Concurrent Refinement Algebra and Rely Quotients

Julian Fell and Ian Hayes and Andrius Velykis

February 23, 2021

Abstract

The concurrent refinement algebra developed here is designed to provide a foundation for rely/guarantee reasoning about concurrent programs. The algebra builds on a complete lattice of commands by providing sequential composition, parallel composition and a novel weak conjunction operator. The weak conjunction operator coincides with the lattice supremum providing its arguments are non-aborting, but aborts if either of its arguments do. Weak conjunction provides an abstract version of a guarantee condition as a guarantee process. We distinguish between models that distribute sequential composition over non-deterministic choice from the left (referred to as being conjunctive in the refinement calculus literature) and those that don’t. Least and greatest fixed points of monotone functions are provided to allow recursion and iteration operators to be added to the language. Additional iteration laws are available for conjunctive models. The rely quotient of processes $c$ and $i$ is the process that, if executed in parallel with $i$ implements $c$. It represents an abstract version of a rely condition generalised to a process.
Contents

1 Overview 4

2 Refinement Lattice 4

3 Sequential Operator 7
   3.1 Basic sequential ................................. 7
   3.2 Distributed sequential ......................... 8

4 Parallel Operator 10
   4.1 Basic parallel operator ........................ 10
   4.2 Distributed parallel ............................ 10

5 Weak Conjunction Operator 12
   5.1 Distributed weak conjunction ................. 13

6 Concurrent Refinement Algebra 14

7 Galois Connections and Fusion Theorems 17
   7.1 Lower Galois connections ....................... 17
   7.2 Greatest fixpoint fusion theorems ............. 18
   7.3 Upper Galois connections ....................... 19
   7.4 Least fixpoint fusion theorems ............... 20

8 Iteration 21
   8.1 Possibly infinite iteration .................... 22
   8.2 Finite iteration ................................ 23
   8.3 Infinite iteration ................................ 25
   8.4 Combined iteration ................................ 26

9 Sequential composition for conjunctive models 27

10 Infimum nat lemmas 28

11 Iteration for conjunctive models 30

12 Rely Quotient Operator 36
   12.1 Basic rely quotient ............................ 36
   12.2 Distributed rely quotient ..................... 39

13 Conclusions 43
1 Overview

The theories provided here were developed in order to provide support for rely/guarantee concurrency [6, 5]. The theories provide a quite general concurrent refinement algebra that builds on a complete lattice of commands by adding sequential and parallel composition operators as well as recursion. A novel weak conjunction operator is also added as this allows one to build more general specifications. The theories are based on the paper by Hayes [3], however there are some differences that have been introduced to correct and simplify the algebra and make it more widely applicable. See the appendix for a summary of the differences.

The basis of the algebra is a complete lattice of commands (Section 2). Sections 3, 4 and 5 develop laws for sequential composition, parallel composition and weak conjunction, respectively, based on the refinement lattice. Section 6 brings the above theories together. Section 7 adds least and greatest fixed points and there associated laws, which allows finite, possibly infinite and strictly infinite iteration operators to be defined in Section 8 in terms of fixed points.

The above theories do not assume that sequential composition is conjunctive. Section 9 adds this assumption and derives a further set of laws for sequential composition and iterations.

Section 12 builds on the general theory to provide a rely quotient operator that can be used to provide a general rely/guarantee framework for reasoning about concurrent programs.

2 Refinement Lattice

theory Refinement-Lattice imports Main HOL-Library.Lattice-Syntax begin

The underlying lattice of commands is complete and distributive. We follow the refinement calculus tradition so that \( \sqcap \) is non-deterministic choice and \( c \sqsubseteq d \) means \( c \) is refined (or implemented) by \( d \).

declare [[show-sorts]]

Remove existing notation for quotient as it interferes with the rely quotient

no-notation Equiv-Relations.quotient (infixl "/'/'" 90)
class refinement-lattice = complete-distrib-lattice
begin

The refinement lattice infimum corresponds to non-deterministic choice for commands.

abbreviation refine :: 'a ⇒ 'a ⇒ bool (infix ⊑)
where
  c ⊑ d ≡ less-eq c d

abbreviation refine-strict :: 'a ⇒ 'a ⇒ bool (infix ⊏)
where
  c ⊏ d ≡ less c d

Non-deterministic choice is monotonic in both arguments

lemma inf-mono-left: a ⊑ b ⇒ a ⊓ c ⊑ b ⊓ c
  using inf-mono by auto

lemma inf-mono-right: c ⊑ d ⇒ a ⊓ c ⊑ a ⊓ d
  using inf-mono by auto

Binary choice is a special case of choice over a set.

lemma Inf2-inf: \{ f x \mid x. x ∈ \{c, d\}\} = f c ⊓ f d
proof
  have \{ f x \mid x. x ∈ \{c, d\}\} = \{f c, f d\} by blast
  then have \{ f x \mid x. x ∈ \{c, d\}\} = \{f c, f d\} by simp
  also have ... = f c ⊓ f d by simp
  finally show ?thesis .
qed

Helper lemma for choice over indexed set.

lemma INF-absorb-args: (∩ i j. (f::complete-lattice) (i + j)) = (∩ k. f k)
proof (rule order-class.order.antisym)
  show (∩ k. f k) ≤ (∩ i j. f (i + j))
    by (simp add: complete-lattice-class.le-INF-iff)
next
  have ∃ k. ∃ i j. f (i + j) ≤ f k

by (metis add.left-neutral order-class.eq-iff)
then have \( \bigwedge k. \exists i. (\bigcap j. f(i + j)) \leq f k \)
by (meson UNIV-I complete-lattice-class.INF-upper2)
then show \( (\bigcap i. j. f(i + j)) \leq (\bigcap k. f k) \)
by (simp add: complete-lattice-class.INF-mono)
qed

lemma (in −) nested-Collect: \( \{ f y \mid y \in \{ g x \mid x. x \in X \}\} = \{ f (g x) \mid x. x \in X \}\)
by blast

A transition lemma for INF distributivity properties, going from Inf to INF,
qualified version followed by a straightforward one.

lemma Inf-distrib-INF-qual:
fixes f :: ‘a ⇒ ‘a ⇒ ‘a
assumes qual: P \( \{ d x \mid x. x \in X \}\)
assumes f-Inf-distrib: \( \bigwedge c D. P D \implies f c (\bigcap D) = \bigcap \{ f c d \mid d. d \in D \}\)
shows f c (\( \bigcap x \in X. d x \)) = (\( \bigcap x \in X. f c (d x) \))
proof –
  have f c (\( \bigcap x \in X. d x \)) = f c (\( \bigcap \{ d x \mid x. x \in X \}\)) by (simp add: INF-Inf)
  also have ... = (\( \bigcap \{ f c dx \mid dx. dx \in \{ d x \mid x. x \in X \}\}\)) by (simp add: qual)
  also have ... = (\( \bigcap \{ f c (d x) \mid x. x \in X \}\)) by (simp only: nested-Collect)
  also have ... = (\( \bigcap x \in X. f c (d x) \)) by (simp add: INF-Inf)
  finally show ?thesis .
qed

lemma Inf-distrib-INF:
fixes f :: ‘a ⇒ ‘a ⇒ ‘a
assumes f-Inf-distrib: \( \bigwedge c D. f c (\bigcap D) = \bigcap \{ f c d \mid d. d \in D \}\)
shows f c (\( \bigcap x \in X. d x \)) = (\( \bigcap x \in X. f c (d x) \))
by (simp add: Setcompr-eq-image f-Inf-distrib image-comp)
end

lemmas refine-trans = order.trans

More transitivity rules to make calculational reasoning smoother
declare ord-le-eq-trans[trans]
declare ord-eq-le-trans[trans]
declare dual-order.order.trans[trans]
abbreviation
dist-over-sup :: ('a::refinement-lattice ⇒ 'a) ⇒ bool
where
dist-over-sup F ≡ (∀ X . F (∪ X) = (∪ x∈X. F (x)))

abbreviation
dist-over-inf :: ('a::refinement-lattice ⇒ 'a) ⇒ bool
where
dist-over-inf F ≡ (∀ X . F (⨌ X) = (⨌ x∈X. F (x)))

end

3 Sequential Operator

theory Sequential
imports Refinement-Lattice
begin

3.1 Basic sequential

The sequential composition operator “;” is associative and has identity nil but it is not commutative. It has ⊥ as a left annihilator.

locale seq =
  fixes seq :: 'a::refinement-lattice ⇒ 'a ⇒ 'a (infixl ; 90)
  assumes seq-bot [simp]: ⊥ ; c = ⊥

locale nil =
  fixes nil :: 'a::refinement-lattice (nil)

The monoid axioms imply “;” is associative and has identity nil. Abort is a left annihilator of sequential composition.

locale sequential = seq + nil + seq: monoid seq nil
begin

declare seq.assoc [algebra-simps, field-simps]

lemmas seq-assoc = seq.assoc
lemmas seq-nil-right = seq.right-neutral
lemmas seq-nil-left = seq.left-neutral

end
3.2 Distributed sequential

Sequential composition distributes across arbitrary infima from the right but only across the binary (finite) infima from the left and hence it is monotonic in both arguments. We consider left distribution first. Note that Section 9 considers the case in which the weak-seq-inf-distrib axiom is strengthened to an equality.

locale seq-distrib-left = sequential +
assumes weak-seq-inf-distrib:
(c::a::refinement-lattice);(d_0 ▪ d_1) ⊆ (c;d_0 ▪ c;d_1)
begin

Left distribution implies sequential composition is monotonic is its right argument

lemma seq-mono-right: c_0 ⊆ c_1 ==> d ; c_0 ⊆ d ; c_1
by (metis inf.absorb-iff2 le-inf-iff weak-seq-inf-distrib)

lemma seq-bot-right [simp]: c⊥ ⊆ c
by (metis bot.extremum seq.right-neutral seq-mono-right)

end

locale seq-distrib-right = sequential +
assumes Inf-seq-distrib:
(⨅ C) ; d = (∨ (c::a::refinement-lattice)∈C. c ; d)
begin

lemma INF-seq-distrib: (∨ c∈C. f c) ; d = (∨ c∈C. f c ; d)
using Inf-seq-distrib by (auto simp add: image-comp)

lemma inf-seq-distrib: (c_0 ▪ c_1) ; d = (c_0 ; d ▪ c_1 ; d)
proof
  have (c_0 ▪ c_1) ; d = (∨ {c_0, c_1}) ; d by simp
  also have ... = (∨ c∈{c_0, c_1}. c ; d) by (fact Inf-seq-distrib)
  also have ... = (c_0 ; d) ▪ (c_1 ; d) by simp
  finally show ?thesis .
qed

lemma seq-mono-left: c_0 ⊆ c_1 ==> c_0 ; d ⊆ c_1 ; d
by (metis inf.absorb-iff2 inf-seq-distrib)

lemma seq-top [simp]: ⊤ ; c = ⊤
proof —
  have \( \top : c = (\prod a \in \{\cdot\}. a ; c) \)
    by (metis Inf-empty Inf-seq-distrib)
  thus ?thesis
    by simp
qed

primrec seq-power :: \('a \Rightarrow \text{nat} \Rightarrow \text{'a}\)
  (infixr \(\cdot\) 80) where
  seq-power-0: \(a \cdot 0 = \text{nil}\)
  | seq-power-Suc: \(a \cdot \text{Suc } n = a ; (a \cdot n)\)

notation (latex output)
  seq-power \((\cdot) [1000] 1000\)
notation (HTML output)
  seq-power \((\cdot) [1000] 1000\)

lemma seq-power-front: \((a \cdot n) ; a = a ; (a \cdot n)\)
  by (induct n, simp-all add: seq-assoc)

lemma seq-power-split-less: \(i < j \Rightarrow (b \cdot j) = (b \cdot i) ; (b \cdot (j - i))\)
proof (induct j arbitrary: i type: nat)
  case 0
  thus ?case by simp
next
  case (Suc j)
  have \(b \cdot \text{Suc } j = b ; (b \cdot i) ; (b \cdot (j - i))\)
    using Suc.hyps Suc.prems less-Suc-eq seq-assoc by auto
  also have ... = \((b \cdot i) ; b ; (b \cdot (j - i))\) by (simp add: seq-power-front)
  also have ... = \((b \cdot i) ; (b \cdot (\text{Suc } j - i))\)
    using Suc.prems Suc-diff-le seq-assoc by force
  finally show ?case .
qed

locale seq-distrib = seq-distrib-right + seq-distrib-left
begin

lemma seq-mono: \(c_1 \subseteq d_1 \Rightarrow c_2 \subseteq d_2 \Rightarrow c_1 ; c_2 \subseteq d_1 ; d_2\)
  using seq-mono-left seq-mono-right by (metis inf.orderE le-infl2)
4 Parallel Operator

theory Parallel
imports Refinement-Lattice
begin

4.1 Basic parallel operator

The parallel operator is associative, commutative and has unit skip and has
as an annihilator the lattice bottom.

locale skip =
  fixes skip :: 'a::refinement-lattice (skip)
locale par =
  fixes par :: 'a::refinement-lattice ⇒ 'a ⇒ 'a (infixl || 75)
  assumes abort-par: ⊥ || c = ⊥
locale parallel = par + skip + par: comm-monoid par skip
begin

lemmas [algebra-simps, field-simps] =
  par.assoc
  par.commute
  par.left-commute
lemmas par-assoc = par.assoc
lemmas par-commute = par.commute
lemmas par-skip = par.right-neutral
lemmas par-skip-left = par.left-neutral
end

4.2 Distributed parallel

The parallel operator distributes across arbitrary non-empty infima.

locale par-distrib = parallel +
  assumes par-Inf-distrib: D ≠ {} ⟹ c || (∩ D) = (∩ d∈D. c || d)
begin

lemma Inf-par-distrib: \( D \neq \{\} \implies (\bigsqcap D) \parallel c = (\bigsqcap d \in D. d \parallel c) \)
using par-Inf-distrib par-commute by simp

lemma par-INF-distrib: \( X \neq \{\} \implies c \parallel (\bigcap x \in X. d x) = (\bigcap x \in X. c \parallel d x) \)
using par-Inf-distrib by (auto simp add: image-comp)

lemma INF-par-distrib: \( X \neq \{\} \implies \bigcap x \in X. d x \parallel c = \bigcap x \in X. d x \parallel c \)
using par-INF-distrib par-commute by (metis mono-tags, lifting) INF-cong

lemma INF-INF-par-distrib: \( X \neq \{\} \implies \bigcap Y \neq \{\} \implies \bigcap x \in X. c x \parallel \bigcap y \in Y. d y = \bigcap x \in X. \bigcap y \in Y. c x \parallel d y \)
proof
  assume nonempty-X: \( X \neq \{\} \)
  assume nonempty-Y: \( Y \neq \{\} \)
  have \( \bigcap x \in X. c x \parallel \bigcap y \in Y. d y = \bigcap x \in X. \bigcap y \in Y. c x \parallel d y \)
  using INF-par-distrib by (metis nonempty-X)
  also have \( \ldots \)
  using par-INF-distrib by (metis nonempty-Y)
  thus ?thesis by (simp add: calculation)
qed

lemma inf-par-distrib: \( (c_0 \sqcap c_1) \parallel d = (c_0 \parallel d) \cap (c_1 \parallel d) \)
proof
  have \( (c_0 \sqcap c_1) \parallel d = (\bigcap \{c_0, c_1\}) \parallel d \) by simp
  also have \( \ldots \)
  using Inf-par-distrib by (meson insert-not-empty)
  also have \( \ldots = c_0 \parallel d \cap c_1 \parallel d \) by simp
  finally show ?thesis .
qed

lemma inf-par-distrib2: \( d \parallel (c_0 \sqcap c_1) = (d \parallel c_0) \cap (d \parallel c_1) \)
using inf-par-distrib par-commute by auto

lemma inf-par-product: \( (a \sqcap b) \parallel (c \sqcap d) = (a \parallel c) \cap (a \parallel d) \cap (b \parallel c) \cap (b \parallel d) \)
by (simp add: inf-commute inf-par-distrib inf-par-distrib2 inf-sup-aci(3))

lemma par-mono: \( c_1 \subseteq d_1 \implies c_2 \subseteq d_2 \implies c_1 \parallel c_2 \subseteq d_1 \parallel d_2 \)
by (metis inf.orderE le-inf-iff order-refl inf-par-distrib par-commute)
5 Weak Conjunction Operator

theory Conjunction
imports Refinement-Lattice
begin

The weak conjunction operator \( \sqcap \) is similar to least upper bound (\( \sqcup \)) but is abort strict, i.e. the lattice bottom is an annihilator: \( c \sqcap \bot = \bot \). It has identity the command chaos that allows any non-aborting behaviour.

locale chaos = fixes chaos :: 'a::refinement-lattice (chaos)
locale conj = fixes conj :: 'a::refinement-lattice \Rightarrow 'a => 'a (infixl \( \sqcap \) 80) assumes conj-bot-right: \( c \sqcap \bot = \bot \)
Conjunction forms an idempotent, commutative monoid (i.e. a semi-lattice), with identity chaos.
locale conjunction = conj + chaos + conj: semilattice-neutr conj chaos

begin
lemmas [algebra-simps, field-simps] =
 conj.assoc
 conj.commute
 conj.left-commute

lemmas conj-assoc = conj.assoc
lemmas conj-commute = conj.commute
lemmas conj-idem = conj.idem
lemmas conj-chaos = conj.right-neutral
lemmas conj-chaos-left = conj.left-neutral

lemma conj-bot-left [simp]: \( \bot \sqcap c = \bot \)
using conj-bot-right local.conj-commute by fastforce

lemma conj-not-bot: \( a \sqcap b \neq \bot \Rightarrow a \neq \bot \land b \neq \bot \)
using conj-bot-right by auto

lemma conj-distrib1: \( c \sqcap (d_0 \sqcap d_1) = (c \sqcap d_0) \sqcap (c \sqcap d_1) \)
by (metis conj-assoc conj-commute conj-idem)

end

5.1 Distributed weak conjunction

The weak conjunction operator distributes across arbitrary non-empty in-
fima.

locale conj-distrib =
  assumes Inf-conj-distrib: \( D \neq \{\} \Rightarrow (\bigsqcap d \in D. d \cap c) \)

begin

lemma conj-Inf-distrib: \( D \neq \{\} \Rightarrow c \cap (\bigsqcap d \in D. d) = (\bigsqcap d \in D. \ (c \cap d)) \)
  using Inf-conj-distrib conj-commute by auto

lemma inf-conj-distrib: \( (c_0 \cap c_1) \cap d = (c_0 \cap d) \cap (c_1 \cap d) \)
  proof
  have \( (c_0 \cap c_1) \cap d = (\bigsqcap \{c_0, c_1\} \cap d) \) by simp
  also have \( \ldots = (\bigsqcap c \in \{c_0, c_1\}. c \cap d) \) by (rule Inf-conj-distrib, simp)
  also have \( \ldots = ((c_0 \cap d) \cap (c_1 \cap d)) \) by simp
  finally show ?thesis .
  qed

lemma inf-conj-product: \( (a \cap b) \cap (c \cap d) = (a \cap c) \cap (a \cap d) \cap (b \cap c) \cap (b \cap d) \)
  by (metis inf-conj-distrib conj-commute inf-assoc)

lemma conj-mono: \( c_0 \subseteq d_0 \Rightarrow c_1 \subseteq d_1 \Rightarrow (c_0 \cap c_1) \subseteq d_0 \cap d_1 \)
  by (metis inf.absorb_iff1 inf-conj-product inf-right-idem)

lemma conj-mono-left: \( c_0 \subseteq c_1 \Rightarrow c_0 \cap d \subseteq c_1 \cap d \)
  by (simp add: conj-mono)

lemma conj-mono-right: \( c_0 \subseteq c_1 \Rightarrow d \cap c_0 \subseteq d \cap c_1 \)
  by (simp add: conj-mono)

lemma conj-refine: \( c_0 \subseteq d \Rightarrow c_1 \subseteq d \Rightarrow c_0 \cap c_1 \subseteq d \)
  by (metis conj-idem conj-mono)

lemma refine-to-conj: \( c \subseteq d_0 \Rightarrow c \subseteq d_1 \Rightarrow c \subseteq d_0 \cap d_1 \)
  by (metis conj-idem conj-mono)

lemma conjoin-non-aborting: \( \text{chaos} \subseteq c \Rightarrow d \subseteq d \cap c \)
by (metis conj-mono order refl conj-chaos)

lemma conjunction-sup: \( c \land d \sqsubseteq c \sqcup d \)
by (simp add: conj-refine)

lemma conjunction-sup-nonaborting:
  assumes chaos \( \sqsubseteq c \) and chaos \( \sqsubseteq d \)
  shows \( c \land d = c \sqcup d \)
proof (rule antisym)
  show \( c \sqcup d \sqsubseteq c \land d \) using assms (1) assms (2) converse(conj-non-aborting) local.conj-commute
  by fastforce
next
  show \( c \land d \sqsubseteq c \sqcup d \) by (metis conjunction-sup)
qed

lemma conjoin-top: chaos \( \sqsubseteq c \Rightarrow c \land \top = \top \)
by (simp add: conjunction-sup-nonaborting)

end

end

6 Concurrent Refinement Algebra

This theory brings together the three main operators: sequential composition, parallel composition and conjunction, as well as the iteration operators.

theory CRA
imports
  Sequential
  Conjunction
  Parallel
begin
Locale sequential-parallel brings together the sequential and parallel operators and relates their identities.
locale sequential-parallel =
  assumes nil-par-nil: \( \text{nil} \parallel \text{nil} \sqsubseteq \text{nil} \)
  and skip-nil: \( \text{skip} \sqsubseteq \text{nil} \)
  and skip-skip: \( \text{skip} \sqsubseteq \text{skip} \)
begin

lemma nil-absorb: \( \text{nil} \parallel \text{nil} = \text{nil} \) using nil-par-nil skip-nil par-skip

end

end
by (metis inf.absorb-iff2 inf.orderE inf-par-distrib2)

lemma skip-absorb [simp]: skip;skip = skip
by (metis antisym seq-mono-right seq-nil-right skip-skip skip-nil)

end

Locale conjunction-parallel brings together the weak conjunction and parallel operators and relates their identities. It also introduces the interchange axiom for conjunction and parallel.

locale conjunction-parallel = conj-distrib + par-distrib +
  assumes chaos-par-top: ⊤ ⊑ chaos || ⊤
  assumes chaos-par-chaos: chaos ⊑ chaos || chaos
  assumes parallel-interchange: (c0 || c1) ∩ (d0 || d1) ⊑ (c0 ∩ d0) || (c1 ∩ d1)
begin

lemma chaos-skip: chaos ⊑ skip
proof −
  have chaos = (chaos || skip) ∩ (skip || chaos) by simp
  then have . . . ⊑ (chaos ∩ skip) || (skip ∩ chaos) using parallel-interchange by blast
  thus ?thesis by auto
qed

lemma chaos-par-chaos-eq: chaos = chaos || chaos
  by (metis antisym chaos-par-chaos chaos-skip order-refl par-mono par-skip)

lemma nonabort-par-top: chaos ⊑ c ⇒ c || ⊤ = ⊤
  by (metis chaos-par-top par-mono top.extremum-uniqueI)

lemma skip-conj-top: skip ⊑ ⊤ = ⊤
by (simp add: chaos-skip conjoin-top)

lemma conj-distrib2: c ⊑ c || c ⇒ c ∩ (d0 || d1) ⊑ (c ∩ d0) || (c ∩ d1)
proof −
  assume c ⊑ c || c
  then have c ∩ (d0 || d1) ⊑ (c || c) ∩ (d0 || d1) by (metis conj-mono order.refl)
  thus ?thesis by (metis parallel-interchange refine-trans)
qed

end

Locale conjunction-sequential brings together the weak conjunction and sequential operators. It also introduces the interchange axiom for conjunction
and sequential.

locale conjunction-sequential = conj-distrib + seq-distrib +
  assumes chaos-seq-chaos: chaos ⊆ chaos;chaos
  assumes sequential-interchange: (c0;c1) ⨃ (d0;d1) ⊆ (c0 ⨃ d0);(c1 ⨃ d1)
begin

lemma chaos-nil: chaos ⊆ nil
  by (metis conj-chaos local.conj-commute seq-nil-left seq-nil-right
      sequential-interchange)

lemma chaos-seq-absorb: chaos = chaos;chaos
  proof (rule antisym)
    show chaos ⊆ chaos;chaos by (simp add: chaos-seq-chaos)
  next
    show chaos;chaos ⊆ chaos using chaos-nil
      using seq-mono-left seq-nil-left by fastforce
  qed

lemma seq-bot-conj: c;⊥ ⨃ d ⊆ (c ⨃ d);⊥
  by (metis (no-types) conj-bot-left seq-nil-right sequential-interchange)

lemma conj-seq-bot-right [simp]: c;⊥ ⨃ c = c;⊥
  proof (rule antisym)
    show lr: c;⊥ ⨃ c ⊆ c;⊥ by (metis seq-bot-conj conj-idem)
  next
    show rl: c;⊥ ⊆ c;⊥ ⨃ c
      by (metis conj-idem conj-mono-right seq-bot-right)
  qed

lemma conj-distrib3: c ⊆ c;c ⟹ c ⨃ (d0 ; d1) ⊆ (c ⨃ d0);(c ⨃ d1)
  proof
    assume c ⊆ c;c
    then have c ⨃ (d0;d1) ⊆ (c;c) ⨃ (d0;d1) by (metis conj-mono order.refl)
    thus ?thesis by (metis sequential-interchange refine-trans)
  qed

end

Locale cra brings together sequential, parallel and weak conjunction.
locale cra = sequential-parallel + conjunction-parallel + conjunction-sequential
7 Galois Connections and Fusion Theorems

The concept of Galois connections is introduced here to prove the fixed-point fusion lemmas. The definition of Galois connections used is quite simple but encodes a lot of information. The material in this section is largely based on the work of the Eindhoven Mathematics of Program Construction Group [1] and the reader is referred to their work for a full explanation of this section.

7.1 Lower Galois connections

lemma Collect-2set [simp]: \{F x \mid x = a \lor x = b\} = \{F a, F b\}
by auto

locale lower-galois-connections
begin

definition l-adjoint :: ('a::refinement-lattice \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) (-♭ [201] 200)
where
(F♭) x = \bigcap\{y. x \subseteq F y\}

lemma dist-inf-mono:
  assumes distF: dist-over-inf F
  shows mono F
proof
  fix x :: 'a and y :: 'a
  assume x \subseteq y
  then have F x = F (x \cap y) by (simp add: le-iff-inf)
  also have ... = F x \cap F y
  proof
    from distF
    have F (\bigcap \{x, y\}) = \bigcap \{F x, F y\} by (drule-tac x = \{x, y\} in spec, simp)
    then show F (x \cap y) = F x \cap F y by simp
  qed
  finally show F x \subseteq F y by (metis le-iff-inf)
qed
lemma l-cancellation: \( F \) \( \Rightarrow \) \( x \subseteq \text{dist-over-inf } F \) \( \Rightarrow \) \( x \subseteq (F \circ F^\circ) x \)
proof
assume dist: \( \text{dist-over-inf } F \)

define \( Y \) where \( Y = \{ F y \mid y \cdot x \subseteq F y \} \)
define \( X \) where \( X = \{ x \} \)

have \( (\forall y \in Y. (\exists x \in X. x \subseteq y)) \) using X-def Y-def CollectD singletonI by auto
then have \( \prod X \subseteq \prod Y \) by (simp add: Inf-mono)
then have \( x \subseteq \prod \{ F y \mid y \cdot x \subseteq F y \} \) by (simp add: X-def Y-def)
then have \( x \subseteq F (\prod \{ y. x \subseteq F y \}) \) by (simp add: dist le-INF-iff)
thus ?thesis by (metis comp-def l-adjoint-def)
qed

lemma l-galois-connection: \( \text{dist-over-inf } F \Rightarrow (F^\flat x \leq y) \iff (x \leq F y) \)
proof
assume \( x \leq F y \)
then have \( \prod \{ y. x \subseteq F y \} \subseteq y \) by (simp add: Inf-lower)
thus \( (F^\flat) x \subseteq y \) by (metis l-adjoint-def)
next
assume dist: \( \text{dist-over-inf } F \) then have monoF: \( \text{mono } F \) by (simp add: dist-inf-mono)
assume \( (F^\flat) x \leq y \) then have a: \( F (F^\flat x) \leq F y \) by (simp add: monoD monoF)
have \( x \leq F (F^\flat x) \) using dist l-cancellation by simp
thus \( x \leq F y \) using a by auto
qed

lemma v-simple-fusion: \( \text{mono } G \Rightarrow \forall x. ((F \circ G) x \leq (H \circ F) x) \Rightarrow F (\text{gfp } G) \leq \text{gfp } H \)
by (metis comp-eq-dest-lhs gfp-unfold gfp-upperbound)

7.2 Greatest fixpoint fusion theorems

Combining lower Galois connections and greatest fixed points allows elegant proofs of the weak fusion lemmas.

theorem fusion-gfp-geq:
  assumes monoH: \( \text{mono } H \)
  and distribF: \( \text{dist-over-inf } F \)
  and comp-geq: \( \forall x. ((H \circ F) x \subseteq (F \circ G) x) \Rightarrow F (\text{gfp } G) \subseteq \text{gfp } H \)
  shows gfp H \( \subseteq F (\text{gfp } G) \)
proof
  have \( (\text{gfp } H) \subseteq (F \circ F^\circ) (\text{gfp } H) \) using distribF l-cancellation by simp
  then have \( H (\text{gfp } H) \subseteq H ((F \circ F^\circ) (\text{gfp } H)) \) by (simp add: monoD monoH)
  then have \( \text{gfp } H \subseteq \text{gfp } H \) by (simp add: dist l-cancellation)
  thus ?thesis by (simp add: gfp-unfold)
qed
then have $H \ (\text{gfp} \ H) \subseteq F \ ((G \circ F^\flat) \ (\text{gfp} \ H))$ using comp-geq by (metis comp-def refine-trans)
then have $(F^\flat) \ (H \ (\text{gfp} \ H)) \subseteq (G \circ F^\flat) \ (\text{gfp} \ H)$ using distribF by (metis (mono-tags) l-galois-connection)
then have $(F^\flat) \ (\text{gfp} \ H) \subseteq (\text{gfp} \ G)$ by (metis comp-apply gfp-unfold gfp-upperbound monoH)
thus $\text{gfp} \ H \subseteq F \ (\text{gfp} \ G)$ using distribF by (metis (mono-tags) l-galois-connection)
qed

**Theorem fusion-gfp-eq:**

assumes monoH: mono $H$ and monoG: mono $G$
and distF: dist-over-inf $F$
and fgh-comp: $\forall x. ((F \circ G) \ x = (H \circ F) \ x)$
shows $F \ (\text{gfp} \ G) = \text{gfp} \ H$
proof (rule antisym)

show $F \ (\text{gfp} \ G) \subseteq (\text{gfp} \ H)$ by (metis fgh-comp le-less v-simple-fusion monoG)
next

have $\forall x. ((H \circ F) \ x \subseteq (F \circ G) \ x)$ using fgh-comp by auto
then show $\text{gfp} \ H \subseteq F \ (\text{gfp} \ G)$ using monoH distF fusion-gfp-geq by blast
qed

end

**7.3 Upper Galois connections**

locale upper-galois-connections
begin

**Definition**

u-adjoint :: $'(\text{a}::\text{refinement-lattice} \Rightarrow \ 'a) \Rightarrow (\ 'a \Rightarrow 'a) \ (# \ [201] \ 200)\$
where

$(F^\#) \ x \equiv \bigcup \{y. \ F \ y \subseteq x\}$

**Lemma dist-sup-mono:**

assumes distF: dist-over-sup $F$
sows mono $F$
proof
fix $x :: 'a$ and $y :: 'a$
assume $x \subseteq y$
then have $F \ y = F \ (x \sqcup y)$ by (simp add: le-iff-sup)
also have $... = F \ x \sqcup F \ y$
proof −
from distF
have F (⨆ \{x, y\}) = ⨆ \{F x, F y\} by (drule-tac x = \{x, y\} in spec, simp)
then show F (x ⊔ y) = F x ⊔ F y by simp
qed
finally show F x ⊑ F y by (metis le-iff-sup)
qed

lemma u-cancellation: dist-over-sup F ⇒ (F ◦ F#) x ⊑ x
proof −
assume dist: dist-over-sup F
define Y where Y = \{y. F y ⊑ x\}
define X where X = \{x\}

have (∀ y ∈ Y. (∃ x ∈ X. y ⊑ x)) using X-def Y-def CollectD singletonI by auto
then have ⨆ Y ⊑ ⨆ X by (simp add: Sup-mono)
then have ⨆ {y. F y ⊑ x} ⊑ x by (simp add: X-def Y-def)
then have F (⨆ \{y. F y ⊑ x\}) ⊑ x using SUP-le-iff dist by fastforce
thus ?thesis by (metis comp-def u-adjoint-def)
qed

lemma u-galois-connection: dist-over-sup F ⇒ (F x ⊑ y) ←→ (x ⊑ (F#) y)
proof
assume dist: dist-over-sup F then have monoF: mono F by (simp add: dist-sup-mono)
assume x ⊑ (F#) y then have a: F x ⊑ F ((F#) y) by (simp add: monoD monoF)
have F ((F#) y) ⊑ y using dist u-cancellation by simp
thus F x ⊑ y using a by auto
next
assume F x ⊑ y
then have x ⊑ ⨆ \{F x ⊑ y\} by (simp add: Sup-upper)
thus x ⊑ (F#) y by (metis u-adjoint-def)
qed

lemma u-simple-fusion: mono H ⇒ ∀ x. ((F ◦ G) x ⊑ (G ◦ H) x) ⇒ lfp F ⊑ G (lfp H)
by (metis comp-def lfp-lowerbound lfp-unfold)

7.4 Least fixpoint fusion theorems
Combining upper Galois connections and least fixed points allows elegant proofs of the strong fusion lemmas.

theorem fusion-lfp-leq:
  assumes monoH: mono H
and distribF: dist-over-sup F
and comp-leq: \( \forall x. \ ((F \circ G) \cdot x \subseteq (H \circ F) \cdot x) \)
shows \( F \cdot (\text{lfp } G) \subseteq (\text{lfp } H) \)
proof –
  have \( ((F \circ F^\#) \cdot (\text{lfp } H)) \subseteq \text{lfp } H \) using distribF u-cancellation by simp
  then have \( H \cdot ((F \circ F^\#) \cdot (\text{lfp } H)) \subseteq H \cdot (\text{lfp } H) \) by (simp add: monoD monoH)
  then have \( F \cdot ((G \circ F^\#) \cdot (\text{lfp } H)) \subseteq H \cdot (\text{lfp } H) \) using comp-leq by (metis comp-def refine-trans)
  then have \( (G \circ F^\#) \cdot (\text{lfp } H) \subseteq (F^\#) \cdot (H \cdot (\text{lfp } H)) \) using distribF by (metis mono-tags u-galois-connection)
  then have \( (\text{lfp } G) \subseteq (F^\#) \cdot (H \cdot (\text{lfp } H)) \) by (metis comp-def-def-lfp-unfold lfp-lowerbound monoH)
  thus \( F \cdot (\text{lfp } G) \subseteq (\text{lfp } H) \) using distribF by (metis (mono-tags) u-galois-connection)
qed

theorem fusion-lfp-eq:
  assumes monoH: Mono H and monoG: Mono G
  and distF: dist-over-sup F
  and fgh-comp: \( \forall x. \ ((F \circ G) \cdot x = (H \circ F) \cdot x) \)
  shows \( F \cdot (\text{lfp } G) = (\text{lfp } H) \)
proof (rule antisym)
  show \( \text{lfp } H \subseteq F \cdot (\text{lfp } G) \) by (metis monoG fgh-comp eq-iff upper-galois-connections.u-simple-fusion)
next
  have \( \forall x. \ (F \circ G) \cdot x \subseteq (H \circ F) \cdot x \) using fgh-comp by auto
  then show \( F \cdot (\text{lfp } G) \subseteq (\text{lfp } H) \) using monoH distF fusion-lfp-leq by blast
qed

8 Iteration

theory Iteration
imports
  Galois-Connections
  CRA
begin
8.1 Possibly infinite iteration

Iteration of finite or infinite steps can be defined using a least fixed point.

locale finite-or-infinite-iteration = seq-distrib + upper-galois-connections
begin

definition
iter :: 'a ⇒ 'a ("\omega [103] 102")
where
\(c^\omega \equiv \text{lfp} (\lambda x. \text{nil} \sqcap c;x)\)

lemma iter-step-mono: mono (\lambda x. \text{nil} \sqcap c;x)
  by (meson inf-mono order-refl seq-mono-right mono-def)

This fixed point definition leads to the two core iteration lemmas: folding and induction.

theorem iter-unfold: \(c^\omega = \text{nil} \sqcap c;c^\omega\)
  using iter-def iter-step-mono lfp-unfold by auto

lemma iter-induct-nil: \(\text{nil} \sqcap c;x \sqsubseteq x \Rightarrow c^\omega \sqsubseteq x\)
  by (simp add: iter-def lfp-lowerbound)

lemma iter0: \(c^\omega \sqsubseteq \text{nil}\)
  by (metis iter-unfold sup.orderI sup-inf-absorb)

lemma iter1: \(c^\omega \sqsubseteq c\)
  by (metis inf-le2 iter0 iter-unfold order.trans seq-mono-right seq-nil-right)

lemma iter2 [simp]: \(c^\omega;c^\omega = c^\omega\)
proof (rule antisym)
  show \(c^\omega;c^\omega \sqsubseteq c^\omega\) using iter0 seq-mono-right by fastforce
  next
    have a: \(\text{nil} \sqcap c;c^\omega \sqsubseteq \text{nil} \sqcap c;c^\omega \sqcap c;c^\omega;c^\omega\)
      by (metis inf-greatest inf-le2 inf-mono iter0 order-refl seq-distrib-left.seq-mono-right seq-distrib-left-axioms seq-nil-right)
    then have b: \(\ldots = c^\omega \sqcap c;c^\omega;c^\omega\) using iter-unfold by auto
    then have c: \(\ldots = (\text{nil} \sqcap c;c^\omega);c^\omega\)
      by (simp add: inf-seq-distrib)
    thus \(c^\omega \sqsubseteq c^\omega;c^\omega\)
      using a iter-induct-nil iter-unfold seq-assoc by auto
  qed

lemma iter-mono: \(c \sqsubseteq d \Rightarrow c^\omega \sqsubseteq d^\omega\)
proof
  assume c \(\sqsubseteq d\)

22
then have \( \text{nil} \sqcap c; d^\omega \sqsubseteq d; d^\omega \) by (metis\ absorb-iff \( \text{inf-left-commute} \) \( \text{inf-seq-distrib} \))
then have \( \text{nil} \sqcap c; d^\omega \sqsubseteq d^\omega \) by (metis\ bounded-iff \( \text{inf-sup-ord(1)} \) \( \text{iter-unfold} \))
thus ?thesis by (simp add: \( \text{iter-induct-nil} \))
qed

lemma iter-abort: \( \bot = \text{nil}^\omega \)
  by (simp add: antisym \( \text{iter-induct-nil} \))

lemma nil-iter: \( \top = \text{nil} \)
  by (metis (no-types) \( \text{inf-top} \) \( \text{right-neutral} \) \( \text{iter-unfold} \) \( \text{seq-top} \))
end

8.2 Finite iteration

Iteration of a finite number of steps (Kleene star) is defined using the greatest fixed point.
locale finite-iteration = seq-distrib + lower-galois-connections
begin
definition fiter :: 'a ⇒ 'a (⋆) [101]
where
  \( c^\star \equiv \text{gfp} (\lambda x. \text{nil} \sqcap c;x) \)

lemma fin-iter-step-mono: mono (\( \lambda x. \text{nil} \sqcap c;x \))
  by (meson \( \text{inf-mono} \) \( \text{order-refl} \) \( \text{seq-mono-right} \) mono-def)

This definition leads to the two core iteration lemmas: folding and induction.

lemma fiter-unfold: \( c^\star = \text{nil} \sqcap c;c^\star \)
  using fiter-def \( \text{gfp-unfold} \) \( \text{fin-iter-step-mono} \) by auto

lemma fiter-induct-nil: \( x \sqsubseteq \text{nil} \sqcap c;x \implies x \sqsubseteq c^\star \)
  by (simp add: fiter-def \( \text{gfp-upperbound} \))

lemma fiter0: \( c^\star \sqsubseteq \text{nil} \)
  by (metis fiter-unfold \( \text{inf-cobounded1} \))

lemma fiter1: \( c^\star \sqsubseteq c \)
  by (metis fiter0 fiter-unfold \( \text{inf-le2} \) \( \text{order.trans} \) \( \text{seq-mono-right} \) \( \text{seq-nil-right} \))

lemma fiter-induct-eq: \( c^\star;d = \text{gfp} (\lambda x. c;x \sqcap d) \)
proof
  define F where F = (λ x. x;d)
  define G where G = (λ x. nil □ c;x)
  define H where H = (λ x. c;x □ d)

  have FG: F ◦ G = (λ x. c;x;d □ d) by (simp add: F-def G-def comp-def inf-commute inf-seq-distrib)
  have HF: H ◦ F = (λ x. c;x;d □ d) by (metis comp-def seq-assoc H-def F-def)

  have adjoint: dist-over-inf F using Inf-seq-distrib F-def by simp
  have monoH: mono H
    by (metis H-def inf-mono-left monoI seq-distrib-left seq-mono-right seq-distrib-left-axioms)
  have monoG: mono G by (metis G-def inf-mono-right mono-def seq-mono-right)
  have ∀ x. ((F ◦ G) x = (H ◦ F) x) using FG HF by simp
  then have F (gfp G) = gfp H using adjoint monoG monoH fusion-gfp-eq by blast

  then have (gfp (λ x. nil □ c;x));d = gfp (λ x. c;x □ d) using F-def G-def H-def inf-commute by simp
    thus ?thesis by (metis fiter-def)
qed

theorem fiter-induct: x ⊑ d □ c;x ⇒ x ⊑ c* □ d
proof
  assume x ⊑ d □ c;x
  then have x ⊑ c;x □ d using inf-commute by simp
  then have x ⊑ gfp (λ x. c;x □ d) by (simp add: gfp-upperbound)
  thus ?thesis by (metis (full-types) fiter-induct-eq)
qed

lemma fiter2 [simp]: c*;c* = c*
proof
  have lr: c*;c* ⊑ c* using fiter0 seq-mono-right seq-nil-right by fastforce
  have rl: c* ⊑ c*;c* by (metis fiter-induct fiter-unfold inf right-idem order-refl)
  thus ?thesis by (simp add: antisym lr)
qed

lemma fiter3 [simp]: (c*)* = c*
  by (metis dual-order.refl fiter0 fiter1 fiter2 fiter-induct commute inf-absorb1 seq-nil-right)

lemma fiter-mono: c ⊑ d ⇒ c* ⊑ d*
proof
  assume c ⊑ d

24
then have $c \sqsubseteq \text{nil} \sqcap d; c$ by (metis fiter0 fiter1 fiter2 inf.bounded-iff refine-trans seq-mono-left)
thus ?thesis by (metis seq-nil-right fiter-induct)
qed

end

8.3 Infinite iteration

Iteration of infinite number of steps can be defined using a least fixed point.
locale infinite-iteration = seq-distrib + lower-galois-connections
begin

definition
infiter :: `'a ⇒ 'a (¬∞ [105] 106)
where
$c^\infty$ ≡ lfp ($\lambda$ x. $c; x$)

lemma infiter-step-mono: mono ($\lambda$ x. $c; x$)
  by (meson inf-mono order-refl seq-mono-right mono-def)

This definition leads to the two core iteration lemmas: folding and induction.

theorem infiter-unfold: $c^\infty = c; c^\infty$
  using infiter-def infiter-step-mono lfp-unfold by auto

lemma infiter-induct: $c; x \sqsubseteq x$ ⇒ $c^\infty \sqsubseteq x$
  by (simp add: infiter-def lfp-lowerbound)

theorem infiter-unfold-any: $c^\infty = (c <^\infty i) ; c^\infty$
proof (induct i)
  case 0
  thus ?case by simp
next
  case (Suc i)
  thus ?case using infiter-unfold seq-assoc seq-power-Suc by auto
qed

lemma infiter-annil: $c^\infty;x = c^\infty$
proof
  have $\forall a. (\bot::'a) \sqsubseteq a$
    by auto
  thus ?thesis

25
by (metis (no-types) eq-iff inf.cobounded2 infiter-induct infiter-unfold inf-sup-ord(1)
seq-assoc seq-bot weak-seq-inf-distrib seq-nil-right)
qed
end

8.4 Combined iteration

The three different iteration operators can be combined to show that finite
iteration refines finite-or-infinite iteration.

locale iteration =
  finite-or-infinite-iteration + finite-iteration +
infinite-iteration
begin

lemma refine-iter:
  \( c^\omega \sqsubseteq c^* \)
  by (metis seq-nil-right order.refl iter-unfold fiter-induct)

lemma iter-absorption [simp]:
  \( (c^\omega)^* = c^\omega \)
  proof
    show \( (c^\omega)^* \sqsubseteq c^\omega \) by (metis fiter1)
  next
    show \( c^\omega \sqsubseteq (c^\omega)^* \) by (metis fiter1 fiter-induct inf-left-idem iter2 iter-unfold seq-nil-right
sup.cobounded2 sup.orderE sup-commute)
  qed

lemma infiter-inf-top:
  \( c^\infty = c^\omega ; \top \)
  proof
    have lr:
      \( c^\infty \sqsubseteq c^\omega ; \top \)
      proof
        have c:
          \( (c^\omega ; \top) = \text{nil} ; \top \cap c ; c^\omega ; \top \)
          using semigroup.assoc seq.semigroup-axioms by fastforce
        then show ?thesis
          by (metis (no-types) eq-refl finite-or-infinite-iteration.iter-unfold
finite-or-infinite-iteration-axioms infiter-induct
seq-distrib-right.inf-seq-distrib seq-distrib-right-axioms)
      qed
    have rl:
      \( c^\omega ; \top \sqsubseteq c^\infty \)
      proof
        have lr:
          \( c^\omega ; \top \sqsubseteq c^\infty \)
          by (metis inf-le2 infiter-annil infiter-induct-nil seq-monotone)
        then show ?thesis using antisym-conv lr by blast
      qed

lemma infiter-fiter-top:
  shows c^\infty \sqsubseteq c^* ; \top
by (metis eq-iff fiter-induct inf-top-left infiter-unfold)

lemma inf-ref-infiter: \(c^\omega \subseteq c^\infty\)
using infiter-unfold iter-induct-nil by auto

end

end

9 Sequential composition for conjunctive models

theory Conjunctive-Sequential
imports Sequential
begin

Sequential left-distributivity is only supported by conjunctive models but
does not apply in general. The relational model is one such example.

locale seq-finite-conjunctive = seq-distrib-right +
  assumes seq-inf-distrib: \(c \cdot (d_0 \cap d_1) = c \cdot d_0 \cap c \cdot d_1\)
begin

sublocale seq-distrib-left
  by (simp add: seq-distrib-left.intro seq-distrib-left-axioms.intro
      seq-inf-distrib sequential-axioms)
end

locale seq-infinite-conjunctive = seq-distrib-right +
  assumes seq-Inf-distrib: \(D \neq \{\} \Rightarrow c \cdot (\bigcap d \in D. c \cdot d)\)
begin

sublocale seq-distrib
proof unfold-locales
  fix c::'a and d0::'a and d1::'a
  have \(\{d_0, d_1\} \neq \{\}\) by simp
  then have c \cdot (\bigcap \{d_0, d_1\}) = \bigcap \{c \cdot d | d \in \{d_0, d_1\}\} using seq-Inf-distrib
  proof
    have \(\bigcap (\cdot \{d_0, d_1\}) = \bigcap \{c \cdot a | a \in \{d_0, d_1\}\}\) using INF-Inf by blast
    then show thesis
      using \(\bigcap (c::'a::refinement-lattice) D::'a::refinement-lattice\in D. c \cdot d)\)
      \(\{d_0::'a::refinement-lattice, d_1::'a::refinement-lattice\} \neq \{\}\) by presburger

27
also have ... = c ; d0 \cap c ; d1 by (simp only: Inf2-inf)
finally show c ; (d0 \cap d1) \subseteq c ; d0 \cap c ; d1 by simp
qed

lemma seq-INF-distrib: X \neq \{\} \Longrightarrow c ; (\bigcap x \in X. d x) = (\bigcap x \in X. c ; d x)
proof -
  assume xne: X \neq \{
  have a: c ; (\bigcap x \in X. d x) = c ; \bigcap (d \setminus X) by auto
  also have b: ... = (\bigcap d \in (d \setminus X). c ; d) by (meson image-is-empty seq-Inf-distrib xne)
  also have c: ... = (\bigcap x \in X. c ; d x) by (simp add: image-comp)
  finally show ?thesis by (simp add: b image-comp)
qed

lemma seq-INF-distrib-UNIV: c ; (\bigcap x. d x) = (\bigcap x. c ; d x)
  by (simp add: seq-INF-distrib)

lemma INF-INF-seq-distrib: Y \neq \{\} \Longrightarrow (\bigcap x \in X. c x) ; (\bigcap y \in Y. d y) = (\bigcap x \in X. \bigcap y \in Y. c x ; d y)
  by (simp add: INF-seq-distrib seq-INF-distrib)

lemma INF-INF-seq-distrib-UNIV: (\bigcap x. c x) ; (\bigcap y. d y) = (\bigcap x. \bigcap y. c x ; d y)
  by (simp add: INF-INF-seq-distrib)

end

end

10 Infimum nat lemmas

theory Infimum-Nat
imports
  Refinement-Lattice
begin

locale infimum-nat
begin

lemma INF-partition-nat3:
  fixes f :: nat \Rightarrow nat \Rightarrow 'a::refinement-lattice
  shows (\bigcap j. f i j) =
\[(\bigsqcap j \in \{i \cdot i = j\}. f i j) \sqcap (\bigsqcap j \in \{i \cdot i < j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot j < i\}. f i j)\]

proof –

have univ-part: \(\text{UNIV} = \{j \cdot i = j\} \cup \{j \cdot i < j\} \cup \{j \cdot j < i\}\) by auto

have \(\bigsqcap j \in \{j \cdot i = j\} \cup \{j \cdot i < j\} \cup \{j \cdot j < i\}. f i j\) =

\((\bigsqcap j \in \{j \cdot i = j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot i < j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot j < i\}. f i j)\)

by (metis \text{INF-union})

with univ-part show \?thesis by simp

qed

lemma \text{INF-INF-partition-nat3}:

fixes \(f::\text{nat} \Rightarrow \text{nat} \Rightarrow 'a::\text{refinement-lattice}\)

shows \((\bigsqcap i. \bigsqcap j. f i j) = (\bigsqcap i. ((\bigsqcap j \in \{j \cdot i = j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot i < j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot j < i\}. f i j)))\)

proof –

have \((\bigsqcap i. \bigsqcap j. f i j) = (\bigsqcap i. ((\bigsqcap j \in \{j \cdot i = j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot i < j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot j < i\}. f i j))\)

by (simp add: \text{INF-partition-nat3})

also have ...

\((\bigsqcap i. \bigsqcap j \in \{j \cdot i = j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot i < j\}. f i j) \sqcap (\bigsqcap j \in \{j \cdot j < i\}. f i j)\)

by (simp add: \text{INF-inf-distrib})

finally show \?thesis .

qed

lemma \text{INF-nat-shift}:

\((\bigsqcap i \in \{i \cdot 0 < i\}. f i) = (\bigsqcap i. f (\text{Suc} i))\)

by (metis \text{greaterThan-0} \text{greaterThan-def} \text{range-composition})

lemma \text{INF-nat-minus}:

fixes \(f::\text{nat} \Rightarrow 'a::\text{refinement-lattice}\)

shows \((\bigsqcap j \in \{j \cdot i < j\}. f (j \cdot i)) = (\bigsqcap k \in \{k \cdot 0 < k\}. f k)\)

apply (rule \text{antisym})

apply (rule \text{INF-mono}, \text{simp})

apply (metis \text{add.right-neutral} \text{add-diff-cancel-left'} \text{add-less-cancel-left} \text{order-refl})

apply (rule \text{INF-mono}, \text{simp})

by (meson \text{order-refl} \text{zero-less-diff})
lemma INF-INF-guarded-switch:
  fixes f :: nat ⇒ nat ⇒ 'a::{refinement-lattice}
  shows (∏\i. ∏\j\in\{j. j < i\}. f j (i - j)) = (∏ j. ∏ i\in\{i. j < i\}. f j (i - j))
proof (rule antisym)
  have ∏\jj\ i. \jj < ii ⇒ ∃ ii < i. f j (i - j) ⊆ f j j (ii - jj)
  by blast
  then have ∏\jj\ i. \jj < ii ⇒ ∃ ii < i. (∏\j\in\{j. j < i\}. f j (i - j)) ⊆ f j j (ii - jj)
  by (meson INF-lower mem-Collect-eq)
  then have ∏\jj\ i. \jj < ii ⇒ (∏\i. ∏\j\in\{j. j < i\}. f j (i - j)) ⊆ f j j (ii - jj)
  by (meson UNIV-I INF-lower dual-order.trans)
  then have (∏\i. ∏\j\in\{j. j < i\}. f j (i - j)) ⊆ (∏\jj\ i. ∏\ii\in\{ii. jj < ii\}. f jj (ii - jj))
  by (metis (mono-tags, lifting) INF-greatest mem-Collect-eq)
  then have (∏\i. ∏\j\in\{j. j < i\}. f j (i - j)) ⊆ (∏\jj\ i. ∏\ii\in\{ii. jj < ii\}. f jj (ii - jj))
  by (simp add: INF-greatest)
  then show (∏\i. ∏\j\in\{j. j < i\}. f j (i - j)) ⊆ (∏\jj\ i. ∏\ii\in\{i. j < i\}. f j (i - j))
  by simp
next
  have ∏\ii\ jj. \jj < ii ⇒ ∃ ii > j. f j (i - j) ⊆ f jj (ii - jj)
  by blast
  then have ∏\ii\ jj. \jj < ii ⇒ ∃ ii > j. (∏\i\in\{i. j < i\}. f j (i - j)) ⊆ f jj (ii - jj)
  by (meson INF-lower mem-Collect-eq)
  then have ∏\ii\ jj. \jj < ii ⇒ (∏\j\in\{i. j < i\}. f j (i - j)) ⊆ f jj j (ii - jj)
  by (meson UNIV-I INF-lower dual-order.trans)
  then have (∏\ii\ jj. ∏\i\in\{i. j < i\}. f j (i - j)) ⊆ (∏\jj\ ii\in\{ii. jj < ii\}. f jj (ii - jj))
  by (metis (mono-tags, lifting) INF-greatest mem-Collect-eq)
  then have (∏\ii\ jj. ∏\i\in\{i. j < i\}. f j (i - j)) ⊆ (∏\jj\ ii\in\{ii. jj < ii\}. f jj (ii - jj))
  by (simp add: INF-greatest)
  then show (∏\ii\ jj. ∏\i\in\{i. j < i\}. f j (i - j)) ⊆ (∏\jj\ i. ∏\ii\in\{i. j < i\}. f j (i - j))
  by simp
qed

end
end

11 Iteration for conjunctive models

theory Conjunctive-Iteration
imports
  Conjunctive-Sequential
  Iteration
  Infimum-Nat
begin
Sequential left-distributivity is only supported by conjunctive models but does not apply in general. The relational model is one such example.

locale iteration-finite-conjunctive = seq-finite-conjunctive + iteration

begin

lemma isolation: $c^\omega = c^* \cap c^\infty$
proof -
define $F$ where $F = (\lambda x. c^* \cap x)$
define $G$ where $G = (\lambda x. c;x)$
define $H$ where $H = (\lambda x. \text{nil} \cap c;x)$

have $FG$: $F \circ G = (\lambda x. c^* \cap c;x)$ using $F$-def $G$-def by auto
have $HF$: $H \circ F = (\lambda x. \text{nil} \cap c:(c^* \cap x))$ using $F$-def $H$-def by auto

have adjoint: dist-over-sup $F$ by (simp add: $F$-def inf-Sup)
have $\text{monoH}$: mono $H$ by (metis $H$-def $H$-def $\text{monoI}$ $\text{seq-mono-right}$)
have $\text{monoG}$: mono $G$ by (metis $G$-def inf $\text{absorb-iff2 monoI}$ $\text{seq-inf-distrib}$)

have $\forall x. ((F \circ G) x = (H \circ F) x)$ using $FG$ $HF$
  by (metis fiter-unfold $\text{inf-sup-aci(2) seq-inf-distrib}$)
then have $F (\text{lfp } G) = \text{lfp } H$ using adjoint $\text{monoH}$ $\text{monoG}$ fusion-lfp-eq by blast
then have $c^* \cap \text{lfp } (\lambda x. c;x) = \text{lfp } (\lambda x. \text{nil} \cap c;x)$
  using $F$-def $H$-def $\text{weak-seq-inf-distrib}$
thus ?thesis by (simp add: $\text{infter-def iter-def}$)
qed

lemma iter-induct-isolate: $c^*;d \cap c^\infty = \text{lfp } (\lambda x. d \cap c;x)$
proof -
define $F$ where $F = (\lambda x. c^*;d \cap x)$
define $G$ where $G = (\lambda x. c;x)$
define $H$ where $H = (\lambda x. d \cap c;x)$

have $FG$: $F \circ G = (\lambda x. c^*;d \cap c;x)$ using $F$-def $G$-def by auto
have $HF$: $H \circ F = (\lambda x. d \cap c;c^*;d \cap c;x)$ using $F$-def $H$-def $\text{weak-seq-inf-distrib}$
  by (metis comp-apply $\text{inf.commute inf.left-commute seq-assoc seq-inf-distrib}$)
have unroll: $c^*;d = (\text{nil} \cap c;c^*);d$ using fiter-unfold by auto
have distribute: $c^*;d = d \cap c;c^*;d$ by (simp add: unroll $\text{inf-seq-distrib}$)
have $FGx$: $(F \circ G) x = d \cap c;c^*;d \cap c;x$ using $FG$ distribute by simp

have adjoint: dist-over-sup $F$ by (simp add: $F$-def inf-Sup)
have $\text{monoH}$: mono $H$ by (metis $H$-def $\text{monoI}$ $\text{order-refl seq-mono-right}$)
have $\text{monoG}$: mono $G$ by (metis $G$-def $\text{absorb-iff2 monoI}$ $\text{seq-inf-distrib}$)
have \( \forall x. ((F \circ G) x = (H \circ F) x) \) using FGx HF by (simp add: FG distribute)
then have \( F (\text{lfp } G) = \text{lfp } H \) using adjoint monoH monoG fusion-lfp-eq by blast
then have \( c^\omega ; d \sqcap \text{lfp } (\lambda x. c;x) = \text{lfp } (\lambda x. d \sqcap c;x) \)
using F-def G-def H-def by blast
thus ?thesis by (simp add: infiter-def)
\qed

lemma iter-induct-eq: \( c^\omega ; d = \text{lfp } (\lambda x. d \sqcap c;x) \)
proof
\have \( c^\omega ; d = c^\star ; d \sqcap c^\omega ; d \) by (simp add: isolation inf-seq-distrib)
then have \( c^\star ; d \sqcap c^\omega ; d = c^\star ; d \sqcap c^\omega \) by (simp add: infiter-annil)
then have \( c^\star ; d \sqcap c^\omega = \text{lfp } (\lambda x. d \sqcap c;x) \) by (simp add: iter-induct-isolate)
thus ?thesis
by (simp add: \( (c^\omega ; d = c^\star ; d \sqcap c^\omega ; d \sqcap c^\omega ; d = c^\star ; d \sqcap c^\omega ) \))
\qed

lemma iter-induct: \( d \sqcap c;x \sqsubseteq x \Rightarrow c^\omega ; d \sqsubseteq x \)
by (simp add: iter-induct-eq lfp-lowerbound)

lemma iter-isolate: \( c^\star ; d \sqcap c^\omega = c^\star ; d \)
by (simp add: iter-induct-eq iter-induct-isolate)

lemma iter-isolate2: \( c;c^\star ; d \sqcap c^\omega = c;c^\star ; d \)
by (metis infiter-unfold iter-isolate seq-assoc seq-inf-distrib)

lemma iter-decomp: \( (c \sqcap d)^\omega = c^\omega ; (d;c^\omega )^\omega \)
proof (rule antisym)
\have \( c;c^\omega ; (d;c^\omega )^\omega \sqcap (d;c^\omega )^\omega \sqsubseteq c^\omega ; (d;c^\omega )^\omega \) by (metis inf-commute order.refl inf-seq-distrib seq-nil-left iter-unfold)
thus \( (c \sqcap d)^\omega \sqsubseteq c^\omega ; (d;c^\omega )^\omega \) by (metis inf.left-commute iter-induct-nil iter-unfold seq-assoc inf-seq-distrib)
next
\have \( (c ; (c \sqcap d)^\omega \sqcap d;(c \sqcap d)^\omega ) \sqsubseteq (c \sqcap d)^\omega \) by (metis inf-commute order.refl inf-seq-distrib iter-unfold)
then have \( a: c^\omega ; (d;c \sqcap d)^\omega \sqcap \text{nil} \sqsubseteq (c \sqcap d)^\omega \)
proof
\have \( \text{nil} \sqcap d;(c \sqcap d)^\omega \sqcap c;(c \sqcap d)^\omega \sqsubseteq (c \sqcap d)^\omega \)
by (metis eq-iff inf.semigroup-axioms inf-commute inf-seq-distrib iter-unfold semigroup.assoc)
thus ?thesis using iter-induct-eq by (metis inf-sup-aci(1) iter-induct)
\qed

then have \( d; c^\omega ; (d;c \sqcap d)^\omega \sqcap \text{nil} \sqsubseteq d;(c \sqcap d)^\omega \sqcap \text{nil} \) by (metis inf-mono

32
proof

lemma fiter-leapfrog:
  seq-nil-left seq-nil-right by fastforce

inf-assoc iter-leapfrog-var inf-seq-distrib
seq-mono iter-unfold iter1 iter2

next
proof

lemma iter-leapfrog-var
qed

proof

lemma iter-leapfrog: (c;d)^ω;c ⊆ c;(d;c)^ω
proof (rule antisym)
  show (c;d)^ω;c ⊆ c;(d;c)^ω by (metis iter-leapfrog-var)
next
  have (d;c)^ω ⊆ ((d;c)^ω;d);c ⊆ nil by (metis inf.bounded-iff order.refl seq-assoc
  seq-mono iter-unfold iter1 iter2)
  then have (d;c)^ω ⊆ (d;(c;d)^ω);c ⊆ nil by (metis inf.absorb-iff2 inf.boundedE
  inf-assoc iter-leapfrog-var inf-seq-distrib)
  then have c;(d;c)^ω ⊆ c;d;(c;d)^ω;c ⊆ nil;c using inf.bounded-iff seq-assoc seq-mono-right
  seq-nil-left seq-nil-right by fastforce
  thus c;(d;c)^ω ⊆ (c;d)^ω;c by (metis inf-commute inf-seq-distrib iter-unfold)
qed

lemma fiter-leapfrog: c;(d;c)^ω = (c;d)^ω;c
proof
  have lr: c;(d;c)^ω ⊆ (c;d)^ω;c
  proof
    have (d : c)^ω = c ▷ d ; c : (d : c)^ω
    proof
      have (d ; c)^ω = nil ▷ d ; c : (d ; c)^ω
      by (meson finite-iteration.fiter-unfold finite-iteration-axioms)
      then show ?thesis
      by (metis fiter-induct seq-assoc seq-distrib-left.weak-seq-distrib
        seq-distrib-left-axioms seq-nil-right)
    qed
  qed
  have rl: (c;d)^ω;c ⊆ c;(d;c)^ω
  proof
    have a1: (c;d)^ω;c = c ▷ c;d;(c;d)^ω;c
    by (metis finite-iteration.fiter-unfold finite-iteration-axioms
      inf-seq-distrib seq-nil-left)
    have a2: (c;d)^ω;c ⊆ c;(d;c)^ω ←→ c ▷ c;d;(c;d)^ω;c ⊆ c;(d;c)^ω by (simp add: a1)
    then have a3: ... ←→ c;( nil ▷ d;(c;d)^ω;c) ⊆ c;(d;c)^ω

33
by (metis a1 eq-iff fiter-unfold lr seq-assoc seq-inf-distrib seq-nil-right)

have a4: (nil ⊓ d;(c;d)*) ⊑ (d;c)* \implies c( nil ⊓ d;(c;d)*) ⊑ c(d;c)*
  using seq-mono-right by blast

have a5: (nil ⊓ d;(c;d)*) ⊑ (d;c)*
proof
  have f1: d ; (c ; d)* ; c ⊓ nil = d ; ((c ; d)* ; c) ⊓ nil ⊓ nil
    by (simp add: seq-assoc)
  have d ; c ; (d ; (c ; d)*) ; c ⊓ nil = d ; ((c ; d)* ; c)
    by (metis (no-types) a1 inf-sup-aci (1) seq-assoc
         seq-finite-conjunctive seq-inf-distrib seq-finite-conjunctive-axioms
         seq-nil-right)
  then show ?thesis
    using f1 by (metis (no-types) finite-iteration.fiter-induct finite-iteration-axioms
             inf.cobounded1 inf-sup-aci (1) seq-nil-right)
qed

thus ?thesis using a2 a3 a4 by blast
qed

thus ?thesis by (simp add: eq-iff lr)
qed

end

locale iteration-infinite-conjunctive = seq-infinite-conjunctive + iteration + infimum-nat

begin

lemma fiter-seq-choice: c* = (\prod_{i::nat.} c \vdash i)
proof (rule antisym)
  show c* ⊑ (\prod_{i::nat.} c \vdash i)
  proof (rule INF-greatest)
    fix i
    show c* ⊑ c \vdash i
      proof (induct i type: nat)
        case 0
        show c* ⊑ c \vdash 0 by (simp add: fiter0)
      next
        case Suc n
        have c* ⊑ c ; c* by (metis fiter-unfold inf-le2)
        also have ... ⊑ c ; (c \vdash n) using Suc.hyps by (simp only: seq-mono-right)
        also have ... = c \vdash Suc n by simp
    qed
  qed

locale iteration-infinite-conjunctive = seq-infinite-conjunctive + iteration + infimum-nat

begin

lemma fiter-seq-choice: c* = (\prod_{i::nat.} c \vdash i)
proof (rule antisym)
  show c* ⊑ (\prod_{i::nat.} c \vdash i)
  proof (rule INF-greatest)
    fix i
    show c* ⊑ c \vdash i
      proof (induct i type: nat)
        case 0
        show c* ⊑ c \vdash 0 by (simp add: fiter0)
      next
        case Suc n
        have c* ⊑ c ; c* by (metis fiter-unfold inf-le2)
        also have ... ⊑ c ; (c \vdash n) using Suc.hyps by (simp only: seq-mono-right)
        also have ... = c \vdash Suc n by simp
    qed

end
finally show $c^* \subseteq c : ^\sim\text{Suc } n$.

qed

next

have $(\prod i. c : ^\sim i) \subseteq (c : ^\sim 0) \sqcap (\prod i. c : ^\sim \text{Suc } i)$

by (meson INF-greatest INF-lower UNIV-I le-inf-iff)

also have $\ldots = \text{nil} \sqcap (\prod i. c; (c : ^\sim i))$ by simp

also have $\ldots = \text{nil} \sqcap c; (\prod i. c : ^\sim i)$ by (simp add: seq-INF-distrib)

finally show $(\prod i. c : ^\sim i) \subseteq c^*$ using fiter-induct by fastforce

qed

lemma fiter-seq-choice-nonempty: $c; c^* = (\prod i \in \{i. 0 < i\}, c : ^\sim i)$

proof

have $(\prod i \in \{i. 0 < i\}, c : ^\sim i) = (\prod i. c; (c : ^\sim \text{Suc } i))$ by (simp add: INF-nat-shift)

also have $\ldots = (\prod i. c; (c : ^\sim i))$ by simp

also have $\ldots = c; (\prod i. c : ^\sim i)$ by (simp add: seq-INF-distrib-UNIV)

also have $\ldots = c; c^*$ by (simp add: fiter-seq-choice)

finally show ?thesis by simp

qed

end

locale conj-iteration = cra + iteration-infinite-conjunctive

begin

lemma conj-distrib4: $c^* \sqcap d^* \subseteq (c \sqcap d)^*$

proof

have $c^* \sqcap d^* = (\text{nil} \sqcap (c; c^*)) \sqcap d^*$ by (metis fiter-unfold)

then have $c^* \sqcap d^* = (\text{nil} \sqcap d^*) \sqcap ((c; c^*) \sqcap d^*)$ by (simp add: inf-conj-distrib)

then have $c^* \sqcap d^* \subseteq \text{nil} \sqcap ((c; c^*) \sqcap (d; d^*))$ by (metis conj-idem fiter0 fiter-unfold inf.bounded-iff inf-le2 local.conj-mono)

then have $c^* \sqcap d^* \subseteq \text{nil} \sqcap ((c \sqcap d); (c^* \sqcap d^*))$ by (meson inf-mono-right order.trans sequential-interchange)

thus ?thesis by (metis seq-nil-right fiter-induct)

qed

end
12 Rely Quotient Operator

The rely quotient operator is used to generalise a Jones-style rely condition to a process [5]. It is defined in terms of the parallel operator and a process \( i \) representing interference from the environment.

theory Rely-Quotient
imports
  CRA
  Conjunctive-Iteration
begin

12.1 Basic rely quotient

The rely quotient of a process \( c \) and an interference process \( i \) is the most general process \( d \) such that \( c \) is refined by \( d \parallel i \). The following locale introduces the definition of the rely quotient \( c // i \) as a non-deterministic choice over all processes \( d \) such that \( c \) is refined by \( d \parallel i \).

locale rely-quotient =
  par-distrib +
  conjunction-parallel
begin

definition
  rely-quotient :: \('a \Rightarrow 'a \Rightarrow 'a\) (infixl '/'/ 85)
where
  \( c // i \equiv \bigsqcap \{ \text{d. } (c \sqsubseteq \text{d } \parallel \text{i}) \} \)

Any process \( c \) is implemented by itself if the interference is skip.

lemma quotient-identity: \( c // \text{skip} = c \)
proof −
  have \( c // \text{skip} = \bigsqcap \{ \text{d. } (c \sqsubseteq \text{d } \parallel \text{skip}) \} \) by (metis rely-quotient-def)
  then have \( c // \text{skip} = \bigsqcap \{ \text{d. } (c \sqsubseteq \text{d}) \} \) by (metis (mono-tags, lifting) Collect-cong par-skip)
  thus \( \text{thesis} \) by (metis Inf-greatest Inf-lower2 dual-order.antisym dual-order.refl mem-Collect-eq)
qed

Provided the interference process \( i \) is non-aborting (i.e. it refines chaos), any process \( c \) is refined by its rely quotient with \( i \) in parallel with \( i \). If interference \( i \) was allowed to be aborting then, because \( (c // \bot) \parallel \bot \) equals \( \bot \), it does not refine \( c \) in general.

theorem rely-quotient:
  assumes nonabort-i: chaos \( \sqsubseteq i \)
  shows \( c \sqsubseteq (c // i) \parallel i \)
proof —

define D where D = \{d \parallel i \mid d. (c \subseteq d \parallel i)\}
define C where C = \{c\}

have \((\forall d \in D. (\exists c \in C. c \subseteq d))\) using D-def C-def CollectD singletonI by auto
then have \(\prod C \subseteq (\prod D)\) by (simp add: Inf-mono)
then have \(c \subseteq \prod\{d \parallel i \mid d. (c \subseteq d \parallel i)\}\) by (simp add: C-def D-def)
also have \(\prod\{d \mid d. (c \subseteq d \parallel i)\} = \prod\{d \mid d. (c \subseteq d \parallel i)\} \parallel i\)
proof (cases \(\{d \mid d. (c \subseteq d \parallel i)\} = \{\}\))
  assume \(\{d \mid d. (c \subseteq d \parallel i)\} = \{\}\)
  then show \(\prod\{d \mid d. (c \subseteq d \parallel i)\} = \prod\{d \mid d. (c \subseteq d \parallel i)\} \parallel i\)
    using nonabort-i Collect-empty-eq top-greatest
    nonabort-par-top par-commute by fastforce
next
  assume a: \(\{d \mid d. (c \subseteq d \parallel i)\} \neq \{\}\)
  have b: \(\{d. (c \subseteq d \parallel i)\} \neq \{\}\) using a by blast
  then have \(\prod\{d \mid d. (c \subseteq d \parallel i)\} = \prod\{d \mid d. (c \subseteq d \parallel i)\} \parallel i\)
    using Inf-par-distrib by simp
  then show \(?\)thesis by auto
qed

also have \(\prod\{d \mid d. (c \subseteq d \parallel i)\} = \prod\{d \mid d. (c \subseteq d \parallel i)\} \parallel i\)
proof (cases \(\{d \mid d. (c \subseteq d \parallel i)\} = \{\}\))
  assume \(\{d \mid d. (c \subseteq d \parallel i)\} = \{\}\)
  then show \(\prod\{d \mid d. (c \subseteq d \parallel i)\} = \prod\{d \mid d. (c \subseteq d \parallel i)\} \parallel i\)
    using nonabort-i Collect-empty-eq top-greatest
    nonabort-par-top par-commute by fastforce
next
  assume a: \(\{d \mid d. (c \subseteq d \parallel i)\} \neq \{\}\)
  have c: \(c \subseteq (c \parallel i)\) using rely-quotient nonabort-i by simp
  thus \(c \subseteq d \parallel i\) using par-mono a
    by (metis inf.absorb-iff inf-commute le-inf2 order_refl)
next
  assume a: \(c \subseteq d \parallel i\)
  have \(\prod\{d. (c \subseteq d \parallel i)\} \subseteq \prod d\) by (simp add: Inf-lower)
  thus \(c \parallel i \subseteq d\) by (metis rely-quotient-def)
qed

The following theorem represents the Galois connection between the parallel operator (upper adjoint) and the rely quotient operator (lower adjoint). This basic relationship is used to prove the majority of the theorems about rely quotient.

theorem rely-refinement:
  assumes nonabort-i: \(\text{chaos} \subseteq i\)
  shows \(c \parallel i \subseteq d \leftrightarrow c \subseteq d \parallel i\)
proof
  assume a: \(c \parallel i \subseteq d\)
  have c: \(c \subseteq (c \parallel i)\) using rely-quotient nonabort-i by simp
  thus \(c \subseteq d \parallel i\) using par-mono a
    by (metis inf.absorb-iff inf-commute le-inf2 order_refl)
next
  assume a: \(c \subseteq d \parallel i\)
  have \(\prod\{d. (c \subseteq d \parallel i)\} \subseteq \prod d\) by (simp add: Inf-lower)
  thus \(c \parallel i \subseteq d\) by (metis rely-quotient-def)
qed

Refining the “numerator” in a quotient, refines the quotient.
lemma rely-mono:
  assumes c-refsto-d: c ⊑ d
  shows (c // i) ⊑ (d // i)
proof −
  have ∃ f. ((d ⊑ f || i) ⇒ ∃ e. (c ⊑ e || i) ∧ (e ⊑ f))
    using c-refsto-d order.trans by blast
  then have b: ∏ { e. (c ⊑ e || i)} ⊑ ∏ { f. (d ⊑ f || i)}
    by (metis Inf-mono mem-Collect-eq)
  show ?thesis using rely-quotient-def b by simp
qed

Refining the “denominator” in a quotient, gives a reverse refinement for the quotients. This corresponds to weaken rely condition law of Jones [5], i.e. assuming less about the environment.

lemma weaken-rely:
  assumes i-refsto-j: i ⊑ j
  shows (c // j) ⊑ (c // i)
proof −
  have ∃ f. ((c ⊑ f || i) ⇒ ∃ e. (c ⊑ e || j) ∧ (e ⊑ f))
    using i-refsto-j order.trans
    by (metis inf.absorb-iff2 inf-le1 inf-par-distrib inf-sup-ord(2) par-commute)
  then have b: ∏ { e. (c ⊑ e || j)} ⊑ ∏ { f. (c ⊑ f || i)}
    by (metis Inf-mono mem-Collect-eq)
  show ?thesis using rely-quotient-def b by simp
qed

lemma par-nonabort:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j: chaos ⊑ j
  shows chaos ⊑ i || j
  by (meson chaos-par-chaos nonabort-i nonabort-j order-trans par-mono)

Nesting rely quotients of j and i means the same as a single quotient which is the parallel composition of i and j.

lemma nested-rely:
  assumes j-nonabort: chaos ⊑ j
  shows ((c // j) // i) = c // (i || j)
proof (rule antisym)
  have ∃ f. ((c ⊑ f || i || j) ⇒ ∃ e. (c ⊑ e || j) ∧ (e ⊑ f || i)) by blast
  then have ∏ { d. (∏ { e. (c ⊑ e || j)} ⊑ d || i)} ⊑ ∏ { f. (c ⊑ f || i || j)}
    by (simp add: Collect-mono Inf-lower Inf-superset-mono)
thus ?thesis using local.rely-quotient-def par-assoc by auto
qed
next
show c // (i || j) ⊆ ((c // j) // i)
proof −
  have c ⊆ ∩ { e. (c ⊆ e || j)} || j
    using j-nonabort local.rely-quotient-def rely-quotient by auto
  then have \( \land \) d. \( \land \) { e. (c ⊆ e || j)} ⊆ d || i \( \implies \) (c ⊆ d || i || j)
    by (meson j-nonabort order-trans rely-refinement)
  thus ?thesis
by (simp add: Collect-mono Inf-superset-mono local.rely-quotient-def par-assoc)
qed
qed
end

12.2 Distributed rely quotient
locale rely-distrib = rely-quotient + conjunction-sequential
begin
The following is a fundamental law for introducing a parallel composition of
process to refine a conjunction of specifications. It represents an abstract
view of the parallel introduction law of Jones [5].

lemma introduce-parallel:
  assumes nonabort-i: chaos ⊆ i
  assumes nonabort-j: chaos ⊆ j
  shows c \( \ominus \) d ⊆ (j \( \ominus \) (c // i)) || (i \( \ominus \) (d // j))
proof −
  have a: c ⊆ (c // i) || i using nonabort-i nonabort-j rely-quotient by auto
  have b: d ⊆ j || (d // j) using rely-quotient par-commute
    by (simp add: nonabort-j)
  have c \( \ominus \) d ⊆ ((c // i) || i) \( \ominus \) (j || (d // j)) using a b by (metis conj-mono)
  also have interchange: c \( \ominus \) d ⊆ ((c // i) \( \ominus \) j) || (i \( \ominus \) (d // j))
    using parallel-interchange refine-trans calculation by blast
  show ?thesis using interchange by (simp add: local.conj-commute)
qed
Rely quotients satisfy a range of distribution properties with respect to the
other operators.

lemma distribute-rely-conjunction:
  assumes nonabort-i: chaos ⊆ i

39
shows $c \cap d \mathrel{\parallel} i \subseteq (c \mathrel{\parallel} i) \cap (d \mathrel{\parallel} i)$

proof

  have $c \cap d \subseteq ((c \mathrel{\parallel} i) \cap (d \mathrel{\parallel} i)) \cap (i \cap i)$ using conj-mono rely-quotient
  by (simp add: nonabort-i)
  then have $c \cap d \subseteq ((c \mathrel{\parallel} i) \cap (d \mathrel{\parallel} i)) \cap (i \cap i)$
  by (metis parallel-interchange refine-trans)
  then have $c \cap d \subseteq ((c \mathrel{\parallel} i) \cap (d \mathrel{\parallel} i)) \cap (i \cap i)$ by (metis conj-idem)
  thus ?thesis using rely-refinement by (simp add: nonabort-i)
qed

lemma distribute-rely-choice:

  assumes nonabort-i: chaos $\subseteq$ i
  shows $(c \cap d) \mathrel{\parallel} i \subseteq (c \mathrel{\parallel} i) \cap (d \mathrel{\parallel} i)$

proof

  have $(c \cap d) \subseteq ((c \mathrel{\parallel} i) \cap (d \mathrel{\parallel} i)) \cap (i \cap i)$
  by (simp add: nonabort-i)
  then have $(c \cap d) \subseteq ((c \mathrel{\parallel} i) \cap (d \mathrel{\parallel} i)) \cap (i \cap i)$
  by (metis nonabort-i inf-mono rely-quotient nonabort-i)
  then have $(c \cap d) \subseteq ((c \mathrel{\parallel} i) \cap (d \mathrel{\parallel} i)) \cap (i \cap i)$ by (metis inf-par-distrib)
  thus ?thesis by (metis nonabort-i rely-refinement)
qed

lemma distribute-rely-parallel1:

  assumes nonabort-i: chaos $\subseteq$ i
  assumes nonabort-j: chaos $\subseteq$ j
  shows $(c \parallel d) \mathrel{\parallel} (i \parallel j) \subseteq (c \mathrel{\parallel} i) \parallel (d \mathrel{\parallel} j)$

proof

  have $(c \parallel d) \subseteq ((c \mathrel{\parallel} i) \parallel (d \mathrel{\parallel} j)) \parallel (i \parallel j)$
  using assms(1) using weaken-rely
  by (simp add: i-par-i par-nonabort)
  then have $(c \parallel d) \subseteq ((c \mathrel{\parallel} i) \parallel (d \mathrel{\parallel} j)) \parallel (i \parallel j)$
  by (metis par-assoc par-commute)
  thus ?thesis using par-assoc par-commute rely-refinement
  by (metis nonabort-i nonabort-j par-nonabort)
qed

lemma distribute-rely-parallel2:

  assumes nonabort-i: chaos $\subseteq$ i
  assumes i-par-i: i $\parallel$ i $\subseteq$ i
  shows $(c \parallel d) \mathrel{\parallel} i \subseteq (c \parallel i) \parallel (d \mathrel{\parallel} i)$

proof

  have $(c \parallel d) \subseteq ((c \parallel i) \parallel (d \mathrel{\parallel} i)) \parallel (i \parallel i)$
  using assms using weaken-rely
  by (simp add: i-par-i par-nonabort)
  then have $(c \parallel d) \subseteq ((c \parallel i) \parallel (d \mathrel{\parallel} i)) \parallel (i \parallel i)$
  by (simp add: nonabort-i par-nonabort)
  thus ?thesis by (metis distribute-rely-parallel1 refine-trans nonabort-i)
qed

lemma distribute-rely-sequential:

40
assumes nonabort-i: chaos ⊑ i
assumes (∀ c. (∀ d. ((c ∥ i);(d ∥ i) ⊑ (c;d) ∥ i)))
shows (c;d) // i ⊑ (c // i);(d // i)
proof
have c;d ⊑ ((c // i) || i):((d // i) || i)
  by (metis rely-quotient nonabort-i seq-mono)
then have c;d ⊑ (c // i):((d // i) || i) using assms(2) by (metis refine-trans)
thus ?thesis by (metis rely-refinement nonabort-i)
qed

lemma distribute-rely-sequential-event:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j: chaos ⊑ j
  assumes nonabort-e: chaos ⊑ e
  assumes (∀ c. (∀ d. ((c ∥ i);(d ∥ j);(e ∥ i);(e ∥ j)) ⊑ (c;d) ∥ i))
  shows (c;d) // (i;e;j) ⊑ (c // i);(d // j)
proof
have c;d ⊑ ((c // i) || i):((d // j) || j)
  by (metis order.refl rely-quotient nonabort-i nonabort-j seq-mono)
then have c;d ⊑ ((c // i);(d // j)) || (i;e;j) using assms
  by (metis refine-trans)
thus ?thesis using rely-refinement nonabort-i nonabort-j nonabort-e
  by (simp add: Inf-lower local.rely-quotient-def)
qed

lemma introduce-parallel-with-rely:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j0: chaos ⊑ j0
  assumes nonabort-j1: chaos ⊑ j1
  shows (c ⋒ d) // i ⊑ (j1 ⋒ (c // (j0 ⊑ i)));(j0 ⋒ (d // (j1 ⊑ i)))
proof
have (c ⋒ d) // i ⊑ (c // i) ⋒ (d // i)
  by (metis distribute-rely-conjunction nonabort-i)
then have (c ⋒ d) // i ⊑ (j1 ⋒ ((c // i) // j0));(j0 ⋒ ((d // i) // j1))
  by (metis introduce-parallel nonabort-j0 nonabort-j1 inf-assoc inf.absorb-iff1)
thus ?thesis by (simp add: nested-rely nonabort-i)
qed

lemma introduce-parallel-with-rely-guarantee:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j0: chaos ⊑ j0
  assumes nonabort-j1: chaos ⊑ j1
  shows (j1 // j0) ⋒ (c ⋒ d) // i ⊑ (j1 ⋒ ((c // (j0 ⊑ i))));(j0 ⋒ ((d // (j1 ⊑ i))))

proof –

have \((j_1 \parallel j_0) \cap (c \cap d) \parallel i \subseteq (j_1 \parallel j_0) \cap ((j_1 \cap (c \parallel (j_0 \parallel i))) \parallel (j_0 \cap (d \parallel (j_1 \parallel i))))\)

by (metis introduce-parallel-with-rely nonabort-i nonabort-j0 nonabort-j1

conj-mono order refl)
also have \(\vdash (j_1 \parallel j_0) \cap (c \parallel (j_0 \parallel i))) \parallel (j_0 \cap (d \parallel (j_1 \parallel i)))\)

by (metis conj-assoc parallel-interchange)
finally show ?thesis by (metis conj-idem)
qed

lemma wrap-rely-guar:
assumes nonabort-rg: chaos \(\subseteq\) rg
and skippable: rg \(\subseteq\) skip
shows c \(\subseteq\) rg \(\cap\) c \(\parallel\) rg
proof –

have c = c \(\parallel\) skip by (simp add: quotient-identity)
also have \(\vdash (j_1 \parallel j_0) \cap (c \parallel (j_0 \parallel i))) \parallel (j_0 \cap (d \parallel (j_1 \parallel i)))\)

by (metis inf-mono order refl rely-quotient nonabort-i seq-mono

also have \(\vdash (j_1 \parallel j_0) \cap (c \parallel (j_0 \parallel i))) \parallel (j_0 \cap (d \parallel (j_1 \parallel i)))\)

by (metis conj-assoc parallel-interchange)
finally show c \(\subseteq\) rg \(\cap\) c \(\parallel\) rg
by auto
finally show c \(\subseteq\) rg \(\cap\) c \(\parallel\) rg
qed

end

locale rely-distrib-iteration = rely-distrib + iteration-finite-conjunctive

begin

lemma distribute-rely-iteration:
assumes nonabort-i: chaos \(\subseteq\) i
assumes \((\forall c. (\forall d. ((c \parallel i) \parallel (d \parallel i)) \subseteq (c \cap (d \parallel i))))\)
shows \((c \cap (d \parallel i) \parallel i) \subseteq ((c \parallel (d \parallel i)) \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i))\)
proof –

have d \(\cap\) c \(\cap\) (\(\cap\) (d \(\parallel\) i) \(\parallel\) i) \(\subseteq\) ((d \(\parallel\) i) \(\parallel\) i) \(\cap\) ((c \(\parallel\) (d \(\parallel\) i)) \(\parallel\) i))

by (metis inf-mono order refl rely-quotient nonabort-i seq-mono

also have \(\vdash (d \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i))\)

using assms inf-mono-right seq-assoc by fastforce
also have \(\vdash (d \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i))\)

by (simp add: inf-par-distrib)
also have \(\vdash (c \parallel (d \parallel i)) \parallel i\)

finally show c \(\subseteq\) rg \(\cap\) c \(\parallel\) rg
by auto
finally show c \(\subseteq\) rg \(\cap\) c \(\parallel\) rg
qed

end

locale rely-distrib-iteration = rely-distrib + iteration-finite-conjunctive

begin

lemma distribute-rely-iteration:
assumes nonabort-i: chaos \(\subseteq\) i
assumes \((\forall c. (\forall d. ((c \parallel i) \parallel (d \parallel i)) \subseteq (c \cap (d \parallel i))))\)
shows \((c \cap (d \parallel i) \parallel i) \subseteq ((c \parallel (d \parallel i)) \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i))\)
proof –

have d \(\cap\) c \(\cap\) (\(\cap\) (d \(\parallel\) i) \(\parallel\) i) \(\subseteq\) ((d \(\parallel\) i) \(\parallel\) i) \(\cap\) ((c \(\parallel\) (d \(\parallel\) i)) \(\parallel\) i))

by (metis inf-mono order refl rely-quotient nonabort-i seq-mono

also have \(\vdash (d \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i))\)

using assms inf-mono-right seq-assoc by fastforce
also have \(\vdash (d \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i) \cap ((c \parallel (d \parallel i)) \parallel i))\)

by (simp add: inf-par-distrib)
also have \(\vdash (c \parallel (d \parallel i)) \parallel i\)

finally show c \(\subseteq\) rg \(\cap\) c \(\parallel\) rg
by auto
finally show c \(\subseteq\) rg \(\cap\) c \(\parallel\) rg
qed

end
by (metis iter-unfold inf-seq-distrib seq-nil-left)
finally show ?thesis by (metis rely-refinement nonabort-i iter-induct)
qed

end

end

13 Conclusions

The theories presented here provide a quite abstract view of the rely/guarantee approach to concurrent program refinement. A trace semantics for this theory has been developed [2]. The concurrent refinement algebra is general enough to also form the basis of a more concrete rely/guarantee approach based on a theory of atomic steps and synchronous parallel and weak conjunction operators [4].

Acknowledgements. This research was supported by Australian Research Council Grant grant DP130102901 and EPSRC (UK) Taming Concurrency grant. This research has benefited from feedback from Robert Colvin, Chelsea Edmonds, Ned Hoy, Cliff Jones, Larissa Meinicke, and Kirsten Winter.

A Differences to earlier paper

This appendix summarises the differences between these Isabelle theories and the earlier paper [3]. We list the changes to the axioms but not all the flow on effects to lemmas.

1. The earlier paper assumes \( c \triangleleft (d_0 \cap d_1) = (c \triangleleft d_0) \cap (c \triangleleft d_1) \) but here we separate the case where this is only a refinement from left to right (Section 3) from the equality case (Section 9).

2. The earlier paper assumes \( (\bigcap C) \parallel d = (\bigcap c \in C. c \parallel d) \) but in Section 4 we assume this only for non-empty \( C \) and furthermore assume that parallel is abort strict, i.e. \( \bot \parallel c = c \).

3. The earlier paper assumes \( c \cap (\bigcup D) = (\bigcup d \in D. c \cap d) \). In Section 5 that assumption is not made because it does not hold for the model we have in mind [2] but we do assume \( c \cap \bot = \bot \).
4. In Section 6 we add the assumption $\text{nil} \sqsubseteq \text{nil} \parallel \text{nil}$ to locale sequential-parallel.

5. In Section 6 we add the assumption $\top \sqsubseteq \text{chaos} \parallel \top$.

6. In Section 6 we assume only $\text{chaos} \sqsubseteq \text{chaos} \parallel \text{chaos}$ whereas in the paper this is an equality (the reverse direction is straightforward to prove).

7. In Section 6 axiom chaos-skip ($\text{chaos} \sqsubseteq \text{skip}$) has been dropped because it can be proven as a lemma using the parallel-interchange axiom.

8. In Section 6 we add the assumption $\text{chaos} \sqsubseteq \text{chaos} ; \text{chaos}$.

9. Section 9 assumes $D \neq \{\} \Rightarrow c; \bigcap D = (\bigcap d \in D. c; d)$. This distribution axiom is not considered in the earlier paper.

10. Because here parallel does not distribute over an empty non-deterministic choice (see point 2 above) in Section 12 the theorem rely-quotient needs to assume the interference process $i$ is non-aborting (refines chaos). This also affects many lemmas in this section that depend on theorem rely-quotient.

References


