The Journal of Logic and Algebraic Programming 81 (2012) 680–704

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Programming from Galois connections

Shin-Cheng Mu [a,*], José Nuno Oliveira [b,*]

[a] Institute of Information Science, Academia Sinica, Taiwan
[b] High Assurance Software Lab/INESC TEC and Univ. Minho, Portugal

ARTICLE INFO

Article history:
Available online 6 June 2012

Keywords:
Program derivation
Algebra of programming
Galois connection

ABSTRACT

Problem statements often resort to superlatives such as in e.g. "…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 one particular such solution, optimal in some sense (the *hard* part).

This article 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.

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 and practising a lot, the learning curve eventually bents, 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. [1] 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 problem solving in a structured way, shedding light into the core mental activities involved in programming and thus identifying which skills one should acquire to become a good programmer.

Abstraction is one such skill [17]. Abstracting from the programming language and underlying technology is generally accepted as mandatory in the early stages of thinking about a software problem, leading to *abstract modeling*, which has become a discipline in itself [14,15]. However, handling abstractions is not 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. This leads to the *divide-and-conquer* programming strategy

* Corresponding authors.
E-mail addresses: scm@iis.sinica.edu.tw (S.-C. Mu), jno@di.uminho.pt (J.N. Oliveira).

[1] See e.g. the following (by no means exhaustive) list of widely acclaimed References: [5,6,9,16,26,28].

1567-8326/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jlap.2012.05.003

which explains many algorithms, often leading into parallel solutions. 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.

Are there other generic skills, or competences, that one should acquire to become a "good programmer"? This article tries to answer this question 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 a natural number x and positive natural number 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 *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 its use of superlatives. For instance, the computation of the "best" (in the sense of "quickest") schedule for a collection of tasks, given their time spans and an acyclic graph describing which tasks depend upon completion of which other tasks[2] 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 (e.g. some ordering): the largest integer, or the longest prefix, that satisfies the first, easy part of the specification.

In this article 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 article we aim at characterizing problem specifications in terms of *Galois connections*, 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 very natural way to split a problem in its parts, as seen below.

1.1. Paper structure

In Section 2 we argue why Galois connections are suitable as calculational specifications. After giving a minimal review of relational program calculation in Section 3, we motivate and introduce the ($\upharpoonright$) operator in Section 4. If some components in the Galois connection are inductively defined, as reviewed in Section 5, we present in Section 6 two theorems that allows us to calculate the wanted adjoint, demonstrated by two examples. A larger example, scheduling a collection of tasks given a Gantt graph, is presented in Section 7, before we conclude in Section 8. Proofs of main and auxiliary results are deferred to appendices.

2. Galois connections as program specifications

Let us take the problem of writing the algorithm of whole division as starting example.[3] What is its specification, to begin with? This has already been stated above, informally:

$x \div y$ is the largest natural number that, when 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,

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

[2] This is widely known as a *Gantt graph*, a term coined after the surname of the mathematician Henry Gantt (1861–1919) who introduced them.

[3] This example is taken from [5,24].

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

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

at the cost of existentially quantifying over remainders, still too involved for reasoning.

A third alternative [24] comes in the form of an equivalence universally quantified in all its variables,

$$z \times y \le x \;\equiv\; z \le x \div y, \quad (y > 0) \tag{3}$$

and is surprisingly simpler. Pragmatically, it expresses a "shunting" rule which enables one to exchange between a whole division in the upper side of a $(\le)$ inequality of natural numbers 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 [1,10,22,24]. In general, a Galois connection is a pair of functions f and g satisfying

$$f\, z \le x \;\equiv\; z \sqsubseteq g\, x,$$

for all z and x, given preorders $(\le)$ 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 adjoint functions are as identified in

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

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 $(x \div y) \times y \le x$: this tells that $x \div y$ is a candidate solution. Now read (3) from left to right, that is, focus on the implication $z \times y \le x \;\Rightarrow\; z \le x \div y$: 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 $x \div y$ than (1). Some of these will arise from mere instantiation, as is the case of

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

Other properties, for instance

$$x \div 1 = x,$$

call for properties of the lower adjoint (multiplication):

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

That is, every natural number z which is at most $x \div 1$ is also at most x and vice versa. 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*[4]:

[4] 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 alternative to, e.g. circular inclusion: $A = B \;\equiv\; A \subseteq B \wedge B \subseteq A$.

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

More elaborate properties can be inferred from (3) using indirect equality and basic properties of the "easy" adjoint (multiplication), for instance (for $m, d > 0$):

$$(n \div m) \div d \;=\; n \div (d \times m).$$

Again Galois connection (3) blends well with indirect equality in delivering an easy proof:

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

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 which would be required if relying on the obvious recursive implementation of $x \div y$ [24]. Simple (non-inductive) proofs of this kind show the calculational power of Galois connections used as specifications and operated via indirect equality.

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[5]:

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

The property being sought,

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

will rely on another Galois connection – that of defining the minimum of two numbers,

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

in a way which shows how Galois connections compose with each other in a natural and effective way[6]:

$$\begin{array}{ll}
 & z \sqsubseteq take\,(n, take\,(m, x)) \\
\equiv & \quad \{ \text{ Galois connection (5) } \} \\
 & length\, z \leq n \;\wedge\; z \sqsubseteq take\,(m, x)
\end{array}$$

[5] See [20]. The authors would like to thank Roland Backhouse for spotting this Galois connection in the first place, whose upper adjoint $g = take$ is *specified* in terms of a lower adjoint involving *id* and *length*: $f\, z = (length\, z, z)$. Thus the lower ordering is the product partial order ($\leq$) $\times$ ($\sqsubseteq$), defined pointwise in the obvious way.

[6] For detailed accounts of the algebra of Galois connections see e.g. [1,2,20,24]. Reference [2], in particular, already resorts to the pointfree approach adopted in the current paper.

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

Once again, the inductive proof of the same property likely to arise from a recursive definition of *take* (such as that available from the Haskell prelude) 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 article:

How calculational is *programming from Galois connections*?

Reference [24] shows how the defining Galois connection of $(\div)$ provides most of what is required for calculating its implementation. Reference [20] 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 point-free level of the algebra of programming [6], capitalizing on the point-free 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 article and leads us into the core of the research being reported.

3. Preliminaries

In this section we give a minimal review of the point-free calculus of relations. For a thorough introduction, the reader is referred to Aarts et al. [1], and to Bird and de Moor [6] for a categorical perspective.

3.1. 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. Note that orderings like $(\leq)$, $(\sqsubseteq)$, etc, are also relations, mapping "larger" elements to "smaller" ones. 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 $id_A = \{(a, a) \mid a \in A\}$ 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 $\mathit{dom}\ R = id \cap (R^\circ \cdot R)$ and $\mathit{ran}\ R = id \cap (R \cdot R^\circ)$. 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 equivalues equality: $f \subseteq g \equiv f = g$. The following *shunting* rules allows us to move functions to the other side of inclusion:

$$f \cdot R \subseteq S \ \equiv\ R \subseteq f^\circ \cdot S, \quad R \cdot f^\circ \subseteq S \ \equiv\ R \subseteq S \cdot f. \tag{8}$$

The relation $R \cdot R^\circ$ (resp. $R^\circ \cdot R$) is called the *image* of R, denoted by $\mathit{img}\ R$ (resp. the *kernel* of R, denoted by $\mathit{ker}\ R$) [19].

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:

$$R \cdot S \subseteq T \ \equiv\ R \subseteq T/S. \tag{9}$$

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

The following conventions will be adopted for saving parentheses in relational expressions, concerning infix operators (such as e.g. composition $(\cdot)$, $(\cup)$) and unary ones (e.g. converse $(^\circ)$, domain, range):

1. unary and prefix operators bind tighter than binary;
2. "multiplicative" binary operators (e.g. composition, $(\cap)$, $(/)$) bind tighter than "additive" ones (e.g. $(\cup)$);
3. $(/)$ binds tighter than $(\cap)$;
4. relation composition binds tighter than any other multiplicative operator.

3.2. Relators, sum, and product

A *relator* is an extension of a *functor* in category theory. For the purpose of this article 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 ($\mathsf{F}id_A = id_{\mathsf{F}A}$), converse ($\mathsf{F}(R^\circ) = (\mathsf{F}R)^\circ$), 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 $fst\ (a, b) = a$ and $snd\ (a, b) = b$. 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, $\langle R, S\rangle = fst^\circ \cdot R \cap snd^\circ \cdot S$. The operation on relations is defined using split:

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

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 $A + B = \{inl\ a \mid a \in A\} \cup \{inr\ b \mid b \in B\}$, with inl and inr being two range-disjoint injections.[7] Given two relations $R :: A \leftarrow B$ and $S :: A \leftarrow C$, their "join" $[R, S] :: A \leftarrow (B + C)$ is defined by:

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

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

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

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

$$[R, S] \cdot (T + U) = [R \cdot T, S \cdot U]. \tag{10}$$

One of the applications of 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:

$$\begin{aligned}(p \rightarrow R, S) &= [R, S] \cdot (inl \cdot p? \cup inr \cdot (\neg p)?)\\ &= R \cdot p? \cup S \cdot (\neg p)?.\end{aligned} \tag{11}$$

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

$$[R, S] \cdot [T, U]^\circ = R \cdot T^\circ \cup S \cdot U^\circ. \tag{12}$$

Clearly,

$$(p \rightarrow R, S) = [R, S] \cdot [p?, (\neg p)?]^\circ$$

follows from (11), since the converse of a coreflexive is itself.

[7] That is, $inl^\circ \cdot inr = \bot$.

3.3. Higher order relations

Relations are not bound to relating "atomic values" only: they can relate functions with other functions, for instance. An example of this is the so-called *Reynolds arrow* combinator, $R \leftarrow S$, which given two relations $R :: D \leftarrow C$, $S :: B \leftarrow A$, is the *relation on functions*[8] such that

$$(f, g) \in (R \leftarrow S) \;\equiv\; f \cdot S \subseteq R \cdot g. \tag{13}$$

So, f being $(R \leftarrow S)$-related to g means that f and g produce R-related outputs $f\ b$ and $g\ a$ provided their inputs are S-related.[9] We will write notation

$$R \xleftarrow{f} S \tag{14}$$

as abbreviation of $(f, f) \in (R \leftarrow S)$ – the same as $f \cdot S \subseteq R \cdot f$. Taking as example $S := (\leq)$ and $R := (\preceq)$, two partial orders, writing $\preceq \xleftarrow{f} \leq$ will mean that f is monotonic.

This generalizes to relations by replacing function symbol f by that of a relation P,

$$R \xleftarrow{P} S \tag{15}$$

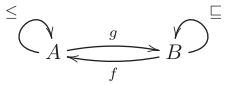
and keeping the meaning $P \cdot S \subseteq R \cdot P$, which is equivalent to $P \subseteq (R \cdot P)/S$, recall (9).[10] Division helps in translating (15) into words: if we regard S and R as relations expressing some kind of *improvement*, (15) states that if x improves y under S, at least one output of P on x is no worse (under R) than any output on y.

The special case $R \xleftarrow{P} \mathsf{F}R$ of (15), for some functor F, is referred to in [6] as a *monotonic condition* on P.

4. Calculating Galois adjoints

Recall the definition of a Galois connection: 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 Galois connection if they satisfy the following equivalence, for all x and y:

$$f\ x \leq y \;\equiv\; x \sqsubseteq g\ y \quad \text{cf. diagram:} \tag{16}$$



It is quite common in Galois connections to have adjoints of disparate complexity. In Galois connection (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:

$$g = (f^{\circ} \cdot (\leq)) \upharpoonright (\sqsupseteq). \tag{17}$$

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 [6], with the significant advantage of not requiring a power allegory.[11] The operator is useful not only for calculating adjoints of Galois connections, but also for constructing certain classes of greedy algorithms, as we will see in Section 6.

4.1. The "shrink" operator

The first step toward manifesting the easy/hard split is to rewrite (16) into point-free 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,

[8] See Reference [3] about the origin of this combinator, which can be regarded as an instance of the concept of *homomorphism* between two relations S and R addressed in [23].

[9] This combinator extensively studied in [2,19] in the context of calculating *theorems for free*.

[10] In the context of program refinement, Engelhardt and de Roever [11] refer to this as "upwards simulation."

[11] Interestingly enough, the combinator arose in [12] in reasoning about sequences in Alloy [14]. This shows how versatile relational algebra is – the same constructs apply evenly at both algorithm and data level.

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:

$$f^\circ \cdot (\leq) \;=\; (\sqsubseteq) \cdot g. \tag{18}$$

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

$$(\sqsubseteq) \cdot g \;\subseteq\; f^\circ \cdot (\leq) \;\wedge \tag{19}$$
$$f^\circ \cdot (\leq) \;\subseteq\; (\sqsubseteq) \cdot g. \tag{20}$$

We show that (19) 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 point-free as

$$(\sqsubseteq) \cdot f^\circ \subseteq f^\circ \cdot (\leq), \tag{21}$$

or $\leq \xleftarrow{f} \sqsubseteq$, recall (14). That (19) implies $g \subseteq f^\circ \cdot (\leq)$ is easy to see – since $(\sqsubseteq)$ is a preorder, $g \subseteq id \cdot g \subseteq (\sqsubseteq) \cdot g$. The other direction goes as follows:

$$\begin{array}{ll} & g \;\subseteq\; f^\circ \cdot (\leq) \\ \Rightarrow & \{\ \text{monotonicity of } (\cdot)\ \} \\ & (\sqsubseteq) \cdot g \;\subseteq\; (\sqsubseteq) \cdot f^\circ \cdot (\leq) \\ \Rightarrow & \{\ \text{assumption: } f \text{ monotonic } (21)\ \} \\ & (\sqsubseteq) \cdot g \;\subseteq\; f^\circ \cdot (\leq) \cdot (\leq) \\ \Rightarrow & \{\ (\leq) \text{ transitive: } (\leq) \cdot (\leq) \subseteq (\leq)\ \} \\ & (\sqsubseteq) \cdot g \;\subseteq\; f^\circ \cdot (\leq). \end{array}$$

Concerning (20):

$$\begin{array}{ll} & f^\circ \cdot (\leq) \;\subseteq\; (\sqsubseteq) \cdot g \\ \equiv & \{\ \text{take converses of both sides}\ \} \\ & (f^\circ \cdot (\leq))^\circ \;\subseteq\; g^\circ \cdot (\sqsupseteq) \\ \equiv & \{\ \text{shunting } (8)\ \} \\ & g \cdot (f^\circ \cdot (\leq))^\circ \;\subseteq\; (\sqsupseteq). \end{array}$$

All in all, we have just factored Galois connection (19) into two parts,

$$f^\circ \cdot (\leq) \;=\; (\sqsupseteq) \cdot g \;\;\equiv\;\; \underbrace{g \;\subseteq\; f^\circ \cdot (\leq)}_{\text{“easy”}} \;\wedge\; \underbrace{g \cdot (f^\circ \cdot (\leq))^\circ \;\subseteq\; (\sqsupseteq)}_{\text{“hard”}}, \tag{22}$$

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

There is nothing surprising in this: we have merely reconstructed an equivalent definition of a Galois connection [1, Theorem 5.29, page 66]:

- f is monotonic,
- $(f \cdot g)\,x \leq x$,
- $f\,x \leq y \;\Rightarrow\; x \sqsubseteq g\,y$.

The calculation above, however, inspires us to capture this pattern by a new relational operator. Given relations $S :: A \leftarrow B$ and $R :: A \leftarrow A$, define $S \upharpoonright R : A \leftarrow B$, pronounced “$S$ shrunk by R”, by

$$X \subseteq S \upharpoonright R \;\equiv\; X \subseteq S \;\wedge\; X \cdot S^\circ \subseteq R\text{, cf. diagram: } \begin{array}{ccc} & & B \\ & {\scriptstyle S \upharpoonright R}\swarrow & \downarrow{\scriptstyle S} \\ A & \xleftarrow[R]{} & A \end{array} \tag{23}$$

The definition states that X must be at most S, and that if X yields an output for an input x, it must be a maximum, with respect to R, among all possible outputs of x. In terms of the easy/hard split, S is the easy part and R defines the (optimisation) 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 $S \upharpoonright R$:

$$S \upharpoonright R \;=\; S \cap R/S^{\circ}. \tag{24}$$

With the new notation we can go back to (22) and rephrase the right hand side of the equivalence in terms of combinator $(\upharpoonright)$:

$$g \subseteq (f^{\circ} \cdot (\leq)) \upharpoonright (\sqsupseteq). \tag{25}$$

4.2. Properties of shrinking

In this section we go through some basic but useful properties of the $(\upharpoonright)$ operator. Proofs of some of the properties are given in Appendix A. From the definition (23), it is clear that $S \upharpoonright R \subseteq S$. It is easy to find out under what condition the other direction of inclusion holds: $S \subseteq S \upharpoonright R$ iff $S \cdot S^{\circ} \subseteq R$, and so

$$S = S \upharpoonright R \;\equiv\; img\, S \subseteq R. \tag{26}$$

since $img\, S = S \cdot S^{\circ}$. Since $\top$ is above anything, we have one immediate consequence:

$$S \upharpoonright \top = S,$$

that is, S stays the same if we put no constraints on the "hard" part.

When $R = \bot$, no maximum exists, and thus $S \upharpoonright \bot$ yields nothing for any input:

$$S \upharpoonright \bot = \bot.$$

Two rules allow us to distribute a function in and out of $(\upharpoonright R)$:

$$(S \cdot f) \upharpoonright R \;=\; (S \upharpoonright R) \cdot f, \tag{27}$$
$$(f \cdot S) \upharpoonright R \;=\; f \cdot (S \upharpoonright (f^{\circ} \cdot R \cdot f)). \tag{28}$$

Generalizing these results to the composition of two relations is not as immediate. Nevertheless, the following inclusion holds wherever T is less defined than S:

$$(S \cdot T^{\circ}) \upharpoonright R \;\subseteq\; (R \cdot S)/T \quad \Leftarrow \quad dom\, T \subseteq dom\, S. \tag{29}$$

The next rule shows how $(\upharpoonright R)$ distributes into relational union:

$$(S \cup T) \upharpoonright R \;=\; ((S \upharpoonright R) \cap R/T^{\circ}) \cup ((T \upharpoonright R) \cap R/S^{\circ}). \tag{30}$$

This arises from (24) and distribution of intersection over union. An important outcome of (30) is that $(\upharpoonright R)$ distributes into joins,

$$[S, T] \upharpoonright R \;=\; [S \upharpoonright R, T \upharpoonright R], \tag{31}$$

– recalling that $[S, T] = (S \cdot inl^{\circ}) \cup (T \cdot inr^{\circ})$ – and therefore into conditionals:

$$(p \rightarrow S, T) \upharpoonright R \;= p \rightarrow (S \upharpoonright R), (T \upharpoonright R). \tag{32}$$

A number of results of the $(\upharpoonright)$ combinator relate to simplicity. Recall that the image of a simple relation S is coreflexive, that is, $img\, S \subseteq id$. Then, from (26) we draw

$$S = S \upharpoonright R \;\Leftarrow\; S \text{ simple and } R \text{ reflexive}, \tag{33}$$

since $img\, S \subseteq id$ and $id \subseteq R$ entail $img\, S \subseteq R$.

Very often, R in (23) is anti-symmetric: $R \cap R^\circ \subseteq id$. In this case it can be shown that $S \upharpoonright R$ is always simple [12]. An application of this result concerns (25), ensuring $(f^\circ \cdot (\preceq)) \upharpoonright (\sqsupseteq)$ simple for $(\sqsupseteq)$ a partial order. Thus equality (17) holds in such a situation.

The special case $R = id$ in (23) deserves some attention. In this situation, each output in the shrunk relation can relate only to itself. Thus $(y, x) \in S \upharpoonright id$ only when y is the sole value that x is mapped to by S. When more than one such y exists, x cannot be in the domain of $S \upharpoonright id$. Therefore, $S \upharpoonright id$ is the largest deterministic fragment of S. [12] Formally,

$$X \subseteq S \upharpoonright id \;\equiv\; X \sqsubseteq S \;\wedge\; X \cdot X^\circ \subseteq id, \tag{34}$$

where $X \sqsubseteq S$ means $S \cdot dom\, X = X$, that is, X is less defined than S but as non-deterministic as S where defined. This is the $\vdash_{pre}$ ordering of [21], where it is shown to be a factor of the standard refinement ordering. The proof of (34), given in Appendix A, essentially shows that the right hand sides of (23) and (34) coincide, for $S = id$.

4.3. Shrinking versus overriding and conditionals

Quite often shrinking is implemented by conditionals in a way to be made precise below. To begin with, consider expressions of the form $R \cap \bot/S^\circ$, whose meaning is best grasped using indirection:

$$X \subseteq R \cap \bot/S^\circ \equiv X \subseteq R \;\wedge\; X \cdot S^\circ \subseteq \bot.$$

So $R \cap \bot/S^\circ$ is the largest subrelation of R whose domain is disjoint of that of S. Using this, we define the *overriding* combinator [13]

$$R \dagger S \;=\; S \cup R \cap \bot/S^\circ \tag{35}$$

meaning the relation which contains the whole of S and that part of R where S is undefined (read $R \dagger S$ as "R overridden by S").

Overriding helps in deriving another corollary of the union rule (30) whose pattern often arises in calculations: the competition between arbitrary R and simple S, under a given partial order $(\preceq)$ whereby the outcome of S is always "better" than that of R:

$$(R \cup S) \upharpoonright (\preceq) \;=\; (R \upharpoonright (\preceq)) \dagger S \;\Leftarrow\; S \mathrel{\dot{\prec}} R. \tag{36}$$

The side condition expands to

$$S \mathrel{\dot{\prec}} R \equiv S \cdot R^\circ \subseteq (\preceq)\ ;\ \bot \tag{37}$$

where lexicographic $(\preceq)\ ;\ \bot$ means that wherever both S and R are defined, the outputs of the former are *strictly* smaller than those of the latter. [14] Details can be found in Appendix A, together with the proof of (36).

Quite often, R in (36) is a function f and S takes the form of a fragment $g \cdot p?$ of another function g; expressed in this way, overriding clearly hides a conditional:

$$(f \cup g \cdot p?) \upharpoonright (\preceq) \;=\; f \dagger (g \cdot p?) \;=\; p \to g\ ,\ f. \tag{38}$$

5. 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 article, in two steps. First, we recall that the "natural" way of ordering inductively defined data (such as e.g. lists and trees) is through inductive relations defined using well-known combinators of the algebra of programming known as *folds* and *unfolds* [6]. Second, we show how specifications written as Galois connections on such inductive orderings "naturally" lead to inductive implementations, by calculation.

5.1. 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 is $1 + n$. Let $\mathsf{F}_{\mathbb{N}}$ be a function from sets to sets defined by $\mathsf{F}_{\mathbb{N}} X = \{0\} \cup \{suc\, n \mid n \in X\}$, where $suc\, n = 1 + n$. The two conditions together

[12] The *univalent part* of S, in the terminology of [27].
[13] This extends the *map override* operator of VDM [15].
[14] Lexicographic relations are dealt with in [6].

are equivalent to saying that $\mathsf{F}_{\mathbb{N}}\,\mathbb{N} \subseteq \mathbb{N}$, and the requirement that $\mathbb{N}$ is the smallest means that $\mathbb{N}$ is the *least prefix-point*, and also the *least fixed-point* of $\mathsf{F}_{\mathbb{N}}$.[15]

If we abstract over 0 and *suc*, representing them respectively by *inl* () and *inr*, F can be expressed as the type operation of relator $\mathsf{F}_{\mathbb{N}}\,X = \mathbf{1} + X$ (where $\mathbf{1}$ and $(+)$ respectively denote the unit and sum relators, defined in Section 3, rather than numerical one and sum). Letting $in_{\mathbb{N}} :: \mathbb{N} \leftarrow \mathsf{F}_{\mathbb{N}}\,\mathbb{N}$ be the isomorphism between $\mathsf{F}_{\mathbb{N}}\,\mathbb{N}$ and $\mathbb{N}$, the number 0 is encoded by $in_{\mathbb{N}}$ (*inl* ()). In point-free calculation, however, we will make more use of the constant function *zero*, which maps () to 0. It and the successor function *suc* can be encoded by[16]

$$\begin{aligned} &zero :: \mathbb{N} \leftarrow \mathbf{1} && suc :: \mathbb{N} \leftarrow \mathbb{N} \\ &zero = in_{\mathbb{N}} \cdot inl, && suc = in_{\mathbb{N}} \cdot inr. \end{aligned}$$

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 $\mathsf{F}_{List}X = \mathbf{1} + A \times X$, with constructors

$$\begin{aligned} &nil :: List\ A \leftarrow \mathbf{1} && cons :: List\ A \leftarrow (A \times List\ A) \\ &nil = in_{List} \cdot inl, && cons = in_{List} \cdot inr. \end{aligned}$$

The type of leaf-valued binary trees, as defined in Haskell by **data** *Tree A* = *Tip A* | *Bin* (*Tree A*) (*Tree A*), is the least fixed-point of $\mathsf{F}_{Tree}X = A + X \times X$.

5.2. Catamorphisms

To design programs on these inductively defined datatypes, one is often encouraged to define the function on the inductive structure of its input. The *catamorphism*, also known as *fold*, is one such useful pattern of induction. Functional programmers are familiar with *foldr* defined on lists:

$$\begin{aligned} &foldr\ f\ e\ [\,] && = e \\ &foldr\ f\ e\ (x : xs) && = f\ (x, foldr\ f\ e\ xs). \end{aligned}$$

Knowing that $\mathbb{N}$ is an inductively defined datatype, a fold function can also be defined on $\mathbb{N}$:

$$\begin{aligned} &foldN\ f\ e\ 0 && = e \\ &foldN\ f\ e\ (1 + n) && = f\ (foldN\ f\ e\ n). \end{aligned}$$

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 $\lambda X \rightarrow R \cdot \mathsf{F}X \cdot in_{\mathsf{T}}^{\circ}$. Thus it is the least relation satisfying:

$$(\!|\, R \,|\!)_{\mathsf{F}}\ \supseteq R \cdot \mathsf{F}(\!|\, R \,|\!)_{\mathsf{F}} \cdot in_{\mathsf{T}}^{\circ}, \tag{39}$$

$$(\!|\, R \,|\!)_{\mathsf{F}}\ = R \cdot \mathsf{F}(\!|\, R \,|\!)_{\mathsf{F}} \cdot in_{\mathsf{T}}^{\circ}. \tag{40}$$

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:

$$(\!|\, T \,|\!) \subseteq S \cdot (\!|\, R \,|\!)_{\mathsf{F}}\ \Leftarrow\ T \cdot \mathsf{F}S \subseteq S \cdot R. \tag{41}$$

It states conditions under which we may promote relations into the body of the fold. We will need this rule later.

[15] 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 [4].

[16] We will soon abuse the notation and generalise the types of *zero* and *suc* respectively to $\mathbb{N} \leftarrow A$ and $List\ A \leftarrow B$. See the end of this section.

5.3. Inductively defined orderings

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

$$\begin{array}{l} x \geq 0 \quad \wedge \\ x \geq y \;\Rightarrow\; suc\, x \geq suc\, y. \end{array}$$

The first line translates, to point-free style, to $(zero \cdot !)^\circ \subseteq (\geq)$, where $! :: \mathbf{1} \leftarrow A$ is the function mapping every value in A to $()$, and thus the relation $zero \cdot !$, with its type restricted to $\mathbb{N} \leftarrow \mathbb{N}$, relates an arbitrary natural number to 0. The second line translates to $(\geq) \subseteq suc^\circ \cdot (\geq) \cdot suc$. We reason:

$$\begin{array}{ll} & (zero \cdot !)^\circ \subseteq (\geq) \;\wedge\; (\geq) \subseteq suc^\circ \cdot (\geq) \cdot suc \\ \equiv & \{\ \text{shunting}\ \} \\ & (zero \cdot !)^\circ \subseteq (\geq) \;\wedge\; suc \cdot (\geq) \cdot suc^\circ \subseteq (\geq) \\ \equiv & \{\ \text{since } R \subseteq T \wedge S \subseteq T \;\equiv\; R \cup S \subseteq T\text{; converses}\ \} \\ & (!^\circ \cdot zero^\circ) \cup (suc \cdot (\geq) \cdot suc^\circ) \subseteq (\geq) \\ \equiv & \{\ \text{by (12): } [R, S] \cdot [T, U]^\circ = (R \cdot T^\circ) \cup (S \cdot U^\circ)\ \} \\ & [!^\circ, suc \cdot (\geq)] \cdot [zero, suc]^\circ \subseteq (\geq) \\ \equiv & \{\ \text{absorption (10)}\ \} \\ & [!^\circ, suc] \cdot (id + (\geq)) \cdot [zero, suc]^\circ \subseteq (\geq) \\ \equiv & \{\ (39)\ \} \\ & (\geq) \;=\; (\!|\, !^\circ, suc \,|\!). \end{array}$$

Thus $(\geq)$ is a fold.

This is 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:

$$\begin{array}{l} 0 \leq y \quad \wedge \\ x \leq y \;\Rightarrow\; suc\, x \leq suc\, y. \end{array}$$

The first line translates to $zero \cdot ! \subseteq (\leq)$. By a similar calculation, we come up with a definition of $(\leq)$ as a fold:

$$(\leq) \;=\; (\!|\, zero, zero \cdot ! \cup suc \,|\!). \tag{42}$$

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:

$$(\sqsubseteq) \;=\; (\!|\, nil, nil \cdot ! \cup cons \,|\!). \tag{43}$$

For the rest of the paper we will make frequent use of functions like $zero \cdot ! :: \mathbb{N} \leftarrow A$ and $nil \cdot ! :: List\, A \leftarrow B$ – the "base case" of a datatype composed with !, forming a constant function that maps any value to the base case. For simplicity we also denote them by *zero* and *nil*.

Knowing that lists are special cases of binary trees, one might define a fold with a similar structure expressing the ordering on trees which "grow" by substitution of empty nodes by any other (sub)trees. Folds (42) and (43) will be regarded in Appendix C as instances of orderings on generic *pointed datatypes* for which a number of useful results are derived.

6. 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 4.1 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 fold followed by the converse of a fold [17]; (2) shrinking the easy part using the hard part ($\upharpoonright(\sqsupseteq)$), hence the *motto*: "shrinking specs into programs".

[17] This is also known as a *hylomorphism* [6].

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 [6]: 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, but with a different precondition arising from its more general setting.

Both theorems are datatype-generic, and in fact applicable not only for problems specified as Galois connections, but also for optimisation problems in general.

6.1. Example of greedy programming

Given a predicate p, *takeWhile p xs* yields the longest prefix of *xs* whose elements all satisfy p:

$$all\ p\ xs\ \wedge\ xs \sqsubseteq ys\ \ \equiv\ \ xs \sqsubseteq takeWhile\ p\ ys. \tag{44}$$

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 *takeWhile p* and the lower adjoint is the embedding of all such sequences into the larger set. To see this we rewrite (44) into the pointfree equality

$$map\ p? \cdot (\sqsubseteq)\ \ =\ \ (\sqsubseteq) \cdot takeWhile\ p$$

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

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

$$takeWhile\ p\ \ =\ \ (map\ p? \cdot (\sqsubseteq)) \restriction (\sqsupseteq).$$

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

Theorem 1. $(\!|\ S \restriction R\ |\!) \subseteq (\!|\ S\ |\!) \restriction R$ if R is transitive and S is monotonic with respect to R°, that is, $R^\circ \prec\!\!\xleftarrow{S} \mathsf{F}R^\circ$ holds.

Proof. See Appendix B. □

Recalling (15), monotonic condition $R^\circ \prec\!\!\xleftarrow{S} \mathsf{F}R^\circ$ means the same as $S \cdot \mathsf{F}R^\circ \subseteq R^\circ \cdot S$ and states that if x_1 is no worse than x_2 under R, at least one output of S 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 $(\restriction R)$ in the fold.

Transforming $map\ p? \cdot (\sqsubseteq)$ into a fold turns out to be easy because, as shown in (43), $(\sqsubseteq)$ is already a fold. By a standard fold-fusion we get:

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

that is, in every step we may choose between taking an empty prefix (*nil*) or, 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. For a formal proof, it expands to

$$\begin{array}{rcl} nil & \subseteq & (\sqsubseteq) \cdot nil\ \ \wedge \\ (nil \cup (cons \cdot (p? \times id))) \cdot (id \times (\sqsubseteq)) & \subseteq & (\sqsubseteq) \cdot (nil \cup (cons \cdot (p? \times id))). \end{array}$$

The first condition easily follows from the fact that $id \subseteq (\sqsubseteq)$. The interesting part of verifying the second condition is verifying that $cons \cdot (p? \times (\sqsubseteq)) \subseteq (\sqsubseteq) \cdot cons \cdot (p? \times id)$, which is true because $cons \cdot (id \times (\sqsubseteq)) \subseteq (\sqsubseteq) \cdot cons$, following from (43). Theorem 3 in Appendix C provides a general result which discharges this verification.

By Theorem 1 we may choose $(\!|\ [nil, nil \cup (cons \cdot (p? \times id))] \restriction (\sqsupseteq)\ |\!)$ as a candidate for *takeWhile p*. By (31), we may distribute $(\restriction(\sqsupseteq))$ into the join. The relation $(nil \cup (cons \cdot (p? \times id))) \restriction (\sqsupseteq)$ returns a longer list whenever possible, that is, whenever the current element satisfies p. We express this intuition using relation overriding notation introduced by (35), since $cons \mathbin{\dot{\sqsupset}} nil$ and therefore (38) is applicable:

$$(nil \cup (cons \cdot (p? \times id))) \restriction (\sqsupseteq) = nil \dagger (cons \cdot (p? \times id)).$$

Thus the fold refines to $(\!|\ nil, ((p \cdot fst) \rightarrow cons, nil)\ |\!)$, which translates to the usual definition of *takeWhile*:

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

6.2. Broadening scope

Connection (44) is paradigmatic of a number of situations in which a programmer is asked to deliver that part of a data structure whose elements all meet a particular criterion. What changes in each situation is what is meant by *being a part of*, that is, what the ordering ($\sqsubseteq$) actually is.

Querying a database file, for instance, fits in the paradigm: one wishes to select from the file those records which satisfy a particular query. Above, "that part" and "those records" obviously mean "the largest part" and "the largest collection of records", respectively. Letting such collections be sequences, we compare parts thereof by the subsequence ordering: $[\,]$ is the only subsequence of itself and, denoting y subsequence of x by $y \preceq x$, $y \preceq (a : t)$ iff either $y \preceq t$ (a dropped) or, for some t', $y = a : t'$ and $t' \preceq t$ (a is kept). This ordering is captured by a fold, once again:

$$(\preceq) = (\!|\ nil, snd \cup cons\ |\!). \tag{45}$$

Clearly, at each step, ($\preceq$) either selects the head of the input list or decides to ignore it, yielding a subsequence.

The upper adjoint of (44) for prefix ($\sqsubseteq$) replaced by subsequence ($\preceq$) is the *filter p* function is Haskell, whose calculation is very similar to that of *takeWhile*: by Theorem 1 we reach $(\!|\ [nil, snd \cup (cons \cdot (p? \times id))]\ |\!)(\succeq)\ |\!)$ as a candidate implementation. Thereupon shrinking the recursive step, $(snd \cup (cons \cdot (p? \times id)))\ \restriction\ (\succeq)$, we easily check that $snd \mathrel{\dot{\prec}} cons$ holds and thus (38) is applicable once again. We obtain $snd \dagger (cons \cdot (p? \times id))$, that is, $p \cdot fst \rightarrow cons\ ,\ snd$. The fold thus obtained translates into the expected code, different from that of *takeWhile* only in the *otherwise* clause:

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

Not every Galois connection leads to algorithms in this way. For instance, the examples given in the introductory part of this paper – *take* and integer division – end up not being folds. Let us see why and how we go about them.

6.3. Example of DP-programming

Given the Galois connection (3) between multiplication and division, $(\div y)$ can be expressed in terms of $\restriction$:

$$(\div y)\ =\ ((\times y)^\circ \cdot (\leq))\ \restriction\ (\geq).$$

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 cannot prove the monotonicity condition.

Fortunately, for this and many other examples, the following Dynamic Programming Theorem applies. Let $\langle \mu X\ ::\ f\ X \rangle$ denote the least fixed point of f, the theorem goes:

Theorem 2. Let $M = ((\!|\ S\ |\!) \cdot (\!|\ T\ |\!)^\circ)\ \restriction\ R$, we have $\langle \mu X\ ::\ (S \cdot \mathsf{F}X \cdot T^\circ)\ \restriction\ R \rangle\ \subseteq\ M$ if S is monotonic with respect to R, that is, $R \xleftarrow{S} \mathsf{F}R$ holds, and $dom\ T \subseteq dom\ (S \cdot \mathsf{F}M)$.

Proof. See Appendix B. □

As a special case, by taking $S = in$ (and thus $(\!|\ S\ |\!) = id$), we have

$$\langle \mu X\ ::\ (in \cdot \mathsf{F}X \cdot T^\circ)\ \restriction\ R \rangle\ \subseteq\ (\!|\ T\ |\!)^\circ\ \restriction\ R,$$

if $in \cdot \mathsf{F}R \subseteq R \cdot in$.

Theorem 2 is named so because it captures the essence of dynamic programming: relation T° transforms the input into sub-problems in all possible ways, on which we recursively compute optimal solutions (using $\mathsf{F}X$). Those optimal solutions for sub-problems are combined using h, before the best one is picked by ($\restriction R$). It is possible that the sub-problems may overlap, and one may want to apply further transform the program into one that caches the results, perhaps in a table or

other data structures, to avoid re-computation. As we will see later, the situation for $(\div y)$ is even simpler – we will be able to determine, before the recursive calls, which choice is better, and produce a greedy algorithm.

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 $(\times y)$ can be defined in terms of a fold: $(\times y) = (\!|\, zero, (+y)\,|\!)$. By fold fusion, we get: $(\geq) \cdot (\times y) = (\!|\, \top, (+y)\,|\!)$. Thus the "easy part" of the specification is relation $E = (\!|\, \top, (+y)\,|\!)^\circ$ which, downto points, unfolds into

$$z\ E\ x \equiv z = 0 \vee \langle \exists m, n\ :\ m\ E\ n :\ z = m + 1 \wedge x = n + y \rangle,$$

meaning that, at any time, E counts a subtraction of y from input x or gives up, yielding 0, as the base case can be any number.

The monotonicity condition in Theorem 2 instantiates to:

$$[zero, suc] \cdot (id + (\geq)) \ \subseteq\ (\geq) \cdot [zero, suc], \tag{46}$$

which expands to two terms: $zero \subseteq (\geq) \cdot zero$, which is true because $id \subseteq (\geq)$, and $suc \cdot (\geq) \subseteq (\geq) \cdot suc$, which follows from the definition of $(\geq)$ as a fold. This is a special case of a more general result:

Theorem 3. Let $R = (\!|\, I\,|\!)$ where $in \subseteq I$. Then $R \xleftarrow{in} \mathsf{F}R$ holds, that is, it is always true that $in \cdot \mathsf{F}R \subseteq R \cdot in$.

Proof.

$$\begin{array}{ll} & in \cdot \mathsf{F}R \subseteq R \cdot in \\ \equiv & \{\ \text{fold cancellation}\ \} \\ & in \cdot \mathsf{F}R \subseteq I \cdot \mathsf{F}R \\ \Leftarrow & \{\ \text{monotonicity}\ \} \\ & in \subseteq I. \quad \square \end{array}$$

Theorem 2 is thus applicable and we get:

$$\langle \mu X\ ::\ ([zero, suc] \cdot (id + X) \cdot [\top, (+y)]^\circ) \restriction (\geq) \rangle \ \subseteq\ (\!|\, zero, (+y)\,|\!)^\circ \restriction (\geq).$$

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 (12), the left hand side simplifies to $\langle \mu X\ ::\ (zero \cup (suc \cdot X \cdot (-y))) \restriction (\geq) \rangle$. It is a recursive definition which, at every step, chooses the largest of either returning 0 or, if possible, subtracting y from the input and adding 1 to the recursively computed result.

We have yet to simplify $(zero \cup (suc \cdot X \cdot (-y))) \restriction (\geq)$. For an intuition, note that since the result, if any, of $suc \cdot Y$ for any Y is strictly larger than 0, [18] to maximise the output, we shall just choose the right branch whenever possible, that is, when the input is no less than y. So we can rely on (36) and convert shrinking into overriding:

$$(zero \cup (suc \cdot X \cdot (-y))) \restriction (\geq) = zero \dagger (suc \cdot X \cdot (-y)).$$

Therefore, $(zero \cup suc \cdot X \cdot (-y)) \restriction (\geq)$ simplifies to $((\geq y) \to suc \cdot X \cdot (-y), zero)$, that is, we perform $suc \cdot X \cdot (-y)$ only if the input is in the domain of $(-y)$. Otherwise we return 0. This results in the usual program for division:

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

As a final check up and further to monotonicity, already dealt with in (46), we need to check that $dom\, T = dom\, [\top, (+y)] \subseteq dom\, (\mathsf{F}(\div y))$ holds, which it does (both entire).

7. Case study: scheduling as a Galois connection

As our closing case study, we will be looking at a larger problem related to task scheduling. Due to its complex nature, 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 $g\ ::\ \mathsf{P}A \leftarrow A$ 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 needed by each task, is given by a function $\mathbb{N} \leftarrow A$ where $\mathbb{N}$ models

[18] See (C.4) for a generic result supporting this claim.

discrete time intervals (e.g. days, months). Dependencies in g 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 $\mathbb{N} \leftarrow A$. In summary, we introduce the following types:

$$Gantt = \mathsf{P}A \leftarrow A, \qquad Spans = \mathbb{N} \leftarrow A, \qquad Schedule = \mathbb{N} \leftarrow A.$$

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, $(t \leftarrow\!\!\!\!\cdot\ x) \uplus sh$ denotes a function *sh*, whose domain does not include x, extended with a mapping from x to t.

$$\begin{aligned}
&lazy_g\ \{\} &&= \{\}\\
&lazy_g\ ((t \leftarrow\!\!\!\!\cdot\ x) \uplus sh)\ \mid\ g\ x \subseteq dom\ sh\ &&=\ (s \leftarrow\!\!\!\!\cdot\ x) \uplus sp\\
&\quad \textbf{where}\ sp = lazy_g\ sh\\
&\qquad\quad s = \sqcap\{sh\ y \mid y \in g\ x\} - t.
\end{aligned}$$

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 $(t \leftarrow\!\!\!\!\cdot\ x) \uplus sh$ 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 $lazy_g$:

$$\begin{aligned}
&lazy_g\ [\,] &&= [\,]\\
&lazy_g\ ((t, x) : sh) &&= (s, x) : lazy_g\ sh\\
&\quad \textbf{where}\ s = \sqcap\{sh\ y \mid y \in g\ x\} - t,
\end{aligned}$$

where, 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 point-free 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:

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

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.

Theorem 4. Let $H = (\!|\ S\ |\!) \cdot (\!|\ T\ |\!)^\circ$ where S is a simple relation, and $M = H \upharpoonright R$. We have $\langle \mu X\ ::\ h \cdot \mathsf{F}X \cdot (T^\circ \upharpoonright Q)\rangle \subseteq M$ if S is monotonic on R and $S \cdot \mathsf{F}H \cdot Q^\circ \subseteq R^\circ \cdot S \cdot \mathsf{F}H$.

Proof. See Appendix B. □

While Theorem 2 potentially makes recursive calls on all possible values suggested by T° and picks a optimal one by R, Theorem 4 allows us to decide earlier on a value returned by T° using $(\upharpoonright Q)$, if we are sure that a better value under Q always leads to a better result under R.

Application of Theorem 4 confines the value of t to the smallest possible: $c\ x$. The development concludes with the following program:

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

8. Conclusions and future work

Poor scalability is often pointed out as a serious problem in 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 article 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. First, the specification of a "hard" operation as adjoint of a Galois connection provides early insight on the properties of the adjoint being sought, well before the actual implementation is derived. This is granted by the rich algebra of Galois connections, which compose with 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 stressed that Galois connections are ubiquitous in mathematics and computer science [18]. In the latter case, they have been shown to offer a powerful way to structure the allegory calculus of Freyd and Ščedrov [6,13], of which Tarski's relation algebra may in retrospect be seen as an instance [25]. Several examples of such Galois connections are given in the current article (see e.g. [1,10,19] for a detailed account). At the other side of the spectrum, they have even been proposed (together with the principle of indirect equality) as the building block of a new brand of theorem provers [24].

In this context, the main contribution of the current article is to be found in the proposed process of deriving, using the algebra of programming [6], 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 relational combinator (named *shrinking*) is proposed which expresses such formulæ at pointfree level.

Thanks to the rich algebra of this combinator, already sketched in [12], one is able to express and generalize previous results on dynamic and greedy programming by Bird and de Moor [6], in a way which dispenses with the heavy artillery of power-allegories [13]. As a side effect, such results 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 Galois connection (3), carried out in [24], now gives place 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 Galois connection. The typical counter-example arises from the (false) lower adjoint being an embedding (or even the identity) and lacking monotonicity. Still calculations can proceed, but more work and experience is required before concluding. Functions arising in bioinformatics (eg, in finding sections of DNA dense with mutations) such as the *shortest maximally-dense prefix* (two superlatives!) [8] remain a challenge.

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 Appendix C 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 [21]. As discussed in Section 4.1, $R \upharpoonright id$ captures 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 matter of future research.

Acknowledgements

The authors are indebted to the anonymous referees for detailed and useful comments which helped to improve this paper. Thanks are also due to Jeremy Gibbons for useful comments on an earlier draft.

Special thanks go to Roland Backhouse for spotting the Galois connection of *take*, which triggered talk [20] and interesting discussions at IFIP WG2.1 thereupon. Previous joint work in the field of one of the authors with Paulo Silva is also acknowledged.

This research was partly supported by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010047.

Appendix A. Proofs concerning the algebra of shrinking

A.1. Distributing functions in and out of $(\upharpoonright R)$

Eq. (27) can be easily proved using indirect equality and function shunting. Eq. (28) is the $(\upharpoonright)$ counterpart of property (7.8) in Bird and de Moor [6] and its proof is similar:

$$
\begin{array}{ll}
 & (f \cdot S) \upharpoonright R \\
= & \quad \{ \text{ definition (24) } \} \\
 & (f \cdot S) \cap (R/(S^\circ \cdot f^\circ)) \\
= & \quad \{ \text{ modular law for simple relations [6] } \} \\
 & f \cdot (S \cap (f^\circ \cdot (R/(S^\circ \cdot f^\circ)))) \\
= & \quad \{ \text{ division: } f^\circ \cdot (R/S) = (f^\circ \cdot R)/S \ \} \\
 & f \cdot (S \cap ((f^\circ \cdot R)/(S^\circ \cdot f^\circ))) \\
= & \quad \{ \text{ division: } R/(S \cdot f^\circ) = (R \cdot f)/S \ \} \\
 & f \cdot (S \cap ((f^\circ \cdot R \cdot f)/S^\circ)) \\
= & \quad \{ \text{ definition (24) } \} \\
 & f \cdot (S \upharpoonright (f^\circ \cdot R \cdot f)).
\end{array}
$$

A.2. Shrinking and division

The calculation of (29) is short:

$$
\begin{array}{ll}
 & (S \cdot T^\circ) \upharpoonright R \\
\subseteq & \quad \{ \text{ since } X \upharpoonright R \subseteq R/X^\circ \ \} \\
 & R/(T \cdot S^\circ) \\
\subseteq & \quad \{ \text{ since } R/(T \cdot S^\circ) \subseteq (R \cdot S)/T \text{ if } dom\ T \subseteq dom\ S, \text{ see below } \} \\
 & (R \cdot S)/T.
\end{array}
$$

The last step relies on a result proved by indirect inclusion:

$$X \subseteq R/(T \cdot S^\circ)$$

$$
\begin{array}{ll}
\Rightarrow & \{\ \text{division; monotonicity of composition}\ \} \\
& X \cdot T \cdot S^\circ \cdot S \subseteq R \cdot S \\
\Rightarrow & \{\ dom\ S \subseteq S^\circ \cdot S\ \} \\
& X \cdot T \cdot dom\ S \subseteq R \cdot S \\
\equiv & \{\ dom\ T \subseteq dom\ S \text{ assumed; division}\ \} \\
& X \subseteq (R \cdot S)/T.
\end{array}
$$

A.3. Checking for deterministic fragments of specifications

The aim is to prove (34):

$$X \subseteq S \upharpoonright id \ \equiv\ S \cdot dom\ X = X \ \wedge\ X \cdot X^\circ \subseteq id.$$

The reasoning is as follows:

$$
\begin{array}{ll}
& S \cdot dom\ X = X \ \wedge\ X \cdot X^\circ \subseteq id \\
\equiv & \{\ \text{replacing } X^\circ \text{ by } (S \cdot dom\ X)^\circ\ \} \\
& S \cdot dom\ X = X \ \wedge\ X \cdot dom\ X \cdot S^\circ \subseteq id \\
\equiv & \{\ X \cdot dom\ X = X\ \} \\
& S \cdot dom\ X = X \ \wedge\ X \cdot S^\circ \subseteq id \\
\equiv & \{\ X \subseteq S \cdot dom\ X \equiv X \subseteq S\ \} \\
& S \cdot dom\ X \subseteq X \ \wedge\ X \subseteq S \ \wedge\ X \cdot S^\circ \subseteq id \\
\equiv & \{\ \text{claim: } X \cdot S^\circ \subseteq id \ \Rightarrow\ S \cdot dom\ X \subseteq X, \text{ see below}\ \} \\
& X \subseteq S \ \wedge\ X \cdot S^\circ \subseteq id \\
\equiv & \{\ \text{universal property of } (\upharpoonright)\ (23)\ \} \\
& X \subseteq S \upharpoonright id.
\end{array}
$$

The claim is easy to discharge:

$$
\begin{array}{ll}
& X \cdot S^\circ \subseteq id \\
\equiv & \{\ \text{converses}\ \} \\
& S \cdot X^\circ \subseteq id \\
\Rightarrow & \{\ \text{monotonicity of } (\cdot)\ \} \\
& S \cdot X^\circ \cdot X \subseteq X \\
\Rightarrow & \{\ dom\ X \subseteq X^\circ \cdot X\ \} \\
& S \cdot dom\ X \subseteq X.
\end{array}
$$

A.4. Shrinking versus overriding

The aim is to prove (36), where $(\preceq)$ is a partial order and S is simple. Unfolding lexicographic order $(\preceq)\ ;\ \bot$ [6], side condition (37) becomes

$$S \mathbin{\dot{\prec}} R \equiv S \cdot R^\circ \subseteq (\preceq) \ \wedge\ S \cdot R^\circ \cap (\preceq)^\circ \subseteq \bot \tag{A.1}$$

$$\equiv S \subseteq (\preceq)/R^\circ \ \wedge\ S \cdot R^\circ \cap id \subseteq \bot \tag{A.2}$$

since $(\preceq)$ is anti-symmetric. This helps in the calculation:

$$
\begin{array}{ll}
& (R \cup S) \upharpoonright (\preceq) \\
= & \{\ \text{union rule (30)}\ ;\ S \upharpoonright (\preceq) = S\ (33)\ \} \\
& ((R \upharpoonright (\preceq)) \cap (\preceq)/S^\circ) \ \cup\ (S \cap (\preceq)/R^\circ) \\
= & \{\ \text{definition of } (\upharpoonright);\ (A.2)\ \}
\end{array}
$$

$$
\begin{array}{ll}
& R \cap (\preceq)/R^\circ \cap (\preceq)/S^\circ \ \cup\ S \\
= & \quad \{\ \text{converses: } S \cdot R^\circ \subseteq (\preceq) \equiv R \subseteq (\preceq)^\circ/S^\circ\ \} \\
& R \cap (\preceq)/R^\circ \cap (\preceq)/S^\circ \cap (\preceq)^\circ/S^\circ \ \cup\ S \\
= & \quad \{\ \text{division ; anti-symmetry: } (\preceq) \cap (\preceq)^\circ = id\ \} \\
& (R \upharpoonright (\preceq)) \cap id/S^\circ \ \cup\ S \\
= & \quad \{\ \text{see below}\ \} \\
& (R \upharpoonright (\preceq)) \cap \bot/S^\circ \ \cup\ S \\
= & \quad \{\ \text{definition (35)}\ \} \\
& (R \upharpoonright (\preceq)) \dagger S.
\end{array}
$$

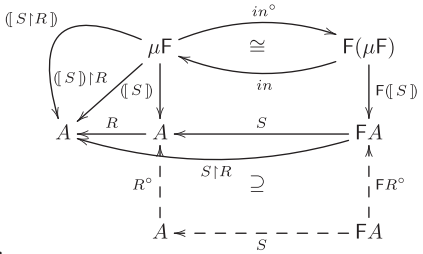
That id can be replaced by $\bot$ in the penultimate step above can be checked by indirect equality:

$$
\begin{array}{ll}
& X \subseteq (R \upharpoonright (\preceq)) \cap id/S^\circ \\
\equiv & \quad \{\ \text{division ; converses}\ \} \\
& X \subseteq (R \upharpoonright (\preceq)) \wedge X \cdot S^\circ \subseteq id \\
\equiv & \quad \{\ A \subseteq B \ \equiv\ A \cap B = A\ \} \\
& X \subseteq (R \upharpoonright (\preceq)) \wedge X \cdot S^\circ \cap id = X \cdot S^\circ \\
\equiv & \quad \{\ X \cdot S^\circ \cap id = \bot \text{ by (A.2), since } X \text{ is at most } R\ \} \\
& X \subseteq (R \upharpoonright (\preceq)) \wedge X \cdot S^\circ = \bot \\
\equiv & \quad \{\ X \cdot S^\circ = \bot \text{ the same as } X \cdot S^\circ \subseteq \bot\ ;\ \text{division}\ \} \\
& X \subseteq (R \upharpoonright (\preceq)) \wedge X \subseteq \bot/S^\circ \\
\equiv & \quad \{\ \text{meet}\ \} \\
& X \subseteq (R \upharpoonright (\preceq)) \cap \bot/S^\circ.
\end{array}
$$

Appendix B. Proofs of theorems

Proof of Theorem 1

Inspect the following diagram before looking at the proof below. Notation $\mu\mathsf{F}$ means the *least prefix point* (and thus also the *least fixed point*) of F. The theorem's side condition is depicted in dashed arrows:



Proof.

$$
\begin{array}{ll}
& (\!|\, S \upharpoonright R \,|\!) \ \subseteq\ (\!|\, S \,|\!) \upharpoonright R \\
\equiv & \quad \{\ \text{universal property of } (\upharpoonright)\ (23)\ \} \\
& (\!|\, S \upharpoonright R \,|\!) \subseteq (\!|\, S \,|\!) \ \wedge\ (\!|\, S \upharpoonright R \,|\!) \cdot (\!|\, S \,|\!)^\circ \subseteq R \\
\equiv & \quad \{\ \text{monotonicity of } (\!|\, \_ \,|\!) \text{ and } S \upharpoonright R \subseteq S\ \} \\
& (\!|\, S \upharpoonright R \,|\!) \cdot (\!|\, S \,|\!)^\circ \subseteq R \\
\equiv & \quad \{\ \text{hylomorphism: } (\!|\, R \,|\!) \cdot (\!|\, S \,|\!)^\circ = \langle \mu X \ ::\ R \cdot \mathsf{F}X \cdot S^\circ \rangle\ \} \\
& \langle \mu X \ ::\ (S \upharpoonright R) \cdot \mathsf{F}X \cdot S^\circ \rangle \subseteq R \\
\Leftarrow & \quad \{\ \text{least prefix point}\ \}
\end{array}
$$

$$
\begin{array}{ll}
& (S \upharpoonright R) \cdot \mathsf{F}R \cdot S^\circ \subseteq R \\
\Leftarrow & \{\ \text{monotonicity condition: } S \cdot \mathsf{F}R^\circ \subseteq R^\circ \cdot S\ \} \\
& (S \upharpoonright R) \cdot S^\circ \cdot R \subseteq R \\
\Leftarrow & \{\ \text{since } S \upharpoonright R \subseteq R/S^\circ\ \} \\
& (R/S^\circ) \cdot S^\circ \cdot R \subseteq R \\
\Leftarrow & \{\ \text{division: } R/S \cdot S \subseteq R\ \} \\
& R \cdot R \subseteq R \\
\equiv & \{\ R \text{ transitive}\ \} \\
& \textsc{True}. \quad \square
\end{array}
$$

Proof of Theorem 2

Proof. For brevity we let $H = (\!|\, S \,|\!) \cdot (\!|\, T \,|\!)^\circ$, and thus $M = H \upharpoonright R$. The aim is to prove $\langle \mu X :: (S \cdot \mathsf{F}X \cdot T^\circ) \upharpoonright R \rangle \subseteq M$. We reason:

$$
\begin{array}{ll}
& \langle \mu X :: (S \cdot \mathsf{F}X \cdot T^\circ) \upharpoonright R \rangle \subseteq M \\
\Leftarrow & \{\ \text{least prefix point}\ \} \\
& (S \cdot \mathsf{F}M \cdot T^\circ) \upharpoonright R \subseteq M \\
\equiv & \{\ \text{universal property of } (\upharpoonright)\ (23)\ \} \\
& (S \cdot \mathsf{F}M \cdot T^\circ) \upharpoonright R \subseteq H \wedge \\
& ((S \cdot \mathsf{F}M \cdot T^\circ) \upharpoonright R) \cdot H^\circ \subseteq R.
\end{array}
$$

The two proof obligations are proved separately. The first of them can be discharged by:

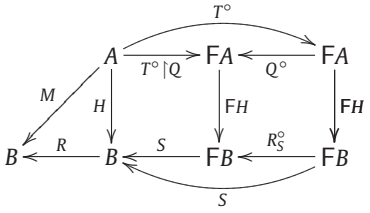
$$
\begin{array}{ll}
& (S \cdot \mathsf{F}(H \upharpoonright R) \cdot T^\circ) \upharpoonright R \\
\subseteq & \{\ X \upharpoonright R \subseteq X\ \} \\
& S \cdot \mathsf{F}(H \upharpoonright R) \cdot T^\circ \\
\subseteq & \{\ X \upharpoonright R \subseteq X, \mathsf{F} \text{ relator}\ \} \\
& S \cdot \mathsf{F}H \cdot T^\circ \\
\subseteq & \{\ H = (\!|\, S \,|\!) \cdot (\!|\, T \,|\!)^\circ, \text{hylomorphism}\ \} \\
& H.
\end{array}
$$

Concerning the second proof obligation:

$$
\begin{array}{ll}
& ((S \cdot \mathsf{F}\, M \cdot T^\circ) \upharpoonright R) \cdot H^\circ \\
\subseteq & \quad \{\ \text{property of shrinking } (29),\ dom\ T \subseteq dom\ (S \cdot \mathsf{F}M)\ \} \\
& ((R \cdot S \cdot \mathsf{F}\, M)/T) \cdot H^\circ \\
= & \quad \{\ H = S \cdot \mathsf{F}\, H \cdot T^\circ\ ;\ \text{converses}\ \} \\
& ((R \cdot S \cdot \mathsf{F}\, M)/T) \cdot T \cdot \mathsf{F}\, H^\circ \cdot S^\circ \\
\subseteq & \quad \{\ \text{division cancellation}\ \} \\
& R \cdot S \cdot \mathsf{F}\, M \cdot \mathsf{F}\, H^\circ \cdot S^\circ \\
= & \quad \{\ \text{substitutions; relator } \mathsf{F}\ \} \\
& R \cdot S \cdot \mathsf{F}((H \upharpoonright R) \cdot H^\circ) \cdot S^\circ \\
\subseteq & \quad \{\ \text{since } (H \upharpoonright R) \cdot H^\circ \subseteq R/H^\circ \cdot H^\circ \subseteq R\ \} \\
& R \cdot S \cdot \mathsf{F}\, R \cdot S^\circ \\
\subseteq & \quad \{\ R \xleftarrow{h} \mathsf{F}\, R\text{, that is, } S \cdot \mathsf{F}\, R \cdot S^\circ \subseteq R\ \} \\
& R \cdot R \\
\subseteq & \quad \{\ R \text{ transitive}\ \} \\
& R. \quad \square
\end{array}
$$

Proof of Theorem 4

Proof. Recall $H = (\!|\, S \,|\!) \cdot (\!|\, T \,|\!)^\circ$ and $M = H \upharpoonright R$. The aim is to prove $\langle \mu X :: S \cdot \mathsf{F}X \cdot (T^\circ \upharpoonright Q) \rangle \subseteq M$ under the hypothesis of S being simple and monotonic on R and $S \cdot \mathsf{F}H \cdot Q^\circ \subseteq R^\circ \cdot S \cdot \mathsf{F}H$ holding. As earlier on, it is useful to draw a diagram in order to see what is going on, where R°_S abbreviates $S^\circ \cdot R^\circ \cdot S$:



The proof goes:

$$
\begin{array}{ll}
 & \langle \mu X :: S \cdot \mathsf{F}X \cdot (T^\circ \upharpoonright Q) \rangle \subseteq M \\
\Leftarrow & \{ \text{ least prefix point } \} \\
 & S \cdot \mathsf{F}M \cdot (T^\circ \upharpoonright Q) \subseteq M \\
\equiv & \{ \text{ universal property of } (\upharpoonright) \ \} \\
 & S \cdot \mathsf{F}M \cdot (T^\circ \upharpoonright Q) \subseteq H \ \wedge \ S \cdot \mathsf{F}M \cdot (T^\circ \upharpoonright Q) \cdot H^\circ \subseteq R.
\end{array}
$$

The first requirement is proved below:

$$
\begin{array}{ll}
 & S \cdot \mathsf{F}M \cdot (T^\circ \upharpoonright Q) \\
\subseteq & \{ \ T^\circ \upharpoonright Q \subseteq T^\circ \ \} \\
 & S \cdot \mathsf{F}M \cdot T^\circ \\
\subseteq & \{ \ M = H \upharpoonright R \subseteq H \ \} \\
 & S \cdot \mathsf{F}H \cdot T^\circ \\
= & \{ \ H = (\!|\, S \,|\!) \cdot (\!|\, T \,|\!)^\circ, \text{ hylomorphism } \} \\
 & H.
\end{array}
$$

For the second, we reason:

$$
\begin{array}{ll}
 & S \cdot \mathsf{F}M \cdot (T^\circ \upharpoonright Q) \cdot H^\circ \\
\subseteq & \{ \text{ since } X \upharpoonright Q \subseteq Q/X^\circ \ \} \\
 & S \cdot \mathsf{F}M \cdot (Q/T) \cdot H^\circ \\
= & \{ H = S \cdot \mathsf{F}H^\circ \cdot T^\circ \ \} \\
 & S \cdot \mathsf{F}M \cdot (Q/T) \cdot T \cdot \mathsf{F}H^\circ \cdot S^\circ \\
\subseteq & \{ \text{ division cancellation } \} \\
 & S \cdot \mathsf{F}M \cdot Q \cdot \mathsf{F}H^\circ \cdot S^\circ \\
\subseteq & \{ \text{ assumption: } S \cdot \mathsf{F}H \cdot Q^\circ \subseteq R^\circ \cdot S \cdot \mathsf{F}H \ \} \\
 & S \cdot \mathsf{F}M \cdot \mathsf{F}H^\circ \cdot S^\circ \cdot R \\
\subseteq & \{ \text{ since } (H \upharpoonright R) \cdot H^\circ \subseteq R/H^\circ \cdot H^\circ \subseteq R \ \} \\
 & S \cdot \mathsf{F}R \cdot S^\circ \cdot R \\
\subseteq & \{ \text{ monotonicity: } S \cdot \mathsf{F}R \subseteq R \cdot S \ \} \\
 & R \cdot S \cdot S^\circ \cdot R \\
\subseteq & \{ \ S \text{ simple } \} \\
 & R \cdot R \\
\subseteq & R. \quad \square
\end{array}
$$

Appendix C. Ordering pointed data types

C.1. Pointed types

Lists and natural numbers are examples of *pointed types* [1,7], of shape $\mathsf{T} \cong 1 + \mathsf{F}\mathsf{T}$. Thus $in :: \mathsf{T} \leftarrow 1 + \mathsf{F}\mathsf{T}$ is made of two constructors, say *base* and *rec* ($in = [base, rec]$), one for the base case and the other for the recursive one. In the natural numbers (resp. lists), $base = zero$ (resp. nil) and $rec = suc$ (resp. $cons$).

Note that $base :: \mathsf{T} \leftarrow 1$ is always a constant function, for instance $zero\ \_ = 0$ and $nil\ \_ = [\,]$. Below we will need a more general version of *base*, $(base \cdot !) :: \mathsf{T} \leftarrow A$ which is free on the input type, the output being the same. For simplicity, we write *base* instead of $base \cdot !$. Algebra *in* being an isomorphism (therefore injective), one has:

$$base^{\circ} \cdot rec = \bot. \tag{C.1}$$

C.2. Ordering pointed types

Two structural orderings can be defined on T,

$$(\geq) = (\!|\, \top, rec \,|\!), \tag{C.2}$$

$$(\leq) = (\!|\, base, base \cup rec \,|\!), \tag{C.3}$$

which are reflexive and transitive (see below) and establish *base* as universal lower bound, cf. $x \geq 0$ in the natural numbers and $[\,] \leq x$, for arbitrary suitably typed x. For lists, $x \leq y$ means that x is a prefix of y. The fact that *base* is strictly smaller than *rec*,

$$base \mathrel{\dot{<}} rec, \tag{C.4}$$

proves useful in applying rules such as e.g. (36) to pointed types. To prove (C.4) we resort to (A.1, A.2):

$$base \mathrel{\dot{<}} rec \equiv base \cdot rec^{\circ} \subseteq (\leq) \wedge base \cdot rec^{\circ} \cap id \subseteq \bot.$$

The first conjunct follows from (C.3):

$$\begin{array}{ll} & base \cdot rec^{\circ} \subseteq (\leq) \\ \equiv & \quad \{\ \text{shunting ; } in = [base, rec]\ \} \\ & base \subseteq (\leq) \cdot in \cdot inr \\ \equiv & \quad \{\ (\text{C.3}) \text{ followed by } (40)\ \} \\ & base \subseteq [base, base \cup rec] \cdot (id + \mathsf{F}(\leq)) \cdot inr \\ \equiv & \quad \{\ \text{coproducts and sums}\ \} \\ & base \subseteq (base \cup rec) \cdot \mathsf{F}(\leq) \\ \equiv & \quad \{\ \text{distribution ; } base \text{ is a constant function}\ \} \\ & base \subseteq base \cup rec \cdot \mathsf{F}(\leq) \\ \equiv & \quad \{\ \text{trivial}\ \} \\ & \text{TRUE}. \end{array}$$

The second conjunct relies on the following corollary of exercise 4.33 of [6],

$$f \cdot g^{\circ} \cap id \subseteq \bot \equiv id \cap f^{\circ} \cdot g \subseteq \bot, \tag{C.5}$$

as follows:

$$base \cdot rec^{\circ} \cap id \subseteq \bot$$

$$
\begin{array}{ll}
\equiv & \{\ (C.5)\ \} \\
& id \cap base^{\circ} \cdot rec \subseteq \bot \\
\Leftarrow & \{\ \text{monotonicity}\ \} \\
& base^{\circ} \cdot rec \subseteq \bot \\
\equiv & \{\ (C.1)\ \} \\
& \textsc{True}.
\end{array}
$$

Both (C.2, C.3) define orderings as folds of the form $(\!|\ I\ |\!)$ where $in \subseteq I$. This grants them reflexive (by fold reflexion) and transitive, as can be checked by fusion (41) – $(\geq) \cdot (\!|\ \top, rec\ |\!) = (\!|\ \top, rec\ |\!)$ and $(\leq) \cdot (\!|\ base, base \cup rec\ |\!) = (\!|\ base, base \cup rec\ |\!)$, respectively. We need only be concerned with the banches in which I differs from in: $(\geq) \cdot \top = \top$ in the case of (C.2), which stems from $(\geq)$ being reflexive and, in the case of (C.3), the recursive branch:

$$
\begin{array}{ll}
& (\leq) \cdot (base \cup rec) = (base \cup rec) \cdot \mathsf{F}(\leq) \\
= & \{\ \text{distribution ; } base \text{ a constant function and } \mathsf{F}(\leq) \text{ entire (total)}\ \} \\
& (\leq) \cdot base \cup (\leq) \cdot rec = base \cup rec \cdot \mathsf{F}(\leq) \\
= & \{\ \text{fold cancellation}\ \} \\
& base \cup ((base \cup rec) \cdot \mathsf{F}(\leq)) = base \cup rec \cdot \mathsf{F}(\leq) \\
= & \{\ \text{same as two steps above}\ \} \\
& base \cup base \cup rec \cdot \mathsf{F}(\leq) = base \cup rec \cdot \mathsf{F}(\leq) \\
= & \{\ \text{trivial}\ \} \\
& \textsc{True}.
\end{array}
$$

Finally note that, thanks to Theorem 3, both $in \subseteq [\top, rec]$ and $in \subseteq [base, base \cup rec]$ hold and we may conclude that in is therefore always monotonic on $(\geq)$ and $(\leq)$.

References

[1] C. Aarts, R.C. Backhouse, P. Hoogendijk, E. Voermans, J. van der Woude, A relational theory of datatypes, December 1992. Available from: <http://www.cs.nott.ac.uk/ rcb>.
[2] K. Backhouse, R.C. Backhouse, Safety of abstract interpretations for free, via logical relations and Galois connections, SCP 15 (1–2) (2004) 153–196.
[3] R.C. Backhouse, On a relation on functions, in: W. Dijkstra (Ed.), Beauty is Our Business: A Birthday Salute to Edsger, Springer-Verlag, New York, NY, USA, 1990, pp. 7–18.
[4] R.C. Backhouse, Galois connections and fixed point calculus, in: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, LNCS, vol. 2297, Springer-Verlag, 2002, pp. 89–148.
[5] R.C. Backhouse, Program Construction: Calculating Implementations from Specifications, John Wiley & Sons Inc., New York, NY, USA, 2003.
[6] R. Bird, O. de Moor, Algebra of Programming, Series in Computer Science, Prentice-Hall International, 1997. Available from: http://progtools.comlab.ox.ac.uk/members/oege/publications/aop97
[7] R. Bird, O. de Moor, P. Hoogend, J. Funct. Programming 6 (1) (1996) 1–28.
[8] K.-M. Chung, H.-I. Lu, An optimal algorithm for the maximum-density segment problem, SIAM J. Comput. 34 (2) (2004) 373–387.
[9] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.
[10] H. Doornbos, R. Backhouse, J. van der Woude, A calculational approach to mathematical induction, TCS 179 (1–2) (1997) 103–135.
[11] K. Engelhardt, W.-P. de Roever, Simulation of specification statements in Hoare logic, in: Mathematical Foundations of Computer Science, LNCS, vol. 1113, 1996, pp. 324–335.
[12] M.A. Ferreira, J.N. Oliveira, Variations on an alloy-centric tool-chain in verifying a journaled file system model, Technical Report DI-CCTC-10-07, University of Minho, January 2010.
[13] P.J. Freyd, A. Scedrov, Categories, Allegories, in: Mathematical Library, vol. 39, North-Holland, 1990.
[14] D. Jackson, Software Abstractions: Logic, Language, and Analysis, revised ed., The MIT Press, Cambridge, MA, 2012.ISBN 0-262-01715-2
[15] C.B. Jones, Software Development – A Rigorous Approach, Prentice-Hall International, 1980.
[16] D.E. Knuth, The Art of Computer Programming, second ed., Addison/Wesley, 1997/98.
[17] J. Kramer, Is abstraction the key to computing?, Commun. ACM 50 (4) (2007) 37–42.
[18] A. Melton, D.A. Schmidt, G.E. Strecker, Galois connections and computer science applications, in: Category Theory and Computer Programming, LNCS, vol. 240, Springer, pp. 299–312, 1986.
[19] J.N. Oliveira, Extended static checking by calculation using the pointfree transform, in: Language Engineering and Rigorous Software Development, LNCS, vol. 5520, Springer-Verlag, 2009, pp. 195–251.
[20] J.N. Oliveira, A Look at Program 'Galculation', Presentation at the IFIP WG 2.1 #65 Meeting, January 2010.
[21] J.N. Oliveira, C.J. Rodrigues, Pointfree factorization of operation refinement, in: FM 2006: Formal Methods, LNCS, vol. 4085, Springer-Verlag, 2006, pp. 236–251.
[22] E. Orłowska, I. Rewitzky, Algebras for Galois-style connections and their discrete duality, Fuzzy Sets and Systems 161 (9) (2010) 1325–1342.Foundations of Lattice-Valued Mathematics with Applications to Algebra and Topology

[23] G. Schmidt, Homomorphism and isomorphism theorems generalized from a relational perspective, in: Relations and Kleene Algebra in Computer Science, LNCS, vol. 4136, Springer, 2006, pp. 328–342.
[24] P.F. Silva, J.N. Oliveira, 'Galculator': functional prototype of a Galois-connection based proof assistant, in: PPDP'08, ACM, NY, 2008, pp. 44–55.
[25] A. Tarski, S. Givant, A Formalization of Set Theory without Variables, American Mathematical Society, 1987.
[26] J.D. Ullman, Principles of Database Systems, Computer Science Press, 1981.
[27] M. Winter, Products in categories of relations, J. Logic Algebr. Program. 76 (1) (2008) 145–159.
[28] N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, 1976.