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

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 *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 :: $'a \Rightarrow {}'a \Rightarrow bool$ (**infix** $\sqsubseteq$ *50*)
where
 $c \sqsubseteq d \equiv less\text{-}eq\ c\ d$

abbreviation
 refine-strict :: $'a \Rightarrow {}'a \Rightarrow bool$ (**infix** $\sqsubset$ *50*)
where
 $c \sqsubset d \equiv less\ c\ d$

Non-deterministic choice is monotonic in both arguments

lemma *inf-mono-left*: $a \sqsubseteq b \Longrightarrow a \sqcap c \sqsubseteq b \sqcap c$
 $\langle proof \rangle$

lemma *inf-mono-right*: $c \sqsubseteq d \Longrightarrow a \sqcap c \sqsubseteq a \sqcap d$
 $\langle proof \rangle$

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

lemma *Inf2-inf*: $\bigsqcap \{ f\,x \mid x.\ x \in \{c, d\} \} = f\,c \sqcap f\,d$
$\langle proof \rangle$

Helper lemma for choice over indexed set.

lemma *INF-Inf*: $(\bigsqcap x \in X.\ f\,x) = (\bigsqcap \{ f\,x \mid x.\ x \in X \})$
 $\langle proof \rangle$

lemma (**in** −) *INF-absorb-args*: $(\bigsqcap i\,j.\ (f::nat \Rightarrow {}'c::complete\text{-}lattice)\ (i + j)) = (\bigsqcap k.\ f\,k)$
$\langle proof \rangle$

lemma (**in** −) *nested-Collect*: $\{ f\,y \mid y.\ y \in \{ g\,x \mid x.\ x \in X \} \} = \{ f\,(g\,x) \mid x.\ x \in X \}$
 $\langle proof \rangle$

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 $f :: {}'a \Rightarrow {}'a \Rightarrow {}'a$
 assumes *qual*: $P\ \{ d\,x \mid x.\ x \in X \}$
 assumes *f-Inf-distrib*: $\bigwedge c\ D.\ P\ D \Longrightarrow f\,c\ (\bigsqcap D) = \bigsqcap \{ f\,c\,d \mid d\ .\ d \in D\ \}$

shows $f\,c\ (\bigsqcap x{\in}X.\ d\ x) = (\bigsqcap x{\in}X.\ f\,c\ (d\ x))$
⟨*proof*⟩

lemma *Inf-distrib-INF*:
  fixes $f :: {'}a \Rightarrow {'}a \Rightarrow {'}a$
  assumes *f-Inf-distrib*: $\bigwedge c\ D.\ f\,c\ (\bigsqcap\ D) = \bigsqcap\ \{f\,c\,d \mid d\ .\ d \in D\ \}$
  shows $f\,c\ (\bigsqcap x{\in}X.\ d\ x) = (\bigsqcap x{\in}X.\ f\,c\ (d\ x))$
  ⟨*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 :: $({'}a{::}refinement\text{-}lattice \Rightarrow {'}a) \Rightarrow bool$
where
  dist-over-sup $F \equiv (\forall\ X\ .\ F\ (\bigsqcup\ X) = (\bigsqcup x{\in}X.\ F\ (x)))$

abbreviation
  dist-over-inf :: $({'}a{::}refinement\text{-}lattice \Rightarrow {'}a) \Rightarrow bool$
where
  dist-over-inf $F \equiv (\forall\ X\ .\ F\ (\bigsqcap\ X) = (\bigsqcap x{\in}X.\ F\ (x)))$

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.

6

locale *seq* =
  fixes *seq* :: $'a$::*refinement-lattice* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ (**infixl** ; *90*)
  assumes *seq-bot* [*simp*]: $\bot$; $c = \bot$

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*$);(d_0 \sqcap d_1) \sqsubseteq (c;d_0 \sqcap c;d_1)$
begin

Left distribution implies sequential composition is monotonic is its right argument

lemma *seq-mono-right*: $c_0 \sqsubseteq c_1 \Longrightarrow d ; c_0 \sqsubseteq d ; c_1$
  $\langle proof \rangle$

lemma *seq-bot-right* [*simp*]: $c;\bot \sqsubseteq c$
  $\langle proof \rangle$

end

locale *seq-distrib-right* = *sequential* +

assumes *Inf-seq-distrib*:

$\quad (\bigsqcap C) \; ; \; d = (\bigsqcap (c ::\,'a::\textit{refinement-lattice}) \in C. \; c \; ; \; d)$

begin

lemma *INF-seq-distrib*: $(\bigsqcap c \in C. \, f \, c) \; ; \; d = (\bigsqcap c \in C. \, f \, c \; ; \; d)$
$\quad \langle \textit{proof} \rangle$

lemma *inf-seq-distrib*: $(c_0 \sqcap c_1) \; ; \; d = (c_0 \; ; \; d \sqcap c_1 \; ; \; d)$
$\langle \textit{proof} \rangle$

lemma *seq-mono-left*: $c_0 \sqsubseteq c_1 \Longrightarrow c_0 \; ; \; d \sqsubseteq c_1 \; ; \; d$
$\quad \langle \textit{proof} \rangle$

lemma *seq-top* [*simp*]: $\top \; ; \; c = \top$
$\langle \textit{proof} \rangle$

primrec *seq-power* :: $'a \Rightarrow \textit{nat} \Rightarrow {}'a$ (**infixr** $^{;\wedge}$ *80*) **where**
$\quad$ *seq-power-0*: $a ^{;\wedge} 0 = \textit{nil}$
$\mid$ *seq-power-Suc*: $a ^{;\wedge} \textit{Suc} \; n = a \; ; \; (a ^{;\wedge} n)$

notation (*latex* **output**)
$\quad$ *seq-power* $((\text{-}^{\text{-}}) \, [\textit{1000}] \, \textit{1000})$

notation (*HTML* **output**)
$\quad$ *seq-power* $((\text{-}^{\text{-}}) \, [\textit{1000}] \, \textit{1000})$

lemma *seq-power-front*: $(a ^{;\wedge} n) \; ; \; a = a \; ; \; (a ^{;\wedge} n)$
$\quad \langle \textit{proof} \rangle$

lemma *seq-power-split-less*: $i < j \Longrightarrow (b ^{;\wedge} j) = (b ^{;\wedge} i) \; ; \; (b ^{;\wedge} (j - i))$
$\langle \textit{proof} \rangle$

end

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

lemma *seq-mono*: $c_1 \sqsubseteq d_1 \Longrightarrow c_2 \sqsubseteq d_2 \Longrightarrow c_1 ; c_2 \sqsubseteq d_1 ; d_2$
$\quad \langle \textit{proof} \rangle$

end

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* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ (**infixl** $\parallel$ *75*)
  assumes *abort-par*: $\bot \parallel c = \bot$

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

begin

lemma *Inf-par-distrib*: $D \neq \{\} \implies (\bigsqcap D) \parallel c = (\bigsqcap d \in D.\ d \parallel c)$
$\langle proof \rangle$

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

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

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

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

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

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

lemma *par-mono*: $c_1 \sqsubseteq d_1 \implies c_2 \sqsubseteq d_2 \implies c_1 \parallel c_2 \sqsubseteq d_1 \parallel d_2$
$\langle proof \rangle$

end
end

5 Weak Conjunction Operator

theory *Conjunction*
imports *Refinement-Lattice*
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.

locale *chaos* =
  fixes *chaos* :: $'a$::*refinement-lattice*   (*chaos*)

locale *conj* =
  fixes *conj* :: $'a$::*refinement-lattice* $\Rightarrow 'a \Rightarrow 'a$ (**infixl** $\Cap$ *80*)

assumes *conj-bot-right*: $c \text{ⓜ} \bot = \bot$

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*]: $\bot \text{ⓜ} c = \bot$
$\langle proof \rangle$

lemma *conj-not-bot*: $a \text{ⓜ} b \neq \bot \implies a \neq \bot \land b \neq \bot$
 $\langle proof \rangle$

lemma *conj-distrib1*: $c \text{ⓜ} (d_0 \text{ⓜ} d_1) = (c \text{ⓜ} d_0) \text{ⓜ} (c \text{ⓜ} d_1)$
 $\langle proof \rangle$

end

5.1 Distributed weak conjunction

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

locale *conj-distrib* $=$ *conjunction* $+$
 assumes *Inf-conj-distrib*: $D \neq \{\} \implies (\bigsqcap D) \text{ⓜ} c = (\bigsqcap d \in D.\ d \text{ⓜ} c)$

begin

lemma *conj-Inf-distrib*: $D \neq \{\} \implies c \text{ⓜ} (\bigsqcap D) = (\bigsqcap d \in D.\ c \text{ⓜ} d)$
 $\langle proof \rangle$

lemma *inf-conj-distrib*: $(c_0 \sqcap c_1) \text{ⓜ} d = (c_0 \text{ⓜ} d) \sqcap (c_1 \text{ⓜ} d)$
$\langle proof \rangle$

lemma *inf-conj-product*: $(a \sqcap b) \Cap (c \sqcap d) = (a \Cap c) \sqcap (a \Cap d) \sqcap (b \Cap c) \sqcap (b \Cap d)$
  ⟨*proof*⟩

lemma *conj-mono*: $c_0 \sqsubseteq d_0 \implies c_1 \sqsubseteq d_1 \implies c_0 \Cap c_1 \sqsubseteq d_0 \Cap d_1$
  ⟨*proof*⟩

lemma *conj-mono-left*: $c_0 \sqsubseteq c_1 \implies c_0 \Cap d \sqsubseteq c_1 \Cap d$
  ⟨*proof*⟩

lemma *conj-mono-right*: $c_0 \sqsubseteq c_1 \implies d \Cap c_0 \sqsubseteq d \Cap c_1$
  ⟨*proof*⟩

lemma *conj-refine*: $c_0 \sqsubseteq d \implies c_1 \sqsubseteq d \implies c_0 \Cap c_1 \sqsubseteq d$
  ⟨*proof*⟩

lemma *refine-to-conj*: $c \sqsubseteq d_0 \implies c \sqsubseteq d_1 \implies c \sqsubseteq d_0 \Cap d_1$
  ⟨*proof*⟩

lemma *conjoin-non-aborting*: $chaos \sqsubseteq c \implies d \sqsubseteq d \Cap c$
  ⟨*proof*⟩

lemma *conjunction-sup*: $c \Cap d \sqsubseteq c \sqcup d$
  ⟨*proof*⟩

lemma *conjunction-sup-nonaborting*:
  assumes $chaos \sqsubseteq c$ **and** $chaos \sqsubseteq d$
  shows $c \Cap d = c \sqcup d$
⟨*proof*⟩

lemma *conjoin-top*: $chaos \sqsubseteq c \implies c \Cap \top = \top$
⟨*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
Parallel
begin

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*: $nil \parallel nil \sqsubseteq nil$
  and *skip-nil*: $skip \sqsubseteq nil$
  and *skip-skip*: $skip \sqsubseteq skip;skip$
begin

lemma *nil-absorb*: $nil \parallel nil = nil$ $\langle proof \rangle$

lemma *skip-absorb* [*simp*]: $skip;skip = skip$
$\langle proof \rangle$

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*: $\top \sqsubseteq chaos \parallel \top$
  assumes *chaos-par-chaos*: $chaos \sqsubseteq chaos \parallel chaos$
  assumes *parallel-interchange*: $(c_0 \parallel c_1) \Cap (d_0 \parallel d_1) \sqsubseteq (c_0 \Cap d_0) \parallel (c_1 \Cap d_1)$
begin

lemma *chaos-skip*: $chaos \sqsubseteq skip$
$\langle proof \rangle$

lemma *chaos-par-chaos-eq*: $chaos = chaos \parallel chaos$
 $\langle proof \rangle$

lemma *nonabort-par-top*: $chaos \sqsubseteq c \implies c \parallel \top = \top$
 $\langle proof \rangle$

lemma *skip-conj-top*: $skip \Cap \top = \top$
$\langle proof \rangle$

lemma *conj-distrib2*: $c \sqsubseteq c \parallel c \implies c \Cap (d_0 \parallel d_1) \sqsubseteq (c \Cap d_0) \parallel (c \Cap d_1)$
$\langle proof \rangle$

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*: $chaos \sqsubseteq chaos;chaos$
  assumes *sequential-interchange*: $(c_0;c_1) \Cap (d_0;d_1) \sqsubseteq (c_0 \Cap d_0);(c_1 \Cap d_1)$
begin

lemma *chaos-nil*: $chaos \sqsubseteq nil$
  $\langle proof \rangle$

lemma *chaos-seq-absorb*: $chaos = chaos;chaos$
$\langle proof \rangle$

lemma *seq-bot-conj*: $c;\bot \Cap d \sqsubseteq (c \Cap d);\bot$
  $\langle proof \rangle$

lemma *conj-seq-bot-right* [*simp*]: $c;\bot \Cap c = c;\bot$
$\langle proof \rangle$

lemma *conj-distrib3*: $c \sqsubseteq c;c \Longrightarrow c \Cap (d_0 ; d_1) \sqsubseteq (c \Cap d_0);(c \Cap d_1)$
$\langle proof \rangle$

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*]: $\{F\ x\ |x.\ x = a \lor x = b\} = \{F\ a, F\ b\}$
 ⟨*proof*⟩

locale *lower-galois-connections*
begin

definition
 l-adjoint :: ($'a$::*refinement-lattice* $\Rightarrow$ $'a$) $\Rightarrow$ ($'a \Rightarrow$ $'a$) (-$^\flat$ [201] 200)
where
 $(F^\flat)\ x \equiv \bigcap \{y.\ x \sqsubseteq F\ y\}$

lemma *dist-inf-mono*:
 assumes *distF*: *dist-over-inf F*
 shows *mono F*
⟨*proof*⟩

lemma *l-cancellation*: *dist-over-inf F* $\Longrightarrow x \sqsubseteq (F \circ F^\flat)\ x$
⟨*proof*⟩

lemma *l-galois-connection*: *dist-over-inf F* $\Longrightarrow ((F^\flat)\ x \sqsubseteq y) \longleftrightarrow (x \sqsubseteq F\ y)$
⟨*proof*⟩

lemma *v-simple-fusion*: *mono G* $\Longrightarrow \forall x.\ ((F \circ G)\ x \sqsubseteq (H \circ F)\ x) \Longrightarrow F\ (gfp\ G) \sqsubseteq gfp\ H$
 ⟨*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*: $\bigwedge x.\ ((H \circ F)\ x \sqsubseteq (F \circ G)\ x)$
 shows *gfp H* $\sqsubseteq F\ (gfp\ G)$
⟨*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 $F\ (gfp\ G) = gfp\ H$
$\langle proof \rangle$

end

7.3   Upper Galois connections

locale *upper-galois-connections*
begin

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

lemma *dist-sup-mono*:
  assumes *distF*: *dist-over-sup F*
  shows *mono F*
$\langle proof \rangle$

lemma *u-cancellation*: *dist-over-sup* $F \implies (F \circ F^{\#})\ x \sqsubseteq x$
$\langle proof \rangle$

lemma *u-galois-connection*: *dist-over-sup* $F \implies (F\ x \sqsubseteq y) \longleftrightarrow (x \sqsubseteq (F^{\#})\ y)$
$\langle proof \rangle$

lemma *u-simple-fusion*: *mono* $H \implies \forall x.\ ((F \circ G)\ x \sqsubseteq (G \circ H)\ x) \implies lfp\ F \sqsubseteq G\ (lfp\ H)$
  $\langle proof \rangle$

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*: $\bigwedge x.\ ((F \circ G)\ x \sqsubseteq (H \circ F)\ x)$

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 $F\ (lfp\ G) = (lfp\ H)$
⟨*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 :: $'a \Rightarrow 'a$ (-$^{\omega}$ [*103*] *102*)
where
  $c^{\omega} \equiv lfp\ (\lambda\ x.\ nil \sqcap c;x)$

lemma *iter-step-mono*: *mono* $(\lambda\ x.\ nil \sqcap c;x)$
  ⟨*proof*⟩

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

theorem *iter-unfold*: $c^{\omega} = nil \sqcap c;c^{\omega}$
  ⟨*proof*⟩

lemma *iter-induct-nil*: $nil \sqcap c;x \sqsubseteq x \implies c^\omega \sqsubseteq x$
  $\langle proof \rangle$

lemma *iter0*: $c^\omega \sqsubseteq nil$
  $\langle proof \rangle$

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*: $\bot = nil^\omega$
  $\langle proof \rangle$

lemma *nil-iter*: $\top^\omega = nil$
  $\langle proof \rangle$

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
 $c^\star \equiv gfp (\lambda x. nil \sqcap c;x)$

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

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

lemma *fiter-unfold*: $c^\star = nil \sqcap c;c^\star$
  $\langle proof \rangle$

lemma *fiter-induct-nil*: $x \sqsubseteq nil \sqcap c;x \Longrightarrow x \sqsubseteq c^\star$
⟨*proof*⟩

lemma *fiter0*: $c^\star \sqsubseteq nil$
⟨*proof*⟩

lemma *fiter1*: $c^\star \sqsubseteq c$
⟨*proof*⟩

lemma *fiter-induct-eq*: $c^\star;d = gfp\ (\lambda\ x.\ c;x \sqcap d)$
⟨*proof*⟩

theorem *fiter-induct*: $x \sqsubseteq d \sqcap c;x \Longrightarrow x \sqsubseteq c^\star;d$
⟨*proof*⟩

lemma *fiter2* [*simp*]: $c^\star;c^\star = c^\star$
⟨*proof*⟩

lemma *fiter3* [*simp*]: $(c^\star)^\star = c^\star$
⟨*proof*⟩

lemma *fiter-mono*: $c \sqsubseteq d \Longrightarrow c^\star \sqsubseteq d^\star$
⟨*proof*⟩

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 :: $'a \Rightarrow 'a$ (*-$^\infty$* [*105*] *106*)
where
 $c^\infty \equiv lfp\ (\lambda\ x.\ c;x)$

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*⟩

19

lemma *infiter-induct*: $c;x \sqsubseteq x \Longrightarrow c^\infty \sqsubseteq x$
⟨*proof*⟩

theorem *infiter-unfold-any*: $c^\infty = (c \mathbin{;\wedge} i) ; c^\infty$
⟨*proof*⟩

lemma *infiter-annil*: $c^\infty;x = c^\infty$
⟨*proof*⟩

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*: $c^\omega \sqsubseteq c^\star$
⟨*proof*⟩

lemma *iter-absorption* [*simp*]: $(c^\omega)^\star = c^\omega$
⟨*proof*⟩

lemma *infiter-inf-top*: $c^\infty = c^\omega ; \top$
⟨*proof*⟩

lemma *infiter-fiter-top*:
  shows $c^\infty \sqsubseteq c^\star ; \top$
  ⟨*proof*⟩

lemma *inf-ref-infiter*: $c^\omega \sqsubseteq c^\infty$
  ⟨*proof*⟩

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*: $c;(d_0 \sqcap d_1) = c;d_0 \sqcap c;d_1$
begin

sublocale *seq-distrib-left*
  ⟨*proof*⟩
end

locale *seq-infinite-conjunctive* = *seq-distrib-right* +
  assumes *seq-Inf-distrib*: $D \neq \{\} \Longrightarrow c \; ; \bigsqcap D = (\bigsqcap d \in D. \; c \; ; d)$
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 $f :: nat \Rightarrow nat \Rightarrow {'a}::refinement\text{-}lattice$
  shows $(\bigsqcap j.\, f\, i\, j) =$
    $(\bigsqcap j \in \{j.\, i = j\}.\, f\, i\, j) \sqcap$
    $(\bigsqcap j \in \{j.\, i < j\}.\, f\, i\, j) \sqcap$
    $(\bigsqcap j \in \{j.\, j < i\}.\, f\, i\, j)$
$\langle proof \rangle$

lemma *INF-INF-partition-nat3*:
  fixes $f :: nat \Rightarrow nat \Rightarrow {'a}::refinement\text{-}lattice$
  shows $(\bigsqcap i.\, \bigsqcap j.\, f\, i\, j) =$
    $(\bigsqcap i.\, \bigsqcap j \in \{j.\, i = j\}.\, f\, i\, j) \sqcap$
    $(\bigsqcap i.\, \bigsqcap j \in \{j.\, i < j\}.\, f\, i\, j) \sqcap$
    $(\bigsqcap i.\, \bigsqcap j \in \{j.\, j < i\}.\, f\, i\, j)$
$\langle proof \rangle$

lemma *INF-nat-shift*: $(\bigsqcap i \in \{i.\, 0 < i\}.\, f\, i) = (\bigsqcap i.\, f\, (Suc\, i))$
  $\langle proof \rangle$

lemma *INF-nat-minus*:
  fixes $f :: nat \Rightarrow {'a}::refinement\text{-}lattice$
  shows $(\bigsqcap j \in \{j.\, i < j\}.\, f\, (j - i)) = (\bigsqcap k \in \{k.\, 0 < k\}.\, f\, k)$
  $\langle proof \rangle$

lemma *INF-INF-guarded-switch*:
  fixes $f :: nat \Rightarrow nat \Rightarrow {'a}::refinement\text{-}lattice$
  shows $(\bigsqcap i.\, \bigsqcap j \in \{j.\, j < i\}.\, f\, j\, (i - j)) = (\bigsqcap j.\, \bigsqcap i \in \{i.\, j < i\}.\, f\, j\, (i - j))$
$\langle proof \rangle$

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*: $c^\omega = c^\star \sqcap c^\infty$
$\langle proof \rangle$

lemma *iter-induct-isolate*: $c^\star;d \sqcap c^\infty = lfp\ (\lambda\ x.\ d \sqcap c;x)$
$\langle proof \rangle$

lemma *iter-induct-eq*: $c^\omega;d = lfp\ (\lambda\ x.\ d \sqcap c;x)$
$\langle proof \rangle$

lemma *iter-induct*: $d \sqcap c;x \sqsubseteq x \Longrightarrow c^\omega;d \sqsubseteq x$
  $\langle proof \rangle$

lemma *iter-isolate*: $c^\star;d \sqcap c^\infty = c^\omega;d$
  $\langle proof \rangle$

lemma *iter-isolate2*: $c;c^\star;d \sqcap c^\infty = c;c^\omega;d$
  $\langle proof \rangle$

lemma *iter-decomp*: $(c \sqcap d)^\omega = c^\omega;(d;c^\omega)^\omega$
$\langle proof \rangle$

lemma *iter-leapfrog-var*: $(c;d)^\omega;c \sqsubseteq c;(d;c)^\omega$
$\langle proof \rangle$

lemma *iter-leapfrog*: $c;(d;c)^\omega = (c;d)^\omega;c$
$\langle proof \rangle$

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

23

⟨*proof*⟩

end

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

begin

lemma *fiter-seq-choice*: $c^\star = (\bigsqcap i::nat.\ c\ ;^\wedge i)$
⟨*proof*⟩

lemma *fiter-seq-choice-nonempty*: $c\ ;\ c^\star = (\bigsqcap i\in\{i.\ 0 < i\}.\ c\ ;^\wedge i)$
⟨*proof*⟩

end

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

begin

lemma *conj-distrib4*: $c^\star \Cap d^\star \sqsubseteq (c \Cap d)^\star$
⟨*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 $d \parallel i$. 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
 $c \mathbin{//} i \equiv \bigsqcap \{\, d.\ (c \sqsubseteq d \parallel i)\}$

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

lemma *quotient-identity*: $c \mathbin{//} skip = c$
$\langle proof \rangle$

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 $(c \mathbin{//} \bot) \parallel \bot$ equals $\bot$, it does not refine c in general.

theorem *rely-quotient*:
 assumes *nonabort-i*: $chaos \sqsubseteq i$
 shows $c \sqsubseteq (c \mathbin{//} i) \parallel i$
$\langle proof \rangle$

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 $c \mathbin{//} i \sqsubseteq d \longleftrightarrow c \sqsubseteq d \parallel i$
$\langle proof \rangle$

Refining the "numerator" in a quotient, refines the quotient.

lemma *rely-mono*:
 assumes *c-refsto-d*: $c \sqsubseteq d$
 shows $(c \mathbin{//} i) \sqsubseteq (d \mathbin{//} i)$
$\langle proof \rangle$

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*: $i \sqsubseteq j$
 shows $(c \mathbin{/\!\!/} j) \sqsubseteq (c \mathbin{/\!\!/} i)$
⟨*proof*⟩

lemma *par-nonabort*:
 assumes *nonabort-i*: $chaos \sqsubseteq i$
 assumes *nonabort-j*: $chaos \sqsubseteq j$
 shows $chaos \sqsubseteq i \parallel j$
 ⟨*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*: $chaos \sqsubseteq j$
 shows $((c \mathbin{/\!\!/} j) \mathbin{/\!\!/} i) = c \mathbin{/\!\!/} (i \parallel j)$
⟨*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*: $chaos \sqsubseteq i$
 assumes *nonabort-j*: $chaos \sqsubseteq j$
 shows $c \Cap d \sqsubseteq (j \Cap (c \mathbin{/\!\!/} i)) \parallel (i \Cap (d \mathbin{/\!\!/} j))$
⟨*proof*⟩

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

lemma *distribute-rely-conjunction*:
 assumes *nonabort-i*: $chaos \sqsubseteq i$
 shows $(c \Cap d) \mathbin{/\!\!/} i \sqsubseteq (c \mathbin{/\!\!/} i) \Cap (d \mathbin{/\!\!/} i)$
⟨*proof*⟩

lemma *distribute-rely-choice*:
 assumes *nonabort-i*: *chaos* $\sqsubseteq$ *i*
 shows $(c \sqcap d) \mathbin{/\!/} i \sqsubseteq (c \mathbin{/\!/} i) \sqcap (d \mathbin{/\!/} i)$
$\langle proof \rangle$

lemma *distribute-rely-parallel1*:
 assumes *nonabort-i*: *chaos* $\sqsubseteq$ *i*
 assumes *nonabort-j*: *chaos* $\sqsubseteq$ *j*
 shows $(c \parallel d) \mathbin{/\!/} (i \parallel j) \sqsubseteq (c \mathbin{/\!/} i) \parallel (d \mathbin{/\!/} j)$
$\langle proof \rangle$

lemma *distribute-rely-parallel2*:
 assumes *nonabort-i*: *chaos* $\sqsubseteq i$
 assumes *i-par-i*: $i \parallel i \sqsubseteq i$
 shows $(c \parallel d) \mathbin{/\!/} i \sqsubseteq (c \mathbin{/\!/} i) \parallel (d \mathbin{/\!/} i)$
$\langle proof \rangle$

lemma *distribute-rely-sequential*:
 assumes *nonabort-i*: *chaos* $\sqsubseteq i$
 assumes $(\forall\ c.\ (\forall\ d.\ ((c \parallel i);(d \parallel i) \sqsubseteq (c;d) \parallel i)))$
 shows $(c;d) \mathbin{/\!/} i \sqsubseteq (c \mathbin{/\!/} i);(d \mathbin{/\!/} i)$
$\langle proof \rangle$

lemma *distribute-rely-sequential-event*:
 assumes *nonabort-i*: *chaos* $\sqsubseteq i$
 assumes *nonabort-j*: *chaos* $\sqsubseteq j$
 assumes *nonabort-e*: *chaos* $\sqsubseteq e$
 assumes $(\forall\ c.\ (\forall\ d.\ ((c \parallel i);e;(d \parallel j) \sqsubseteq (c;e;d) \parallel (i;e;j))))$
 shows $(c;e;d) \mathbin{/\!/} (i;e;j) \sqsubseteq (c \mathbin{/\!/} i);e;(d \mathbin{/\!/} j)$
$\langle proof \rangle$

lemma *introduce-parallel-with-rely*:
 assumes *nonabort-i*: *chaos* $\sqsubseteq i$
 assumes *nonabort-j0*: *chaos* $\sqsubseteq j_0$
 assumes *nonabort-j1*: *chaos* $\sqsubseteq j_1$
 shows $(c \Cap d) \mathbin{/\!/} i \sqsubseteq (j_1 \Cap (c \mathbin{/\!/} (j_0 \parallel i))) \parallel (j_0 \Cap (d \mathbin{/\!/} (j_1 \parallel i)))$
$\langle proof \rangle$

lemma *introduce-parallel-with-rely-guarantee*:
 assumes *nonabort-i*: *chaos* $\sqsubseteq$ *i*
 assumes *nonabort-j0*: *chaos* $\sqsubseteq j_0$
 assumes *nonabort-j1*: *chaos* $\sqsubseteq j_1$
 shows $(j_1 \parallel j_0) \Cap (c \Cap d) \mathbin{/\!/} i \sqsubseteq (j_1 \Cap (c \mathbin{/\!/} (j_0 \parallel i))) \parallel (j_0 \Cap (d \mathbin{/\!/} (j_1 \parallel i)))$

⟨*proof*⟩

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 ($\forall$ *c*. ($\forall$ *d*. (($c \parallel i$);($d \parallel i$) ⊑ (c;d) $\parallel i$)))
 shows (c^ω;d) // i ⊑ (c // i)$^\omega$;(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].

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 $(\sqcap C) \parallel d = (\sqcap c \in C.c \parallel d)$ 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 $c \pitchfork (\bigsqcup D) = (\bigsqcup d \in D.c \pitchfork d)$. 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 $c \pitchfork \bot = \bot$.

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

[1] C. Aarts, R. Backhouse, E. Boiten, H. Doombos, N. van Gasteren, R. van Geldrop, P. Hoogendijk, E. Voermans, and J. van der Woude. Fixed-point

calculus. *Information Processing Letters*, 53:131–136, 1995. Mathematics of Program Construction Group.

[2] R. J. Colvin, I. J. Hayes, and L. A. Meinicke. Designing a semantic model for a wide-spectrum language with concurrency. *Formal Aspects of Computing*, pages 1–22, 2016. Accepted 28 November 2016.

[3] I. J. Hayes. Generalised rely-guarantee concurrency: An algebraic foundation. *Formal Aspects of Computing*, 28(6):1057–1078, November 2016.

[4] I. J. Hayes, R. J. Colvin, L. A. Meinicke, K. Winter, and A. Velykis. An algebra of synchronous atomic steps. In J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, editors, *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 352–369, Cham, November 2016. Springer International Publishing.

[5] C. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, Oct. 1983.

[6] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Available as: Oxford University Computing Laboratory (now Computer Science) Technical Monograph PRG-25.