s\ ''x'')\ @\ s\ ''y'' = rev\ X)$
DO
$(''y'' ::= (\lambda s.\ hd\ (s\ ''x'')\ \#\ s\ ''y''));$
$(''x'' ::= (\lambda s.\ tl\ (s\ ''x'')))$
$OD)$
$POST\ (\lambda s.\ s\ ''y'' = rev\ X\ )$
⟨*proof*⟩

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 = ⋃ {*P*. *P* ⊆ *relfdia X P*}

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

lemma *rel-nabla-d-closed* [*simp*]: *rdom* (*rel-nabla x*) = *rel-nabla x*
⟨*proof*⟩

lemma *rel-nabla-bin-d-closed* [*simp*]: *rdom* (*rel-nabla-bin x q*) = *rel-nabla-bin x q*
⟨*proof*⟩

lemma *rel-nabla-unfold*: *rel-nabla X* ⊆ *relfdia X* (*rel-nabla X*)
⟨*proof*⟩

lemma *rel-nabla-bin-unfold*: *rel-nabla-bin X Q* ⊆ *relfdia X* (*rel-nabla-bin X Q*) ∪ *rdom Q*
⟨*proof*⟩

lemma *rel-nabla-coinduct-var*: *P* ⊆ *relfdia X P* ⟹ *P* ⊆ *rel-nabla X*
⟨*proof*⟩

lemma *rel-nabla-bin-coinduct*: *P* ⊆ *relfdia X P* ∪ *rdom Q* ⟹ *P* ⊆ *rel-nabla-bin X Q*
⟨*proof*⟩

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*⟩

lemma *rel-ad-inter-seq*: *rel-ad X* ∩ *rel-ad Y* = *rel-ad X* ; *rel-ad Y*

⟨*proof*⟩

lemma *fusion2-aux2*: *rdom* (*rel-nabla-bin X Q*) ⊆ *rdom* (*rel-nabla-bin X Q* ∩ *rel-ad* (*relfdia* (*X**) *Q*) ∪ *relfdia* (*X**) *Q*)
⟨*proof*⟩

lemma *nabla-fusion2*: *rel-nabla-bin X Q* ⊆ *rel-nabla X* ∪ *relfdia* (*X**) *Q*
⟨*proof*⟩

lemma *rel-nabla-coinduct*: *P* ⊆ *relfdia X P* ∪ *rdom Q* ⟹ *P* ⊆ *rel-nabla X* ∪ *relfdia* (*rtrancl X*) *Q*
⟨*proof*⟩

interpretation *rel-fdivka*: *fdivergence-kleene-algebra rel-ad* (∪) (;) *Id* {} (⊆) (⊂) *rtrancl rel-nabla*
⟨*proof*⟩

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*: ∇*x* = *0* ⟹ *d p* ≤ *d q* + |*x*⟩ *p* ⟹ *d p* ≤ |*x**⟩ *q*
⟨*proof*⟩

lemma *fdia-arden-eq*: ∇*x* = *0* ⟹ *d p* = *d q* + |*x*⟩ *p* ⟹ *d p* = |*x**⟩ *q*
⟨*proof*⟩

lemma *fdia-arden-iff*: ∇*x* = *0* ⟹ (*d p* = *d q* + |*x*⟩ *p* ⟷ *d p* = |*x**⟩ *q*)
⟨*proof*⟩

lemma |*x**] *p* ≤ |*x*] *p*
⟨*proof*⟩

lemma *fbox-arden*: ∇*x* = *0* ⟹ *d q* · |*x*] *p* ≤ *d p* ⟹ |*x**] *q* ≤ *d p*
⟨*proof*⟩

lemma *fbox-arden-eq*: ∇*x* = *0* ⟹ *d q* · |*x*] *p* = *d p* ⟹ |*x**] *q* = *d p*
⟨*proof*⟩

lemma *fbox-arden-iff*: ∇*x* = *0* ⟹ (*d p* = *d q* · |*x*] *p* ⟷ *d p* = |*x**] *q*)
⟨*proof*⟩

lemma *fbox-arden-while-iff*: ∇ (*d t* · *x*) = *0* ⟹ (*d p* = (*d t* + *d q*) · |*d t* · *x*] *p* ⟷ *d p* = |*while t do x od*] *q*)
⟨*proof*⟩

lemma *fbox-arden-whilei*: $\nabla\ (d\ t \cdot x) = 0 \Longrightarrow (d\ i = (d\ t + d\ q) \cdot |d\ t \cdot x]\ i \Longrightarrow d\ i = |while\ t\ inv\ i\ do\ x\ od]\ q)$
⟨*proof*⟩

lemma *fbox-arden-whilei-iff*: $\nabla\ (d\ t \cdot x) = 0 \Longrightarrow (d\ i = (d\ t + d\ q) \cdot |d\ t \cdot x]\ i \longleftrightarrow d\ i = |while\ t\ inv\ i\ do\ x\ od]\ q)$
⟨*proof*⟩

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*: *Noetherian* $x \Longrightarrow \nabla\ x = 0$
⟨*proof*⟩

lemma *nabla-noether-iff*: *Noetherian* $x \longleftrightarrow \nabla\ x = 0$
⟨*proof*⟩

lemma *nabla-preloeb-iff*: $\nabla\ x = 0 \longleftrightarrow PreLoebian\ x$
⟨*proof*⟩

end

lemma *rel-nabla-prop*: *rel-nabla* $R = \{\} \longleftrightarrow (\forall P.\ P \subseteq relfdia\ R\ P \longrightarrow P = \{\})$
⟨*proof*⟩

lemma *fdia-rel-im1*: *s2r* $((converse\ R)\ ``\ P) = relfdia\ R\ (s2r\ P)$
⟨*proof*⟩

lemma *fdia-rel-im2*: *s2r* $((converse\ R)\ ``\ (r2s\ (rdom\ P))) = relfdia\ R\ P$
⟨*proof*⟩

lemma *wf-nabla-aux*: $(P \subseteq (converse\ R)\ ``\ P \longrightarrow P = \{\}) \longleftrightarrow (s2r\ P \subseteq relfdia\ R\ (s2r\ P) \longrightarrow s2r\ P = \{\})$
⟨*proof*⟩

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* $(converse\ R) \longleftrightarrow rel\text{-}nabla\ R = \{\}$
⟨*proof*⟩

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-nabla (
⌈λs::*nat store*. *0* < *s* ″*y*″⌉ ;
((″*z*″ ::= (λs. *s* ″*y*″)) ;
(″*y*″ ::= (λs. *s* ″*x*″ *mod s* ″*y*″)) ;
(″*x*″ ::= (λs. *s* ″*z*″))))
= {}
⟹
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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ppath-assign* [*simp*]: *ppath-aka.fbox* (*v* ::= *e*) (*p2pp Q*) = *p2pp* (λ*s*. *Q* (*s*(*v* := *e s*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma [*simp*]: *p2pp P*; *p2pp Q* = *p2pp* (*P* ⊓ *Q*)
⟨*proof*⟩

lemma [*intro*!]: *P* ≤ *Q* ⟹ ⌈*P*⌉ ⊆ ⌈*Q*⌉
⟨*proof*⟩

lemma [*simp*]: *rdom* ⌈*P*⌉ = ⌈*P*⌉
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*''))
⟨*proof*⟩

lemma *integer-division*:
PRE (λs::*nat store*. $x \geq 0$)
($''q''$ ::= (λs. 0));
($''r''$ ::= (λs. x));
(*WHILE* (λs. $y \leq s\ ''r''$) *INV* (λs. $x = s\ ''q'' * y + s\ ''r'' \wedge s\ ''r'' \geq 0$)
DO
($''q''$ ::= (λs. $s\ ''q'' + 1$));
($''r''$ ::= (λs. $s\ ''r'' - y$))
OD)
POST (λs. $x = s\ ''q'' * y + s\ ''r'' \wedge s\ ''r'' \geq 0 \wedge s\ ''r'' < y$)
⟨*proof*⟩

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* (λs. $s\ ''y'' \neq 0$) *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)
⟨*proof*⟩

lemma *euclid-diff2*:
PRE (λs::*nat store*. $s\ ''x'' = x \wedge s\ ''y'' = y \wedge x > 0 \wedge y > 0$)
(*WHILE* (λs. $s\ ''x'' \neq s\ ''y''$) *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)
⟨*proof*⟩

lemma *varible-swap2*:
PRE (λs. $s\ ''x'' = a \wedge s\ ''y'' = b$)
($''z''$ ::= (λs. $s\ ''x''$));
($''x''$ ::= (λs. $s\ ''y''$));

```
   (''y'' ::= (λs. s ''z''))
  POST (λs. s ''x'' = b ∧ s ''y'' = a)
 ⟨proof⟩

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''))
  ⟨proof⟩

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)
  ⟨proof⟩

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)
 ⟨proof⟩

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 )
 ⟨proof⟩

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 (λ*s* :: *'a state*. *List* (*projr* (*s ''h''*)) (*projl* (*s ''p''*)) *Ps*
∧ *List* (*projr* (*s ''h''*)) (*projl* (*s ''q''*)) *Qs*
∧ *set Ps* ∩ *set Qs* = {})
(*WHILE* (λ*s*. *projl* (*s ''p''*) ≠ *Null*)
INV (λ*s*. ∃ *ps qs*. *List* (*projr* (*s ''h''*)) (*projl* (*s ''p''*)) *ps*
∧ *List* (*projr* (*s ''h''*)) (*projl* (*s ''q''*)) *qs*
∧ *set ps* ∩ *set qs* = {} ∧ *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*))
⟨*proof*⟩

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* (+) (·) *1 0* (≤) (<) *star antidomain-op*
⟨*proof*⟩

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

lemma *H-kat-to-kad*: $H\ p\ x\ q \longleftrightarrow d\ p \leq |x]\ (d\ q)$
⟨*proof*⟩

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*
⟨*proof*⟩

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* = (⌈*p*⌉ ≤ (*rel-antidomain-kleene-algebra.fbox x*) ⌈*q*⌉)
⟨*proof*⟩

lemma *cond-iff*: *rel-kat.ifthenelse* ⌈*P*⌉ *X Y* = *rel-antidomain-kleene-algebra.cond* ⌈*P*⌉ *X Y*
⟨*proof*⟩

lemma *gets-iff*: *v* ::= *e* = *VC-KAD.gets v e*
⟨*proof*⟩

Finally we present two examples to test the integration.

lemma *maximum*:
PRE (λ*s*:: *nat store*. *True*)
(*IF* (λ*s*. *s* ″*x*″ ≥ *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*″))
⟨*proof*⟩

lemma *maximum2*:
PRE (λ*s*:: *nat store*. *True*)
(*IF* (λ*s*. *s* ″*x*″ ≥ *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*″))
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *dioid-one-zero* (⊔) (·) *1 bot* (≤) (<)
⟨*proof*⟩

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* (⊔) (·) *1* ⊥ *d* (≤) (<)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *inf-bot-iff-le*: $x \sqcap y = \bot \longleftrightarrow x \leq -y$
  ⟨*proof*⟩

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* $(\cdot)$ $(\sqcap)$ $(\leq)$ $(<)$ $(\sqcup)$ $\bot$ $\top$
  ⟨*proof*⟩

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

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

lemma *topr*: $-(x \cdot \top) \cdot \top = -(x \cdot \top)$
⟨*proof*⟩

lemma *dd-a*: $dblo.d\ x = a\ (a\ x)$
  ⟨*proof*⟩

lemma *ad-a* [*simp*]: $a\ (dblo.d\ x) = a\ x$
  ⟨*proof*⟩

lemma *da-a* [*simp*]: $dblo.d\ (a\ x) = a\ x$
  ⟨*proof*⟩

lemma *a1* [*simp*]: $a\ x \cdot x = \bot$
⟨*proof*⟩

lemma *a2* [*simp*]: $a\ (x \cdot y) \sqcup a\ (x \cdot a\ (a\ y)) = a\ (x \cdot a\ (a\ y))$

⟨*proof*⟩

lemma *a3* [*simp*]: $a\ (a\ x) \sqcup a\ x = 1$
⟨*proof*⟩

subclass *domain-bdlo-monoid* ⟨*proof*⟩

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* $a\ (\sqcup)\ (\cdot)\ 1\ \bot\ (\leq)\ (<)$
rewrites *ad-eq*: $ad.ads\text{-}d\ x = d\ x$
⟨*proof*⟩

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*: $ar\ x = 1 \sqcap -(\top \cdot x)$
⟨*proof*⟩

lemma *topl*: $\top \cdot (-(\top \cdot x)) = -(\top \cdot x)$
⟨*proof*⟩

lemma *r-ar*: $r\ x = ar\ (ar\ x)$
⟨*proof*⟩

lemma *ar-ar* [*simp*]: $ar\ (r\ x) = ar\ x$
⟨*proof*⟩

lemma *rar-ar* [*simp*]: $r\ (ar\ x) = ar\ x$
⟨*proof*⟩

lemma *ar1* [*simp*]: $x \cdot ar\ x = \bot$
⟨*proof*⟩

lemma *ars*: $r\ (r\ x \cdot y) = r\ (x \cdot y)$
⟨*proof*⟩

lemma *ar2* [*simp*]: $ar\ (x \cdot y) \sqcup ar\ (ar\ (ar\ x) \cdot y) = ar\ (ar\ (ar\ x) \cdot y)$

⟨*proof*⟩

lemma *ar3* [*simp*]: *ar* (*ar x*) ⊔ *ar x* = *1*
⟨*proof*⟩

sublocale *ar*: *antirange-semiring* (⊔) (·) *1* ⊥ *ar* (≤) (<)
rewrites *ar-eq*: *ar.ars-r x* = *r x*
⟨*proof*⟩

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* · *y* = *Sup* {*z*. ∃ *x* ∈ *X*. *z* = *x* · *y*}
and *Sup-distl*: *x* · *Sup Y* = *Sup* {*z*. ∃ *y* ∈ *Y*. *z* = *x* · *y*}

begin

lemma *bot-annil''* [*simp*]: ⊥ · *x* = ⊥
⟨*proof*⟩

lemma *bot-annirr''* [*simp*]: *x* · ⊥ = ⊥
⟨*proof*⟩

lemma *sup-distl*: *x* · (*y* ⊔ *z*) = *x* · *y* ⊔ *x* · *z*
⟨*proof*⟩

lemma *sup-distr*: (*x* ⊔ *y*) · *z* = *x* · *z* ⊔ *y* · *z*
⟨*proof*⟩

sublocale *semiring-one-zero* (⊔) (·) *1* ⊥
⟨*proof*⟩

sublocale *dioid-one-zero* (⊔) (·) *1* ⊥ (≤) (<)
⟨*proof*⟩

lemma *Sup-sup-pred*: *x* ⊔ *Sup*{*y*. *P y*} = *Sup*{*y*. *y* = *x* ∨ *P y*}
⟨*proof*⟩

definition *star* :: *'a* ⇒ *'a* **where**
star x = (*SUP i*. *x* ^ *i*)

lemma *star-def-var1*: *star x* = *Sup*{*y*. ∃ *i*. *y* = *x* ^ *i*}
⟨*proof*⟩

lemma *star-def-var2*: *star x* = *Sup*{*x* ^ *i* |*i*. *True*}
⟨*proof*⟩

lemma *star-unfoldl'* [*simp*]: $1 \sqcup x \cdot (star\ x) = star\ x$
⟨*proof*⟩

lemma *star-unfoldr'* [*simp*]: $1 \sqcup (star\ x) \cdot x = star\ x$
⟨*proof*⟩

lemma (**in** *dioid-one-zero*) *power-inductl*: $z + x \cdot y \leq y \Longrightarrow (x \hat{} n) \cdot z \leq y$
⟨*proof*⟩

lemma (**in** *dioid-one-zero*) *power-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot (x \hat{} n) \leq y$
⟨*proof*⟩

lemma *star-inductl'*: $z \sqcup x \cdot y \leq y \Longrightarrow (star\ x) \cdot z \leq y$
⟨*proof*⟩

lemma *star-inductr'*: $z \sqcup y \cdot x \leq y \Longrightarrow z \cdot (star\ x) \leq y$
⟨*proof*⟩

sublocale *ka*: *kleene-algebra* $(\sqcup)$ $(\cdot)$ 1 $\bot$ $(\leq)$ $(<)$ *star*
⟨*proof*⟩

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*
⟨*proof*⟩

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

⟨*proof*⟩

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*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

end

7.6 Boolean Domain Quantales

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

begin

subclass *domain-boolean-monoid*
  ⟨*proof*⟩

lemma *fbox-eq*: $ad.fbox\ x\ q = Sup\{d\ p\ |p.\ d\ p \cdot x \leq x \cdot d\ q\}$
  ⟨*proof*⟩

lemma *fdia-eq*: $ad.fdia\ x\ p = Inf\{d\ q\ |q.\ x \cdot d\ p \leq d\ q \cdot x\}$
  ⟨*proof*⟩

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*⟩

lemma $d\ p \leq ad.fbox\ x\ (d\ q) \Longrightarrow x \leq R\ p\ q$
  ⟨*proof*⟩

end

7.7 Relational Model of Boolean Domain Quantales

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

7.8 Modal Boolean Quantales

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

begin

subclass *range-boolean-monoid*
⟨*proof*⟩

lemma *fbox-eq*: *ar.bbox x* (*r q*) = *Sup*{*r p* |*p*. *x* · *r p* ≤ (*r q*) · *x*}
⟨*proof*⟩

lemma *fdia-eq*: *ar.bdia x* (*r p*) = *Inf*{*r q* |*q*. (*r p*) · *x* ≤ *x* · *r q*}
⟨*proof*⟩

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* (⊔) (·) *1* ⊥ (≤) (<) *star a ar*
⟨*proof*⟩

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* ⇒ *′a* :: *domain-boolean-quantale*) ⟹
(⋀*x*. *d p* ≤ |*x*] *d q* ⟹ *d p* ≤ |*f x*] *d q*) ⟹ *d p* ≤ |*lfp f*] *d q*
⟨*proof*⟩

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.