Programming from Galois Connections

Shin-Cheng Mu[1] and José Nuno Oliveira[2]

[1] IIS, Academia Sinica, Taiwan, Taiwan
[2] High Assurance Software Lab, University of Minho, Portugal

Abstract. Problem statements often resort to superlatives such as in eg. "...the smallest such number", "...the best approximation", "...the longest such list" which lead to specifications made of two parts: one defining a broad class of solutions (the *easy* part) and the other requesting the optimal such solution (the *hard* part).
This paper introduces a binary relational combinator which mirrors this linguistic structure and exploits its potential for calculating programs by optimization. This applies in particular to specifications written in the form of Galois connections, in which one of the adjoints delivers the optimal solution being sought.
The framework encompasses re-factoring of results previously developed by Bird and de Moor for greedy and dynamic programming, in a way which makes them less technically involved and therefore easier to understand and play with.

1 Introduction

Computer programming is admittedly a challenging intellectual activity, calling for experience and training under a *read-understand-repeat* learning cycle. By acquiring good practices, relying on experienced teachers, the learning curve eventually bends, but reliability cannot be fully ensured. If one asks a student in programming about *why* she/he programs in that way (whatever this is) the answer is likely to be: *I don't know — my teachers used to do it this way.*

Why is this so? Isn't programming a *scientific discipline*? Surely it is, as several landmark textbooks show [3]. But, perhaps the question

Why *and* in what measure *is programming difficult?*

is yet to be given a satisfactory answer. By satisfactory we mean one which should unravel the ingredients of problem solving in a structured way, thus identifying which skills one should acquire to become a good programmer.

Abstraction is one such skill [12]. Abstracting from the programming language and underlying technology is generally accepted as mandatory in the early stages of thinking about a software problem. This has lead to *abstract modeling*, which has become a discipline in itself [10, 9]. However, handling abstractions is not

[3] See eg. the following (by no means exhaustive) list of widely acclaimed references: [11, 5, 21, 20, 4, 3].

easy either (many will say it is harder) and the question persists: *why* and *in what measure* is abstract modeling difficult?

Induction is another such skill, to which programmers unconsciously appeal whenever solving a complex problem by (temporarily) imagining some (smaller) parts of it already solved (the *divide-and-conquer* strategy). However, where and how does *induction* crop up in the design of a program? For instance, where exactly in the design of *quicksort* from its specification: *yield an ordered permutation of the input sequence*, does the *doubly recursive* strategy of the algorithm show up? The starting specification does not look inductive at all.

This paper tries to answer the questions above by splitting algorithmic specifications generically in two parts, to be addressed in different stages. Let us see where these come from.

In program construction one often encounters specifications asking for the "best" solution among a collection of solution candidates. Such specifications may have the form "the smallest such number . . . ", "the best approximation such that . . . ", "the longest prefix of a list satisfying . . . ", etc. A typical example is the definition of whole number division $x \div y$, for natural numbers x and (positive) y. A specification in words would say that $x \div y$ is the *largest* natural number that, when multiplied by y, is at most x. The standard function $\mathit{takeWhile}\ p$, as another example, returns the *longest* prefix of the input list such that all elements satisfy predicate p.

Many other, less classroom-like problem statements share the same linguistic pattern in their use of superlatives. For instance, the computation of the "best" schedule for a collection of tasks, given their time spans and an acyclic *Gantt* graph (describing which tasks depend upon completion of which other tasks) is another problem of the same kind. Such a schedule is "best" (among other schedules paying respect to the given graph of dependencies) in the sense that its tasks start as early as possible.

It is often relatively easy to construct a program that meets half of such specifications: returning or enumerating the feasible solution candidates, such as a natural number, or prefixes of the input list. This is the easy part. The hard part of the specification, however, demands that we return a candidate that is "best" in some sense (eg. some ordering): the largest integer, or the longest prefix, that satisfies the first, easy part of the specification.

In this paper we propose a new relational operator mirroring this "easy/hard" dichotomy of problem statements into mathematics. The operator is of the form

$$E \upharpoonright H,$$

where E specifies the easy part — the collection of solution candidates — while H specifies the hard part — criteria under which a best solution is chosen.

One might wonder how to come up with the easy/hard split in the first place. In this paper we aim at characterizing problem specifications in terms of *Galois connections* [6], in which one of the adjoints specifies the easy part (usually a known function) and the other specifies the one at target (the hard one). For instance, the (easy) adjoint of whole division is multiplication. This setting, which suggests that *"mathematics comes in easy/hard pairs"*, provides a natural way to split a problem in its components, as seen below.

Paper structure. In Section 2 we argue why Galois connections are suitable as calculational specifications, before motivating and introducing the $(\upharpoonright)$ operator in Section 3. If some components in the Galois connection are inductively defined, as reviewed in Section 4, the two theorems presented in Section 5, demonstrated by two examples, allow us to calculate the wanted adjoint. A larger example, scheduling a collection of tasks given a Gantt graph, is presented in Section 6, before we conclude in Section 7. A minimal review of relational program calculation is given in Appendix A and B.

2 Galois connections as program specifications

Let us take the problem of writing the algorithm of *whole division* as starting example [4]. Its specification has already been stated above, informally:

$x \div y$ is the largest natural number that, if multiplied by y, is at most x.

Which mathematics should we write to capture the text above? One possibility is to write a "literal" one,

$$x \div y = \langle \bigvee z \ :: \ z \times y \leq x \rangle, \tag{1}$$

encoding superlative *largest* explicitly as a supremum. Handling suprema, however, is not easy in general. A second version will circumvent this difficulty,

$$z = x \div y \ \equiv \ \langle \exists r \ : \ 0 \leq r < y : \ x = z \times y + r \rangle, \qquad \begin{array}{r|l} x & y \\ \hline r & z \end{array} \tag{2}$$

at the cost of existentially quantifying over remainders.

A third alternative is surprisingly simpler [18]: an equivalence

$$z \times y \leq x \ \equiv \ z \leq x \div y \qquad (y > 0) \tag{3}$$

universally quantified in all its variables. Pragmatically, it expresses a "shunting" rule which enables one to exchange between a whole division in the upper side of a $(\leq)$ inequality and a multiplication in the lower side, very much like in handling equations in school algebra.

Equivalences such as (3) are known as Galois connections [1, 6, 18]. In general, a Galois connection (GC) is a pair of functions f and g satisfying the equivalence $f\ z \leq x \ \equiv \ z \sqsubseteq g\ x$, for all z and x, given preorders $(\leq)$ and $(\sqsubseteq)$ (which can be the same). Functions f and g are said to be *adjoints* of each other — f is the *lower* adjoint and g the *upper* adjoint. In the case of (3) the adjoints are

$$z \underbrace{(\times y)}_{f} \leq x \ \equiv \ z \leq x \underbrace{(\div y)}_{g}.$$

[4] This example is taken from [18].

Why can one be so confident of the adequacy of (3) in the face of the given requirements? Do substitution $z := x \div y$ in (3) and obtain $(x \div y) \times y \leq x$: this tells that $x \div y$ is a candidate solution. Now read (3) from left to right, that is, focus on the implication $z \times y \leq x \;\Rightarrow\; z \leq x \div y$: conclude that $x \div y$ is largest among all other candidate solutions z.

So (3) means the same as (1). What are the advantages of the former over the latter? It turns up that (3) is far more generous with respect to inference of properties of $x \div y$, some of which are mere instantiations:

$$\begin{array}{rlr} & 0 \leq x \div y, & (z := 0) \\ y \leq x \;\equiv & 1 \leq x \div y. & (z := 1) \end{array}$$

Other facts, for instance $x \div 1 = x$, call for properties of the lower adjoint:

$$\begin{array}{ll} & z \leq x \div 1 \\ \equiv & \quad \{ \text{ Galois connection (3), for } y := 1 \ \} \\ & z \times 1 \leq x \\ \equiv & \quad \{ \text{ 1 is the unit of } \times \ \} \\ & z \leq x. \end{array}$$

That is, every natural number z which is at most $x \div 1$ is also at most x. We conclude that $x \div 1$ and x are the same. The rationale behind this style of reasoning is known as the principle of *indirect equality* [5]:

$$a = b \;\equiv\; \langle \forall x \;::\; x \leq a \equiv x \leq b \rangle. \tag{4}$$

More elaborate properties can be inferred from (3) using indirect equality and basic properties of the "easy" adjoint, for instance $(n \div m) \div d \;=\; n \div (d \times m)$, for $m, d > 0$. Again GC (3) blends well with indirect equality in an easy proof:

$$\begin{array}{ll} & z \leq (n \div m) \div d \\ \equiv & \quad \{ \text{ Galois connection (3), twice } \} \\ & (z \times d) \times m \leq n \\ \equiv & \quad \{ \ \times \text{ is associative } \} \\ & z \times (d \times m) \leq n \\ \equiv & \quad \{ \text{ Galois connection (3) again, in the opposite direction } \} \\ & z \leq n \div (d \times m) \\ :: & \quad \{ \text{ indirect equality (4) } \} \\ & (n \div m) \div d = n \div (d \times m). \end{array}$$

[5] See [1]. Readers unaware of this way of indirectly establishing algebraic equalities will recognize that the same pattern of indirection is used when establishing set equality via the membership relation, cf. $A = B \;\equiv\; \langle \forall x \;::\; x \in A \equiv x \in B \rangle$ as opposed to, e.g. circular inclusion: $A = B \;\equiv\; A \subseteq B \wedge B \subseteq A$.

Readers are challenged to compare this with alternative proofs of the same result using (1) or (2) instead of (3), not to mention the inductive proof required if relying on the obvious recursive implementation of $x \div y$ [18].

This strategy is applicable to arbitrarily complex problem domains, provided candidate solutions are ranked by a partial order such as $\leq$ above. This is shown in our next example, in which the underlying partial order is the *prefix* relation $\sqsubseteq$ on finite sequences and what is being specified is *take*, the function which yields the longest prefix of its input sequence up to some given length n [6]:

$$\mathit{length}\ z \leq n\ \wedge\ z \sqsubseteq x\ \equiv\ z \sqsubseteq \mathit{take}(n, x). \tag{5}$$

The property being sought,

$$\mathit{take}(n, \mathit{take}(m, x))\ =\ \mathit{take}(\mathit{min}(n, m), x), \tag{6}$$

will rely on another GC — that of defining the minimum of two numbers,

$$x \leq n\ \wedge\ x \leq m\ \equiv\ x \leq \mathit{min}(n, m), \tag{7}$$

in a way which shows how effectively GCs compose with each other [7]:

$$\begin{array}{ll}
& z \sqsubseteq \mathit{take}(n, \mathit{take}(m, x)) \\
\equiv & \quad \{\ \text{Galois connection (5), twice}\ \} \\
& \mathit{length}\ z \leq n \wedge \mathit{length}\ z \leq m \wedge z \sqsubseteq x \\
\equiv & \quad \{\ \text{Galois connection of } \mathit{min} \text{ of two numbers (7)}\ \} \\
& \mathit{length}\ z \leq \mathit{min}(n, m) \wedge z \sqsubseteq x \\
\equiv & \quad \{\ \text{(5) again, now folding}\ \} \\
& z \sqsubseteq \mathit{take}(\mathit{min}(n, m), x) \\
:: & \quad \{\ \text{indirect equality over prefix partial ordering } \sqsubseteq\ \} \\
& \mathit{take}(n, \mathit{take}(m, x)) = \mathit{take}(\mathit{min}(n, m), x).
\end{array}$$

Once again, the inductive proof of the same property performed over the recursive definition of *take* can but be regarded as an over-kill in face of such a simple calculation relying on the Galois connection concept.

One may wonder about the extent to which such a calculational style carries over to supporting the actual *synthesis* of the implementation of *take* given its specification (5) in the form of a Galois connection. This brings us to the core subject of the current paper: *how calculational is programming from Galois*

[6] See [16]. The authors would like to thank Roland Backhouse for spotting this Galois connection, whose upper adjoint $g = \mathit{take}$ is *specified* in terms of a lower adjoint involving *id* and *length*: $f\ z = (\mathit{length}\ z, z)$. Thus the lower ordering is the product partial order $(\leq) \times (\sqsubseteq)$, defined pointwise in the obvious way.

[7] For a detailed account of the algebra of Galois connections see eg. [1, 18, 16].

connections? Reference [18] shows how the defining Galois connection of $(\div)$ provides most of what is required for calculating its implementation. Reference [16] does the same for *take*, but Galois connection (5) is productive only after an inductive definition of prefix $(\sqsubseteq)$ is given explicitly, at point level. This somehow suggests that similar, but more economic and generic reasoning could be performed at the pointfree level of the algebra of programming [4], capitalizing on the pointfree definition of partial orderings such as prefix as relational folds.

Presenting such a generic, pointfree style of *programming from Galois connections* is the main aim of the current paper and leads us into the core of the research being reported.

3 Calculating Galois adjoints

Recall the definition of a GC: given two preorders $(\leq)$ on A and $(\sqsubseteq)$ on B, we say that two functions $f : A \leftarrow B$ and $g : B \leftarrow A$ form a GC if they satisfy the following equivalence:

$$f\ x \leq y \ \equiv\ x \sqsubseteq g\ y \qquad \text{cf. diagram:} \qquad \begin{tikzcd} A \arrow[loop left, "\leq"] \arrow[r, bend left, "g"] & B \arrow[l, bend left, "f"] \arrow[loop right, "\sqsubseteq"] \end{tikzcd} \tag{8}$$

It is quite common in GCs to have adjoints of disparate complexity. In GC (3) relating multiplication $(\times y)$ and whole division $(\div y)$, for example, the former is easier to define than the latter. A common scenario is that of one being given the two preorders and an easy adjoint, thereupon targeting at calculating the other adjoint.

Recall the easy/hard split discussed in Section 1. We will propose in this section a relational operator that manifests the split: by $E \upharpoonright H$ we denote a problem specification where the easy part E is "shrunk" by the requirements of the hard part H. It will then be shown that given $(\leq)$, $(\sqsubseteq)$, and lower adjoint f in a Galois connection, the upper adjoint can be expressed by:

$$g = (f^{\circ} \cdot (\leq)) \upharpoonright (\sqsupseteq). \tag{9}$$

We will then discuss, in this section and the next, some properties of $(\upharpoonright)$ that help us to calculate g. The operator $(\upharpoonright)$ is similar to, and shares many properties of, the *min* operator of Bird and de Moor [4], with the significant advantage of not requiring a power allegory.

3.1 The "shrink" operator

From now on we will be using a number of definitions and rules of the pointfree calculus of relations. For the reader's convenience, minimal review is given in Appendix A. For a thorough introduction, the reader is referred to Aarts et al. [1], and to Bird and de Moor [4] for a categorical perspective.

The first step toward manifesting the easy/hard split is to rewrite (8) to pointfree style by turning both sides into relations between x and y. Since partial orders such as $(\leq)$ and $(\sqsubseteq)$ are relations that map "larger" elements to "smaller" ones, the right hand side trivially translates to $(\sqsubseteq) \cdot g$. The left hand side, noting that $(x, f\,x) \in f^\circ$ and that $f\,x \leq y$ is another way of writing $(f\,x, y) \in (\leq)$, translates to $f^\circ \cdot (\leq)$. The equivalence means that the two relations are equal:

$$f^\circ \cdot (\leq) \;=\; (\sqsubseteq) \cdot g. \tag{10}$$

Such equality splits into two inclusions to be dealt with separately:

$$(\sqsubseteq) \cdot g \;\subseteq\; f^\circ \cdot (\leq) \quad\wedge\quad f^\circ \cdot (\leq) \;\subseteq\; (\sqsubseteq) \cdot g. \tag{11}$$

We show that the first inclusion in (11) is equivalent to $g \subseteq f^\circ \cdot (\leq)$ provided that f is monotonic, that is, $x \sqsubseteq y \;\Rightarrow\; f\;x \leq f\;y$, which can be written pointfree as $(\sqsubseteq) \cdot f^\circ \subseteq f^\circ \cdot (\leq)$. That it implies $g \subseteq f^\circ \cdot (\leq)$ is easy to see — since $(\sqsubseteq)$ is a preorder, $g \subseteq id \cdot g \subseteq (\sqsubseteq) \cdot g$. For the other direction, we reason:

$$\begin{array}{ll}
& g \;\subseteq\; f^\circ \cdot (\leq) \\
\Rightarrow & \{\; \text{monotonicity of } (\cdot) \;\} \\
& (\sqsubseteq) \cdot g \;\subseteq\; (\sqsubseteq) \cdot f^\circ \cdot (\leq) \\
\Rightarrow & \{\; \text{assumption: } f \text{ monotonic} \;\} \\
& (\sqsubseteq) \cdot g \;\subseteq\; f^\circ \cdot (\leq) \cdot (\leq) \\
\Rightarrow & \{\; \leq \text{ transitive: } (\leq) \cdot (\leq) \subseteq (\leq) \;\} \\
& (\sqsubseteq) \cdot g \;\subseteq\; f^\circ \cdot (\leq).
\end{array}$$

Concerning the second condition in (11), by taking converses of both sides and using the function shunting (23) rule, we transforming it to $g \cdot (f^\circ \cdot (\leq))^\circ \;\subseteq\; (\sqsupseteq)$. All in all, we have just factored Galois connection (11) into two parts,

$$f^\circ \cdot (\leq) \;=\; (\sqsubseteq) \cdot g \quad\equiv\quad \underbrace{g \;\subseteq\; f^\circ \cdot (\leq)}_{\text{"easy"}} \;\wedge\; \underbrace{g \cdot (f^\circ \cdot (\leq))^\circ \;\subseteq\; (\sqsupseteq)}_{\text{"hard"}}. \tag{12}$$

uncovering the easy/hard blend which is implicit in the original formulation. To see this, let us first abbreviate $f^\circ \cdot (\leq)$ to R. The left hand operand of the conjunction, $g \subseteq R$, states that g must return a result permitted by R — the "easy" part. The right hand operand $g \cdot R^\circ \subseteq (\sqsupseteq)$, on the other hand, states that if R maps x to y (therefore $(x, y) \in R^\circ$), it must be the case that $g\;x \sqsupseteq y$. That is, g returns a maximum result, under $(\sqsupseteq)$, among those results allowed by R. This is the "hard" part of the connection.

This is in fact nothing surprising: we have merely reconstructed an equivalent definition of a Galois connection [1, Theorem 5.29, page 66]: (1) f is monotonic, (2) $(f \cdot g)\;x \leq x$, (3) $(f\;x) \leq y \Rightarrow x \sqsubseteq (g\;y)$. The calculation above, however, inspires us to capture this pattern by a new relational operator. Given $R :: A \leftarrow B$ and $S :: A \leftarrow A$, define $R \upharpoonright S :: A \leftarrow B$, pronounced "$R$ shrunk by S", by

$$X \subseteq R \upharpoonright S \;\;\equiv\;\; X \subseteq R \;\wedge\; X \cdot R^\circ \subseteq S, \tag{13}$$

The definition states that X must be at most R, and that if X yields an output for an input x, it must be a maximum, with respect to S, among all possible outputs of x. In terms of the easy/hard split, R is the easy part and S defines the (optimization) criterion to be taken into account in the hard part. Using the properties of relational intersection and division, one may come up with a closed form for $R \upharpoonright S$:

$$R \upharpoonright S \;=\; R \cap S/R^{\circ}. \tag{14}$$

With the new notation we can go back to (12) and rephrase the right hand side of the equivalence in terms of $(\upharpoonright)$:

$$g \subseteq (f^{\circ} \cdot (\leq)) \upharpoonright (\sqsupseteq). \tag{15}$$

3.2 Properties of shrinking

From the definition (13), it is clear that $R \upharpoonright S \subseteq R$. It is easy to find out under what condition the other direction of inclusion holds: $R \subseteq R \upharpoonright S$ iff $R \cdot R^{\circ} \subseteq S$, and so

$$R = R \upharpoonright S \;\equiv\; \mathit{img}\ R \subseteq S. \tag{16}$$

since $\mathit{img}\ R = R \cdot R^{\circ}$. Since $\top$ is above anything, we have $R \upharpoonright \top = R$, that is, R stays the same if we put no constraints in the "hard" part. When $S = \bot$, no maximum exists, and thus $R \upharpoonright \bot$ yields nothing for any input: $R \upharpoonright \bot = \bot$.

The following rule shows how $(\upharpoonright Q)$ distributes into relational union:

$$(R \cup S) \upharpoonright Q \;=\; ((R \upharpoonright Q) \cap Q/S^{\circ}) \cup ((S \upharpoonright Q) \cap Q/R^{\circ}). \tag{17}$$

This arises from (14) and distribution of intersection over union. A most important consequence of (17) is that $(\upharpoonright Q)$ distributes into joins,

$$[R, T] \upharpoonright S \;=\; [R \upharpoonright S, T \upharpoonright S], \tag{18}$$

— recalling that $[R, S] = (R \cdot \mathit{inl}^{\circ}) \cup (S \cdot \mathit{inr}^{\circ})$ — and therefore conditionals,

$$(p \rightarrow R, T) \upharpoonright S \;= (p \rightarrow (R \upharpoonright S), (T \upharpoonright S)). \tag{19}$$

The following two rules allow us to distribute a function in and out of $(\upharpoonright)$:

$$(R \cdot f) \upharpoonright S \;=\; (R \upharpoonright S) \cdot f, \qquad (f \cdot R) \upharpoonright S \;=\; f \cdot (R \upharpoonright (f^{\circ} \cdot S \cdot f)).$$

The first equality can be proved using shunting and indirect equality, while the second generalizes a similar result in [4].

A number of results of the $(\upharpoonright)$ combinator relate to simplicity. Recall that the image of a simple relation R is coreflexive, that is, $\mathit{img}\ R \subseteq \mathit{id}$. Then, from (16) we draw $R = R \upharpoonright S$ if R is simple and S is reflexive, since $\mathit{img}\ R \subseteq \mathit{id}$ and $\mathit{id} \subseteq S$ entail $\mathit{img}\ R \subseteq S$.

Very often, S in (13) is anti-symmetric: $S \cap S^{\circ} \subseteq \mathit{id}$. In this case it can be shown that $R \upharpoonright S$ is always simple [7]. An application of this result concerns (15), ensuring $(f^\circ \cdot (\leq)) \upharpoonright (\sqsupseteq)$ simple for $(\sqsupseteq)$ a partial order. Thus equality (9) holds in such a situation.

The special case $S = id$ in (13) deserves some attention. In this situation, each output in the shrunk relation can relate only to itself. Thus $(y, x) \in R \upharpoonright id$ only when y is the sole value that x is mapped to by R. When more than one such y exists, x cannot be in the domain of $R \upharpoonright id$. Therefore, $R \upharpoonright id$ is the largest deterministic fragment of R. Formally,

$$X \subseteq R \upharpoonright id \quad \equiv \quad X \sqsubseteq R \ \wedge \ X \cdot X^\circ \subseteq id. \tag{20}$$

where $X \sqsubseteq R$ means $R \cdot dom\ X = X$, that is, X is less defined than R but as non-deterministic as R where defined [8].

4 Inductive relations

A question was raised in Section 1: where and how does induction crop up in the design of a program? An answer is provided in the remainder of this paper, in two steps. First, we recall that the "natural" way of ordering inductively defined data (such as eg. lists and trees) is through inductive relations defined using well-known combinators of the algebra of programming known as *folds* and *unfolds* [4]. Second, we show how specifications written as GCs on such inductive orderings "naturally" lead to inductive implementations, by calculation.

Functional programmers are familiar with inductive definitions of datatypes such as the natural number $\mathbb{N}$ and finite lists $List\ A$, and the fold function defined on them. The notion can be generalised to relations. For a review, the reader is referred to Appendix B. While functional folds are often used to define operations on inductively defined datatypes, it is often overlooked that many relations between inductively defined data can also be inductively defined as relational folds. The $(\geq)$ ordering on $\mathbb{N}$, for example, is nothing but the *least* relation satisfying

$$x \geq 0 \qquad \wedge \qquad x \geq y \Rightarrow (x + 1) \geq (y + 1).$$

The two conditions respectively translate to $\top \cdot zero^\circ \subseteq (\geq)$ and $(\geq) \subseteq suc^\circ \cdot (\geq) \cdot suc$ in pointfree style. It turns out that $(\geq)$ is a fold:

$$\begin{array}{ll}
& \top \cdot zero^\circ \subseteq (\geq) \ \wedge \ (\geq) \subseteq suc^\circ \cdot (\geq) \cdot suc \\
\equiv & \{ \ \text{shunting, since } R \subseteq T \wedge S \subseteq T \ \equiv \ R \cup S \subseteq T \ \} \\
& (\top \cdot zero^\circ) \cup (suc \cdot (\geq) \cdot suc^\circ) \subseteq (\geq) \\
\equiv & \{ \ \text{by (25): } [R, S] \cdot [T, U]^\circ = (R \cdot T^\circ) \cup (S \cdot U^\circ) \ \}
\end{array}$$

[8] This is the $\vdash_{pre}$ ordering of [17], where it is shown to be a factor of the standard refinement ordering. The proof of (20), omitted for space economy, essentially shows that the right hand sides of (13) and (20) coincide, for $S = id$.

$$
\begin{array}{ll}
 & [\top, suc \cdot (\geq)] \cdot [zero, suc]^{\circ} \subseteq (\geq) \\
\equiv & \{ \text{ absorption (24) } \} \\
 & [\top, suc] \cdot (id + (\geq)) \cdot [zero, suc]^{\circ} \subseteq (\geq) \\
\equiv & \{ \text{ (26) } \} \\
 & (\geq) \ = \ (\![\top, suc]\!).
\end{array}
$$

Note that $(+)$ in the penultimate line denotes the sum functor (see Appendix A) rather than numerical sum.

This not the only way the ordering on natural numbers can be defined, however. If we instead perform case analysis on the lesser side of the ordering, we come up with:

$$0 \leq y \qquad \wedge \qquad x \leq y \Rightarrow (x+1) \leq (y+1).$$

The first line translates to $zero \cdot \top \subseteq (\leq)$, where $\top$, having type $A \leftarrow \mathbb{N}$, is equivalent to $[zero, suc]^{\circ}$. By a similar calculation, we come up with a definition of $(\leq)$ as a fold (see Appendix A for the definition of $(\![\_, \_]\!)$):

$$(\leq) \ = \ (\![zero, zero \cup suc]\!).$$

Given two finite lists xs and ys, let $xs \sqsubseteq ys$ mean that xs is a prefix of ys. Natural numbers and finite lists are similar in structure and, through a similar calculation, one comes up with the following definition of $(\sqsubseteq)$ as a fold:

$$(\sqsubseteq) \ = \ (\![nil, nil \cup cons]\!). \tag{21}$$

Since lists are special cases of binary trees, one can generalise $(\leq)$ and $(\geq)$ to regular datatypes such that trees "grow larger" by substituting of empty nodes to other (sub)trees, and prove generically that $(\leq)^{\circ} = (\geq)$ (see [14]). The two orderings above are enough for our purposes of showing their role in calculating implementations of adjoints of Galois connections, as is shown in the sequel.

5 Program calculation by optimization — "shrinking specs into programs"

Given a Galois connection $f\ x \leq y \equiv x \sqsubseteq g\ y$, recall the conclusion of Section 3.1 that g can be expressed as $g = (f^{\circ} \cdot (\leq)) \upharpoonright (\sqsupseteq)$. The next step is triggered by a question: what can we do wherever $(\leq)$ and/or $(\sqsubseteq)$ are inductive relations?

In this section we will see two examples that follow a standard scheme we propose: (1) fusion, in the easy part, of the inner ordering $(\leq)$ with f°, to form either a fold or a restricted form of a hylomorphism (a fold followed by the converse of a fold); (2) shrinking the easy part using the hard part $(\upharpoonright(\sqsupseteq))$, hence the *motto*: "shrinking specs into programs".

We present two theorems to perform the shrinking: the *Greedy Theorem*, which applies when the easy part is a fold, and the *Dynamic Programming (DP) Theorem*, when it is a hylomorphism where the folding phase is a function. The Greedy Theorem is a simplification of that of Bird and de Moor [4]: it does not need a power allegory, and thus is applicable in more categories and, we believe, easier to comprehend. The DP-Theorem is similar to that of Bird and de Moor, with a different precondition, arising from its more general setting.

Both theorems are datatype-generic, and applicable not only for Galois connections, but also for optimisation problems in general. Due to space constraints we are unable to cover this aspect and will defer the discussion to a later work.

5.1 Example of greedy programming

Given a predicate p, $\mathit{takeWhile}\ p\ xs$ yields the longest prefix of xs whose elements all satisfy p:

$$\mathit{all}\ p\ xs\ \wedge\ xs \sqsubseteq ys \quad \equiv \quad xs \sqsubseteq \mathit{takeWhile}\ p\ ys. \tag{22}$$

This expresses a Galois connection between the set of all finite sequences ys and that of the ones (xs) whose elements all satisfy p. The upper adjoint is $\mathit{takeWhile}\ p$ and the lower adjoint is the embedding of all such sequences into the larger set. To see this we rewrite (22) into the pointfree equality

$$\mathit{map}\ p? \cdot (\sqsubseteq) \quad = \quad (\sqsubseteq) \cdot \mathit{takeWhile}\ p$$

by expressing $\mathit{all}\ p$ by coreflexive relation $\mathit{map}\ p?$. Recall that $(a, a) \in p? \ \equiv\ p\ a$. Therefore, $(xs, xs) \in \mathit{map}\ p? \ \equiv\ \mathit{all}\ p\ xs$.

Note how $\mathit{map}\ p?$ captures the lower-adjoint of the connection, as it is simple and entire over the set of all sequences satisfying p. Since $(\mathit{map}\ p?)^\circ$ is the same as $\mathit{map}\ p?$ (coreflexives are symmetric) we have that $\mathit{takeWhile}\ p$ can be defined in terms of $(\upharpoonright)$:

$$\mathit{takeWhile}\ p \ = \ (\mathit{map}\ p? \cdot (\sqsubseteq)) \upharpoonright (\sqsupseteq).$$

What to do now? If we manage to transform the easy part $\mathit{map}\ p? \cdot (\sqsubseteq)$ into a fold, the following *Greedy Theorem* gives us conditions under which we may promote $(\upharpoonright(\sqsupseteq))$ into a fold:

Theorem 1. *$(\!|\, R \upharpoonright S \,|\!) \subseteq (\!|\, R \,|\!) \upharpoonright S$ if S is transitive and R is monotonic with respect to S°, that is, $R \cdot \mathsf{F}S^\circ \subseteq S^\circ \cdot R$.* ***Proof:*** *see appendix C.* □

The "monotonic condition" $R \cdot \mathsf{F}S^\circ \subseteq S^\circ \cdot R$ states that if x_1 is no worse than x_2 under S, at least one output of R on x_1 is no worse than any output on x_2. Thus we lose nothing if we compute only the locally optimal answers, that is, doing $(\upharpoonright S)$ in the fold.

Transforming $\mathit{map}\ p? \cdot (\sqsubseteq)$ into a fold turns out to be easy because, as shown in (21), $(\sqsubseteq)$ is already a fold. By a standard fold-fusion we get:

$$\mathit{map}\ p? \cdot (\sqsubseteq) = (\!|\, \mathit{nil}, \mathit{nil} \cup (\mathit{cons} \cdot (p? \times \mathit{id})) \,|\!),$$

that is, in every step we may choose between taking an empty prefix (nil) and, if the current element satisfies p, attach it to the previously computed prefix ($cons \cdot (p? \times id)$). The monotonicity condition basically says that a longer prefix remains longer after such an operation. Its formal proof makes use of (21), the fact that $(\sqsubseteq)$ is a fold.

By Theorem 1 we may choose $(\![\, [nil, nil \cup (cons \cdot (p? \times id))] \upharpoonright (\sqsupseteq) \,]\!)$ as a candidate for $takeWhile\ p$. By (18), we may distribute $(\upharpoonright(\sqsupseteq))$ into the join. The relation $(nil \cup (cons \cdot (p? \times id))) \upharpoonright (\sqsupseteq)$ returns a longer list whenever possible, that is, whenever the current element satisfies p. Thus the fold refines to $(\![\, nil, ((p \cdot fst) \to nil, cons) \,]\!)$, which translates to the usual definition of $takeWhile$:

$$\begin{array}{lll} takeWhile\ p\ [\,] & = [\,] & \\ takeWhile\ p\ (x : xs) & \mid\ p\ x\ = & x : takeWhile\ p\ xs \\ & \mid\ otherwise\ = & [\,]. \end{array}$$

5.2 Example of DP-programming

Given GC (3) between $(\times y)$ and $(\div y)$, this can be expressed in terms of $(\upharpoonright)$:

$$(\div y)\ =\ ((\times y)^{\circ} \cdot (\leq)) \upharpoonright (\geq).$$

To calculate $(\div y)$, one may proceed the same way as in the previous section and fuse $(\times y)^{\circ}$ into $(\leq)$ to form a fold, and attempt to apply Theorem 1. This time, however, we can not prove the monotonicity condition. Fortunately, for this and many other examples, the following Dynamic Programming Theorem applies:

Theorem 2. *$\mu(\lambda X \to (in \cdot \mathsf{F}X \cdot T^{\circ}) \upharpoonright S)\ \subseteq\ (\![\, T \,]\!)^{\circ} \upharpoonright S$ if in is monotonic with respect to S, that is, $in \cdot \mathsf{F}S \subseteq S \cdot in$, and $dom\ T \subseteq dom\ \mathsf{F}((\![\, T \,]\!)^{\circ} \upharpoonright S)$.* ***Proof:*** *see [14].* □

The notation μf denotes the least fixed point of f. The constructor in can in fact be an arbitrary function, a generalisation we do not need here.

To apply Theorem 2, we aim at turning $(\times y)^{\circ} \cdot (\leq)$ to converse of a fold or, equivalently, turning $(\geq) \cdot (\times y)$ into a fold. It is known that $(\times y) = (\![\, zero, (+y) \,]\!)$: starting with 0, and add y in each step. By fold fusion, we get $(\geq) \cdot (\times y) = (\![\, \top, (+y) \,]\!)$: the base case can be any number.

The monotonicity condition in Theorem 2 instantiates to: $[zero, suc] \cdot (id + (\geq)) \subseteq (\geq) \cdot [zero, suc]$, which can be proved formally using the definition of $(\geq)$ as a fold. Theorem 2 is thus applicable and we get:

$$\mu(\lambda X \to ([zero, suc] \cdot (id + X) \cdot [\top, (+y)]^{\circ}) \upharpoonright (\geq))\ \subseteq\ (\![\, zero, (+y) \,]\!)^{\circ} \upharpoonright (\geq).$$

Denote $(+y)^{\circ}$, a partial function that applies only to input no less than y, by $(-y)$, and note that $zero \cdot \top = zero$. By (25), the left hand side simplifies to $\mu(\lambda X \to (zero \cup (suc \cdot X \cdot (-y))) \upharpoonright (\geq))$. It is a recursive definition where, in every step, we may choose to simply return 0 or, if possible, subtract y from the input and add 1 to the recursively computed result.

We have yet to simplify $(zero \cup (suc \cdot X \cdot (-y))) \upharpoonright (\geq)$. Not having space for the formal detail, we simply note here that since the result of $suc \cdot R$ is strictly larger than 0, to maximise the output, we shall just choose the right branch whenever possible, that is, when the input is no less than y. This results in the usual program for division:

$$\begin{array}{l} x \;\div\; y \mid x \geq y \;=\; 1 + ((x-y) \div y) \\ \qquad\quad\;\; \mid \textbf{otherwise} \;=\; 0. \end{array}$$

6 Case study: scheduling as a Galois Connection

As our closing case study, we will be looking at a more complex problem related to task scheduling. The full detail cannot be covered in this paper, and we will be proceeding in a less formal manner, sketching only an outline of the development.

Let A be a set of tasks, and let $g :: Gantt = \mathsf{P}A \leftarrow A$ such that for each $x \in A$, $g\ x$ is the set of tasks that have to wait for x to complete before commencing, while the spans, time need by each task, is given by a function $Spans = \mathbb{N} \leftarrow A$ where $\mathbb{N}$ models discrete time intervals (eg. days, months). *Gantt* and *Spans* form an acyclic graph, known as a *Gantt graph*, coined after Henry Gantt (1861-1919) who introduced them. A time schedule associating starting times to tasks (optimal or not), is also modelled by a function of type $Schedule = \mathbb{N} \leftarrow A$. The types *Spans* and *Schedule* will be refined later. We use variables sp for *Spans*, sh for *Schedule*, x, y, etc. for tasks, and s, t for time.

Given $g :: Gantt$, the goal is to calculate a function $bsch_g :: Schedule \leftarrow Spans$ that computes the "best" schedule for the tasks — "best" in the sense that tasks start as early as possible. Take, for instance, $A = \{a, b, c, d\}$, for task spans $sp = \{(1, a), (5, b), (10, c), (20, d)\}$ and graph $g = \{(\{b\}, a), (\{c\}, b), (\{\}, c), (\{c\}, d)\}$, the best schedule will be $bsch_g\ sp = \{(0, a), (1, b), (20, c), (0, d)\}$.

How do we specify $bsch_g$? Note that "best" means smallest and that $bsch_g\ sp$ should be monotonic in both arguments: more dependencies in g and/or longer tasks in sp can only defer tasks start-up times into the future. This suggests specifying $bsch_g$ as adjoint of a Galois connection between schedules and spans. Let $lazy_g :: Spans \leftarrow Schedule$ be a function that, given a schedule, computes for each task the maximum time it is allowed to take (hence the name), we have

$$lazy_g\ sh \mathrel{\dot{\geq}} sp \;\equiv\; sh \mathrel{\dot{\geq}} bsch_g\ sp,$$

where $(\dot{\geq})$ denotes $(\geq)$ lifted to functions: $f \mathrel{\dot{\geq}} h \equiv \langle \forall x\ :\ x \in A :\ f\ x \geq h\ x \rangle$.

The function $lazy_g$ appears to be easier to define than $bsch_g$. In the definition below, $(t \hookleftarrow x) \uplus sh$ denotes a function sh, whose domain does not include x, extended with a mapping from x to t.

$$\begin{array}{l} lazy_g\ \{\} \qquad\qquad\qquad\quad = \{\} \\ lazy_g\ ((t \hookleftarrow x) \uplus sh) \;\mid\; g\ x \subseteq dom\ sh \;=\; (s \hookleftarrow x) \uplus sp \\ \qquad \textbf{where}\ sp = lazy_g\ sh \\ \qquad\qquad\qquad s = \sqcap\{sh\ y \mid y \in g\ x\} - t. \end{array}$$

The $\sqcap$ operator in the non-empty case takes the minimum of a set, thus the span allowed for each task x is the difference between the earliest scheduled time among tasks that follow x and t, the time scheduled for x. The non-deterministic pattern $(t \hookleftarrow x) \uplus sh$ does not explicitly specify an order in which tasks are picked. However, the guard $g\ x \subseteq dom\ sh$, needed because we want to look up all the y's in sh, implicitly enforces the topological order — x is processed before all tasks that depend on it. Equivalently, we could have treated the schedule as a list of pairs sorted in topological order: $Schedule = Spans = [(\mathbb{N}, A)]$. One may thus drop the domain check and come up with the following definition for $lazy_g$:

$$\begin{array}{l} lazy_g\ [\,] \qquad\qquad = [\,] \\ lazy_g\ ((t,x) : sh) = (s,x) : lazy_g\ sh \\ \qquad \mathbf{where}\ s = \sqcap\{sh\ y \mid y \in g\ x\} - t. \end{array}$$

For brevity we still use the syntax $sh\ y$ for looking up.

To calculate $bsch_g = ((lazy\ g)^\circ \cdot (\dot{\geq})) \upharpoonright (\dot{\leq})$, we have to construct the converse of $lazy_g$. Consider, in $s = \sqcap\{sh\ y \mid y \in g\ x\} - t$, what t could be given s and x. If $g\ x$ is empty, $s = \infty$, and t could be any finite value. With $g\ x$ non-empty, we have $t = \sqcap\{sh\ y \mid y \in g\ x\} - s$. However, $t :: \mathbb{N}$ must be non-negative. So we are putting an constraint on sh: $\sqcap\{sh\ y \mid y \in G\ x\}$ must be no smaller than s. That gives us a very non-deterministic program for $(lazy_g)^\circ$: we go through the graph in topological order until we reach a task say y, for which $g\ y$ is empty, guess a possible time to schedule it, and go back to some task x that must be done before y. If y is scheduled late enough that x can finish, that's fine. Otherwise this trial fails and we backtrack.

We can refine $(lazy_g)^\circ$ to a more deterministic program that explicitly pass the constraint $\sqcap\{sh\ y \mid y \in g\ x\} \geq s$ down through the recursive calls, so that the choice of t for when $g\ x = \{\}$ is guaranteed to be late enough. We use an extra argument, a mapping from tasks to time, that records the earliest time each task must be scheduled. Initially it is all zero, meaning that there is no constraint yet: $(lazy_g)^\circ\ sp = sche_g\ (sp, \{(z, 0) \mid z \in dom\ sp\})$. In pointfree style, let $init\ sp = (sh, \{(z, 0) \mid z \in dom\ sp\})$, we have $(lazy\ g)^\circ = sche_g \cdot init$.

The main computation happens in $sche$, the name suggesting that it returns a scheduling, but not always the best one. It can be defined as:

$$\begin{array}{l} sche_g\ ([\,], \_) \qquad\quad = [\,] \\ sche_g\ ((s,x) : sp, c) = (t,x) : sche_g\ (sp, c') \\ \qquad \mathbf{where}\ t = \mathbf{if}\ null\ (g\ x)\ \mathbf{then}\ (\text{something no less than } c\ x)\ \mathbf{else}\ c\ x \\ \qquad\qquad\quad c'\ y = \mathbf{if}\ y \notin g\ x\ \mathbf{then}\ c\ y\ \mathbf{else}\ (t+s) \sqcup (c\ y). \end{array}$$

This is an unfold, that is, converse of a fold, on lists. In each step, the next task in topological order is scheduled, and the constraint set c is updated to c' to schedule the rest of the tasks.

Now that we have $bsch_g = (sche_g \cdot init \cdot (\dot{\geq})) \upharpoonright (\dot{\leq})$, the next steps are to fuse $(\dot{\geq})$ into $sche_g \cdot init$ to form an unfold, and to promote $(\upharpoonright(\dot{\leq}))$ into the unfold. Fusing $(\dot{\geq})$ with $sche_g$ merely makes the value of t more non-deterministic: we are left with only $t \geq c\ x$. To promote $(\upharpoonright(\dot{\leq}))$ we need a theorem related to

Theorem 2 that needs a stronger antecedent. It confines the value of t to the smallest possible: $c\ x$. The development concludes with the following program:

$$\begin{array}{ll} bsch_g\ ([\,], \_) & = [\,] \\ bsch_g\ ((s, x) : sp, c) & = (t, x) : bsch_g\ (sp, c') \\ \quad \textbf{where}\ c'\ y = \textbf{if}\ y \notin g\ x\ \textbf{then}\ c\ y\ \textbf{else}\ (c\ x + s) \sqcup (c\ y). & \end{array}$$

7 Conclusions and future work

Poor scalability is often pointed out as a problem of the mathematics of program construction. By contrast, Galois connections are a well-known example of mathematical device which scales up from trivial to complex problem domains. The research programme which embodies this paper starts from the conjecture that the latter could help the former to scale up.

In this context, "programming from Galois connections" is proposed as a way of calculating programs from specifications which take the form of Galois connections. This (emerging) discipline is beneficial in several respects. In particular, the specification of a "hard" operation as adjoint of a GC provides early insight on its properties, well before the actual implementation is derived. This is granted by the rich algebra of GCs, which compose which each other in several ways (thus growing larger and larger) and offer a powerful framework for reasoning about suprema without making these explicit in the calculations.

It should be noted that Galois connections are ubiquitous in mathematics and computer science [13]. In the latter case, they have been shown to offer a powerful way to structure the allegory calculus of Freyd and Ščedrov [8, 4], of which Tarski's relation algebra may in retrospect be seen as an instance [19]. Several examples of such GCs are given in the current paper (see eg. [1, 6, 15] for a detailed account). At the other side of the spectrum, GCs have even been proposed (together with the principle of indirect equality) as the building block of a new brand of theorem provers [18].

In this context, the main contribution of the current paper is to be found in the proposed process of deriving, using the algebra of programming [4], the algorithmic implementation of Galois adjoints, expressed in closed formulæ which record what is "easy" and "hard" to implement. However, instead of resorting to explicit, point-level suprema, as is usual in textbooks, a new pointfree relational combinator (named *shrinking*) is proposed. Thanks to the rich algebra of this combinator, already sketched in [7], one is able to express and generalize previous results on dynamic and greedy programming [4], in a way which dispenses with the heavy artillery of power-allegories [8]. Such results thus become accessible to a wider audience and easier to apply.

The *whole division* example provides a measure of progress: the *verification* of a given algorithm against the given GC (3), carried out in [18], gives place in the current paper to its *construction* from the connection itself.

So much for *pros*. Future work is concerned with a number of *cons*, namely the fact that not every problem casts into a GC. The typical counter-example arises from the (false) lower adjoint being an embedding (or even the identity) and lacking monotonicity.

Still on the negative side, we feel that the conceptual economy of the overall approach is still unmatched by the effort needed to carry out particular examples. A body of knowledge around these results needs to be developed, structured in corollaries, special cases, etc. The general result concerning checking monotonicity in the side conditions of Theorems 1 and 2 given in [14] is an example of what is required.

Last but not least, we find that the *shrinking* combinator has a lot more to offer to algorithmic refinement, in particular with respect to its two-dimensional factorization: either increasing definition or reducing non-determinism [17]. As discussed in Section 3.1, $R \upharpoonright id$ is the largest deterministic fragment of a specification R, that is, that part of R which cannot be further refined. So, in a sense, all effort should go into refining the complement of $R \upharpoonright id$ with respect to R. Embodying this intuition in the greedy and dynamic programming theorems is clearly a subject for future research.

Acknowledgements. Special thanks go to Roland Backhouse for spotting the Galois connection of *take*, which triggered talk [16] and interesting discussions at IFIP WG2.1 thereupon. Thanks are also due to Jeremy Gibbons for his comments on an earlier draft of this paper.

This research was partly supported by the Mondrian Project funded by the Portuguese NSF under contract PTDC/EIA-CCO/108302/2008.

References

1. Aarts, C., Backhouse, R., Hoogendijk, P., E.Voermans, van der Woude, J.: A relational theory of datatypes (December 1992), available from `http://www.cs.nott.ac.uk/~rcb`
2. Backhouse, R.: Galois connections and fixed point calculus. In: LNCS 2297, pp. 89–148. Springer-Verlag (2002)
3. Backhouse, R.: Program Construction: Calculating Implementations from Specifications. John Wiley & Sons, Inc., New York, NY, USA (2003)
4. Bird, R., de Moor, O.: Algebra of Programming. Series in Computer Science, Prentice-Hall International (1997), c.A.R. Hoare, series editor
5. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
6. Doornbos, H., Backhouse, R., van der Woude, J.: A calculational approach to mathematical induction. Theor. Comp. Science 179(1–2), 103–135 (1997)
7. Ferreira, M., Oliveira, J.: Variations on an Alloy-centric tool-chain in verifying a journaled file system model. Technical Report DI-CCTC-10-07, Univ. of Minho (January 2010)
8. Freyd, P., Scedrov, A.: Categories, Allegories, Mathematical Library, vol. 39. North-Holland (1990)
9. Jackson, D.: Software abstractions: logic, language, and analysis. The MIT Press, Cambridge Mass. (2006), iSBN 0-262-10114-9
10. Jones, C.: Software Development — A Rigorous Approach. Prentice-Hall International (1980)

11. Knuth, D.: The Art of Computer Programming. Addison/Wesley, 2nd edn. (1997/98)
12. Kramer, J.: Is abstraction the key to computing? Commun. ACM 50(4), 37–42 (April 2007)
13. Melton, A., Schmidt, D.A., Strecker, G.E.: Galois connections and computer science applications. In: LNCS 240. pp. 299–312. Springer (1986)
14. Mu, S.C., Oliveira, J.: Programming from Galois Connections — Principles and Applications. Tech. Report TR-IIS-10-009, Academia Sinica (December 2010)
15. Oliveira, J.: Extended Static Checking by Calculation using the Pointfree Transform. In: LNCS 5520. pp. 195–251. Springer-Verlag (2009)
16. Oliveira, J.: A Look at Program "Galculation" (January 2010), presentation at the IFIP WG 2.1 #65 Meeting
17. Oliveira, J., Rodrigues, C.: Pointfree Factorization of Operation Refinement. In: FM'06, LNCS, vol. 4085, pp. 236–251. Springer-Verlag (2006)
18. Silva, P., Oliveira, J.: 'Galculator': functional prototype of a Galois-connection based proof assistant. In: PPDP'08. pp. 44–55. ACM, NY (2008)
19. Tarski, A., Givant, S.: A Formalization of Set Theory without Variables. A. M. Society (1987), AMS Colloquium Publications, volume 41.
20. Ullman, J.: Principles of Database Systems. Computer Science Press (1981)
21. Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall (1976)

A Relational calculus

Relations. A relation R from set B to set A, written $R :: A \leftarrow B$, is a subset of the set $\top = \{(a,b) \mid a \in A \wedge b \in B\}$. When $(a,b) \in R$, we say R maps b to a. Set operations such as union, intersection, etc., apply to relations as well. The largest relation (with respect to set inclusion $(\subseteq)$) of its type is $\top$, while the empty relation is denoted by $\bot$. Given $R :: A \leftarrow B$ and $S :: B \leftarrow C$, their composition $R \cdot S :: A \leftarrow C$ is defined by:

$$(a,c) \in (R \cdot S) \;\equiv\; \langle \exists b \;::\; (a,b) \in R \;\wedge\; (b,c) \in S \rangle .$$

Composition is monotonic with respect to $(\subseteq)$. The identity relation $id_A :: A \leftarrow A$ defined by $\langle \forall a \;:\; a \in A : \; (a,a) \in id_A \rangle$ is the unit of composition. We often omit the subscript when it is clear from the context. Given a relation $R :: A \leftarrow B$, its *converse* $R^\circ :: B \leftarrow A$ is defined by $(b,a) \in R^\circ \equiv (a,b) \in R$.

A relation that is a subset of id is said to be *coreflexive*, often used to filter results satisfying certain conditions. Given a predicate p, the coreflexive relation $p?$ is defined by: $(a,a) \in p? \;\equiv\; p\ a$. The domain and range of a relation R are given respectively by $dom\ R = id \cap (R^\circ \cdot R)$ and $ran\ R = id \cap (R \cdot R^\circ)$. A relation R is said to be (1) *simple*, if $(a,b) \in R$ and $(a',b) \in R$ implies $a = a'$, or $R \cdot R^\circ \subseteq id$; (2) *entire*, if every $b \in B$ is mapped to some a, or $id \subseteq R^\circ \cdot R$. A (total) function is a relation that is both simple and entire. As a convention, single small-case letters refer to functions. One nice property of functions is that inclusion equivals equality: $f \subseteq g \equiv f = g$. The following *shunting* rules allows us to move functions to the other side of inclusion:

$$f \cdot R \subseteq S \;\equiv\; R \subseteq f^\circ \cdot S, \qquad\qquad R \cdot f^\circ \subseteq S \;\equiv\; R \subseteq S \cdot f. \tag{23}$$

The relation $R \cdot R^{\circ}$ is called the *image* of R, denoted by $\mathit{img}\ R$.

Given $R :: A \leftarrow B$, $S :: B \leftarrow C$, and $T :: A \leftarrow C$, the relation $T/S :: A \leftarrow B$ is defined by the Galois connection:

$$R \cdot S \subseteq T \ \equiv\ R \subseteq T/S.$$

If $(\cdot S)$ is like multiplication, $(/S)$ is like division: T/S is the largest relation such that $T/S \cdot S \subseteq T$.

Relators, Sum, and Product. A *relator* is an extension of a *functor* in category theory. For the purpose of this paper it suffices to know that a relator F consists of an operation on types that takes a type A to another type $\mathsf{F}A$, and an operation on relations, denoted by the same symbol F, that takes $R :: A \leftarrow B$ to $\mathsf{F}R :: \mathsf{F}A \leftarrow \mathsf{F}B$. A relator is supposed to preserve identity ($\mathsf{F}\mathit{id}_A = \mathit{id}_{\mathsf{F}A}$) and composition ($\mathsf{F}R \cdot \mathsf{F}S = \mathsf{F}(R \cdot S)$), and is monotonic with respect to $(\subseteq)$ ($R \subseteq S \Rightarrow \mathsf{F}R \subseteq \mathsf{F}S$). The unit relator $\mathbf{1}$ takes any type to the unit type (with one element denoted by $()$), and any relation to id.

A bi-relator is a relator generalised to having two arguments. We will need two bi-relators: sum $(+)$ and product $(\times)$. For $(\times)$, the operation on types is the Cartesian product $A \times B$, defined by $\{(a,b) \mid a \in A \wedge b \in B\}$. The projections are $\mathit{fst}\ (a,b) = a$ and $\mathit{snd}\ (a,b) = b$. Given $R :: A \leftarrow C$ and $S :: B \leftarrow C$, the "split" $\langle R, S\rangle :: (A \times B) \leftarrow C$ is defined by:

$$((a,b),c) \in \langle R, S\rangle \ \equiv\ (a,c) \in R \wedge (b,c) \in S.$$

Equivalently, $\langle R, S\rangle = (\mathit{fst}^{\circ} \cdot R) \cap (\mathit{snd}^{\circ} \cdot S)$. The operation on relations is defined using split:

$$(R \times S) \ =\ \langle R \cdot \mathit{fst}, S \cdot \mathit{snd}\rangle.$$

Functional programmers may be more familiar with the special case for functions: $\langle f, g\rangle\ a = (f\ a, g\ a)$, and $(f \times g)\ (a,b) = (f\ a, g\ b)$.

The disjoint sum of two sets A and B is defined by $A + B = \{\mathit{inl}\ a \mid a \in A\} \cup \{\mathit{inr}\ b \mid b \in B\}$, with inl and inr being two injections. Given two relations $R :: A \leftarrow B$ and $S :: A \leftarrow C$, their "join" $[R, S] :: A \leftarrow (B + C)$ is defined by:

$$(a, \mathit{inl}\ b) \in [R,S] \ \equiv\ (a,b) \in R \qquad (a, \mathit{inr}\ c) \in [R,S] \ \equiv\ (a,c) \in S.$$

Equivalently, $[R,S] = (R \cdot \mathit{inl}^{\circ}) \cup (S \cdot \mathit{inr}^{\circ})$. This gives rise to the relator operation on relations:

$$R + S \ =\ [\mathit{inl} \cdot R, \mathit{inr} \cdot S].$$

Note the symmetry between the definitions for sum and product. We will often need this absorption law:

$$[R,S] \cdot (T + U) = [R \cdot T, S \cdot U]. \tag{24}$$

One of the applications of the join is to define the branching operator $(P \rightarrow R, S)$, corresponding to the **if** P **then** R **else** S construct in many programming languages:

$$\begin{aligned}(p \to R, S) &= [R, S] \cdot ((\mathit{inl} \cdot p?) \cup (\mathit{inr} \cdot (\neg p)?)) \\ &= (R \cdot p?) \cup (S \cdot (\neg p)?).\end{aligned}$$

More generally, a common programming pattern is to use the converse of a join $[T, U]^\circ = (\mathit{inl} \cdot T^\circ) \cup (\mathit{inr} \cdot U^\circ)$ to simulate possibly non-deterministic case analysis, and process the two cases by another join. In such situations the following rule comes in handy:

$$[R, S] \cdot [T, U]^\circ \;=\; (R \cdot T^\circ) \cup (S \cdot U^\circ). \tag{25}$$

B Inductively defined datatypes and catamorphisms

Inductively defined datatypes. Natural numbers are often inductively defined to be the smallest set $\mathbb{N}$ such that (a) $0 \in \mathbb{N}$; (b) if $n \in \mathbb{N}$, so does $1 + n$. Let $\mathsf{F}_{\mathbb{N}}$ be a function from sets to sets defined by $\mathsf{F}_{\mathbb{N}} X = \{0\} \cup \{1 + n \mid n \in X\}$. The two conditions together are equivalent to saying that $\mathsf{F}_{\mathbb{N}}\mathbb{N} \subseteq \mathbb{N}$, and the requirement that $\mathbb{N}$ being the smallest means that $\mathbb{N}$ is the *least prefix-point*, and also the *least fixed-point* of $\mathsf{F}_{\mathbb{N}}$. [9]

If we abstract over 0 and $(1+)$, representing them respectively by $\mathit{inl}\ ()$ and inr, F can be expressed as the type operation of relator $\mathsf{F}_{\mathbb{N}} X = \mathbf{1} + X$, where $\mathbf{1}$ is the unit type. Letting $\mathit{in}_{\mathbb{N}} :: \mathbb{N} \leftarrow \mathsf{F}_{\mathbb{N}}\mathbb{N}$ be the isomorphism between $\mathsf{F}_{\mathbb{N}}\mathbb{N}$ and $\mathbb{N}$, the successor function $(1+)$ can be encoded by $\mathit{suc} = \mathit{in}_{\mathbb{N}} \cdot \mathit{inr}$. The number 0 is encoded by $\mathit{in}_{\mathbb{N}}\ (\mathit{inl}\ ())$. In calculations, however, we often find the constant function $\mathit{zero} = \mathit{in}_{\mathbb{N}} \cdot \mathit{inl} \cdot \top$ (that always yields 0 for any input) more useful.

Many inductively defined datatypes can be encoded this way. A finite list of elements of type A, for example, can be defined as the least fixed-point of $\mathsf{F}_{\mathit{List}} X = \mathbf{1} + A \times X$, with constructors $\mathit{nil} :: \mathit{List}\ A \leftarrow B$ defined by $\mathit{in}_{\mathit{List}} \cdot \mathit{inl} \cdot \top$ and $\mathit{cons} :: \mathit{List}\ A \leftarrow (A \times \mathit{List}\ A)$ by $\mathit{in}_{\mathit{List}} \cdot \mathit{inr}$. The type of leaf-valued binary trees, as defined in Haskell notation by **data** $\mathit{Tree}\ A = \mathit{Tip}\ A \mid \mathit{Bin}\ (\mathit{Tree}\ A)\ (\mathit{Tree}\ A)$, is the least fixed-point of $\mathsf{F}_{\mathit{Tree}} X = A + X \times X$.

Catamorphisms. To design programs on these inductively defined datatypes, one is often encouraged to define them over the input inductive structure. The so-called *catamorphism*, also known as *fold*, is one such useful pattern of induction.

Folds exist for all datatypes defined as least fixed-points of so-called *regular* relators: those defined in terms of $\mathbf{1}$, $(+)$, $(\times)$, constants, and type relators. Let T denote the least fixed-point of the type operation of relator F. Given a relation $R :: B \leftarrow \mathsf{F}B$, the catamorphism $(\!|R|\!)_{\mathsf{F}} :: B \leftarrow \mathsf{T}$ is the least prefix point, and also the least fixed-point, of $\lambda X \to R \cdot \mathsf{F}X \cdot \mathit{in}_{\mathsf{T}}^\circ$. Thus it is the least relation satisfying:

$$\begin{aligned}(\!|R|\!)_{\mathsf{F}} &\supseteq R \cdot \mathsf{F}(\!|R|\!)_{\mathsf{F}} \cdot \mathit{in}_{\mathsf{T}}^\circ, \qquad (26)\\ (\!|R|\!)_{\mathsf{F}} &= R \cdot \mathsf{F}(\!|R|\!)_{\mathsf{F}} \cdot \mathit{in}_{\mathsf{T}}^\circ.\end{aligned}$$

[9] For f monotonic on $(\leq)$, x is a prefix-point of f if $f\ x \leq x$, and a fixed-point if $f\ x = x$. The least prefix-point is also the least fixed-point [2].

Take $\mathsf{F}X = \mathbf{1} + A \times X$ as an example, and note that every relation $R :: B \leftarrow (\mathbf{1} + A \times B)$ can be factored to $[R_1, R_2]$ with $R_1 :: B \leftarrow \mathbf{1}$ and $R_2 :: B \leftarrow (A \times B)$. By taking $in_{\mathsf{T}} = [nil, cons]$ and instantiating R_1 and R_2 respectively to a constant and a function we recover *foldr* above.

The fold fusion rule is one of the most important properties of folds:

$$(\!|\, T \,|\!) \subseteq S \cdot (\!|\, R \,|\!)_{\mathsf{F}} \quad \Leftarrow \quad T \cdot \mathsf{F}S \subseteq S \cdot R.$$

It states conditions under which we may promote relations into the body of the fold.

C Proof of Theorem 1

Proof.

$$\begin{array}{ll}
& (\!|\, S \upharpoonright R \,|\!) \;\subseteq\; (\!|\, S \,|\!) \upharpoonright R \\
\equiv & \{ \text{ universal property of } (\upharpoonright) \; \} \\
& (\!|\, S \upharpoonright R \,|\!) \subseteq (\!|\, S \,|\!) \;\wedge\; (\!|\, S \upharpoonright R \,|\!) \cdot (\!|\, S \,|\!)^{\circ} \subseteq R \\
\equiv & \{ \text{ monotonicity of } (\!|\, \_ \,|\!) \text{ and } X \upharpoonright R \subseteq R \; \} \\
& (\!|\, S \upharpoonright R \,|\!) \cdot (\!|\, S \,|\!)^{\circ} \subseteq R \\
\equiv & \{ \text{ hylomorphism: } (\!|\, R \,|\!) \cdot (\!|\, S \,|\!)^{\circ} = \langle \mu X \; :: \; R \cdot \mathsf{F}X \cdot S^{\circ} \rangle \; \} \\
& \langle \mu X \; :: \; (S \upharpoonright R) \cdot \mathsf{F}X \cdot S^{\circ} \rangle \subseteq R \\
\Leftarrow & \{ \text{ least prefix point } \} \\
& (S \upharpoonright R) \cdot \mathsf{F}R \cdot S^{\circ} \subseteq R \\
\Leftarrow & \{ \text{ monotonic condition: } S \cdot \mathsf{F}R^{\circ} \subseteq R^{\circ} \cdot S \; \} \\
& (S \upharpoonright R) \cdot S^{\circ} \cdot R \subseteq R \\
\Leftarrow & \{ \text{ since } S \upharpoonright R \subseteq R/S^{\circ} \; \} \\
& (R/S^{\circ}) \cdot S^{\circ} \cdot R \subseteq R \\
\Leftarrow & \{ \text{ division: } R/S \cdot S \subseteq R \; \} \\
& R \cdot R \subseteq R \\
\equiv & \{ \; R \text{ transitive } \} \\
& \mathit{true}.
\end{array}$$

□