Program Derivation with Relations, Crescents, Bendy Loops and Colored Wires

Gustavo de Mendonça Freire*, Hugo Musso Gualandi, Hugo Nobrega, and João Paixão

Institute of Computing
Universidade Federal do Rio de Janeiro, Brazil
{gustavofreire,hugomg,hugonobrega,jpaixao}@ic.ufrj.br

Abstract. Program derivations calculate, via step-by-step reasoning, an algorithm which implements a given specification. In practice, however, the effectiveness of calculational methods depends on the mathematical language in which they are expressed, and known techniques tend to be verbose. As a way to make calculations more approachable and easier to discover, we study the application of a graphical language of string diagrams to the process of program derivation. By having diagrams represent *feasibility relations*, the notion of a Galois connection, which specifies functions as computing solutions to an optimization problem, becomes very simple to represent. This enables elegant point-free specifications, as well as proofs and program derivations that are built step-by-step from equalities between diagrams. We discuss how to represent derivational techniques graphically, and propose a novel point-free induction principle suited for string diagrams. For graphical induction we need to encode additional typing information in the diagrams, which we represent using colored wires. Finally, we showcase the methodology by deriving correct-by-construction recursive algorithms for problems on natural numbers and lists.

Keywords: Functional programming, String diagrams, Program calculation, Galois connection, Feasibility relations

1 Introduction

The algebraic approach to programming seeks practical mathematical tools to derive correct algorithms from specifications. However, since the goal is not just to demonstrate that a given implementation is correct, but also to give tools that allow a correct implementation to be *calculated*, the practicality of these methods depends on the syntax in which they are expressed. Ideally, it should be concise, legible, and intuitive. If any of the above fail, the calculational process can be rendered infertile: step-by-step rewriting with long formulas quickly becomes

* Corresponding author.

tiring, and working with hard-to-decipher expressions can make it difficult to spot rewrite opportunities or possible next logical steps.

From this point of view, we can contrast two ways to reason about programs:

- the point-full style, which expresses specifications and derivations as sequences of implications or equivalences between quantified logical formulas; (Cousot and Cousot, 1979; Silva and Oliveira, 2008)
- the point-free relational algebra approach, which works with relations and relational operators, such as composition, union, intersection, catamorphisms, and shrinking. Relations are compared using equality or inclusion. (Backhouse and Backhouse, 2004; Mu and Oliveira, 2012)

Each has its drawbacks. The first can end up with an excess of auxiliary variables and nested quantifiers, which complicate propositions and make calculating more difficult, while expressing the required forms of composition with the operators from relational algebra can create long (and somewhat inscrutable) expressions.

As an alternative, we turn to the two-dimensional graphical syntax of *string diagrams* (Selinger, 2011). They are a formal notation whose meaning is mathematically specified by composition rules, but their expressions are given by intuitive diagrams in which boxes denoting relations or mappings are connected by wires. However, to reason about recursive programs, the calculational strategies should support induction over recursive data types, which is something the existing string diagram literature is not well-equipped for. It is rich in examples of induction over the structure of the diagrams themselves, but not of induction over the recursive data carried by the wires (Piedeleu and Zanasi, 2023b).

We propose a novel graphical approach to induction, which utilizes the underlying types to track where it is justified to apply the induction hypothesis. We color the wires based on their types, making this distinction easy to spot. Our approach is remarkably simple, but we have not seen it applied before.

To evaluate our method, we choose to study program derivations specified by Galois connections (Bird and de Moor, 1997; Backhouse, 2002), which have provided a structure for program specifications which is both expressive, in the sense that many programs can be specified in this way, and productive, serving as a useful starting point for derivations (Mu and Oliveira, 2012; Oliveira, 2023). We employ string diagrams that denote *feasibility relations* (Censi, 2015; Piedeleu and Zanasi, 2023a; Fong and Spivak, 2019), which bake in comparisons in their structure and thus can aptly represent the optimization specifications given by Galois connections.

The main contribution of this work is that we show how one can reason inductively in a graphical point-free syntax for feasibility relations. Concretely,

1. we formulate principles of induction and case-analysis for graphical point-free proofs (Lemmas 4, 5, and 6);
2. using these principles, we give equational and fully graphical derivations of the functions *predecessor* (4.1), *monus* (4.2), *min* (4.3), and *take* (4.4).

The programs we derive are simple functions over natural numbers and lists, but they are sufficient to demonstrate the viability of our approach.

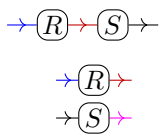
In what follows, we briefly review the required notions related to feasibility relations (Section 2) and Galois connections (Section 3), introducing diagrams as we go along, and, in Section 4, we give our main contributions. In Section 5 we review related work and in Section 6 we include concluding remarks. Finally, the appendix collects omitted proofs.

2 Feasibility Relations and Their Diagrams

As a means to give sufficient flexibility to our language of diagrams, we will work with *feasibility relations*, which are a kind of order-preserving relation. Below, we abbreviate $(x, y) \in R$ to $x\ R\ y$.

Definition 1. *A* ***feasibility relation*** $R \colon P \to Q$ *between partially ordered sets is a subset* $R \subseteq P \times Q$ *satisfying both* $x' \le x \wedge x\ R\ y \implies x'\ R\ y$ *and* $x\ R\ y \wedge y \le y' \implies x\ R\ y'$. *We represent this relation graphically as* .

Definition 2. *Feasibility relations compose in sequence and in parallel:*



$$= \{\, (x, z) \mid \exists y.\ x\ R\ y \wedge y\ S\ z \,\}$$

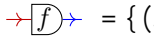
$$= \{\, ((x_1, x_2), (y_1, y_2)) \mid x_1\ R\ y_1 \wedge x_2\ S\ y_2 \,\}.$$

Implicit above is the fact that these compositions are themselves feasibility relations. For this to be the case for parallel composition, the ordering on the domain and codomain pairs is taken to be that of the product poset, where $(a, b) \le (a', b') \iff a \le a' \wedge b \le b'$.

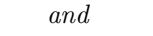
To aid in the interpretation of diagrams, we will be giving distinct colors to wires whenever their underlying types (here, partially ordered sets) are not assumed to be the same. These will become more useful later, when, in the same calculation, we will see a varied mix of types interacting. However, they are not part of the formal system and are intentionally redundant, as the typing information will always be deducible from the surrounding context.

Next, we will see the key feasibility relations that will be important for the derivations. First, monotone functions, which get a "crescent" shape.

Definition 3. *To a monotone function* $f \colon P \to Q$ *we associate two feasibility relations, namely*



$$= \{\, (x, y) \mid f(x) \le y \,\} \qquad \textit{and} \qquad$$

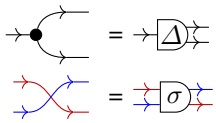


$$= \{\, (y, x) \mid y \le f(x) \,\}.$$

As a special case of the above, whenever the input poset is the unitary poset $I = \{\bullet\}$ we draw the diagram , with no incoming wires, and when the input is a product poset $P \times Q$ we draw , which has multiple, and analogously for output wires. Also, when f is the identity function, the resulting relation is exactly $x \le y$, which is drawn . This relation is an identity for sequential composition of feasibility relations, and, when the underlying poset is I, is also an

identity for parallel composition, in which case it is drawn as the empty diagram. When in the poset P^{op}, which has the ordering reversed, this same relation, now meaning $x \geq y$, will be drawn —≺— . As other notable special cases of feasibility relations of the form of Definition 3, we have the two relations below, which get special syntax.

Definition 4. *From the maps $\Delta(x) = (x, x)$ and $\sigma(x, y) = (y, x)$, with types $\Delta \colon P \to P \times P$ and $\sigma \colon Q \times U \to U \times Q$, we define feasibility relations*



$$= \{\, (x, (y, z)) \mid x \leq y \wedge x \leq z \,\}$$

$$= \{\, ((x_1, x_2), (y_1, y_2)) \mid x_1 \leq y_2 \wedge x_2 \leq y_1 \,\}.$$

The black node is called the copy *and the crossing wires the* twist.

Just like with wires, the copying node and the twist will also be colored according to the poset to facilitate reading. To complete our set of basic graphical elements, we have a few feasibility relations which do not necessarily come from functions.

Definition 5. *For all posets P, we define graphical representations for the feasibility relations below.*



$$= \{\, (\bullet, (x, y)) \mid x \leq y \,\} \colon I \to P^{\mathrm{op}} \times P, \quad$$



$$= \{\, (x, \bullet) \mid x \in P \,\} \colon P \to I,$$

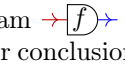


$$= \{\, ((x, y), \bullet) \mid x \leq y \,\} \colon P^{\mathrm{op}} \times P \to I, \quad$$

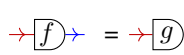


$$= \{\, (\bullet, x) \mid x \in P \,\} \colon I \to P.$$

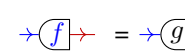
We follow analogous coloring conventions for those.

When deriving a functional algorithm, a mapping may be placed in the diagram  to unlock the flexibility of the world of relations. Then, to reflect our conclusions back into the world of functions, we need another reasoning principle. This is the role that the principle of *indirect equality* plays. To fix notation, when writing quantified formulas, we separate each of the the variables being quantified with commas and separate the variables from the proposition with a dot. We take the scope of the quantifier to extend as far as possible to the right.

Lemma 1 (Indirect Equality). *For all monotone $f, g \colon P \to Q$,*



$$\implies f = g \qquad \textit{and} \qquad$$



$$\implies f = g.$$

Explicitly, with Definition 3, those are

$$(\forall x \in P, y \in Q.\ f(x) \leq y \iff g(x) \leq y) \implies f = g, \textit{ and}$$
$$(\forall y \in Q, x \in P.\ y \leq f(x) \iff y \leq g(x)) \implies f = g.$$

Then, to make use of these relations for calculating implementations, we will need to rely on rules which dictate how they interact with each other, which we

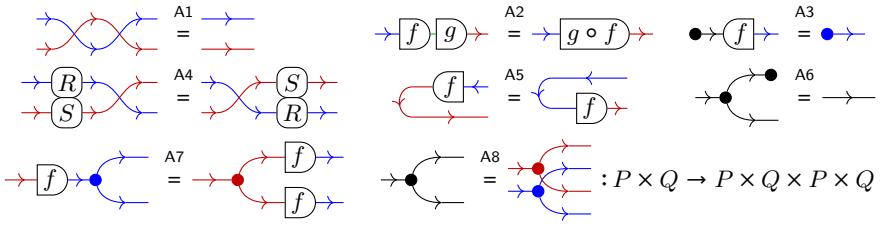


Fig. 1. Some general rules about feasibility relations

will introduce as we go along. The rules concerning general feasibility relations are listed in Figure 1.

Each equation has a purpose: (A1) untangles wires; (A2) shows that sequential composition of function diagrams agrees with the diagram for the composition of the functions, and this rule is often used implicitly when interpreting diagrams; (A3) shows that a discard can destroy any function, which is valid because functions are total; (A4) slides arbitrary relations across twists; (A5) slides a function through a bent wire; (A6) shows that the discard is a neutral element for the copy; (A7) means that functions can distribute over copies, which is valid because functions are deterministic; (A8) means that copying an element of a product poset $P \times Q$ is the same as copying each component individually.

A more thorough discussion of these rules can be found in Piedeleu and Zanasi (2023a), which also describes a variety of others that we have left out the list. We only included the rules that we will use.

To get a more concrete perspective on how to interpret diagrammatic equalities, let's take a closer look at (A7). It's an equality of feasibility relations of type $P \to Q \times Q$, as we see from the number of wires on each side. Its left-hand side is the sequential composition of  and , and its right-hand side is the sequential composition of  with the parallel composition of two  diagrams. When translated to traditional notation, it says

$$\left(\exists z{\in}Q.\quad f(x) \le z \quad \begin{matrix} z \le y_1 \\ z \le y_2 \end{matrix}\right) \iff \left(\exists w_1{\in}P, w_2{\in}P. \quad \begin{matrix} x \le w_1 & f(w_1) \le y_1 \\ x \le w_2 & f(w_2) \le y_2 \end{matrix}\right). \tag{1}$$

We have arranged the inequalities so as to mirror the diagrams, and inside each existential they are joined by implicit conjunctions. Sequential compositions turned into existential quantifiers, parallel compositions became (vertically stacked) conjuntions, and the overall equality was translated into an if-and-only-if. The soundness of this rule follows from the fact that f is monotonic.

Lemma 2. *For all $f\colon P \to Q$ and all $x \in P$, $y_1, y_2 \in Q$,*

$$\left(\exists z{\in}Q.\quad f(x) \le z \quad \begin{matrix} z \le y_1 \\ z \le y_2 \end{matrix}\right) \iff \left(\exists w_1{\in}P, w_2{\in}P. \quad \begin{matrix} x \le w_1 & f(w_1) \le y_1 \\ x \le w_2 & f(w_2) \le y_2 \end{matrix}\right).$$

Proof. Assuming the first, take $w_1 = w_2 = x$ in the second and use reflexivity for x and transitivity over z. Assuming the second, take $z = f(x)$ in the first, get $f(x) \leq f(w_1)$ and $f(x) \leq f(w_2)$ from monotonicity, and conclude with reflexivity for $f(x)$ and transitivity over $f(w_1)$ and $f(w_2)$. □

All of the other equivalences have similarly elementary proofs, though we do not provide them in this work. Next, we turn our attention to the underlying structure behind the program specifications we will consider.

3 Galois Connections

Concretely, we will seek program derivations and properties specified by *Galois connections*. Let us first review this concept.

Definition 6. *Two monotone functions* $f : P \to Q$ *and* $g : Q \to P$ *form a* ***Galois connection*** *whenever* *. That is,*

$$f(x) \leq y \iff x \leq g(y).$$

The function f *is called the* left adjoint *and* g *is called the* right adjoint*.*

The graphical version is more direct, since it avoids having to mention the auxiliary variables x and y, which helps make it more ergonomic for calculating. In the particular case where the ordering relations are equalities, the connection states that f and g are inverses: $f(x) = y \iff x = g(y)$. In the general case, a connection approximates the notion of an inverse function.

Lemma 3 (Galois Connections as Optimization Problems). *Monotone functions* $f : P \to Q$ *and* $g : Q \to P$ *form a Galois connection if and only if for all* $x \in P$ *we have that* $f(x)$ *is the smallest* $y \in Q$ *such that* $x \leq g(y)$*. Algebraically,*

$$f(x) = \mathrm{Min}\,\{\, y \in Q \mid x \leq g(y) \,\}.$$

Dually, f *and* g *form a Galois connection if and only if for all* $y \in Q$ *we have that* $g(y)$ *is the greatest amongst all values of* $x \in P$ *such that* $f(x) \leq y$*. Algebraically,*

$$g(y) = \mathrm{Max}\,\{\, x \in P \mid f(x) \leq y \,\}.$$

Proof. We will show the first equivalence, the dual one being analogous. Assume that f and g form a Galois connection. In the equivalence $f(x) \leq y \iff x \leq g(y)$, take $x = g(y)$. The right proposition becomes trivial and thus the left proposition, $f(g(y)) \leq y$, must hold, meaning that $g(y)$ is a value $x \in P$ which satisfies the restriction $f(x) \leq y$. Then, if an arbitrary $x \in P$ also satisfies it, the left-to-right implication shows that $x \leq g(y)$. Conversely, assume that $g(y)$ is the greatest amongst all values of $x \in P$ that satisfy the restriction $f(x) \leq y$. Then, $g(y)$ being greater than or equal to every value satisfying the restriction is precisely the left-to-right implication of the definition of a Galois connection, while, from the fact that $g(y)$ satisfies the restriction, we get the right-to-left implication using monotonicity and transitivity: $x \leq g(y) \implies f(x) \leq f(g(y)) \leq y \implies f(x) \leq y$. □

The left adjoint f is the *best approximation from above* to an inverse of g: the value $f(x)$ is the smallest y such that $g(y)$ is above x. Dually, the right adjoint g is the *best approximation from below* to an inverse of f.

Like regular inverses, adjoints are unique in the sense that every monotone function has at most one left adjoint and at most one right adjoint. Therefore, fixing a monotone function and asking for its left (or right) adjoint serves as an unambiguous specification. We will see a few examples in the following section.

4 Diagrammatic Program Calculation

In this section, we derive correct-by-construction algorithms for the functions *predecessor*, *monus*, *min*, and *take*. These are simple functions that operate on natural numbers and finite lists, but they are sufficient to demonstrate the core proof principles. To start, we will see the strategy of backwards reasoning and the diagrammatic version of case-analysis.

4.1 Predecessor

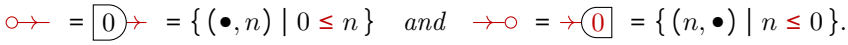
Consider the successor function on the Peano natural numbers, $S\colon \mathbb{N} \to \mathbb{N}$, given by $S(n) = n + 1$. Since zero is not the successor of any natural, S cannot have an inverse. But does it have a left adjoint? That is, can we find some $p\colon \mathbb{N} \to \mathbb{N}$ that satisfies



(G1)

Our goal is to obtain an explicit implementation for p, the natural number predecessor. First, we introduce a bit of notation to be used when studying maps on natural numbers.

Definition 7 (Graphical Zero). *We define*



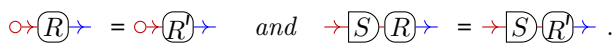
We will adopt the convention that wires of type $\mathbb{N}$ *are colored red.*

The list of graphical rules about natural numbers that we will apply is given below. Each represents a basic fact about the way the constructors zero and successor interact with each other and with the ordering on the poset of naturals.



Our construction of the predecessor function will use a graphical proof by case analysis. Here, we show how to perform case analysis on a relation whose left wire has type $\mathbb{N}$.

Lemma 4 (Diagrammatic Proof by Cases on the Naturals). *To conclude that* *it is sufficient to show*



Proof. Since R and R' are feasibility relations, the first equality implies $0\ R\ x \iff 0\ R'\ x$, and the second implies $S(n)\ R\ x \iff S(n)\ R'\ x$. Therefore, since every $k \in \mathbb{N}$ is either 0 or $S(n)$ for some n, for all $k \in \mathbb{N}$ it holds that $k\ R\ x \iff k\ R'\ x$, which means that $R = R'$. □

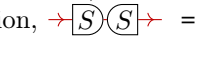
The lemma above has been formulated with case-analysis on the left wire, but this was an arbitrary choice. Case-analysis is also possible on the right and when the relation has multiple incoming wires, as we justify in Appendix A.

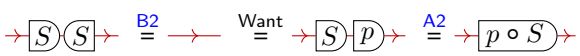
With this lemma, we may now calculate the predecessor function p. Our strategy will be to start writing the proof that p obeys specification (G1) *before* we know the definition of p. As the proof reveals sufficient conditions on p, we discover how to define p in order to satisfy the specification.

Given that we do not yet know the conditions, we cannot give a theorem statement at this point. For now, we structure our reasoning around numbered *calculations*, and, after each one is concluded, state the formal result which was discovered. We signal the end of a calculation with the symbol ⊣.

Calculation 1. We apply Lemma 4 attempting to conclude (G1) and calculate. We begin with the zero case :



The first three steps invoke rules B1 and A3. The last step, labeled Want, indicates the missing hypothesis that completes the proof. It tells us that, for the clause to be satisfied, it suffices to define $p(0) = 0$. Continuing with the second proof obligation,  , we have



To satisfy this, we need $p \circ S$ to be the identity and thus $p(S(n)) = n$. ⊣

From this calculation, we obtained two equations which allow us to compute the natural number predecessor of any number, and which could be implemented in any programming language we desire. Crucially, since these equations were obtained precisely as sufficient conditions for the Galois connection to hold, the resulting code is necessarily correct, and is so *by construction*. We may summarize the result as follows.

Theorem 1. *With* $p\colon \mathbb{N} \to \mathbb{N}$ *implemented by*

$$p(0) = 0$$
$$p(S(n)) = n,$$

we have that p *satisfies its specification, the Galois connection* (G1).

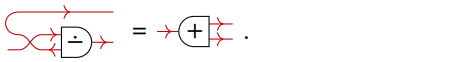
Curiously, the clause $p(0) = 0$, which might seem an unprincipled way to resolve the fact that -1 is not an allowed output, appears naturally as the correct answer required to obtain a left adjoint, as revealed during Calculation 1.

4.2 Monus

In this section we will show how to reason graphically using induction, through the example of *monus*, the subtraction function for natural numbers. It is specified as the left adjoint of addition:

$$a \dot{-} b \leq c \iff a \leq b + c. \tag{2}$$

Under the optimization perspective of Lemma 3, this specification says that $a \dot{-} b$ is the least natural number that, when added to b, is greater than or equal to a. This is technically a family of Galois connections between the maps $b+$ and $\dot{-}b$, one connection for each choice of parameter $b \in \mathbb{N}$. Rewritten graphically, it is



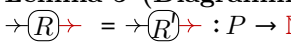
(G2)

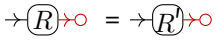
For the direction of the arrows that are exposed to the outside of the monus diagram to match up with the ones for $+$, the second input for monus has an arrow pointing inwards. This syntax represents that monus, as a monotonic function, has type $\dot{-} : \mathbb{N} \times \mathbb{N}^{\mathrm{op}} \to \mathbb{N}$. The ordering is inverted on the second parameter: if $m \geq n$, then $a \dot{-} m \leq a \dot{-} n$, like with regular subtraction.

Our goal will be to derive a recursive definition for monus satisfying (G2) given the recursive definition of addition, whose graphical equations are



We will first need to understand how to apply the technique of proof by induction diagrammatically. This is expressed in the lemma below, in which we refer to the auxiliary posets $\mathbb{N}_{\leq k} = \{\, n \in \mathbb{N} \mid n \leq k \,\}$ and give the sucessor the more specific type $S : \mathbb{N}_{\leq n} \to \mathbb{N}_{\leq n+1}$.

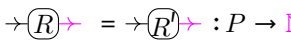
Lemma 5 (Diagrammatic Induction over the Naturals). *To conclude*  *: $P \to \mathbb{N}$, it suffices to show that the base case*



holds and that, for all $n \in \mathbb{N}$,

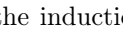
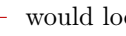


follows from the induction hypothesis



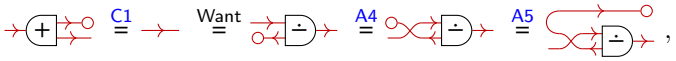
Proof. With R and R' being feasibility relations, the first clause implies they relate the same elements to 0, while the implication guarantees that if they relate the same elements to each natural up to n, they will do so for every natural up to $n + 1$. The equivalence thus follows from ordinary induction. □

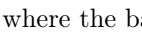
Just like with case-analysis (Lemma 4), we can perform induction on either side and on a wire which is not the sole wire on its side (see Appendix A).

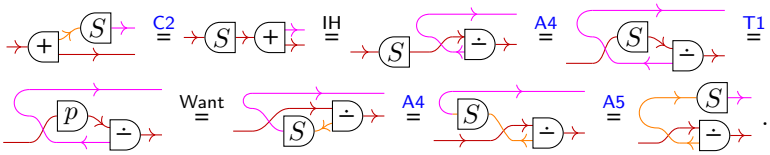
By delegating to the partially ordered set $\mathbb{N}_{\leq n}$ the obligation of tracking the size of the input n, we know when it is justified to apply the induction hypothesis. This unlocks the possibility of proof by induction in string diagrams. Note that, had we not added colors, the induction hypothesis  and the conclusion  would look exactly the same. The colors make the distiction visually apparent while relieving us of the necessity of spelling out the restricted types which appear in the hypothesis and conclusion of the induction step, which are the formal mechanism at play.

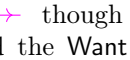
With Lemma 5, we can calculate the algorithm for monus. We begin a proof that it satisfies the Galois connection and, in the process, try to find inductive clauses that are sufficient.

Calculation 2. We attempt to conclude (G2) using induction (Lemma 5) on the top-right wire. For the base case, we compute



from which we discover that it is sufficient to take $a \dot{-} 0 = a$. For the inductive case, our goal has  where the base case had :



To follow this calculation it is easier to start at the ends and work towards the Want step in the middle. We start with C2 to commute + with S and move the + to the right. This makes the top input wire for + have type $\mathbb{N}_{\leq n}$, which allows the induction hypothesis to be used. The next two steps use A4 and Theorem 1 to further simplify the formula. Having reached the Want step it is not obvious how to proceed, so we jump to the end and work backwards. Steps A5 and A4 respectively push the  though the bend and the twist. Comparing the simplified diagrams around the Want step, we discover that to conclude the inductive case it suffices to introduce the assumption $p(a) \dot{-} b = a \dot{-} S(b)$. ⊣

From this calculation, we deduce a correct algorithm for monus.

Theorem 2. *Implementing* $\dot{-} : \mathbb{N} \times \mathbb{N}^{\mathrm{op}} \to \mathbb{N}$ *recursively by*

$$a \dot{-} 0 = a$$
$$a \dot{-} S(b) = p(a) \dot{-} b,$$

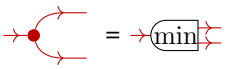
it satisfies its specification, the Galois connection (G2).

4.3 Minimum

As our next example, we derive a correct algorithm for $\min\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, which computes the minimum between two natural numbers. The result $\min(a, b)$ is specified as being the greatest lower bound of a and b. As a Galois connection,

$$x \leq a \wedge x \leq b \iff x \leq \min(a, b),$$

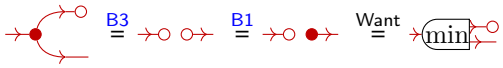
and in graphical form,



(G3)

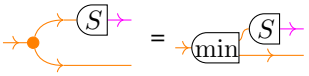
Calculation 3. We start to prove the Galois connection (G3) by induction (Lemma 5) and seek recursive clauses for min to complete the proof.

We do induction on the top right wire. For the base case, we can calculate

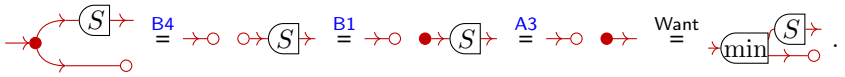


We note that  is the feasibility relation $\{(0, b) \mid b \in \mathbb{N}\}$, coming from the constant map $b \mapsto 0$; and that  comes from the map $b \mapsto \min(0, b)$. Therefore, for the last equality to hold true, we need to have $\min(0, b) = 0$.

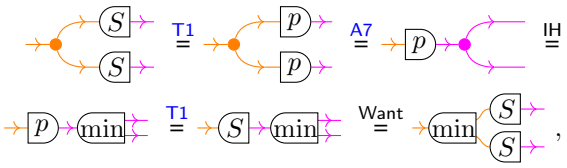
In the inductive case, we are required to show



We may do so by cases (Lemma 4) on the bottom-right wire. In the case for zero we do not rely on the induction hypothesis, and may simply work on the general type $\mathbb{N}$.



To satisfy this clause, we may take $\min(S(a), 0) = 0$. Then, for the nonzero case, we calculate



and see that it is satisfied by taking $S(\min(a, b)) = \min(S(a), S(b))$. ⊣

From the calculation above, we also deduce a correct algorithm for $\min(a, b)$.

Theorem 3. *Implementing* $\min\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *recursively by*

$$\begin{aligned} \min(0, b) &= 0 \\ \min(S(a), 0) &= 0 \\ \min(S(a), S(b)) &= S(\min(a, b)), \end{aligned}$$

it satisfies its specification, the Galois connection (G3).

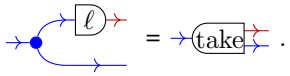
4.4 Take

As a final example, we will derive a correct implementation for the function take: $\mathbb{N} \times \mathit{List}(A) \to \mathit{List}(A)$. Informally, $\mathrm{take}(n, xs)$ selects the first n elements of a finite list xs whose entries have type A. However, this is not yet a total specification: what should take do if the input has fewer than n elements? A reasonable choice is to take as many elements as possible: we can specify $\mathrm{take}(n, xs)$ as the longest prefix of xs that has length of at most n.

To see this as a Galois connection, we can turn $\mathit{List}(A)$ into a poset by associating it to the *prefix* order, defined by $xs \sqsubseteq zs \iff (\exists ys.\ xs +\!\!+ ys = zs)$, where $+\!\!+$ is list concatenation. A longest prefix is then a maximal element in this order. Thus the specification becomes the optimization problem

$$\mathrm{take}(n, xs) = \mathrm{Max}\,\{\, ys \in \mathit{List}(A) \mid \ell(ys) \leq n \wedge ys \sqsubseteq xs \,\},$$

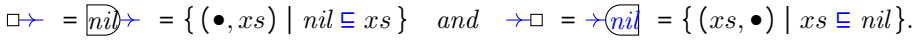
where $\ell(ys)$ denotes the length of ys. Rewriting with Lemma 3, we get the Galois connection



(G4)

To be able to work graphically with lists, as take requires, we use graphical primitives corresponding to the list constructors $\mathit{nil}\colon \mathit{List}(A)$, and $\mathit{cons}\colon A \times \mathit{List}(A) \to \mathit{List}(A)$. We abbreviate $\mathit{cons}(x, xs)$ to $x : xs$.

Definition 8 (Graphical Nil). *We define*



We will draw wires with type $\mathit{List}(A)$ in blue and wires of type A in green.

Moreover, to bring the set A into the graphical language, it must also be given a poset structure. To satisfy this formal requirement, we associate it to the discrete order, where $a \leq b$ means $a = b$.

We will also need the functions $\mathit{length}\colon \mathit{List}(A) \to \mathbb{N}$, abbreviated ℓ, and $\mathit{tail}\colon \mathit{List}(A) \to \mathit{List}(A)$, abbreviated t. The graphical equations that govern these functions and the list constructors can be found in Figure 2.

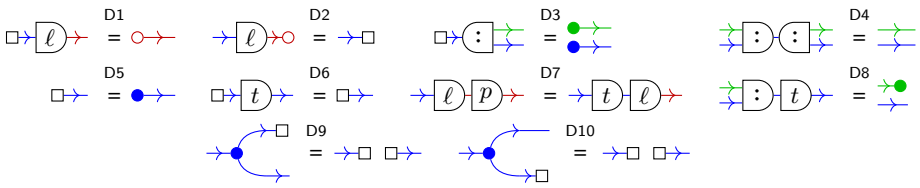
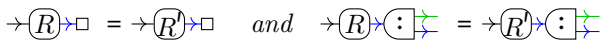


Fig. 2. Graphical properties about lists

Briefly, (D1) says that the length of an empty list is zero; (D2) says that a list with length zero must be empty; (D3) says that cons is a total function; (D4) is about the injectivity of cons; (D5) says that the empty list is a prefix of every list; (D6) says that the tail of an empty list is empty; (D7) says that the predecessor of a length is the length of the tail of a list; (D8) says that to get the tail of a cons is to discard the head; and (D9) and (D10) say that being a prefix of the empty list forces a list to be empty.

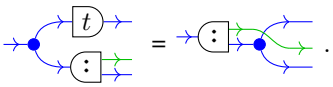
Like natural numbers, lists admit a principle of proof by cases.

Lemma 6 (Diagrammatic Proof by Cases on Lists). *To conclude that* $\to\boxed{R}\to = \to\boxed{R'}\to : P \to \mathit{List}(A)$*, it is sufficient to show*

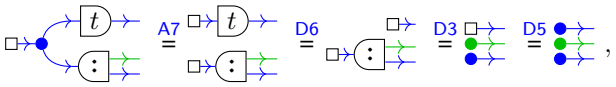


To calculate take from its Galois connection, we will make use of the following lemma. It expresses that when a copied list is destructured by cons, we are allowed to perform the destructuring first if we assume that the other copy only has its tail inspected.

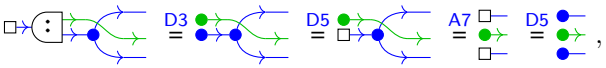
Lemma 7. *The functions tail and nil satisfy the graphical identity*



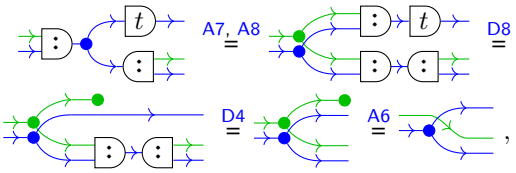
Proof. We apply proof by cases for lists (Lemma 6) on the left wire. In the nil case, we simplify the left-hand side,



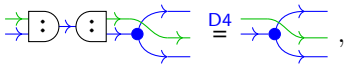
and simplify the right-hand side,



noting that the results are equal. In the cons case, we simplify the left-hand side,



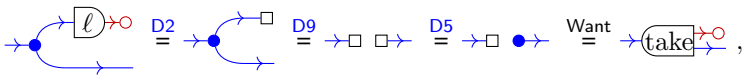
and the right-hand side,



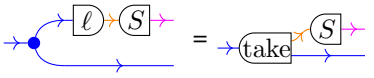
and the equality is again verified. □

Armed with this lemma, we are ready to derive take from its specification.

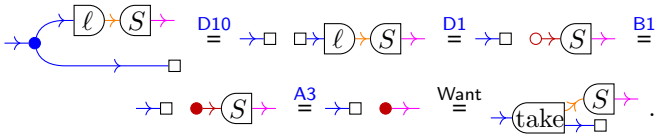
Calculation 4. We attempt to prove the Galois connection (G4) by induction on the top-right wire, applying Lemma 5. In the base case, we compute



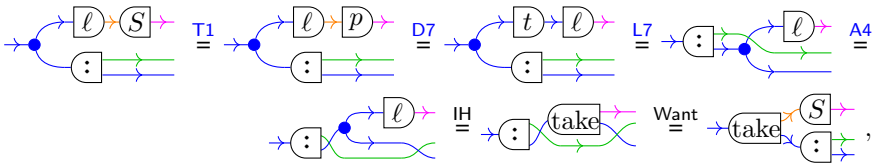
and discover that $\mathrm{take}(0, xs) = nil$ suffices. The inductive case asks for



We prove this by cases on the bottom-right list wire. If it is *nil*, we have



To satisfy it, it suffices to define $\mathrm{take}(S(n), nil) = nil$. Finally, in the inductive case with cons,



and so we define $\mathrm{take}(S(n), x : xs) = x : \mathrm{take}(n, xs)$. ⊣

Making the result explicit, we have the following correctness theorem.

Theorem 4. *With* $\mathrm{take} : \mathbb{N} \times List(A) \to List(A)$ *implemented recursively by*

$$\begin{aligned} \mathrm{take}(0, xs) &= nil \\ \mathrm{take}(S(n), nil) &= nil \\ \mathrm{take}(S(n), x : xs) &= x : \mathrm{take}(n, xs), \end{aligned}$$

it satisfies its specification, the Galois connection (G4).

This example makes clear the calculational effectiveness of the approach. As a comparison, Pereira (2023) also gives a derivation for *take*, applying relational algebra and its operators to the same Galois connection. Their calculations are equational and fully point-free, but their proof takes almost three entire pages, in stark contrast to the derivation above. The diagrammatic derivation is much more concise even if one takes into account the proof of Lemma 7.

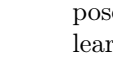
With *take*, we finish our set of examples. Through them, we have seen the diagrammatic language in action, and how to perform both case-analysis and induction over the natural numbers carried by the wires. We have kept to our promise of reasoning only through equations, and managed to obtain correct-by-construction implementations for the specifications we studied.

5 Related Work

Earlier works by Dijkstra and Feijen (1988), and Backhouse (2003), were pioneering in establishing the discipline of program calculation and showcased the basic techniques we have applied here, albeit never in diagrammatic form.

In this work, we restricted our examples to four, each specified by a Galois connection. But this was not for a lack of functions that can be specified in this way. In Pereira (2023), which inspired our choice of examples, many more are given, such as natural number division, maximum of a list, *takeWhile*, *drop*, *dropWhile*, *filter*, *zip* and *replicate*, which should be familiar to functional programmers. In Bird and de Moor (1997); Mu and Oliveira (2012), algorithms for greedy and dynamic programming are calculated from Galois connections. With our examples, we have showed that string diagrams for feasibility relations and the diagrammatic induction principle work for derivations of this kind.

To apply string diagrams to problem calculation, we gave a novel point-free formulation of induction. A different approach to the problem of point-free induction is the algebra of catamorphisms (Meijer et al., 1991), which comes with its own set of rules, but which does not mix naturally with string diagrams due to its higher-order nature. Also, the calculus considered in (Meijer et al., 1991) is lazy, whereas here we have only worked with total functions. This paper has also inspired our choice of title.

In Section 2, we saw a variety of rules about feasibility relations, and one may wonder where they come from. It is common practice to work with abstractions from category theory to be specific about the properties of a language of string diagrams, and, in this sense, there is a sophisticated way to say all of Figure 1 (and more): by assembling posets and feasibility relations into a category, one gets a compact-closed monoidal bicategory in which the category of posets is faithfully included in two canonical ways ( and). To learn more about the accompanying theory, one may consult Fong and Spivak (2019); Carboni and Walters (1987); Bonchi et al. (2025), though we have deliberately avoided this categorical perspective to keep the presentation accessible to a wider public.

Paixão et al. (2020) also use string diagrams to reason equationally, but in the setting of relational linear algebra. Their choice to use string diagrams parallels ours, however they work with linear relations (which preserve sums), whereas we work with feasibility relations (which respect poset orderings). In (Mota et al., 2025), they are used to derive algorithms in linear algebra which recurse over the block structure of matrices, which is part of the inductive structure of the diagrams in their language of study, unlike our derivations which proceed inductively over data in wires.

6 Conclusion

In this work, we have given a diagrammatic principle for point-free inductive proofs. Unlike most examples of induction in the context of string diagrams,

which recurse over the structure of the diagrams, we recurse over the structure of the data carried by the wires.

We applied our strategy to give four complete program derivations from Galois connections, working with both natural numbers and lists, obtaining in each case a correct-by-construction algorithm. The resulting graphical derivations are concise and elegant (and also very colorful).

Acknowledgments. This work was funded by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) – Project Codes 403601/2023-1 and 136149/2025-1.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

Backhouse, K., Backhouse, R.: Safety of abstract interpretations for free, via logical relations and Galois connections. Science of Computer Programming **51**(1), 153–196 (2004), ISSN 0167-6423, https://doi.org/10.1016/j.scico.2003.06.002, mathematics of Program Construction (MPC 2002)

Backhouse, R.: Galois connections and fixed point calculus. In: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction: International Summer School and Workshop Oxford, UK, April 10–14, 2000 Revised Lectures, pp. 89–150, Springer, Berlin, Heidelberg (2002), ISBN 978-3-540-47797-6, https://doi.org/10.1007/3-540-47797-7_4

Backhouse, R.: Program Construction: Calculating implementations from specifications. Wiley (2003), ISBN 9780470864548

Bird, R., de Moor, O.: Algebra of Programming. Prentice Call (1997), ISBN 013507245X

Bonchi, F., Pavlovic, D., Sobociński, P.: Functorial semantics for relational theories. Polynesian Journal of Mathematics **2**(6), 1–56 (2025)

Carboni, A., Walters, R.F.C.: Cartesian bicategories. Journal of Pure and Applied Algebra **49**(1-2), 11–32 (1987)

Censi, A.: A mathematical theory of co-design. CoRR **abs/1512.08055** (2015)

Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, p. 269–282, POPL '79, Association for Computing Machinery, New York, NY, USA (1979), ISBN 9781450373579, https://doi.org/10.1145/567752.567778, URL https://doi.org/10.1145/567752.567778, note: Fixed-point fusion is Theorem 7.1.0.4

Dijkstra, E.W., Feijen, W.H.J.: A Method of Programming. Addison-Wesley series in computer science, Addison-Wesley (1988), ISBN 9780201175363

Fong, B., Spivak, D.I.: An Invitation to Applied Category Theory: Seven sketches in compositionality. Cambridge University Press, Cambridge, UK (2019)

Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture, pp. 124–144, Springer Berlin Heidelberg, Berlin, Heidelberg (1991), ISBN 978-3-540-47599-6

Mota, J.d.A., Paixão, J.A., Martelotte, L.R.: Point-free calculational proofs and program derivation in linear algebra using a graphical syntax. Journal of Functional Programming **35**, e13 (2025), https://doi.org/10.1017/S0956796825000085

Mu, S.C., Oliveira, J.N.: Programming from Galois connections. The Journal of Logic and Algebraic Programming **81**(6), 680–704 (2012), ISSN 1567-8326, 12th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2011)

Oliveira, J.N.: Why adjunctions matter—a functional programmer perspective. In: Madeira, A., Martins, M.A. (eds.) Recent Trends in Algebraic Development Techniques, pp. 25–59, Springer Nature Switzerland, Cham (2023)

Paixão, J., , Sobociński, P.: Calculational proofs in relational graphical linear algebra. In: Formal Methods: Foundations and Applications, pp. 83–100, Springer International Publishing (2020), ISBN 978-3-030-63882-5"

Pereira, P.R.A.: Back to Programming from Galois Connections. Master's thesis, Universidade do Minho (2023)

Piedeleu, R., Zanasi, F.: A finite axiomatisation of finite-state automata using string diagrams. Logical Methods in Computer Science **19**(1), 13–38 (2023a), ISSN 1860-5974, https://doi.org/10.46298/lmcs-19(1:13)2023

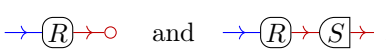
Piedeleu, R., Zanasi, F.: An introduction to string diagrams for computer scientists (2023b), https://doi.org/10.48550/arXiv.2305.08768, URL https://arxiv.org/abs/2305.08768

Selinger, P.: A Survey of Graphical Languages for Monoidal Categories, pp. 289–355. Springer, Berlin, Heidelberg (2011)

Silva, P.F., Oliveira, J.N.: 'galculator': functional prototype of a Galois-connection based proof assistant. In: Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, p. 44–55, PPDP '08, Association for Computing Machinery, New York, NY, USA (2008), ISBN 9781605581170

A Flexibilities in Case-Analysis

In Lemma 4 we presented the technique of proof by cases on a natural number with the natural number wire on the left, but this was simply a matter of presentation. An entirely symmetric argument applies when the $\mathbb{N}$ wire is on the right, and the resulting hypothesis will then involve diagrams that look like



We may also want to perform case analysis on a natural number which is not the single input (or output) to the relation, as in 

. In this case we may think of the choices of values for the other wires as a fixed parameter, giving, for each such choice, a relation of the type required by Lemma 4. Then, applying the lemma yields a family of hypothesis to be proven, which may be demonstrated all at once by once again letting the parameter vary. The resulting pattern looks like



Analogous comments apply to induction over naturals (Lemma 5) and case analysis over lists (Lemma 6).