Exploiting Galois connections for ′Rely/Guarantee Thinking′

José N. Oliveira

Techn. Report TR-HASLab:01:2016

May 2016

HASLab - High-Assurance Software Laboratory
Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://haslab.di.uminho.pt

TR-HASLab:01:2016
Exploiting Galois connections for 'Rely/Guarantee Thinking'
by José N. Oliveira

Abstract

This research report presents a basis for RG reasoning in the spirit of the *Programming from Galois connections* [15] approach. The starting point is van Staden's work on *On Rely-Guarantee Reasoning* [20] together with Jones et al's *Laws and semantics for rely-guarantee refinement* [7] and *Balancing expressiveness in formal approaches to concurrency* [14].

Exploiting Galois connections for 'Rely/Guarantee Thinking'

José N. Oliveira

May 2016

Abstract

This research report presents a basis for RG reasoning in the spirit of the *Programming from Galois connections* [15] approach. The starting point is van Staden's work on *On Rely-Guarantee Reasoning* [20] together with Jones et al's *Laws and semantics for rely-guarantee refinement* [7] and *Balancing expressiveness in formal approaches to concurrency* [14].

1 Context

For *functional* and *concurrency-free* imperative programs there are stable program development methodologies. However, programs don't run in isolation and rather interfere with each other and, unfortunately, we haven't got the same level of stability in *concurrent programming*. There are shared-state concurrency abstractions, for instance *separation* logic and "*rely/guarantee* (R/G) thinking". Can we make such abstractions *algebraic* and *calculational*? How can we ensure *reliability* in presence of interference?

This research report tries to set up a basis for RG reasoning in the spirit of the *Programming from Galois connections* [15] approach. The starting point is van Staden's *On Rely-Guarantee Reasoning* [20] together with Jones et al's *Laws and semantics for rely-guarantee refinement* [7] and *Balancing expressiveness in formal approaches to concurrency* [14]. The approach is similar to [6] but perhaps simpler to understand because not the full expressiveness of R/G is encompassed, which calls for more studying.

In [15] a relational combinator $R \upharpoonright S$ is introduced to express a *superlative*: we want to specify *the best* (here is the superlative) approximation to specification R (a binary relation) according to some optimization criterion S (another binary relation). This combinator is expressive enough to encode adjoints of Galois connections, which are regarded as generic and mathematically rich specification devices. For instance, using one of the examples of [15],

$$(\div y) = R \upharpoonright \geqslant \qquad \textbf{where } z \; R \; x \;\Leftrightarrow\; z \times y \leqslant x$$

(for $y > 0$) specifies whole division: $x \div y$ is the *largest* (here is the superlative) integer z such that $z \times y \leqslant x$ holds. Of course, this is nothing but a (*"pointfree"*)

way of stating the equivalence

$$x \times y \leqslant z \quad \Leftrightarrow \quad x \leqslant z \div y \tag{1}$$

which is a Galois connection (GC). Leaving the ordering(s) implicit, the standard (very succinct) way to express the same is:

$$(\times y) \dashv (\div y)$$

In words, *multiplication* (lower adjoint of the connection) is the (*easy*) operation which helps to define the (*difficult*) operation of *division* (the upper adjoint of the connection).

Paper [15] exploits the algebraic properties of combinator $R \upharpoonright S$ applied to the derivation of functional programs which implement adjoints of GCs regarded as problem specifications. How much of this extends to concurrent programming?

2 R/G

Sequential programming and its fundamental concepts of choice, sequential composition and finite iteration has found an effective mathematical characterization in the concept of a Kleene algebra (KA) [4]. This includes Hoare logic [10] and what is commonly known as *contract-oriented* programming, a style based on *pre/ post* conditions. This logic enables us to reason about programs such as, for instance,

$$\begin{array}{l} \{\, r := n; i := 0; \\ \textbf{while}\ (i < r)\ \{ \\ \quad \textbf{if}\ p\ i\ \textbf{then}\ r := i \\ \quad \textbf{else}\ i := i + 1 \} \\ \} \end{array}$$

that finds the least index $i < n$ for which a predicate p holds.

The following alternative, concurrent program [8] achieves the same effect but allows *interference* over a shared state:

$$\begin{array}{l} l := n; \\ r := n; \\ \left\{ \begin{array}{l} i := 0; \\ \textbf{while}\ (i < min\ l\ r)\ \{ \\ \quad \textbf{if}\ p\ i\ \textbf{then}\ l := i \\ \quad \textbf{else}\ i := i + 2 \} \end{array} \middle\| \begin{array}{l} j := 1; \\ \textbf{while}\ (j < min\ l\ r)\ \{ \\ \quad \textbf{if}\ p\ j\ \textbf{then}\ r := j \\ \quad \textbf{else}\ j := j + 2 \} \end{array} \right\} \end{array}$$

What's new is the $P \parallel Q$ syntax enabling programs P and Q to run in parallel and interfere with each other. This calls for an orthogonal dimension in contracts based on so-called R/G (rely/guarantee) (R/G) conditions [12], so that interference among concurrent processes becomes compositional too.

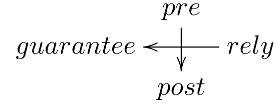
The table alongside summarizes such a two-dimensional framework for contract-oriented, concurrent programming. In presence of $P \parallel Q$, Hoare triples $\{\, p \,\}\ P\ \{\, q \,\}$ become *quintuples*

regime	**asset**	**onus**
sequential (;)	pre (p)	post (q)
concurrent (‖)	rely (r)	guarantee (g)

$$\{p\}\,\{r\}\;P\;\{g\}\,\{q\}$$

where r (resp. g) is the *rely* (resp. *guarantee*) condition, a proof style pioneered by Jones [11, 12, 13].

Contract-oriented, concurrent software therefore calls for two axes of compositionality, one sequential and the other concurrent, leading to two-dimensional contracts expressed by the picture alongside. This impacts on the algebraic structures needed to formalize such contracts, namely by requiring "bi"-extended algebras in which two multiplicative operators exist, ";" capturing sequential composition and "||" concurrent composition.



Defining the rely and guar constructs

An approach to specifying and reasoning about concurrent programs could be based on combinators similar to $\restriction$. Interestingly, this already seems to be the trend in the R/G approach of [7, 14, 20, 6]: two combinators rely and guar are defined which specify similar approximations ("superlatives") in a context where concurrent programs can interfere with each other over a shared state:

- rely $R\;P$ is the *largest* program which refines P under the interference of other programs bound by rely-condition R (a binary relation on program states).
- guar $G\;P$ is the *largest* refinement of program P whose steps satisfy guarantee-condition G (another such binary relation).

Reference [20] expresses (control of) external interference on a program Q by terms of the form $Q \parallel \mathit{traces}\;R$, where $\parallel$ is a "run in parallel" combinator and $\mathit{traces}\;R$ gives all possible program traces whose atomic steps are in R. Thus the GC

$$Q \parallel \mathit{traces}\;R \subseteq P \;\Leftrightarrow\; Q \subseteq \mathsf{rely}\;R\;P \tag{2}$$

holds[1], that is:

$$(\parallel \mathit{traces}\;R) \dashv (\mathsf{rely}\;R)$$

In words: rely $R\;P$ specifies the largest program Q which, in spite of the interference captured by $\mathit{traces}\;R$, still approximates (refines) P. Similarly, given a program P, the equivalence

$$\mathit{steps}\;Q \subseteq G \wedge Q \subseteq P \;\Leftrightarrow\; Q \subseteq \mathsf{guar}\;G\;P \tag{3}$$

is a GC on the refinement space of P: for each guarantee condition G, combinator guar $G\;P$ yields the largest refinement of P whose steps do not violate G.[2] As GCs compose, from (2,3) we get

$$\mathit{steps}\;Q \subseteq G \wedge Q \parallel \mathit{traces}\;R \subseteq P \;\Leftrightarrow\; Q \subseteq \mathsf{guar}\;G\;(\mathsf{rely}\;R\;P) \tag{4}$$

[1] This GC, also central to [6], corresponds to law 31 **(rely-refinement)** of [7], but note that the ordering in [7, 6] is the converse of the one used in [20] and in this note.

[2] This corresponds to the generic *dial*(ect) operator on languages (46).

This is stated (as an implication from left to right) in law 32 **(guar+rely)** of [7]. Based on (4), reference [7] considers

$$\mathsf{guar}\ G\ (\mathsf{rely}\ R\ [S, S']) \tag{5}$$

as the "most general form of specification" of interfering programs, where $[S, S']$ denotes standard (non-interfering) pre/post specification in Hoare style — $[S, S']$ is the largest program Q such that $S\ \{Q\}\ S'$ holds:[3]

$$Q \subseteq [S, S'] \ \Leftrightarrow\ S\ \{Q\}\ S' \tag{6}$$

By expanding assertion

$$P \subseteq \mathsf{guar}\ G\ (\mathsf{rely}\ R\ [S, S'])$$

via (4) and (6) we obtain $S\ \{P \parallel \mathit{traces}\ R\}\ S' \wedge \mathit{steps}\ P\ \subseteq\ G$, which [20] regards as one of two possible trace semantics for *RG-quintuple* $S\ R\ \{P\}\ G\ S'$.

In this report I stick to this meaning of RG-quintuples and try to exploit the algebraic potential of "universal" properties (2) and (3) in the derivation of R/G laws and inference rules. Some of these are immediately available from the definitions, for instance law 28 (**Intro-g**) of [14],

$$\mathsf{guar}\ G\ P \subseteq P$$

which is nothing but one of the cancellation corollaries of (3):

$$\mathit{steps}\ (\mathsf{guar}\ G\ P) \subseteq G \wedge \mathsf{guar}\ G\ P \subseteq P \tag{7}$$

(Replace $\mathsf{guar}\ G\ P$ for Q in (3) and simplify.)

RG-quintuples

I will denote RG-quintuples by arrows of the form $S \xrightarrow[R \to G]{P} S'$ — a notation which extends a similar use of arrows to express Hoare triples in [17] — and define, as in [20],

$$S \xrightarrow[R \to G]{P} S' \qquad \Leftrightarrow \qquad P \subseteq \mathsf{guar}\ G\ (\mathsf{rely}\ R\ [S, S']) \tag{8}$$

equivalent to:

$$S \xrightarrow[R \to G]{P} S' \qquad \Leftrightarrow \qquad S\ \{P \parallel \mathit{traces}\ R\}\ S'\ \wedge\ \mathit{steps}\ P\ \subseteq\ G \tag{9}$$

RG-quintuples such as e.g.

$$S \xrightarrow[\bot \to \top]{P} S'\ = S\ \{P\}\ S' \tag{10}$$

— no interference, nothing to guarantee — suggest that the proposed arrow notation probably makes sense. (Symbols $\bot$ / $\top$ denote the smallest / largest relations

[3]Cf. Morgan, C.: The specification statement. ACM Trans. Program. Lang. Syst. 10, 403-419 (1988).

on program states.) The "limit situation" expressed by (10) arises from the monotonicity of the operators involved in (9)

$$S_1 \xrightarrow[R_1 \to G_1]{P} S_1' \;\wedge \left\{ \begin{array}{rcl} S_2 & \subseteq & S_1 \\ S_1' & \subseteq & S_2' \end{array} \right. \wedge \left\{ \begin{array}{rcl} R_2 & \subseteq & R_1 \\ G_1 & \subseteq & G_2 \end{array} \right. \quad \Rightarrow \quad S_2 \xrightarrow[R_2 \to G_2]{P} S_2' \tag{11}$$

Galois connections (2,3) arise from similar connections involving trace operators such as e.g. $steps$, $\|$ etc. Appendix A reviews such a background (languages and alphabets), which is below instantiated to program traces and state steps, leading to a brief account of the R/G calculus expressed in such a way.

3 Programs, steps and traces

We follow [20] in regarding a program as a language of *traces*, a trace being a sequence of *steps*. Each step is a pair (s, s') of two states — s is the *before* state and s' is the *after* state. State pairs are needed because of program interference. We define:

$$Step = S \times S$$
$$Prog = \mathsf{P} Step^*$$
$$Steps = \mathsf{P} Step$$

Each $R \in Steps$ is therefore a relation between before and after states. In [20], GC $steps \dashv traces$ instantiates GC $alph \dashv lang$ detailed in appendix A, for $A = Step$:

$$steps\ P \ \subseteq\ R \quad \Leftrightarrow \quad P \subseteq traces\ R \tag{12}$$

Thus $traces\ R$ is the largest program P whose steps are in R:

$$traces\ R = \{x \mid x \in Step^*, \mathsf{elems}\ s \subseteq R\}$$

Thus

$$\langle \forall\ s, s' \ :\ (s, s') \in \cdot \in traces\ R :\ s'\ R\ s \rangle$$

GC $steps \dashv traces$ grants a number of properties for free, for instance

- $steps$ ans $traces$ are monotonic
- lower (resp. upper) adjoint distributes by suprema (resp. infima) and thus:

$$steps\ (P \cup Q) = steps\ P \cup steps\ Q \tag{13}$$
$$traces\ (R \cap S) = traces\ R \cap traces\ S \tag{14}$$

- the ranges of $steps$ and $traces$ are isomorphic posets.[4]
- composition $traces \cdot steps$ is a *closure* operator:

$$P \subseteq traces\ (steps\ P) \tag{15}$$

- composition $steps \cdot traces$ is an *interior* operator:

$$steps\ (traces\ R) \ \subseteq\ R \tag{16}$$

[4]Cf. the *unity of opposites* theorem of [2].

It turns out that this GC is *perfect* on the steps side, so (16) is actually an equality[5]

$$steps\ (traces\ R) = R \tag{17}$$

meaning that $steps$ is surjective and $traces$ is injective. So the range of $traces$ is isomorphic to the $Steps$ poset. Moreover, this ensures

$$traces\ R \subseteq traces\ R' \Leftrightarrow R \subseteq R' \tag{18}$$

— instance of (45) in the appendix.

The closure operator $free\ P = traces\ (steps\ P)$ (15) defined by the GC is the largest program with the same steps as P: $P \subseteq free\ P$ and $free\ (free\ P) = free\ P$. We identify three special programs:

- $skip$, which has no steps, $steps\ skip = \bot$. Then, by (12),

  $$skip \subseteq traces\ R \tag{19}$$

  for any R: $skip = \{[\,]\}$.
- $chaos = traces\ \top$, the largest of all programs.
- $abort = \{\,\}$, the zero of sequential and parallel composition.

Sequential composition. Instantiating the general definition (53), we define:

$$P; Q = \mathbf{do}\ \{r \leftarrow P; s \leftarrow Q; \mathsf{return}\ (r +\!\!+ s)\} \tag{20}$$

Instance of (56):

$$steps\ (P; Q) = steps\ P \cup steps\ Q \tag{21}$$

Thus $steps\ (P; skip) = steps\ P$, etc.

Concurrency. Instantiating (48), we define the shuffle product (interleaving) of two programs (sets of traces) as a lower adjoint:

$$X \parallel Q \subseteq P \quad \Leftrightarrow \quad X \subseteq P \mathbin{/\!\!/} Q \tag{22}$$

Operator $(_ \mathbin{/\!\!/} Q)$ — referred to in [6] as **rely quotient** — is therefore the largest program X which, running in parallel and interfering with Q, still refines P. From property

$$P \parallel skip = P = skip \parallel P \tag{23}$$

and the GC above we get $P \mathbin{/\!\!/} skip = P$. Analogy: $x \times 1 = x = x \div 1$. Facts

$$(P \parallel traces\ R) = traces\ R \Leftarrow 1 \subseteq P \subseteq traces\ R \tag{24}$$
$$traces\ (R \cup S) = traces\ R \parallel traces\ S \tag{25}$$
$$steps\ (P \parallel Q) = steps\ P \cup steps\ Q \tag{26}$$

hold — just instantiate (51), (52) and (49), respectively. Thus $steps\ (P; Q) = steps\ (P \cup Q) = steps\ (P \parallel Q)$.

[5] Cf. (44) in the appendix.

Operator $\cdot /\!\!/ \cdot$ enables a closed definition of rely (2):

$$\mathsf{rely}\ R\ P = P /\!\!/ \ traces\ R \tag{27}$$

cf:

$$
\begin{array}{ll}
 & Q \subseteq \mathsf{rely}\ R\ P \\
\Leftrightarrow & \quad \{\ \} \\
 & Q \parallel traces\ R \subseteq P \\
\Leftrightarrow & \quad \{\ \} \\
 & Q \subseteq P /\!\!/ \ traces\ R \\
\Box &
\end{array}
$$

From (3) we also get

$$\mathsf{guar}\ G\ P = P \cap traces\ G \tag{28}$$

Thus (8) rewrites to:

$$S \xrightarrow[R \to G]{P} S' \quad \Leftrightarrow \quad P \subseteq ([S, S'] /\!\!/ \ traces\ R) \cap traces\ G \tag{29}$$

A similar definition can be found in [6].

Relation with CKA Laws in [9] — commutativity ("fairness")

$$P \parallel Q = Q \parallel P \tag{30}$$

Frame laws

$$P; Q \subseteq P \parallel Q \tag{31}$$

$$P; (Q \parallel R) \subseteq (P; Q) \parallel R \tag{32}$$

$$(P \parallel Q); R \subseteq P \parallel (Q; R) \tag{33}$$

Exchange law

$$(P \parallel Q); (R \parallel S) \subseteq (P; R) \parallel (Q; S) \tag{34}$$

R/G consistency How large can rely $R\ P$ and guar $R\ P$ be? And how small are they *forced* to be? This will give us some measure of the (in)consistency between R and P.

From the cancellation of (3) — let $Q :=$ guar $G\ P$ and simplify — we get guar $G\ P \subseteq P$, as intended from the beginning. A similar argument shows rely $R\ P \subseteq P$, as also expected: from (2) we get another cancellation law, $(\mathsf{rely}\ R\ P) \parallel traces\ R \subseteq P$. Then:

$$
\begin{array}{ll}
 & (\mathsf{rely}\ R\ P) \parallel traces\ R \subseteq P \\
\Rightarrow & \quad \{\ (19)\ ;\ \text{monotonicity}\ \} \\
 & (\mathsf{rely}\ R\ P) \parallel skip \subseteq P \\
\Leftrightarrow & \quad \{\ skip\ (23)\ \} \\
 & \mathsf{rely}\ R\ P \subseteq P \\
\Box &
\end{array}
$$

Concerning lower bounds, fact (24) leads to the following corollary of (2):

$$(traces\ R \subseteq P \ \Leftrightarrow\ Q \subseteq \mathsf{rely}\ R\ P) \quad \Leftarrow \quad skip\ \subseteq\ Q\ \subseteq\ traces\ R \tag{35}$$

Thus $traces\ R \subseteq \mathsf{rely}\ R\ P$ iff $traces\ R \subseteq P$ iff $skip \subseteq \mathsf{rely}\ R\ P$.

In the opposite direction, let us now show under what conditions we have $\mathsf{rely}\ R\ P = \bot$, that is, R and P are incompatible. Since $\mathsf{rely}\ R\ P = \bot$ coincides with $\neg(skip \subseteq \mathsf{rely}\ R\ P)$, by

$$traces\ R \subseteq P \ \Leftrightarrow\ skip \subseteq \mathsf{rely}\ R\ P \tag{36}$$

— corollary of (2) — $\mathsf{rely}\ R\ P = \bot$ is equivalent to $\neg(traces\ R \subseteq P)$. This happens e.g. when R allows transitions that are not steps of P.

4 R/G calculus

Recall definitions (2) and (3) of R/G combinators rely and guar as Galois connections, and the two equivalent definitions (8) and (9) of a RG-quintuple. This section gives a first, brief account of the R/G calculus [7, 14] expressed in such a GC-based way. Note that this is work in progress which is by no means complete.

A basic property such as e.g. law 26 (**guarantee-true**)

$$\mathsf{guar}\ \top\ P = P \tag{37}$$

of [14] is immediate from (3) by indirect equality [19], since every relation is below $\top$. The no-interference law[6]

$$\mathsf{rely}\ \bot\ P = P \tag{38}$$

is also immediate from the corresponding GC by indirect equality:

$$\begin{array}{ll} & Q \subseteq \mathsf{rely}\ \bot\ P \\ \Leftrightarrow & \quad \{\ (2)\ ;\ traces\ \bot = skip \text{ and } (23)\ \} \\ & Q \subseteq P \\ :: & \quad \{\ \text{indirect equality}\ \} \\ & \mathsf{rely}\ \bot\ P = P \\ \Box & \end{array}$$

From (38) and (37) the proof of (10) is also immediate:

$$\begin{array}{ll} & S \xrightarrow[\bot \to \top]{P} S' \\ \Leftrightarrow & \quad \{\ \text{definition } (8)\ \} \\ & P \subseteq \mathsf{guar}\ \top\ (\mathsf{rely}\ \bot\ [S, S']) \end{array}$$

[6]This corresponds to Law 35 (**rely-id**) of [7], since program $traces\ \bot = skip = skip; skip; \ldots; skip; \ldots$ models stuttering steps. In this sense, $traces\ id = skip$ too, but this needs to be worked out in detail and seems to be related to the reflexivity of the rely condition.

$\Leftrightarrow$ { upper adjoint guar preserves suprema (37) }

$P \subseteq \mathsf{rely}\ \bot\ [S, S']$

$\Leftrightarrow$ { (38) }

$P \subseteq [S, S']$

$\Leftrightarrow$ { (6) }

$S\ \{P\}\ S'$

$\square$

The following proof of the **Nested-g** property

$$\mathsf{guar}\ G\ (\mathsf{guar}\ G'\ P) = \mathsf{guar}\ (G \cap G')\ P$$

of [14] illustrates the typical, iterated use of GCs [15] in conjunction with indirect equality:

$Q \subseteq \mathsf{guar}\ G\ (\mathsf{guar}\ G'\ P)$

$\Leftrightarrow$ { (3) twice }

$\mathit{steps}\ Q \subseteq G \wedge \mathit{steps}\ Q \subseteq G' \wedge Q \subseteq P$

$\Leftrightarrow$ { universal property of meet }

$\mathit{steps}\ Q \subseteq G \cap G' \wedge Q \subseteq P$

$\Leftrightarrow$ { (3) }

$Q \subseteq \mathsf{guar}\ (G \cap G')\ P$

:: { indirect equality }

$\mathsf{guar}\ G\ (\mathsf{guar}\ G'\ P) = \mathsf{guar}\ (G \cap G')\ P$

$\square$

By indirect equality too we can derive a closed formula for guar,

$$\mathsf{guar}\ G\ P = \mathit{traces}\ G \cap P$$

corresponding to (9) in [14]:

$Q \subseteq \mathsf{guar}\ G\ P$

$\Leftrightarrow$ { (3) }

$\mathit{steps}\ Q \subseteq G \wedge Q \subseteq P$

$\Leftrightarrow$ { GC (12) }

$Q \subseteq \mathit{traces}\ Q \wedge Q \subseteq P$

$\Leftrightarrow$ { universal property of meet }

$Q \subseteq \mathit{traces}\ Q \cap P$

:: { indirect equality }

$\mathsf{guar}\ G\ P = \mathit{traces}\ Q \cap P$

$\square$

Indirect equality is also the essence of the following calculation showing that, wherever $R \subseteq G$, terms $\mathsf{rely}\ R\ (\mathsf{guar}\ G\ P)$ and $\mathsf{guar}\ G\ (\mathsf{rely}\ R\ P)$ are interchangeable:

$$
\begin{array}{ll}
 & Q \subseteq \mathsf{rely}\ R\ (\mathsf{guar}\ G\ P) \\
\Leftrightarrow & \quad \{\ \text{GC (2)}\ \} \\
 & (Q \parallel \mathit{traces}\ R) \subseteq \mathsf{guar}\ G\ P \\
\Leftrightarrow & \quad \{\ \text{(3)}\ \} \\
 & (Q \parallel \mathit{traces}\ R) \subseteq P \wedge \mathit{steps}\ (Q \parallel \mathit{traces}\ R) \subseteq G \\
\Leftrightarrow & \quad \{\ \text{(26) ; (17)}\ \} \\
 & (Q \parallel \mathit{traces}\ R) \subseteq P \wedge \mathit{steps}\ Q \subseteq G \wedge R \subseteq G \\
\Leftrightarrow & \quad \{\ R \subseteq G \text{ assumed}\ \} \\
 & (Q \parallel \mathit{traces}\ R) \subseteq P \\
:: & \quad \{\ \text{indirect equality}\ \} \\
 & \mathsf{rely}\ R\ (\mathsf{guar}\ G\ P)\ =\ \mathsf{guar}\ G\ (\mathsf{rely}\ R\ P) \\
\Box &
\end{array}
$$

Thus:

$$
\mathsf{rely}\ R\ (\mathsf{guar}\ G\ P) = \mathsf{guar}\ G\ (\mathsf{rely}\ R\ P) \quad \Leftarrow \quad R \subseteq G \tag{39}
$$

Side condition $R \subseteq G$ is regarded in [7] as an inadvertent requirement for rely/guar permutation; but then law 33 **(swap-rely-guarantee)** has no side condition at all, so $R \subseteq G$ does not seem to be necessary. (Study this!) The calculation of law 39 **(nested-rely)**

$$
\mathsf{rely}\ R_1\ (\mathsf{rely}\ R_2\ P) = \mathsf{rely}\ (R_1 \cup R_2)\ P
$$

of [7] is similar:

$$
\begin{array}{ll}
 & Q \subseteq \mathsf{rely}\ R_1\ (\mathsf{rely}\ R_2\ P) \\
\Leftrightarrow & \quad \{\ \text{GC (2) twice}\ \} \\
 & Q \parallel \mathit{traces}\ R_1 \parallel \mathit{traces}\ R_2 \subseteq P \\
\Leftrightarrow & \quad \{\ \mathit{traces}\ R_1 \parallel \mathit{traces}\ R_2 = \mathit{traces}\ (R_1 \cup R_2), \text{cf. (25)}\ \} \\
 & Q \parallel \mathit{traces}\ (R_1 \cup R_2) \subseteq P \\
\Leftrightarrow & \quad \{\ \text{GC (2)}\ \} \\
 & Q \subseteq \mathsf{rely}\ (R_1 \cup R_2)\ P \\
:: & \quad \{\ \text{indirect equality}\ \} \\
 & \mathsf{rely}\ R_1\ (\mathsf{rely}\ R_2\ P) = \mathsf{rely}\ (R_1 \cup R_2)\ P \\
\Box &
\end{array}
$$

Keeping with the style of the proofs thus far, we use *indirect inclusion*[7] prove law 29 of [6] — **rely-distribute-conjunction**:

$$\mathsf{rely}\ (R_1 \cup R_2)\ P \subseteq \mathsf{rely}\ R_1\ P \cap \mathsf{rely}\ R_2\ P$$

Proof:

$$
\begin{array}{ll}
 & Q \subseteq \mathsf{rely}\ (R_1 \cup R_2)\ P \\
\Leftrightarrow & \quad \{\ \text{CG (2)}\ \} \\
 & Q \parallel (\mathit{traces}\ (R_1 \cup R_2)) \subseteq P \\
\Rightarrow & \quad \{\ \text{monotonicity of } \mathit{traces} \text{ and } \parallel\ \} \\
 & Q \parallel ((\mathit{traces}\ R_1) \cup (\mathit{traces}\ R_2)) \subseteq P \\
\Leftrightarrow & \quad \{\ \parallel \text{ is a lower adjoint}\ \} \\
 & (Q \parallel (\mathit{traces}\ R_1)) \cup (Q \parallel (\mathit{traces}\ R_2)) \subseteq P \\
\Leftrightarrow & \quad \{\ \text{universal property of } \cup\ ;\ \text{CG (2) twice}\ \} \\
 & Q \subseteq \mathsf{rely}\ R_1\ P\ \wedge\ Q \subseteq \mathsf{rely}\ R_2\ P \\
\Leftrightarrow & \quad \{\ \text{universal property of } \cap\ \} \\
 & Q \subseteq \mathsf{rely}\ R_1\ P \cap \mathsf{rely}\ R_2\ P \\
:: & \quad \{\ \text{indirect inclusion}\ \} \\
 & \mathsf{rely}\ (R_1 \cup R_2)\ P \subseteq \mathsf{rely}\ R_1\ P \cap \mathsf{rely}\ R_2\ P \\
\Box &
\end{array}
$$

Law 25 **(guarantee-monotonic)** [14] is proved in the same way, assuming $P \subseteq P'$:

$$
\begin{array}{ll}
 & Q \subseteq \mathsf{guar}\ G\ P \\
\Leftrightarrow & \quad \{\ (3)\ \} \\
 & \mathit{steps}\ Q \subseteq G \wedge Q \subseteq P \\
\Leftrightarrow & \quad \{\ P \subseteq P' \text{ assumed}\ \} \\
 & \mathit{steps}\ Q \subseteq G \wedge Q \subseteq P \wedge P \subseteq P' \\
\Rightarrow & \quad \{\ \text{transitivity}\ \} \\
 & \mathit{steps}\ Q \subseteq G \wedge Q \subseteq P' \\
\Leftrightarrow & \quad \{\ (3)\ \} \\
 & Q \subseteq \mathsf{guar}\ G\ P' \\
:: & \quad \{\ \text{indirect inclusion}\ \} \\
 & \mathsf{guar}\ G\ P \subseteq \mathsf{guar}\ G\ P' \\
\Box &
\end{array}
$$

[7] For any partial order, indirect equality arises from two indirect inclusions, by anti-symmetry.

The cancellation property (7) of guar's GC does all the work in the following proof of (part of) **Distribute-g-parallel** (law 30 in [14]):[8]

$$(\mathsf{guar}\ G\ P_1 \parallel \mathsf{guar}\ G\ P_2) \subseteq \mathsf{guar}\ G\ (P_1 \parallel P_2)$$

$\Leftrightarrow$ { (3) }

$$\left\{\begin{array}{l} \mathit{steps}\ (\mathsf{guar}\ G\ P_1 \parallel \mathsf{guar}\ G\ P_2) \subseteq G \\ \mathsf{guar}\ G\ P_1 \parallel \mathsf{guar}\ G\ P_2 \subseteq P_1 \parallel P_2 \end{array}\right.$$

$\Leftrightarrow$ { cancellation rule (7) ; (26) }

$$\mathsf{guar}\ G\ P_1 \parallel \mathsf{guar}\ G\ P_2 \subseteq P_1 \parallel P_2$$

$\Leftarrow$ { monotonicity of $\parallel$ }

$$\mathsf{guar}\ G\ P_1 \subseteq P_1 \wedge \mathsf{guar}\ G\ P_2 \subseteq P_2$$

$\Leftrightarrow$ { cancellation rule (7) }

TRUE

□

Concerning the logic of RG-quintuples, we prove law (Jconc) of [20] as example:

$$\left\{\begin{array}{l} S_1 \xrightarrow[R_1 \to G_1]{P_1} S_1' \\ S_2 \xrightarrow[R_2 \to G_2]{P_2} S_2' \end{array}\right. \wedge \left\{\begin{array}{l} G_1 \subseteq R_2 \\ G_2 \subseteq R_1 \end{array}\right. \quad \Rightarrow \quad S_1 \cap S_2 \xrightarrow[R_1 \cap R_2 \to G_1 \cup G_2]{P_1 \parallel P_2} S_1' \cap S_2'$$

Proof: By transitivity of $\subseteq$ we have the premises:

$$\mathit{steps}\ P_1 \subseteq R_2 \quad \wedge \quad \mathit{steps}\ P_2 \subseteq R_1 \tag{40}$$

Then:

$$S_1 \cap S_2 \xrightarrow[R_1 \cap R_2 \to G_1 \cup G_2]{P_1 \parallel P_2} S_1' \cap S_2'$$

$\Leftrightarrow$ { (9) }

$$\left\{\begin{array}{l} (S_1 \cap S_2\ \{P_1 \parallel P_2 \parallel \mathit{traces}\ (R_1 \cap R_2)\}\ S_1' \cap S_2') \\ \mathit{steps}\ (P_1 \parallel P_2)\ \subseteq\ G_1 \cup G_2 \end{array}\right.$$

$\Leftrightarrow$ { (26) ; Hoare triples }

$$\left\{\begin{array}{l} \left\{\begin{array}{l} (S_1 \cap S_2\ \{P_1 \parallel P_2 \parallel \mathit{traces}\ (R_1 \cap R_2)\}\ S_1') \\ (S_1 \cap S_2\ \{P_1 \parallel P_2 \parallel \mathit{traces}\ (R_1 \cap R_2)\}\ S_2') \end{array}\right. \\ (\mathit{steps}\ P_1 \subseteq G_1 \cup G_2 \wedge \mathit{steps}\ P_2 \subseteq G_1 \cup G_2) \end{array}\right.$$

$\Leftarrow$ { $\subseteq$-transitivity }

$$\left\{\begin{array}{l} \left\{\begin{array}{l} S_1\ \{P_1 \parallel P_2 \parallel \mathit{traces}\ (R_1 \cap R_2)\}\ S_1' \\ S_2\ \{P_1 \parallel P_2 \parallel \mathit{traces}\ (R_1 \cap R_2)\}\ S_2' \end{array}\right. \\ (\mathit{steps}\ P_1 \subseteq G_1 \wedge \mathit{steps}\ P_2 \subseteq G_2) \end{array}\right.$$

$\Leftarrow$ { (40) ; Hoare logic }

[8] Note that (3) does not help in proving the equality — study this.

$$\left\{\begin{array}{l}\left\{\begin{array}{l}S_1\ \{P_1 \parallel \mathit{traces}\ R_1 \parallel \mathit{traces}\ (R_1 \cap R_2)\}\ S_1' \\ S_2\ \{\mathit{traces}\ R_2 \parallel P_2 \parallel \mathit{traces}\ (R_1 \cap R_2)\}\ S_2'\end{array}\right. \\ (\mathit{steps}\ P_1 \subseteq G_1 \wedge \mathit{steps}\ P_2 \subseteq G_2)\end{array}\right.$$

$\Leftrightarrow$ { (24) }

$$\left\{\begin{array}{l}\left\{\begin{array}{l}S_1\ \{P_1 \parallel \mathit{traces}\ R_1\}\ S_1' \\ S_2\ \{\mathit{traces}\ R_2 \parallel P_2\}\ S_2'\end{array}\right. \\ (\mathit{steps}\ P_1 \subseteq G_1 \wedge \mathit{steps}\ P_2 \subseteq G_2)\end{array}\right.$$

$\Leftrightarrow$ { (9) twice }

$$\left\{\begin{array}{l}S_1 \xrightarrow[R_1 \to G_1]{P_1} S_1' \\ S_2 \xrightarrow[R_2 \to G_2]{P_2} S_2'\end{array}\right.$$

$\square$

5 Concurrent Kleene algebras

The overall setting of R/G presented in this report is that of a concurrent Kleene algebra (KA) [9]. A standard KA (or *quantale*) is an

> idempotent semiring $(S, +, ;, 0, 1)$ in which $a \leqslant b \Leftrightarrow a + b = b$ is a complete lattice and ; distributes over arbitrary suprema.

A concurrent KA (CKA) is a "double quantale"

> $(S, +, ;, \|, 0, 1)$ where the two multiplications (; , ‖) are linked by a so-called *exchange* axiom:
>
> $$(a \parallel b)\ ;\ (c \parallel d) \leqslant (b\ ;\ c) \parallel (a\ ;\ d) \qquad (41)$$

Example: take S the set of all (non-deterministic) programs under sequential and parallel composition. CKAs offer quite rich a structure, namely the Galois connections

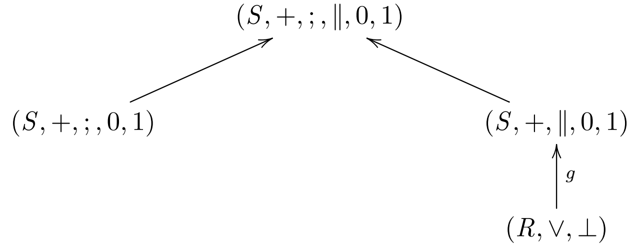
$$a\ ;\ b \leqslant c \ \Leftrightarrow\ a \leqslant c\ /\ b$$
$$a \parallel b \leqslant c \ \Leftrightarrow\ a \leqslant c\ /\!\!/\ b$$

which, together with the *exchange* law (41), yield a number of useful properties, eg.

$$a \parallel b = b \parallel a$$
$$a\ ;\ b \leqslant a \parallel b$$
$$(a \parallel b)\ ;\ c \leqslant a \parallel (b\ ;\ c)$$
$$a\ ;\ (b \parallel c) \leqslant (a\ ;\ b) \parallel c$$

etc.

CKAs + interference Add an "interference *monoid*" R to the overall scheme:



$g : R \to S$ is an injective monoid homomorphism telling how interference impacts on programs. As (——) must be idempotent and commutative, monoid $(R, \vee, \bot)$ becomes a *bounded semilatice*. So, by construction

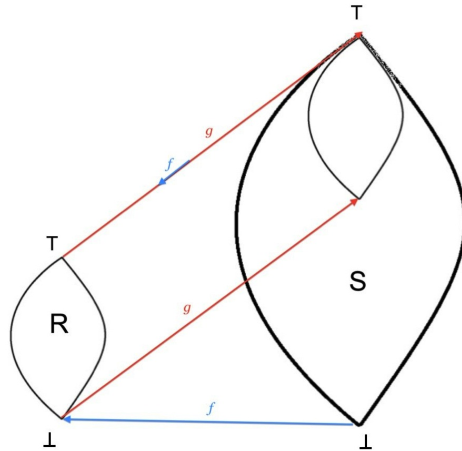
$$g\ (r \vee s) = g\ r \parallel g\ s$$
$$g\ \bot = 1$$

Simple way of defining g: think of it as upper adjoint of a GC

$$f\ p\ \subseteq\ r\ \Leftrightarrow\ p \leqslant g\ r \tag{42}$$

perfect on the interference-semilatice side: $f \cdot g = id$. In our R/G setting, $f = steps$ and $g = traces$.

Overall picture, adapting a well-known diagram by Roland [2] – in case R is also a lattice:



'Unity of opposites' — Theorem 6.42 in [2]:

$$f\ 1 = \bot$$
$$f\ (p \parallel q) = (f\ p) \vee (f\ q)$$

— *"inverse functions have inverse properties"* — and so on. Thus we have all we need to define the two *generic* combinators rely and guar,

$$g\ r \parallel q \leqslant p \quad \Leftrightarrow \quad q \leqslant \mathsf{rely}\ r\ p$$
$$f\ q \subseteq r \wedge q \leqslant p \quad \Leftrightarrow \quad q \leqslant \mathsf{guar}\ r\ p$$

granting the R/G algebra briefly presented before.

6 Summary

This report first arose from studying [20]. RG-quintuples in which $[S, S']$ denotes a pre/post pair where S' is binary are considered in [20] but not yet in this reportİ use monadic notation for what the author of [20] writes in standard set theory, implicitly relying on the powerset monad.[9]

The main reason for my monadic notation is the prospect of extending R/G reasoning to other semantic models of programming. In particular, by switching from the powerset to the (sub)distribution monad, one could eventually develop a stochastic model of interfering programs using (hopefully) the same R/G laws, possibly side-conditioned. Then we could reason about the risk of putting together faulty programs over shared memory (or else correct programs over faulty shared memory), in a similar way to the approach of [16] to calculating risk propagation in functional programs developed by source-to-source transformation.

Prior to this, reference [6] should be studied carefully studied, as the two approaches share the same algebraic flavour. The generalization of R/G *relations* to *processes* — mirrored in the use of GC (22) instead of (2) — is elegant and promises a number of simplifications, e.g. it reduces $(\mathsf{guar}\ \ G\ \ _)$ to $(G \cap _)$ — or is this simplifying too much?

Whether a suitable strategy for (parallel) program development by *shrinking* [15] — in the parallel rather than sequential context — could be defined, remains for the moment speculative.

A Alphabets and languages

Let $A = \mathsf{P}S$ be a datatype of *alphabets* of symbols in S and $L = \mathsf{P}S^*$ be a datatype of *languages* (sets of strings of S). One can always get the alphabet of a language,

$$
\begin{array}{l}
alph : L \rightarrow A \\
alph = (\gg\!= \mathsf{elems})
\end{array}
$$

where $\mathsf{elems} : X^* \rightarrow \mathsf{P}X$ is the operator which yields the elements of a finite list, and monadic function $\gg\!=\mathsf{elems}$ promotes elems from lists to sets of lists.

Clearly, $alph$ is monotonic (a larger language cannot have a strictly smaller alphabet) and distributes over suprema, $\langle \cup\, i\ :\ i = 1, n :\ \lambda_i \rangle \gg\!= \mathsf{elems} = \langle \cup\, i\ :\ i = 1, n :\ \lambda_i \gg\!= \mathsf{elems} \rangle$. Thus[10] $alph$ is the lower adjoint of a GC:

$$
alph\ \lambda \ \subseteq\ \alpha \ \Leftrightarrow\ \lambda \ \subseteq\ lang\ \alpha \tag{43}
$$

What is the meaning of $lang : A \rightarrow L$ such that $alph \dashv lang$ holds? $alph\ \lambda$ is the alphabet of λ, the smallest set of symbols needed to understand λ. What (43) means[11] is that $lang\ \alpha$ is the largest language λ one can build by taking symbols from alphabet α. From this we can infer a closed definition of the upper adjoint $lang$:

$$
lang\ \alpha \quad = \quad \langle \cup\, \lambda\ :\ alph\ \lambda\ \subseteq\ \alpha :\ \lambda \rangle
$$

[9] Standard set theory notation is heavily monadic in this sense, even if it doesn't look like at first sight.

[10] By the *Fundamental theorem of Galois connections* [2].

[11] Read the $\Rightarrow$ part of the $\Leftrightarrow$.

$$= \langle \cup \lambda \; : \; \langle \cup s \; : \; s \in \lambda : \; \mathsf{elems}\; s \rangle \subseteq \; \alpha : \; \lambda \rangle$$
$$= \langle \cup \lambda \; : \; \langle \forall s \; : \; s \in \lambda : \; \mathsf{elems}\; s \; \subseteq \; \alpha \rangle \; : \; \lambda \rangle$$
$$= \{ s \mid \mathsf{elems}\; s \subseteq \alpha \}$$

The smallest language is $\lambda = empty$, with empty alphabet. From (43) necessarily $alph\ empty = empty$. Calculating $lang\ (alph\ empty)$ we don't get $empty$ back but rather $\{[\,]\}$, the language with one sole sentence, the empty one. This illustrates the cancellation property

$$\lambda \; \subseteq \; lang\ (alph\ \lambda)$$

which we get from (43) by replacing α by $lang\ \lambda$ and simplifying. The other cancellation corollary is actually an equality,

$$alph\ (lang\ \alpha) = \alpha \tag{44}$$

meaning that the GC is perfect on the alphabets' side: $lang$ is injective and $alph$ is surjective. To prove (44) we only need to prove:

$$\alpha \; \subseteq \; alph\ (lang\ \alpha)$$
$\Leftrightarrow$ { definitions of $lang$ and $alph$ }
$$\alpha \; \subseteq \; \{ t \mid \mathsf{elems}\; t \; \subseteq \; \alpha \} \ggg \mathsf{elems}$$
$\Leftrightarrow$ { monads: $x \ggg f = \mu\, \{ f\ a \mid a \in x \}$ where μ is big union }
$$\alpha \; \subseteq \; \mu\, \{ \mathsf{elems}\; t \mid \mathsf{elems}\; t \; \subseteq \; \alpha \}$$
$\Leftrightarrow$ { change of variable $X := \mathsf{elems}\; t$ }
$$\alpha \; \subseteq \; \mu\, \{ X \mid X \; \subseteq \; \alpha \}$$
$\Leftrightarrow$ { α is the largest of its subsets }

TRUE

□

From (44) we immediately get

$$lang\ \alpha \subseteq lang\ \beta \Leftrightarrow \alpha \subseteq \beta \tag{45}$$

cf:

$$lang\ \alpha \subseteq lang\ \beta$$
$\Leftrightarrow$ { (43) }
$$alph\ (lang\ \alpha) \; \subseteq \; \beta$$
$\Leftrightarrow$ { (44) }
$$\alpha \subseteq \beta$$

□

Dialects Let $\rho \subseteq \lambda$ be a sub-language λ restricted to (sub) alphabet β, that is, $alph\ \rho \subseteq \beta$. The *dialect* of λ characterized by only using symbols in β is the largest such ρ, denoted by $dial\ \beta\ \lambda$:

$$dial\ \rho\ \lambda = \langle \bigcup\ \rho\ :\ \rho \subseteq \lambda \wedge alph\ \rho \subseteq \beta :\ \rho \rangle$$

Clearly, this is equivalent to GC:

$$\rho \subseteq \lambda \wedge alph\ \rho \subseteq \beta\ \Leftrightarrow\ \rho \subseteq dial\ \beta\ \lambda \tag{46}$$

NB: the GC becomes more evident as we write:

$$\langle alph, id \rangle\ \rho \subseteq^2 (\beta, \lambda)\ \Leftrightarrow\ \rho \subseteq dial\ (\beta, \lambda)$$

Mixing languages. Quite often we mix languages, e.g. by saying sentences such as

"o stack da esquerda está off"

which mixes English (stack, off) with Portuguese (o, da, esquerda, está). Given two languages $x, y \subseteq L$ we define their mixing, or *shuffle product* [12] by:

$$\begin{array}{l} \| : L \to L \to L \\ x \parallel y = \mathbf{do}\ \{\, t \leftarrow x; t' \leftarrow y; t \otimes t' \,\} \end{array}$$

where

$$\begin{array}{l} (\otimes) : S^* \times S^* \to \mathsf{P} S^* \\ t \otimes [\,] = \mathsf{return}\ t \\ [\,] \otimes t = \mathsf{return}\ t \\ (a : t) \otimes (b : p) = u \cup s\ \mathbf{where} \\ \quad u = \mathbf{do}\ \{\, x \leftarrow t \otimes (b : p); \mathsf{return}\ (a : x) \,\} \\ \quad s = \mathbf{do}\ \{\, y \leftarrow (a : t) \otimes p; \mathsf{return}\ (b : y) \,\} \end{array}$$

cf. [3]. Example mixing a language of letters with a language of numbers:

```
{ "ab", "xy" } || { "12", "45" } =
{ "12ab", "1a2b", "1ab2", "a12b", "a1b2", "ab12",
"45ab", "4a5b", "4ab5", "a45b", "a4b5", "ab45",
"12xy", "1x2y", "1xy2", "x12y", "x1y2", "xy12",
"45xy", "4x5y", "4xy5", "x45y", "x4y5", "xy45" }
```

As in [5], we use the following names for the two smallest languages: $1 = \{\, [\,] \,\}$ and $0 = empty$. From the definition we get $x \parallel 1 = \mathbf{do}\ \{\, t \leftarrow x; t \otimes [\,] \,\} = \mathbf{do}\ \{\, t \leftarrow x; \{\, t \,\} \,\} = x$, similarly for $1 \parallel x = x$:

$$x \parallel 1 = x = 1 \parallel x \tag{47}$$

If $x \parallel y$ can be thought of as "mixing two languages" together, we may also think of the adjoint operator of "separating" (or "dividing") languages:[13]

$$x \parallel y \subseteq z \Leftrightarrow x \subseteq z \mathbin{/\!\!/} y \tag{48}$$

[12] Also called Hurwitz product in [18].
[13] Mind the formal similarity with (1).

In this GC, the division $z \mathbin{/\!\!/} y$ is the largest language which, mixed with y, does not contain sentences foreign to z. From (48) we infer $(0 \parallel y) = 0$ and $z \mathbin{/\!\!/} 1 = z$:

$$
\begin{array}{ll}
& 0 \parallel y \ \subseteq\ z \Leftrightarrow 0 \subseteq z \mathbin{/\!\!/} y \\
\Leftrightarrow & \qquad \{ \text{ anything is larger than } 0 \ \} \\
& 0 \parallel y \ \subseteq\ z \\
\Leftrightarrow & \qquad \{ \text{ only } 0 \text{ is smaller than any } z \ \} \\
& 0 \parallel y = 0 \\
\Box &
\end{array}
$$

$$
\begin{array}{ll}
& x \parallel 1 \ \subseteq\ z \Leftrightarrow x \subseteq z \mathbin{/\!\!/} 1 \\
\Leftrightarrow & \qquad \{ \ x \parallel 1 = x \ (47) \ \} \\
& x \ \subseteq\ z \Leftrightarrow x \subseteq z \mathbin{/\!\!/} 1 \\
:: & \qquad \{ \text{ indirect equality over } \subseteq \} \\
& z = z \mathbin{/\!\!/} 1 \\
\Box &
\end{array}
$$

Moreover, we have

$$alph\ (x \parallel y) = alph\ x \cup alph\ y \qquad (49)$$

$$lang\ \alpha \parallel lang\ \alpha = lang\ \alpha \qquad (50)$$

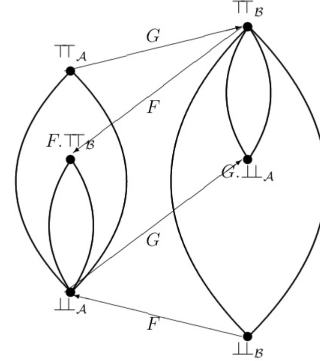
the latter as a corollary of:

$$(x \parallel lang\ \alpha) = lang\ \alpha \quad \Leftarrow \quad 1 \ \subseteq\ x \ \subseteq\ lang\ \alpha \qquad (51)$$

Proof of (51): from the side condition, $x \ \subseteq\ lang\ \alpha \ \Leftrightarrow\ alph\ x \ \subseteq\ \alpha$ including case $x = lang\ \alpha$. By (49) we get $(x \parallel lang\ \alpha) \ \subseteq\ lang\ \alpha$. Moreover:

$$
\begin{array}{ll}
& lang\ \alpha \ \subseteq\ (x \parallel lang\ \alpha) \\
\Leftrightarrow & \qquad \{ \ (47) \ \} \\
& (1 \parallel lang\ \alpha) \ \subseteq\ (x \parallel lang\ \alpha) \\
\Leftarrow & \qquad \{ \text{ monotonicity of } \parallel \} \\
& 1 \ \subseteq\ x \\
\Leftrightarrow & \qquad \{ \text{ side condition assumed } \} \\
& \text{TRUE} \\
\Box &
\end{array}
$$

Unity of opposites. It is well known that the ranges of the adjoints of a GC are isomorphic, as so well depicted in the picture, taken from [1]. In our case, GC (43) is perfect on the alphabets' side — $alph\ (lang\ \alpha) = \alpha$ (44) — so $A \cong L'$ where $L' = \{lang\ \alpha \mid \alpha \in A\}$. Let us solve equation $lang\ \omega = lang\ \alpha \parallel lang\ \beta$ in L' for ω:



$$\mathsf{F}\,x \leqslant y \ \Leftrightarrow\ x \leqslant G\ y$$

$$
\begin{array}{ll}
& lang\ \omega = lang\ \alpha \parallel lang\ \beta \\
\Leftrightarrow & \qquad \{ \ alph \text{ is injective in } L' \ \} \\
& \omega = alph\ (lang\ \alpha \parallel lang\ \beta) \\
\Leftrightarrow & \qquad \{ \ (49) \ \} \\
& \omega = alph\ (lang\ \alpha) \cup alph\ (lang\ \beta) \\
\Leftrightarrow & \qquad \{ \ (44) \ \} \\
& \omega = \alpha \cup \beta
\end{array}
$$

and therefore

$$lang\ (\alpha \cup \beta) = lang\ \alpha \parallel lang\ \beta \tag{52}$$

holds, generalizing (50).

Sequencing. In $x \parallel y$ we can identify the subset of sentences where all symbols from $alph\ x$ come first, followed by those from $alph\ y$:

$$x; y = \textbf{do}\ \{ s \leftarrow x; s' \leftarrow y; \mathsf{return}\ (s +\!\!+ s') \} \tag{53}$$

(**NB:** $s +\!\!+ s'$ is the monad(plus) join in the list monad.) Alternatively, define $(w;) = fmap\ ((w +\!\!+))$ and then

$$\lambda; \rho = \textbf{do}\ \{ w \leftarrow \lambda; (w; \rho) \} \tag{54}$$

Clearly,

$$x; y \ \subseteq\ x \parallel y \tag{55}$$

since $s +\!\!+ s' \in s \otimes s'$. Moreover:

$$alph\ (x; y) \quad = \quad alph\ x \cup alph\ y \tag{56}$$

This stems from $\mathsf{elems}\ (s +\!\!+ s') = \mathsf{elems}\ s \cup \mathsf{elems}\ s'$. Sequencing distributes by language suprema and is therefore a lower-adjoint. To get the upper adjoint note the GC [1]

$$w; \lambda \subseteq \rho \ \Leftrightarrow\ \lambda \subseteq \partial_w \rho \tag{57}$$

where $\partial_w x = \textbf{do}\ \{ w +\!\!+ t \leftarrow x; \mathsf{return}\ t \}$. Then [1]:

$$
\begin{array}{cl}
 & x; y \ \subseteq\ z \\
\Leftrightarrow & \qquad \{ \text{ definition } \} \\
 & \textbf{do}\ \{ w \leftarrow x; (w; y) \} \subseteq z \\
= & \qquad \{ \text{ set monad } \} \\
 & \langle \forall\ w\ :\ w \in x :\ w; y \subseteq z \rangle \\
= & \qquad \{ \text{ (57) } \} \\
 & \langle \forall\ w\ :\ w \in x :\ y \subseteq \partial_w z \rangle \\
= & \qquad \{ \text{ infima } \} \\
 & y \subseteq \langle (\cap)\ w\ :\ w \in x :\ \partial_w z \rangle \\
= & \qquad \{ \ \textbf{do}\text{-notation } \} \\
 & y \subseteq \underbrace{\bigcap\ \textbf{do}\ \{ w \leftarrow x; \partial_w z \}}_{x \backslash z}
\end{array}
$$

□

Because sequencing is not commutative:

$$x; y \subseteq z \Leftrightarrow y \subseteq x \setminus z$$
$$x; y \subseteq z \Leftrightarrow x \subseteq z \,/\, y$$

Language $x \setminus z$ (resp. $z \,/\, x$) is are said to be a *right* (resp. *left*) factor of z. Language factorization is studied at length in [1].

Theorem 1 (Th.5.35 page 117, dualized) *GC $f \dashv g$ holds iff*

- *f monotonic*
- $f\ (g\ y) \subseteq y$
- $f\ x \subseteq y \Rightarrow x \sqsubseteq g\ y$

Theorem 2 (Fundamental theorem of Galois connections) *Let f preserve suprema,*

$$f\langle \cup\, i : i = 1, n : x_i \rangle = \langle \cup\, i : i = 1, n : f\ x_i \rangle \quad (58)$$

and be monotonic. Define:

$$g\ y = \langle \cup\, x : f\ x \subseteq y : x \rangle \quad (59)$$

Then $f \dashv g$ holds.
Proof*: we prove that the two last clauses of theorem 1 hold (f monotonic already):*

$$f\ (g\ y) \subseteq y$$
$$\Leftrightarrow \quad \{\ (59)\ \}$$
$$f\langle \cup\, x : f\ x \subseteq y : x \rangle \subseteq y$$
$$\Leftrightarrow \quad \{\ (58)\ \}$$
$$\langle \cup\, x : f\ x \subseteq y : f\ x \rangle \subseteq y$$
$$\Leftrightarrow \quad \{\ \textit{univ-}\cdot \cup \cdot\ \}$$
$$\langle \forall\, x : f\ x \subseteq y : f\ x \subseteq y \rangle$$
$$\Leftrightarrow \quad \{\ \textit{trivial}\ \}$$
$$\text{TRUE}$$

□

Now the third one:

$$\langle \forall\, x : f\ x \subseteq y : x \sqsubseteq g\ y \rangle$$
$$\Leftrightarrow \quad \{\ \textit{univ-}\cdot \cup \cdot\ \}$$
$$\langle \cup\, x : f\ x \subseteq y : x \rangle \sqsubseteq g\ y$$
$$\Leftrightarrow \quad \{\ (59)\ \}$$
$$g\ y \sqsubseteq g\ y$$
$$\Leftrightarrow \quad \{\ \textit{trivial}\ \}$$
$$\text{TRUE}$$

□

B Aczel traces

The evolution from the simple program traces above to so-called Aczel traces first of all calls for *heterogeneus* traces, that is, traces whose steps are marked as either having been carried out by the program or by the environment.

We use disjoint union for this, furthermore relying on some arbitrary monad(plus) F supporting program traces. Shuffling becomes more elaborate, generically:

$$
\begin{array}{l}
_ \parallel _ :: \mathsf{F}\ A^* \to \mathsf{F}\ A^* \to \mathsf{F}\ (A + A)^* \\
x \parallel y = \mathbf{do}\ \{ t \leftarrow x; t' \leftarrow y; t \otimes t' \}
\end{array}
$$

$$
\begin{array}{l}
t \otimes [\,] = \mathsf{return}\ (\mathsf{map}\ i_1\ t) \\
[\,] \otimes t = \mathsf{return}\ (\mathsf{map}\ i_2\ t)
\end{array}
$$

$$
\begin{array}{l}
(a : t) \otimes (b : p) = u \overset{+}{\cup} s\ \mathbf{where} \\
\quad u = \mathbf{do}\ \{ x \leftarrow t \otimes (b : p); \mathsf{return}\ (i_1\ a : x) \} \\
\quad s = \mathbf{do}\ \{ y \leftarrow (a : t) \otimes p; \mathsf{return}\ (i_2\ b : y) \}
\end{array}
$$

Note the property

$$
\pi_1 (x \parallel y) = x \wedge \pi_2 (x \parallel y) = y \tag{60}
$$

where

$$
\begin{array}{l}
\pi_1 \cdot :: \mathsf{F}\ (A + B)^* \to \mathsf{F}\ A^* \\
\pi_1 \cdot = \mathit{fmap}\ s\ \mathbf{where}\ s\ x = [\, a \mid i_1\ a \leftarrow x \,] \\
\pi_2 \cdot :: \mathsf{F}\ (A + B)^* \to \mathsf{F}\ B^* \\
\pi_2 \cdot = \mathit{fmap}\ s\ \mathbf{where}\ s\ x = [\, b \mid i_2\ b \leftarrow x \,]
\end{array}
$$

Here is an example for F the distribution monad:

$$
\begin{array}{l}
\mathbf{let}\ y = D\ [\texttt{"12"}\ |{-}{>}\ 0.1, \texttt{"23"}\ |{-}{>}\ 0.9] \\
\mathbf{let}\ x = D\ [\texttt{"ab"}\ |{-}{>}\ 0.7, \texttt{"xy"}\ |{-}{>}\ 0.3]
\end{array}
$$

Then:

$$
\begin{array}{l}
x \parallel y = \\
[i_1\ \texttt{'a'}, i_1\ \texttt{'b'}, i_2\ \texttt{'2'}, i_2\ \texttt{'3'}]\ 15.8\,\% \\
[i_2\ \texttt{'2'}, i_2\ \texttt{'3'}, i_1\ \texttt{'a'}, i_1\ \texttt{'b'}]\ 15.8\,\% \\
[i_1\ \texttt{'a'}, i_2\ \texttt{'2'}, i_1\ \texttt{'b'}, i_2\ \texttt{'3'}]\ 7.9\,\% \\
[i_1\ \texttt{'a'}, i_2\ \texttt{'2'}, i_2\ \texttt{'3'}, i_1\ \texttt{'b'}]\ 7.9\,\% \\
[i_2\ \texttt{'2'}, i_1\ \texttt{'a'}, i_1\ \texttt{'b'}, i_2\ \texttt{'3'}]\ 7.9\,\% \\
[i_2\ \texttt{'2'}, i_1\ \texttt{'a'}, i_2\ \texttt{'3'}, i_1\ \texttt{'b'}]\ 7.9\,\% \\
[i_1\ \texttt{'x'}, i_1\ \texttt{'y'}, i_2\ \texttt{'2'}, i_2\ \texttt{'3'}]\ 6.8\,\% \\
[i_2\ \texttt{'2'}, i_2\ \texttt{'3'}, i_1\ \texttt{'x'}, i_1\ \texttt{'y'}]\ 6.8\,\% \\
[i_1\ \texttt{'x'}, i_2\ \texttt{'2'}, i_1\ \texttt{'y'}, i_2\ \texttt{'3'}]\ 3.4\,\% \\
[i_1\ \texttt{'x'}, i_2\ \texttt{'2'}, i_2\ \texttt{'3'}, i_1\ \texttt{'y'}]\ 3.4\,\% \\
[i_2\ \texttt{'2'}, i_1\ \texttt{'x'}, i_1\ \texttt{'y'}, i_2\ \texttt{'3'}]\ 3.4\,\% \\
[i_2\ \texttt{'2'}, i_1\ \texttt{'x'}, i_2\ \texttt{'3'}, i_1\ \texttt{'y'}]\ 3.4\,\% \\
[i_1\ \texttt{'a'}, i_1\ \texttt{'b'}, i_2\ \texttt{'1'}, i_2\ \texttt{'2'}]\ 1.8\,\% \\
[i_2\ \texttt{'1'}, i_2\ \texttt{'2'}, i_1\ \texttt{'a'}, i_1\ \texttt{'b'}]\ 1.8\,\% \\
[i_1\ \texttt{'a'}, i_2\ \texttt{'1'}, i_1\ \texttt{'b'}, i_2\ \texttt{'2'}]\ 0.9\,\% \\
[i_1\ \texttt{'a'}, i_2\ \texttt{'1'}, i_2\ \texttt{'2'}, i_1\ \texttt{'b'}]\ 0.9\,\%
\end{array}
$$

$[i_2\ \texttt{'1'}, i_1\ \texttt{'a'}, i_1\ \texttt{'b'}, i_2\ \texttt{'2'}]\ 0.9\,\%$
$[i_2\ \texttt{'1'}, i_1\ \texttt{'a'}, i_2\ \texttt{'2'}, i_1\ \texttt{'b'}]\ 0.9\,\%$
$[i_1\ \texttt{'x'}, i_1\ \texttt{'y'}, i_2\ \texttt{'1'}, i_2\ \texttt{'2'}]\ 0.8\,\%$
$[i_2\ \texttt{'1'}, i_2\ \texttt{'2'}, i_1\ \texttt{'x'}, i_1\ \texttt{'y'}]\ 0.8\,\%$
$[i_1\ \texttt{'x'}, i_2\ \texttt{'1'}, i_1\ \texttt{'y'}, i_2\ \texttt{'2'}]\ 0.4\,\%$
$[i_1\ \texttt{'x'}, i_2\ \texttt{'1'}, i_2\ \texttt{'2'}, i_1\ \texttt{'y'}]\ 0.4\,\%$
$[i_2\ \texttt{'1'}, i_1\ \texttt{'x'}, i_1\ \texttt{'y'}, i_2\ \texttt{'2'}]\ 0.4\,\%$
$[i_2\ \texttt{'1'}, i_1\ \texttt{'x'}, i_2\ \texttt{'2'}, i_1\ \texttt{'y'}]\ 0.4\,\%$

Explicit termination leads to traces of type $\mathsf{F}\ (A+1)^*$. Our convention is that the first occurence of the unit marks the end of the trace, irrespective of any possible subsequent steps. Thus we can think of a function $rep : A^* \rightarrow (A+1)^+$ which adds a termination check mark at the end of a simple trace and its adjoint $trim$ converting a trace with termination mark(s) into its longest termination-mark free prefix,

$$rep\ x \preceq y \ \Leftrightarrow\ x \leqslant trim\ y \tag{61}$$

where $z \leqslant y$ means z is a prefix of y and $z \preceq y$ means $z \leqslant y$ and $\langle \forall\ i\ ::\ z\ i \sqsubseteq y\ i \rangle$ where $\sqsubseteq$ is the ordering on $A+1$. Thus $rep\ [\,] = [\checkmark]$. Implementation:

$trim\ [\,] = [\,]$
$trim\ (\checkmark : _) = [\,]$
$trim\ (\mathsf{Just}\ a : t) = a : trim\ t$

Perfection: $trim\ (rep\ x)) = x$.

References

[1] R. Backhouse. Factor theory and the unity of opposites. *JLAMP*, 85(5, Part 2):824–846, 2016.

[2] R.C. Backhouse. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.

[3] S.L. Bloom, N. Sabadini, and R.F.C. Walters. Matrices, machines and behaviors. *Applied Categorical Structures*, 4(4):343–360, 1996.

[4] J.H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[5] I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace theory. *ENTCS*, 164(1):47–65, 2006. Proceedings of CMCS 2006.

[6] I.J. Hayes. Generalised rely-guarantee concurrency: an algebraic foundation. *FAOC*, 28(6):1057–1078, 2016.

[7] I.J. Hayes, C.B. Jones, and R.J. Colvin. Refining rely-guarantee thinking. Technical Report CS-TR-1334, Newcastle University, 2012.

[8] I.J. Hayes, C.B. Jones, and R.J. Colvin. Laws and semantics for rely-guarantee refinement. Technical Report CS-TR-1425, Newcastle University, 2014.

[9] C.A. Hoare, S. van Staden, B. Möller, G. Struth, J. Villard, H. Zhu, and P. OHearn. Developments in concurrent Kleene algebra. In *RAMiCS*, volume 8428 of *LNCS*, pages 1–18. 2014.

[10] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12,10:576–580, 583, October 1969.

[11] C.B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[12] C.B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[13] C.B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[14] C.B. Jones, I.J. Hayes, and R.J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, 27(3):475–497, 2015.

[15] S.-C. Mu and J.N. Oliveira. Programming from Galois connections. *JLAP*, 81(6):680–704, 2012.

[16] D. Murta and J.N. Oliveira. A study of risk-aware program transformation. *SCP*, 110:51–77, 2015.

[17] J.N. Oliveira. *Extended Static Checking by Calculation using the Pointfree Transform*. volume 5520 of *LNCS*, pages 195–251. Springer-Verlag, 2009.

[18] J. Sakarovitch. Handbook of weighted automata. Springer Publishing Company, Incorporated, 2009.

[19] P.F. Silva and J.N. Oliveira. 'Galculator': functional prototype of a Galois-connection based proof assistant. In *PPDP '08: 10th int. ACM SIGPLAN conf. on Principles and practice of declarative programming*, pages 44–55. ACM, 2008.

[20] S. Staden. On rely-guarantee reasoning. In *MPC 2015*, pages 30–49, 2015.