University of Minho
School of Engineering

Paulo Ricardo Antunes Pereira

Back to Programming from Galois Connections

October 2023

University of Minho
School of Engineering

Paulo Ricardo Antunes Pereira

Back to Programming from Galois Connections

Master's Dissertation
Master in Informatics Engineering

Work carried out under the supervision of
José Nuno Oliveira (U.Minho)

October 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

I am very grateful to my supervisor and mentor José Nuno Oliveira for the opportunities he has given me, all the weekly conversations in his office and the care taken to guide me very rigorously during my journey. It is an honor for me to be his student.

I also want to extend my thanks to Paulo Silva, the developer of the Galculator proof assistant, for his generous allocation of time and practical assistance in refactoring Galculator, as well as for providing guidance on utilizing his proof assistant to automate the method analyzed in this thesis.

Finally, I express my gratitude to my beloved ones, especially my parents and grandparents for their unconditional support and encouragement.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, October 2023

Paulo Ricardo Antunes Pereira

Abstract

It is clear that the trend towards higher levels of abstraction in programming methods, as well as the effort to make software design more of a scientific, engineering discipline, has led to the development of various programming paradigms and the use of rigorous proof methods to ensure the reliability and safety of critical software systems. However, the implementation of these formal methods can be challenging due to their reliance on inductive proofs following the invent-and-verify method. Despite this, some in the field continue to seek out and use these theoretical foundations in an attempt to produce high-quality software. Therefore, this study presents the potential for the correct-by-construction method, using Galois Connections and theoretical concepts from computer science to develop a methodology for constructing practically applicable software systems whose correctness is guaranteed from the outset.

Keywords software engineering, formal methods, correct-by-construction

Resumo

É clara a tendência em direção a níveis mais elevados de abstração nos métodos de programação, bem como o esforço para tornar o *design* de software mais uma disciplina científica e de engenharia, levando ao desenvolvimento de vários paradigmas de programação e ao uso de métodos rigorosos de prova para garantir a confiabilidade e segurança de sistemas de software críticos. No entanto, a implementação desses métodos formais pode ser desafiadora devido à sua dependência de provas indutivas uma vez seguido o método de "inventar-e-verificar". Apesar disso, alguns na área continuam a procurar e a utilizar tais fundamentos teóricos numa tentativa de produzir software de alta qualidade. Assim, este estudo apresenta o potencial do método "correção-por-construção", utilizando conexões de Galois e conceitos teóricos das ciências da computação para desenvolver uma metodologia para construção de sistemas de software praticamente aplicáveis e cuja correção é garantida desde o início.

Palavras-chave engenharia de *software*, métodos formais, correção-por-construção

Contents

Part I

Introductory material

Chapter 1

Introduction

How sound has *Software Engineering* proved to be as a body of knowledge since 1968, the year regarded as its birthdate? The term was coined in a conference supported by NATO that took place in October, 1968, in Garmisch, Germany, with the purpose of suggesting that software manufacture should be based *on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering* (Naur and Randell, 1969). However, how well established are such theoretical foundations and practical disciplines in the *Software Engineering* branch?

These last 50 years have shown that only a few have taken these concerns seriously (Dijkstra, 1976; Jones, 1980; Gonthier, 2008). As a consequence, software lacks much in quality still in our days. Development costs are high, teams are too big, with inefficient methods that lead to an uncontrolled increase of complexity, all leading to bad software products. This is not engineering at all (Bogost, 2015). Still, there is hope.

1.1 Context and aims

The evolution of software engineering has been characterized by a trend towards higher levels of abstraction in programming methods. In the early days of computing, programmers would manually input machine code. Eventually, tools such as assemblers, linkers, compilers, and interpreters for high-level languages were developed to make the programming process easier. This led to the emergence of various programming paradigms, including imperative and object-oriented programming, which aimed to provide programmers with ways to more clearly express the desired behavior of a software system.

Simultaneously, there was a push to make software design more of a scientific, engineering discipline, leveraging the connection between mathematical proofs and computer programs — known as the Curry-Howard correspondence (Sørensen and Urzyczyn, 2006) — to guide the creation of large software systems, i.e., the aforementioned theoretical foundations that any branch of engineering requires. This led to the development of functional and logic programming paradigms, which advocate a more declarative style of programming, in which the programmer only specifies *what* a system should do rather than *how* it should do it (Cohen, 1995).

The reliability and safety of critical software systems, such as those used in life-critical applications, depend on the use of rigorous proof methods to guarantee the complete correctness of the software. Many researchers and engineers have been eager to find and use such theoretical foundations. However, these formal methods can be challenging to implement due to their reliance on inductive proofs. In fact, were it easy and all software producers would use these methods and produce great quality products. But this is not the case: formal methods end up having to be based on inductive proofs that hinder widespread acceptance.

Despite this, some in the field of software engineering continue to seek out and use these theoretical foundations in an effort to produce high-quality products. While the difficulties inherent in these proof methods may hinder their widespread adoption in industry, they remain essential for ensuring the safety and reliability of critical software systems. Unfortunately, most prefer to take the easy way out. As the *Verification* stage in the software construction line

Specification → *Modeling* → *Implementation* → *Verification*

tends to be quite complex, most software producers eventually replace it with a *Testing* phase. However, tests do not prove the correctness of software. Tests only prove the existence of errors, never their absence. This is because it is impossible to test every possible input and scenario that a piece of software may encounter. As a result, it is possible for a piece of software to pass all tests but still contain defects that have not been discovered. Thus, ensuring correctness always requires use of formal methods in the *Verification* phase, with the inductive proof methods already mentioned. In this approach, known as *invent & verify* (Roggenbach et al., 2022), a system is designed and implemented based on a specification or set of requirements and then verified or validated to ensure that it meets the intended specification.

But — what if one could, from the specification, immediately derive by calculation its correct implementation? Software would be ensured to be "correct inside" by mere construction, the implementation becoming a direct result of the specification. This would ensure that the software is correctly implemented according to its specification, as the process of calculation would eliminate any potential errors or deviations from the intended behavior. In other words, the software would be correct by construction because it would have been constructed according to a precise and error-free process.

This methodology — known as *correct-by-construction* (CbC) — emphasizes the importance of designing and developing systems in such a manner as to ensure their correctness from the outset (Bordis et al., 2022). It advocates the integration of verification techniques during the design and development process, rather than relying on post-deployment validation methods. This can be achieved through the use of specific formal methods that, in the end, ensure a well-established implementation fulfilling the specification. Of course, both approaches provide a high level of assurance that a system is error-free and meets its specifications since they ensure that a system is designed and implemented according to a precise, formally-specified set of requirements, and can provide a rigorous, mathematical proof of the system's correctness. This can be particularly important for safety-critical systems, such as those used in aviation, healthcare, or transportation, where the consequences of errors can be severe. However, the verification phase in the *invent & verify* approach relies, in the majority of cases, on inductive proofs, which are quite complex. Inductive proofs often involve reasoning about countably infinite sets, being challenging to understand and manipulate. Even an engineer with the expertise and a deep understanding of the underlying mathematical concepts and with a significant training and practice on proof assistants would consider these proofs quite difficult and long in real applications.

Therefore, *correct-by-construction* approaches are highly desirable in the field of software engineering. These approaches align with the methodology employed in traditional engineering disciplines, such as civil engineering, in which the integrity of a structure is established through the utilization of sound design principles, and subsequently validated through rigorous analysis, as opposed to trial and error. The discipline of software engineering seeks to apply these principles of engineering to the development of software systems. The implementation of the correct-by-construction methodology represents a significant step towards the realization of software engineering as a *bona-fide* engineering discipline. What evidence have proponents of the correct-by-construction method presented in support of its effectiveness?

It has been shown how certain recursive computations could be formally specified using Galois connections in an induction-free manner (Mu and Oliveira, 2012). Furthermore, relation algebra (Bird and de Moor, 1997) can be used to derive functionally correct implementations from such specifications, dispensing with inductive proofs. However, this technique is not easily accessible because it requires extensive knowledge of the relation calculus (Bird and de Moor, 1997) and, in particular, of a non-trivial operation called "shrinking" (Mu and Oliveira, 2012). Meanwhile, the same principle was explored in a more accessible way that does not require such technical knowledge (Silva and Oliveira, 2008). However, such experiments did not go beyond the exploratory phase.

Therefore, the main aim of this dissertation is to start from such exploratory research and work it out extensively with as many case-studies as possible to assess its practicality. Such case-study portfolio should include examples beyond classical functional programming libraries, possibly reaching dynamic programming and encompassing specification patterns that might extend the strict GC layout. The ultimate goal is to deliver a system able to support this kind of formal program derivation using the "Galculator", a proof-assistant developed in a doctoral dissertation some time ago (Silva, 2009) that is primarily based on Galois connections.

This work can be framed in the broad discipline of formal methods applied to software design, stepping up the paradigm of deriving correct-by-construction programs from logic specifications.

1.2 Structure of the dissertation

Chapter 2 begins by introducing the concept of a Galois connection with a well-known example, the problem of integer division, along with its application in computer science. The chapter further delves into the study of Galois connections, focusing on aspects most pertinent to this thesis.

Chapter 3 provides the foundational principles of the algebra of programming, pointfree calculus with functions and relations, culminating in a critical juncture: the expression of Galois connections in a pointfree manner. This is regarded as crucial for automating the strategy proposed in the thesis.

Chapter 4 introduces programming based on Galois connections, initially addressing partial orders in two significant algebras — the Peano algebra of the natural numbers and the algebra of lists. Furthermore, it expounds on the process of integrating Galois connections with predicates, as many functions in their specification involve the use of predicates and filters based on such predicates. The chapter concludes with two pivotal sections: a repertoire of functions from the Haskell Standard Prelude calculated through Galois connections and the study of functions that, despite appearing to be Galois connections, are not.

In Chapter 5, three carefully selected functions from the repertoire are re-calculated, this time at the pointfree level. This style of calculation is essential to the final chapter, which addresses the use of the Galculator proof assistant. Since the Galculator exclusively operates with relational equalities (at the pointfree level), understanding how to proceed in calculating a function from its specification as a Galois connection under relational equality is crucial. Thus, Chapter 6 explains how to launch the Galculator and how to replicate Galois-connection-based proofs in this proof assistant.

Chapter 2

Background

This chapter focuses on the state of the art related to Galois connections. Given the paramount objective of this thesis, particular attention is devoted to exploring the applications of GCs. This focus allows for the examination of the contrasting perspective inherent in GCs, specifically within the context of the *easy/hard dichotomy* of (Mu and Oliveira, 2012; Oliveira, 2023). The discussion culminates with a presentation of GCs equivalent definitions and algebraic properties.

2.1 Introducing Galois connections and its applications

In general, a Galois connection (GC) is a pair of functions f and g satisfying

$$f\,z \leqslant x \;\equiv\; z \sqsubseteq g\,x \tag{2.1}$$

for all z and x, given preorders $(\leqslant)$ and $(\sqsubseteq)$ (which can be the same). It expresses a “shunting” rule which enables one to exchange between function g in the upper side of a preorder $(\sqsubseteq)$ and a function f in the lower side of a preorder $(\leqslant)$, in a way very similar to handling (in)equations in school algebra. Functions f and g are said to be *adjoints* of each other where f is the lower adjoint and g the upper adjoint, with the (conventional) notation $f \dashv g$. But are Galois connections a novelty for the software designer?

In primary school, integer division is taught according to the following specification:

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

An algorithm for calculating the division is also taught, where one finds not just the result but also the remainder, which is as follows[1]

$$\begin{array}{r|l} x & y \\ \cline{2-2} r & z \end{array} \qquad x \div y = z * y + r$$

[1] The symbol $*$ is used to denote multiplication in order to distinguish it from $\times$, which is used to denote Cartesian products.

Thus, when questioned how to formally specify integer division, one may recur to that same algorithm and write

$$z = x \div y \;\equiv\; \langle \exists\, r : 0 \leqslant r < y : x = z * y + r \rangle \tag{2.2}$$

which is adequate. However, the remainder r is not even mentioned in the specification. This addition only results in increased complexity. In fact, the existential quantifier makes reasoning a bit difficult. So, let us try to rewrite it using only the known facts in the specification:

$$z * y \leqslant x \;\equiv\; z \leqslant x \div y \tag{2.3}$$

This is precisely (2.1) instantiated with natural number inequality preorder $(\leqslant)$, $f = (*y)$ and $g = (\div y)$.

$$z \underbrace{(*y)}_{f} \leqslant x \;\equiv\; z \leqslant x \underbrace{(\div y)}_{g}$$

But is this formal specification adequate? Split the equivalence into two implications:

- $z \leqslant x \div y \Rightarrow z * y \leqslant x$

  This one tells what is needed for z to be a candidate solution of $x \div y$ — it is required that $z * y \leqslant x$.

But there is only one solution, the largest one, which is given by the other implication:

- $z * y \leqslant x \Rightarrow z \leqslant x \div y$

  On the other hand, this one tells that, among all the candidates z (if z satisfies $z * y \leqslant x$, therefore being a candidate solution), $z \leqslant x \div y$. So, $x \div y$ is the largest of the candidates.

How good is this way of specifying integer division? Note that it is an equivalence universally quantified in all its variables, it is closest to the natural language specification, and it is tremendously generous with respect to inference of properties. Some of these arise from mere instantiation, as is the case of e.g.

$$0 \leqslant x \div y, \quad (z := 0)$$

$$y \leqslant x \;\equiv\; 1 \leqslant x \div y, \quad (z := 1)$$

Other properties, for instance

$$x \div 1 = x,$$

call for properties of the lower adjoint (multiplication). Let us check its proof:

$$z \leqslant x \div 1$$

$$
\begin{array}{ll}
\equiv & \qquad \{\ \text{GC (2.3)}\ \} \\
 & z * 1 \leqslant x \\
\equiv & \qquad \{\ 1 \text{ is the unit of multiplication}\ \} \\
 & z \leqslant x
\end{array}
$$

That is, every natural number z which is at most $x \div 1$ is also at most x and vice versa. So $x \div 1$ and x are the same. The logic behind this style of reasoning is known as the principle of *indirect equality* (Dijkstra, 2001): [2]

$$
a = b \;\equiv\; \langle \forall x :: x \leqslant a \;\equiv\; x \leqslant b \rangle \tag{2.4}
$$

Using this principle, $x \div x = 1$ (for $x > 0$) can be derived as follows:

$$
\begin{array}{ll}
 & z \leqslant x \div x \\
\equiv & \qquad \{\ \text{GC (2.3)}\ \} \\
 & z * x \leqslant x \\
\equiv & \qquad \{\ 1 \text{ is the unit of multiplication}\ \} \\
 & z * x \leqslant 1 * x \\
\equiv & \qquad \{\ \text{operation } (*x) \text{ is injective}\ \} \\
 & z \leqslant 1 \\
:: & \qquad \{\ \text{indirect equality (2.4)}\ \} \\
 & x \div x = 1 \\
\Box &
\end{array}
$$

More elaborate properties can be inferred from (2.3) together with indirect equality and basic properties of the “easy” adjoint (multiplication), for instance (for $m, d > 0$):

$$
(n \div m) \div d = n \div (d * m) \tag{2.5}
$$

Again, blending GC (2.3) with indirect equality, one delivers and elegant and easy proof:

$$
\begin{array}{ll}
 & z \leqslant (n \div m) \div d \\
\equiv & \qquad \{\ \text{GC (2.3)}\ \} \\
 & z * d \leqslant n \div m
\end{array}
$$

[2] This applies to any reflexive and antisymmetric order, which encompasses of course the partial order $(\leqslant)$ on the real numbers.

$$\begin{array}{ll}
\equiv & \quad \{ \text{ GC (2.3) } \} \\
& (z * d) * m \leqslant n \\
\equiv & \quad \{ \text{ multiplication is associative } \} \\
& z * (d * m) \leqslant n \\
\equiv & \quad \{ \text{ GC (2.3) } \} \\
& z \leqslant n \div (d * m) \\
:: & \quad \{ \text{ indirect equality (2.4) } \} \\
& (n \div m) \div d = n \div (d * m)
\end{array}$$

□

Simple (non-inductive) proofs of this kind show the calculational power of Galois connections used as specifications and operated via indirect equality. Alternative proofs would require induction, given the recursive implementation of $x \div y$, which would be a mess for more complex proofs. This strategy can be used to solve complex problems as long as solutions can be ranked using a partial order such as $(\leqslant)$.

Next, let us try to calculate the recursive implementation of $x \div y$ itself. For that, the following GC is needed,

$$a - b \leqslant c \ \equiv \ a \leqslant c + b \tag{2.6}$$

which explains subtraction over the integers (another operator used in the algorithm). This connection can be put together with the connection (2.3) restricted to non-negative integers and keeping $y \not\equiv 0$. Heading for a 2-case algorithm, one for $x \geqslant y$ and another for $x < y$, we split the reasoning as follows:

- case $x \geqslant y$

$$\begin{array}{ll}
& z \leqslant x \div y \\
\equiv & \quad \{ \text{ GC (2.3) assuming } x \geqslant 0,\, y > 0 \ \} \\
& z * y \leqslant x \\
\equiv & \quad \{ \text{ operation } (-y) \text{ is injective } \} \\
& z * y - y \leqslant x - y \\
\equiv & \quad \{ \text{ distribution law } \} \\
& (z - 1) * y \leqslant x - y \\
\equiv & \quad \{ \text{ again GC (2.3) assuming } x \geqslant y \ \}
\end{array}$$

$$
\begin{array}{ll}
 & z-1 \leqslant (x-y) \div y \\
\equiv & \qquad \{ \text{ GC (2.6) } \} \\
 & z \leqslant (x-y) \div y + 1 \\
:: & \qquad \{ \text{ indirect equality (2.4) } \} \\
 & x \div y = (x-y) \div y + 1 \\
\square &
\end{array}
$$

- otherwise ($x < y$):

$$
\begin{array}{ll}
 & z \leqslant x \div y \\
\equiv & \qquad \{ \text{ GC (2.3) and transitivity, since } x < y \text{ } \} \\
 & z * y \leqslant x \ \wedge \ z * y < y \\
\equiv & \qquad \{ \ y \neq 0 \ \} \\
 & z * y \leqslant x \ \wedge \ z \leqslant 0 \\
\equiv & \qquad \{ \ z \leqslant 0 \text{ entails } z * y \leqslant x\text{, since } 0 \leqslant x \ \} \\
 & z \leqslant 0 \\
:: & \qquad \{ \text{ indirect equality (2.4) } \} \\
 & x \div y = 0 \\
\square &
\end{array}
$$

Finally, putting these together, one obtains a correct-by-construction implementation of the integer division function, which can be written in Haskell as follows:

$$x \div y = \mathbf{if}\ x < y\ \mathbf{then}\ 0\ \mathbf{else}\ (x-y) \div y + 1$$

Clearly, the whole strategy seems worthwhile. Thus let us see another (and more complex) example.

Example with finite sequences Consider the following specification of the function *take* which yields the longest prefix of a sequence up to some given length n. For this, we shall use the prefix partial order $(\preccurlyeq)$ defined point-wise as follows [3]

$$
\begin{cases}
s \preccurlyeq [\,] \Leftrightarrow s = [\,] \\
s \preccurlyeq (h:t) \Leftrightarrow \langle \exists y \,:\, s = (h:y) \,:\, y \preccurlyeq t \rangle \vee s = [\,]
\end{cases}
\tag{2.7}
$$

[3] Subsection 4.1.2 will expound upon the implementation of the prefix ordering, as well as other orders on finite sequences.

Some warming-up examples of *take*:

```
> take 2 [1,2,3]
[1,2]
> take 10 [1,2,3]
[1,2,3]
> take 0 [1,2,3]
[]
```

For reasons that will become clear later on, we will be working with the *uncurried*[4] version of *take*. Thus, let us write the given specification "$\widehat{take}\ (n,x)$ yields the longest prefix of a sequence up to some given length n" in terms of a GC.

$$\mathsf{length}\ z \leqslant n \ \wedge\ z \preccurlyeq x \ \equiv\ z \preccurlyeq \widehat{take}\ (n,x) \tag{2.8}$$

First of all, can we identify the lower adjoint? Let us rewrite (2.8) in the following way to make it easier to identify the orders and the adjoints f and g:

$$\underbrace{\langle \mathsf{length}, id\rangle}_{f}\ z \ \trianglelefteq\ (n,x) \ \equiv\ z \preccurlyeq \underbrace{\widehat{take}}_{g}\ (n,x)$$

where[5]

$$\langle f,g\rangle\ x = (f\ x, g\ x)$$

$$(\trianglelefteq) = (\leqslant) \times (\preccurlyeq)$$

Again one asks: is this specification adequate? Let us work it out by using (2.8) and, again, splitting the equivalence into two implications:

- $\mathsf{length}\ z \leqslant n \ \wedge\ z \preccurlyeq x \Leftarrow z \preccurlyeq \widehat{take}\ (n,x)$

  This means that z is a candidate solution for $\widehat{take}\ (n,x)$.

But there is only one solution, the largest one:

- $\mathsf{length}\ z \leqslant n \ \wedge\ z \preccurlyeq x \Rightarrow z \preccurlyeq \widehat{take}\ (n,x)$

  Among all candidates z, $z \preccurlyeq \widehat{take}\ (n,x)$ holds. So, $\widehat{take}\ (n,x)$ is the largest one.

Now, let us again blend this GC with indirect equality (on partial order $(\preccurlyeq)$) to prove these expected properties:

[4] Uncurried functions receive arguments wrapped within a pair, instead of receiving them sequentially, and a special notation will be used: $uncurry\, f$ will be denoted by $\widehat{f}$. Section 3.1 will delve deeper into this concept.

[5] The product of two partial orders is a partial order too (Backhouse, 2004). As it will be seen later on, the product ("tensor") of two relations is defined by $(c,d)\ (R \times S)\ (a,b)\ \equiv\ c\, R\, a \wedge d\, S\, b$.

1. $\widehat{take}\ ((\mathsf{length}\ x),x) = x$

   Proof:

$$z \preccurlyeq \widehat{take}\ ((\mathsf{length}\ x),x)$$

$$\equiv \qquad \{\ \text{GC (2.8)}\ \}$$

$$\mathsf{length}\ z \leqslant \mathsf{length}\ x \wedge z \preccurlyeq x$$

$$\equiv \qquad \{\ \mathsf{length}\ z \leqslant \mathsf{length}\ \Leftarrow z \preccurlyeq x\ \}$$

$$z \preccurlyeq x$$

$$:: \qquad \{\ \text{indirect equality over list prefixing } (\preccurlyeq)\ \}$$

$$\widehat{take}\ ((\mathsf{length}\ x),x) = x$$

$$\Box$$

2. $\widehat{take}\ (0,x) = [\,]$

   Proof:

$$z \preccurlyeq \widehat{take}\ (0,x)$$

$$\equiv \qquad \{\ \text{GC (2.8)}\ \}$$

$$\mathsf{length}\ z \leqslant 0 \wedge z \preccurlyeq x$$

$$\equiv \qquad \{\ \mathsf{length}\ z \leqslant 0\ \equiv\ z = [\,]\ \}$$

$$z = [\,] \wedge z \preccurlyeq x$$

$$\equiv \qquad \{\ z \preccurlyeq x \Leftarrow z = [\,]\ \}$$

$$z = [\,]$$

$$\equiv \qquad \{\ (\preccurlyeq) \text{ definition (4.18)}\ \}$$

$$z \preccurlyeq [\,]$$

$$:: \qquad \{\ \text{indirect equality over list prefixing } (\preccurlyeq)\ \}$$

$$\widehat{take}\ (0,xs) = [\,]$$

$$\Box$$

3. $\widehat{take}\ (n,[\,]) = [\,]$

   Proof:

$$z \preccurlyeq \widehat{take}\ (n,[\,])$$

$$
\begin{array}{ll}
\equiv & \{\ \text{GC (2.8)}\ \} \\
& \mathsf{length}\ [\,] \leqslant n \wedge z \preccurlyeq [\,] \\
\equiv & \{\ \mathsf{length}\ [\,] = 0 \leqslant n\ \} \\
& z \preccurlyeq [\,] \\
:: & \{\ \text{indirect equality over list prefixing } (\preccurlyeq)\ \} \\
& \widehat{take}\ (n, [\,]) = [\,] \\
\Box &
\end{array}
$$

Note how such elegant proofs expose some expected properties before the implementation of *take* itself. Indeed, an advantage of this kind of formal specification is precisely questing the specification for properties of the design before the implementation phase.

By the way, note that the already inferred

$$
\begin{array}{l}
\widehat{take}\ (0, x) = [\,] \\
\widehat{take}\ (n, [\,]) = [\,]
\end{array}
$$

can be regarded as base cases of a possible implementation. As such, by pattern matching, only the following case remains to be addressed:

$$\widehat{take}\ (n+1, x : xs)$$

Can this be inferred from (2.8) too? Let us unfold $z \preccurlyeq \widehat{take}\ (n+1, x : xs)$ and see what happens.

$$
\begin{array}{ll}
& z \preccurlyeq \widehat{take}\ (n+1, x : xs) \\
\equiv & \{\ \text{GC (2.8); prefix definition (2.7)}\ \} \\
& \mathsf{length}\ z \leqslant n+1 \wedge (\langle \exists\, b : z = (x : b) : b \preccurlyeq xs \rangle \vee z = [\,]) \\
\equiv & \{\ \text{distribution; } \mathsf{length}\ z \leqslant n+1 \Leftarrow z = [\,]\ \} \\
& \langle \exists\, b : z = (x : b) : \mathsf{length}\ z \leqslant n+1 \wedge b \preccurlyeq xs \rangle \vee z = [\,] \\
\equiv & \{\ \mathsf{length}\ (x : t) = 1 + \mathsf{length}\ t\ \} \\
& \langle \exists\, b : z = (x : b) : \mathsf{length}\ b \leqslant n \wedge b \preccurlyeq xs \rangle \vee z = [\,] \\
\equiv & \{\ \text{GC (2.8)}\ \} \\
& \langle \exists\, b : z = (x : b) : b \preccurlyeq \widehat{take}\ (n, xs) \rangle \vee z = [\,] \\
\equiv & \{\ \text{prefix definition (2.7)}\ \}
\end{array}
$$

$$z \preccurlyeq x : \widehat{take}\ (n, xs)$$

$$:: \qquad \{\ \text{indirect equality over list prefixing } (\preccurlyeq)\ \}$$

$$\widehat{take}\ (n+1, x : xs) = x : \widehat{take}\ (n, xs)$$

$\square$

Altogether, the implementation of $\widehat{take}$ has been derived by calculation in a correct-by-construction way:

$$\begin{array}{l} \widehat{take}\ (0, \_) = [\,] \\ \widehat{take}\ (\_, [\,]) = [\,] \\ \widehat{take}\ (n+1, h : xs) = h : \widehat{take}\ (n, xs) \end{array}$$

Clearly, verifying this implementation is not needed, as it was calculated from its formal specification — recall the CbC design principle.

These examples provide two clear illustrations of the Formal Methods “golden triad”:

- specification (via GC in this case) — **what** the program should do;
- implementation — **how** the program does it;
- justification — **why** the program does it (correct-by-construction in this case).

However, some ingredients of the calculations, such as the use of products in the previous example ($take$), have been used before being duly defined. Section 2.3 will provide such explicit definitions in detail. It will also provide a deeper understanding of the GC concept by presenting alternative definitions. The importance of being able to define a concept in multiple, equivalent ways is emphasized, as it helps to recognize it in other contexts and adds to the understanding of what it means for one function to be the adjoint of another (Backhouse, 2004). Let us first introduce the concept of a GC from the perspective of the *easy/hard dichotomy*.

2.2 The antithetical perspective of Galois connections

Dichotomies are frequently observed in various aspects of daily life, wherein pairs of opposing concepts, such as *good/bad*, *action/reaction*, *left/right*, *lower/upper*, *easy/hard*, commonly emerge. Each element within these pairs finds its definition and significance in relation to its opposite, thereby embodying an antithetical nature. Despite the inherent circularity involved, everyday language persistently employs and sustains such dualities. As proposed by Oliveira (2023), Galois connections capture such dualities in an effective and calculational way, as explained next.

Perfect antithesis The concept of a perfect antithesis, characterized by opposition or inversion, is embodied by the notion of a bijection or isomorphism. This involves two functions f and g that satisfy the following properties:

$$\begin{tikzcd} B \arrow[r, bend left, "g"] \arrow[r, phantom, "\cong"] & A \arrow[l, bend left, "f"] \end{tikzcd} \qquad \left\{ \begin{array}{l} f\ (g\ b) = b \\ g\ (f\ a) = a \end{array} \right. \tag{2.9}$$

That is, both are *lossless* transformations. For example, multiplication and division are inverses of each other in the real numbers. That is, undertaking these operations does not result in any loss of information — for $g := (/y)$ and $f := (*y)$, one gets

$$\begin{tikzcd} \mathbb{R} \arrow[r, bend left, "(/y)"] \arrow[r, phantom, "\cong"] & \mathbb{R} \arrow[l, bend left, "(*y)"] \end{tikzcd} \qquad \left\{ \begin{array}{l} (b\,/\,y) * y = b \\ (a * y)\,/\,y = a \end{array} \right.$$

Imperfect antithesis In practice, data transformations often result in *loss* of information, e.g.

$$\begin{tikzcd} \text{PNG} \arrow[r, bend left, "png2pdf"] \arrow[r, phantom, "\not\cong"] & \text{PDF} \arrow[l, bend left, "pdf2png"] \end{tikzcd} \qquad \left\{ \begin{array}{l} png2pdf \cdot pdf2png \neq id \\ pdf2png \cdot png2pdf \neq id \end{array} \right.$$

although our eyes may not spot the difference in most cases. These imperfect inversions result in a loss of information which does not fit in equation (2.9). However, it may be the case that one can write

$$\left\{ \begin{array}{l} f\ (g\ b)\ \leqslant\ b \\ g\ (f\ a)\ \sqsubseteq\ a \end{array} \right. \tag{2.10}$$

telling "how bad" each inversion is, by relying on two preorders $(\leqslant)$ and $(\sqsubseteq)$ that capture *under* and *over approximations*:

$$\begin{tikzcd} (\leqslant) \arrow[r, bend left, "g"] & (\sqsubseteq) \arrow[l, bend left, "f"] \end{tikzcd} \tag{2.11}$$

(Functions f and g are assumed monotonic above.)

How Galois connections arise Let us now handle these approximations by analyzing the following diagram:

$$\begin{tikzcd} g\ x & f\ (g\ x) \arrow[r, "(\leqslant)"] & x & & g\ x \\ a \arrow[u, "(\sqsubseteq)"] & f\ a \arrow[u, "(\leqslant)"] \arrow[ur, "(\leqslant)"'] & & g\ (f\ a) \arrow[ur, "(\sqsubseteq)"] & a \arrow[u, "(\sqsubseteq)"'] \arrow[l, "(\sqsubseteq)"] \end{tikzcd}$$

where $x \xrightarrow{(\leqslant)} y$ and $x \xrightarrow{(\sqsubseteq)} y$ denote $x \leqslant y$ and $x \sqsubseteq y$ respectively. Starting with $a \xrightarrow{(\sqsubseteq)} g\,x$, which means that $a \sqsubseteq g\,x$,

1. since f is monotonic, one obtains $f\;a \leqslant f\;(g\,x)$;
2. from (2.10), one obtains $f\;(g\,x) \leqslant x$;
3. by transitivity ("arrow composition"), one obtains $f\;a \leqslant x$;
4. since g is monotonic, one obtains $g\;(f\;a) \leqslant g\,x$;
5. again, from (2.10) and transitivity, one obtains $a \sqsubseteq g\,x$.

Thus,

$$a \sqsubseteq g\,x \Rightarrow f\;a \leqslant x \Rightarrow a \sqsubseteq g\,x$$

which leads, by circular implication, to the following equivalence

$$f\;a \leqslant x \Leftrightarrow a \sqsubseteq g\,x \tag{2.12}$$

that is, Galois connection (2.1). Thus, one may also use the notation in (2.11) in order to render GCs. Back to the integer division algorithm, it is the experience of every child that $x * y$ is much simpler to calculate than $x \div y$. Nevertheless, an intriguing observation arises wherein the perceived complexity of the operation $x \div y$ can be effectively explained by the inherent simplicity of the operation $x * y$. In fact, $(*y) \dashv (\div y)$ conveys the underlying message:

> *hard* $(\div y)$ is explained by *easy* $(*y)$.

Exploring program specifications as GCs within the framework of this easy/hard dichotomy is at the very core of this thesis, so as to calculate "hard" adjoints from easy ones. For instance, extending this dichotomy to the aforementioned function $\widehat{take}$, one has

$$\underbrace{\text{length}\;z \leqslant n \wedge z \preceq x}_{\textit{easy}} \Leftrightarrow \underbrace{z \preceq \widehat{take}\;(n,x)}_{\textit{hard}}$$

2.3 Algebraic properties and equivalent definitions

One of the main advantages of this rich theory is that once a concept is identified as an adjoint of a Galois connection, all generic properties are inherited, even when the other adjoint is not known. Let us brief these key properties which relate GCs to the underlying ordered structures.

- "Shunting rule" $-f$ "shunts to the other side" and becomes g, and vice-versa:

$$f\ a \sqsubseteq_B b \Leftrightarrow a \sqsubseteq_A g\ b \tag{2.13}$$

- Upper adjoint distributes over meet (wherever this exists):

$$g\ (b \sqcap_B b') = g\ b \sqcap_B g\ b' \tag{2.14}$$

- Lower adjoint distributes over join (wherever this exists):

$$f\ (a \sqcup_A a') = f\ a \sqcup_A f\ a' \tag{2.15}$$

- Cancellation of the lower adjoint:

$$a \sqsubseteq_A g\ (f\ a) \tag{2.16}$$

- Cancellation of the upper adjoint:

$$f\ (g\ b) \sqsubseteq_B b \tag{2.17}$$

- Lower adjoint is monotonic:

$$a \sqsubseteq_A a' \Rightarrow f\ a \sqsubseteq_B f\ a' \tag{2.18}$$

- Upper adjoint is monotonic:

$$b \sqsubseteq_B b' \Rightarrow g\ b \sqsubseteq_A g\ b' \tag{2.19}$$

- Upper adjoints preserve top elements:

$$g\ \top_B = \top_A \tag{2.20}$$

- Lower adjoints preserve bottom-elements:

$$f\ \bot_A = \bot_B \tag{2.21}$$

- for partial orders, the so-called semi-inverse properties:

$$f = f \cdot g \cdot f \tag{2.22}$$

$$g = g \cdot f \cdot g \tag{2.23}$$

Besides, a most useful ingredient of Galois connections lies in the fact that they build up on top of themselves thanks to a number of combinators which enable one to construct (on the fly) new connections out of existing ones (Oliveira, 2020b). A fundamental result that Galois-connected functions can be combined to form new GCs is given via functional composition. Given two GCs as follows

$$\begin{tikzcd}
(\leqslant) \arrow[r, bend left, "g"] & (\sqsubseteq) \arrow[l, bend left, "f"] \arrow[r, bend left, "k"] & (\preccurlyeq) \arrow[l, bend left, "h"]
\end{tikzcd}$$

a new GC arises:

$$\begin{tikzcd}
(\leqslant) \arrow[r, bend left, "k \cdot g"] & (\preccurlyeq) \arrow[l, bend left, "f \cdot h"]
\end{tikzcd}$$

The equivalence is quite straightforward, cf.

$$\begin{array}{ll}
 & (f \cdot h)\, x \leqslant z \\
\Leftrightarrow & \quad \{\ \text{composition};\ f \dashv g\ \} \\
 & h\, x \sqsubseteq g\, z \\
\Leftrightarrow & \quad \{\ h \dashv k;\ \text{composition}\ \} \\
 & x \preccurlyeq (k \cdot g)\, z \\
\Box &
\end{array}$$

A second major result is that every relator[6] that distributes through binary intersections preserves GCs (Backhouse and Backhouse, 2004). This means that, for every such relator R and given a GC

$$\begin{tikzcd}
(\leqslant) \arrow[r, bend left, "g"] & (\sqsubseteq) \arrow[l, bend left, "f"]
\end{tikzcd}$$

a new GC arises:

$$\begin{tikzcd}
\mathsf{R}\,(\leqslant) \arrow[r, bend left, "\mathsf{R}\, g"] & \mathsf{R}\,(\sqsubseteq) \arrow[l, bend left, "\mathsf{R} f"]
\end{tikzcd}$$

[6] See Section 3.2 which expounds the notion of a relator within the relation algebra framework.

Finally, Backhouse (2004) presents some alternative definitions in order to easily identify (f, g) as a GC between posets (partially ordered sets)[7] $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$. Among them, the following two are noteworthy:

Alternative definition 1

- f and g are both monotonic;
- for all $x \in B$ and $y \in A$, $x \sqsubseteq_B g\ (f\ x)$ and $f\ (g\ y) \sqsubseteq_A y$

Alternative definition 2

- g is monotonic;
- for all $x \in B$, $x \sqsubseteq_B g\ (f\ x)$;
- for all $x \in B$ and $y \in A$, $x \sqsubseteq_B g\ y \Rightarrow f\ x \sqsubseteq_A y$

2.4 Summary

The calculational power of Galois connections has been shown to be effective in function specification and in deriving them by calculation using the indirect equality method, dispensing with explicit inductive proofs. This same method allows for the derivation of properties of the adjoint functions prior to their implementation — an excellent and highly desired application of GCs.

Having demonstrated its practicality and usefulness, important algebraic properties as well as equivalent definitions of GCs have also been revealed, which will certainly facilitate the specification of further functions in terms of GCs.

[7] A poset is a pair $(P, (\leqslant))$, where P is a set and $(\leqslant)$ is partial order on P.

Chapter 3

Going pointfree

Oliveira (2020b) conducts a comprehensive examination of GCs within the framework of relation algebra. In this chapter, the mathematical concept of GC is presented from this perspective. Furthermore, this chapter will provide us with the necessary means to formally specify programs through GCs and carry out precise calculations to derive their implementations. To fulfill that purpose, the chapter begins with a brief overview of the pointfree relational calculus and concludes with the formulation of GCs in the form of relational equalities. The use of pointfree notation is essential to provide a rigorous justification for the developed theory and for proof-support purposes. In fact, by exploring the domain of calculating with functions and relations, we shall unlock the potential for precise program specification and its correct-by-construction implementation.

It should be noted that some laws are not introduced in this chapter for the purpose of streamlining the content. However, all the laws and properties used are presented in Appendix B.

3.1 Calculating with functions

The concept of a function is well-known to anyone with basic school education. The understanding of functions permeates mathematics, as it is grounded in the established framework of sets and set-theoretical functions. Functional programming, which is basically programming with functions, extends beyond the act of writing code for computers. It embodies the notion that different branches of programming have the potential to adopt a functional structure or expression, while also encompassing the concept of transforming abstract and inefficient programs into efficient ones through calculation.

Functions and types In Functional Programming (FP), functions are the most important objects. They act like a “black box” that produces a specific result from a given input. Functions will be denoted by lowercase letters. We write $f : A \rightarrow B$ to indicate that f receives values of type A and produces values of type B. This is usually referred as the function signature or type.

How is the output of a function produced? The answer is given when inspecting the inside of the black box — that is where the calculation rule of the function is to be found. This calculation rule, also called the function's behavior, is specified directly at an arbitrary point in the domain (also called variable). For example, in the definition $f\ x = x + 1$, it is directly specified on an arbitrary x in the domain of f that the behavior of f on that x is to add 1 to it. It is precisely this way of defining functions that is studied in high school.

By contrast, pointfree definitions are characterized by the absence of variables, being built instead through the combination of simpler functions using a limited set of combinators. The choice of these combinators is dictated by the power of the laws associated with them. Some of these combinators are analyzed below.

Identity and constant functions Identity functions are those that merely copy the input to the output: $f\ a = a$, for $f : A \to A$ and $a \in A$. In this case, f is said to be the identity function on A. As expected, every type X has its own identity function id_X. However, subscripts will be omitted whenever they are implicit in the context. Thus, a "single" identity function can be assumed.

Unlike the identity function, which does not lose any information, constant functions lose all (or nearly all) information. Regardless of the input data, the output is always the same value. The notation to be used will be underlining. Therefore, let C be a non-empty datatype and $c \in C$. The *everywhere c* function, for an arbitrary type A, is defined as $\underline{c}_A\ = c$ whose signature is $A \to C$. Similar to what occurs with identity functions, subscripts will be omitted whenever they are implicit in the context.

Functional composition A cornerstone of FP is the functional composition combinator, which "chains" two functions in the following way:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[rr, bend right, "g \cdot f"'] & B \arrow[r, "g"] & C \end{tikzcd}$$

In mathematics, one usually says that the outputs of function f must be contained within the domain of function g. However, in computer science the rule is simpler, yet more restrictive: the output type of f must match the input type of g. Nonetheless, the notation employed is the same: $g \cdot f$, which can be read as "g composed with f" or "g after f", and is precisely defined as $(g \cdot f)\ x = g\ (f\ x)$.

Products As expected, not all functions can be combined via functional composition. For example, functions $f : A \to B$ and $g : A \to C$ are a case where functional composition cannot be applied. However, since both functions share the same input type, they can be combined via the binary operator $\langle \_, \_ \rangle$ called *split*. For functions f and g, one has $\langle f,g\rangle : A \rightarrow B \times C$. This operator satisfies

$$h = \langle f,g\rangle \Leftrightarrow \begin{cases} \pi_1 \cdot h = f \\ \pi_2 \cdot h = g \end{cases} \tag{3.1}$$

for all $h : A \rightarrow B \times C$ and given projections $\pi_1 : A \times B \rightarrow A$ and $\pi_2 : A \times B \rightarrow B$.
Set $B \times C$ is given by the Cartesian product of B with C, i.e.,

$$B \times C = \{(b,c) \mid b \in B \wedge c \in C\}$$

and the split combinator is defined by

$$\langle f,g\rangle\, a = (f\, a, g\, a) \tag{3.2}$$

The "tensor product" operator × can also be defined on functions via the split combinator:

$$f \times g = \langle f \cdot \pi_1, g \cdot \pi_2\rangle \tag{3.3}$$

IT preserves composition and identity:

$$(f \times g) \cdot (h \times k) = (f \cdot h) \times (g \cdot k) \tag{3.4}$$

$$id_A \times id_B = id_{A \times B} \tag{3.5}$$

Coproducts As previously explained, the functional combinator *split* was created with the aim of combining functions that do not meet the requirements of functional composition but share the same domain. The "dual" situation corresponds to functions sharing the codomain instead. Thus, functions $f : A \rightarrow C$ and $g : B \rightarrow C$ can be combined via the binary operator $[\_,\_]$ called *either* or *join*. One has $[f,g] : A + B \rightarrow C$, where $A + B$ is the *disjoint union*,

$$A + B \overset{\text{def}}{=} \{i_1\, a \mid a \in A\} \cup \{i_2\, b \mid b \in B\}$$

assuming the "tagging" functions i_1 and i_2, whose signatures are $A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$. These functions associate different tags in order to ensure that values of type A and values of type B do not mix in the set $A + B$. They are called injections of the disjoint union — i_1 is said to "inject" values to the left, while i_2 "injects" values to the right. In words, "either we are on the left side and execute f or we are in the right side and execute g."
Therefore, the *either* combinator is defined by

$$[f,g]\, x \overset{\text{def}}{=} \begin{cases} x = i_1\, a \Rightarrow f\, a \\ x = i_2\, b \Rightarrow g\, b \end{cases} \tag{3.6}$$

and satisfies

$$k = [f, g] \Leftrightarrow \begin{cases} k \cdot i_1 = f \\ k \cdot i_2 = g \end{cases} \tag{3.7}$$

for all $k : A + B \to C$.

Similarly to $\times$, the operator $+$ can also be defined on functions, now via the either combinator:

$$f + g = [i_1 \cdot f, i_2 \cdot g] \tag{3.8}$$

Again, it preserves composition and identity:

$$(f + g) \cdot (h + k) = (f \cdot h) + (g \cdot k) \tag{3.9}$$

$$id_A + id_B = id_{A+B} \tag{3.10}$$

McCarthy's conditional Most functional programming languages provide pointwise conditional expressions in the form

$$\mathbf{if}\ p\ x\ \mathbf{then}\ f\ x\ \mathbf{else}\ g\ x \tag{3.11}$$

which translates into the following process: given a predicate $p : A \to \mathbb{B}$ and two functions $f, g : A \to B$, if $p\ x$ holds then the process results in $f\ x$, otherwise in $g\ x$, i.e.,

$$\begin{cases} p\ x \Rightarrow f\ x \\ \neg\ (p\ x) \Rightarrow g\ x \end{cases}$$

In order to rewrite the expression (3.11) in a pointfree style, let us start by assuming that $p\ x$ has already been calculated. In this case, either f is executed or g is executed. This latter operation is clearly the combinator $[f, g]$. So, the goal is, if $p\ x$ holds, the argument is injected on the left side, so that the operation $f\ x$ is executed; otherwise, the argument is injected on the right side and $g\ x$ is executed instead of $f\ x$. This injection choice is performed by the operation $p?$, called *the guard* associated with the predicate p, defined by

$$p?\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ i_1\ x\ \mathbf{else}\ i_2\ x$$

Thus, the expression (3.11), rewritten in a pointfree style, is given by $[f, g] \cdot p?$. Note that $p?$ is much more informative than the predicate itself, as it already gives us the result of testing p on a given input. All this leads to the well-known functional combinator "McCarthy's conditional" that is usually denoted by the expression $p \to f, g$. Therefore,

$$p \to f, g \stackrel{\text{def}}{=} [f, g] \cdot p? \tag{3.12}$$

The use of the *either* combinator suggests that, when reasoning about conditionals, one can turn to the algebra of coproducts as a potential aid. This results in the following properties:

$$h \cdot (p \to f, g) = p \to h \cdot f, h \cdot g \tag{3.13}$$

$$(p \to f, g) \cdot h = (p \cdot h) \to f \cdot h, g \cdot h \tag{3.14}$$

$$p? \cdot f = (f + f) \cdot (p \cdot f)? \tag{3.15}$$

namely, the fusion laws and the natural property of the guard, respectively.

Exponentials Given a function $f : C \times A \to B$, one intends to construct a family of functions of the type $A \to B$ according to the following approach: for each $c \in C$, the function

$$\begin{aligned} & f_c : A \to B \\ & f_c\ a \overset{\text{def}}{=} f\ (c, a) \end{aligned} \tag{3.16}$$

is constructed. In other words, the construction of this family is a function of the type $C \to (A \to B)$, meaning that given a $c \in C$, it produces a function of the type $A \to B$. Functions of this kind are called *higher-order functions* — functions that not only produce functions but also receive functions as arguments. To represent the type $A \to B$ (or $B \leftarrow A$), we shall use the notation B^A. Thus,

$$B^A \overset{\text{def}}{=} \{g \mid g : A \to B\} \tag{3.17}$$

corresponds to the type inhabited by functions from A to B. This means that the functional declaration $g : A \to B$ is equivalent to $g \in B^A$.

Since the purpose of functions is to be applied to arguments, the introduction of the *apply* combinator is quite intuitive:

$$\begin{aligned} & ap : B^A \times A \to B \\ & ap\ (f, a) \overset{\text{def}}{=} f\ a \end{aligned} \tag{3.18}$$

Now, going back to the function $f : C \times A \to B$, let's recall the strategy of producing a function $f_c \in B^A$ for each $c \in C$. As mentioned before, this process corresponds to a function of the type $C \to B^A$, which expresses f as a family of functions of the type $A \to B$ indexed by the type C. Such functions will be referred to as transposes, and the notation $\overline{f}$ will be used to represent them, read as "curry of f". As expected, f and $\overline{f}$ are mutually related by the following property:

$$f\ (c, a) = (\overline{f}\ c)\ a \tag{3.19}$$

However, despite the equality, $\overline{f}$ has the advantage of being more "tolerant" than f. While f requires both arguments under the pair (c,a), $\overline{f}$ is satisfied with receiving the argument c first and, later, if the process allows, the argument a.

Just like the product $A \times B$ and the coproduct $A+B$, the exponential B^A also has a universal property:

$$k = \overline{f} \Leftrightarrow f = ap \cdot (k \times id) \tag{3.20}$$

Finally, a new functional combinator arises from $\overline{f \cdot ap}$ whose signature is $B^A \rightarrow C^A$. The notation to express this functional combinator $\overline{f \cdot ap}$ will be f^A, which follows

$$\frac{C \xleftarrow{\;f\;} B}{C^A \xleftarrow{\;f^A\;} B^A}$$

But what does this new combinator mean? Well, f^A takes a function $g : A \rightarrow B$ as an argument and returns a function of type $A \rightarrow C$. Therefore, given a particular $a \in A$, $(f^A\ g)\ a$ will produce a C-value. It is known that the function f produces values of type C when given a B-value. It so happens that g produces values of type B. Thus, $g\ a$ is executed, resulting in a value $b \in B$, which is passed to the function f to produce a value $c \in C$. What is happening here is precisely the functional composition of f with g, that is, f^A translates to the combinator "composition with f":

$$f^A\ g \overset{\text{def}}{=} f \cdot g \tag{3.21}$$

i.e.,

$$f^A \overset{\text{def}}{=} (f \cdot) \tag{3.22}$$

Back to property (3.19), the chosen notation allows us to express the equality $f\ (a,b) = (\overline{f}\ a)\ b$ using the isomorphism,

$$C \times A \rightarrow B \;\cong\; C \rightarrow B^A$$

which can be rewritten as:

$$B^{C \times A} \;\cong\; (B^A)^C \tag{3.23}$$

Isomorphism (3.23) is at the core of functional programming. In Haskell, the pre-defined functions that witness it are:

$$\begin{tikzcd} B^{C \times A} \arrow[r, bend left, "curry"] \arrow[r, phantom, "\cong"] & (B^A)^C \arrow[l, bend left, "uncurry"] \end{tikzcd} \tag{3.24}$$

This means that $curry$ corresponds to the transposition in Haskell. Besides, to simplify algebraic notation, the inverse of transposition will also have its own notation: $uncurry\, f$ will be abbreviated by $\widehat{f}$.

Functors A so-called functor F can be regarded as a datatype constructor which, given datatype A, builds a more elaborate datatype $\mathsf{F}\,A$; given another datatype B, it builds a similarly elaborate datatype F B; and so on.

The most important feature of a functor F is that its data-structuring effect extends smoothly to functions. Given a function $B \xleftarrow{f} A$, note that A and B are parameters of $\mathsf{F}\,A$ and $\mathsf{F}\,B$ respectively. Thus this data-structuring effect extends to a new function $\mathsf{F}\,B \xleftarrow{\mathsf{F} f} \mathsf{F}\,A$, depicted by the following diagram:

$$\begin{tikzcd}
A \arrow[r, dotted, no head] \arrow[d, "f"'] & \mathsf{F}\,A \arrow[d, "\mathsf{F} f"] \\
B \arrow[r, dotted, no head] & \mathsf{F}\,B
\end{tikzcd} \tag{3.25}$$

By definition, a functor obeys to two very basic properties: it commutes with identity

$$\mathsf{F}\; id_A = id_{\mathsf{F}\,A} \tag{3.26}$$

and with composition

$$\mathsf{F}\;(g \cdot h) = (\mathsf{F}\;g) \cdot (\mathsf{F}\;h) \tag{3.27}$$

Catamorphisms Given a functor F, any arrow $A \xleftarrow{\alpha} \mathsf{F}\,A$ is said to be an F-algebra, where A is called the *carrier* of the F-algebra α and contains the values that α operates on. This results in the computation of new A-values based on existing ones which are "encapsulated" in a F-pattern structure. Furthermore, given a function $B \xleftarrow{f} A$ and another F-algebra $B \xleftarrow{\beta} \mathsf{F}\,B$, one may consider to relate the F-algebra α to the other F-algebra β in the following manner:

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathsf{F}\,A \arrow[l, "\alpha"'] \arrow[d, "\mathsf{F} f"] \\
B & \mathsf{F}\,B \arrow[l, "\beta"]
\end{tikzcd} \qquad f \cdot \alpha = \beta \cdot (\mathsf{F} f) \tag{3.28}$$

This states that A-objects are mapped to B-objects in a structural way, according to the F-pattern. Arrows with this structure are usually referred to as *homomorphisms*.

It may happen that α is an isomorphism (or a bijective function), i.e., that exists some function α° such that $\alpha^\circ \cdot \alpha = id$ and $\alpha \cdot \alpha^\circ = id$. Such algebras α are said to be *initial* and usually denoted by in_T, that is, $\mathsf{F}\;\mathsf{T} \xrightarrow{\mathsf{in}_\mathsf{T}} \mathsf{T}$ assuming their carrier set denoted by T. Besides, the converse of the algebra in_T is called the coalgebra out_T. An F-coalgebra is an arrow $\mathsf{F}\,A \longleftarrow A$, for a functor F, where A is also called the *carrier*.

In this particular case, α is such that, for every β, f is unique. The uniqueness of f is denoted by the banana-brackets notation, $f = \llparenthesis \beta \rrparenthesis$, and captured by the universal property:

$$\begin{tikzcd}
\mathsf{T} \arrow[rr, bend left, "\mathsf{out}_\mathsf{T}"] \arrow[rr, phantom, "\cong"] \arrow[d, "f=\llparenthesis \beta \rrparenthesis"'] & & \mathsf{F}\,\mathsf{T} \arrow[ll, bend left, "\mathsf{in}_\mathsf{T}"] \arrow[d, "\mathsf{F}\,f"] \\
B & & \mathsf{F}\,B \arrow[ll, bend left, "\beta"]
\end{tikzcd}
\qquad f = \llparenthesis \beta \rrparenthesis \Leftrightarrow f \cdot \mathsf{in}_\mathsf{T} = \beta \cdot \mathsf{F}\,f \qquad (3.29)$$

This leads to the definition:

$$\llparenthesis \beta \rrparenthesis \stackrel{\mathrm{def}}{=} \beta \cdot \mathsf{F}\,\llparenthesis \beta \rrparenthesis \cdot \mathsf{out}_\mathsf{T} \qquad (3.30)$$

$\llparenthesis \beta \rrparenthesis$ is referred to as *the (unique) catamorphism* induced by algebra β (or *fold* over β). This construct is a generic and recursive expression that transforms T into B, following a "recursive-descent" approach as dictated by functor F.

Parameterization and type functors In order to properly characterize parametric inductive datatypes, the use of functors such as $\mathsf{T} \cong \mathsf{F}\,\mathsf{T}$ proves inadequate, as they do not provide a parametric perspective of datatype T. Then we may factor this out via the type variable X and write $\mathsf{T}\,X \cong \mathsf{B}\,(X, \mathsf{T}\,X)$ where B is called the type's *base functor*. Moreover, one has

$$\mathsf{F}\,f = \mathsf{B}\,(id, f) \qquad (3.31)$$

Concerning the functorial behavior of T, for a given function $f : A \to B$, $\mathsf{T}\,f$ can be expressed in terms of a $B\,(A, \_)$-catamorphism:

$$\begin{tikzcd}
\mathsf{T}\,A \arrow[d, "\mathsf{T}\,f"'] & & \mathsf{B}\,(A, \mathsf{T}\,A) \arrow[ll, "\mathsf{in}_{\mathsf{T}A}"'] \arrow[d, "\mathsf{B}\,(id, \mathsf{T}\,f)"] \\
\mathsf{T}\,B & \mathsf{B}\,(B, \mathsf{T}\,B) \arrow[l, "\mathsf{in}_{\mathsf{T}B}"] & \mathsf{B}\,(A, \mathsf{T}\,B) \arrow[l, "\mathsf{B}\,(f, id)"]
\end{tikzcd}
\qquad \mathsf{T}\,f \stackrel{\mathrm{def}}{=} \llparenthesis \mathsf{in}_\mathsf{T} \cdot \mathsf{B}\,(f, id) \rrparenthesis \qquad (3.32)$$

Finally, type functors allow us to define the absorption law for the catamorphism combinator:

$$\llparenthesis g \rrparenthesis \cdot \mathsf{T}\,f = \llparenthesis g \cdot \mathsf{B}\,(f, id) \rrparenthesis \qquad (3.33)$$

Anamorphisms By inverting the arrows of a catamorphism diagram (3.29) one is lead to the concept of the *(unique) anamorphism* induced by algebra β. Under some conditions, this construct is a generic and recursive expression that synthesizes the inductive datatype T, following a recursive approach as dictated by functor F.

$$\begin{tikzcd}
\mathsf{T} \arrow[r, bend left, "\mathsf{out}_{\mathsf{T}}"] \arrow[r, phantom, "\cong"] & \mathsf{F}\,\mathsf{T} \arrow[l, bend left, "\mathsf{in}_{\mathsf{T}}"] \\
B \arrow[u, "k=[\!(\beta)\!]"] \arrow[r, bend right, "\beta"'] & \mathsf{F}\,B \arrow[u, "\mathsf{F}\,k"']
\end{tikzcd}
\qquad k = [\!(\beta)\!] \Leftrightarrow \mathsf{out}_{\mathsf{T}} \cdot k = (\mathsf{F}\,k) \cdot \beta \tag{3.34}$$

While a catamorphism is the unique homomorphism from the initial algebra to another algebra of a functor, an anamorphism is the unique homomorphism from a coalgebra to its final coalgebra. English speaking, while a catamorphism consumes a datatype, an anamorphism synthesizes it.

Hylomorphisms The composition of a catamorphism with an anamorphism is referred to as a hylomorphism, with the following notation:

$$\begin{tikzcd}
A \arrow[r, bend left, "g"] \arrow[d, "[\!(g)\!]"'] & \mathsf{F}\,A \arrow[d, "\mathsf{F}\,[\!(g)\!]"] \\
\mathsf{T} \arrow[r, bend left, "\mathsf{out}_{\mathsf{T}}"] \arrow[r, phantom, "\cong"] \arrow[d, "(\!|f|\!)"'] & \mathsf{F}\,\mathsf{T} \arrow[l, bend left, "\mathsf{in}_{\mathsf{T}}"] \arrow[d, "\mathsf{F}\,(\!|f|\!)"] \\
B & \mathsf{F}\,B \arrow[l, bend left, "f"]
\end{tikzcd}
\qquad \llbracket f, g \rrbracket = (\!|f|\!) \cdot [\!(g)\!] \tag{3.35}$$

T is said to be the intermediate structure of the hylomorphism. Often, this structure remains concealed — it is a virtual data structure — as one adopts the *Divide & Conquer* perspective of a hylomorphism, cf.

$$\begin{tikzcd}
A \arrow[r, "divide"] \arrow[d, "h"'] & \mathsf{F}\,A \arrow[d, "\mathsf{F}\,h"] \\
B & \mathsf{F}\,B \arrow[l, "conquer"]
\end{tikzcd}
\qquad h = conquer \cdot \mathsf{F}\,h \cdot divide \tag{3.36}$$

Adjunctions In general, given two functors R and L, an isomorphism of shape

$$\begin{tikzcd}
\mathsf{L}\,A \to B \arrow[r, bend left, "\lceil \_ \rceil"] \arrow[r, phantom, "\cong"] & A \to \mathsf{R}\,B \arrow[l, bend left, "\lfloor \_ \rfloor"]
\end{tikzcd} \tag{3.37}$$

is called an *adjunction* of R and L, which are said to be *adjoint* of each other. One writes L ⊣ R and says that L is the left adjoint and R the right adjoint. As already shown, the witnesses of the isomorphism carry a very simple notation, cf.

$$k = \lceil f \rceil \Leftrightarrow f = \underbrace{\varepsilon \cdot \mathsf{L}\,k}_{\lfloor k \rfloor} \qquad \begin{tikzcd} \mathsf{R}\,B \\ A \arrow[u, "k=\lceil f \rceil"] \end{tikzcd} \qquad \begin{tikzcd} \mathsf{L}\,(\mathsf{R}\,B) \arrow[r, "\varepsilon"] & B \\ \mathsf{L}\,A \arrow[u, "\mathsf{L}\,k"'] \arrow[ur, "f"'] \end{tikzcd} \tag{3.38}$$

As is well known as is detailed in (Oliveira, 2023), GCs are particular cases of adjunctions.

Interplay between catamorphisms and adjunctions Given an adjunction $\mathsf{L} \dashv \mathsf{R}$, an inductive datatype $\mathsf{T} \cong \mathsf{F}\,\mathsf{T}$ and $\phi : \mathsf{L}\,\mathsf{F} \rightarrow \mathsf{G}\,\mathsf{L}$ a natural transformation for some functor G, then

$$f \cdot (\mathsf{L}\,\mathsf{in}_{\mathsf{F}}) = h \cdot \mathsf{G} f \cdot \phi \Leftrightarrow \lceil f \rceil = (\!| \lceil h \cdot \mathsf{G}\,\varepsilon \cdot \phi \rceil |\!) \tag{3.39}$$

holds (Oliveira, 2023). This states that the G-hylomorphism (left-hand side of (3.41))

$$\begin{tikzcd} \mathsf{L}\,\mathsf{T} \arrow[d, "f"'] & & \mathsf{G}\,\mathsf{L}\,\mathsf{T} \arrow[d, "\mathsf{G} f"'] & \mathsf{L}\,\mathsf{F}\,\mathsf{T} \arrow[l, "\phi"] \arrow[lll, bend right, "\mathsf{L}\,\mathsf{in}_{\mathsf{F}}"'] \\ A & & \mathsf{G}\,A \arrow[ll, "h"] & \end{tikzcd}$$

$$f = h \cdot \mathsf{G} f \cdot \phi \cdot \mathsf{L}\,out$$

is equivalent to the F-catamorphism

$$\begin{tikzcd} \mathsf{T} \arrow[d, "\lceil f \rceil"'] & \mathsf{F}\,\mathsf{T} \arrow[l, "\mathsf{in}_{\mathsf{F}}"'] \arrow[d, "\mathsf{F}\,\lceil f \rceil"] \\ \mathsf{R}\,A & \mathsf{F}\,\mathsf{R}\,A \arrow[l, "\lceil h \cdot G\,\varepsilon \cdot \phi \rceil"] \end{tikzcd}$$

$$\begin{tikzcd} A & \mathsf{G}\,A \arrow[l, "h"'] & \mathsf{G}\,\mathsf{L}\,\mathsf{R}\,A \arrow[l, "\mathsf{G}\,\varepsilon"'] & \mathsf{L}\,\mathsf{F}\,\mathsf{R}\,A \arrow[l, "\phi"'] \end{tikzcd}$$

$$\lceil f \rceil = (\!| \lfloor h \cdot G\,\varepsilon \cdot \phi \rfloor |\!)$$

Particularly, for the aforementioned adjunction (3.23), that is,

$$\begin{tikzcd} C \times A \rightarrow B \arrow[r, bend left, "curry"] \arrow[r, phantom, "\cong"] & C \rightarrow (A \rightarrow B) \arrow[l, bend left, "uncurry"] \end{tikzcd} \tag{3.40}$$

the following isomorphism arises

$$f \cdot (\mathsf{in}_{\mathsf{T}} \times id) = h \cdot \mathsf{G} f \cdot \phi \Leftrightarrow \overline{f} = (\!| \overline{h \cdot \mathsf{G}\,ap \cdot \phi} |\!) \tag{3.41}$$

which will prove very useful in calculating the curried version of some recursive functions.

3.2 Calculating with relations

We acknowledge that calculating programs from specifications in the form of Galois connections is intricately tied to defining the orders to which the connections are subject. As anticipated, these orders manifest as relations rather than functions, constituting a significantly more generic and expressive conceptual framework. Consequently, the exploration of relations assumes a pivotal role within this domain. It is within this context that the present section endeavors to address the essential aspects of relational calculus, thereby ensuring a succinct and rigorous examination of the process of programming from Galois connections.

Binary relations In relation algebra, functions are generalized to binary relations, which may produce multiple outputs. First of all, such relations are denoted by arrows exactly in the same way functions are. That is, relations are typed in the same way as functions. So, $B \xleftarrow{R} A$ indicates that relation R relates B-values to A-values, writing $b\,R\,a$ which is read as "b is related by R to a". This is the same as writing $(b,a) \in R$ since relations are sets of pairs.

The adopted convention is to employ lowercase letters to denote functions and uppercase letters to denote these generalized binary relations.

Relation composition Generalized to relation algebra, composition ($R \cdot S$) takes into account the multiple values that may be produced by S. It is defined as follows:

$$\begin{tikzcd} B & A \arrow[l, "R"'] & C \arrow[l, "S"'] \arrow[ll, bend left, "R\cdot S"] \end{tikzcd} \qquad b(R\cdot S)c \;\equiv\; \langle \exists\, a \;:\; b\,R\,a :\; a\,S\,c \rangle \tag{3.42}$$

An element b is related to an element c by $(R \cdot S)$ if (and only if) there exists an element a to which b is related by R and which is related to c by S. This compact notation for relation composition allows us to deal with existential quantification in a pointfree style in our proofs.

Converses Unlike functions, which do not always have corresponding converse functions, *every* relation $A \xrightarrow{R} B$ has a converse $A \xleftarrow{R^\circ} B$ defined by:

$$b\,R\,a \;\Leftrightarrow\; a\,R^\circ\,b \tag{3.43}$$

In terms of grammar, R° corresponds to the passive voice — compare e.g.

$$\underbrace{John}_{b}\;\underbrace{loves}_{R}\;\underbrace{Mary}_{a}$$

with

$$\underbrace{Mary}_{a}\ \underbrace{is\ loved\ by}_{R^\circ}\ \underbrace{John}_{b}$$

That is, $loves^\circ = (is\ loved\ by)$.

Naturally, the converse of a converse is the relation itself — converse is an involution

$$(R^\circ)^\circ = R \tag{3.44}$$

and commutes with composition in a contravariant way:

$$R \cdot S^\circ = S^\circ \cdot R^\circ \tag{3.45}$$

Converses of functions enjoy a number of properties from which the following is singled out as a way to introduce/remove them from logical expressions (useful when dealing with more complex types):

$$b\,(f^\circ \cdot R \cdot g)\,a \;\Leftrightarrow\; (f\,a)\,R\,(g\,b) \tag{3.46}$$

Relation inclusion and equality While function equality can be expressed by extensionality,

$$f = g \;\equiv\; \langle \forall\, a \;:\; a \in A : f\,a = g\,a \rangle$$

there are two methods, besides direct equality, to prove relational equality: *circular inclusion*, and *indirect equality*. Both of these rely on the notion of relation inclusion, defined as follows:

$$R \subseteq S \;\equiv\; \langle \forall\, a, b \;::\; a\,R\,b \Rightarrow a\,S\,b \rangle \tag{3.47}$$

Circular inclusion arrives at equality by proving, in both directions, that one relation is smaller than, or, at most, equal to the other. This method is often called "ping-pong", and in proofs of this type the two steps will be referred to as the "ping" step and the "pong" step.

$$R = S \;\equiv\; R \subseteq S \wedge S \subseteq R \tag{3.48}$$

In indirect equality, we prove that inclusion under one relation is equivalent to inclusion under the other. That is to say, one shows that all relations having R as a subset have S as a subset, and vice-versa.

$$R = S \;\equiv\; \langle \forall\, X \;::\; X \subseteq R \Leftrightarrow X \subseteq S \rangle \tag{3.49}$$

$$\;\equiv\; \langle \forall\, X \;::\; R \subseteq X \Leftrightarrow S \subseteq X \rangle \tag{3.50}$$

Meet and join Like sets, two relations of the same type, say $B \xleftarrow{R,S} A$, can be intersected or joined in the obvious way:

$$b\,(R \cap S)\,a \;\equiv\; a\,R\,a \wedge b\,S\,a \tag{3.51}$$

$$b\,(R \cup S)\,a \;\equiv\; a\,R\,a \vee b\,S\,a \tag{3.52}$$

$R \cap S$ is usually called *meet* (intersection) and $R \cup S$ is called *join* (union). They lift pointwise conjunction and disjunction, respectively, to the pointfree level. Their meaning is nicely captured by the following universal properties:

$$X \subseteq R \cap S \;\equiv\; X \subseteq R \wedge X \subseteq S \tag{3.53}$$

$$R \cup S \subseteq X \;\equiv\; R \subseteq X \wedge S \subseteq X \tag{3.54}$$

Taxonomy of binary relations Let us now define some basic properties of some relations, which will be the starting point to describe more complex properties.

- A relation is *reflexive* if and only if every element is related to itself:

$$R \text{ is reflexive } \;\Leftrightarrow\; id \subseteq R \tag{3.55}$$

- A relation is *correflexive* if and only if it is a subset of the identity relation:

$$R \text{ is correflexive } \;\Leftrightarrow\; R \subseteq id \tag{3.56}$$

- A relation is *transitive* if and only if $c\,R\,a$ whenever $c\,R\,b$ and $b\,R\,a$:

$$R \text{ is transitive } \;\Leftrightarrow\; R \cdot R \subseteq R \tag{3.57}$$

- A relation is *antisymmetric* iff $b = a$ wherever $b\,R\,a$ and $a\,R\,b$:

$$R \text{ is antisymmetric } \;\equiv\; R \cap R^{\circ} \subseteq id \tag{3.58}$$

A *preorder* is a relation that is both reflexive and transitive. An antisymmetric preorder is referred to as a *partial order*. Well-known examples of such partial orders are the numerical comparison operator $(\leqslant)$ and relation inclusion. In subsequent chapters, the properties of this and other operators as partial orders will play a crucial role in many proofs.

Additionally, a relation is:

- *entire* if it is defined for every element of its domain.
- *injective* if no two elements of its domain are related to the same element.
- *surjective* if, for every element of its range, there is at least one element related to it.
- *simple* if each element of its domain is related to at most one element.

To define these last four properties using relation algebra, we first introduce the notions of *kernel* and *image*:

$$\ker R = R^{\circ} \cdot R \tag{3.59}$$

$$\mathrm{img}\, R = R \ \cdot R^{\circ} \tag{3.60}$$

Thee kernel of R relates inputs that share shared outputs, and its image relates outputs to shared inputs.

$$R \text{ injective:} \quad \ker R \subseteq id \tag{3.61}$$

$$R \text{ simple:} \quad \mathrm{img}\, R \subseteq id \tag{3.62}$$

$$R \text{ entire:} \quad id \subseteq \ker \cdot R \tag{3.63}$$

$$R \text{ surjective:} \quad id \subseteq \mathrm{img} \cdot R \tag{3.64}$$

This can be “translated” to the following table:

	Reflexive	*Coreflexive*
$\ker R$	entire R	injective R
$\mathrm{img}\, R$	surjective R	simple R

Figure 1 shows a taxonomy of binary relations based on the combination of the previous four properties. It is quite interesting because it denotes the central role played by functions.

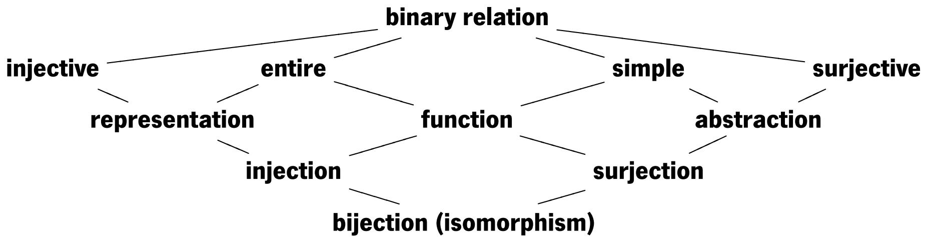

Fig. 1: Binary relation taxonomy

The properties of different kinds of relations have dualities under converse, that is:

$$R \text{ is a bijection} \;\Leftrightarrow\; R \text{ and } R^{\circ} \text{ are both functions} \tag{3.65}$$

$$R \text{ is injective} \;\Leftrightarrow\; R^{\circ} \text{ is simple} \tag{3.66}$$

$$R \text{ is surjective} \;\Leftrightarrow\; R^{\circ} \text{ is entire} \tag{3.67}$$

By extension, the converse of a representation is an abstraction, the converse of an injection is a surjection, and vice-versa.

Due to transitivity of inclusion and the fact that a larger relation also has a larger (or at most equal) kernel and image, making a relation bigger or smaller can preserve some of its properties, as follows:

$$S \text{ is injective (simple)} \;\Leftarrow\; S \subseteq R \text{ and } R \text{ is injective (simple)} \tag{3.68}$$

$$S \text{ is entire (surjective)} \;\Leftarrow\; R \subseteq S \text{ and } R \text{ is entire (surjective)} \tag{3.69}$$

Relators As a generalization of a functor, a parametric datatype G is said to be a relator wherever, given a relation $B \xleftarrow{R} A$, G R extends R to G-structures forming the new relation $\mathsf{G}\,B \xleftarrow{\mathsf{G}\,R} \mathsf{G}\,A$ depicted by the following diagram:

$$\begin{tikzcd}
A \arrow[r, dotted, no head] \arrow[d, "R"'] & \mathsf{G}\,A \arrow[d, "\mathsf{G}\,R"] \\
B \arrow[r, dotted, no head] & \mathsf{G}\,B
\end{tikzcd} \tag{3.70}$$

By definition, relators, in addition to obeying the two properties of functors — commutation with identity

$$\mathsf{G}\; id_A = id\;(\mathsf{G}\,A) \tag{3.71}$$

and with composition

$$\mathsf{G}\;(R \cdot S) = (\mathsf{G}\,R) \cdot (\mathsf{G}\,S) \tag{3.72}$$

— they also obey a third property — commutation with converse

$$\mathsf{G}\;(R^{\circ}) = (\mathsf{G}\,R)^{\circ} \tag{3.73}$$

Top and bottom The $\top$ ("top") and $\bot$ ("bottom") relations

$$b \top a \;\equiv\; True \tag{3.74}$$

$$b \bot a \;\equiv\; False \tag{3.75}$$

are the upper and lower bound for all relations of a certain relational type, meaning that, for any relation $B \xleftarrow{R} A$:

$$\perp \subseteq R \subseteq \top \tag{3.76}$$

These relations are the zero and identity elements of the relational join and meet operations:

$$R \cup \perp = R \tag{3.77}$$

$$R \cap \perp = \perp \tag{3.78}$$

$$R \cup \top = \top \tag{3.79}$$

$$R \cap \top = R \tag{3.80}$$

Sometimes it is possible to discard relations composed with $\top$ to use the laws introduced above. This can be done if the relation in question is entire:

$$\top \cdot R = \top \ \Leftarrow \ R \ is \ entire \tag{3.81}$$

Since $\top$ is larger than any relation (3.76), it suffices to prove $\top \subseteq \top \cdot R$:

$$\begin{array}{cl} & \top \subseteq \top \cdot R \\ \Leftarrow & \quad \{ \ R \text{ is assumed entire; raising the lower side } \} \\ & \top \cdot R^{\circ} \cdot R \subseteq \top \cdot R \\ \Leftarrow & \quad \{ \text{ monotonicity } \} \\ & \top \cdot R^{\circ} \subseteq \top \\ \equiv & \quad \{ \ \top \text{ above everything } \} \\ & True \end{array}$$

□

Predicates as relations Given a predicate $p : A \to \mathbb{B}$, the relation $\phi_p : A \to A$ defined by

$$\phi_p = id \cap true^{\circ} \cdot p \tag{3.82}$$

for $true\ x = True$, is said to be the *coreflexive* relation that represents predicate p as binary relation, cf.

$$y \ \phi_p \ x \Leftrightarrow y = x \wedge p \ y \tag{3.83}$$

Two crucial derived properties are:

$$R \cdot \phi_p = R \cap \top \cdot \phi_p \tag{3.84}$$

$$\phi_q \cdot R = R \cap \phi_q \cdot \top \tag{3.85}$$

which respectively mean

$$b\ (R \cdot \phi_p)\ a \Leftrightarrow b\ R\ a \wedge (p\ a) \tag{3.86}$$

$$b\ (\phi_q \cdot R)\ a \Leftrightarrow b\ R\ a \wedge (q\ b) \tag{3.87}$$

Finally, a notable property is that the composition of coreflexives coincides with their meet:

$$\phi_q \cdot \phi_p = \phi_q \cap \phi_p \tag{3.88}$$

Relational catamorphisms They are identical to functional catamorphisms, with the distinction that now the F-algebra β in diagram (3.29) is a relation, say R, and F is a relator instead of a functor. Thus, there is a unique relation of type $B \leftarrow \mathsf{T}$, denoted by $(\!|R|\!)$, such that $(\!|R|\!) \cdot \mathsf{in}_{\mathsf{T}} = R \cdot \mathsf{F}\ (\!|R|\!)$ holds, cf.

$$\begin{tikzcd}
\mathsf{T} \arrow[r, bend left, "\mathsf{out}_{\mathsf{T}}"] \arrow[r, phantom, "\cong"] \arrow[d, "(\!|R|\!)"'] & \mathsf{F}\ \mathsf{T} \arrow[l, bend left, "\mathsf{in}_{\mathsf{T}}"] \arrow[d, "\mathsf{F}\ (\!|R|\!)"] \\
B & \mathsf{F}\ B \arrow[l, bend left, "R"]
\end{tikzcd} \tag{3.89}$$

A crucial property is the *fusion law*:

$$S \cdot (\!|R|\!) = (\!|Q|\!) \Leftarrow S \cdot R = Q \cdot \mathsf{F}\ S \tag{3.90}$$

along with its weaker versions:

$$Q \cdot (\!|S|\!) \subseteq (\!|R|\!) \Leftarrow Q \cdot S \subseteq R \cdot \mathsf{F}\ Q \tag{3.91}$$

$$(\!|R|\!) \subseteq Q \cdot (\!|S|\!) \Leftarrow R \cdot \mathsf{F}\ Q \subseteq Q \cdot S \tag{3.92}$$

Relational catamorphism are central to relation algebra (Bird and de Moor, 1997). Oliveira (2023) shows how they arise from (3.41) for the adjunction that yields the powerset monad.

3.3 Galois connections go pointfree

In Section 2.1, a Galois connection was defined pointwise as a pair of functions f and g satisfying

$$f\ z \leqslant x \Leftrightarrow z \sqsubseteq g\ x$$

for all z and x, given preorders $(\leqslant)$ and $(\sqsubseteq)$. Now, relation composition, relational equality and converses can be used to derive the GC pointfree definition, as follows.

$$
\begin{array}{ll}
 & f\,z \leqslant x \Leftrightarrow z \sqsubseteq g\,x \\
\equiv & \qquad \{\ \text{(3.46); identity}\ \} \\
 & z\,(f^{\circ} \cdot (\leqslant))\,x \Leftrightarrow z\,((\sqsubseteq) \cdot g)\,x \\
\equiv & \qquad \{\ \text{(3.47) twice; circular inclusion (3.48)}\ \} \\
 & f^{\circ} \cdot (\leqslant) = (\sqsubseteq) \cdot g
\end{array}
$$

The rendering of Galois connections as relational equalities,

$$f^{\circ} \cdot (\leqslant) = (\sqsubseteq) \cdot g \tag{3.93}$$

will prove very useful in the sequel.

3.4 Summary

A brief introduction to pointfree calculus and relation algebra was presented, highlighting relevant concepts such as functions, functional combinators, binary relations, relation composition, converses, relation inclusion and equality, and the taxonomy of binary relations. This study is crucial in formulating Galois connections as relational equalities, as the use of pointfree notation is essential for rigorously justifying the entire developed theory.

Part II

Contribution

Chapter 4

Programming from Galois connections

A pivotal factor in Galois connection-based programming lies in the specification of the underlying orders. This chapter will show how the (pointfree) relational definition of these (inductive) orders is essential for accurately inferring the functions specified by the connections they rely upon.

4.1 Inductive partial orderings

4.1.1 Peano algebra

Giuseppe Peano (1858-1932) was a famous Italian mathematician who made significant contributions to the field of mathematics, particularly in the area of mathematical logic and the foundations of arithmetic. Peano is well-known for developing the so-called *Peano axioms*, which provide a set of five axioms for the construction of the natural numbers. The study of these axioms is not relevant to this discussion. The relevance lies in what these axioms translate into, that is, a *unique* way of constructing natural numbers:

> *Every natural number in* $\mathbb{N}_0$ *is either* 0 *or the successor of another natural number.*

Thus, for

$$\begin{aligned} &succ : \mathbb{N}_0 \to \mathbb{N}_0 \\ &succ\ n \overset{\mathsf{def}}{=} n + 1 \end{aligned} \tag{4.1}$$

and

$$\mathsf{in}_{\mathbb{N}_0} = [\underline{0}, succ] \tag{4.2}$$

$$\begin{cases} \mathsf{out}_{\mathbb{N}_0}\ 0 = i_1\ () \\ \mathsf{out}_{\mathbb{N}_0}\ (n+1) = i_2\ n \end{cases} \tag{4.3}$$

one gets the Peano algebra isomorphism:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[r, bend left, "\mathsf{out}_{\mathbb{N}_0}"] \arrow[r, phantom, "\cong"] & 1+\mathbb{N}_0 \arrow[l, bend left, "{[\underline{0}\,,succ]}"]
\end{tikzcd} \tag{4.4}$$

which will be used to define the aforementioned orderings in the domain of $\mathbb{N}_0$.

Concerning the *greater or equal partial order*, it may be defined via the $\mathbb{N}_0$-catamorphism combinator by

$$(\geqslant) = \llparenthesis [\top\,, succ] \rrparenthesis \tag{4.5}$$

$$\equiv \qquad \{ \text{ catamorphisms } \}$$

$$\begin{cases} (\geqslant) \cdot \underline{0} = \top \\ (\geqslant) \cdot succ = succ \cdot (\geqslant) \end{cases} \tag{4.6}$$

$$\equiv \qquad \{ \text{ going pointwise } \}$$

$$\begin{cases} y \geqslant 0 \Leftrightarrow True \\ y \geqslant (n+1) \Leftrightarrow \langle \exists x : y = x+1 : x \geqslant n \rangle \end{cases} \tag{4.7}$$

As in (Oliveira, 2022), the monotonicity of the catamorphism combinator ensures reflexivity:

$$id \subseteq (\geqslant)$$

$$\equiv \qquad \{ \ id_{\mathsf{T}} = \llparenthesis \mathsf{in}_{\mathsf{T}} \rrparenthesis ;\ \mathsf{in}_{\mathsf{T}} = [\underline{0}, succ] \ \}$$

$$\llparenthesis [\underline{0}, succ] \rrparenthesis \subseteq \llparenthesis [\top, succ] \rrparenthesis$$

$$\Leftarrow \qquad \{ \text{ monotonicity } \}$$

$$[\underline{0}, succ] \subseteq [\top, succ]$$

$$\equiv \qquad \{ \text{ coproducts } \}$$

$$\begin{cases} \underline{0} \subseteq \top \\ succ \subseteq succ \end{cases}$$

$$\equiv \qquad \{ \text{ trivial } \}$$

$$True$$

□

On the other hand, fusion and absorption provide us with the proof of transitivity:

$$(\geqslant) \cdot (\geqslant) \subseteq (\geqslant)$$

$$\equiv \qquad \{ \text{ definitions } \}$$

$$(\geqslant)\cdot(\![\top,succ]\!) \subseteq (\![\top,succ]\!)$$

$\Leftarrow$ { catamorphism fusion (3.91) }

$$(\geqslant)\cdot[\top,succ] \subseteq [\top,succ]\cdot(id+(\geqslant))$$

$\equiv$ { fusion and absorption }

$$\begin{cases} (\geqslant)\cdot\top \subseteq \top \\ (\geqslant)\cdot succ \subseteq succ\cdot(\geqslant) \end{cases}$$

$\equiv$ { trivial; $(\geqslant)\cdot succ = succ\cdot(\geqslant)$ (4.7) }

$$True$$

□

Proving antisymmetry requires knowledge of $(\geqslant)^\circ$, which Oliveira (2020b) has shown to be $(\leqslant)$, the *less than or equal* defined by:

$$(\leqslant) = (\![\underline{0},\underline{0} \cup succ]\!) \tag{4.8}$$

$\equiv$ { catamorphisms }

$$\begin{cases} (\leqslant)\cdot\underline{0} = \underline{0} \\ (\leqslant)\cdot succ = (\underline{0} \cup succ)\cdot(\leqslant) \end{cases}$$

$\equiv$ { going pointwise }

$$\begin{cases} y \leqslant 0 \Leftrightarrow y = 0 \\ y \leqslant (n+1) \Leftrightarrow y = 0 \vee \langle \exists x : y = x+1 : x \leqslant n \rangle \end{cases} \tag{4.9}$$

Note that proving $(\leqslant)$ being a partial order is trivial, since by applying converses on both sides of

$$id \subseteq (\leqslant)$$

$$(\leqslant)\cdot(\leqslant) \subseteq (\leqslant)$$

$$(\leqslant) \cap (\geqslant)^\circ \subseteq id$$

one obtains the conditions for $(\geqslant)$ to be a partial order. However, the proof of antisymmetry, $(\leqslant) \cap (\geqslant)^\circ \subseteq id$, is still lacking. As a foretaste, note that

$$(\leqslant) = (\!\underbrace{\mathsf{in}_{\mathbb{N}_0} \cup [\bot,\underline{0}]}_{R}\!) \qquad (\geqslant) = (\!\underbrace{\mathsf{in}_{\mathbb{N}_0} \cup [\top,\bot]}_{S}\!) \tag{4.10}$$

and so:

$$R \cap S = \mathsf{in}_{\mathbb{N}_0} \cup ([\bot,\underline{0}] \cap [\top,\bot]) = \mathsf{in}_{\mathbb{N}_0} \cup \bot = \mathsf{in}_{\mathbb{N}_0} \tag{4.11}$$

Clearly, (4.11) is an indication of how R and S cancel each other as far as what they "add" to the initial algebra $\mathsf{in}_{\mathbb{N}_0}$, which is such that $\llparenthesis \mathsf{in}_{\mathbb{N}_0} \rrparenthesis = id$, as is well known.

Let us calculate the meaning of $(\leqslant) \cap (\geqslant)$. By mere convenience we use indirect equality:

$$(\leqslant) \cap (\geqslant) \subseteq X$$

$\equiv \quad \{$ definitions (4.10); shunt $\mathsf{in}_{\mathbb{N}_0}$ to the right $\}$

$$(\mathsf{in}_{\mathbb{N}_0} \cup [\bot, \underline{0}]) \cdot (id + (\leqslant)) \cap (\mathsf{in}_{\mathbb{N}_0} \cup [\top, \bot]) \cdot (id + (\geqslant)) \subseteq X \cdot \mathsf{in}_{\mathbb{N}_0}$$

$\equiv \quad \{$ distributivity; simplifications $\}$

$$(\mathsf{in}_{\mathbb{N}_0} \cdot (id + (\leqslant)) \cup [\bot, \underline{0}]) \cap (\mathsf{in}_{\mathbb{N}_0} \cdot (id + (\geqslant)) \cup [\top, \bot]) \subseteq X \cdot \mathsf{in}_{\mathbb{N}_0}$$

$\equiv \quad \{$ $\cup/\cap$- distributivity; simplifications; (4.11) $\}$

$$\begin{array}{ll} \mathsf{in}_{\mathbb{N}_0} \cdot ((id + (\leqslant)) \cap (id + (\geqslant))) \cup (\mathsf{in}_{\mathbb{N}_0} \cdot (id + (\leqslant)) \cap [\top, \bot]) & \\ \quad \cup & \subseteq X \cdot \mathsf{in}_{\mathbb{N}_0} \\ ([\bot, \underline{0}] \cap [\underline{0}, succ \cdot (\leqslant)]) \cup \bot & \end{array}$$

$\equiv \quad \{$ coproducts: meet of sums and alternatives; $\bot$ and $\top$ $\}$

$$\mathsf{in}_{\mathbb{N}_0} \cdot (id + ((\leqslant) \cap (\geqslant))) \cup [\underline{0}, \bot] \cup [\bot, \underline{0} \cap (succ \cdot (\leqslant))] \subseteq X \cdot \mathsf{in}_{\mathbb{N}_0}$$

$\equiv \quad \{$ $0 = succ\ n$ impossible for any n $\}$

$$\mathsf{in}_{\mathbb{N}_0} \cdot (id + ((\leqslant) \cap (\geqslant))) \cup [\underline{0}, \bot] \subseteq X \cdot \mathsf{in}_{\mathbb{N}_0}$$

$\equiv \quad \{$ $[\underline{0}, \bot] = \mathsf{in}_{\mathbb{N}_0} \cdot (id + \bot)$; linearity; bring $\mathsf{in}_{\mathbb{N}_0}$ back $\}$

$$\mathsf{in}_{\mathbb{N}_0} \cdot (id + ((\leqslant) \cap (\geqslant)) \cdot \mathsf{in}^{\circ}_{\mathbb{N}_0} \subseteq X$$

$\equiv \quad \{$ $\mathsf{F}\, R = id + R$ $\}$

$$\mathsf{in}_{\mathbb{N}_0} \cdot \mathsf{F}\, ((\leqslant) \cap (\geqslant)) \cdot \mathsf{in}^{\circ}_{\mathbb{N}_0} \subseteq X$$

$:: \quad \{$ indirect equality $\}$

$$(\leqslant) \cap (\geqslant) = \mathsf{in}_{\mathbb{N}_0} \cdot \mathsf{F}\, ((\leqslant) \cap (\geqslant)) \cdot \mathsf{in}^{\circ}_{\mathbb{N}_0}$$

$\equiv \quad \{$ relational catamorphisms (3.89) $\}$

$$(\leqslant) \cap (\geqslant) = \llparenthesis \mathsf{in}_{\mathbb{N}_0} \rrparenthesis = id$$

In summary, both inductive orders are indeed partial orders.

4.1.2 Algebra of lists

Lists (or finite sequences) are foundational data structures in diverse problem-solving contexts. They exhibit the following recursive structure: a list is either empty, serving as the base case, or a constructed list composed of an element (commonly referred as the head) and another list (commonly referred as the tail). Given some type A, notation A^* is taken to express that type of all lists of A-values. Thus,

$$\begin{tikzcd} A^* \arrow[rr, bend left, "\mathsf{out}_{\mathsf{List}}"] & \cong & 1 + A \times A^* \arrow[ll, bend left, "\mathsf{in}_{\mathsf{List}}"] \end{tikzcd} \tag{4.12}$$

forms the algebra of lists, an isomorphism where

$$\mathsf{in}_{\mathsf{List}} = [nil, cons] \tag{4.13}$$

$$nil = \underline{[\,]} \tag{4.14}$$

$$cons = \widehat{(:)} \tag{4.15}$$

$$\begin{cases} \mathsf{out}_{\mathsf{List}}\ [\,] = i_1\ () \\ \mathsf{out}_{\mathsf{List}}\ (h:t) = i_2\ (h,t) \end{cases} \tag{4.16}$$

The partial orders on lists relevant for this dissertation are given next.

Prefix partial order It may be defined via the List-catamorphism combinator by

$$(\preceq) = (\!|\, [nil, cons \ \cup\ nil] \,|\!)$$

$$\equiv \qquad \{ \text{ catamorphisms } \}$$

$$\begin{cases} (\preceq) \cdot nil = nil \\ (\preceq) \cdot cons = (cons \ \cup\ nil) \cdot (id \times (\preceq)) \end{cases}$$

$$\equiv \qquad \{ \text{ left linearity; constants } \}$$

$$\begin{cases} (\preceq) \cdot nil = nil \\ (\preceq) \cdot cons = cons \cdot (id \times (\preceq)) \ \cup\ nil \end{cases} \tag{4.17}$$

$$\equiv \qquad \{ \text{ going pointwise } \}$$

$$\begin{cases} s\ ((\preceq) \cdot nil)\ x \Leftrightarrow s\ nil\ x \\ s\ ((\preceq) \cdot cons)\ (h,t) \Leftrightarrow s\ (cons \cdot (id \times (\preceq)))\ (h,t) \ \vee\ s\ nil\ (h,t) \end{cases}$$

$$\equiv \qquad \{ \text{ composition } \}$$

$$\begin{cases} s \preceq [\,] \Leftrightarrow s = [\,] \\ s \preceq (h:t) \Leftrightarrow \langle \exists\, (x,y) : s = (x:y) : x = h \wedge y \preceq t \rangle \vee s = [\,] \end{cases}$$

$$\equiv \qquad \{ \text{ one-point } \}$$

$$\begin{cases} s \preceq [\,] \Leftrightarrow s = [\,] \\ s \preceq (h:t) \Leftrightarrow \langle \exists\, y : s = (h:y) : y \preceq t \rangle \vee s = [\,] \end{cases} \tag{4.18}$$

Again, catamorphism monotonicity provides us with the proof that $(\preceq)$ is reflexive, cf.

$$
\begin{array}{ll}
 & id \subseteq (\preceq) \\
\equiv & \quad \{ \ id_{\mathsf{T}} = \llparenthesis \mathsf{in}_{\mathsf{T}} \rrparenthesis ;\ \mathsf{in}_{\mathsf{T}} = [nil, cons] \ \} \\
 & \llparenthesis [nil, cons] \rrparenthesis \subseteq \llparenthesis [nil, cons \cup nil] \rrparenthesis \\
\Leftarrow & \quad \{ \ \text{monotonicity} \ \} \\
 & [nil, cons] \subseteq [nil, cons \cup nil] \\
\equiv & \quad \{ \ \text{coproducts} \ \} \\
 & \left\{ \begin{array}{l} nil \subseteq nil \\ cons \subseteq cons \cup nil \end{array} \right. \\
\equiv & \quad \{ \ \text{trivial; } R \subseteq R \cup S \ \} \\
 & True \\
\Box &
\end{array}
$$

The fusion and absorption laws of catamorphisms ensure $(\preceq)$-transitivity, cf.

$$
\begin{array}{ll}
 & (\preceq) \cdot (\preceq) \subseteq (\preceq) \\
\equiv & \quad \{ \ \text{definitions} \ \} \\
 & (\preceq) \cdot \llparenthesis [nil, cons \cup nil] \rrparenthesis \subseteq \llparenthesis [nil, cons \cup nil] \rrparenthesis \\
\Leftarrow & \quad \{ \ \text{catamorphism fusion} \ \} \\
 & (\preceq) \cdot [nil, cons \cup nil] \subseteq [nil, cons \cup nil] \cdot (id + id \times (\preceq)) \\
\equiv & \quad \{ \ \text{fusion and absorption} \ \} \\
 & \left\{ \begin{array}{l} (\preceq) \cdot nil \subseteq nil \\ (\preceq) \cdot (cons \cup nil) \subseteq (cons \cup nil) \cdot (id \times (\preceq)) \end{array} \right. \\
\equiv & \quad \{ \ (\preceq) \cdot nil = nil \text{ (4.17); linearity} \ \} \\
 & (\preceq) \cdot cons \cup (\preceq) \cdot nil \subseteq cons \cdot (id \times (\preceq)) \cup nil \cdot (id \times (\preceq)) \\
\equiv & \quad \{ \ \cup\text{-universal property; } (\preceq) \cdot nil = nil \ \} \\
 & \left\{ \begin{array}{l} (\preceq) \cdot cons \subseteq cons \cdot (id \times (\preceq)) \cup nil \cdot (id \times (\preceq)) \\ nil \subseteq cons \cdot (id \times (\preceq)) \cup nil \cdot (id \times (\preceq)) \end{array} \right. \\
\equiv & \quad \{ \ \subseteq\text{-transitivity since } nil \cdot R \subseteq nil \text{ and } (\preceq) \cdot cons = cons \cdot (id \times (\preceq)) \cup nil \text{ (4.17)} \ \} \\
 & True
\end{array}
$$

□

Proving antisymmetry relies on defining the converse of the order through a reasoning similar to that of Oliveira (2022),

$$\begin{cases} y \succcurlyeq [\,] \Leftrightarrow \mathit{True} \\ y \succcurlyeq (h:t) \Leftrightarrow \langle \exists\, z : y = (h:z) : z \succcurlyeq t \rangle \end{cases}$$

$$\equiv \qquad \{ \text{ going pointfree } \}$$

$$\begin{cases} (\succcurlyeq) \cdot \mathit{nil} = \top \\ (\succcurlyeq) \cdot \mathit{cons} = \mathit{cons} \cdot (\mathit{id} \times (\succcurlyeq)) \end{cases}$$

$$\equiv \qquad \{ \text{ catamorphisms } \}$$

$$(\succcurlyeq) = \llparenthesis [\top , \mathit{cons}] \rrparenthesis$$

according to which $(\preccurlyeq) = (\succcurlyeq)^{\circ}$. The proof of antisymmetry is thereafter analogous to the one given earlier concerning the *less than or equal* and the *greater than or equal* partial orders on natural numbers — compare (4.10) with

$$(\preccurlyeq) = \llparenthesis \mathsf{in}_{\mathsf{List}} \cup [\bot , \mathit{nil}] \rrparenthesis \qquad\qquad (\succcurlyeq) = \llparenthesis \mathsf{in}_{\mathsf{List}} \cup [\top , \bot] \rrparenthesis \tag{4.19}$$

Sublist partial order This order is defined as a relational catamorphism, cf.

$$(\sqsubseteq) = \llparenthesis [\mathit{nil} , \mathit{cons} \cup \pi_2] \rrparenthesis \tag{4.20}$$

$$\equiv \qquad \{ \text{ catamorphisms } \}$$

$$\begin{cases} (\sqsubseteq) \cdot \mathit{nil} = \mathit{nil} \\ (\sqsubseteq) \cdot \mathit{cons} = (\mathit{cons} \cup \pi_2) \cdot (\mathit{id} \times (\sqsubseteq)) \end{cases}$$

$$\equiv \qquad \{ \text{ left linearity; } \pi_2\text{-natural property } \}$$

$$\begin{cases} (\sqsubseteq) \cdot \mathit{nil} = \mathit{nil} \\ (\sqsubseteq) \cdot \mathit{cons} = \mathit{cons} \cdot (\mathit{id} \times (\sqsubseteq)) \cup (\sqsubseteq) \cdot \pi_2 \end{cases} \tag{4.21}$$

$$\equiv \qquad \{ \text{ going pointwise } \}$$

$$\begin{cases} s\; ((\sqsubseteq) \cdot \mathit{nil})\; x \Leftrightarrow s\; \mathit{nil}\; x \\ s\; ((\sqsubseteq) \cdot \mathit{cons})\; (h,t) \Leftrightarrow s\; (\mathit{cons} \cdot (\mathit{id} \times (\sqsubseteq)))\; (h,t) \vee s\; ((\sqsubseteq) \cdot \pi_2)\; (h,t) \end{cases}$$

$$\equiv \qquad \{ \text{ composition; } \pi_2\text{-definition } \}$$

$$\begin{cases} s \sqsubseteq [\,] \Leftrightarrow s = [\,] \\ s \sqsubseteq (h:t) \Leftrightarrow \langle \exists\, (x,y) : s = (x:y) : x = h \wedge y \sqsubseteq t \rangle \vee s \sqsubseteq t \end{cases}$$

$$\equiv \qquad \{\ \text{one-point}\ \}$$

$$\begin{cases} s \sqsubseteq [\,] \Leftrightarrow s = [\,] \\ s \sqsubseteq (h:t) \Leftrightarrow \langle \exists y : s = (h:y) : y \sqsubseteq t \rangle \vee s \sqsubseteq t \end{cases} \tag{4.22}$$

The proofs of reflexivity and transitivity are identical to those of the prefix order, cf.

$$id \subseteq (\sqsubseteq)$$

$$\equiv \qquad \{\ id_{\mathsf{T}} = \llparenthesis \mathsf{in}_{\mathsf{T}} \rrparenthesis ; \mathsf{in}_{\mathsf{T}} = [nil, cons]\ \}$$

$$\llparenthesis [nil, cons] \rrparenthesis \subseteq \llparenthesis [nil, cons \cup \pi_2] \rrparenthesis$$

$$\Leftarrow \qquad \{\ \text{monotonicity}\ \}$$

$$[nil, cons] \subseteq [nil, cons \cup \pi_2]$$

$$\equiv \qquad \{\ \text{coproducts}\ \}$$

$$\begin{cases} nil \subseteq nil \\ cons \subseteq cons \cup \pi_2 \end{cases}$$

$$\equiv \qquad \{\ \text{trivial; } R \subseteq R \cup S\ \}$$

$$True$$

$$\square$$

and

$$(\sqsubseteq) \cdot (\sqsubseteq) \subseteq (\sqsubseteq)$$

$$\equiv \qquad \{\ \text{definitions}\ \}$$

$$(\sqsubseteq) \cdot \llparenthesis [nil, cons \cup \pi_2] \rrparenthesis \subseteq \llparenthesis [nil, cons \cup \pi_2] \rrparenthesis$$

$$\Leftarrow \qquad \{\ \text{catamorphism fusion}\ \}$$

$$(\sqsubseteq) \cdot [nil, cons \cup \pi_2] \subseteq [nil, cons \cup \pi_2] \cdot (id + id \times (\sqsubseteq))$$

$$\equiv \qquad \{\ \text{fusion and absorption}\ \}$$

$$\begin{cases} (\sqsubseteq) \cdot nil \subseteq nil \\ (\sqsubseteq) \cdot (cons \cup \pi_2) \subseteq (cons \cup \pi_2) \cdot (id \times (\sqsubseteq)) \end{cases}$$

$$\equiv \qquad \{\ (\sqsubseteq) \cdot nil = nil \text{ (4.21); linearity}\ \}$$

$$(\sqsubseteq) \cdot cons \cup (\sqsubseteq) \cdot \pi_2 \subseteq cons \cdot (id \times (\sqsubseteq)) \cup \pi_2 \cdot (id \times (\sqsubseteq))$$

$$\equiv \qquad \{\ \cup\text{-universal property}\ \}$$

$$\begin{cases} (\sqsubseteq) \cdot cons \subseteq cons \cdot (id \times (\sqsubseteq)) \cup \pi_2 \cdot (id \times (\sqsubseteq)) \\ (\sqsubseteq) \cdot \pi_2 \subseteq cons \cdot (id \times (\sqsubseteq)) \cup \pi_2 \cdot (id \times (\sqsubseteq)) \end{cases}$$

$$\equiv \qquad \{ \ \pi_2\text{-natural property} \ \}$$

$$\begin{cases} (\sqsubseteq) \cdot cons \ \subseteq \ cons \cdot (id \times (\sqsubseteq)) \ \cup \ (\sqsubseteq) \cdot \pi_2 \\ (\sqsubseteq) \cdot \pi_2 \ \subseteq \ cons \cdot (id \times (\sqsubseteq)) \ \cup \ (\sqsubseteq) \cdot \pi_2 \end{cases}$$

$$\equiv \qquad \{ \ (\sqsubseteq) \cdot cons = cons \cdot (id \times (\sqsubseteq)) \ \cup \ (\sqsubseteq) \cdot \pi_2 \ \text{(4.21); trivial} \ \}$$

$$True$$

□

However, the method for proving antisymmetry does not follow the same scheme, since the definition of the converse of $(\sqsubseteq)$ is not a relational catamorphism:

$$\begin{cases} y \sqsupseteq [\,] \Leftrightarrow True \\ y \sqsupseteq (h\!:\!t) \Leftrightarrow \langle \exists (a,b) \ : \ y = (a\!:\!b) \ : \ (a = h \wedge b \sqsupseteq t) \vee b \sqsupseteq (h\!:\!t) \rangle \end{cases}$$

$$\equiv \qquad \{ \ \text{going pointfree} \ \}$$

$$\begin{cases} (\sqsupseteq) \cdot nil = \top \\ (\sqsupseteq) \cdot cons = cons \cdot ((id \times (\sqsupseteq)) \ \cup \ \pi_2^\circ \cdot (\sqsupseteq) \cdot cons) \end{cases}$$

$$\equiv \qquad \{ \ \text{right linearity} \ \}$$

$$\begin{cases} (\sqsupseteq) \cdot nil = \top \\ (\sqsupseteq) \cdot cons = cons \cdot (id \times (\sqsupseteq)) \ \cup \ cons \cdot \pi_2^\circ \cdot (\sqsupseteq) \cdot cons \end{cases}$$

Because this calls for a concept that extends that of a catamorphism, known as *hylomorphism* (Bird and de Moor, 1997), this proof is left for future work, see Section 7.2.

Suffix partial order Another order that will be required in the sequel is the suffix order, namely when calculating functions such as $drop$ and $dropWhile$. Taking as starting point

$$\begin{cases} s \preccurlyeq [\,] \Leftrightarrow s = [\,] \\ s \preccurlyeq (h\!:\!t) \Leftrightarrow s = (h\!:\!t) \vee s \preccurlyeq t \end{cases} \tag{4.23}$$

one easily seems that it is another example of a non-catamorphic definition:

$$\begin{cases} s \preccurlyeq [\,] \Leftrightarrow s = [\,] \\ s \preccurlyeq (h\!:\!t) \Leftrightarrow s = (h\!:\!t) \vee s \preccurlyeq t \end{cases}$$

$$\equiv \qquad \{ \ \text{go pointfree} \ \}$$

$$\begin{cases} (\preccurlyeq) \cdot nil = nil \\ (\preccurlyeq) \cdot cons = cons \ \cup \ (\preccurlyeq) \cdot \pi_2 \end{cases} \tag{4.24}$$

$$\equiv \qquad \{ \text{ coproducts } \}$$

$$(\preccurlyeq) \cdot \mathsf{in}_{\mathsf{List}} = [nil, cons \; \cup \; (\preccurlyeq) \cdot \pi_2] \tag{4.25}$$

For the same reasons as above, the proofs for reflexivity, transitivity, and antisymmetry require *hylomorphisms* and are deferred for future work.

4.2 Predicates in Galois connections

Some GCs that will be subject of study in Section 4.3 deal with predicates in a very particular pattern — when expressed via pointwise

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

z needs to satisfy some given predicate p. Thus, one could simply write

$$p\,z \wedge f\,z \leqslant x \;\equiv\; p\,z \wedge z \sqsubseteq g\,x$$

However, this approach is not particularly suitable for indirect equality proofs, nor does it resemble the typical format of a GC. Nevertheless, this is an easily surmountable issue — observe that in these cases, the upper adjoint yields values that satisfy p. Thus, it suffices to restrict the respective order to values that satisfy p, and the lower adjoint merely receives z's that satisfy the given predicate, that is,

$$(f \cdot cast_p)\,z \leqslant x \;\equiv\; z \sqsubseteq_p \; g\,x \tag{4.26}$$

where $cast_p$ type-casts A_p (type A restricted to the elements that satisfy predicate p) to its general type A. Note that, since the orders are parametric, the subscript can be omitted.

For instance, consider the function $takeWhile$ specified as

$takeWhile\; p\; l$ yields the longest prefix of a sequence whose elements satisfy predicate p.

Recurring to (4.26), it can be formally specified as

$$cast_{all\,p}\,z \preccurlyeq l \;\equiv\; z \preccurlyeq takeWhile\,p\,l \tag{4.27}$$

Someone might complain it can be quite tiresome to handle two "different" types. But what does this function $cast$ practically mean? Clearly,

$$cast_p\,z\,R\,y \;\equiv\; p\,z \wedge z\,R\,y \tag{4.28}$$

Thus, (4.27) could be rewritten as

$$all\,p\,z \wedge z \preccurlyeq l \;\equiv\; z \preccurlyeq takeWhile\,p\,l \tag{4.29}$$

which reads very effectively "the largest prefix whose elements satisfy predicate p". Generally, equivalence (4.26) is rewritten as, dropping the subscript,

$$p\,z \wedge f\,z \leqslant x \equiv z \sqsubseteq g\,x \tag{4.30}$$

Finally, concerning to relation equalities, what can be said about this type-casting? First, GC (4.26) is directly given in pointfree as

$$cast_p^\circ \cdot f^\circ \cdot (\leqslant) = (\sqsubseteq_p)^\circ \cdot g \tag{4.31}$$

However, when calculating it from (4.30), one obtains

$$\begin{aligned}
& p\,z \wedge f\,z \leqslant x \Leftrightarrow z \sqsubseteq g\,x \\
\equiv \quad & \{ \text{ one-point and trading } \} \\
& \langle \exists a : z = a \wedge p\,a : f\,a \leqslant x \rangle \Leftrightarrow z \sqsubseteq g\,x \\
\equiv \quad & \{ \text{ coreflexives } \} \\
& \langle \exists a : z\,\phi_p\,a : f\,a \leqslant x \rangle \Leftrightarrow z \sqsubseteq g\,x \\
\equiv \quad & \{ \text{ going pointfree } \} \\
& \phi_p \cdot f^\circ \cdot (\leqslant) = (\sqsubseteq) \cdot g
\end{aligned} \tag{4.32}$$

So, $cast_p^\circ$ operates just like ϕ_p — it enforces the coreflexive in the "lower" type. This result will prove very useful when dealing with pointfree specifications in Chapter 5. For instance, $takeWhile\,p$ can be specified via GC as a relational equality by

$$cast_{all\,p}^\circ \cdot (\preccurlyeq) = (\preccurlyeq) \cdot takeWhile\,p \tag{4.33}$$

which can be rewritten as

$$\phi_{all\,p} \cdot (\preccurlyeq) = (\preccurlyeq) \cdot takeWhile\,p \tag{4.34}$$

4.3 Repertoire of functions calculated from GCs

In order to evaluate the scope of the proposed correct-by-construction method, a series of functions taken from the Haskell standard libraries have been subject to the check whether they are specifiable (and implementable) by GCs or not. Positive examples are given first, along with their pointwise derivations. Counter-examples will be given later.

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

$$z \times y \leqslant x \;\equiv\; z \leqslant x \div y \tag{4.35}$$

As already calculated in Section 2.1:

```
-- Calculated implementation of div
div     :: (Ord t, Num a, Num t) => t -> t -> a
div x y = if x < y then 0 else 1 + div (x - y) y
```

- *Given a partial order $(\leqslant)$, $\mathit{min}\ m\ n$ yields the greatest value which is simultaneously 'less or equal' than m and n according to the given order.*

$$z \leqslant m \wedge z \leqslant n \;\equiv\; z \leqslant \mathit{min}\ m\ n \tag{4.36}$$

  - if $m \leqslant n$

$$\begin{array}{ll}
 & z \leqslant \mathit{min}\ m\ n \\
\equiv & \{\ \text{GC (4.36)}\ \} \\
 & z \leqslant m \wedge z \leqslant n \\
\equiv & \{\ \text{since } m \leqslant n,\ z \leqslant m \text{ entails } z \leqslant n\ \} \\
 & z \leqslant m \\
:: & \{\ \text{indirect equality over } (\leqslant) \text{ partial order}\ \} \\
 & \mathit{min}\ m\ n = m
\end{array}$$

  - if $\neg\,(m \leqslant n)$

$$\begin{array}{ll}
 & z \leqslant \mathit{min}\ m\ n \\
\equiv & \{\ \text{GC (4.36)}\ \} \\
 & z \leqslant m \wedge z \leqslant n \\
\equiv & \{\ \text{since } \neg\,(m \leqslant n) = n < m,\ z \leqslant n \text{ entails } z < m \text{ which entails } z \leqslant m\ \} \\
 & z \leqslant n \\
:: & \{\ \text{indirect equality over } (\leqslant) \text{ partial order}\ \} \\
 & \mathit{min}\ m\ n = n
\end{array}$$

```
-- Calculated implementation of min
min     :: Ord a => a -> a -> a
min m n = if m <= n then m else n
```

- *Given a partial order* $(\geqslant)$, *max* $m\ n$ *yields the greatest value which is simultaneously 'greater or equal' than* m *and* n *according to the given order.*

  $$z \geqslant m \wedge z \geqslant n \ \equiv\ z \geqslant \mathit{max}\ m\ n \tag{4.37}$$

  Its calculation is analogous to *min* (4.36).

  ```
  -- Calculated implementation of max
  max      :: Ord a => a -> a -> a
  max m n = if m >= n then m else n
  ```

- *Given a partial ordering* $(\leqslant)$, *minimum* l *yields the greatest value which is 'less or equal' than all elements of* l *according to the given order.*

  $$\mathit{all}\ (z\ \leqslant)\ l \ \equiv\ z \leqslant \mathit{minimum}\ l \tag{4.38}$$

  - $\mathit{minimum}\ [h]$

    $$\begin{array}{cl} & z \leqslant \mathit{minimum}\ [h] \\ \equiv & \quad \{\ \text{GC (4.38)}\ \} \\ & \mathit{all}\ (z\ \leqslant)\ [h] \\ \equiv & \quad \{\ \text{definition of } \mathit{all}\ \} \\ & z \leqslant h \\ :: & \quad \{\ \text{indirect equality over } (\leqslant) \text{ partial order}\ \} \\ & \mathit{minimum}\ [h] = h \end{array}$$

  - $\mathit{minimum}\ (h : t)$

    $$\begin{array}{cl} & z \leqslant \mathit{minimum}\ (h : t) \\ \equiv & \quad \{\ \text{GC (4.38)}\ \} \\ & \mathit{all}\ (z\ \leqslant)\ (h : t) \\ \equiv & \quad \{\ \text{definitions of } \mathit{all}\ \} \\ & z \leqslant h \wedge \mathit{all}\ (z\ \leqslant)\ t \\ \equiv & \quad \{\ \text{GC (4.38)}\ \} \\ & z \leqslant h \wedge z \leqslant \mathit{minimum}\ t \\ \equiv & \quad \{\ \text{GC (4.36)}\ \} \end{array}$$

$$z \leqslant min\ h\ (minimum\ t)$$

$::$ $\{$ indirect equality over $(\leqslant)$ partial order $\}$

$$minimum\ (h:t) = min\ h\ (minimum\ t)$$

```
-- Calculated implementation of minimum
minimum       :: Ord a => [a] -> a
minimum [h]   = h
minimum (h:t) = min h (minimum t)
```

- *Given a partial ordering* $(\geqslant)$*, maximum l yields the smallest value which is 'greater or equal' than all elements of l according to the given order.*

$$all\ (\geqslant z)\ l \ \equiv\ z \geqslant maximum\ l \tag{4.39}$$

Calculation is analogous to that of *minimum* (4.38).

```
-- Calculated implementation of maximum
maximum       :: Ord a => [a] -> a
maximum [h]   = h
maximum (h:t) = max h (maximum t)
```

- *take n x yields the longest prefix of a sequence up to some given length n.*

$$\text{length}\ z \leqslant n \wedge z \preccurlyeq x \ \equiv\ z \preccurlyeq take\ n\ x \tag{4.40}$$

Already calculated in Section 2.1, but this time given on its original (curried) form.

```
-- Calculated implementation of take
take              :: Int -> [a] -> [a]
take 0     _      = []
take _     []     = []
take (n+1) (h:xs) = h: take n xs
```

- *takeWhile p l yields the longest prefix of a sequence whose elements satisfy predicate p.*

$$cast_{all\ p}\ z \preccurlyeq l \ \equiv\ z \preccurlyeq takeWhile\ p\ l \tag{4.41}$$

  - *takeWhile* $p\ [\,]$

$$z \preccurlyeq takeWhile\ p\ [\,]$$

$\equiv$ $\{$ GC (4.41) $\}$

$$cast_{all\ p}\ z \preccurlyeq [\,]$$

$\equiv \qquad \{ \text{ type-casting (4.28) } \}$

$z \preccurlyeq [\,]$

$:: \qquad \{ \text{ indirect equality over list prefixing } (\preccurlyeq) \}$

$\mathit{takeWhile}\ p\ [\,] = [\,]$

- $\mathit{takeWhile}\ p\ (h : t)$

$z \preccurlyeq \mathit{takeWhile}\ p\ (h : t)$

$\equiv \qquad \{ \text{ GC (4.41) } \}$

$\mathit{cast}_{\mathit{all}\ p}\ z \preccurlyeq l$

$\equiv \qquad \{ \text{ type-casting (4.28) } \}$

$\mathit{all}\ p\ z \wedge z \preccurlyeq l$

$\equiv \qquad \{ (\preccurlyeq) \text{ definition (4.18) } \}$

$\mathit{all}\ p\ z \wedge (\langle \exists b : z = (h : b) : b \preccurlyeq t \rangle \vee z = [\,])$

$\equiv \qquad \{ \text{ distribution } \}$

$(\mathit{all}\ p\ z \wedge z = [\,]) \vee \langle \exists b : z = (h : b) : \mathit{all}\ p\ z \wedge (b \preccurlyeq t) \rangle$

$\equiv \qquad \{ \mathit{all}\ p\ z \Leftarrow z = [\,] \}$

$z = [\,] \vee \langle \exists b : z = (h : b) : \mathit{all}\ p\ z \wedge (b \preccurlyeq t) \rangle$

$\equiv \qquad \{ \mathit{all}\ p\ (x : xs) \equiv p\ x \wedge \mathit{all}\ p\ xs \}$

$z = [\,] \vee \langle \exists b : z = (h : b) : p\ h \wedge \mathit{all}\ p\ b \wedge b \preccurlyeq t \rangle$

$\equiv \qquad \{ \text{ type-casting (4.28) } \}$

$z = [\,] \vee \langle \exists b : z = (h : b) : p\ h \wedge \mathit{cast}_{\mathit{all}\ p}\ b \preccurlyeq t \rangle$

$\equiv \qquad \{ \text{ GC (4.41) } \}$

$z = [\,] \vee \langle \exists b : z = (h : b) : p\ h \wedge b \preccurlyeq \mathit{takeWhile}\ p\ t \rangle$

  - if $p\ h$

$z = [\,] \vee \langle \exists b : z = (h : b) : b \preccurlyeq \mathit{takeWhile}\ p\ t \rangle$

$\equiv \qquad \{ (\preccurlyeq) \text{ definition (4.18) } \}$

$z \preccurlyeq h : \mathit{takeWhile}\ p\ t$

$:: \qquad \{ \text{ indirect equality over list prefixing } (\preccurlyeq) \}$

$\mathit{takeWhile}\ p\ (h : t) = h : \mathit{takeWhile}\ p\ t$

⋆ if $\neg\,(p\;h)$

$$z = [\,]$$

$\equiv$ { $(\preceq)$ definition (4.18) }

$$z \preceq [\,]$$

:: { indirect equality over list prefixing $(\preceq)$ }

$$\mathit{takeWhile}\;p\;(h:t) = [\,]$$

```
-- Calculated implementation of takeWhile
takeWhile         :: (a -> Bool) -> [a] -> [a]
takeWhile p []    = []
takeWhile p (h:t) = if p h then h : takeWhile p t else []
```

- *drop n x yields the longest suffix of a sequence whose length does not exceed* length $x - n$*.*

$$\text{length}\;z \leqslant \text{length}\;x - n \wedge z \preccurlyeq x \;\equiv\; z \preccurlyeq \mathit{drop}\;n\;x \tag{4.42}$$

  - $\mathit{drop}\;0\;x$

$$z \preccurlyeq \mathit{drop}\;0\;x$$

$\equiv$ { GC (4.42) }

$$\text{length}\;z \leqslant \text{length}\;x \wedge z \preccurlyeq x$$

$\equiv$ { $z \preccurlyeq x$ entails length $z \leqslant$ length }

$$z \preccurlyeq x$$

:: { indirect equality over list suffixing $(\preccurlyeq)$ }

$$\mathit{drop}\;0\;x = x$$

  - $\mathit{drop}\;n\;[\,]$

$$z \preccurlyeq \mathit{drop}\;n\;[\,]$$

$\equiv$ { GC (4.42) }

$$\text{length}\;z \leqslant n \wedge z \preccurlyeq [\,]$$

$\equiv$ { $z \preccurlyeq [\,] \Leftrightarrow z = [\,]$ which entails length $z \leqslant n$ }

$$z \preccurlyeq [\,]$$

:: { indirect equality over list suffixing $(\preccurlyeq)$ }

$$\mathit{drop}\;n\;[\,] = [\,]$$

- $\mathit{drop}\ (n+1)\ (h:t)$

$$
\begin{array}{ll}
 & z \preccurlyeq \mathit{drop}\ (n+1)\ (h:t) \\
\equiv & \qquad \{\ \text{GC (4.42)}\ \} \\
 & \text{length}\ z \leqslant \text{length}\ (h:t) - (n+1)\ \wedge\ z \preccurlyeq\ (h:t) \\
\equiv & \qquad \{\ \text{length definition; arithmetics; suffix definition (4.23)}\ \} \\
 & \text{length}\ z \leqslant \text{length}\ t - n\ \wedge\ (z = (h:t)\ \vee\ z \preccurlyeq\ t) \\
\equiv & \qquad \{\ \text{distribution}\ \} \\
 & (\text{length}\ z \leqslant \text{length}\ t - n\ \wedge\ z = (h:t))\ \vee\ (\text{length}\ z \leqslant \text{length}\ t - n\ \wedge\ z \preccurlyeq\ t) \\
\equiv & \qquad \{\ \text{GC (4.42)}\ \} \\
 & (\text{length}\ t + 1 \leqslant \text{length}\ t - n)\ \vee\ (z \preccurlyeq\ \mathit{drop}\ n\ t) \\
\equiv & \qquad \{\ 1 \leqslant -n \text{ is false}\ \} \\
 & z \preccurlyeq\ \mathit{drop}\ n\ t \\
:: & \qquad \{\ \text{indirect equality over list suffixing } (\preccurlyeq)\ \} \\
 & \mathit{drop}\ (n+1)\ (h:t) = \mathit{drop}\ n\ t
\end{array}
$$

```
-- Calculated implementation of drop
drop                :: Int -> [a] -> [a]
drop 0      x       = x
drop n      []      = []
drop (n+1) (h:t) = drop n t
```

Assert $\mathit{drop}\ (\text{length}\ l)\ l = [\,]$

$$
\begin{array}{ll}
 & z \preccurlyeq\ \mathit{drop}\ (\text{length}\ l)\ l \\
\equiv & \qquad \{\ \text{GC (4.42)}\ \} \\
 & \text{length}\ z \leqslant \text{length}\ l - \text{length}\ l\ \wedge\ z \preccurlyeq\ [\,] \\
\equiv & \qquad \{\ \text{trivial}\ \} \\
 & \text{length}\ z \leqslant 0\ \wedge\ z \preccurlyeq\ [\,] \\
\equiv & \qquad \{\ \text{length}\ z \leqslant 0 \Leftrightarrow z = [\,] \Leftrightarrow z \preccurlyeq\ [\,]\ \} \\
 & z \preccurlyeq\ [\,] \\
:: & \qquad \{\ \text{indirect equality over list suffixing } (\preccurlyeq)\ \} \\
 & \mathit{drop}\ (\text{length}\ l)\ l = [\,]
\end{array}
$$

□

Assert $drop\ m\ (drop\ n\ l) = drop\ (m+n)\ l$

$$z \preccurlyeq drop\ m\ (drop\ n\ l)$$

$$\equiv \quad \{\ \text{GC (4.42)}\ \}$$

$$\text{length}\ z \leqslant \text{length}\ (drop\ n\ l) - m \wedge z \preccurlyeq (drop\ n\ l)$$

$$\equiv \quad \{\ \text{GC (4.42) and associativity}\ \}$$

$$(\text{length}\ z \leqslant \text{length}\ (drop\ n\ l) - m \wedge \text{length}\ z \leqslant \text{length}\ l - n) \wedge z \preccurlyeq l$$

$$\equiv \quad \{\ \text{length}\ (drop\ n\ l) \leqslant \text{length}\ l - n\ \}$$

$$(\text{length}\ z \leqslant \text{length}\ l - n - m \wedge \text{length}\ z \leqslant \text{length}\ l - n) \wedge z \preccurlyeq l$$

$$\equiv \quad \{\ \text{length}\ z \leqslant \text{length}\ l - n - m \Rightarrow \text{length}\ z \leqslant \text{length}\ l - n\ \}$$

$$\text{length}\ z \leqslant \text{length}\ l - n - m \wedge z \preccurlyeq l$$

$$\equiv \quad \{\ \text{trivial}\ \}$$

$$\text{length}\ z \leqslant \text{length}\ l - (m+n) \wedge z \preccurlyeq l$$

$$\equiv \quad \{\ \text{GC (4.42)}\ \}$$

$$z \preccurlyeq drop\ (m+n)\ l$$

$$:: \quad \{\ \text{indirect equality over list suffixing}\ (\preccurlyeq)\ \}$$

$$drop\ m\ (drop\ n\ l) = drop\ (m+n)\ l$$

□

- *dropWhile p l yields the longest suffix of a sequence whose head fails to satisfy predicate p*

  For that, consider the following function

```
headFails          :: (t -> Bool) -> [t] -> Bool
headFails p []     = True
headFails p (h:t) = not p h
```

This way, $dropWhile\ p\ l$ is specified via the following GC:

$$cast_{headFails\ p}\ z \preccurlyeq l \equiv z \preccurlyeq dropWhile\ p\ l \qquad (4.43)$$

  - $dropWhile\ [\,]$

$$z \preccurlyeq dropWhile\ p\ [\,]$$

$$
\begin{aligned}
&\equiv \qquad \{\ \text{GC (4.43)}\ \} \\
&\qquad \mathit{cast}_{\mathit{headFails}\ p}\ z \preccurlyeq [\,] \\
&\equiv \qquad \{\ \text{type-casting (4.28)}\ \} \\
&\qquad \mathit{headFais}\ p\ [\,] \wedge z \preccurlyeq [\,] \\
&\equiv \qquad \{\ \mathit{headFais}\ p\ [\,] = \mathit{True}\ \} \\
&\qquad z \preccurlyeq [\,] \\
&:: \qquad \{\ \text{indirect equality over list suffixing } (\preccurlyeq)\ \} \\
&\qquad \mathit{dropWhile}\ p\ [\,] = [\,]
\end{aligned}
$$

- $\mathit{dropWhile}\ (h : t)$

$$
\begin{aligned}
&\qquad z \preccurlyeq \mathit{dropWhile}\ p\ (h : t) \\
&\equiv \qquad \{\ \text{GC (4.43)}\ \} \\
&\qquad \mathit{cast}_{\mathit{headFails}\ p}\ z \preccurlyeq (h : t) \\
&\equiv \qquad \{\ \text{type-casting (4.28)}\ \} \\
&\qquad \mathit{headFails}\ p\ z \wedge z \preccurlyeq (h : t) \\
&\equiv \qquad \{\ \text{suffix definition (4.23)}\ \} \\
&\qquad \mathit{headFails}\ p\ z \wedge (z = (h : t) \vee z \preccurlyeq t) \\
&\equiv \qquad \{\ \text{distribution}\ \} \\
&\qquad \mathit{headFails}\ p\ (h : t) \vee (\mathit{headFails}\ p\ z \wedge z \preccurlyeq t) \\
&\equiv \qquad \{\ \text{definition of } \mathit{headFails}\ \} \\
&\qquad \neg\, (p\ h) \vee (\mathit{headFails}\ p\ z \wedge z \preccurlyeq t)
\end{aligned}
$$

  - ⋆ if $p\ h$

$$
\begin{aligned}
&\qquad \mathit{headFails}\ p\ z \wedge z \preccurlyeq t \\
&\equiv \qquad \{\ \text{type-casting (4.28)}\ \} \\
&\qquad \mathit{cast}_{\mathit{headFails}\ p}\ z \preccurlyeq t \\
&\equiv \qquad \{\ \text{GC (4.43)}\ \} \\
&\qquad z \preccurlyeq \mathit{dropWhile}\ p\ t \\
&:: \qquad \{\ \text{indirect equality over list suffixing } (\preccurlyeq)\ \} \\
&\qquad \mathit{dropWhile}\ p\ (h : t) = \mathit{dropWhile}\ p\ t
\end{aligned}
$$

⋆ if $\neg\,(p\ h)$

$$\neg\,(p\ h) \vee (\mathit{headFails}\ p\ z \wedge z \preccurlyeq t)$$

$\equiv$ { logic }

$$\mathit{True}$$

$\equiv$ { going back to the third step knowing $\neg\,(p\ h)$ }

$$\mathit{headFails}\ p\ z \wedge z \preccurlyeq (h : t) \wedge \neg\,(p\ h)$$

$\equiv$ { both $(h : t)$ and z meet $\mathit{headFails}\ p$ }

$$z \preccurlyeq (h : t)$$

$::$ { indirect equality over list suffixing $(\preccurlyeq)$ }

$$\mathit{dropWhile}\ p\ (h : t) = (h : t)$$

```
-- Calculated implementation of dropWhile
dropWhile          :: (a -> Bool) -> [a] -> [a]
dropWhile p []     = []
dropWhile p (h:t) = if p h then dropWhile p t else (h:t)
```

- *filter p l yields the longest sublist of a sequence whose elements satisfy predicate p.*

$$\mathit{cast}_{\mathit{all}\ p}\ z \sqsubseteq l \;\equiv\; z \sqsubseteq \mathit{filter}\ p\ l \tag{4.44}$$

– $\mathit{filter}\ [\,]$

$$z \sqsubseteq \mathit{filter}\ p\ [\,]$$

$\equiv$ { GC (4.44) }

$$\mathit{cast}_{\mathit{all}\ p}\ z \sqsubseteq [\,]$$

$\equiv$ { type-casting (4.28) }

$$\mathit{all}\ p\ z \wedge z \sqsubseteq [\,]$$

$\equiv$ { $\mathit{all}\ p\ [\,] \Leftarrow z \sqsubseteq [\,]$ }

$$z \sqsubseteq [\,]$$

$::$ { indirect equality over sublist partial order $(\sqsubseteq)$ }

$$\mathit{filter}\ p\ [\,] = [\,]$$

– $\mathit{filter}\ (h : t)$

$$z \sqsubseteq \mathit{filter}\ p\ (h : t)$$

$$\equiv \qquad \{ \text{ GC (4.44); } \}$$

$$cast_{all\,p}\; z \sqsubseteq (h\!:\!t)$$

$$\equiv \qquad \{ \text{ type-casting (4.28); sublist definition (4.22) } \}$$

$$all\,p\,z \wedge (z \sqsubseteq t \vee \langle \exists b : z = (h\!:\!b) : b \sqsubseteq t \rangle)$$

$$\equiv \qquad \{ \text{ distribution } \}$$

$$(all\,p\,z \wedge z \sqsubseteq t) \vee \langle \exists b : z = (h\!:\!b) : b \sqsubseteq t \wedge all\,p\,(h\!:\!b) \rangle$$

$$\equiv \qquad \{ \text{ type-casting (4.28) } \}$$

$$cast_{all\,p}\; z \sqsubseteq t \vee \langle \exists b : z = (h\!:\!b) : b \sqsubseteq t \wedge all\,p\,(h\!:\!b) \rangle$$

$$\equiv \qquad \{ \text{ GC (4.44) } \}$$

$$z \sqsubseteq filter\,p\,t \vee \langle \exists b : z = (h\!:\!b) : b \sqsubseteq t \wedge all\,p\,(h\!:\!b) \rangle$$

$$\equiv \qquad \{ \; all\,p\,(x\!:\!xs) \equiv p\,x \wedge all\,p\,xs\text{; association } \}$$

$$z \sqsubseteq filter\,p\,t \vee \langle \exists b : z = (h\!:\!b) : b \sqsubseteq t \wedge p\,h \wedge all\,p\,b \rangle$$

$$\equiv \qquad \{ \text{ type-casting (4.28) } \}$$

$$z \sqsubseteq filter\,p\,t \vee \langle \exists b : z = (h\!:\!b) : p\,h \wedge cast_{all\,p}\; b \sqsubseteq t \rangle$$

$$\equiv \qquad \{ \text{ GC (4.44) } \}$$

$$z \sqsubseteq filter\,p\,t \vee \langle \exists b : z = (h\!:\!b) : p\,h \wedge b \sqsubseteq filter\,p\,t \rangle$$

- ⋆ if $p\,h$

$$z \sqsubseteq filter\,p\,t \vee \langle \exists b : z = (h\!:\!b) : b \sqsubseteq filter\,p\,t \rangle$$

$$\equiv \qquad \{ \text{ } (\sqsubseteq) \text{ definition (4.22) } \}$$

$$z \sqsubseteq h\!:\!filter\,p\,t$$

$$:: \qquad \{ \text{ indirect equality over sublist partial order } (\sqsubseteq) \}$$

$$filter\,p\,(h\!:\!t) = h\!:\!filter\,p\,t$$

- ⋆ if $\neg\,(p\,h)$

$$z \sqsubseteq filter\,p\,t$$

$$:: \qquad \{ \text{ indirect equality over sublist partial order } (\sqsubseteq) \}$$

$$filter\,p\,(h\!:\!t) = filter\,p\,t$$

```
-- Calculated implementation of filter
filter          :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (h:t) = if p h then h : filter p t else filter p t
```

- *$zip\ l_1\ l_2$ yields the longest prefix of pairs whose first elements of its pairs form a prefix of l_1 and the second ones form a prefix of l_2.*

$$\text{map}\ \pi_1\ z \preccurlyeq l_1 \wedge \text{map}\ \pi_2\ z \preccurlyeq l_2 \equiv z \preccurlyeq zip\ l_1\ l_2 \tag{4.45}$$

  - $zip\ l_1\ [\,]$

$$\begin{array}{ll}
 & z \preccurlyeq zip\ l_1\ [\,] \\
\equiv & \qquad \{\ \text{GC (4.45)}\ \} \\
 & \text{map}\ \pi_1\ z \preccurlyeq l_1 \wedge \text{map}\ \pi_2\ z \preccurlyeq [\,] \\
\equiv & \qquad \{\ \text{prefix definition (4.18)}\ \} \\
 & \text{map}\ \pi_1\ z \preccurlyeq l_1 \wedge \text{map}\ \pi_2\ z = [\,] \\
\equiv & \qquad \{\ z = [\,] \text{ by map } \pi_2\ z = [\,] \text{ and map definition; prefix definition (4.18) again }\ \} \\
 & z \preccurlyeq [\,] \\
:: & \qquad \{\ \text{indirect equality over prefix partial order } (\preccurlyeq)\ \} \\
 & zip\ l_1\ [\,] = [\,]
\end{array}$$

  - $zip\ [\,]\ l_2 = [\,]$

    Analogous to the previous calculation.

  - $zip\ (h_1 : t_1)\ (h_2 : t_2)$

$$\begin{array}{ll}
 & z \preccurlyeq zip\ (h_1 : t_1)\ (h_2 : t_2) \\
\equiv & \qquad \{\ \text{GC (4.45)}\ \} \\
 & \text{map}\ \pi_1\ z \preccurlyeq (h_1 : t_1) \wedge \text{map}\ \pi_2\ z \preccurlyeq (h_2 : t_2) \\
\equiv & \qquad \{\ \text{prefix definition (4.18) twice}\ \} \\
 & \left\{ \begin{array}{l} \langle \exists\, y : \text{map}\ \pi_1\ z = (h_1 : y) : y \preccurlyeq t_1 \rangle \vee \text{map}\ \pi_1\ z = [\,] \\ \langle \exists\, y : \text{map}\ \pi_2\ z = (h_2 : y) : y \preccurlyeq t_2 \rangle \vee \text{map}\ \pi_2\ z = [\,] \end{array} \right. \\
\equiv & \qquad \{\ \text{map definition; distribution}\ \} \\
 & \left\{ \begin{array}{l} (\langle \exists\, y : \text{map}\ \pi_1\ z = (h_1 : y) : y \preccurlyeq t_1 \rangle) \\ (\langle \exists\, y : \text{map}\ \pi_2\ z = (h_2 : y) : y \preccurlyeq t_2 \rangle) \end{array} \right. \quad \vee\ z = [\,] \\
\equiv & \qquad \{\ \text{map} f\ (h : t) = f\ h : \text{map} f\ t\ \} \\
 & \left\{ \begin{array}{l} \langle \exists\, y : \text{map}\ \pi_1\ (tail\ z) = y \wedge \pi_1\ (head\ z) = h_1 : y \preccurlyeq t_1 \rangle \\ \langle \exists\, y : \text{map}\ \pi_2\ (tail\ z) = y \wedge \pi_2\ (head\ z) = h_2 : y \preccurlyeq t_2 \rangle \end{array} \right. \quad \vee\ z = [\,]
\end{array}$$

$$\equiv \qquad \{ \text{ one-point rule twice } \}$$

$$\left\{\begin{array}{l} \pi_1\ (head\ z) = h_1\ \wedge\ \mathsf{map}\ \pi_1\ (tail\ z)\ \preccurlyeq\ t_1 \\ \pi_2\ (head\ z) = h_2\ \wedge\ \mathsf{map}\ \pi_2\ (tail\ z)\ \preccurlyeq\ t_2 \end{array}\right. \vee\ z = [\,]$$

$$\equiv \qquad \{ \text{ products } \}$$

$$\left(\left\{\begin{array}{l} \mathsf{map}\ \pi_1\ (tail\ z)\ \preccurlyeq\ t_1 \\ \mathsf{map}\ \pi_2\ (tail\ z)\ \preccurlyeq\ t_2 \end{array}\right. \wedge\ head\ z = (h_1,h_2)\right)\ \vee\ z = [\,]$$

$$\equiv \qquad \{ \text{ one-point rule } \}$$

$$\langle \exists\, y\ :\ z = ((h_1,h_2):y)\ :\ \mathsf{map}\ \pi_1\ y\ \preccurlyeq\ t_1\ \wedge\ \mathsf{map}\ \pi_2\ y\ \preccurlyeq\ t_2 \rangle\ \vee\ z = [\,]$$

$$\equiv \qquad \{ \text{ GC (4.45) } \}$$

$$\langle \exists\, y\ :\ z = ((h_1,h_2):y)\ :\ y\ \preccurlyeq\ zip\ t_1\ t_2 \rangle\ \vee\ z = [\,]$$

$$\equiv \qquad \{ \text{ prefix definition (4.18) } \}$$

$$z\ \preccurlyeq\ (h_1,h_2):zip\ t_1\ t_2$$

$$:: \qquad \{ \text{ indirect equality over prefix partial order } (\preccurlyeq) \}$$

$$zip\ (h_1:t_1)\ (h_2:t_2) = (h_1,h_2):zip\ t_1\ t_2$$

```
-- Calculated implementation of zip
zip :: [a] -> [b] -> [(a,b)]
zip l1      []      = []
zip []      l2      = []
zip (h1:t1) (h2:t2) = (h1, h2) : zip t1 t2
```

- *replicate n x yields the sequence of length n where all elements are equal to x.*

$$(\mathsf{length} \cdot cast_{all\ (\equiv x)})\ z = n\ \equiv\ z = replicate\ n\ x \tag{4.46}$$

Although (4.46) does not directly fit into the GC format, it is easy to see that once *replicate* is flipped it does so:

$$\underbrace{(\mathsf{length} \cdot cast_{all\ (\equiv x)})}_{\text{lower adjoint}}\ z = n\ \equiv\ z = \underbrace{flip\ replicate\ x}_{\text{upper adjoint}}\ n \tag{4.47}$$

So isomorphism *flip* plays the role of *curry* in the case of *take* earlier on. Thus, it may be calculated as follows:

$$z\ \preccurlyeq\ replicate\ n\ x$$

$$\equiv \qquad \{ \text{ GC (4.46) } \}$$

$$\begin{array}{ll}
 & \text{length}\, z \leqslant n \wedge all\,(\equiv x)\, z \\
\equiv & \quad \{\ repeat\ \text{property (4.48)}\ \} \\
 & \text{length}\, z \leqslant n \wedge z \preccurlyeq repeat\, x \\
\equiv & \quad \{\ \text{GC (4.40)}\ \} \\
 & z \preccurlyeq take\, n\, (repeat\, x) \\
:: & \quad \{\ \text{indirect equality over list prefixing } (\preccurlyeq)\ \} \\
 & replicate\, n\, x = take\, n\, (repeat\, x)
\end{array}$$

```
-- Calculated implementation of replicate
replicate     :: a -> Int -> [a]
replicate n x = take n (repeat x)
```

4.4 Non-Galois connections

The study carried out in this dissertation of identifying functions that can be specified as Galois connections was primarily targetted at predefined functions from Haskell Standard Prelude. In some cases, in spite of exhibiting similarities with the ones addressed in the previous section, upon closer examination they could not be specified as GCs. As exmples of such counter-examples, the functions *repeat* and *words* & *unwords* are scrutinized below.

4.4.1 *repeat*

Function *repeat* of the Haskell Standard Prelude operates by generating an infinite sequence through the perpetual replication of a specified element. For instance, the outcome of *repeat 7* will be an unending sequence $[7,7,7,...]$. The idea that immediately comes to mind is to postulate the following property that relates function ‘repeat‘ with the prefix partial order:

$$all\,(\equiv x)\, z \Leftrightarrow z \preccurlyeq repeat\, x \qquad (4.48)$$

The question is: is this property a Galois connection? Clearly, this is not the case — there is no order on the lower side of the connection. Even by employing the *cast* function as studied in Section 4.2, it is not possible to express the property in the form:

$$f\, z \leqslant x \Leftrightarrow z \preccurlyeq repeat\, x$$

form some order $(\leqslant)$ and some lower adjoint f.

Nevertheless, nothing prevents one from using such a property (cf. the *replicate* calculation in the repertoire Section 4.3) and from employing indirect equality over the prefix partial order to derive the definition of *repeat*, cf.

$$
\begin{array}{ll}
 & z \preccurlyeq \mathit{repeat}\ x \\
\equiv & \quad \{\ \text{property (4.48)}\ \} \\
 & \mathit{all}\ (\equiv x)\ z \\
\equiv & \quad \{\ \text{definition of } \mathit{all}\ \} \\
 & \langle \exists\, t : z = x : t : \mathit{all}\ (\equiv x)\ t \rangle \lor z = [\,] \\
\equiv & \quad \{\ \text{property (4.48)}\ \} \\
 & \langle \exists\, t : z = x : t : t \preccurlyeq \mathit{repeat}\ x \rangle \lor z = [\,] \\
\equiv & \quad \{\ \text{prefix definition (4.18)}\ \} \\
 & z \preccurlyeq x : \mathit{repeat}\ x \\
:: & \quad \{\ \text{indirect equality over list prefixing } (\preccurlyeq)\ \} \\
 & \mathit{repeat}\ x = x : \mathit{repeat}\ x
\end{array}
$$

```
-- Calculated implementation of repeat
repeat   :: a -> [a]
repeat x = x : repeat x
```

This example demonstrates that indirect equality serves as an effective proof technique and is not limited to Galois connections exclusively.

4.4.2 *words* & *unwords*

In the Haskell Standard Prelude, some pairs of functions can be found that, *at first glance*, *appear* to constitute Galois connections. One example is the pair *words* & *unwords* that plays a fundamental role in text processing:

- $\mathit{words} : \mathit{String} \to [\mathit{String}]$ is employed to split a string by spaces or whitespace characters (spaces, tabs, and newline characters) — it essentially parses a textual input into a list of individual words.

- "conversely", $\mathit{unwords} : [\mathit{String}] \to \mathit{String}$ takes a list of words and concatenates them into a single string, using the space character as a delimiter.

Note that the expression ‘conversely’ referring to $unwords$ is enclosed in commas. In fact, the pair does not form an isomorphism, cf.

```
> words . unwords $ ["ola "]
["ola"]
```

But — is it a GC? If so, by (2.22), $unwords \cdot words \cdot unwords = unwords$ should hold. However, it clearly does not, since

```
> unwords $ ["Hello   "]
"Hello   "
> unwords . words . unwords $ ["Hello   "]
"Hello"
```

shows such a property failing at least once. This suffices to state that $words$ and $unwords$ do not form a Galois connection.

Another pair of similar functions in the Haskell Standard Prelude, is *lines* & *unlines*. The distinction lies in the fact that they operate with newline characters rather than spaces:

- $lines : String \to [String]$ divides a string into a list of substrings, each representing a line of text. The division is determined by the presence of a newline character.

- “conversely”, $unlines : [String] \to String$ accepts a list of strings (each representing a line of text) and concatenates them into a single string. Basically, it inserts newline characters between the substrings.

Again, the expression ‘conversely’ is enclosed in commas, since it does not form an isomorphism, cf.

```
> lines . unlines $ ["Hello.\n"]
["Hello.",""]
```

But, this time, no cases were found where the laws of semi-inverses (2.22) and (2.23) failed. However, this does not guarantee that it is a Galois connection, even though, through other properties, it may indeed seem to be (recall Section 2.3).

Hence, it appears quite probable to define the following Galois connection:

$$unlines\ z \leqslant l \equiv z \sqsubseteq lines\ l \tag{4.49}$$

with the orders yet to be defined.

Note, however, that both $lines$ & $unlines$ and $words$ & $unwords$ could well be a particular case of a pair of functions defined by

```haskell
sep     :: Eq a => a -> [a] -> [[a]]
sep   a = splitOn [a]

unsep   :: a -> [[a]] -> [a]
unsep a = concat . (intersperse [a])
```

It can be argued that *words* is more powerful since it separates not just by a single character, but the truth is that *unwords* uses only one such character. Nevertheless, one could define *words*, *unwords*, *lines*, and *unlines* in the following way,

```haskell
lines   = sep '\n'
unlines = unsep '\n'
words   = sep ' '
unwords = unsep ' '
```

leading to a likely Galois connection,

$$unsep\ a\ z \leqslant l \equiv z \sqsubseteq sep\ a\ l \tag{4.50}$$

the analysis of which is left for future work.

4.5 Summary

Programming from Galois connections was examined as a method for calculating programs from their specifications under Galois connections. To do so, the most common partial orders pertaining to both the Peano and the finite list algebras were initially defined. Particular attention was given to rigorously proving that they indeed constitute partial orders.

Following this, a significant result regarding predicates in Galois connections was presented. This result is subsequently demonstrated in the repertoire, where several functions from the Haskell Standard Prelude were specified under Galois connections, and their respective implementations were computed.

Finally, some attention was paid to functions that initially appeared to belong to Galois connections but were subsequently shown not to do so. Alternative approaches were outlined in such cases.

Chapter 5

Pointfree programming from Galois connections

As previously mentioned in the introduction, one of the objectives of this work is to automate the proofs using the Galculator proof assistant, a process which will be examined in the following chapter. However, it is worth noting that the Galculator operates exclusively at the pointfree level, with relational equalities. To assess such pointfree proof style, three functions were selected from the repertoire of the previous chapter for pointfree recalculation, now with their GC specifications turned into relational equalities.

Pointfree reasoning is usually advocated to ensure a fully correct and error-free alternative to pointwise proofs, which "conceal" many intermediate steps. Indeed, the rigor of pointfree proofs will become evident when comparing the respective proofs for each selected function.

The chosen functions are $take$, $takeWhile$, and zip. We shall analyze each of them in sequence. These functions were selected since, collectively, they suffice to characterize the aforementioned calculation method. With the calculation of these three functions as a foundation, the remainder are calculated in a similar manner.

Note that, since relational equalities are being employed, the uncurried versions of the $take$ and zip will be used. Towards the end, the power of adjoint recursion (3.41) will allow us to derive the original curried functions. In the case of $takeWhile$, the predicate will be fixed, i.e., we shall work with $takeWhile\ p$ for a given suitable predicate p.

5.1 Pointfree calculation of *take*

Function $\widehat{take}$ has been specified via pointwise GC (2.8) and a pointwise derivation was made. Now, rendered as a relational equality (3.93), let us work with the pointfree specification of $\widehat{take}$:

$$\langle \mathsf{length}, id\rangle^{\circ} \cdot ((\leqslant) \times (\preccurlyeq)) = (\preccurlyeq) \cdot \widehat{take} \tag{5.1}$$

First of all, since its specification recurs to the converse of length, one may calculate length° to derive some important equalities to be used in the proofs.

Therefore, and since length $= (\!| [\underline{0}, succ \cdot \pi_2] |\!)$,

$$
\begin{array}{ll}
& \mathsf{length}^\circ = (\!| [\underline{0}, succ \cdot \pi_2] |\!)^\circ \\
= & \quad \{ \text{ catamorphisms definition } \} \\
& ([\underline{0}, succ \cdot \pi_2] \cdot (id + id \times \mathsf{length}) \cdot \mathsf{out}_{\mathsf{List}})^\circ \\
= & \quad \{ \text{ converses } \} \\
& \mathsf{in}_{\mathsf{List}} \cdot (id + id \times \mathsf{length}^\circ) \cdot [\underline{0}, succ \cdot \pi_2]^\circ \\
= & \quad \{ \text{ +-definition } \} \\
& \mathsf{in}_{\mathsf{List}} \cdot [i_1, i_2 \cdot (id \times \mathsf{length}^\circ)] \cdot [\underline{0}, succ \cdot \pi_2]^\circ \\
= & \quad \{ \text{ either and converse } \} \\
& \mathsf{in}_{\mathsf{List}} \cdot (i_1 \cdot \underline{0}^\circ \ \cup \ i_2 \cdot (id \times \mathsf{length}^\circ) \cdot \pi_2^\circ \cdot succ^\circ) \\
= & \quad \{ \text{ either and converse; } \mathsf{in}_{\mathbb{N}_0} \text{ definition } \} \\
& \mathsf{in}_{\mathsf{List}} \cdot [i_1, i_2 \cdot (id \times \mathsf{length}^\circ) \cdot \pi_2^\circ] \cdot \mathsf{out}_{\mathbb{N}_0} \\
= & \quad \{ \text{ +-definition; converses } \} \\
& \mathsf{in}_{\mathsf{List}} \cdot (id + (id \times \mathsf{length})^\circ \cdot \pi_2^\circ) \cdot \mathsf{out}_{\mathbb{N}_0} \\
= & \quad \{ \text{ converses } \} \\
& \mathsf{in}_{\mathsf{List}} \cdot (id + \pi_2^\circ \cdot \mathsf{length}^\circ) \cdot \mathsf{out}_{\mathbb{N}_0} \\
= & \quad \{ \text{ +-composition } \} \\
& \mathsf{in}_{\mathsf{List}} \cdot (id + \pi_2^\circ) \cdot (id + \mathsf{length}^\circ) \cdot \mathsf{out}_{\mathbb{N}_0} \\
= & \quad \{ \text{ catamorphisms definition } \} \\
& (\!| \mathsf{in}_{\mathsf{List}} \cdot (id + \pi_2^\circ) |\!)
\end{array}
$$

which means that

$$
\begin{cases}
\mathsf{length}^\circ \cdot \underline{0} = nil \\
\mathsf{length}^\circ \cdot succ = cons \cdot (\mathsf{length} \cdot \pi_2)^\circ
\end{cases}
\tag{5.2}
$$

and from which one may derive

$$\mathsf{length}^\circ \cdot (\leqslant) \ \cap \ nil = nil \tag{5.3}$$

$$nil \ \subseteq \ \mathsf{length}^\circ \cdot (\leqslant) \tag{5.4}$$

$$(\mathsf{length} \cdot cons)^\circ \cdot (\leqslant) \cdot succ = (\mathsf{length} \cdot \pi_2)^\circ \cdot (\leqslant) \quad \text{(cf. Proof 1)} \tag{5.5}$$

Note that while length is a List-catamorphism, length$^\circ$ is an $\mathbb{N}_0$-catamorphism.

Finally, let us work with the three aforementioned cases of $\widehat{take}$:

- $z \preceq \widehat{take}\ (0,x)$ which becomes the pointfree expression $(\preceq) \cdot \widehat{take} \cdot (\underline{0} \times id)$

$$\begin{array}{ll}
 & (\preceq) \cdot \widehat{take} \cdot (\underline{0} \times id) \\
= & \qquad \{ \text{ GC (5.1) } \} \\
 & \langle \mathsf{length}, id \rangle^\circ \cdot ((\leqslant) \times (\preceq)) \cdot (\underline{0} \times id) \\
= & \qquad \{ \ \times\text{-definition and composition; constants natural property; pairing and converse } \} \\
 & \mathsf{length}^\circ \cdot (\leqslant) \cdot \underline{0} \ \cap \ (\preceq) \cdot \pi_2 \\
= & \qquad \{ \ (\leqslant) \text{ definition (4.9); } \mathsf{length}^\circ \cdot \underline{0} = nil \text{ (5.2) } \} \\
 & nil \ \cap \ (\preceq) \cdot \pi_2 \\
= & \qquad \{ \ nil \subseteq (\preceq) \cdot \pi_2 \text{ since } \mathrm{img}\ \pi_2 = id \text{ and } nil = \llparenthesis [nil, nil] \rrparenthesis \subseteq (\preceq) \text{ by monotonicity } \} \\
 & (\preceq) \cdot nil \\
:: & \qquad \{ \text{ indirect equality over prefix partial order } (\preceq) \ \} \\
 & \widehat{take} \cdot (\underline{0} \times id) = nil
\end{array} \tag{5.6}$$

- $z \preceq \widehat{take}\ (n, [\,])$ which becomes $(\preceq) \cdot \widehat{take} \cdot (id \times nil)$

$$\begin{array}{ll}
 & (\preceq) \cdot \widehat{take} \cdot (id \times nil) \\
= & \qquad \{ \text{ GC (5.1) } \} \\
 & \langle \mathsf{length}, id \rangle^\circ \cdot ((\leqslant) \times (\preceq)) \cdot (id \times nil) \\
= & \qquad \{ \ \times\text{-definition and composition; constants natural property; pairing and converse } \} \\
 & \mathsf{length}^\circ \cdot (\leqslant) \cdot \pi_1 \ \cap \ (\preceq) \cdot nil \\
= & \qquad \{ \ (\preceq) \cdot nil = nil \subseteq \mathsf{length}^\circ \cdot (\leqslant) \cdot \pi_1 \quad \text{(cf. Proof 3) } \} \\
 & (\preceq) \cdot nil \\
:: & \qquad \{ \text{ indirect equality over prefix partial order } (\preceq) \ \} \\
 & \widehat{take} \cdot (id \times nil) = nil
\end{array} \tag{5.7}$$

- $z \preceq \widehat{take}\ (n+1, h:t)$ which becomes $(\preceq) \cdot \widehat{take} \cdot (succ \times cons)$

$$\begin{array}{ll}
 & (\preceq) \cdot \widehat{take} \cdot (succ \times cons) \\
= & \qquad \{ \text{ GC (5.1) } \}
\end{array}$$

$$\langle \mathsf{length}, id\rangle^\circ \cdot ((\leqslant)\times(\preccurlyeq))\cdot(succ\times cons)$$

$$= \quad \{ \ \times\text{-composition and definition} \ \}$$

$$\langle \mathsf{length}, id\rangle^\circ \cdot \langle(\leqslant)\cdot succ\cdot\pi_1,(\preccurlyeq)\cdot cons\cdot\pi_2\rangle$$

$$= \quad \{ \ \text{pairing and converse} \ \}$$

$$\mathsf{length}^\circ\cdot(\leqslant)\cdot succ\cdot\pi_1 \ \cap \ (\preccurlyeq)\cdot cons\cdot\pi_2$$

$$= \quad \{ \ (\preccurlyeq)\text{ definition (4.17)} \ \}$$

$$\mathsf{length}^\circ\cdot(\leqslant)\cdot succ\cdot\pi_1 \ \cap \ (nil \ \cup \ cons\cdot(id\times(\preccurlyeq))\cdot\pi_2)$$

$$= \quad \{ \ \text{distributivity; } \underline{k}\cdot R=\underline{k} \text{ when } R \text{ is entire; } id\times(\preccurlyeq) \text{ is entire since } (\preccurlyeq) \text{ is reflexive} \ \}$$

$$(\mathsf{length}^\circ\cdot(\leqslant)\cdot succ\cdot\pi_1 \ \cap \ nil) \ \cup \ (\mathsf{length}^\circ\cdot(\leqslant)\cdot succ\cdot\pi_1 \ \cap \ cons\cdot(id\times(\preccurlyeq))\cdot\pi_2)$$

$$= \quad \{ \ nil \subseteq \mathsf{length}^\circ\cdot(\leqslant)\cdot succ\cdot\pi_1 \text{ (check (5.4))} \ \}$$

$$nil \ \cup \ (\mathsf{length}^\circ\cdot(\leqslant)\cdot succ\cdot\pi_1 \ \cap \ cons\cdot(id\times(\preccurlyeq))\cdot\pi_2)$$

$$= \quad \{ \ \text{distributivity, since } cons^\circ\cdot cons \text{ is injective; contravariance} \ \}$$

$$nil \ \cup \ cons\cdot((\mathsf{length}\cdot cons)^\circ\cdot(\leqslant)\cdot succ\cdot\pi_1 \ \cap \ (id\times(\preccurlyeq))\cdot\pi_2)$$

$$= \quad \{ \ \text{(5.5)} \ \}$$

$$nil \ \cup \ cons\cdot((\mathsf{length}\cdot\pi_2)^\circ\cdot(\leqslant)\cdot\pi_1 \ \cap \ (id\times(\preccurlyeq))\cdot\pi_2)$$

$$= \quad \{ \ \text{contravariance; } \times\text{-definition and fusion; pairing definition} \ \}$$

$$nil \ \cup \ cons\cdot(\mathsf{length}^\circ\cdot\pi_2^\circ\cdot(\leqslant)\cdot\pi_1 \ \cap \ (\pi_1^\circ\cdot\pi_1\cdot\pi_2 \ \cap \ \pi_2^\circ\cdot(\preccurlyeq)\cdot\pi_2\cdot\pi_2))$$

$$= \quad \{ \ \text{associativity} \ \}$$

$$nil \ \cup \ cons\cdot(\pi_1^\circ\cdot\pi_1\cdot\pi_2 \ \cap \ (\pi_2^\circ\cdot\mathsf{length}^\circ\cdot(\leqslant)\cdot\pi_1 \ \cap \ \pi_2^\circ\cdot(\preccurlyeq)\cdot\pi_2\cdot\pi_2))$$

$$= \quad \{ \ \text{distributivity since } \pi_2^\circ \text{ is injective; pairing definition} \ \}$$

$$nil \ \cup \ cons\cdot\langle\pi_1\cdot\pi_2,\mathsf{length}^\circ\cdot(\leqslant)\cdot\pi_1 \ \cap \ (\preccurlyeq)\cdot\pi_2\cdot\pi_2\rangle$$

$$= \quad \{ \ \text{pairing and converse; identity} \ \}$$

$$nil \ \cup \ cons\cdot\langle\pi_1\cdot\pi_2,\langle\mathsf{length},id\rangle^\circ\cdot\langle(\leqslant)\cdot\pi_1,(\preccurlyeq)\cdot\pi_2\cdot\pi_2\rangle\rangle$$

$$= \quad \{ \ \times\text{-definition} \ \}$$

$$nil \ \cup \ cons\cdot\langle\pi_1\cdot\pi_2,\langle\mathsf{length},id\rangle^\circ\cdot((\leqslant)\times(\preccurlyeq)\cdot\pi_2)\rangle$$

$$= \quad \{ \ \times\text{-absorption and composition} \ \}$$

$$nil \ \cup \ cons\cdot(id\times\langle\mathsf{length},id\rangle^\circ\cdot((\leqslant)\times(\preccurlyeq))\cdot\langle\pi_1\cdot\pi_2,id\times\pi_2\rangle)$$

$$= \quad \{ \ \text{GC (5.1)} \ \}$$

$$
\begin{array}{cl}
& nil \cup cons \cdot ((id \times (\preccurlyeq) \cdot \widehat{take}) \cdot \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle) \\
= & \qquad \{ \ \times\text{-composition} \ \} \\
& nil \cup cons \cdot (id \times (\preccurlyeq)) \cdot (id \times \widehat{take}) \cdot \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle) \\
= & \qquad \{ \ \text{constants natural property} \ \} \\
& nil \cdot (id \times \widehat{take}) \cdot \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle \cup cons \cdot (id \times (\preccurlyeq)) \cdot (id \times \widehat{take}) \cdot \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle) \\
= & \qquad \{ \ \text{left linearity} \ \} \\
& (nil \cup cons \cdot (id \times (\preccurlyeq))) \cdot (id \times \widehat{take}) \cdot \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle \\
= & \qquad \{ \ (\preccurlyeq) \text{ definition } (4.17) \ \} \\
& (\preccurlyeq) \cdot cons \cdot (id \times \widehat{take}) \cdot \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle \\
:: & \qquad \{ \ \text{indirect equality over prefix partial order } (\preccurlyeq) \ \} \\
& \widehat{take} \cdot (succ \times cons) = cons \cdot (id \times \widehat{take}) \cdot \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle
\end{array}
\tag{5.8}
$$

This leads to the already presented definition of $\widehat{take}$:

$$
\begin{array}{l}
\widehat{take}\ (0, \_) = [\,] \\
\widehat{take}\ (\_, [\,]) = [\,] \\
\widehat{take}\ (n+1, h:t) = h : \widehat{take}\ (n, t)
\end{array}
$$

which is clearly the following G-hylomorphism on lists:

$$\begin{tikzcd}
\mathbb{N}_0 \times A^* \arrow[d, "\widehat{take}"'] & & 1 + A \times (\mathbb{N}_0 \times A^*) \arrow[d, "\mathsf{G}\,\widehat{take}"'] & (1+\mathbb{N}_0) \times A^* \arrow[l, "\psi"'] \arrow[lll, bend right, "\mathsf{L}\ \mathsf{in}_{\mathbb{N}_0}"'] \\
A^* & & 1 + A \times A^* \arrow[ll, "\mathsf{in}_{\mathsf{List}}"] &
\end{tikzcd}$$

where

$$
\begin{array}{l}
\psi = [i_1 \cdot \pi_1 \,, (\pi_2 + xr) \cdot distr \cdot (id \times \mathsf{out}_{\mathsf{List}})] \cdot distl \\
xr = \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle \\
\mathsf{L}\, f = f \times id \\
\mathsf{G}\, f = id + id \times f
\end{array}
$$

That is,

$$\widehat{take} = \mathsf{in}_{\mathsf{List}} \cdot \mathsf{G}\ \widehat{take} \cdot \psi \cdot \mathsf{L}\ \mathsf{out}_{\mathbb{N}_0}$$

Now, let us recur to the power of adjoint recursion in order to define the original (curried) version of *take* as an adjoint catamorphism:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "take"'] & & & 1+\mathbb{N}_0 \arrow[lll, "\mathsf{in}_{\mathbb{N}_0}"'] \arrow[d, "id+take"] \\
(A^*)^{A^*} & & & 1+(A^*)^{A^*} \arrow[lll, "\overline{\mathsf{in}_{\mathsf{List}}\cdot \mathsf{G}\,\varepsilon\cdot\psi}"]
\end{tikzcd}$$

$$\begin{aligned}
&take = (\!|\overline{\mathsf{in}_{\mathsf{List}}\cdot \mathsf{G}\,\varepsilon\cdot\psi}|\!)\ \textbf{where}\\
&\quad \varepsilon = ap
\end{aligned}$$

Simplifying the gene of catamorphism *take*:

$$\begin{aligned}
& [f\,,g] = \overline{\mathsf{in}_{\mathsf{List}}\cdot \mathsf{G}\,\varepsilon\cdot\psi}\\
\equiv & \qquad \{\ \text{definitions of } \mathsf{in}_{\mathsf{List}},\ \mathsf{G}\,\varepsilon \text{ and } \psi\ \}\\
& [f\,,g] = \overline{[nil\,,cons]\cdot(id+id\times\varepsilon)\cdot[i_1\cdot\pi_1\,,(\pi_2+xr)\cdot distr\cdot(id\times\mathsf{out}_{\mathsf{List}})]\cdot distl}\\
\equiv & \qquad \{\ \text{coproducts}\ \}\\
& [f\,,g] = \overline{[nil\,,[nil\,,cons\cdot(id\times\varepsilon)\cdot xr]\cdot distr\cdot(id\times\mathsf{out}_{\mathsf{List}})]\cdot distl}\\
\equiv & \qquad \{\ [\bar{f}\,,\bar{g}] = \overline{[f\,,g]\cdot distl}\ \}\\
& \begin{cases} f = \overline{nil}\\ g = \overline{[nil\,,cons\cdot(id\times\varepsilon)\cdot xr]\cdot distr\cdot(id\times\mathsf{out}_{\mathsf{List}})}\end{cases}\\
\equiv & \qquad \{\ \overline{f\cdot(id\times g)} = (\cdot g)\cdot\bar{f}\ \}\\
& \begin{cases} f = \overline{nil}\\ g = (\cdot\mathsf{out}_{\mathsf{List}})\cdot\overline{[nil\,,cons\cdot(id\times\varepsilon)\cdot xr]\cdot distr}\end{cases}
\end{aligned}$$

In order to continue simplifying the gene, recall that $A\times B\ \cong\ B\times A$. Thus, recurring to the function $flip :: (a\to b\to c)\to b\to a\to c$, one may encounter the adjunction $(K\times \_)\dashv(\_^K)$, i.e.,

$$\begin{tikzcd}
K\times A\to B \arrow[rr, bend left, <-, "uncurry\cdot flip"] \arrow[rr, bend right, "flip\cdot curry"'] & \cong & A\to B^K
\end{tikzcd} \tag{5.9}$$

leading to the following property:

$$[flip\,\bar{f}\,,flip\,\bar{g}] = flip\,\overline{[f\,,g]\cdot distr} \tag{5.10}$$

Now, let us proceed with

$$\begin{cases} f = \overline{nil} \\ g = (\cdot \mathsf{out}_{\mathsf{List}}) \cdot (\mathit{flip}\ (\mathit{flip}\ \overline{[nil\,, cons \cdot (id \times \varepsilon) \cdot xr] \cdot distr})) \end{cases}$$

$$\equiv \qquad \{\ (5.10)\ \}$$

$$\begin{cases} f = \overline{nil} \\ g = (\cdot \mathsf{out}_{\mathsf{List}}) \cdot (\mathit{flip}\ [\mathit{flip}\ \overline{nil}\,, \mathit{flip}\ \overline{cons \cdot (id \times \varepsilon) \cdot xr}]) \end{cases}$$

$$\equiv \qquad \{\ \text{isomorphism } \mathit{flip} = \mathit{flip}^{\circ}\ \}$$

$$\begin{cases} f = \overline{nil} \\ \mathit{flip}\ ((\cdot \mathsf{in}_{\mathsf{List}}) \cdot g) = [\mathit{flip}\ \overline{nil}\,, \mathit{flip}\ \overline{cons \cdot (id \times \varepsilon) \cdot xr}] \end{cases}$$

$$\equiv \qquad \{\ +\text{-universal property}\ \}$$

$$\begin{cases} f = \overline{nil} \\ \begin{cases} (\mathit{flip}\ ((\cdot \mathsf{in}_{\mathsf{List}}) \cdot g)) \cdot i_1 = \mathit{flip}\ \overline{nil} \\ (\mathit{flip}\ ((\cdot \mathsf{in}_{\mathsf{List}}) \cdot g)) \cdot i_2 = \mathit{flip}\ \overline{cons \cdot (id \times \varepsilon) \cdot xr} \end{cases} \end{cases}$$

$$\equiv \qquad \{\ \mathit{flip}\text{-fusion (Oliveira, 2020a); remove } \mathit{flip} \text{ both sides}\ \}$$

$$\begin{cases} f = \overline{nil} \\ \begin{cases} (\cdot i_1) \cdot (\cdot \mathsf{in}_{\mathsf{List}}) \cdot g = \overline{nil} \\ (\cdot i_2) \cdot (\cdot \mathsf{in}_{\mathsf{List}}) \cdot g = \overline{cons \cdot (id \times \varepsilon) \cdot xr} \end{cases} \end{cases}$$

$$\equiv \qquad \{\ \text{composition of pre-compositions is contravariant}\ \}$$

$$\begin{cases} f = \overline{nil} \\ \begin{cases} (\cdot (\mathsf{in}_{\mathsf{List}} \cdot i_1)) \cdot g = \overline{nil} \\ (\cdot (\mathsf{in}_{\mathsf{List}} \cdot i_2)) \cdot g = \overline{cons \cdot (id \times \varepsilon) \cdot xr} \end{cases} \end{cases}$$

$$\equiv \qquad \{\ \mathsf{in}_{\mathsf{List}} \cdot i_1 = nil \text{ and } \mathsf{in}_{\mathsf{List}} \cdot i_2 = cons\text{; going pointwise on the pre-compositions}\ \}$$

$$\begin{cases} f = \overline{nil} \\ \begin{cases} (g\ k) \cdot nil = nil \\ (g\ k) \cdot cons = \overline{cons \cdot (id \times \varepsilon) \cdot xr}\ k \end{cases} \end{cases}$$

$$\equiv \qquad \{\ \text{going pointwise}\ \}$$

$$\begin{cases} f\ \_\ \_ = [\,] \\ \begin{cases} g\ k\ [\,] = [\,] \\ g\ k\ (h : t) = h : k\ t \end{cases} \end{cases}$$

Therefore, function *take* is given by the following (higher-order) catamorphism:

$take :: Integral\ c \Rightarrow c \rightarrow [a] \rightarrow [a]$

$take = (\![[f\,, g]]\!)$ **where**

$\quad f\ \_\ \_ = [\,]$

$\quad g\ k\ [\,] = [\,]$

$\quad g\ k\ (h : t) = h : k\ t$

5.2 Pointfree calculation of *takeWhile*

Recall from Section 4.2 that *takeWhile* can be specified via relational equality recurring to the use of coreflexives, cf. (4.34). That is,

$$\phi_{all\ p} \cdot (\preccurlyeq) = (\preccurlyeq) \cdot takeWhile\ p \qquad (5.11)$$

Besides, function *takeWhile* has already been calculated using a pointwise approach. Now, similar to what occurred with the preceding function, *takeWhile* will be calculated based on the aforementioned relational equality through a pointfree methodology. In order to do so, predicate p will be held constant, and thus we shall be working with $takeWhile\ p$ instead, leading to the following calculations:

- $z \preccurlyeq takeWhile\ p\ [\,]$ which becomes the pointfree expression $(\preccurlyeq) \cdot takeWhile\ p \cdot nil$

$$\begin{array}{cl}
 & (\preccurlyeq) \cdot takeWhile\ p \cdot nil \\
= & \quad \{\ \text{GC (5.11)}\ \} \\
 & \phi_{all\ p} \cdot (\preccurlyeq) \cdot nil \\
= & \quad \{\ (\preccurlyeq)\ \text{definition (4.17)}\ \} \\
 & \phi_{all\ p} \cdot nil \\
= & \quad \{\ all\ q\ [\,]\ \text{holds for every predicate}\ q\ \} \\
 & nil \\
= & \quad \{\ (\preccurlyeq) \cdot nil = nil\ \text{(4.17)}\ \} \\
 & (\preccurlyeq) \cdot nil \\
:: & \quad \{\ \text{indirect equality over prefix partial order}\ (\preccurlyeq)\ \} \\
 & takeWhile\ p \cdot nil = nil
\end{array}$$

- $z \preccurlyeq takeWhile\ p\ (h : t)$ which becomes the pointfree expression $(\preccurlyeq) \cdot takeWhile\ p \cdot cons$

$$(\preccurlyeq) \cdot takeWhile\ p \cdot cons$$

$=$ { GC (5.11) }

$$\phi_{all\ p} \cdot (\preccurlyeq) \cdot cons$$

$=$ { $(\preccurlyeq)$ definition (4.17) }

$$\phi_{all\ p} \cdot (cons \cdot (id \times (\preccurlyeq)) \ \cup \ nil)$$

$=$ { right linearity; $\phi_{all\ p} \cdot nil = nil$ since $all\ q\ [\,]$ holds for every suitable predicate q }

$$\phi_{all\ p} \cdot cons \cdot (id \times (\preccurlyeq)) \ \cup \ nil$$

$=$ { $\phi_{all\ p} = \mathsf{map}\ \phi_p$ }

$$cons \cdot (\phi_p \times \phi_{all\ p}) \cdot (id \times (\preccurlyeq)) \ \cup \ nil$$

$=$ { $\times$-functor composition }

$$cons \cdot (\phi_p \times id) \cdot (id \times \phi_{all\ p} \cdot (\preccurlyeq)) \ \cup \ nil$$

$=$ { GC (5.11) }

$$cons \cdot (\phi_p \times id) \cdot (id \times (\preccurlyeq) \cdot takeWhile\ p) \ \cup \ nil$$

$=$ { $\times$-functor composition twice }

$$cons \cdot (id \times (\preccurlyeq)) \cdot (id \times takeWhile\ p) \cdot (\phi_p \times id) \ \cup \ nil$$

$=$ { $\phi_p \ \cup \ \phi_{\neg\, p} = id$ }

$$cons \cdot (id \times (\preccurlyeq)) \cdot (id \times takeWhile\ p) \cdot (\phi_p \times id) \ \cup \ nil \cdot (\phi_p \times id \ \cup \ \phi_{\neg\, p} \times id)$$

$=$ { right linearity and associativity }

$$cons \cdot (id \times (\preccurlyeq)) \cdot (id \times takeWhile\ p) \cdot (\phi_p \times id) \ \cup \ nil \cdot (\phi_p \times id) \ \cup \ nil \cdot (\phi_{\neg\, p} \times id)$$

$=$ { left linearity }

$$(cons \cdot (id \times (\preccurlyeq)) \cdot (id \times takeWhile\ p) \ \cup \ nil) \cdot (\phi_p \times id) \ \cup \ nil \cdot (\phi_{\neg\, p} \times id)$$

$=$ { left linearity again, since $nil = nil \cdot (id \times takeWhile\ p)$ }

$$(cons \cdot (id \times (\preccurlyeq)) \ \cup \ nil) \cdot (id \times takeWhile\ p) \cdot (\phi_p \times id) \ \cup \ nil \cdot (\phi_{\neg\, p} \times id)$$

$=$ { $(\preccurlyeq)$ definition (4.17) }

$$(\preccurlyeq) \cdot cons \cdot (id \times takeWhile\ p) \cdot (\phi_p \times id) \ \cup \ nil \cdot (\phi_{\neg\, p} \times id)$$

$=$ { $\phi_{p \cdot \pi_1} = \phi_p \times id$ }

$$(\preccurlyeq) \cdot cons \cdot (id \times takeWhile\ p) \cdot \phi_{p \cdot \pi_1} \ \cup \ nil \cdot \phi_{\neg\, (p \cdot \pi_1)}$$

$=$ { either and converse since R being coreflexive means $R^\circ = R$ }

$$[(\preccurlyeq) \cdot cons \cdot (id \times takeWhile\ p)\, , nil] \cdot [\phi_{p \cdot \pi_1}\, , \phi_{\neg\, (p \cdot \pi_1)}]^\circ$$

$$
\begin{array}{ll}
= & \quad \{\ nil = (\preccurlyeq) \cdot nil \ \text{(4.17); +-fusion; } p? = [\phi_p \, , \phi_{\neg\, p}]^\circ \ \} \\
& (\preccurlyeq) \cdot [cons \cdot (id \times takeWhile\ p) \, , nil] \cdot (p \cdot \pi_1)? \\
:: & \quad \{\ \text{McCarthy's conditional and indirect equality over prefix partial order } (\preccurlyeq) \ \} \\
& takeWhile\ p \cdot cons = p \cdot \pi_1 \rightarrow cons \cdot (id \times takeWhile\ p), \ nil
\end{array}
$$

Finally, since

$$
\begin{array}{ll}
& p \cdot \pi_1 \rightarrow cons \cdot (id \times takeWhile\ p), \ nil \\
= & \quad \{\ \text{McCarthy's conditional and coproducts} \ \} \\
& (p \cdot \pi_1 \rightarrow cons, \ nil) \cdot (id \times takeWhile\ p)
\end{array}
$$

one gets the definition of *takeWhile* as the following List-catamorphism:

$$
\begin{array}{l}
takeWhile :: (a \rightarrow \mathbb{B}) \rightarrow [a] \rightarrow [a] \\
takeWhile\ p = (\!|\, [nil \, , cond\ (p \cdot \pi_1)\ cons\ nil] \,|\!)
\end{array}
$$

5.3 Pointfree calculation of *uncurry zip*

As a final example, we shall calculate function $\widehat{zip}$ based on a relational equality. As in previous functions, we shall commence by converting its pointwise specification (4.45) to a pointfree representation, cf.

$$
\langle \mathsf{map}\ \pi_1 , \mathsf{map}\ \pi_2 \rangle^\circ \cdot ((\preccurlyeq) \times (\preccurlyeq)) = (\preccurlyeq) \cdot \widehat{zip} \tag{5.12}
$$

Thus, one deals with the respective cases:

- $z \preccurlyeq \widehat{zip}\ (l_1, [\,])$ which becomes $(\preccurlyeq) \cdot \widehat{zip} \cdot (id \times nil)$

$$
\begin{array}{ll}
& (\preccurlyeq) \cdot \widehat{zip} \cdot (id \times nil) \\
= & \quad \{\ \text{GC (5.12)} \ \} \\
& \langle \mathsf{map}\ \pi_1 , \mathsf{map}\ \pi_2 \rangle^\circ \cdot ((\preccurlyeq) \times (\preccurlyeq)) \cdot (id \times nil) \\
= & \quad \{\ \times\text{-functor composition} \ \} \\
& \langle \mathsf{map}\ \pi_1 , \mathsf{map}\ \pi_2 \rangle^\circ \cdot \langle (\preccurlyeq) \cdot \pi_1 , nil \rangle \\
= & \quad \{\ \text{pairing and converse} \ \} \\
& (\mathsf{map}\ \pi_1)^\circ \cdot (\preccurlyeq) \cdot \pi_1 \ \cap \ (\mathsf{map}\ \pi_2)^\circ \cdot nil
\end{array}
$$

$$= \qquad \{ \text{ map converse and definition } \}$$

$$(\mathsf{map}\ \pi_1)^\circ \cdot (\preccurlyeq) \cdot \pi_1 \ \cap \ nil$$

$$= \qquad \{ \ nil \subseteq (\mathsf{map}\ \pi_1)^\circ \cdot (\preccurlyeq) \cdot \pi_1 ; \ (\preccurlyeq) \cdot nil = nil \ (4.17) \ \}$$

$$(\preccurlyeq) \cdot nil$$

$$:: \qquad \{ \text{ indirect equality over prefix partial order } (\preccurlyeq) \ \}$$

$$\widehat{zip} \cdot (id \times nil) = nil$$

- $z \preccurlyeq \widehat{zip}\ ([\,], l_2)$ which becomes $(\preccurlyeq) \cdot \widehat{zip} \cdot (nil \times id)$

  Analogous to the previous calculation, leading to $\widehat{zip} \cdot (nil \times id) = nil$

- $z \preccurlyeq \widehat{zip}\ (h_1 : t_1, h_2 : t_2)$ which is equivalent to $(\preccurlyeq) \cdot \widehat{zip} \cdot (cons \times cons)$

  For this, we shall denominate function $\langle \pi_1 \times \pi_1, \pi_2 \times \pi_2 \rangle$ as *transpose*. This function is an isomorphism:

$$\begin{tikzcd}(A \times B) \times (C \times D) \arrow[r, bend left, "transpose"] \arrow[r, phantom, "\cong"] & (A \times C) \times (B \times D) \arrow[l, bend left, "transpose^\circ"]\end{tikzcd}$$

It is trivial to check that $transpose^\circ = transpose$. Besides, its natural property,

$$((f \times g) \times (h \times i)) \cdot transpose = transpose \cdot ((f \times h) \times (g \times i)) \tag{5.13}$$

will prove very useful in the following proof.

Therefore, one has

$$(\preccurlyeq) \cdot \widehat{zip} \cdot (cons \times cons)$$

$$= \qquad \{ \text{ GC } (5.12) \ \}$$

$$\langle \mathsf{map}\ \pi_1, \mathsf{map}\ \pi_2 \rangle^\circ \cdot ((\preccurlyeq) \times (\preccurlyeq)) \cdot (cons \times cons)$$

$$= \qquad \{ \ \times\text{-functor-composition } \}$$

$$\langle \mathsf{map}\ \pi_1, \mathsf{map}\ \pi_2 \rangle^\circ \cdot ((\preccurlyeq) \cdot cons \times (\preccurlyeq) \cdot cons)$$

$$= \qquad \{ \text{ pairing definition and converse; functor map } \}$$

$$\mathsf{map}\ \pi_1^\circ \cdot (\preccurlyeq) \cdot cons \cdot \pi_1 \ \cap \ \mathsf{map}\ \pi_2^\circ \cdot (\preccurlyeq) \cdot cons \cdot \pi_2$$

$$= \qquad \{ \ (\preccurlyeq) \text{ definition; left and right linearity twice; constants' natural property; map } \_ \cdot nil = nil \ \}$$

$$(\mathsf{map}\ \pi_1^\circ \cdot cons \cdot (id \times (\preccurlyeq)) \cdot \pi_1 \ \cup \ nil) \ \cap \ (\mathsf{map}\ \pi_2^\circ \cdot cons \cdot (id \times (\preccurlyeq)) \cdot \pi_2 \ \cup \ nil)$$

$$= \qquad \{\ \text{union distributes through intersection}\ \}$$

$$(\text{map}\ \pi_1^\circ \cdot cons \cdot (id \times (\preccurlyeq)) \cdot \pi_1 \ \cap\ \text{map}\ \pi_2^\circ \cdot cons \cdot (id \times (\preccurlyeq)) \cdot \pi_2) \ \cup\ nil$$

$$= \qquad \{\ \text{map}\ R \cdot cons = cons \cdot (R \times \text{map}\ R)\text{; } cons \text{ is injective}\ \}$$

$$cons \cdot ((\pi_1^\circ \times \text{map}\ \pi_1^\circ \cdot (\preccurlyeq)) \cdot \pi_1 \ \cap\ (\pi_2^\circ \times \text{map}\ \pi_2^\circ \cdot (\preccurlyeq)) \cdot \pi_2) \ \cup\ nil$$

$$= \qquad \{\ \times\text{-functor composition}\ \}$$

$$cons \cdot ((\pi_1^\circ \times \text{map}\ \pi_1^\circ) \cdot (id \times (\preccurlyeq)) \cdot \pi_1 \ \cap\ (\pi_2^\circ \times \text{map}\ \pi_2^\circ) \cdot (id \times (\preccurlyeq)) \cdot \pi_2) \ \cup\ nil$$

$$= \qquad \{\ \text{pairing and converse; } \times\text{-functor definition}\ \}$$

$$cons \cdot \langle \pi_1 \times \text{map}\ \pi_1, \pi_2 \times \text{map}\ \pi_2 \rangle^\circ \cdot ((id \times (\preccurlyeq)) \times (id \times (\preccurlyeq))) \ \cup\ nil$$

$$= \qquad \{\ \langle \pi_1 \times \text{map}\ \pi_1, \pi_2 \times \text{map}\ \pi_2 \rangle^\circ = (id \times \langle \text{map}\ \pi_1, \text{map}\ \pi_2 \rangle)^\circ \cdot transpose \ \text{(Proof 2)}\ \}$$

$$cons \cdot (id \times \langle \text{map}\ \pi_1, \text{map}\ \pi_2 \rangle)^\circ \cdot transpose \cdot ((id \times (\preccurlyeq)) \times (id \times (\preccurlyeq))) \ \cup\ nil$$

$$= \qquad \{\ \text{converses; } transpose \text{ natural property}\ \}$$

$$cons \cdot (id \times \langle \text{map}\ \pi_1, \text{map}\ \pi_2 \rangle^\circ) \cdot (id \times ((\preccurlyeq) \times (\preccurlyeq))) \cdot transpose \ \cup\ nil$$

$$= \qquad \{\ \times\text{-functor composition}\ \}$$

$$cons \cdot (id \times \langle \text{map}\ \pi_1, \text{map}\ \pi_2 \rangle^\circ \cdot ((\preccurlyeq) \times (\preccurlyeq))) \cdot transpose \ \cup\ nil$$

$$= \qquad \{\ \text{GC (5.12)}\ \}$$

$$cons \cdot (id \times (\preccurlyeq) \cdot \widehat{zip}) \cdot transpose \ \cup\ nil$$

$$= \qquad \{\ \times\text{-functor-composition}\ \}$$

$$cons \cdot (id \times (\preccurlyeq)) \cdot (id \times \widehat{zip}) \cdot transpose \ \cup\ nil$$

$$= \qquad \{\ \text{left linearity; constants' natural property}\ \}$$

$$(cons \cdot (id \times (\preccurlyeq)) \ \cup\ nil) \cdot (id \times \widehat{zip}) \cdot transpose$$

$$= \qquad \{\ (\preccurlyeq) \text{ definition}\ \}$$

$$(\preccurlyeq) \cdot cons \cdot (id \times \widehat{zip}) \cdot transpose$$

$$:: \qquad \{\ \text{indirect equality over prefix partial order } (\preccurlyeq)\ \}$$

$$\widehat{zip} \cdot (cons \times cons) = cons \cdot (id \times \widehat{zip}) \cdot transpose$$

This leads to the already presented definition of $\widehat{zip}$:

$$\begin{aligned}
&\widehat{zip}\ ([\,], \_) = [\,] \\
&\widehat{zip}\ (\_, [\,]) = [\,] \\
&\widehat{zip}\ (h_1 : t_1, h_2 : t_2) = (h_1, h_2) : \widehat{zip}\ (n, t)
\end{aligned}$$

which is clearly the following G-hylomorphism on lists:

$$\begin{tikzcd}
A^* \times A^* \arrow[d, "\widehat{zip}"'] & 1+(A\times A)\times(A^*\times A^*) \arrow[d, "\mathsf{G}\,\widehat{zip}"'] & (1+A\times A^*)\times A^* \arrow[l, "\psi"] \arrow[ll, bend right, "\mathsf{L}\ \mathsf{in}_{\mathsf{List}}"'] \\
(A\times A)^* & 1+(A\times A)\times(A\times A)^* \arrow[l, "\mathsf{in}_{\mathsf{List}}"] &
\end{tikzcd}$$

where

$$\psi\ (i_1\ (), \_) = i_1\ ()$$
$$\psi\ (\_, [\,]) = i_1\ ()$$
$$\psi\ (i_2\ (h_1,t_1), h_2 : t_2) = i_2\ ((h_1,h_2),(t_1,t_2))$$
$$\mathsf{L} f = f \times id$$
$$\mathsf{G} f = id + id \times f$$

That is,

$$\widehat{zip} = \mathsf{in}_{\mathsf{List}} \cdot \mathsf{G}\ \widehat{zip} \cdot \psi \cdot \mathsf{L}\ \mathsf{out}_{\mathsf{List}}$$

Now, let us again recur to the power of adjoint recursion in order to define the original (curried) version of zip as an adjoint catamorphism:

$$\begin{tikzcd}
A^* \arrow[d, "take"'] & & 1+A\times A^* \arrow[ll, "\mathsf{in}_{\mathsf{List}}"'] \arrow[d, "id+id\times take"] \\
((A\times A)^*)^{A^*} & & 1+A\times((A\times A)^*)^{A^*} \arrow[ll, "\overline{\mathsf{in}_{\mathsf{List}}\cdot\mathsf{G}\,\varepsilon\cdot\psi}"]
\end{tikzcd}$$

$$zip = \llparenthesis \overline{\mathsf{in}_{\mathsf{List}} \cdot \mathsf{G}\ \varepsilon \cdot \psi} \rrparenthesis\ \textbf{where}$$
$$\quad \varepsilon = ap$$

The process to simplify the gene of catamorphism zip is similar to the catamorphism $take$. Thus, zip is defined by the following pointwise definition:

$$zip :: [a] \rightarrow [b] \rightarrow [(a,b)]$$
$$zip = \llparenthesis [f,g] \rrparenthesis\ \textbf{where}$$
$$\quad f\ \_\ \_ = [\,]$$
$$\quad g\ \_\ [\,] = [\,]$$
$$\quad g\ (h_1,k)\ (h_2 : t_2) = (h_1,h_2) : k\ t_2$$

5.4 Summary

Pointfree level reasoning ensures complete correctness in program calculation, in contrast with pointwise calculations that often “conceal” many intermediate steps. Moreover, automation is enabled by tools that, such as the Galculator, adopt such a reasoning paradigm.

In this chapter three functions — namely, $take$, $takeWhile$ and zip — were chosen to be recalculated, this time at the pointfree level. In the case of the $take$ and zip, whose uncurried versions had been calculated, the study of adjunctions and catamorphisms ultimately facilitated the calculation of the original curried version.

Galois-connection pointfree proofs rely, as previously mentioned, on relational equalities, which is the framework employed by the aforementioned Galculator proof assistant. This is the subject of the chapter that follows.

Chapter 6

Galculator

The proposal of a design strategy for software development starting from specifications cast in the form of Galois connections, leading to correct-by-construction (CbC) artifacts, has taken most of the current dissertation. Looking at some of the calculations that have been performed by hand, it becomes clear that CbC comes with a cost: it may take a significant number of steps to reach the final implementation. This calls for some kind of proof-assistance.

The prototype proof assistant known as Galculator (Silva and Oliveira, 2008) is distinctively centered on the algebra of Galois connections. Its effectiveness is heightened when integrated with techniques like the pointfree transform (recall Section 3.3) and proof methods such as the indirect equality principle.

Clearly, the Galculator is a first choice for assisting in calculating functions specified by Galois connections, provided these are appropriately expressed at the pointfree level, as shown in the previous chapter. This chapter will demonstrate how the tool can be used as proof assistant in such proofs.

The potential for proof automation supported by the Galculator tool will be showcased taking the calculation of the function $take$ as demonstration example.

6.1 Launching Galculator

After refactoring part of the code of the Galculator due to updates in the Haskell programming language and its libraries since the tool was developed years ago, all modules run properly. The Galculator was containerized from the Haskell image, and the base command is `ghci` with the necessary flags for compiling all modules. With the container up and running, it is sufficient to execute the main function, cf.

```
ghci> main
   ___    __    _      ___  _   _  _         __   _____  __   ___
  / _ \  /  \  | |    / __|| | | || |       /  \ |_   _|/   \|     \
 / /_\/ / /\ \ | |   | |   | | | || |      / /\ \  | |  | | ||    /
/ /_\\ / ___  \| |___| |__ | |_| || |___  / ___  \ | |  | | || |\ \
\____//_/   \_/ \____|\___|\____/  \____|/_/   \_/ |_|  \___/|_|  \_\

 Paulo Silva (paufil@di.uminho.pt)
 Universidade do Minho, Braga, Portugal

Galculator>
```

and the Galculator is on.

Within the scope of this study, it is imperative to understand the process of creating modules encompassing the laws, properties, and definitions pertinent to the programs whose calculations are sought. Equally important is to,learn how to conduct indirect equality proofs.

Modules and terminology Concerning modules, these must be stored in files with the `.gal` extension and must begin by specifying the module's name, e.g. `module integer`. The terminology used in the modules is quite user-friendly. Some examples that elucidate what one can expect to find in a Galculator module follow:

-
  ```
  EQUIV meet_assoc
        (MEET (MEET (Var r) (Var s)) (Var t))
        (MEET (Var r) (MEET (Var s) (Var t)));
  ```

  This equivalence reflects the associative property of the meet operation. As observed, `MEET` is part of a set of reserved words and in this case refers to the operator $\cap$. Note that the expressions are polymorphic, as they can take any expression — cf. the use of `Var`.

-
  ```
  EQUIV involution
        (CONV (CONV (Var r)))
        (Var r);

  EQUIV contravariance
        (CONV (COMP (Var r) (Var s)))
        (COMP (CONV (Var s)) (CONV (Var r)));
  ```

  The two equivalences presented just above — namely, contravariance and converse involution — emphasize the use of the reserved word `CONV` denoting the converse of a relation.

-
  ```
  DEF const (Fun (TVar a) (Prod (TVar a) (TVar b)));

  DEF zero INT;
  ```

```
DEF succ (Fun Int Int);

DEF leq (Ord Int);
DEF prefix (Ord (List (TVar t3)));
DEF leq_prefix (Ord (Prod Int (List (TVar t4))));
```

It is noteworthy to highlight function declaration via signatures declared from right to left. Please note that the constant function is defined in an uncurried way, as it takes a pair. (Recall the need for making functions uncurried due to the use of pointfree notation.) An integer `zero` and the function `succ` are defined, as well as the orders *less than or equal* in the natural numbers, prefix in lists, and the order as the product of these last two.

-
  ```
  EQUIV leq_zero
      (COMP (ORD (REF leq)) (FUN (LEFTSEC (REF const) (REF zero))))
      (FUN (LEFTSEC (REF const) (REF zero)));

  EQUIV prefix_nil
      (COMP (ORD (REF prefix)) (FUN (LEFTSEC (REF const) (REF
       ↪  empty_list))))
      (FUN (LEFTSEC (REF const) (REF empty_list)));
  ```

  The code above is the Galculator equivalent to:

  $$(\leqslant)\cdot\underline{0} = 0$$
  $$(\preccurlyeq)\cdot nil = nil$$

  One thus needs to define the functions $\underline{0}$ and nil, obtained through the use of `LEFTSEC`. This operator takes a function that accepts a pair of arguments and will fix the first argument — it corresponds to fixing arguments in Haskell's transposition.

The examples given above are sufficient to get a glimpse of how the Galculator is implemented. For a more comprehensive understanding, it is recommended to delve into some of the modules provided in the Galculator's Github repository.

Galois connections and proofs by indirect equality To establish a Galois connection in the Galculator, one only needs to input the lower adjoint, followed by the upper adjoint, the lower-side order and then the upper-side order right after `GDef <name_of_gc>`. For instance, the Galois connection that specifies *take* (5.1) is given by

```
GDef take_gc
    (APPLY (REF to_func) (SPLIT (FUN (REF length)) ID))
    (REF take)
    (REF leq_prefix)
    (REF prefix);
```

NB: `to_func` is used to retrieve the function embedded in a relation since `SPLIT` is, as defined in Galculator, a relation.

Indirect equality proofs are conducted within the command line using the `prove` command followed by the equality to be demonstrated. For instance, after composing within a module, say `take.gal` including all that is needed for proving the first base case of function *take* (5.6), the following commands are executed,

```
Galculator> load take
Galculator> prove EQUIV take_zero_id (COMP (FUN (REF take)) (PROD (FUN
 ↪  (LEFTSEC (REF const) (REF zero))) ID)) (FUN (LEFTSEC (REF const) (REF
 ↪  empty_list)))
Galculator> left
Galculator> indirect low (REF prefix)
```

indicating the intent to start the proof named *take_zero_id* from the left-hand side of the equality through indirect equality over the prefix order. This proof alongside with the second base case of *take* will be undertaken in the next section.

6.2 “Galculating” *uncurry take*

Let us recall the two base cases of function $\widehat{take}$, that is,

$$\widehat{take} \cdot (\underline{0} \times id) = nil$$
$$\widehat{take} \cdot (id \times nil) = nil$$

Also recall their calculations (5.6) and (5.7), respectively. Note that Galculator carries out the proof step by step, and some trivial steps, like associativity, are hidden, but the Galculator needs to perform them explicitly.

The first case starts with the pointfree equality $\widehat{take} \cdot (\underline{0} \times id) = nil$, cf. (5.6), and then it is stated that the proof starts from the left side of the equality. Subsequently, the proof by indirect equality over the prefix partial order is introduced, having the following first steps:[1]

```
Galculator> prove EQUIV take_zero_id (COMP (FUN (REF take)) (PROD (FUN
 ↪  (LEFTSEC (REF const) (REF zero))) ID)) (FUN (LEFTSEC (REF const) (REF
 ↪  empty_list)))
Galculator> show
Current proof:
```

[1] Some text of the Galculator logs is omitted to streamline the reading.

```
---------------------------------------------------------------
(take . ((<zero>const) >< id)) <=> (<empty_list>const)
---------------------------------------------------------------
Galculator> left
Galculator> indirect low (REF prefix)
Galculator> show
Current proof:
---------------------------------------------------------------
(take . ((<zero>const) >< id)) <=> (<empty_list>const)
---------------------------------------------------------------

(prefix . (take . ((<zero>const) >< id)))
Galculator>
```

Now, proceeding with the proof, one must use the associativity of composition. Since it is defined by

```
EQUIV comp_assoc
      (COMP (COMP (Var r) (Var s)) (Var t))
      (COMP (Var r) (COMP (Var s) (Var t)));
```

that is, it is inverted, one need to recur to the command `inv`, cf.

```
Galculator> inv apply comp_assoc
Galculator> show

((prefix . take) . ((<zero>const) >< id))
```

The proof proceeds by calling the shunting rule (automatically derived from the GC):

```
Galculator> once inv shunt take_gc
Galculator> show

(((to_func <length, id>)* . leq_prefix) . ((<zero>const) >< id))
```

Now, one must apply the associativity of composition again:

```
Galculator> apply comp_assoc
Galculator> show
Current proof:

((to_func <length, id>)* . (leq_prefix . ((<zero>const) >< id)))
```

Now, notice that the product of $\leqslant$ with $\preccurlyeq$ was defined as a specific order, meaning that the order is the product itself. Therefore, it is necessary to return the product of the two orders in a way to proceed with the definition of the product. This step is accomplished by the following equivalence

```
EQUIV leq_prefix_def (REF leq_prefix) (APPLY (REF to_ord) (PROD (ORD (REF
 ↪  leq)) (ORD (REF prefix))));
```

which uses the function *to_ord* that handles the retrieval of the order embedded in a relation, cf.

```
-- Retrieving the order embedded in a relation
DEF to_ord (Fun (Ord (TVar t22)) (Rel (TVar t22) (TVar t22)));
EQUIV to_ord_cancel (ORD (APPLY (REF to_ord) (Var o))) (Var o);
```

In this way, the following commands are executed:

```
Galculator> once apply leq_prefix_def
Galculator> once apply to_ord_cancel
Galculator> show
Current proof:

((to_func <length, id>)* . ((leq >< prefix) . ((<zero>const) >< id)))
```

The proof proceeds with the respective laws — ×-fusion, *id*-natural property, ×-definition, associativity of composition, “free-theorem” of a constant function, cf.

```
Galculator> once apply prod_fusion
Galculator> once apply natural_id_right
Galculator> once apply prod_def
Galculator> once apply comp_assoc
Galculator> once apply const_nat
Galculator> show
Current proof:

((to_func <length, id>)* . <(leq . (<zero>const)), (prefix . pi2)>)
```

Notice that `<length, id>` is a parameter of `to_func`. Recall this is needed because the `SPLIT` combinator within the Galculator definitions is defined as a relation. Thus, one has to retrieve the function embedded in that relation. The same happens with the identity function, since `ID` is defined as a relation. For that, one uses the following:

```
-- Retrieving the function embedded in a relation
DEF to_func (Fun (Fun (TVar t20) (TVar t21)) (Rel (TVar t20) (TVar t21)));
EQUIV to_func_cancel (FUN (APPLY (REF to_func) (Var f))) (Var f);
EQUIV to_func_id (APPLY (REF to_func) ID) FId;
```

Therefore, the proof continues as follows with

```
Galculator> once apply to_func_cancel
Galculator> show
Current proof:

(<length, id>* . <(leq . (<zero>const)), (prefix . pi2)>)
```

and the laws of pairing and converse, $id^{\circ} = id$, id-natural property and $(\leqslant) \cdot \underline{0} = \underline{0}$, cf.

```
Galculator> once apply pairing_converse
Galculator> once apply conv_id
Galculator> once apply natural_id_left
Galculator> once apply leq_zero
Galculator> show
Current proof:

((length* . (<zero>const)) /\ (prefix . pi2))
```

Now, recall that properties length$^{\circ} \cdot \underline{0} = nil$ and $nil \cap (\preccurlyeq \cdot \pi_2) = nil$ were used in the proof. Thus, one has to add these to the module,

```
-- conv length . zero = nil
EQUIV conv_length_zero
    (COMP (CONV (FUN (REF length))) (FUN (LEFTSEC (REF const) (REF zero))))
    (FUN (LEFTSEC (REF const) (REF empty_list)));

-- nil `cap` (pref . p2) = nil
EQUIV assert1
    (MEET (FUN (LEFTSEC (REF const) (REF empty_list))) (COMP (ORD (REF
     ↪  prefix)) (FUN (REF pi2))))
    (FUN (LEFTSEC (REF const) (REF empty_list)));
```

Using these equivalences one gets

```
Galculator> once apply conv_length_zero
Galculator> once apply assert1
Galculator> show
Current proof:

(<empty_list>const)
```

which is equivalent to $(\preccurlyeq) \cdot nil$ by (4.17), thus

```
Galculator> inv apply prefix_nil
Galculator> show
Current proof:

(prefix . (<empty_list>const))
```

Thus, the proof by indirect equality is now complete:

```
Galculator> indirect end
Galculator> qed
Galculator> show

Qed
```

The second case, cf. (5.7), will be similar to the previous. Thus let us look to the commands used:

```
Galculator> prove EQUIV take_id_nil (COMP (FUN (REF take)) (PROD ID (FUN
 ↪  (LEFTSEC (REF const) (REF empty_list))))) (FUN (LEFTSEC (REF const)
 ↪  (REF empty_list)))
Galculator> left
Galculator> indirect low (REF prefix)
Galculator> inv apply comp_assoc
Galculator> once inv shunt take_gc
Galculator> apply comp_assoc
Galculator> once apply leq_prefix_def
Galculator> once apply to_ord_cancel
Galculator> once apply prod_fusion
Galculator> once apply natural_id_right
Galculator> once apply prod_def
Galculator> once apply comp_assoc
Galculator> once apply const_nat
Galculator> once apply to_func_cancel
Galculator> once apply pairing_converse
Galculator> once apply conv_id
Galculator> once apply natural_id_left
Galculator> once apply prefix_nil
Galculator> apply assert2
Galculator> inv apply prefix_nil
Galculator> indirect end
Galculator> qed
Galculator> show
```

And now to the logs:[2]

```
---------------------------------------------------------------
(take . ((<zero>const) >< id)) <=> (<empty_list>const)
---------------------------------------------------------------
(take . ((<zero>const) >< id))

indirect low:

(prefix . (take . ((<zero>const) >< id)))

    { comp_assoc }
```

[2] Again, some text will be omitted to streamline the reading.

```
((prefix . take) . ((<zero>const) >< id))
    { Shunting: Shunting }
(((to_func <length, id>)* . leq_prefix) . ((<zero>const) >< id))
    { to_func_cancel }
((<length, id>* . leq_prefix) . ((<zero>const) >< id))
    { comp_assoc }
(<length, id>* . (leq_prefix . ((<zero>const) >< id)))
    { leq_prefix_def }
(<length, id>* . ((to_ord (leq >< prefix)) . ((<zero>const) >< id)))
    { to_ord_cancel }
(<length, id>* . ((leq >< prefix) . ((<zero>const) >< id)))
    { prod_fusion }
(<length, id>* . ((leq . (<zero>const)) >< (prefix . id)))
    { natural_id_right }
(<length, id>* . ((leq . (<zero>const)) >< prefix))
    { prod_def }
(<length, id>* . <((leq . (<zero>const)) . pi1), (prefix . pi2)>)
    { comp_assoc }
(<length, id>* . <(leq . ((<zero>const) . pi1)), (prefix . pi2)>)
    { const_nat }
(<length, id>* . <(leq . (<zero>const)), (prefix . pi2)>)
    { pairing_converse }
((length* . (leq . (<zero>const))) /\ (id* . (prefix . pi2)))
    { conv_id }
((length* . (leq . (<zero>const))) /\ (id . (prefix . pi2)))
    { natural_id_left }
```

```
((length* . (leq . (<zero>const))) /\ (prefix . pi2))

    { leq_zero }

((length* . (<zero>const)) /\ (prefix . pi2))

    { conv_length_zero }

((<empty_list>const) /\ (prefix . pi2))

    { assert1 }

(<empty_list>const)

    { prefix_nil }

indirect end

Qed
```

Concerning the general case, its proof is too long to be presented in this dissertation and actually dispensable, since it does not bring any new insight. So it has been decided not to include it, as it follows the same structure as the previous ones.

The Galculator proves to be the right tool for this type of proof by indirect equality in Galois connections. Of course, the take function does not cover all the intricacies studied in this work, particularly the use of coreflexives. Therefore, it would be interesting to see how the Galculator performs with the use of coreflexives, an experience that is left for future work.

6.3 Summary

The Galculator proof assistant was "awoken" from its legacy state, requiring some code refactoring due to updates in the Haskell programming language and the associated libraries.

Upon launching the Galculator, a succinct explanation was given on how to use the proof assistant, including key commands and instructions concerning loading the necessary modules containing all the required information for proofs.

Finally, the base cases of the function *take* were given as examples of using the tool in concrete programming situations.

Chapter 7

Conclusions

The starting motivation for the work reported in this dissertation was a critical inquiry into the efficacy and establishment of Software Engineering as a robust body of knowledge since its start in 1968. Indeed, the fundamental principles then proposed should be assessed in light of the ensuing five decades. Regrettably, the field has been grappled with pervasive challenges, including deficient software quality, exorbitant development costs and unwieldy team sizes. These issues have culminated in the production of sub-optimal software products, falling short of the engineering ideals envisioned.

To ensure the reliability and safety of critical software systems, rigorous proof methods are essential. However, widespread adoption of such methods, often reliant on complex inductive proofs, has proved difficult.

This research decided to go back to first principles in its adoption of a correct-by-construction (CbC) methodology, capable of ensuring software correctness from its very construction process. Such an approach aligns with engineering principles and emphasizes correctness from the outset.

In particular, the exploration of recursive computations specified by and derived from Galois connections whose proofs do not require induction eventually became the main theme of the dissertation. This method shows promise for deriving functionally correct implementations by construction.

Overall, correct-by-construction methodologies of this kind present a potential avenue for advancing Software Engineering as a *bona fide* engineering discipline. By prioritizing correctness and integrating rigorous techniques, this approach is hopefully a starting point for researchers and engineers to eventually seek improvements in the quality and reliability of the software systems they build.

7.1 Summary of contributions

This work's main contribution is on Galois Connection-based programming — on how to find a method for calculating programs from their specifications written as GCs. In particular, it focuses on functions that involve the Peano and finite list algebras, establishing them as valid partial orders through rigorous proofs. Handling of predicates within Galois connections is addressed too, an ingredient needed and illustrated with functions from the Haskell Standard Prelude that recur to the use of filters and predicates. This has led to a repertoire of such functions specified by Galois connections and how to calculate their implementation through indirect equality.

Hopefully, this provides valuable insight into how to document and develop such widely used libraries. In some cases, another crucial aspect of the overall strategy was illustrated: deriving useful properties of a function before even implementing it.

Along this process, the approach also showed potential for library function classification. Indeed, some functions were identified as clearly not being adjoints of Galois connections, even though some initially appeared to be. Those that proved to be so were documented and, in one case, its implementation was calculated through indirect equality, highlighting the power of this calculational approach.

Bearing automation and proof support in mind, the importance of reasoning at pointfree level in program calculation was emphasized. In particular, specific functions, namely $take$, $takeWhile$ and zip, were recalculated at the pointfree level. Notably, the interplay between catamorphisms and adjunctions played a pivotal role in calculating $take$ and zip.

The fact that Galois-connections at pointfree level are relational equalities aligns the approach with the framework implemented by the Galculator proof assistant, which was found in legacy state since its development in Haskell by Silva (2009).

After some refactoring due to updates in the Haskell programming language and associated libraries, the Galculator was "awoken" and proved to be the right proof assistant for helping making the approach defended in this thesis practical. In this respect, the preceding chapter provided some practical guidance on using the Galculator proof assistant. The proof process is exemplified through the concrete example of proving the base cases of the $take$ function.

7.2 Future work

As is to be expected in a project of this kind, the research often raises more questions than provides answers. One particular concern has to do with scalability. In this respect, delving into the foundational aspects concerning the additive category of GCs as presented in (Oliveira, 2020a) promises a clear scale-up in GC-based reasoning. This subject holds promise, particularly in the exploration of biproduct constructs which offer (for free) interesting ways for combining and fine-tuning GCs in a scalable, correct-by-construction way.

Another future endeavor for this thesis is to generalize the algebras that were studied, a generalization that seems to be required in some particular situations. For example, the Peano algebra can be generalized as

$$\begin{tikzcd} \mathbb{N}_0 \arrow[rr, bend left, "\mathsf{out}_{(\mathbb{N}_0,k)}"] \arrow[rr, phantom, "\cong"] & & \mathbb{N}_0 + \mathbb{N}_0 \arrow[ll, bend left, "{[id\,,(k+)]}"] \end{tikzcd} \tag{7.1}$$

parametric on k:

$$\mathsf{in}_{(\mathbb{N}_0,k)} = [id\,,(k+)]$$
$$\mathsf{out}_{(\mathbb{N}_0,k)}\ n = \textbf{if}\ n < k\ \textbf{then}\ i_1\ \ n\ \textbf{else}\ \ i_2\ \ (n-k)$$

(Note that, by instantiating $k := 1$ one obtains the previously analyzed Peano algebra.) The corresponding catamorphism combinator is defined by[1]

$$\langle\!| g |\!\rangle_k = g \cdot (id + \langle\!| g |\!\rangle_k) \cdot \mathsf{out}_{(\mathbb{N}_0,k)}$$

With this generalization, implementing, for instance, the algorithms for integer division and remainder is straightforward:

$$(\div y) = \langle\!| [\underline{0}, succ] |\!\rangle_y$$
$$(\%\ y) = \langle\!| [id\,, id] |\!\rangle_y$$

It is known that the algorithm for $gcd\ (m,n)$ (greatest common divisor) is specified as *the greatest divisor that divides both* m *and* n, i.e.,

$$z \mid m \wedge z \mid n \;\equiv\; z \mid gcd\ (m,n) \tag{7.2}$$

for the yet-to-be-defined divisibility partial order $(|)$. The intention is to assess the feasibility of defining this ordering employing the catamorphism above over this new algebra and to calculate function gcd.

In the trip through the standard Haskell libraries searching for GC-specifiable functions, some were found particularly tricky to handle. One illustrative example is *nub*. Intuitively, it seems to *yield the longest sublist of a sequence whose elements do no repeat*, that is:

$$bag\, z \leqslant \underline{1} \wedge z \sqsubseteq l \;\equiv\; z \sqsubseteq nub\, l \tag{7.3}$$

Recall that $bag : A^* \rightarrow {\mathbb{N}_0}^A$ is the function that, given a finite sequence (list), indicates the number of occurrences of any element. It may be defined by:

[1] The traditional banana brackets were replaced by triangular ones in order to distinguish the algebras.

```
bag          :: Eq a => [a] -> a -> Int
bag [] _     = 0
bag (h:t) x = bag t x + (if x == h then 1 else 0)
```

However, (7.3) is an over simplification inadequate for what the function actually does: *nub* cannot yield just the longest sublist, because there may exist more than one and, in that case, it must return the one that preserves the order of the initial occurrences. As an example, consider *nub* $[1,2,1,3,2,1,0] = [1,2,3,0]$. Sublist $[1,3,2,0]$ is also one of the longest sublists whose elements do not repeat but the order of the initial occurrences must be preserved — 2 appears before 3. The aim is to reformulate the sublist partial order in a manner that ensures this nuance, thus correctly calculating the function *nub*.

In the field of the inductive partial orders that were shown to underpin the derivation of GC-adjoints as recursive functions, some orderings were found to be harder to handle than others. For instance, the antisymmetry of the sublist ordering still needs to be proved, as the converse of this order is not a catamorphism. Therefore, an alternative approach to the proof needs to be explored. Another example is the suffix ordering which, not being a relational catamorphism, needs a different strategy to prove its partial order basic properties.

Finally, as previously mentioned about the Galculator proof assistant, the presented proofs were limited to function *take*, and the use of coreflexives and predicates was not tested. It would be interesting to observe how Galculator performs with proofs like *takeWhile*, whose pointfree calculation is provided in Section 5.2. Additionally, implementing the `auto` command, which attempts to automatically complete the proof, would be beneficial. This is because it was observed that the proofs follow a very similar structure. Furthermore, it would be even more intriguing if, with this automation, Galculator could autonomously draw conclusions about real-life, sizeable specifications. This would (will?) be indeed software correct-by-construction.

Bibliography

K. Backhouse and R. Backhouse. Safety of abstract interpretations for free, via logical relations and galois connections. *Science of Computer Programming*, 51(1):153–196, 2004. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2003.06.002. URL https://www.sciencedirect.com/science/article/pii/S0167642304000164. Mathematics of Program Construction (MPC 2002).

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

R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall, 1997.

I. Bogost. Programmers: Stop calling yourselves engineers. *The Atlantic*, November 2015. URL https://www.theatlantic.com/technology/archive/2015/11/programmers-should-not-call-themselves-engineers/414271/.

T. Bordis, L. Cleophas, A. Kittelmann, T. Runge, I. Schaefer, and B. W. Watson. Re-corc-ing key: Correct-by-construction software development based on key. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen, editors, *The Logic of Software. A Tasting Menu of Formal Methods*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 80–104, Germany, 2022. Springer. ISBN 978-3-031-08165-1. doi: 10.1007/978-3-031-08166-8_5. Publisher Copyright: © 2022, Springer Nature Switzerland AG.

B. Cohen. A brief history of formal methods. *Formal Aspects of Computing*, 01 1995.

E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

E.W. Dijkstra. Indirect equality enriched (and a proof by netty). https://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1315.PDF, 2001.

G. Gonthier. Formal proof — the fourcolor theorem. *Notices of the AMS*, 2008. URL https://www.ams.org/notices/200811/tx081101382p.pdf.

C.B. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, Upper Saddle River, NJ, USA, 1980. ISBN 0138218846. C.A.R. Hoare (series editor).

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

P. Naur and B. Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO, 1969.

J.N. Oliveira. A note on the under-appreciated for-loop. Technical Report TR-HASLab:01:2020 (pdf), HASLab/U.Minho and INESC TEC, 2020a.

J.N. Oliveira. Biproducts of galois connections, January 2020b. Presentation at the IFIP WG 2.1 #79 Meeting, Otterlo, NL. (Slides available from the author's website: https://ifipwg21wiki.cs.kuleuven.be/IFIP21/Otterlo).

J.N. Oliveira. Program Design by Calculation, 2022. Draft of textbook in preparation, current version: September 2022. Informatics Department, University of Minho (PDF).

J.N. Oliveira. Why adjunctions matter — a functional programmer perspective. In *Proceedings WADT'22*, LNCS. Springer-Verlag, 2023. In the press.

M. Roggenbach, A. Cerone, B.-H. Schlingloff, G. Schneider, and S. A. Shaikh. *Formal Methods for Software Engineering - Languages, Methods, Application Domains*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2022. ISBN 978-3-030-38799-0. doi: 10.1007/978-3-030-38800-3. URL https://doi.org/10.1007/978-3-030-38800-3.

P.F. Silva. *On the Design of a Galculator*. PhD thesis, University of Minho, May 2009.

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

M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, Amsterdam; Oxford, 2006. ISBN 0444520775 9780444520777. URL https://www.worldcat.org/title/lectures-on-the-curry-howard-isomorphism/oclc/1171193011&referer=brief_results.

Part III
Appendices

Appendix A
Proofs

Proof 1

$$
\begin{array}{ll}
 & (\mathsf{length} \cdot cons)^\circ \cdot (\leqslant) \cdot succ \\
= & \qquad \{ \ \text{length definition} \ \} \\
 & (succ \cdot \mathsf{length} \cdot \pi_2)^\circ \cdot (\leqslant) \cdot succ \\
= & \qquad \{ \ (\leqslant) \ \text{definition ; distribution (twice) ; contravariance} \ \} \\
 & \pi_2^\circ \cdot \mathsf{length}^\circ \cdot succ^\circ \cdot \underline{0} \cdot (\leqslant) \ \cup \ \pi_2^\circ \cdot \mathsf{length}^\circ \cdot succ^\circ \cdot succ \cdot (\leqslant) \\
= & \qquad \{ \ succ \ \text{is injective} \ \} \\
 & \pi_2^\circ \cdot \mathsf{length}^\circ \cdot succ^\circ \cdot \underline{0} \cdot (\leqslant) \ \cup \ \pi_2^\circ \cdot \mathsf{length}^\circ \cdot (\leqslant) \\
= & \qquad \{ \ succ^\circ \cdot \underline{0} = \bot \ \} \\
 & \pi_2^\circ \cdot \mathsf{length}^\circ \cdot (\leqslant) \\
\Box &
\end{array}
$$

Proof 2

$$
\begin{array}{ll}
 & \langle \pi_1 \times \mathsf{map}\ \pi_1, \pi_2 \times \mathsf{map}\ \pi_2 \rangle^\circ \\
= & \qquad \{ \ \text{pairing definition} \ \} \\
 & (\pi_1^\circ \cdot (\pi_1 \times \mathsf{map}\ \pi_1) \ \cap \ \pi_2^\circ \cdot (\pi_2 \times \mathsf{map}\ \pi_2))^\circ \\
= & \qquad \{ \ \text{converses} \ \} \\
 & (\pi_1^\circ \times \mathsf{map}\ \pi_1^\circ) \cdot \pi_1 \ \cap \ (\pi_2^\circ \times \mathsf{map}\ \pi_2^\circ) \cdot \pi_2 \\
= & \qquad \{ \ \times\text{-definition} \ \} \\
 & \langle \pi_1^\circ \cdot \pi_1, \mathsf{map}\ \pi_1^\circ \cdot \pi_2 \rangle \cdot \pi_1 \ \cap \ \langle \pi_2^\circ \cdot \pi_1, \mathsf{map}\ \pi_2^\circ \cdot \pi_2 \rangle \cdot \pi_2 \\
= & \qquad \{ \ \text{pairing definition} \ \}
\end{array}
$$

$$(\pi_1^\circ \cdot \pi_1^\circ \cdot \pi_1 \;\cap\; \pi_2^\circ \cdot \mathsf{map}\ \pi_1^\circ \cdot \pi_2) \cdot \pi_1 \;\cap\; (\pi_1^\circ \cdot \pi_2^\circ \cdot \pi_1 \;\cap\; \pi_2^\circ \cdot \mathsf{map}\ \pi_2^\circ \cdot \pi_2) \cdot \pi_2$$

$=$ { distributivity twice }

$$(\pi_1^\circ \cdot \pi_1^\circ \cdot \pi_1 \cdot \pi_1 \;\cap\; \pi_2^\circ \cdot \mathsf{map}\ \pi_1^\circ \cdot \pi_2 \cdot \pi_1) \;\cap\; (\pi_1^\circ \cdot \pi_2^\circ \cdot \pi_1 \cdot \pi_2 \;\cap\; \pi_1^\circ \cdot \mathsf{map}\ \pi_2^\circ \cdot \pi_2 \cdot \pi_2)$$

$=$ { $\cap$ -associativity }

$$(\pi_1^\circ \cdot \pi_1^\circ \cdot \pi_1 \cdot \pi_1 \;\cap\; \pi_1^\circ \cdot \pi_2^\circ \cdot \pi_1 \cdot \pi_2) \;\cap\; (\pi_2^\circ \cdot \mathsf{map}\ \pi_1^\circ \cdot \pi_2 \cdot \pi_1 \;\cap\; \pi_2^\circ \cdot \mathsf{map}\ \pi_2^\circ \cdot \pi_2 \cdot \pi_2)$$

$=$ { distributivity twice }

$$\pi_1^\circ \cdot (\pi_1^\circ \cdot \pi_1 \cdot \pi_1 \;\cap\; \pi_2^\circ \cdot \pi_1 \cdot \pi_2) \;\cap\; \pi_2^\circ \cdot (\mathsf{map}\ \pi_1^\circ \cdot \pi_2 \cdot \pi_1 \;\cap\; \mathsf{map}\ \pi_2^\circ \cdot \pi_2 \cdot \pi_2)$$

$=$ { pairing definition; pairing and converse }

$$\pi_1^\circ \cdot \langle \pi_1 \cdot \pi_1, \pi_1 \cdot \pi_2 \rangle \;\cap\; \pi_2^\circ \cdot \langle \mathsf{map}\ \pi_1, \mathsf{map}\ \pi_2 \rangle^\circ \cdot \langle \pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle$$

$=$ { $\times$-definition twice; pairing and converse }

$$\langle \pi_1, \langle \mathsf{map}\ \pi_1, \mathsf{map}\ \pi_2 \rangle \cdot \pi_2 \rangle^\circ \cdot \langle \pi_1 \times \pi_1, \pi_2 \times \pi_2 \rangle$$

$=$ { $\times$-functor and *transpose* definition }

$$(id \times \langle \mathsf{map}\ \pi_1, \mathsf{map}\ \pi_2 \rangle)^\circ \cdot transpose$$

□

Proof 3

$$nil \;\subseteq\; \mathsf{length}^\circ \cdot (\leqslant) \cdot \pi_1$$

$\equiv$ { constants' natural property }

$$nil \cdot \pi_1 \;\subseteq\; \mathsf{length}^\circ \cdot (\leqslant) \cdot \pi_1$$

$\equiv$ { shunting; $\mathrm{img}\ \pi_1$ is surjective }

$$nil \;\subseteq\; \mathsf{length}^\circ \cdot (\leqslant)$$

$\equiv$ { (5.4) }

$$True$$

□

Appendix B
Laws of functional and relation calculus

Lowercase letters denote functions, and uppercase letters denote relations. Uppercase letters in sans serif font indicate functors and relators.

Composition

Pointwise definition	$b\,(R.S)\,c \;\equiv\; \langle \exists\, a : b\,R\,a : a\,S\,c \rangle$
Identity	$R \cdot id = id \cdot R = R$
Associativity of composition	$R \cdot (S \cdot T) = (R \cdot S) \cdot T$

Converses

Universal	$X^{\circ} \subseteq Y \;\equiv\; X \subseteq Y^{\circ}$
Involution	$(R^{\circ})^{\circ} = R$
Contravariance	$(R \cdot S)^{\circ} = S^{\circ} \cdot R^{\circ}$
Isomorphism	$R \subseteq S = S^{\circ} \subseteq R^{\circ}$
"A pocket rule"	$b\,(f^{\circ} \cdot R \cdot g)\,a \;\equiv\; (f\,b)\,R\,(g\,a)$

Relation inclusion

Pointwise	$R \subseteq S \Leftrightarrow \langle \exists\, a,b :: b\,R\,a \Rightarrow b\,S\,a \rangle$
Reflection	$R \subseteq R$
Transitivity	$R \subseteq S \wedge S \subseteq T \Rightarrow R \subseteq T$
Top and bottom	$\bot \subseteq R \subseteq \top$
Absorption	$R \cdot \bot = \bot \cdot R = \bot$

Relation equality

Pointwise	$R = S \Leftrightarrow \langle \forall\, a,b : (a \in A \wedge b \in B) : b\,R\,a \Leftrightarrow b\,S\,a \rangle$
Indirect equality (1)	$R = R \;\equiv\; \langle \forall X :: X \subseteq R \Leftrightarrow X \subseteq S \rangle$
Indirect equality (2)	$R = R \;\equiv\; \langle \forall X :: R \subseteq X \Leftrightarrow S \subseteq X \rangle$

Functions

Shunting (1)	$f \cdot R \subseteq S \equiv R \subseteq f^{\circ} \cdot S$
Shunting (2)	$R \cdot f^{\circ} \subseteq S \equiv R \subseteq S \cdot f$

Constants

Natural property (functions)	$\underline{k} \cdot f = \underline{k}$
Natural property (relations)	$\underline{k} \cdot R \subseteq \underline{k}$
Truth functions	$true = \underline{True} \wedge false = \underline{False}$
Fusion	$f \cdot \underline{k} = \underline{f\ k}$

Relation union

Universal property	$R \cup S \subseteq X \equiv R \subseteq X \wedge S \subseteq X$
Right linearity	$R \cdot (S \cup T) = (R \cdot S) \cup (R \cdot T)$
Left linearity	$(S \cup T) \cdot R = (S.R) \cup (T.R)$
Converse	$(R \cup S)^{\circ} = R^{\circ} \cup S^{\circ}$
Bottom	$R \cup \bot = R$
Top	$R \cup \top = \top$

Relation intersection

Universal property	$X \subseteq R \cap S \equiv X \subseteq R \wedge X \subseteq S$
Right distribution	$(S \cap Q) \cdot R = (S.R) \cap (Q.R) \Leftarrow \begin{array}{c} Q \cdot \mathrm{img}\, R \subseteq Q \\ \vee \\ S \cdot \mathrm{img}\, R \subseteq S \end{array}$
Left distribution	$R \cdot (Q \cap S) = (R.Q) \cap (R.S) \Leftarrow \begin{array}{c} \ker R \cdot Q \subseteq Q \\ \vee \\ \ker R \cdot S \subseteq S \end{array}$
Converse	$(R \cap S)^{\circ} = R^{\circ} \cap S^{\circ}$
Bottom	$R \cap \bot = \bot$
Top	$R \cap \top = R$

Pairing and products

Pairing universal property	$X \subseteq \langle R,S\rangle \Leftrightarrow \pi_1 \cdot X \subseteq R \wedge \pi_2 \cdot X \subseteq S$
Pairing Fusion	$\langle R,S\rangle \cdot T = \langle R\cdot T, S\cdot T\rangle \Leftarrow \begin{array}{c} R\cdot \mathrm{img}\,T \subseteq R \\ \vee \\ S\cdot \mathrm{img}\,T \subseteq S \end{array}$
Fusion (functions)	$\langle R,S\rangle \cdot f = \langle R.f, S.f\rangle$
Pairing definition	$\langle R,S\rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S$
Pairing and converse	$\langle R,S\rangle^\circ \cdot \langle X,Y\rangle = R^\circ \cdot X \cap S^\circ \cdot Y$
Functor-× composition	$(R\times S)\cdot(P\times Q) = (R\cdot P)\times(S\cdot Q)$
Functor-× definition	$R\times S = \langle R\cdot\pi_1, S\cdot\pi_2\rangle$
Functor-× converse	$(R\times S)^\circ = R^\circ \times S^\circ$

Coproducts

Universal property	$X = [R,S] \Leftrightarrow X\cdot i_1 = R \wedge X\cdot i_2 = S$
Fusion	$R\cdot[S,T] = [R\cdot S, R\cdot T]$
Definition	$[R,S] = R\cdot i_1{}^\circ \cup S\cdot i_2{}^\circ$
Absorption	$[R,S]\cdot(P+Q) = [R\cdot P, S\cdot Q]$
Either and converse	$[R,S]\cdot[T,U]^\circ = R\cdot T^\circ \cup S\cdot U^\circ$
i_1-natural property	$(i+j)\cdot i_1 = i_1\cdot i$
i_2-natural property	$(i+j)\cdot i_2 = i_2\cdot j$
Functor-+ composition	$(R+S)\cdot(P+Q) = (R\cdot P)+(S\cdot Q)$
Functor-+ definition	$R+S = [i_1\cdot R, i_2\cdot S]$
Functor-+ converse	$(R+S)^\circ = R^\circ + S^\circ$

Catamorphisms

Universal property	$X = \llparenthesis R \rrparenthesis \equiv X\cdot \mathsf{in}_\mathsf{T} = R\cdot(\mathsf{F}\,X)$
Monotonicity	$\llparenthesis R \rrparenthesis \subseteq \llparenthesis S \rrparenthesis \Leftarrow R \subseteq S$
Fusion	$S\cdot\llparenthesis R \rrparenthesis = \llparenthesis Q \rrparenthesis \Leftarrow S\cdot R = Q\cdot \mathsf{F}\,S$
Fusion weaker version 1	$Q\cdot\llparenthesis S \rrparenthesis \subseteq \llparenthesis R \rrparenthesis \Leftarrow Q\cdot S \subseteq R\cdot\mathsf{F}\,Q$
Fusion weaker version 2	$\llparenthesis R \rrparenthesis \subseteq Q\cdot\llparenthesis S \rrparenthesis \Leftarrow R\cdot\mathsf{F}\,Q \subseteq Q\cdot S$
Reflection	$\llparenthesis \mathsf{in}_\mathsf{T} \rrparenthesis = id_\mathsf{T}$

Eindhoven quantifier calculus The notation standards employed by the Eindhoven quantifier calculus are as follows:

- $\langle \forall x : R : T \rangle$ means *for **all** x in the range R, term T holds*, where R and T are logic expressions involving x.
- $\langle \exists x : R : T \rangle$ means *for **some** x in the range R, term T holds*, where R and T are logic expressions involving x.

Below are some of the primary rules of the Eindhoven quantifier calculus:

Trading-∀	$\langle \forall k : R \wedge S : T \rangle = \langle \forall k : R : S \Rightarrow T \rangle$
Trading-∃	$\langle \exists k : R \wedge S : T \rangle = \langle \exists k : R : S \wedge T \rangle$
One-point-∀	$\langle \forall k : k = e : T \rangle = T\,[k := e]$
One-point-∃	$\langle \exists k : k = e : T \rangle = T\,[k := e]$