Concurrent Refinement Algebra and Rely Quotients

Julian Fell and Ian Hayes and Andrius Velykis

September 13, 2023

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 6
  3.1 Basic sequential 6
  3.2 Distributed sequential 7

4 Parallel Operator 9
  4.1 Basic parallel operator 9
  4.2 Distributed parallel 9

5 Weak Conjunction Operator 10
  5.1 Distributed weak conjunction 11

6 Concurrent Refinement Algebra 12

7 Galois Connections and Fusion Theorems 14
  7.1 Lower Galois connections 15
  7.2 Greatest fixpoint fusion theorems 15
  7.3 Upper Galois connections 16
  7.4 Least fixpoint fusion theorems 16

8 Iteration 17
  8.1 Possibly infinite iteration 17
  8.2 Finite iteration 18
  8.3 Infinite iteration 19
  8.4 Combined iteration 20

9 Sequential composition for conjunctive models 21

10 Infimum nat lemmas 22

11 Iteration for conjunctive models 23

12 Rely Quotient Operator 24
  12.1 Basic rely quotient 25
  12.2 Distributed rely quotient 26

13 Conclusions 28
A Differences to earlier paper
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
begin

unbundle lattice-syntax

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 Eqiuv-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 ⊑ 50)
where
  c ⊑ d ≡ less-eq c d

abbreviation refine-strict :: 'a ⇒ 'a ⇒ bool (infix ⊏ 50)
where
  c ⊏ d ≡ less c d

Non-deterministic choice is monotonic in both arguments

lemma inf-mono-left: a ⊑ b ⇒ a ⊓ c ⊑ b ⊓ c
⟨proof⟩

lemma inf-mono-right: c ⊑ d ⇒ a ⊓ c ⊑ a ⊓ d
⟨proof⟩

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

lemma Inf2-inf: \{ f x | x ∈ {c, d} \} = f c ⊓ f d
⟨proof⟩

Helper lemma for choice over indexed set.

lemma INF-Inf: (\{ f x | x ∈ X \}) = (\{ f x | x ∈ X \})
⟨proof⟩

lemma (in −) INF-absorb-args: (∏ i j. (f::nat ⇒ 'c::complete-lattice) (i + j)) = (∏ k. f k)
⟨proof⟩

lemma (in −) nested-Collect: \{ f y | y ∈ \{ g x | x ∈ X \} \} = \{ f (g x) | x ∈ X \}
⟨proof⟩

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 | x ∈ X \}
  assumes f-Inf-distrib: (∏ c D. P D ⇒ f c (∏ D) = ∏ f c d | d . d ∈ D {]

5
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))

⟨proof⟩
end

lemmas refine-trans = order.trans

More transitivity rules to make calculational reasoning smoother

declare ord-eq-le-trans[trans]
declare ord-le-eq-trans[trans]
declare dual-order.trans[trans]

abbreviation dist-over-sup :: ('a::refinement-lattice ⇒ 'a ⇒ bool)
  where
  dist-over-sup F ≡ (\forall X . F (\bigcup X) = (\bigcup x\in X. F (x)))

abbreviation dist-over-inf :: ('a::refinement-lattice ⇒ 'a ⇒ bool)
  where
  dist-over-inf F ≡ (\forall X . F (\bigcap X) = (\bigcap x\in 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 \( \bot \) 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);((d0 ∩ d1) ⊑ (c;d0 ∩ c;d1))
begin  

Left distribution implies sequential composition is monotonic is its right argument

lemma seq-mono-right: c0 ⊑ c1 ⇒ d ; c0 ⊑ d ; c1
⟨proof⟩

lemma seq-bot-right [simp]: c;⊥ ⊑ c
⟨proof⟩

end

locale seq-distrib-right = sequential +
assumes Inf-seq-distrib:
(\prod \ C) \cdot d = (\prod (c::a::\text{refinement-lattice}) \in C) \cdot d)

begin

lemma INF-seq-distrib: (\prod c \in C. f c) \cdot d = (\prod c \in C. f c) \cdot d
⟨proof⟩

lemma inf-seq-distrib: (c_0 \sqcap c_1) \cdot d = (c_0 \cdot d \sqcap c_1 \cdot d)
⟨proof⟩

lemma seq-mono-left: c_0 \sqsubseteq c_1 \Rightarrow c_0 \cdot d \sqsubseteq c_1 \cdot d
⟨proof⟩

lemma seq-top [simp]: \top \cdot c = \top
⟨proof⟩

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

notation (latex output)
seq-power ((\cdot)^ \text{1000}) \text{1000}

notation (HTML output)
seq-power ((\cdot)^ \text{1000}) \text{1000}

lemma seq-power-front: (a \^ n) \cdot a = a \cdot (a \^ n)
⟨proof⟩

lemma seq-power-split-less: i < j \Rightarrow (b \^ j) = (b \^ i) \cdot (b \^ (j - i))
⟨proof⟩

end

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

lemma seq-mono: c_1 \sqsubseteq d_1 \Rightarrow c_2 \sqsubseteq d_2 \Rightarrow c_1 \cdot c_2 \sqsubseteq d_1 \cdot d_2
⟨proof⟩

end
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: $D \neq \{\} \implies (\bigsqcap d \in D) \parallel c = (\bigsqcap d \in D) \parallel c$

⟨proof⟩

lemma par-INF-distrib: $X \neq \{\} \implies c \parallel (\bigsqcap x \in X. d x) = (\bigsqcap x \in X. c \parallel d x)$

⟨proof⟩

lemma INF-par-distrib: $X \neq \{\} \implies (\bigsqcap x \in X. d x) \parallel c = (\bigsqcap x \in X. d x) \parallel c$

⟨proof⟩

lemma INF-INF-par-distrib:
  $X \neq \{\} \implies Y \neq \{\} \implies (\bigsqcap x \in X. c x) \parallel (\bigsqcap y \in Y. d y) = (\bigsqcap x \in X. \bigsqcap y \in Y. c x \parallel d y)$

⟨proof⟩

lemma inf-par-distrib: $(c_0 \sqcap c_1) \parallel d = (c_0 \parallel d) \sqcap (c_1 \parallel d)$

⟨proof⟩

lemma inf-par-distrib2: $d \parallel (c_0 \sqcap c_1) = (d \parallel c_0) \sqcap (d \parallel c_1)$

⟨proof⟩

lemma inf-par-product: $(a \sqcap b) \parallel (c \sqcap d) = (a \parallel c) \sqcap (a \parallel d) \sqcap (b \parallel c) \sqcap (b \parallel d)$

⟨proof⟩

lemma par-mono: $c_1 \sqsubseteq d_1 \implies c_2 \sqsubseteq d_2 \implies c_1 \parallel c_2 \sqsubseteq d_1 \parallel d_2$

⟨proof⟩

d end

d end

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 $\Rightarrow$ 'a  (infixl $\sqcap$ 80)
assumes conj-bot-right: c ⊓ ⊥ = ⊥

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]: ⊥ ⊓ c = ⊥
⟨proof⟩

lemma conj-not-bot: a ⊓ b ≠ ⊥ ⟹ a ≠ ⊥ ∧ b ≠ ⊥
⟨proof⟩

lemma conj-distrib1: c ⊓ (d₀ ⊓ d₁) = (c ⊓ d₀) ⊓ (c ⊓ d₁)
⟨proof⟩

end

5.1 Distributed weak conjunction

The weak conjunction operator distributes across arbitrary non-empty infima.

locale conj-distrib = conjunction +
  assumes Inf-conj-distrib: D ≠ { } ⟹ (∩ D) ⊓ c = (∩ d∈D. d ⊓ c)

begin

lemma conj-Inf-distrib: D ≠ { } ⟹ c ⊓ (∩ D) = (∩ d∈D. c ⊓ d)
⟨proof⟩

lemma inf-conj-distrib: (c₀ ⊓ c₁) ⊓ d = (c₀ ⊓ d) ⊓ (c₁ ⊓ d)
⟨proof⟩
lemma inf-conj-product: \((a \sqcap b) \sqcap (c \sqcap d) = (a \sqcap c) \sqcap (a \sqcap d) \sqcap (b \sqcap c) \sqcap (b \sqcap d)\)
\langle proof \rangle

lemma conj-mono: \(c_0 \subseteq d_0 \implies c_1 \subseteq d_1 \implies c_0 \sqcap c_1 \subseteq d_0 \sqcap d_1\)
\langle proof \rangle

lemma conj-mono-left: \(c_0 \subseteq c_1 \implies c_0 \sqcap d \subseteq c_1 \sqcap d\)
\langle proof \rangle

lemma conj-mono-right: \(c_0 \subseteq c_1 \implies d \sqcap c_0 \subseteq d \sqcap c_1\)
\langle proof \rangle

lemma conj-refine: \(c_0 \subseteq d \implies c_1 \subseteq d \implies c_0 \sqcap c_1 \subseteq d\)
\langle proof \rangle

lemma refine-to-conj: \(c \subseteq d_0 \implies c \subseteq d_1 \implies c \sqcap d_0 \sqcap d_1\)
\langle proof \rangle

lemma conjoin-non-aborting: \(\text{chaos} \subseteq c \implies d \subseteq d \sqcap c\)
\langle proof \rangle

lemma conjunction-sup: \(c \sqcap d \subseteq c \sqcup d\)
\langle proof \rangle

lemma conjunction-sup-nonaborting:
  assumes \(\text{chaos} \subseteq c\) and \(\text{chaos} \subseteq d\)
  shows \(c \sqcap d = c \sqcup d\)
\langle proof \rangle

lemma conjoin-top: \(\text{chaos} \subseteq c \implies c \sqcap \top = \top\)
\langle proof \rangle

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
Locale sequential-parallel brings together the sequential and parallel operators and relates their identities.

locale sequential-parallel = seq-distrib + par-distrib +
  assumes nil-par-nil: nil || nil ⊑ nil
  and skip-nil: skip ⊑ nil
  and skip-skip: skip ⊑ skip;skip
begin

lemma nil-absorb: nil || nil = nil ⟨proof⟩

lemma skip-absorb [simp]: skip;skip = skip
⟨proof⟩

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: T ⊑ chaos || T
  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⟩

lemma chaos-par-chaos-eq: chaos = chaos || chaos
⟨proof⟩

lemma nonabort-par-top: chaos ⊑ c =⇒ c || T = T
⟨proof⟩

lemma skip-conj-top: skip ⋒ T = T
⟨proof⟩

lemma conj-distrib2: c ⊑ c || c =⇒ c ⋒ (d0 || d1) ⊑ (c ⋒ d0) || (c ⋒ d1)
⟨proof⟩
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: (c_0;c_1) △ (d_0;d_1) ⊑ (c_0 ⊕ d_0);(c_1 ⊕ d_1)
begin

lemma chaos-nil: chaos ⊑ nil
⟨proof⟩

lemma chaos-seq-absorb: chaos = chaos;chaos
⟨proof⟩

lemma seq-bot-conj: c;⊥ ⊕ d ⊑ (c ⊕ d);⊥
⟨proof⟩

lemma conj-seq-bot-right [simp]: c;⊥ ⊕ c = c;⊥
⟨proof⟩

lemma conj-distrib3: c ⊑ c;c ⇒ c ⊕ (d_0 ; d_1) ⊑ (c ⊕ d_0);(c ⊕ d_1)
⟨proof⟩

end

Locale cra brings together sequential, parallel and weak conjunction.

locale cra = sequential-parallel + conjunction-parallel + conjunction-sequential
end

7 Galois Connections and Fusion Theorems

theory Galois-Connections
imports Refinement-Lattice
begin

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 | x = a \lor x = b \} = \{ F a, F b \}
⟨proof⟩

locale lower-galois-connections
begin

definition l-adjoint :: ('a::refinement-lattice ⇒ 'a ⇒ 'a) ⇒ ('a ⇒ '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⟩

lemma l-cancellation: dist-over-inf F \implies x \subseteq (F ◦ F♭) x
⟨proof⟩

lemma l-galois-connection: dist-over-inf F \implies ((F♭) x \subseteq y) \iff (x \subseteq F y)
⟨proof⟩

lemma v-simple-fusion: mono G \implies \forall x. ((F ◦ G) x \subseteq (H ◦ F) x) \implies F (gfp G) \subseteq gfp H
⟨proof⟩

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: mono H
and distribF: dist-over-inf F
and comp-geq: \forall x. ((H ◦ F) x \subseteq (F ◦ G) x)
shows gfp H \subseteq F (gfp G)
⟨proof⟩
**Theorem** fusion-gfp-eq:

**Assumes** monoH: mono H and monoG: mono G

and distF: dist-over-inf F

and fgh-comp: \( \bigwedge x. ((F \circ G) x = (H \circ F) x) \)

**Shows** \( F (\text{gfp } G) = \text{gfp } H \)

(\text{proof})

7.3 Upper Galois connections

**Locale** upper-galois-connections

**Begin**

**Definition**

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

**Where**

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

**Lemma** dist-sup-mono:

**Assumes** distF: dist-over-sup F

**Shows** mono F

(\text{proof})

**Lemma** u-cancellation: dist-over-sup F \( \Rightarrow (F \circ F^\#) x \subseteq x \)

(\text{proof})

**Lemma** u-galois-connection: dist-over-sup F \( \Rightarrow (F x \subseteq y) \leftrightarrow (x \subseteq (F^\#) y) \)

(\text{proof})

**Lemma** u-simple-fusion: mono H \( \Rightarrow \forall x. ((F \circ G) x \subseteq (G \circ H) x) \Rightarrow \text{lfp } F \subseteq G (\text{lfp } H) \)

(\text{proof})

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: \( \bigwedge x. ((F \circ G) x \subseteq (H \circ F) x) \)

16
\( \text{shows } F (\text{f} p G) \subseteq (\text{f} p H) \)

\( \langle \text{proof} \rangle \)

\textbf{theorem} fusion-lfp-eq:
\begin{itemize}
  \item \textbf{assumes} \( \text{mono} H: \text{mono } H \) \textbf{and} \( \text{mono} G: \text{mono } G \)
  \item \textbf{and} \( \text{dist} F: \text{dist-over-sup } F \)
  \item \textbf{and} \( \text{fgh-comp: } \forall x. ((F \circ G) x = (H \circ F) x) \)
\end{itemize}
\( \text{shows } F (\text{f} p G) = (\text{f} p H) \)

\( \langle \text{proof} \rangle \)

8 Iteration

\textbf{theory} Iteration
\textbf{imports}
\begin{itemize}
  \item Galois-Connections
  \item CRA
\end{itemize}
\textbf{begin}

8.1 Possibly infinite iteration

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

\textbf{locale} finite-or-infinite-iteration =
\begin{itemize}
  \item seq-distrib
  \item upper-galois-connections
\end{itemize}
\textbf{begin}

\textbf{definition}
\begin{itemize}
  \item \textbf{iter} :: 
\end{itemize}
\( \langle \text{proof} \rangle \)

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

\textbf{theorem} iter-unfold: \( c^\omega = \text{nil} \sqcap c\cdot c^\omega \)
\( \langle \text{proof} \rangle \)
lemma iter-induct-nil: nil □ c; x ⊆ x ⇒ c^ω ⊆ x
\langle proof \rangle

lemma iter0: c^ω ⊆ nil
\langle proof \rangle

lemma iter1: c^ω ⊆ c
\langle proof \rangle

lemma iter2 [simp]: c^ω; c^ω = c^ω
\langle proof \rangle

lemma iter mono: c ⊆ d ⇒ c^ω ⊆ d^ω
\langle proof \rangle

lemma iter abort: ⊥ = nil^ω
\langle proof \rangle

lemma nil iter: ⊤^ω = nil
\langle proof \rangle

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] 100)
where
  c^* ≡ gfp (λ x. nil □ c;x)

lemma fin iter step mono: mono (λ x. nil □ c;x)
\langle proof \rangle

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

lemma fiter unfold: c^* = nil □ c;c^*
\langle proof \rangle

18
lemma fiter-induct-nil: \( x \sqsubseteq \text{nil} \cap c; x \Rightarrow x \sqsubseteq c^* \)  
\langle proof \rangle

lemma fiter0: \( c^* \sqsubseteq \text{nil} \)  
\langle proof \rangle

lemma fiter1: \( c^* \sqsubseteq c \)  
\langle proof \rangle

lemma fiter-induct-eq: \( c^*; d = \text{gfp} (\lambda x. c; x \cap d) \)  
\langle proof \rangle

theorem fiter-induct: \( x \sqsubseteq d \cap c; x \Rightarrow x \sqsubseteq c^*; d \)  
\langle proof \rangle

lemma fiter2 [simp]: \( c^*; c^* = c^* \)  
\langle proof \rangle

lemma fiter3 [simp]: \( (c^*)^* = c^* \)  
\langle proof \rangle

lemma fiter-mono: \( c \sqsubseteq d \Rightarrow c^* \sqsubseteq d^* \)  
\langle proof \rangle

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 (-\( \infty \) [105] 106)  
where \( c^\infty \equiv \text{lfp} (\lambda x. c; x) \)

lemma infiter-step-mono: mono (\( \lambda x. c; x \) \rangle
\langle proof \rangle

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

theorem infiter-unfold: \( c^\infty = c; c^\infty \)  
\langle proof \rangle
Lemma infiter-induct: \( c; x \sqsubseteq x \implies c^\infty \sqsubseteq x \rhd \)

Theorem infiter-unfold-any: \( c^\infty = (c ^:: i) ; c^\infty \rhd \)

Lemma infiter-annil: \( c^\infty ; x = c^\infty \rhd \)

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: \( c^\omega \sqsubseteq c^* \rhd \)

Lemma iter-absorption [simp]: \( (c^\omega)^* = c^\omega \rhd \)

Lemma infiter-inf-top: \( c^\infty = c^\omega ; \top \rhd \)

Lemma infiter-fiter-top: 
  Shows \( c^\infty \sqsubseteq c^* ; \top \rhd \)

Lemma inf-ref-infiter: \( c^\omega \sqsubseteq c^\infty \rhd \)

end

end
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; (d_0 \cap d_1) = c;d_0 \cap c;d_1
begin

sublocale seq-distrib-left
  ⟨proof⟩
end

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

sublocale seq-distrib
  ⟨proof⟩

lemma seq-INF-distrib: X \neq \{\} \Longrightarrow c ; (\bigsqcap x \in X. d x) = (\bigsqcap x \in X. c ; d x)
  ⟨proof⟩

lemma seq-INF-distrib-UNIV: c ; (\bigsqcap x. d x) = (\bigsqcap x. c ; d x)
  ⟨proof⟩

lemma INF-INF-seq-distrib: Y \neq \{\} \Longrightarrow (\bigsqcap x \in X. c x) ; (\bigsqcap y \in Y. d y) = (\bigsqcap x \in X. \bigsqcap y \in Y. c x ; d y)
  ⟨proof⟩

lemma INF-INF-seq-distrib-UNIV: (\bigsqcap x. c x) ; (\bigsqcap y. d y) = (\bigsqcap x. \bigsqcap y. c x ; d y)
  ⟨proof⟩

end

end
10 Infimum nat lemmas

theory Infimum-Nat
imports
  Refinement-Lattice
begin

locale infimum-nat
begin

lemma INF-partition-nat3:
  fixes f :: nat ⇒ nat ⇒ 'a::refinement-lattice
  shows \((\prod j\in\{i. i = j\}. f i j) \sqcap
  (\prod j\in\{i. i < j\}. f i j) \sqcap
  (\prod j\in\{j. j < i\}. f i j)\) =

⟨proof⟩

lemma INF-INF-partition-nat3:
  fixes f :: nat ⇒ nat ⇒ 'a::refinement-lattice
  shows \((\prod i. \prod j\in\{i. i = j\}. f i j) \sqcap
  (\prod i. \prod j\in\{i. i < j\}. f i j) \sqcap
  (\prod i. \prod j\in\{j. j < i\}. f i j)\) =

⟨proof⟩

lemma INF-nat-shift: \((\prod i\in\{i. 0 < i\}. f i) = \prod i. f (Suc i)\)

⟨proof⟩

lemma INF-nat-minus:
  fixes f :: nat ⇒ 'a::refinement-lattice
  shows \((\prod j\in\{j. i < j\}. f (j - i)) = \prod k\in\{k. 0 < k\}. f k\)

⟨proof⟩

lemma INF-INF-guarded-switch:
  fixes f :: nat ⇒ nat ⇒ 'a::refinement-lattice
  shows \((\prod i. \prod j\in\{j. j < i\}. f j (i - j)) = \prod j. \prod i\in\{i. j < i\}. f j (i - j)\)

⟨proof⟩

end

end

22
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⟩

lemma iter-induct-isolate: \( c^*;d \cap c^\infty = \text{lfp} (\lambda x. d \cap c;x) \)
  ⟨proof⟩

lemma iter-induct-eq: \( c^\omega;d = \text{lfp} (\lambda x. d \cap c;x) \)
  ⟨proof⟩

lemma iter-induct: \( d \cap c;x \subseteq x \implies c^\omega;d \subseteq x \)
  ⟨proof⟩

lemma iter-isolate: \( c^*;d \cap c^\infty = c^\omega;d \)
  ⟨proof⟩

lemma iter-isolate2: \( c;c^*;d \cap c^\infty = c;c^\omega;d \)
  ⟨proof⟩

lemma iter-decomp: \( (c \cap d)^\omega = c^\omega;(d;c^\omega)^\omega \)
  ⟨proof⟩

lemma iter-leapfrog-var: \( (c;d)^\omega;c \subseteq c;(d;c)^\omega \)
  ⟨proof⟩

lemma iter-leapfrog: \( c;(d;c)^\omega = (c;d)^\omega;c \)
  ⟨proof⟩

lemma fiter-leapfrog: \( c;(d;c)^* = (c;d)^*;c \)

⟨proof⟩
locale iteration-infinite-conjunctive = seq-infinite-conjunctive + iteration + infimum-nat

begin

lemma fiter-seq-choice: \( c^* = (\prod i:\text{nat}. c^{\hat{i}}) \)

⟨proof⟩

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

⟨proof⟩

end

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

begin

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

⟨proof⟩

end

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.
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 $'$a $'$a (infixl '//' 85)
where
$c//i \equiv \bigcap \{d. (c \sqsubseteq d \parallel i)\}$

Any process $c$ is implemented by itself if the interference is skip.

lemma quotient-identity: $c//\text{skip} = c$
(proof)

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)

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: chaos $\sqsubseteq i$
  shows $c//i \sqsubseteq d \leftrightarrow c \sqsubseteq d \parallel i$
(proof)

Refining the “numerator” in a quotient, refines the quotient.

lemma rely-mono: 
  assumes $c$-refsto-d: $c \sqsubseteq d$
  shows $(c//i) \sqsubseteq (d//i)$
(proof)

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⟩

lemma par-nonabort:
  assumes nonabort-i: chaos ⊑ i
  assumes nonabort-j: chaos ⊑ j
  shows chaos ⊑ i ∥ j
⟨proof⟩

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⟩

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 ∩ d ⊑ (j ∩ (c // i)) ∥ (i ∩ (d // j))
⟨proof⟩

Rely quotients satisfy a range of distribution properties with respect to the other operators.

lemma distribute-rely-conjunction:
  assumes nonabort-i: chaos ⊑ i
  shows (c ∩ d) // i ⊑ (c // i) ∩ (d // i)
⟨proof⟩
lemma distribute-rely-choice:
    assumes nonabort-i: chaos ⊆ i
    shows (c ∩ d) // i ⊑ (c // i) ∩ (d // i)
⟨proof⟩

lemma distribute-rely-parallel1:
    assumes nonabort-i: chaos ⊆ i
    assumes nonabort-j: chaos ⊆ j
    shows (c || d) // (i || j) ⊑ (c // i) || (d // j)
⟨proof⟩

lemma distribute-rely-parallel2:
    assumes nonabort-i: chaos ⊆ i
    assumes i-par-i: i || i ⊑ i
    shows (c || d) // i ⊑ (c // i) || (d // i)
⟨proof⟩

lemma distribute-rely-sequential:
    assumes nonabort-i: chaos ⊆ i
    assumes (∀ c. (∀ d. ((c || i);(d || i) ⊑ (c;d) || i)))
    shows (c;d) // i ⊑ (c // i);(d // i)
⟨proof⟩

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);e;(d || j) ⊑ (c;e;d) || (i;e;j))))
    shows (c;e;d) // (i;e;j) ⊑ (c // i);e;(d // j)
⟨proof⟩

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⟩

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)))
lemma wrap-rely-guar:
  assumes nonabort-rg: chaos ⊑ rg
  and skippable: rg ⊑ skip
  shows c ⊑ rg ⩲ c // rg
⟨proof⟩
end

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

lemma distribute-rely-iteration:
  assumes nonabort-i: chaos ⊑ i
  assumes (∀ c. (∀ d. ((c || i);(d || i)) ⊑ (c;d) || i)))
  shows (cω,d) // i ⊑ (c // i)ω;(d // i)
⟨proof⟩
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 $c;(d_0 \cap d_1) = (c;d_0) \cap (c;d_1)$ 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 $nil \sqsubseteq nil \parallel nil$ to locale sequential-parallel.

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

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

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

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

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


