Concurrent Refinement Algebra and Rely Quotients

Julian Fell and Ian Hayes and Andrius Velykis

August 16, 2018

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 20

10 Infimum nat lemmas 21

11 Iteration for conjunctive models 22

12 Rely Quotient Operator 24
   12.1 Basic rely quotient .................................... 24
   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 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 ⊑ 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. x ∈ {c, d}}) = f c ∩ f d
⟨proof⟩

Helper lemma for choice over indexed set.

lemma INF-Inf: (∏ x∈X. f x) = (∏ { f x | 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 ∈ X}} = {f (g x) | 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 ∈ X}
  assumes f-Inf-distrib: (∀ c D. P D =⇒ f (c ∩ D) = (∏ { f c d | d. d ∈ D})
  shows f c (∏ x∈X. d x) = (∏ x∈X. f c (d x))

Lemma Inf-distrib-INF:
fixes f :: 'a ⇒ 'a ⇒ 'a
assumes f-Inf-distrib: ∀c D. f c (∧ D) = (∧ {f c d | d ∈ D})
sows f c (∧ x∈X. d x) = (∧ x∈X. f c (d x))

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 ≡ (∀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 =
\textbf{fixes} \texttt{seq :: 'a::refinement-lattice ⇒ 'a ⇒ 'a (infixl 90)}
\textbf{assumes} \texttt{seq-bot [simp]: ⊥ ; c = ⊥}

\textbf{locale} \texttt{nil =}
\textbf{fixes} \texttt{nil :: 'a::refinement-lattice (nil)}

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

\textbf{locale} \texttt{sequential = seq + nil + seq: monoid seq nil}
\textbf{begin}

\textbf{declare} \texttt{seq.assoc [algebra-simps, field-simps]}

\textbf{lemmas} \texttt{seq-assoc = seq.assoc}
\textbf{lemmas} \texttt{seq-nil-right = seq.right-neutral}
\textbf{lemmas} \texttt{seq-nil-left = seq.left-neutral}

\textbf{end}

\textbf{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.

\textbf{locale} \texttt{seq-distrib-left = sequential +}
\textbf{assumes} \texttt{weak-seq-inf-distrib:}
\quad \texttt{(c::'a::refinement-lattice);(d_0 ⊓ d_1) ⊑ (c;d_0 ⊓ c;d_1)}
\textbf{begin}

Left distribution implies sequential composition is monotonic is its right argument

\textbf{lemma} \texttt{seq-mono-right: c_0 ⊑ c_1 ⇒ d ; c_0 ⊑ d ; c_1}
\langle proof \rangle

\textbf{lemma} \texttt{seq-bot-right [simp]: c;⊥ ⊑ c}
\langle proof \rangle

\textbf{end}

\textbf{locale} \texttt{seq-distrib-right = sequential +}
\textbf{assumes} \texttt{Inf-seq-distrib:}

\textbf{7}
\[ (\prod C) : d = (\prod (c::\text{refinement-lattice})\in C. c : d) \]

\begin{proof}
\end{proof}

\textbf{lemma INF-seq-distrib:} \((\prod c\in C. f c) : d = (\prod c\in C. f c ; d)\)

\begin{proof}
\end{proof}

\textbf{lemma inf-seq-distrib:} \((c_0 \sqcap c_1) : d = (c_0 ; d \sqcap c_1 ; d)\)

\begin{proof}
\end{proof}

\textbf{lemma seq-mono-left:} \(c_0 \sqsubseteq c_1 \Rightarrow c_0 ; d \sqsubseteq c_1 ; d\)

\begin{proof}
\end{proof}

\textbf{lemma seq-top [simp]:} \(\top ; c = \top\)

\begin{proof}
\end{proof}

\textbf{primrec seq-power ::} 'a \Rightarrow nat \Rightarrow 'a (\text{infixr} ^{80}) \textbf{where}

\textit{seq-power-0:} \(a ^{0} = \text{nil}\)

\textit{seq-power-Suc:} \(a ^{\text{Suc } n} = a ; (a ^{n})\)

\textbf{notation (latex output)}

seq-power \((\text{\textasciitilde} ) [1000] 1000\)

\textbf{notation (HTML output)}

seq-power \((\text{\textasciitilde} ) [1000] 1000\)

\textbf{lemma seq-power-front:} \((a ^{n}) ; a = a ; (a ^{n})\)

\begin{proof}
\end{proof}

\textbf{lemma seq-power-split-less:} \(i < j \Rightarrow (b ^{j}) = (b ^{i}) ; (b ^{j - i})\)

\begin{proof}
\end{proof}

\end

\textbf{locale seq-distrib = seq-distrib-right + seq-distrib-left}

\begin{proof}
\end{proof}

\textbf{end}

\end

\end

8
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 ≠ {} ⇒ (∩ D) ∥ c = (∩ d∈D. d ∥ c)
⟨proof⟩
lemma \textit{par-INF-distrib}: \( X \neq \{\} \implies c \parallel (\bigcap_{x \in X} d x) = (\bigcap_{x \in X} c \parallel d x) \)

\langle \text{proof} \rangle

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

\langle \text{proof} \rangle

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

\langle \text{proof} \rangle

\textit{lemma inf-par-distrib}: \( (c_0 \cap c_1) \parallel d = (c_0 \parallel d) \cap (c_1 \parallel d) \)

\langle \text{proof} \rangle

\textit{lemma inf-par-distrib2}: \( d \parallel (c_0 \cap c_1) = (d \parallel c_0) \cap (d \parallel c_1) \)

\langle \text{proof} \rangle

\textit{lemma inf-par-product}: \( (a \cap b) \parallel (c \cap d) = (a \parallel c) \cap (a \parallel d) \cap (b \parallel c) \cap (b \parallel d) \)

\langle \text{proof} \rangle

\textit{lemma par-mono}: \( c_1 \sqsubseteq d_1 \implies c_2 \sqsubseteq d_2 \implies c_1 \parallel c_2 \sqsubseteq d_1 \parallel d_2 \)

\langle \text{proof} \rangle

\end

\end

5 Weak Conjunction Operator

\texttt{theory Conjunction}
\texttt{imports Refinement-Lattice}
\texttt{begin}

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

\texttt{locale chaos =}
\texttt{  \fixes chaos :: 'a::refinement-lattice \ (chaos) }

\texttt{locale conj =}
\texttt{  \fixes conj :: 'a::refinement-lattice \Rightarrow 'a \Rightarrow 'a \ (infixl \ 80) }
\texttt{  \assumes conj-bot-right: c \cap \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
conj-commute = conj.commute
conj-idem = conj.idem
conj-chaos = conj.right-neutral
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 ⊓ b) ⊓ (c ⊓ d) = (a ⊓ c) ⊓ (a ⊓ d) ⊓ (b ⊓ c) ⊓ (b ⊓ d)
⟨proof⟩
**Lemma** conj-mono: \(c_0 \sqsubseteq d_0 \implies c_1 \sqsubseteq d_1 \implies c_0 \sqcap c_1 \sqsubseteq d_0 \sqcap d_1\)

⟨proof⟩

**Lemma** conj-mono-left: \(c_0 \sqsubseteq c_1 \implies c_0 \sqcap d \sqsubseteq c_1 \sqcap d\)

⟨proof⟩

**Lemma** conj-mono-right: \(c_0 \sqsubseteq c_1 \implies d \sqcap c_0 \sqsubseteq d \sqcap c_1\)

⟨proof⟩

**Lemma** conj-refine: \(c_0 \sqsubseteq d \implies c_1 \sqsubseteq d \implies c_0 \sqcap c_1 \sqsubseteq d\)

⟨proof⟩

**Lemma** refine-to-conj: \(c \sqsubseteq d_0 \implies c \sqsubseteq d_1 \implies c \sqsubseteq d_0 \sqcap d_1\)

⟨proof⟩

**Lemma** conjoin-non-aborting: \(\text{chaos} \sqsubseteq c \implies d \sqsubseteq d \sqcap c\)

⟨proof⟩

**Lemma** conjunction-sup: \(c \sqcap d \sqsubseteq c \sqcup d\)

⟨proof⟩

**Lemma** conjunction-sup-nonaborting:
assume \(\text{chaos} \sqsubseteq c\) and \(\text{chaos} \sqsubseteq d\)
show \(c \sqcap d = c \sqcup d\)

⟨proof⟩

**Lemma** conjoin-top: \(\text{chaos} \sqsubseteq c \implies c \sqcap \top = \top\)

⟨proof⟩

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**

12
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: ⊤ ⊑ chaos || ⊤
  assumes chaos-par-chaos: chaos ⊑ chaos || chaos
  assumes parallel-interchange: (c_0 || c_1) && (d_0 || d_1) ⊑ (c_0 && d_0) || (c_1 && d_1)
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 && (d_0 || d_1) ⊑ (c && d_0) || (c && d_1) ⟨proof⟩
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
⟨proof⟩

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

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

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

lemma conj-distrib3: c ⊑ c;c ⇒ c ⊑ (d₀ ; d₁) ⊑ (c ⊑ d₀);(c ⊑ d₁)
⟨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]*: \( \{ F \, x \mid x = a \lor x = b \} = \{ F \, a, F \, b \} \)

\[ \text{proof} \]

**locale** *lower-galois-connections*

**begin**

**definition**

\( l\text{-}\text{adjoint} :: \langle 'a::\text{refinement-lattice} \Rightarrow 'a \rangle \Rightarrow \langle 'a \Rightarrow 'a \rangle \ (\Rightarrow [201]) \ 200 \)

**where**

\[ (F^\alpha) \, x \equiv \bigcap \{ y. \, x \sqsubseteq F \, y \} \]

**lemma** *dist-inf-mono*:

**assumes** *distF*:*dist-over-inf F*

**shows** *mono F*

\[ \text{proof} \]

**lemma** *l-cancellation*:

**dist-over-inf F**\(\Rightarrow x \sqsubseteq (F \circ F^\alpha) \, x \)

\[ \text{proof} \]

**lemma** *l-galois-connection*:

**dist-over-inf F**\(\Rightarrow \)\( (F^\alpha) \, x \sqsubseteq y \)\(\iff x \sqsubseteq F \, y \)

\[ \text{proof} \]

**lemma** *v-simple-fusion*:

\( \text{mono G} \Rightarrow \forall x. \ ((F \circ G) \, x \sqsubseteq (H \circ F) \, x) \Rightarrow F \ (\text{gfp} \, G) \sqsubseteq \text{gfp} \, H \)

\[ \text{proof} \]

### 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*: \( \forall x. \ ((H \circ F) \, x \sqsubseteq (F \circ G) \, x) \)

**shows** *gfp H \subseteq F \ (gfp \, G) *

\[ \text{proof} \]

**theorem** *fusion-gfp-eq*:
assumes monoH: mono H and monoG: mono G
and distF: dist-over-inf F
and fgh-comp: \( \land x. (F \circ G) x = (H \circ F) x \)
shows \( F \ (gfp \ G) = gfp \ H \)

(proof)

end

7.3 Upper Galois connections

locale upper-galois-connections
begin

definition u-adjoint :: ('a::refinement-lattice \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) (\# [201] 200)
where
\( (F\#) \ x \equiv \bigcup \{y. F \ y \sqsubseteq x\} \)

lemma dist-sup-mono:
  assumes distF: dist-over-sup F
  shows mono F
  (proof)

lemma u-cancellation: dist-over-sup F \implies (F \circ F\#) x \sqsubseteq x
  (proof)

lemma u-galois-connection: dist-over-sup F \implies (F x \sqsubseteq y) \iff (x \sqsubseteq (F\#) y)
  (proof)

lemma u-simple-fusion: mono H \implies \forall x. ((F \circ G) x \sqsubseteq (G \circ H) x) \implies lfp F \sqsubseteq G (lfp H)
  (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: \( \land x. ((F \circ G) x \sqsubseteq (H \circ F) x) \)
  shows \( F \ (lfp \ G) \sqsubseteq (lfp H) \)
  (proof)
theorem fusion-lfp-eq:
assumes monoH: mono H and monoG: mono G
and distF: dist-over-sup F
and fgh-comp: \( \bigwedge x. ((F \circ G) x = (H \circ F) x) \)
shows \( F (\text{lfp } G) = (\text{lfp } H) \)

⟨proof⟩

end

end

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
where
  \( c^\omega \equiv \text{lfp } (\lambda x. \text{nil } \sqcap c; x) \)

lemma iter-step-mono: mono (\( \lambda x. \text{nil } \sqcap c; x \))
  ⟨proof⟩

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

theorem iter-unfold: \( c^\omega = \text{nil } \sqcap c; c^\omega \)
  ⟨proof⟩

lemma iter-induct-nil: \( \text{nil } \sqcap c; x \sqsubseteq x \implies c^\omega \sqsubseteq x \)
  ⟨proof⟩
lemma iter0: \( c^\omega \sqsubseteq \text{nil} \)
\langle proof \rangle

lemma iter1: \( c^\omega \sqsubseteq c \)
\langle proof \rangle

lemma iter2 [simp]: \( c^\omega ; c^\omega = c^\omega \)
\langle proof \rangle

lemma iter-mono: \( c \sqsubseteq d \implies c^\omega \sqsubseteq d^\omega \)
\langle proof \rangle

lemma iter-abort: \( \bot = \text{nil}^\omega \)
\langle proof \rangle

lemma nil-iter: \( \top^\omega = \text{nil} \)
\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 \Rightarrow 'a (-[^101] 100)
where
c* = gfp (\lambda x. \text{nil} \sqcap c;x)

lemma fin-iter-step-mono: mono (\lambda x. \text{nil} \sqcap c;x)
\langle proof \rangle

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

lemma fiter-unfold: \( c^* = \text{nil} \sqcap c;c^* \)
\langle proof \rangle

lemma fiter-induct-nil: \( x \sqsubseteq \text{nil} \sqcap c;x \implies x \sqsubseteq c^* \)
\langle proof \rangle
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
begins

definition infiter :: 'a ⇒ 'a (¬∞ [105] 106)
where
  \( c^{\infty} \equiv \text{lfp} (\lambda x. c;x) \)

lemma infiter-step-mono: mono (\lambda x. c;x)
  ⟨proof⟩

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

theorem infiter-unfold: \( c^{\infty} = c;c^{\infty} \)
  ⟨proof⟩

lemma infiter-induct: \( c;x \sqsubseteq x \implies c^{\infty} \sqsubseteq x \)
\begin{proof}

\textbf{Theorem} \textit{infiter-unfold-any}: \(c^\infty = (c \uparrow i) \comp c^\infty\)
\end{proof}

\textbf{Lemma} \textit{infiter-annil}: \(c^\infty); x = c^\infty\)
\end{proof}

\section{Combined iteration}

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

\begin{locale}
\textbf{locale} \textit{iteration} = \textit{finite-or-infinite-iteration} + \textit{finite-iteration} + \textit{infinite-iteration}
\begin{proof}
\textbf{lemma} \textit{refine-iter}: \(c^\omega \subseteq c^\star\)
\end{proof}

\textbf{lemma} \textit{iter-absorption} \textit{[simp]}: \((c^\omega)^\star = c^\omega\)
\end{proof}

\textbf{lemma} \textit{infiter-inf-top}: \(c^\infty = c^\omega; \top\)
\end{proof}

\textbf{lemma} \textit{infiter-fiter-top}:
\begin{proof}
shows \(c^\infty \subseteq c^\star; \top\)
\end{proof}

\textbf{lemma} \textit{inf-ref-infiter}: \(c^\omega \subseteq c^\infty\)
\end{proof}

\end{proof}

\end{locale}

\section{Sequential composition for conjunctive models}

\begin{proof}

\textbf{theory} \textit{Conjunctive-Sequential}
\end{proof}

\begin{proof}

\textbf{imports} \textit{Sequential}
\end{proof}

20
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: $D \not= \{\} \implies c;\bigcap D = (\bigcap d \in D. c ; d)$
begin

sublocale seq-distrib
  ⟨proof⟩

lemma seq-INF-distrib: $X \not= \{\} \implies c; (\bigcap x \in X. d x) = (\bigcap x \in X. c ; d x)$$
  ⟨proof⟩

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

lemma INF-INF-seq-distrib: $Y \not= \{\} \implies (\bigcap x \in X. c x) ; (\bigcap y \in Y. d y) = (\bigcap x \in X. \bigcap y \in Y. c x ; d y)$$
  ⟨proof⟩

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

end

end

10 Infimum nat lemmas

theory Infimum-Nat
imports
  Refinement-Lattice
locale infimum-nat

begin

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

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

11 Iteration for conjunctive models

theory Conjunctive-Iteration
imports
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⟩

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

begin

lemma fiter-seq-choice: c* = (∏ i::nat. c i:i)
⟨proof⟩

lemma fiter-seq-choice-nonempty: c ; c* = (∏ i∈{i. 0 < i}. c i:i)
⟨proof⟩

end

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

begin

lemma conj-distrib4: c* ⊑ d* ⊑ (c ⊓ 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.

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 || 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 || i.
locale rely-quotient = par-distrib + conjunction-parallel
begin

definition
  rely-quotient :: 'a ⇒ 'a ⇒ 'a (infixl '⁄/ 85)
where
  c ⁄ i ≡ ∩ { d. (c ⊆ d || i)}

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

lemma quotient-identity: c ⁄ 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 ⁄ ⊥) || ⊥ equals ⊥, it does not refine
c in general.

theorem rely-quotient:
  assumes nonabort-i: chaos ⊆ i
  shows c ⊆ (c ⁄ i) || i
(proof)

The following theorem represents the Galois connection between the parallel op-
erator (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 ⊆ i
  shows c ⁄ i ⊆ d ⇐⇒ c ⊆ d || i
(proof)

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

lemma rely-mono:
  assumes c-refsto-d: c ⊆ d
  shows (c ⁄ i) ⊆ (d ⁄ i)
(proof)

Refining the “denominator” in a quotient, gives a reverse refinement for the quo-
tients. 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 \textit{par-nonabort}:  
\textbf{assumes} \textit{nonabort-i}: \textit{chaos} \sqsubseteq \textit{i}  
\textbf{assumes} \textit{nonabort-j}: \textit{chaos} \sqsubseteq \textit{j}  
\textbf{shows} \textit{chaos} \sqsubseteq \textit{i} \parallel \textit{j}  
⟨\textit{proof}⟩

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

lemma \textit{nested-rely}:  
\textbf{assumes} \textit{j-nonabort}: \textit{chaos} \sqsubseteq \textit{j}  
\textbf{shows} ((\textit{c} \parallel \textit{j}) \parallel \textit{i}) = \textit{c} \parallel (\textit{i} \parallel \textit{j})  
⟨\textit{proof}⟩

end

12.2 Distributed rely quotient

locale \textit{rely-distrib} = \textit{rely-quotient} + \textit{conjunction-sequential}  
\begin{proof}

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 \textit{introduce-parallel}:  
\textbf{assumes} \textit{nonabort-i}: \textit{chaos} \sqsubseteq \textit{i}  
\textbf{assumes} \textit{nonabort-j}: \textit{chaos} \sqsubseteq \textit{j}  
\textbf{shows} \textit{c} \sqcap \textit{d} \sqsubseteq (\textit{j} \sqcap (\textit{c} \parallel \textit{i})) \parallel (\textit{i} \sqcap (\textit{d} \parallel \textit{j}))  
⟨\textit{proof}⟩

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

lemma \textit{distribute-rely-conjunction}:  
\textbf{assumes} \textit{nonabort-i}: \textit{chaos} \sqsubseteq \textit{i}  
\textbf{shows} (\textit{c} \sqcap \textit{d}) \parallel \textit{i} \sqsubseteq (\textit{c} \parallel \textit{i}) \sqcap (\textit{d} \parallel \textit{i})  
⟨\textit{proof}⟩

lemma \textit{distribute-rely-choice}:  
\textbf{assumes} \textit{nonabort-i}: \textit{chaos} \sqsubseteq \textit{i}  
\textbf{shows} (\textit{c} \sqcap \textit{d}) \parallel \textit{i} \sqsubseteq (\textit{c} \parallel \textit{i}) \cap (\textit{d} \parallel \textit{i})  
⟨\textit{proof}⟩

lemma \textit{distribute-rely-parallel1}:

26
\begin{align*}
\text{assumes } & \text{nonabort-i: } \text{chaos } \subseteq i \\
\text{assumes } & \text{nonabort-j: } \text{chaos } \subseteq j \\
\text{shows } & (c \parallel d) \parallel (i \parallel j) \subseteq (c \parallel (d \parallel i)) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma} \text{distribute-rely-parallel2:}

\begin{align*}
\text{assumes } & \text{nonabort-i: } \text{chaos } \subseteq i \\
\text{assumes } & \text{i-par-i: } i \parallel i \subseteq i \\
\text{shows } & (c \parallel d) \parallel i \subseteq (c \parallel (d \parallel i)) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma} \text{distribute-rely-sequential:}

\begin{align*}
\text{assumes } & \text{nonabort-i: } \text{chaos } \subseteq i \\
\text{assumes } & (\forall c. (\forall d. ((c \parallel i) \parallel (d \parallel i)) \subseteq (c;d) \parallel (i;e;j)))) \\
\text{shows } & (c;e;d) \parallel (i;e;j) \subseteq (c \parallel (d \parallel i);(d \parallel i)) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma} \text{distribute-rely-sequential-event:}

\begin{align*}
\text{assumes } & \text{nonabort-i: } \text{chaos } \subseteq i \\
\text{assumes } & \text{nonabort-j: } \text{chaos } \subseteq j \\
\text{assumes } & \text{nonabort-e: } \text{chaos } \subseteq e \\
\text{assumes } & (\forall c. (\forall d. ((c \parallel i) : c;e:d \parallel (i;e;j)))) \\
\text{shows } & (c;e;d) \parallel (i;e;j) \subseteq (c \parallel (d \parallel i);(d \parallel i)) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma} \text{introduce-parallel-with-rely:}

\begin{align*}
\text{assumes } & \text{nonabort-i: } \text{chaos } \subseteq i \\
\text{assumes } & \text{nonabort-j0: } \text{chaos } \subseteq j_0 \\
\text{assumes } & \text{nonabort-j1: } \text{chaos } \subseteq j_1 \\
\text{shows } & (c \parallel d) \parallel i \subseteq (j_1 \parallel (c \parallel (j_0 \parallel i))) \parallel (j_0 \parallel (d \parallel (j_1 \parallel i))) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma} \text{introduce-parallel-with-rely-guarantee:}

\begin{align*}
\text{assumes } & \text{nonabort-i: } \text{chaos } \subseteq i \\
\text{assumes } & \text{nonabort-j0: } \text{chaos } \subseteq j_0 \\
\text{assumes } & \text{nonabort-j1: } \text{chaos } \subseteq j_1 \\
\text{shows } & (j_1 \parallel j_0) \parallel (c \parallel d) \parallel i \subseteq (j_1 \parallel (c \parallel (j_0 \parallel i))) \parallel (j_0 \parallel (d \parallel (j_1 \parallel i))) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma} \text{wrap-rely-guar:}

\begin{align*}
\text{assumes } & \text{nonabort-rg: } \text{chaos } \subseteq \text{rg} \\
\text{and } & \text{skippable: } \text{rg } \subseteq \text{skip} \\
\text{shows } & c \subseteq \text{rg} \parallel c \parallel \text{rg} \\
\langle \text{proof} \rangle
\end{align*}
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
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 \( c;(d_0 \sqcap d_1) = (c;d_0) \sqcap (c;d_1) \) 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 $(\prod C) \parallel d = (\prod c \in C.c \parallel d)$ but in Section 4
we assume this only for non-empty $C$ and furthermore assume that parallel
is abort strict, i.e. $\perp \parallel c = c$.

3. The earlier paper assumes $c \mathbin{\cap} (\bigsqcup D) = (\bigsqcup d \in D.c \mathbin{\cap} d)$. 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 $c \mathbin{\cap} \perp = \perp$.

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 $D \neq \{\} \Rightarrow c ; \prod D = (\prod d \in D.c ; d)$. 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

Geldrop, P. Hoogendijk, E. Voermans, and J. van der Woude. Fixed-point
Program Construction Group.

a wide-spectrum language with concurrency. Formal Aspects of Computing,

