Paulo Filipe Araújo da Silva

On the Design of a Galculator

Agosto 2009

Paulo Filipe Araújo da Silva **On the Design of a Galculator**

UMinho|2009

Universidade do Minho
Escola de Engenharia

Paulo Filipe Araújo da Silva

On the Design of a Galculator

Doutoramento em Informática
Ramo de conhecimento em Fundamentos da Computação

Trabalho efectuado sob a orientação do
Professor Doutor José Nuno Fonseca de Oliveira

Agosto 2009

Acknowledgments

First of all, I would like to thank José Nuno Oliveira, who kindly accepted being my supervisor. He is a continuous source of ideas and inspiration, and his pursuit of simplicity has been an example to me. Moreover, his contagious enthusiasm and energy infect all who work with him.

Joost Visser has been another great influence in my work. His sharp comments together with his ability of asking the "hard" questions always stimulated me to go further. The few moments I have shared with him were invaluable to this dissertation.

I thank João Saraiva for all the talks in which he tried to persuade me to embrace research. I also thank Luís Barbosa and José João Almeida for "saving" me on the deadline of my grant application.

Working becomes much more pleasant when our colleagues are also our friends. I am grateful to all members of *"Os Sem Estatuto"* for the great working atmosphere and the good moments spent together. A special mention goes to those who have shared the office with me during these years: Alexandra, Francisco, Jácome, João, João Paulo, Nélio, Sara, Tiago, Vilaça, and Zé, not forgetting our foreign visitors, Fábio and Pablo.

I apologize to my friends for the time I was absent because of this work. Nevertheless, they have been always available to support me whenever I needed, for which I am grateful.

Nothing of this would be possible without the unconditional support from my parents. All their effort and love is deeply acknowledged.

I would like to dedicate this dissertation to Paula. She entered my life during the time of this work like a promise fulfilled. I fell as if I have reborn after I met her. Paula, I love you!

This dissertation was supported by the Fundação para a Ciência e a Tecnologia, Portugal, under grant number `SFRH/BD/19195/2004`.

On the Design of a *Galculator*

The increasing complexity of software systems together with the lack of tools and techniques to support their development has led to the so-called "software crisis". Different views about the problem originated diverse approaches to a possible solution, although it is now generally accepted that a "silver bullet" does not exist.

The formal methods view considers mathematical reasoning as fundamental to fulfill the most important property of software systems: correctness. However, since correctness proofs are generally difficult and expensive, only critical applications are regarded as potential targets for their use. Developments in tool support such as proof assistants, model checkers and abstract interpreters allow for reducing this cost and making proofs affordable to a wider range of applications. Nevertheless, the effectiveness of a tool is highly dependent of the underlying theory.

This dissertation follows a calculational proof style in which equality plays the fundamental role. Instead of the traditional logical approach, fork algebras, an extension of relation algebras, are used. In this setting, Galois connections are important because they reinforce the calculational nature of the algebraic approach, bringing additional structure to the calculus. Moreover, Galois connections enjoy several valuable properties and allow for transformations in the domain of problems. In this dissertation, it is shown how fork algebras and Galois connections can be integrated together with the indirect equality principle. This combination offers a very powerful, generic device to tackle the complexity of proofs in program verification. This power is enhanced with the design of an innovative proof assistant prototype, the *Galculator*.

Concepção e Implementação de um *Galculator*

O aumento da complexidade dos sistemas de *software*, juntamente com a falta de ferramentas e técnicas de suporte ao seu desenvolvimento, originaram a chamada "crise do *software*". Diferentes visões sobre o problema resultaram em várias abordagens a uma possível solução, embora, actualmente, se considere que não existe uma "solução milagrosa". A visão dos métodos formais encara o raciocínio matemático como fundamental para satisfazer a propriedade mais importante de um sistema de *software*: a correcção. Todavia, sendo as provas de correcção, geralmente, difíceis e dispendiosas, apenas as aplicações críticas são encaradas como alvos potenciais para a sua utilização. Desenvolvimentos em ferramentas de suporte, como assistentes de prova, *model checkers* e interpretadores abstractos, permitem reduzir esse custo, tornando as provas acessíveis a um leque mais alargado de aplicações. Apesar disso, a aplicabilidade destas ferramentas é muito dependente da teoria subjacente.

Esta dissertação segue um estilo de prova baseado em cálculo, em que a igualdade assume o papel fundamental. Em vez da tradicional abordagem lógica, são utilizadas *fork algebras*, uma extensão das algebras de relações. Neste contexto, as conexões de Galois são importantes pois reforçam a natureza baseada no cálculo conferida pela abordagem algébrica, assim como a sua estrutura. Para mais, as conexões de Galois gozam de várias propriedades interessantes, além de permitirem transformações no domínio dos problemas. Nesta dissertação mostra-se como as *fork algebras* e as conexões de Galois podem ser integradas com o princípio da igualdade indirecta. Esta combinação oferece um dispositivo poderoso e genérico na abordagem às complexas provas da verificação de programas. Este poder é reforçado com a concepção e implementação de um protótipo de um assistente de prova, o *Galculator*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The introduction of the general-purpose electronic computer is currently seen as a milestone in history, paving the way for the so-called "Information Era". Computers brought not only unprecedented computation power but also new approaches to information processing and dissemination, dramatically changing processes and organizations. After less more than half a century, computer systems became ubiquitous in our world, ranging from large scale supercomputers to smart cards embedded systems.

The hardware industry success is unparalleled by any other technology in civilization history, attaining steady gains in both price and performance [Brooks, 1987]. However, software was never able to mimic the increasing power of machines. In the first times, many thought that the problem of programming being so difficult was due to the limitations of the early computers [Dijkstra, 1972]. This believe was dismissed as machines evolved, mostly because the perceived power of computers always demands for applications capable of using it. People started to realize that software is not able to follow the rate of improvements of hardware in speed, cost and reliability [Dijkstra, 1972].

Software crisis. The demand for more complex applications led to system failures, lack of performance, increased costs of development or missed deadlines. In 1968, a group of eminent computer scientists joined to discuss the situation in the First NATO Software Engineering Conference coined the term *"software crisis"* to describe this [Naur and Randell, 1969].

Over the years, several solutions to the problem were appointed: better programming languages, improved tools and methodologies, machine support, among other.

However, the expectations about what software systems can do continues to increase and consequently also their complexity. Currently, most accept that a *"silver bullet"* does not exist, i.e., a *"... single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity."* [Brooks, 1987]. Brooks [1987] justifies this impossibility by arguing that difficulties in software development are essential and not only accidental, i.e., they come directly from the complexity of the problems at hands. Even after eliminating all the accidental factors — inadequate languages, lack of tool support, management difficulties, etc. — the problem remains complex mostly because *"the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly."* [Brooks, 1987].

Formal methods. Although a miraculous solution does not seem to exist, several efforts have been made to improve the situation. The formal methods community argues that Mathematics and its associated techniques and reasoning are essential to ensure correctness, just like in any other mature engineering field. Several different theories, techniques and tools were developed with varying level of success. Some methods are still just too difficult and complex to use in an effective way. The ultimate objective would be to label software with the stamp *"correct inside"* to attest its quality.

Software correctness. Despite significant advances in the field, software correctness is still an ambitious challenge. Over the years, many techniques have been developed and applied in order to augment our confidence on programs we write, ranging from informal techniques and guidance principles to formal methods. The success of each of these methods varies greatly but there seems to be evidence that success is proportional to tool support [Jackson, 2006].

Logic based approaches benefit from the help of theorem provers in the conduction of proofs. Using annotations and tools such as *Why* and *Caduceus* [Filliâtre and Marché, 2007], programs can be verified and formal proof obligations be discharged. Ideally, all proofs should be fully automated but there are theoretical limits imposed by the undecidability of general predicate calculus.

Informal and formal proofs. *Informal proofs* abound in textbooks about mathematics and software engineering. They are simpler to write than their formal counterparts, still evidencing the basic principles which underly the reasoning. In fact, an informal proof is just a convincing argument for the existence of the corresponding formal proof. However, when proving software correctness, informal proofs are only useful when dealing with small algorithms or simple examples. In real software systems, one can easily overlook important details and non-trivial interactions since their complexity is high. Moreover, the verification of informal proofs is error prone and tedious; for very large systems it is just unfeasible.

Although difficult to build in general, *formal proofs* can be automatically verified using software programs known as theorem provers [McCarthy, 1962]. A formal proof follows from a set of axioms and previously proved theorems using inference rules; the proved statement becomes a theorem. Usually, natural deduction or Hilbert-style of deduction are used to build the proof. However, as discussed by Gries and Schneider [1995] and Lifschitz [2001], deriving proofs in these formal systems is not always an intuitive and easy task.

Equational reasoning. An alternative is to use equational reasoning instead. Axioms and theorems are expressed as equalities; inference rules are just substitutions of equals by equals. The goal must be proven from axioms or known theorems by using them as substitution rules. For instance, Gries and Schneider [1995] and Lifschitz [2001] prove how an equational calculus based on the ideas of Dijkstra and Scholten [1990] can be seen as a full-fledged deductive system.

Relation algebras. The simplicity of equational reasoning was one of factors that inspired Tarski to develop his calculus for the formalization of set theory [Tarski and Givant, 1987]. This is based on the De Morgan-Peirce-Schröder relation algebra and besides being equational, has no variables nor quantifiers and is thus “point-free” in style. Throughout logical proofs, nested quantifiers are messy and variables lead to side conditions in inference rules and substitutions in order to avoid the capture of free variables. As proved by Tarski and Givant [1987], relation algebra is equivalent in expressive and deductive power to a three variable fragment of first-order logic.

Fork algebras. As pointed out by Tarski and Givant [1987] and van den Bussche [2001] the absence of some kind of pairing operation causes the lack of expressiveness of relation algebra. Fork algebras extend relation algebras with a kind of pairing function, becoming equivalent to first-order logic in expressive and deductive power.

Furthermore, the approach is abstract and completely symbolic; however, a natural interpretation in terms of concrete binary relations exists.

Galois connections. Although an equational proof using relations (both fork and relation algebra) is easy to follow and mostly driven by the available equalities, it can benefit from the introduction of a general concept widely spread across several mathematical fields: Galois connections [Ore, 1944]. The best known application of Galois connections in computer science is perhaps that of *abstract interpretation* [Cousot and Cousot, 1977; Cousot, 2001]. References [Aarts et al., 1992; Backhouse and Backhouse, 2004; Backhouse, 2004; Erné et al., 1993; Denecke et al., 2004] provide a far more expressive account of such applications, ranging over the predicate calculus, parametric polymorphism, number theory, abstract algebra, topology, etc.

Basically, a Galois connection relates two functions (adjoints) between pre-ordered domains providing “shunting” laws between them thanks to their “good” preservation properties. Functions which are inverses of each other form a special case of a Galois connection where both orders are the equality relation. Often, problems in one of the domains are easier to solve than problems in the other domain. Using a Galois connection it is possible to map a “hard” problem to an equivalent but easier one in the other domain, to find its solution, and then map it back to the result in the original domain.

Galois connections form their own algebra, thus we can combine them in order to build arbitrarily complex connections. This ability makes the structure of a proof clearer while making it easier to scale up to more complex problems. Additionally, Galois connections have interesting properties that can also be exploited in proofs; if a function participates in a Galois connection it enjoys such general properties. Galois connection theory thus caters for genericity in proofs.

Why a *Galculator*. The rich algebra, genericity and scalability of the concept of a Galois connection lead to the idea of building a tool able to take advantage of such expressive power, termed *Galculator* (= **Gal**ois connections + cal**culator**), tuned to reasoning and calculating about software problems which can be modelled in terms of adjoints of Galois connections.

1.1 Motivating examples

It is often the case that practical application of a tool is hindered by the underlying theory itself, whenever this is too "heavy" for the problem in hands. In this section, we present two small examples about whole division to clarify the objectives of *Galculator*. The explanation is introductory and very lightweight; the later chapters will provide for deeper understanding of the involved concepts.

1.1.1 Correction of a whole division implementation

Let us consider a simple example: we want to prove the correctness of the following *Haskell* function[1]

$$\begin{aligned} x \,\text{`}\mathit{div}\text{`}\, y \mid x < y &= 0 \\ \mid x \geqslant y &= (x - y)\,\text{`}\mathit{div}\text{`}\, y + 1 \end{aligned}$$

which computes whole division, for non-negative x and positive y. A standard proof would involve some kind of induction [Manna et al., 1973], e.g., structural induction [Burstall, 1969] or fixed point induction [Park, 1969]. However, what we have above is *code* — where is the corresponding *specification*?

Let us denote such a specification by $x \div y$ (over the natural numbers), for which at least two definitions can be found in maths books, for $y > 0$: one *implicit*[2]

$$c = x \div y \quad \overset{\mathit{def}}{=} \quad \langle \exists\, r :\ 0 \leqslant r < y\ : x = c \times y + r \rangle \tag{1.1}$$

and the other *explicit*

$$x \div y \quad \overset{\mathit{def}}{=} \quad \langle \bigsqcup z :: z \times y \leqslant x \rangle \tag{1.2}$$

where notation $\langle \bigsqcup z :: p(z) \rangle$ means the largest z such that $p(z)$ holds, for a predicate p.

Checking the correctness of the given *Haskell* code against implicit definition (1.1) in the *Coq* proof assistant [Coquand and Huet, 1988] entails a number of steps which are described by Bertot and Castéran [2004] and Almeida [2008]. Still for the same

[1]Throughout this text, we resort to package `lhs2TeX` [Hinze and Löh, 2008] for type-setting symbols and code in *Haskell*.

[2]The notation for quantifiers is described in Chapter 2. By now, consider that $\langle \exists\, x :\ R\ : T \rangle$ means *there exists some x in the range R such that T holds.*

purpose, one might dive into real number arithmetics by defining $x \div y$ to be $(x - (x \mathbf{\ mod\ } y))/y$, and exploiting the properties of the *modulo* operator.

Correctness verification assumes that both specification and implementation are available before proofs take place. A different, more constructive alternative would be to *calculate* the implementation from the specification itself. In the current example, it can be observed that the following Galois connection [Ore, 1944] arises from (1.2),

$$z \times y \leqslant x \quad \Leftrightarrow \quad z \leqslant x \div y \qquad\qquad (y > 0) \tag{1.3}$$

assuming x, y, z universally quantified over the natural numbers. Note how this property matches with (1.2): fixing x and y and reading (1.3) as an implication from left to right, this already tells us that $x \div y$ is the largest z such that $z \times y \leqslant x$ holds.

A simple calculation of the given *Haskell* code can be performed based on two Galois connections: the one just given explaining whole division and the following,

$$a - b \leqslant c \quad \Leftrightarrow \quad a \leqslant c + b \tag{1.4}$$

which explains subtraction over the integers, another operator used in the algorithm. We can put these two connections together by restricting (1.3) to non-negative integers, and keeping $y \neq 0$. We reason:

$$\begin{array}{cl}
 & z \leqslant x \div y \\
\Leftrightarrow & \quad \{ \text{ Galois connection (1.3) assuming } x \geqslant 0, y > 0. \} \\
 & z \times y \leqslant x \\
\Leftrightarrow & \quad \{ \text{ Cancellation, thanks to (1.4). } \} \\
 & z \times y - y \leqslant x - y \\
\Leftrightarrow & \quad \{ \text{ Distribution law. } \} \\
 & (z - 1) \times y \leqslant x - y \\
\Leftrightarrow & \quad \{ \text{ (1.3) again, assuming } x - y \geqslant 0 \text{, that is, } x \geqslant y. \} \\
 & z - 1 \leqslant (x - y) \div y \\
\Leftrightarrow & \quad \{ \text{ (1.4) again. } \} \\
 & z \leqslant (x - y) \div y + 1
\end{array}$$

That is, every natural number z which is at most $x \div y$ (for $x \geqslant y$) is also at most

$(x - y) \div y + 1$ and vice versa. The principle whereby we may conclude that the two expressions are the same

$$x \div y \quad = \quad (x - y) \div y + 1 \tag{1.5}$$

thus calculating the second clause of the *div* function is known as *indirect equality* (see below). Concerning the first clause of the algorithm, we assume $x < y$ and reason in the same style:

$$\begin{array}{ll}
 & z \leqslant x \div y \\
\Leftrightarrow & \quad \{ \text{ (1.3). } \} \\
 & z \times y \leqslant x \\
\Leftrightarrow & \quad \{ \text{ Transitivity, since } x < y. \} \\
 & z \times y \leqslant x \wedge z \times y < y \\
\Leftrightarrow & \quad \{ \text{ Since } y \neq 0. \} \\
 & z \times y \leqslant x \wedge z \leqslant 0 \\
\Leftrightarrow & \quad \{ z \leqslant 0 \text{ entails } z \times y \leqslant x \text{, since } 0 \leqslant x. \} \\
 & z \leqslant 0
\end{array}$$

This time we get $x \div y = 0$ under the same principle which supported clause (1.5), known as the principle of *indirect equality* [Aarts et al., 1992]:

$$a = b \quad \Leftrightarrow \quad \langle \forall\, x :: x \leqslant a \Leftrightarrow x \leqslant b \rangle \tag{1.6}$$

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

The simple (non inductive) proof above illustrates the calculational power of Galois connections operated via indirect equality, a device which is applicable to arbitrarily complex problem domains. Prior to studying such generalization let us consider yet another calculational proof concerning whole division.

1.1.2 Simple property about whole division

Proving algebraic equalities can be a hard task even in presence of intuitively simple mathematical operators. Take equality $(a/b)/c = a/(c{\times}b)$, for $b, c \neq 0$, for instance. If a/b denotes division of two real numbers (in a field, in general), that is, $a/b = a \times b^{-1}$, the task is not difficult at all: $(a/b)/c = (a \times b^{-1}) \times c^{-1}$ yielding $a \times (c \times b)^{-1}$ almost at once.

Let, however, $a \div b$ denote the *whole division* of two natural numbers a and b ($b \neq 0$) as earlier on. Does $(a \div b) \div c = a \div (c \times b)$ still hold? It does but the proof is not so immediate because, although intuitive, the definition of division on natural numbers is not easy to manipulate, be it an implicit definition (Equation (1.1)), be it an explicit definition (Equation (1.2)) or be it defined by recursion like in the previous section leading to an inductive proof.

Altogether, difficulties clearly arise from the simple fact that the existence of multiplicative inverses, captured by equivalence

$$c \times b = a \quad \Leftrightarrow \quad c = a \times b^{-1} \qquad\qquad (b \neq 0) \tag{1.7}$$

is not ensured once we move from real to natural numbers. However, looking closer at the properties of whole division we can see that the two equalities in (1.7) can be weakened to inequalities leading us back to Galois connection (1.3), involving adjoint functions $(\times b)$ and $(\div b)$, for $b \neq 0$, valid in the natural numbers. Adopting the same strategy as before, a simple proof follows, where n is a universally quantified variable over natural numbers:

$$\begin{array}{cl}
& n \leqslant (a \div b) \div c \\
\Leftrightarrow & \qquad \{ \text{ By (1.3). } \} \\
& n \times c \leqslant a \div b \\
\Leftrightarrow & \qquad \{ \text{ (1.3) again. } \} \\
& (n \times c) \times b \leqslant a \\
\Leftrightarrow & \qquad \{ \text{ Multiplication is associative. } \} \\
& n \times (c \times b) \leqslant a \\
\Leftrightarrow & \qquad \{ \text{ (1.3) again. } \} \\
& n \leqslant a \div (c \times b) \ .
\end{array}$$

As before, we calculate that every natural number n at most $(a \div b) \div c$ is also at most $a \div (c \times b)$, and vice versa. By indirect equality, the two expressions are equal.

In retrospect, a fundamental ingredient in this surprisingly simple proof is the ability to transform an expression involving a "hard" operator (whole division) into an expression involving an "easy" one (multiplication). Also essential is the step of the proof in which associativity of multiplication is assumed; all other steps are a kind of "shunting" of operators between the two sides of each inequality. After these steps, all that is needed is to bring whole division back into the expression by "shunting" in the opposite direction. But, above all, it is the rule of indirect equality which implicitly shapes the whole strategy of the proof.

Generalization. Clearly, these ingredients can be put together in order to solve more complex problems. Let, for instance, multiplication and whole division in (1.3) be replaced by other operators which exhibit the same algebraic properties in a different domain: that of binary relations ordered by inclusion. In fact, Galois connection

$$X \circ R \subseteq Y \quad \Leftrightarrow \quad X \subseteq Y \,/\, R \tag{1.8}$$

holds for arbitrary relations X, R and Y operated by relational composition

$$b(R \circ S)c \quad \Leftrightarrow \quad \langle \exists\, a :: b\; R\; a \wedge a\; S\; c \rangle$$

and division:

$$c(S \,/\, R)a \quad \Leftrightarrow \quad \langle \forall\, b :\; a\; R\; b\; : c\; S\; b \rangle$$

(References [Aarts et al., 1992; Backhouse, 2004] give a comprehensive account of how to structure the calculus of binary relations around Galois connections such as the one just above.) Since relational composition is associative, it should be clear that the calculation of relational equality

$$(S \,/\, R) \,/\, U \quad = \quad S \,/\, (U \circ R) \tag{1.9}$$

would be made along the very same steps as in inferring $(a \div b) \div c = a \div (c \times b)$ above — despite the fact that the calculated equality is far less immediate once its meaning is spelt out: it actually means, for all a, b, the equivalence

$$\langle \forall\, j :\ aUj\ : \langle \forall\, k :\ jRk\ : bSk \rangle \rangle \quad \Leftrightarrow \quad \langle \forall\, k :\ \langle \exists\, j :\ aUj\ : jRk \rangle\ : bSk \rangle$$

known as the $\forall, \exists$-"splitting rule" [Backhouse, 2004][3].

This capability of dealing with identical structures despite their complexity makes Galois connections a very powerful and scalable tool. Transposition of results such as seen above shows the *magic* of the concept, which turns reasoning about complex mathematical objects such as those found in theoretical computer science quite simple.

1.2 Objectives

The appreciation of such wide applicability and potential for program reasoning has led the author to embark on a project whose main aim is the design and implementation of a proof assistant — the *Galculator*— solely based on Galois connections, their algebra and associated tactics such as (1.6) above.

The main objectives are:

- Integrate Galois connections with fork algebras and effectively use them in equational formal proofs by adding indirect equality as inference rule.
- Define a domain-specific language (to be named *Galois*) for expressing formal proofs based on Galois connections and fork algebras. The formal syntax and semantics of the language should be a direct consequence of the theoretical algebraic concepts.
- Develop a front-end for *Galois* and implement a prototype of a proof assistant (the actual *Galculator*) based on the equational use of Galois connections and fork algebras. The prototype should be publicly available from the project's home page[4].
- Exploit the state-of-the-art technology of the *Haskell* [Peyton Jones, 2003] programming language such as generalized algebraic data types, existential data types, parsing combinators, strategic term rewriting combinators and polymorphic type representation,..., in the development of the proof assistant prototype.

[3]Section 2.2 and Appendix B provide more details about quantification and its rules.

[4]`http://www.di.uminho.pt/research/galculator`

As a foretaste of what is going to be presented in this dissertation, let us go back to the example of Section 1.1.2 and see how our prospective *Galculator* and *Galois* DSL can be used to conduct this proof using fork algebraic terms instead of point-wise operations. We can define a module with the operations and the axioms (and possibly additional theorems) of natural numbers arithmetics. However, here we just present the fragment useful for this proof.

We start by declaring (using *Galois* syntax) multiplication `Mul` and division `Div` as binary functions on natural numbers and `Leq` as a partial order on natural numbers.

```
Mul : Nat <- Nat >< Nat;
Div : Nat <- Nat >< Nat;
Leq : Ord Nat;
```

In this proof, the only axiom we need about the declared operators is the associativity of multiplication:

```
Axiom Mul_assoc := Fun [Mul<a> . Mul<b>] = Fun [Mul<Mul<a,b>>];
```

By now, this definition may seem awkward but it will be explained latter on. Notation `Mul<a>` means that the right argument of the multiplication is fixed with value `a` (this is called the right section). In the case `Mul<a,b>` both arguments of multiplication are fixed. The `Fun [...]` notation is used to embedded functions on relations as required by the type system.

The point-free proof further requires two additional axioms of fork algebras: the associativity of composition and the contravariance of converse and composition:

```
Axiom Comp_assoc := (r . s) . t = r . (s . t);
Axiom Contravariance := (r . s)* = s* . r*;
```

Finally, the Galois connection given by Equation (1.3) is declared by stating the two adjoint functions and the two associated partial orders:

```
Galois Whole_division := (Mul<b>) (Div<b>) Leq Leq;
```

where `Mul<b>` represents the $(\times b)$ adjoint, `Div<b>` represents the $(\div b)$ adjoint and `Leq` represents the $\leqslant$ order.

After declaring the theory, we can use the interactive proof assistant to build the proof. Another alternative is to declare the statement to prove as a theorem so that it can be used later on. For this purpose, a sequence of proof steps must be provided so that *Galculator* can verify its validity:

```
Theorem Div_mult := Fun [Div<c> . Div<b>] = Fun [Div<Mul<c,b>>]
  { indirect_left Leq > left                              >
    inv Comp_assoc    > once inv shunt Whole_division >
    Comp_assoc        > once inv shunt Whole_division >
    inv Comp_assoc    > once inv Contravariance       >
    once Mul_assoc    > once shunt Whole_division     >
    indirect_end      > qed }
```

Details about the meaning of scripts of this kind will be given in due time. For the moment, it suffices to tally the script's step with the calculation provided earlier on. Some individual steps (`indirect_left`, `indirect_end`, `left`) are related to the use of indirect equality; `qed` completes the proof script. The other steps are either the application of the axioms, or the properties of the Galois connection (1.3) using a proof strategy labeled by keyword `once`. Individual steps are combined with sequential composition and follow the same structure as the calculational version.

1.3 Structure of the dissertation

Part I. The first part concerns the basic theoretical concepts necessary to understanding the design of the *Galculator* prototype.

Chapter 2. We discuss related work and introduce the notation standards adopted in the rest of the document. We also discuss the proof format and some proof techniques. Finally, we give a short overview of the *Haskell* programming language.

Chapter 3. We explain the basic concepts of term rewriting systems upon which strategic rewriting systems are introduced.

Chapter 4. The concept of a fork algebra is presented as extension of that of a relation algebra, itself an extension of Boolean algebras. The use of these algebras leads to a point-free calculus of relations and the introduction of a point-free transform between the logical level and the relational level.

Chapter 5. Galois connections, their algebra and properties are described.

Chapter 6. Some examples and applications of Galois connections are described.

Part II. The second part of this dissertation describes the design of the *Galculator* prototype.

Chapter 7. The design of the *Galois* language is discussed together with the definition of its syntax and semantics.

Chapter 8. We theoretically justify the design of the *Galculator* resorting to the concepts introduced in the previous chapters.

Chapter 9. We present a description of the *Galculator* prototype using the *Haskell* programming language.

Chapter 10. We draw conclusions and evaluate our work. Furthermore, we discuss some open questions and possible future work.

Appendix A. This provides additional proofs of some results mentioned in the text. These have been separated from the main text wherever not essential to understanding the main results. Another reason to move these proof into an appendix is that, in the first chapters we provide proofs using proof techniques, namely point-free style, not yet introduced in the main discussion at that point.

Appendix B. The summary of rules associated with the manipulation of quantifiers is provided.

Index of concepts. For quick reference, we provide an index with references to the places where the main concepts are introduced.

Part I

Basics

Chapter 2

Preliminaries

This chapter is intended as a kind of "warming up" for the rest of the dissertation. Starting from a summary of related work, it discusses notation and proof conventions to be adopted in chapters to follow. Finally, a short introduction to *Haskell* is provided which should help "non-Haskellers" in understanding the code examples.

2.1 Related work

In this section, several systems or works related with *Galculator* are described. Their comparison with *Galculator* will be postponed to Section 10.3, after the complete design of the prototype has been discussed.

aRa. *aRa* [Sinz, 2000] is an automatic theorem prover for relation algebras. It has a front-end to translate relation algebraic formulas to Gordeev's Reduction Predicate Calculi logic. Thus, relation algebraic formulas are translated to logical sentences and proved using logic. *aRa* implements a set of simplification rules and reduction strategies for these calculi in order to automatically derive proofs.

RALL. *RALL* [von Oheimb and Gritzner, 1997] takes a similar approach to *aRa* although no translation is actually performed. Relation operators are formalized directly in *Isabelle/HOL* offering interactive and automatic proving facilities. Unlike *aRa*, *RALL* checks for type-correctness of all formulas.

RELVIEW. *RELVIEW* [Behnke et al., 1998] is a system for manipulation of relation algebras. All data are represented as binary relations using an efficient internal representation and optimized algorithms perform relational operations. Relations have to be explicitly defined and therefore, *RELVIEW* only works with concrete finite cases.

[Höfner and Struth, 2008]. Höfner and Struth [2008] propose the use of "off-the-shelf" automated theorem provers in order to prove theorems of relation algebras instead of special purpose approaches. According with the authors, more than one hundred theorems, many of them non-trivial, have been proved from an axiomatization of relation algebra using *Prover9*. *Prover9* is the successor of the *Otter Prover* and is described as *"a resolution/paramodulation automated theorem prover for first-order and equational logic"* [McCune, 2009]. The approach includes also the use of *Mace4* [McCune, 2009] to find counterexamples and avoid unnecessary search of proofs of invalid propositions.

2LT. *2LT* [Cunha et al., 2006b] is aimed at schema transformation of both data and migration functions in a type safe manner. Further developments deal with calculating data retrieving functions in the context of data schema evolution [Cunha et al., 2006a] and invariant preservation through data refinement [Alves et al., 2008]. *2LT* is not a prover: it calculates data and functional transformations using a correct-by-construction philosophy.

PF-ESC. This tool performs *point-free extended static checking* [Necco et al., 2007] using the relation calculus to simplify PF-transformed proof obligations. Galois connections are used implicitly in the underlying calculus as rewriting rules. Representations of relations in *PF-ESC* are strongly typed. Its design is inspired on *2LT*.

Proof processor system. Bohórquez and Rocha [2005] advocate the use of the calculational approach proposed by Dijkstra and Scholten in teaching discrete maths. Based on the *E* logical calculus, a tool was developed in *Haskell* to exploit equational proofs written in the *Z* notation [Spivey, 1989]. The system helps the user in detecting errors in proofs and suggesting valid deductive steps.

Galois connections in *Coq*. Pichardie [2005] presents a representation of Galois connections in *Coq* [Coquand and Huet, 1988] developed in the context of work on *abstract interpretation*. Adjoints are defined over complete lattices (a stricter requirement than in the general theory). Proofs of the general properties that Galois connections enjoy are defined in order to be executed in *Coq*.

2.2 Notation

In this section, we provide an overview of the notation used throughout this document. We also briefly discuss the importance of adopting good notation standards. Readers can skip this section in a first reading and return whenever missing some notational detail.

2.2.1 Notation — Is it really important?

Stewart [1992] tells the story about someone who said that a certain theorem about prime numbers could never be proved because there was no good notation for prime numbers. Carl Friedrich Gauss took the problem and proved it in five minutes, adding that what the other man needed were notions and not notations. This little story remind us of an essential point: ideas are the essence of mathematical reasoning and notations do not prove theorems. However, a good notation to represent and reason about problems is, without any doubt, important: it is hard to imagine someone performing complex calculus with Roman numerals although they are equivalent to the Arabic ones.

Leibniz and Euler established part of the modern standard mathematical notation as a simplification of previous notations, making it much more symbolic and compact. However, further developments in Mathematics posed new challenges which led to the introduction of new notations or adaptations of the classical one. The consequence is the existence of several variations (more or less distant) of the same notation. This poses additional difficulties to the readers, specially because notations are rarely explained.

Another problem occurs when notations evolve without careful thinking and their connection to the application subject is lost. This way, notation can become an obstacle to the understanding of the underlying ideas. This is often the case when notation become extremely "overloaded", denoting a possible lack of abstraction.

However, we must bear in mind is that a perfect notation does not exist. Depending upon its context of use, a notation may be adequate or not: it is always a trade-off between different factors.

Computer science, in general, and the construction of programs by calculus, in particular, impose some challenges to the design of a notation to handle them. Roland Backhouse is one of today's most active computer scientists concerned with designing a notation suitable for program construction. Backhouse [1989], which was inspired by Bird [1988] and Meertens [1986], provides an excellent discussion about the importance of notation in proofs. Backhouse [1988, 2004] provide extra details about the subject.

2.2.2 General principles

In this document, we follow the principles advocated by Backhouse [2004] namely:

- Notation should be uniform and consistent.
- Operator symbols should try to capture the meaning of the represented operations in an intuitive way. Dual concepts should be represented by symbols that resemble the duality, just like inverse concepts can be expressed using mirror symbols.
- Infix notation should be reserved to associative operators. Thus, we can drop parentheses without ambiguity and exploit the associativity property.
- Symmetric symbols should be chosen to denote symmetric operations and asymmetric symbols to denote asymmetric operations. However, sometimes the standard notation may force us to deviate from this guideline.
- Different concepts which share common properties can be *overloaded*, i.e., they can share the same symbol provided that it is clear from the context which one is being used and ambiguities are not possible.

Layout and precedence are also important to help the understanding of the reader:

- Precedences of the operators should be natural: the reader should not have to always check a precedence table to understand the meaning of an expression. For instance, one expects infix operators to have lower precedence than prefix ones.

Moreover, dual or inverse operators should have the same precedence because they denote equivalent concepts. Parentheses should be explicitly used to disambiguate expressions.

- Expression layout and spacing have no formal meaning. However, they are important to help the understanding of the expressions and should be consistent with the precedence of the operators.
- In an expression, it should always be clear the scope in which a certain definition is valid.

2.2.3 Notation used in this dissertation

In the following chapters, concepts will be introduced along with their notation specifying. In this section, we discuss the notation that is general and common throughout the whole document.

Quantification. Our notation deviates from the standard when dealing with quantification. We follow the uniform treatment presented in Backhouse [2004] by adopting pattern

$$\langle \bigoplus v \in \text{type} : \ \text{range} \ : \text{term} \rangle$$

where

- $\bigoplus$ is the *quantifier* symbol associated with some binary associative and commutative operator $\oplus$.
- v is the *bound variable* (or dummy variable) associated to the quantifier. The scope of the binding is delimited by the angle braces $\langle \ldots \rangle$. Lists of bound variables are also allowed.
- The *type* indication is optional and provides set information about the bound variable(s).
- The *range* is a predicate on the bound variable which determines the set of values for which it is true. When the predicate is not specified, the true predicate is implicitly assumed.

		Notation	
Operator	**Unit**	**Traditional**	**Uniform**
$\wedge$	true	$\forall i.0 \leqslant i \leqslant n.x_i \leqslant k$	$\langle \forall\, i : \ 0 \leqslant i \leqslant n \ : x_i \leqslant k \rangle$
$\vee$	false	$\exists i.0 \leqslant i \leqslant n.x_i \leqslant k$	$\langle \exists\, i : \ 0 \leqslant i \leqslant n \ : x_i \leqslant k \rangle$
$+$	0	$\sum_{i=0}^{n}(k + i^2)$	$\langle \Sigma\, i : \ 0 \leqslant i \leqslant n \ : k + i^2 \rangle$
$\times$	1	$\prod_{i=1}^{n}(i - 1)$	$\langle \Pi\, i : \ 1 \leqslant i \leqslant n \ : i - 1 \rangle$
$\cup$	$\emptyset$	$\bigcup_{i=1}^{n} S_i$	$\langle \bigcup i : \ 1 \leqslant i \leqslant n \ : S_i \rangle$

Table 2.1: Illustration of the correspondence between traditional notations and the uniform notation for quantification.

When the range is empty (the predicate always evaluates to false), the quantification is well-defined only if $\oplus$ has identity.

- The *term* is the expression being evaluated. It is a function from the range of the bound variable (although the bound variable may not occur in the term).

Backhouse [2004] summarizes the semantics of the quantification as follows: *"The value of the quantification is the result of applying the operator $\oplus$ to all the values generated by evaluation the term at all instances of the dummy in the range."*

This notation allows us to unify several different traditional notations in just one representation, as illustrated in Table 2.1. One of the advantages over traditional notations is the explicit indication of the scope of the bound variable, which eliminates ambiguities. Another positive point is the ease way quantifications are nested. But perhaps the most important aspect of this notation is the existence of an uniform set of calculation rules which allow for the manipulation of expressions [Backhouse, 2004]. Appendix B presents an overview of those rules.

Generalized quantification. The same notation can be uniformly extended to sequences and sets, by changing the properties imposed to the operator $\oplus$ [Backhouse, 1988]. In fact, our previous definition of quantification is equivalent to the case where it is defined over a bag of values. Table 2.2 provides the generalization of quantification to sequences, bags and sets as a consequence of the properties of the associated operator.

Generalization	**Notation**	**Properties of $\oplus$**
Sequence	$[\bigoplus v \in \text{type} : \text{range} : \text{term}]$	Associative
Bag	$\langle\bigoplus v \in \text{type} : \text{range} : \text{term}\rangle$	Associative and commutative
Set	$\{\bigoplus v \in \text{type} : \text{range} : \text{term}\}$	Associative, commutative and idempotent

Table 2.2: Generalization of the quantification.

Sets and sequences. Omission of the quantifier symbol in the unified notation presented leads to a definition by comprehension of a set or sequence. For instance, the following set comprehension in the traditional notation

$$\mathcal{A} = \{\, n^2 \mid n \in \mathbb{N} \quad \wedge \quad 5 \leqslant n \leqslant 13\}$$

is defined, in the unified notation, as

$$\mathcal{A} = \{\, n \in \mathbb{N} : \; 5 \leqslant n \leqslant 13 \; : n^2\}$$

The definition of sequences by comprehension requires additional conditions since the order in which the bound variable is evaluated matters. Thus, the *range* must implicitly define a linear order[1], i.e., the values that the bound variable can take form an ascending (or descending) chain. The order of the output list values follows from the evaluation of the bound variable in the *term* according with the order implicit in the *range*.

Functions. Traditional *lambda* notation for functions [Church, 1936] suffers from some of the same drawbacks as traditional quantifier notation. Thus, we use a similar notation for functions:

$$\langle\; v \in \text{type} : \; \text{range} \; : \text{term}\rangle$$

which is equivalent to the traditional λv : type.term. The *range* information is not used in the *lambda* notation, but its interpretation is similar to the one provided for quantifications: it specifies the *domain* of the (thus partial) function.

In the case of function operations like, e.g., fixed point operators, we will write

[1]Different kinds of orders are discussed in Section 4.1.

$\langle \mu\ v\ :\ \text{range}\ :\ \text{term} \rangle$ instead of $\mu \langle\ v\ :\ \text{range}\ :\ \text{term} \rangle$. This is consistent with our quantifier notation because v is the variable associated with the recursion and thus it is bounded to the fixed point operator.

2.3 Proofs

When talking about proofs, we should usually distinguish between informal and formal proofs as anticipated in the previous chapter. Due to their complexity, formal proofs are usually built using software programs like theorem provers and proof assistants. Their verification is also automatically performed by software proof checkers.

However, most proofs are informal. Mathematics has a long tradition of informal proofs and to most people this is their only concept of proof. Lamport [1995] argues that in the last 300 years the proof structure has not evolved and that proofs in textbooks are usually hard to read. This difficulty often causes errors to go unnoticed even after careful reviewing. The same author willingly declares: *"Anecdotal evidence suggests that as many as a third of all papers published in mathematical journals contain mistakes — not just minor errors, but incorrect theorems and proofs."*

In this section, we address this problem and describe our proof presentation format which can be used both in formal and informal proofs. We also discuss the use of some proof techniques.

2.3.1 Proof presentation format

Hierarchical proof format. Lamport [1995] proposes a hierarchical proof format inspired by natural deduction in order to prove the correctness of algorithms. He argues that structure is crucial to these understanding, and although it does not eliminate all the errors, they are greatly reduced.

Equational proofs. The proof presentation format we chose is substantially different from the one proposed by Lamport [1995], however sharing some of its principles. Such a proof format is extensively used in [Backhouse, 1988; Aarts et al., 1992; Backhouse, 2004] which is based on the so-called "calculational style" proposed by Feijen [Dijkstra and Feijen, 1988] and further developed by Dijkstra and Scholten [1990]. One can use this proof format either in informal proofs or in formal proofs. Gries and

Schneider [1995] define a deductive system for the propositional logic using the calculational style and show its equivalence to the Hilbert proof style. Lifschitz [2001] establishes a deductive system for predicate logic.

Leibniz principle. Equational proofs are based on a very simple, yet powerful, principle due to Leibniz. Basically, if we have two equal terms and we replace the occurrences of one for the other in some expression, the truth of the expression does not change.

Proof steps. Each proof step establishes an equality between two expressions and it is justified by a hint:

$$\begin{array}{ll} & \text{expr}_1 \\ = & \qquad \{\text{ Hint }\} \\ & \text{expr}_2 \end{array}$$

Hints. Hints can be formal or semi-formal. Formal hints refer to an equality like $A = B$ which justifies the equality in the split according to the Leibniz principle, i.e., $\text{expr}_1[A] = \text{expr}_2[B]$. We may give the explicit equality or refer to the number of an already presented one. In either case, we try to provide textual in-place information to help the reader in following the reasoning.

Sometimes, we want to omit some details because they are uninteresting, or because they are commonplace, such as the use of associativity. In other occasions, we just want to use standard properties of operators without defining these explicitly. In this case, we use semi-formal hints instead. However, we should be careful about semi-formal hints because we may be introducing errors.

Proof format. A proof with two steps will look like

Proof

$$\begin{array}{ll} & E[A][B] \\ = & \quad \{\, A = C \,\} \\ & E[C][B] \\ = & \quad \{\, B = D \,\} \\ & E[C][D] \end{array}$$

□

The complete proof is based on the transitivity of equality. Thus, the first and the last expressions are equal, i.e., $E[A][B] = E[C][D]$.

Equivalence instead of equality. When dealing with logical expressions we should use logical equivalence instead of equality. Backhouse [2004] eliminates the use of logical equivalence in proofs and uses equality everywhere. The author considers logical expressions as Boolean values, and thus equality is well-defined. However, we will not deviate thus far from the traditional notation and we will still use the logical equivalence operator.

Other relations. Sometimes, we need to establish inequalities rather than equalities. For instance, we want to use the *at most* ($\leqslant$) order of integers in a proof:

Proof

$$\begin{array}{ll} & a \\ \leqslant & \quad \{\, \text{Hint} \,\} \\ & b \\ \leqslant & \quad \{\, \text{Hint} \,\} \\ & c \end{array}$$

□

Then again, we should read the proof conjunctively, i.e., $a \leqslant b$ and $b \leqslant c$. Since *at most* is transitive, we can again conclude that $a \leqslant c$. We can even mix steps involving the *at most* order with steps using equality because the transitivity is still preserved.

However, it is not possible to mix steps involving the *at most* order with steps using the *at least* order.

The same general principles apply to the use of logical implication and its connection with the use of logical equivalence.

Adoption of the format. This proof format corresponds to the natural notion of replacing equals by equals and to one of the simpler kind of mathematical proof: the direct proof. Despite its simplicity, this format has not achieved general acceptance, although some positive experiences exist. Bohórquez and Rocha [2005] report how this proof style has been successfully used to teach discrete mathematics to higher education students.

Lamport [1995] concludes that mathematicians are conservative in the way they write proofs and that they are not prepared to consider better ways of doing it. He also thinks that computer scientists can be more open to new proof formats but, since incorrect results are not considered *"embarrassing"*, there is not a real will to change.

2.3.2 Proof techniques

In order to derive proofs, mathematicians have invented a wide range of techniques. It turns out that while some of them are very useful, others should be avoided. Angluin [1983] provides a humorous account of non-recommended proof techniques that, when carefully analyzed, can be insightful.

In this section, we will discuss some proof techniques which are related with our work, either because we want to use them or to avoid them.

Mutual inclusion (vulg. "ping-pong"). In proofs by *mutual inclusion*, also known as *"ping-pong" proofs*, one tries to establish an equivalence between two expressions by splitting it in two implications. That is, $a \Leftrightarrow b$ is proved if and only if $a \Rightarrow b$ and $a {\Leftarrow} b$ are so. Equivalently, an equality can be split in two relations of an anti-symmetric order. For instance, to establish $a = b$ it is sufficient to prove $a \sqsubseteq b$ and $b \sqsubseteq a$, where $\sqsubseteq$ is an anti-symmetric order.

This kind of proof is frequently abused when establishing equivalences, often overshadowing the equation nature of the reasoning. In this document, we avoid proofs by mutual inclusion whenever they are not strictly necessary.

Proofs by mathematical induction. Proofs by *mathematical induction*, also known just as proofs by induction, are used to establish results in denumerable infinite well-founded structures. Well-founded structures arise from the existence of binary well-founded relations (we will discuss this subject with more detail in Chapter 3). Intuitively, a relation is well-founded if it does not allow infinite descending chains of elements.

Proving a property by induction involves two steps:

1. The base case — We must show that the property holds in the basic cases, i.e., in the minimum values of the chains.

2. The inductive step — We assume that the property holds for an arbitrary value. Then, taking this assumption as (induction) hypothesis we must prove that the property also holds for its successors with respect to the well-founded relation.

The ability to cope with denumerable infinite structures makes induction a very powerful and popular mathematical device, in particular those over the natural number ($\mathbb{N}$). However, proofs by induction may become very complex and hard to handle.

Strengthening. A *strengthening step* occurs when the statement to prove is "stronger" than the original one, thus implying it. In our proof convention, we introduce strengthening steps by using the implication sign in the form $\Leftarrow$, be read as *"if"*:

Proof

$$\begin{array}{ll} & a \\ \Leftrightarrow & \qquad \{ \text{ Hint } \} \\ & b \\ \Leftarrow & \qquad \{ \text{ Hint } \} \\ & c \end{array}$$

$\square$

Strengthening steps are sometimes necessary but should be avoided because implication has not the same "nice" properties of equivalence. In fact, strengthening steps lose information and can lead us to too strong conditions which are eventually impossible to prove. The extreme case happens where $c = \mathsf{false}$, the proof being still valid but meaningless [Backhouse, 2004].

Weakening. A *weakening step* occurs when the statement to prove is "weaker" than the original one, thus being implied by it. In our proof convention, we introduce weakening steps by using the implication sign in form $\Rightarrow$ be read as *"only if"*,

Proof

$$\begin{array}{ll} & a \\ \Leftrightarrow & \qquad \{ \text{ Hint } \} \\ & b \\ \Rightarrow & \qquad \{ \text{ Hint } \} \\ & c \end{array}$$

$\square$

Proving the validity of the weaker statement does not give us any useful information about the validity of the original statement. However, weakening steps are useful to prove the falsity of a statement. When $c = \mathsf{false}$, it implies that $a = \mathsf{false}$ also. Hoogerwoord [2001] uses weakening to derive an elegant equational proof that $\sqrt{p}$ is not rational, for a prime number p (Backhouse [2003] presents a contrapositive version of this proof where equality is replaced by disequality and weakening steps are replaced by a strengthening ones).

2.4 Brief overview of *Haskell*

This section provides a brief introduction to the functional programming *Haskell* [Peyton Jones, 2003] language, ranging from simple notation conventions to the advanced features of its type system that will play a major role later in this dissertation. This includes a brief explanation of the potential of using functional programming in the implementation of domain specific languages.

For readers interested in learning more about *Haskell*, references [Thompson, 1996; Bird, 1998] are classical starting points; the official web page[2] is also a valuable source of information providing pointers to many books, articles and tutorials.

2.4.1 *Haskell*

Haskell is a purely lazy functional language [Peyton Jones, 2003]. It is *strongly-typed* meaning that programs cannot fail due to run-time type errors. Type checking is performed statically even if type declarations are not provided, thanks to type-inference.

Data types. In *Haskell*, every object has an associated type. Primitive data types include integers (Int for machine word-sized integers and $\mathit{Integer}$ for arbitrary precision integers), floating point numbers (Float and Double with different precisions), characters (Char) and Boolean values (Bool). Data type constructors are used to build-up more complex types from the primitive ones. For instance, the Cartesian product of arbitrary types of objects $a :: A$ (read *"object a has associated type A"*) and $b :: B$ is given by the rule

$$\frac{a :: A \qquad b :: B}{(a, b) :: (A, B)}\,(\times)$$

The type of $((1, \texttt{'a'}), \mathit{True})$ can be inferred during the compilation phase as follows:

$$\frac{\dfrac{1 :: \mathit{Integer} \qquad \texttt{'a'} :: \mathit{Char}}{(1, \texttt{'a'}) :: (\mathit{Integer}, \mathit{Char})}\,(\times) \qquad \mathit{True} :: \mathit{Bool}}{((1, \texttt{'a'}), \mathit{True}) :: ((\mathit{Integer}, \mathit{Char}), \mathit{Bool})}\,(\times)$$

The type of an object may not be unique as we shall see next when polymorphism is discussed.

Functions. In functional programming, functions resemble mathematical functions where argument values are transformed in results without any side-effects. This means that when applied to the same value, the function always returns the same result.

The typing rule for functions is the following, for arbitrary types A and B:

$$\frac{f :: A \to B \qquad a :: A}{f\ a :: B}\,(\to)$$

[2] `http://www.haskell.org`

This means that a function cannot be applied to an argument with the wrong type.

For instance, the squaring function on integers $f(x) = x \times x$ can be declared as

$$\begin{aligned}&\mathit{square} :: \mathit{Integer} \rightarrow \mathit{Integer}\\&\mathit{square}\ n = n * n\end{aligned}$$

Since this should be regarded like a mathematical function, instances of the left-hand side of the equation can be replaced by corresponding instances of the right-hand side, i.e., whenever we have $\mathit{square}\ 3$ we can replace it by $3 * 3$.

Functions are first-order citizens meaning that they are treated like other ordinary data.

Recursive functions. Functional programming treats iteration as *recursion*. A recursive function uses itself in its own definition. A well-known recursive function is factorial which can be defined as the following:

$$\begin{aligned}&\mathit{factorial} :: \mathit{Integer} \rightarrow \mathit{Integer}\\&\mathit{factorial}\ 0 = 1\\&\mathit{factorial}\ n = n * \mathit{factorial}\ (n - 1)\end{aligned}$$

Since we can replace left-hand sides of equations by the right-hand sides, the factorial of 3 can be computed as:

$$\begin{aligned}&\ \ \mathit{factorial}\ 3\\&= 3 * \mathit{factorial}\ (3 - 1) = 3 * \mathit{factorial}\ 2\\&= 3 * (2 * \mathit{factorial}\ (2 - 1)) = 3 * (2 * \mathit{factorial}\ 1)\\&= 3 * (2 * (1 * \mathit{factorial}\ (1 - 1))) = 3 * (2 * (1 * \mathit{factorial}\ 0))\\&= 3 * (2 * (1 * 1))\\&= 3 * (2 * 1)\\&= 3 * 2\\&= 6\end{aligned}$$

This example also illustrates the use of *pattern-matching*, i.e., the possibility of matching values or patterns of values in the left-hand side of the equations. In this case, $\mathit{factorial}\ \ 0$ is used when the argument is 0 while $\mathit{factorial}\ \ n$ matches all the remaining cases.

Parametric polymorphism. The language supports *parametric* as well as *ad-hoc* polymorphism. In parametric polymorphism type variables can range over the universe of types. For instance, the identity function is defined in *Haskell* as

$$\begin{aligned}&id :: a \rightarrow a\\&id\ x = x\end{aligned}$$

meaning that it "ignores" the type of its argument. It is parametric because a will be instantiated with the actual type of the argument when this is provided. For instance, expression $id\ \ True$ will instantiate a to $Bool$ and return a value of type $Bool$ according with the inference

$$\frac{id :: a \rightarrow a \qquad True :: Bool}{id\ \ True :: Bool}\ (\rightarrow)$$

where the typing rule for functions is relaxed to accept type variables instead of just (arbitrary) fixed types.

Ad-hoc polymorphism. Ad-hoc polymorphism, also known as function *overloading*, allows for functions to be applied to arguments of different types which this time behave differently according of the type of their arguments. In *Haskell*, ad-hoc polymorphism is implemented using type *classes*. For instance, the equality class

$$\begin{aligned}&\mathbf{class}\ Eq\ a\ \mathbf{where}\\&\quad (\equiv), (\not\equiv) :: a \rightarrow a \rightarrow Bool\end{aligned}$$

defines two class functions: $\equiv$ for equality and $\not\equiv$ for inequality. Every instance of the Eq class must provide an implementation of the two functions (in fact, only one of them is needed because the other is, by default, its negation). For instance, we can declare Boolean values as instances of the equality class as

$$\begin{aligned}&\mathbf{instance}\ Eq\ Bool\ \mathbf{where}\\&\quad True \equiv True\ = True\\&\quad False \equiv False = True\\&\quad \_ \quad\ \equiv \_ \quad\ = False\end{aligned}$$

where the $\_$ notation matches all the remaining cases.

Type classes allow for generic code. For instance, we can define a generic function which removes duplicates from a list, providing that the type of its elements is an instance of the equality class:

$$nub :: Eq\ a \Rightarrow [\,a\,] \rightarrow [\,a\,]$$

where $Eq\ \ a\ \Rightarrow\ \ldots$ indicates that nub uses the equality function and thus requires instances of Eq.

Higher-order functions. Since functions are treated as first-order objects in *Haskell*, they can be used whenever a value can, including as arguments of other functions. Functions which take a function as argument and/or return a function has result are known as *higher-order functions*. These allows for building common patterns such as, for instance, the map higher-order function

$$map :: (a \rightarrow b) \rightarrow [\,a\,] \rightarrow [\,b\,]$$

which takes a function from values of type a to values of type b, and applies it to every elements of a list of type $[\,a\,]$ obtaining a list of type $[\,b\,]$.

Given a list of integers, suppose we want to increase all its elements by one unit. This can be done by the following function which resorts to the above map function:

$$increase :: [\,Integer\,] \rightarrow [\,Integer\,]$$
$$increase = map\ (\lambda x \rightarrow x + 1)$$

Expression $\lambda x\ \rightarrow\ x + 1$ uses the so-called *lambda notation* which allows for introducing an unnamed function where the left-hand side of the expression declares the argument variables and the right-hand side specifies the function. This notation is inspired on the lambda calculus [Church, 1936] where this function would be written as $\lambda x.x + 1$ (or, using our uniform notation introduced in Section 2.2, as $\langle\ x :: x + 1\rangle$). For instance, given the list $[1, 2, 3]$ we have that $increase\ [1, 2, 3] = [2, 3, 4]$.

Recursion patterns can be defined as higher-order functions, allowing for reusable code and modularity. Moreover, patterns help keeping the concepts clear and organized.

Currying and uncurrying. *Haskell* functions take just a single argument. In fact, a function with several arguments, such as $f\ ::\ a\ \rightarrow\ b\ \rightarrow\ c$, is a function from values of type a into functions of type $b \rightarrow c$. More precisely, its type is $f :: a \rightarrow (b \rightarrow c)$. This subtle difference allows for a mechanism of *partial application*: given a value $a' :: a$, $f\ \ a'$ is function "waiting" for values of type b to produce results of type c, i.e.,

$f\ a' :: b \to c$. Thus, given a value $b' :: b$, expression $(f\ \ a')\ b'$ denotes a value of type c (the parenthesis around $(f\ \ a')$ can be omitted).

The situation is different when the argument of a function is a pair. For instance, let us suppose a new function f' identical to function f above, the only difference being that its arguments are paired, i.e., $f' :: (a, b) \to c$. Function f' has only one argument, requiring that both values are provided *simultaneously* in order to compute the result of type c.

Currying is a technique which allows for partial application of functions whose arguments are pairs. The inverse of currying is known as *uncurrying*. *Haskell* offers two high-order functions to this purpose:

$$\begin{aligned}
&\mathit{curry} &&:: ((a, b) \ \to c) \to (a \to b \to c) \\
&\mathit{uncurry} &&:: (a \to b \to c) \to ((a, b) \ \to c)
\end{aligned}$$

Using functions f and f' introduced above, we have that $\mathit{curry}\ \ f' = f$ and $\mathit{uncurry}\ f = f'$.

Sections. The situation when a function with more than one argument is partially applied, i.e., one of its arguments is specified, is known as *sectioning*. A useful function when sectioning is *flip*, which allows for reversing the order of the arguments of a curried function:

$$\mathit{flip} :: (a \to b \to c) \to (b \to a \to c)$$

Thus, given a curried function $f :: a \to b \to c$ and values $a' :: a$ and $b' :: b$, the *left section* of f is defined as $f\ \ a' :: b \to c$ and the *right section* of f is defined as $\mathit{flip}\ f\ \ b' :: a \to c$.

Uncurried functions can be sectioned resorting to the curry function. Thus, for an uncurried function $f' :: (a, b) \to c$, the *left section* is defined as $\mathit{curry}\ f'\ \ a' :: b \to c$ and the *right section* is defined as $\mathit{flip}\ (\mathit{curry}\ f')\ b' :: a \to c$.

For a more concrete example, let us consider the uncurried version of the subtraction function on integers:

$$\begin{aligned}
&\mathit{subtract} :: (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \\
&\mathit{subtract}\ (a, b) = a - b
\end{aligned}$$

For the value 5, the left section of this function is $\mathit{curry}\ \ \mathit{subtract}\ 5$ which is equivalent to function $\lambda x \to 5 - x$. For the same value, the right section of $\mathit{subtract}$ is $\mathit{flip}\ (\mathit{curry}\ \mathit{subtract})\ 5$ which is equivalent to function $\lambda x \to x - 5$.

Algebraic data types. *Algebraic data types* (ADTs) are the mechanism used in *Haskell* in order to declare new data types. An ADT declaration specifies how inhabitants of the type can be built, i.e., its constructors, like in an algebraic definition. ADT notation in *Haskell* is particularly effective in the sense that it subsumes parameterized, union, enumeration and recursive types in just one device. For instance,

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

declares the recursive parametric type of finite lists, where Nil is the empty list and Cons the list append constructor function. A list of integers $\mathit{List}\ \mathit{Int}$ takes the form $\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 4\ \mathit{Nil})))$ which is written in *Haskell*'s built-in notation as $1{:}(2{:}(3{:}(4{:}[\,])))$, or simply as $[1, 2, 3, 4]$. The type variable a is determined by the type of the elements of the list, implying that *all* of them share the same type. Thus, the list $[1, \mathit{True}]$ is not valid because of the conflicting types $[\mathit{Int}]$ and $[\mathit{Bool}]$.

Haskell allows for pattern-matching on constructors of ADTs. For instance, we can define the function that returns the number of elements of a list resorting to pattern-matching on ADTs:

$$\begin{array}{l}
\mathit{length} :: \mathit{List}\ a \rightarrow \mathit{Integer} \\
\mathit{length}\ \mathit{Nil} \qquad\quad\ = 0 \\
\mathit{lenght}\ (\mathit{Cons}\ \_\ \mathit{xs}) = 1 + \mathit{length}\ \mathit{xs}
\end{array}$$

or using the *Haskell* built-in syntax

$$\begin{array}{l}
\mathit{length} :: [a] \rightarrow \mathit{Integer} \\
\mathit{length}\ [\,] \qquad\ = 0 \\
\mathit{length}\ (\_ : \mathit{xs}) = 1 + \mathit{length}\ \mathit{xs}
\end{array}$$

Generalized algebraic data types. *Generalized algebraic data types* (GADTs) provide an extension to this device. They extend the capabilities of ADTs by introducing a new syntax for declarations where constructor types are explicitly spelt out. For instance, $\mathit{List}\ a$ will be written in the GADT format as follows,

$$\begin{array}{l}
\mathbf{data}\ \mathit{List}\ a\ \mathbf{where} \\
\quad \mathit{Nil} \quad\ :: \mathit{List}\ a \\
\quad \mathit{Cons} :: a \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a
\end{array}$$

using GADTs notation. What is this useful for?

Unlike ADTs, GADTs allow for restricting the result type parameter of each constructor. A “classical” example of the use of GADTs is the construction of a type representation mechanism [Baars and Swierstra, 2002; Cheney and Hinze, 2002]:

```
data Type a where
  Int  :: Type Int
  Bool :: Type Bool
  List :: Type a → Type [a]
  · × · :: Type a → Type b → Type (a, b)
  ...
```

If ADTs were used, the return type of all the constructors above would be bound to $\mathit{Type}\ a$; with GADTs each such type is restricted to a more *precise* type. The value $\mathit{List}\ \mathit{Int}$ has type $\mathit{Type}\ \mathit{Int}$ while the value $\mathit{List}\ \mathit{Bool}$ has type $\mathit{Type}\ \mathit{Bool}$. This means that a is no longer a parameter; it has become an *index type* which reflects the type of the term built. Thus, for the value $\mathit{List}\ \mathit{Int}$ the index type a takes the type $[\mathit{Int}]$ while in for $\mathit{List}\ \mathit{Bool}$ it takes the type $[\mathit{Bool}]$.

Like in ADTs, pattern-matching can be used with constructors of GADTs.

Singleton types. The above example introduces another feature of GADTs: the possibility of use of *singleton* (or *representation*) types. As Sheard et al. [2005] put it, *“every singleton type completely characterizes the structure of its single inhabitant, and the structure of a value in a singleton type completely characterizes its type”*. More precisely, a singleton type establishes a bijection between its inhabitants and their respective type. We should notice that not every GADT definition is a singleton type but only those for which this property holds. In the above examples, Type is a singleton type while List is not.

For instance, constructors $\mathit{List}\ \mathit{Int}$ and $\mathit{List}\ \mathit{Bool}$ are the *only* possible ways of building values of type $\mathit{Type}\ [\mathit{Int}]$ and $\mathit{Type}\ [\mathit{Bool}]$, respectively. Conversely, given a value type $\mathit{Type}\ [\mathit{Int}]$ (resp. $\mathit{Type}\ [\mathit{Bool}]$) we know that this value must *necessarily* be $\mathit{List}\ \mathit{Int}$ (resp. $\mathit{List}\ \mathit{Bool}$).

As we will see in the sequel, singleton types allows for a *reflection* mechanism on the *Haskell* type system, and for introducing dynamic typing in a static context.

Existential types. Another important feature of the *Haskell* type system are *existential types*. These allow for the introduction of arbitrary types into type definitions, hiding them from the outer context. A traditional example of this device is the definition of a list with elements of different types. We can define an existentially quantified type as follows:

$$\mathbf{data}\ T = \forall a.\ \mathit{MkT}\ a$$

It should be noticed that the quantified variable a only exists in the context of the quantification, it is not a parameter of the type T. The *Haskell* syntax is somewhat misleading since an universal quantifier is used. However, this definition is isomorphic to

$$\mathbf{data}\ T = \mathit{MkT}\ (\exists a.\ a)$$

written in *Haskell* pseudo-code [Wikibooks, 2009].

The intuition is of quantification over types is similar to quantification in logic. The universal quantification corresponds to the intersection (meet, and) of types while existential quantification corresponds to union (joint, or) of types. Thus, universal quantification on types intersects them, i.e., it only allows objects which are common to all types. In *Haskell* types are lifted meaning that there is a bottom element, denoted as $\perp$, which belongs to every type. This is the only object which belongs to the intersection of all types. The existential quantification, by the other hand, join all the types, meaning that it allows objects of any type.

Using the type T, a heterogeneous list can be built, e.g.,

$$[\mathit{MkT}\ 1, \mathit{MkT}\ [\mathit{True}, \mathit{False}], \mathit{MkT}\ \texttt{'a'}\ ] :: [\,T\,]$$

However, values cannot be taken outside the constructor because they are existentially quantified. Since they can have any type that would break static type safety. Moreover, no operation can be performed because the type is too general. Nevertheless, using type classes the existentially quantified variable can be constrained. For instance, if we restrict the quantified types to instances of the Show class (this provides a method $\mathit{show} :: \mathit{Show}\ a \Rightarrow a \rightarrow \mathit{String}$ for building string representations) we can define

$$\mathbf{data}\ T' = \forall a.\ \mathit{Show}\ a \Rightarrow \mathit{MkT}'\ a$$

This is equivalent to the pseudo-code [Wikibooks, 2009]

$$\mathbf{data}\ T' = \mathit{MkT}'\ \exists a.\ \mathit{Show}\ a \Rightarrow a$$

This means that we are constraining the union of types to only those which are instances of the *Show* class. Thus, they now have some kind of common behavior which can be exploited. A heterogeneous list of this type can be traversed in order to obtain string representations of its elements, i.e., it is possible to define a function showT' of type $[T'] \to \mathit{String}$ by applying the method *show* to each element of a list of type $[T']$.

GADTs and existential types. GADTs also subsume the use of existential quantified types. Variables which appear in the type constructors but do not appear in their return type are existentially quantified. Thus, type T above can be defined as follows, giving an explicit signature to its constructor:

$$\begin{array}{l}\mathbf{data}\ T\ \mathbf{where}\\ \quad \mathit{MkT} :: a \to T\end{array}$$

while type T' can be defined as

$$\begin{array}{l}\mathbf{data}\ T'\ \mathbf{where}\\ \quad \mathit{MkT}' :: \mathit{Show}\ a \Rightarrow a \to T'\end{array}$$

Although this is a different way of defining an existential data type, the constructors (MkT and MkT') can be used exactly as before.

2.4.2 Monads

Pure functional languages have referential transparency and are side-effect free. How can programming ingredients such as input-output, state updates, etc., that usually do not accommodate very well in the functional paradigm, be treated in such a side-effect-free way? The concept of *monad* arising from category theory and programming language semantics [Wadler, 1990] has been implemented in *Haskell* as a mechanism to deal with such computations.

Definition. Monads are available in *Haskell* via standard type class

$$\begin{array}{l}\mathbf{class}\ \mathit{Monad}\ m\ \mathbf{where}\\ \quad \mathit{return} :: a \qquad\ \to m\ a\\ \quad (\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

where $\gg\!=$ is referred to as *bind*. Every instance of this class should obey the monadic laws [Wadler, 1990]:

$$
\begin{aligned}
&\mathit{return}\ a \gg\!= f = f\ a\\
&m \gg\!= \mathit{return} = m\\
&(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)
\end{aligned}
$$

Although all instances of *Monad* must instantiate both functions, very different effects can be achieved by changing the definitions. Thus, the semantics of programs depend of the underlying monads.

Syntactic sugar. *Haskell* provides a **do** notation similar to an imperative programming style as syntactical sugar for successive binds. Thus, the following expression

$$m1 \gg\!= (\lambda x1 \rightarrow m2 \gg\!= (\lambda x2 \rightarrow \ldots \gg\!= (\lambda xn \rightarrow \mathit{return}\ (f\ x1\ x2\ \ldots\ xn))\ \ldots))$$

can be replaced by the equivalent

```
do x1 ← m1
   x2 ← m2
   ...
   xn ← mn
   return (f x1 x2 ... xn)
```

Example (*Maybe*). The *Maybe* data type is one of the simpler instances of the concept of monad. *Maybe* is used to model possibly failing computation and is defined in *Haskell* as

```
data Maybe a = Nothing | Just a
```

where *Nothing* represents the failure of a computation while *Just* indicates success together with the return value.

The declaration of *Maybe* as a monad is

```
instance Monad Maybe where
  return x      = Just x
  Just x  >>= f = f x
  Nothing >>= _ = Nothing
```

The $return$ method always succeeds returning the input value and is used to inject values inside the monad. The bind operator keeps the flow of values through successful computations. However, when a failure is found this is propagated through the remaining computations.

For instance, suppose we want to define a function which returns the head element of a list. This function is clearly not defined when the input list is empty. The *Haskell* implementation of the $head$ function throws a run-time error when it is applied to a empty list aborting the execution of the program. An alternative would be to test the input value and return a failure value when a empty list is found:

```
$head' :: [a] \to Maybe\ a$
$head'\ [\,]\qquad = Nothing$
$head'\ (x : \_) = x$
```

Another well-known partial function example comes from mathematics where dividing by 0 is not defined. Using $Maybe$ the function which return the multiplicative inverse of a number can be defined as

```
$inv :: Float \to Maybe\ Float$
$inv\ 0 = Nothing$
$inv\ x = 1\ /\ x$
```

Thus, we can use monadic code to combine the two functions so we get the multiplicative inverse of the head of a list:

```
$invHead\ lst = \mathbf{do}$
  $h \leftarrow head'\ lst$
  $i \leftarrow inv\ h$
  $return\ i$
```

As expected, the function returns a failure value when applied to an empty list, i.e., $invHead\ [\,] = Nothing$, or when the head of the list is 0, $invHead\ [0, 2] = Nothing$. Otherwise, it signals the success together with the result: $invHead\ [2, 0] = Just\ 0.5$. The solution without the monadic code would be less clearer because explicit case analysis is necessary.

Example (Lists). Lists are another basic and important instance of a monad:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ [\,]\ \textbf{where}\\ \quad \mathit{return}\ x = [\,x\,]\\ \quad \mathit{xs} \gg\!= f\ \ = \mathit{concat}\ (\mathit{map}\ f\ \mathit{xs})\end{array}$$

The return method returns a list just containing the input value. The definition of the bind operator is more complex, requiring the use of the map function introduced before,

$$\mathit{map} :: (a \to b) \to [\,a\,] \to [\,b\,]$$

which takes a function from values of type a to values of type b, and applies it to every elements of a list of type $[\,a\,]$ obtaining a list of type $[\,b\,]$. Since, by the definition of a monad, f must have type $a \to [\,b\,]$, then $\mathit{map}\ f\ \mathit{xs}$ must have type $[\,[\,b\,]\,]$. This result is "flattened" using the function

$$\mathit{concat} :: [\,[\,a\,]\,] \to [\,a\,]$$

which takes a list of lists of the same type and returns a single list. For instance, suppose we have the following function

$$\begin{array}{l}\mathit{idSquare} :: \mathit{Int} \to [\,\mathit{Int}\,]\\ \mathit{idSquare}\ a = [\,a, a * a\,]\end{array}$$

which returns a list with the input value and the corresponding square. We can apply $\mathit{idSquare}$ to all elements of the list $[1,2,3]$ and then concat the result[3]

$$\begin{array}{l}\quad \mathit{concat}\ (\mathit{map}\ \mathit{idSquare}\ [1,2,3])\\ = \mathit{concat}\ [[1,1],[2,4],[3,9]]\\ = [1,1,2,4,3,9]\end{array}$$

or simply

$$[1,2,3] \gg\!= \mathit{idSquare} = [1,1,2,4,3,9]$$

Monad transformers. Another advantage of using monads is that computations can be composed using *monad transformers* [Jones, 1995]. For instance, adding error support to a program that already uses the input-output monad amounts to combining the two monads with a transformer and changing the parts of the program where errors are generated or caught; everything else remains unchanged.

[3]Haskell library includes a function which combines concat and map named $\mathit{concatMap}$.

Failure monads. Of special importance in this work is the use of $\mathit{MonadPlus}$ and $\mathit{MonadOr}$. Although providing different behaviors $\mathit{MonadPlus}$ and $\mathit{MonadOr}$ are many times confused. Currently, only $\mathit{MonadPlus}$ is part of the *Haskell* standard libraries but there is a discussion for reformulating the structure in order to introduce $\mathit{MonadOr}$ and to group the common failure behavior in $\mathit{MonadZero}$. (More details are available from the *Haskell* wiki[4].)

MonadZero. The $\mathit{MonadZero}$ monad is basically used to model failure which is represented by the method mzero:

$$\begin{array}{l}\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZero}\ m\ \mathbf{where}\\ \quad \mathit{mzero} :: m\ a\end{array}$$

The definition of $\mathit{MonadZero}$ requires that mzero must be a *left zero* of the binding operator:

$$\mathit{mzero} \gg\!= r = \mathit{mzero}$$

Both lists and the Maybe data type are instances of $\mathit{MonadZero}$, being the failure values the empty list and $\mathit{Nothing}$, respectively,

$$\begin{array}{l}\mathbf{instance}\ \mathit{MonadZero}\ \mathit{Maybe}\ \mathbf{where}\\ \quad \mathit{mzero} = \mathit{Nothing}\\ \mathbf{instance}\ \mathit{MonadZero}\ [\,]\ \mathbf{where}\\ \quad \mathit{mzero} = [\,]\end{array}$$

MonadPlus. $\mathit{MonadPlus}$ extends the behavior of $\mathit{MonadZero}$ with an additive operation named mplus:

$$\begin{array}{l}\mathbf{class}\ \mathit{MonadZero}\ m \Rightarrow \mathit{MonadPlus}\ m\ \mathbf{where}\\ \quad \mathit{mplus} :: m\ a \to m\ a \to m\ a\end{array}$$

The definition of $\mathit{MonadPlus}$ requires that mplus must form a monoid structure with mzero, i.e., mplus must be associative and take mzero as unit:

[4]http://www.haskell.org/haskellwiki/MonadPlus_reform_proposal

$$\begin{aligned}
&a \quad \text{`}\mathit{mplus}\text{`}\ \mathit{mzero} &&= a \\
&\mathit{mzero}\ \text{`}\mathit{mplus}\text{`}\ b &&= b \\
&(a\ \text{`}\mathit{mplus}\text{`}\ b)\ \text{`}\mathit{mplus}\text{`}\ c &&= a\ \text{`}\mathit{mplus}\text{`}\ (b\ \text{`}\mathit{mplus}\text{`}\ c)
\end{aligned}$$

Moreover, mplus must obey the *left-distribution* law:

$$(s\ \text{`}\mathit{mplus}\text{`}\ r) \gg\!= t = (s \gg\!= t)\ \text{`}\mathit{mplus}\text{`}\ (r \gg\!= t)$$

This specifies a backtracking behavior, where all the possible combinations are tried. Haskell's infix notation $a\ \text{`}\mathit{op}\text{`}\ b$ is equivalent to $\mathit{op}\ a\ b$, for a binary function op.

Lists are instances of $\mathit{MonadPlus}$:

```
instance MonadPlus [] where
  mplus = (++)
```

where mplus is just the concatenation operation on lists $(+\!\!+)$ (e.g., $[1, 2, 3] +\!\!+ [4, 5] = [1, 2, 3, 4, 5]$). However, Maybe is not an instance of $\mathit{MonadPlus}$ because it fails to accomplish the left-distribution law.

The *Parsec* library of parsing combinators [Leijen and Meijer, 2001] to which we shall resort in Section 9.5 uses $\mathit{MonadPlus}$.

MonadOr. $\mathit{MonadOr}$ extends the behavior of $\mathit{MonadZero}$ with a left-choice operator named $\mathit{morelse}$:

```
class MonadZero m ⇒ MonadOr m where
  morelse :: m a → m a → m a
```

Like in $\mathit{MonadPlus}$, the method of $\mathit{MonadOr}$ must also form a monoid structure with mzero:

$$\begin{aligned}
&a \quad \text{`}\mathit{morelse}\text{`}\ \mathit{mzero} &&= a \\
&\mathit{mzero}\ \text{`}\mathit{morelse}\text{`}\ b &&= b \\
&(a\ \text{`}\mathit{morelse}\text{`}\ b)\ \text{`}\mathit{morelse}\text{`}\ c &&= a\ \text{`}\mathit{morelse}\text{`}\ (b\ \text{`}\mathit{morelse}\text{`}\ c)
\end{aligned}$$

However, $\mathit{morelse}$ must obey the *left-catch* law instead:

$$\mathit{return}\ a\ \text{`}\mathit{morelse}\text{`}\ r = \mathit{return}\ a$$

This specifies a left biased behavior: the second argument is only tried if the first one fails.

Maybe is an instance of *MonadOr*:

```
instance MonadOr Maybe where
  Nothing `morelse` ys = ys
  xs      `morelse` _  = xs
```

Method *morelse* only returns its right argument if the left one is *Nothing*. For instance, $Just\ 1$ ‘*morelse*‘ $Just\ 2 = Just\ 1$ and $Just\ 1$ ‘*morelse*‘ $Nothing = Just\ 1$; however, $Nothing$ ‘*morelse*‘ $Just\ 2 = Just\ 2$ and $Nothing$ ‘*morelse*‘ $Nothing = Nothing$.

Lists are also instances of *MonadOr*:

```
instance MonadOr [] where
  [] `morelse` b = b
  a `morelse` _ = a
```

When dealing with the empty list, the behavior is similar to *mplus*, e.g., $[\,]$ ‘*mplus*‘ $[1, 2] = [1, 2]$ and $[\,]$‘*morelse*‘$[1, 2] = [1, 2]$. However, when the left list is not empty, the right list is ignored, e.g., $[1, 2]$ ‘*mplus*‘ $[3, 4] = [1, 2, 3, 4]$ and $[1, 2]$ ‘*morelse*‘ $[3, 4] = [1, 2]$. This exemplifies how changing the underlying monad can change the semantics of the program.

2.4.3 GADTs and domain-specific languages

Grammars and parsers are central to computer science. Besides checking for input correctness, a parser for a given grammar returns a representation where all the syntactical details are omitted, known as abstract syntax tree (AST) (or just syntax tree) [Aho et al., 1986].

Functional languages resort to ADTs in order to define ASTs: from a grammar specification it is straightforward to extract an ADT which represents the type of the corresponding AST. Conversely, every ADT may be seen as a specification of an abstract language. Polymorphic ADTs define families of abstract languages.

Languages can be catalogued as *general-purpose* or *domain-specific*. The former are suited to solve problems in general while the latter are tailored to particular, well-defined problem domains. They tend to be relatively small (although this is not always the case) and specialized.

The extra cost of developing a domain-specific language (DSL), due to the need for infrastructures such as compilers, parsers, etc. are trimmed down by *embedding* the new language into a *host language*. Such embedded DSLs (EDSLs) are usually provided in the form of libraries sharing the host language's infrastructure.

Functional languages are particularly apt to EDSL development thanks to their natural support for ADTs and the availability of generalized algebraic data types (GADTs), which offer new possibilities for EDSL implementation. While ADT data constructors only keep term information, GADT's constructors add types to terms. Moreover, as described in Section 2.4.1, the type index of a GADT reflects the type of the term built. Using this index with a type representation such as the one above it is possible to have a reflection mechanism and to know terms' types at run-time. This allows for type-dependent behavior and dynamic typing. In summary, with GADTs ill-typed terms are simply not possible to build.

2.5 Summary

This chapter provided an overview of some non-essential although useful subjects to the rest of this dissertation. Some related systems which inspired the development of this work were analysed. Later on, after introducing the complete design of the *Galculator*, we will compare our approach with this related work.

The description and justification of the used notation and proof techniques are usually overlooked by most authors. However, in our case, we think it is important to give it an insight in order to help the reader understand the rest of the text. Moreover, both play a relevant role in the design of the *Galculator*, as we shall see in the following chapters.

Since some parts of this dissertation require the understanding of some *Haskell* code, a brief introduction was provided to those readers with no prior knowledge of the language. This introduction is not intended to be a course on *Haskell*. Some of the described concepts like existential data types, GADTs and monads are quite complex, and usually require some familiarity with functional programming before they can be fully understood. Thus, even if the reader is not able grasp all the details of these concepts, he or she should gain some intuition about the way they work.

Chapter 3

Term rewriting systems and strategic programming

Term rewriting [Baader and Nipkow, 1998] is a mature field of computer science spreading through areas such as the implementation of functional and logical programming languages, automated deduction, theorem proving, algebraic computation or decision procedures [Dershowitz, 1993; Baader and Nipkow, 1998]. Simple syntax and semantics, together with nice mathematical properties, makes term rewriting systems (TRS) attractive for describing and automating computations.

In this chapter, we will cover the basic concepts of TRSs needed in the sequel. We begin by introducting abstract reduction systems, the more general concept which TRS systems are particular instances of. Other examples include string rewriting, tree rewriting, ground rewriting, higher-order rewriting or infinite rewriting.

Last but not least, a particular kind of term rewriting, that of *strategic term rewriting* [Visser and Benaissa, 1998], will be introduced.

3.1 Abstract reduction systems

Definition. An *abstract reduction system* (ARS) is a pair $(\mathcal{A}, \rightarrow)$ where $\mathcal{A}$ is a set and $\rightarrow$ is a binary relation on $\mathcal{A}$, usually called *reduction relation*. Instead of writing $(a, b) \in \rightarrow$ for elements a and b of $\mathcal{A}$ which belong to relation $\rightarrow$, it is usual to write $a \rightarrow b$.

This is a very broad definition since it allows any binary endo-relation to be considered as a reduction. However, we are usually interested in those relations which reduce or decrease (in some sense) something in each step. Thus, more important than the definition itself are the properties that reduction relations should enjoy. Two properties essential to the study of reduction systems are termination and confluence.

Termination. An ARS is *terminating* if and only if there is no infinite chain of reductions, i.e., one always reaches, in a finite number of reduction steps, an element $a \in \mathcal{A}$ which cannot be further reduced (i.e., such that there is no $b \in \mathcal{A}$ such that $a \to b$). Such irreducible elements are referred to as *normal forms*.

Termination is, in general, an undecidable property since it is equivalent to the halting problem [Turing, 1936]. However, it is possible to prove termination for numerous ARSs. Some of them are naturally terminating because the reduction relation is *well-founded*. A relation R on set $\mathcal{A}$ is said to be well-founded if and only if every non-empty subset of $\mathcal{A}$ has a minimal element with respect to R [Baader and Nipkow, 1998]. For instance, the natural numbers are a well-founded ARS with reduction order $>$.

A common technique for proving termination of a given ARS $(A, \to)$ consists in finding a monotone mapping into another ARS $(B, >)$ which is known to terminate. A discussion of several methods for proving termination can be found in [Baader and Nipkow, 1998].

Confluence. Sometimes, it is possible to reduce an element in several different ways, i.e., for some $a \in \mathcal{A}$ there may exist $b, c \in \mathcal{A}$ such that $a \to b$ and $a \to c$. This is equivalent to stating that the reduction relation is not functional. An important question is: will further reductions eventually reach a common element? If this is always true then the order in which reductions are applied does not matter; in the end the result will always be the same.

In order to define confluence formally it is necessary to introduce some notions first. Let $\to$ be a reduction relation. Then the following relations are defined:

$\leftarrow$	Converse of $\to$;
$\overset{*}{\to}$	Reflexive transitive closure of $\to$;
$\leftrightarrow$	Symmetric closure of $\to$;
$\overset{*}{\leftrightarrow}$	Reflexive transitive symmetric closure of $\to$.

We say that $a, b \in \mathcal{A}$ are *joinable*, denoted by $a \downarrow b$, if and only if there is some $x \in \mathcal{A}$ such that $a \xrightarrow{*} x \xleftarrow{*} b$.

An ARS is *confluent* if and only if for all $a, b, c \in \mathcal{A}$, $a \xleftarrow{*} c \xrightarrow{*} b \Rightarrow a \downarrow b$.

Interpretation. Abstract reduction systems (and their concrete instances) can be interpreted in two different ways [Baader and Nipkow, 1998]:

Computations. Reductions can be seen as computation steps tranforming some input to the result. Should the termination property hold in the system, a normal form is always achieved. Thus, normalization is equivalent to program evaluation in this case. In this way, the study of ARSs is connected to operational semantics and the implementation of programming languages.

Deductions. ARSs can be used as decision procedures in order to decide equivalence between elements. If we consider the reflexive transitive symmetric closure of the reduction relation, it can be seen as an identity between elements since there is a path between them in both directions. The use of such kind of decision procedures for identities is very important in systems for symbolic and algebraic computation and theorem proving.

3.2 Equational logic

Because there is a close connection between term rewriting and equational logic, it is important to understand some basic concepts of the latter. In this section, a more formal presentation could be made resorting to universal and term algebras, like in [Baader and Nipkow, 1998]. However, we will avoid the extra complexity and try to keep the presentation simple, yet rigorous like in [Plaisted, 1993].

Identities. Terms in *equational logic* are inductively defined as:

- Variables;
- Constants;
- Function symbols (also called operations).

Function symbols are used to built up new from existing terms. The arity of a function symbol is the number of terms it takes as arguments. Constants can be seen as function symbols with arity 0. The set of variables is countably infinite and disjoint from function symbols and constant names.

Expressions in equation logic are called *equalities* or *identities*; they are built up from two terms together with the equality symbol[1]:

$$s \approx t$$

where s and t are terms, s being referred to as the left-hand side (lhs) and t as the right-hand side (rhs) of the identity.

Substitutions. A *substitution* is a mapping σ from terms to terms which satisfies $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$, for every n-ary function symbol f and $\sigma(x) \neq x$ for only finitely many x's [Baader and Nipkow, 1998]. Thus, substitutions are defined by replacing terms for variables. The *composition of two substitutions* is again a substitution and is denoted by juxtaposition: given two substitutions σ and δ their composition is $\sigma\delta$.

If, given terms t and t' there is a substitution σ such that $\sigma(t) = t'$ we say that t' is an *instance* of t.

Equational systems. An *equational system* $\mathcal{E}$ is a set of identities. We write

$$\mathcal{E} \vdash s \approx t$$

meaning that $s \approx t$ is *valid* in $\mathcal{E}$ (or that it is a *syntactic consequence* of $\mathcal{E}$). $s \approx t$ is valid in $\mathcal{E}$ if and only if $s \approx t$ is derivable from the set of equations $\mathcal{E}$ using the following inference rules, where f is a function symbol with arity $n \geqslant 0$, and σ is a substitution:

$$\frac{}{\mathcal{E} \vdash t \approx t} \text{ Reflexivity}$$

$$\frac{\mathcal{E} \vdash s \approx t}{\mathcal{E} \vdash t \approx s} \text{ Symmetry}$$

$$\frac{\mathcal{E} \vdash s \approx t \qquad \mathcal{E} \vdash t \approx u}{\mathcal{E} \vdash s \approx u} \text{ Transitivity}$$

[1]We will use symbol $\approx$ for identities in order to avoid confusions with equality at the meta-level.

$$\frac{\mathcal{E} \vdash s \approx t}{\mathcal{E} \vdash \sigma(s) \approx \sigma(t)} \text{ Substitution}$$

$$\frac{\mathcal{E} \vdash s_1 \approx t_1 \quad \dots \quad \mathcal{E} \vdash s_n \approx t_n}{\mathcal{E} \vdash f(s_1, \dots, s_n) \approx f(t_1, \dots, t_n)} \text{ Congruence}$$

Note that congruence defines a closure under function symbols. An assumption rule is also needed, asserting that any element of $\mathcal{E}$ is valid in $\mathcal{E}$:

$$\frac{s \approx t \in \mathcal{E}}{\mathcal{E} \vdash s \approx t} \text{ Assumption}$$

Context of a term. Let l and u be terms and l is a sub-term of u. We denote the context of l on u by $u[l]$ [Dershowitz, 1993]. $u[.]$ represents the term u *except* the sub-term l. This can be seen as a term with a "hole" which can be fulfilled with another term. For instance, if we replace l by l' in the context we get $u[l']$.

Reduction relation. From an equational system $\mathcal{E}$, a reduction relation can be inferred. The *reduction relation* $\rightarrow_{\mathcal{E}}$ on terms is defined as $s \rightarrow_{\mathcal{E}} t$ if and only if exists an identity $(l \approx r) \in \mathcal{E}$ such that $s[l'] = \sigma(l)$ and $t = s[\sigma(r)]$, for some substitution σ.

Symbolisms $\overset{*}{\rightarrow}_{\mathcal{E}}$ and $\overset{*}{\leftrightarrow}_{\mathcal{E}}$ denote the reflexive transitive closure and the reflexive transitive symmetric closure, respectively, of $\rightarrow_{\mathcal{E}}$. Relation $\overset{*}{\leftrightarrow}_{\mathcal{E}}$ is important in equational reasoning because of its equivalence with derivation, i.e., $s \overset{*}{\leftrightarrow}_{\mathcal{E}} t$ holds if and only if $\mathcal{E} \vdash s \approx t$.

Semantics of equational systems. In order to provide the semantics of equational systems, an algebraic approach is convenient. However, as already mentioned, this would mean the introduction of too much mathematical background, only to justify a few important results. Thus, we prefer the style of first-order logic to explain the semantics of equational systems, such as done by Plaisted [1993].

First, we need the notion of structure which will avoid the use of a term algebra. A structure Σ is a pair consisting of a non-empty domain set $\mathcal{A}$ and a mapping from each function symbol f in the equational system $\mathcal{E}$ to an endo-function f^{Σ} in $\mathcal{A}$.

A interpretation mapping ϕ^{Σ} from terms of an equational system $\mathcal{E}$ to the structure

Σ is defined as:

$$\phi^{\Sigma}(t) \quad \overset{def}{=} \quad \begin{cases} a \in \mathcal{A} & \text{if } t \text{ is a variable} \\ c \in \mathcal{A} & \text{if } t \text{ is a constant} \\ f^{\Sigma}(\phi^{\Sigma}(t_1), \ldots, \phi^{\Sigma}(t_n)) & \text{if } t \text{ is a function symbol } f(t_1, \ldots, t_n) \\ & \text{of } \mathcal{E}\text{, with arity } n > 0 \end{cases}$$

We say that Σ *satisfies* the identity $s \approx t$ (or that $s \approx t$ *holds* in Σ) if and only if $\phi^{\Sigma}(s) = \phi^{\Sigma}(t)$, i.e., they have the same interpretation in Σ. We also say that Σ is a *model* of an equational system $\mathcal{E}$ (or Σ *satisfies* $\mathcal{E}$) if and only if Σ satisfies every identity of $\mathcal{E}$. We write $\Sigma \models \mathcal{E}$ to denote this.

An identity $s \approx t$ is a *semantic consequence* (or logical consequence) of $\mathcal{E}$ ($\mathcal{E} \models s \approx t$) if and only if all models of $\mathcal{E}$ satisfy it. An equational system $\mathcal{E}_2$ is a logical consequence of $\mathcal{E}_1$ ($\mathcal{E}_1 \models \mathcal{E}_2$) if and only if all models of $\mathcal{E}_1$ are also models of $\mathcal{E}_2$.

Finally, an equational system $\mathcal{E}$ induces an *equational theory* defined by the relation:

$$\approx_{\mathcal{E}} \quad \overset{def}{=} \quad \{\, s, t \in \text{Terms} : \ \mathcal{E} \models s \approx t \ : (s,t)\}$$

Birkhoff's theorem. An important result due to [Birkhoff, 1935] relates syntax and semantics of equational systems. If $\mathcal{E}$ is an equational system then $\mathcal{E} \models s \approx t$ if and only if $\mathcal{E} \vdash s \approx t$, i.e., $\overset{*}{\leftrightarrow}_{\mathcal{E}}$ and $\approx_{\mathcal{E}}$ coincide.

Word problem. Given an equational theory $\mathcal{E}$, a fundamental problem is that of deciding if two arbitrary terms are equal, i.e., if $s \approx_{\mathcal{E}} t$ holds. By Birkhoff's theorem [Birkhoff, 1935], an equivalent problem is to decide if it is possible to transform a term s into a term t using the reduction relation induced by the identities in $\mathcal{E}$. This is known as the *word problem* which, in general, is undecidable.

Unification. Another relevant problem concerning any equational theory $\mathcal{E}$, is that of finding a substitution σ for terms s and t such that, $\sigma(s) \approx_{\mathcal{E}} \sigma(t)$. This is called the *satisfiability problem* which is again undecidable, in general. The process of solving this problem is known as *unification*.

Syntactic unification. A special case of the satisfiability problem arises when the equational theory $\mathcal{E}$ is empty. In this setting, it boils down to finding a substitution σ

End	$\eta(\emptyset)$	$= \emptyset$
Delete	$\eta(\{t \approx^? t\} \uplus S)$	$= \eta(S)$
Decompose	$\eta(\{f(\vec{t_n}) \approx^? f(\vec{u}_n)\} \uplus S)$	$= \eta(\{t_1 \approx^? u_1, \ldots, t_n \approx^? u_n\} \cup S)$
Clash	$\eta(\{f(\vec{t_n}) \approx^? g(\vec{u}_n)\} \uplus S)$ if $f \neq g$	$= \bot$
Orient	$\eta(\{t \approx^? x\} \uplus S)$ if t is not a variable	$= \eta(\{x \approx^? t\} \cup S)$
Eliminate	$\eta(\{x \approx^? t\} \uplus S)$ if x is a variable and x does not occurs in t	$= \eta(\{x \to t\}(S))\{x \to t\}$
Occurs-Check	$\eta(\{x \approx^? t\} \uplus S)$ if $x \neq t$ and x occurs as a variable in t	$= \bot$

Table 3.1: Unification by transformation algorithm. $\uplus$ denotes disjoint union; $\bot$ denotes failure; and $\{x \to t\}(S)$ denotes the application of substitution $x \to t$ to S.

such that $\sigma(s) = \sigma(t)$, i.e., proving the syntactic equality between terms $\sigma(s)$ and $\sigma(t)$. The process of solving this simplified problem is known as *syntactic unification*.

In general, the *unification problem* deals with finite sets of equations $S \overset{def}{=} \{s_1 \approx^? t_1, \ldots, s_n \approx^? t_n\}$. In this text we distinguish identities ($\approx$) from equations ($\approx^?$): the former state equalities between elements while the latter represent hypothetic equalities still to be established (possibly false). The solution of a set of equations is a substitution σ (referred to as *unifier*) such that $\sigma(s_i) = \sigma(t_i)$ for $i = 1, \ldots, n$. A set of equations can have none, one or many unifiers. However, an important theorem establishes that if the problem has a solution then it has an idempotent most general unifier, a special case of a well-behaved unifier [Baader and Nipkow, 1998]. Being a *most general unifier* means that it is minimum, i.e., the smallest substitution that solves problem S. Formally, a substitution σ is more general than σ' if there is another substitution δ such that $\sigma' = \delta\sigma$. A substitution σ is *idempotent* if and only if $\sigma = \sigma\sigma$.

Unification by transformation. A possible approach to syntactic unification of a set of equations S is that of finding equations of the form $x \approx^? t$ and replacing all occurrences of x in S by t, i.e., $\{x \to t\}(S)$, until reaching the solution. This is known as *unification by transformation* and it works much like the Gaussian elimination algorithm for systems of linear equations.

Table 3.1 presents the complete unification by transformation algorithm adapted from [Baader and Nipkow, 1998] re-written in a more functional flavor. The η function takes a set of equations S and returns a substitution $\eta(S)$. Disjoint union $\uplus$ is used in the left-hand side of definitions, meaning that each equation is unique and each step removes it from the set of remaining equations. The *Clash* and *Occurs-Check* rules recognize impossible unifications returning a failure value, denoted by $\perp$.

As proved by Baader and Nipkow [1998], η terminates for all inputs, and if S is solvable then $\eta(S)$ does not fail and returns an idempotent most general unifier of S. It should be noted that the order in which the rules are applied does not matter.

Matching Another problem related with unification is *matching*. In matching we are only interested in finding a substitution σ that transforms a term t in an instance of a term s, that is, finding σ such that $\sigma(s) = t$. Matching reduces to unification wherever variables in t are seen as constants.

3.3 Term rewriting systems

Term rewriting systems (TRSs) are instances of abstract reduction systems in which the reduction relation is defined by rewriting rules over terms. TRSs are closely related with equational systems and equational logic. The difference is that in TRS, rewriting rules are oriented, i.e., only left-hand sides can be replaced by right-hand sides. Ensuring that a TRS has the same computational meaning as the corresponding equational system constitutes most of the study of TRSs [Plaisted, 1993].

Definition. A *term rewriting system* (TRS) is a set of rewriting rules. A *rewriting rule* is an identity denoted by $l \rightarrow r$, where l and r are terms, l is not a variable and every variable in r occurs in l. Term l is referred to as the left-hand side and r as the right-hand side of the rewriting rule. Let σ be a unifier for l and another term t. Then, $\sigma(t)$ is a *redex* (reducible expression) that can be reduced (or contracted) to the corresponding instance of the rhs of the rewriting rule $\sigma(r)$.

Rewriting strategies. TRSs are, in general, non-deterministic. Which rule is to be applied to which term is not usually known. However, sometimes it is useful to choose one redexes from the other, leading to the definition of *rewriting strategies*. Table 3.2 summarizes some of the most common rewriting strategies [Plaisted, 1993].

Reduction strategy	**Meaning**
Leftmost innermost	The leftmost among all the innermost redexes is rewritten.
Parallel innermost	All innermost redexes are simultaneously rewritten.
Leftmost outermost	The leftmost among all the outermost redexes is rewritten.
Parallel outermost	All outermost redexes are simultaneously rewritten.
Full substitution	All redexes are simultaneously rewritten.

Table 3.2: Most common rewriting strategies.

An *innermost redex* is a subterm which has no subterms which are themselves redexes; an *outermost redex* is a subterm that is not contained in any other redexes.

Some of the these strategies are important because of their connection to other fields: innermost rewriting is related with denotational semantics [Plaisted, 1993]; outermost rewriting corresponds to lazy evaluation [Plaisted, 1993].

3.4 Strategic term rewriting systems

"Traditional" term rewriting systems. "Traditional" term rewriting environments clearly distinguish the set of rewriting rules of a particular TRS from the strategy used to apply these rules to terms [Alves et al., 2005]. However, unlike the set of rules that can be changed, strategies are usually hard-wired into the environment. Leftmost innermost or outermost are examples of strategies commonly used by rewriting environments.

However, a fixed strategy provides very little control over the rewriting process and the order in which rules are tried. Even in confluent systems, the order in which rules are applied can have a significant impact on performance. In other cases, termination is easier, or even only possible if a certain strategy is adopted. Furthermore, many interesting systems are not confluent and/or terminating, and yet are usable in practice thanks to the application of rules in some restricted sense.

If more control over the rewriting process is needed the solution is to introduce function symbols in rules. In this way, the rewriting strategy becomes explicit in the function symbols. However, the meaning of the rules is obfuscated since the strategy and equations become entangled into each other. Moreover, the use of the rules in a different context is hard, if not at all possible.

Strategic term rewriting systems. To the best of our knowledge, Paulson [1983] was first in proposing the use of modular rewriting strategies in the context of the implementation of tactics for the *LCF* theorem prover. *Strategic term rewriting* uses simple basic strategies and combinators in order to build arbitrarily complex strategies in a declarative style. Strategies can be reused and combined in different ways to obtain different TRSs. In this way, strategies become programmable, just as the equations are.

Strategies resemble other combinatorial approaches, for instance, parsing combinators [Wadler, 1985; Hutton, 1992; Leijen and Meijer, 2001], or pretty-printing combinators [Hughes, 1995]. Combinator techniques, besides modularity and reusability, also offer a declarative style, usually with nice algebraic properties. In a sense, they define a domain specific language whose programs or scripts can be regarded as executable specifications.

Two of the first notable TRSs to use strategic rewriting were *Stratego* [Visser and Benaissa, 1998] and the Rewriting Calculus [Cirstea et al., 2001, 2003] associated with the *ELAN* specification language.

Typed strategic term rewriting systems. In the literature, the discussion about the merits and disadvantages of compile-time typing and run-time typing is rather long; it has already been called a *"cold war"* among programming languages [Meijer and Drayton, 2005]. The same arguments apply to TRSs in general.

Typed term rewriting is used in order to ensure that rules are only applied to terms of the appropriate type. When working with untyped strategies, unexpected fails can occur if there is any fault in the design of the strategy. Beside type safeness, typed strategies also allow for type dependent behavior in which rules are only applied to terms of a specific type.

The combination of strategic term rewriting with strong typing was introduced in *Haskell* by the *Strafunski* bundle [Lämmel and Visser, 2003] and in *Java* by the *JJTraveler* framework [Visser, 2001b; Kuipers and Visser, 2001]. Further generalizations are provided in the *Haskell* context by Lämmel and Peyton Jones [2003] and Lämmel and Visser [2002]. An account of a formal semantics of typed strategic programming is given by Lämmel [2003].

Strategy combinators. Strategic term rewriting provides a small set of strategy combinators which are useful in building up compound strategies of arbitrary complexity. In this setting, a strategy has a slightly different meaning than in traditional TRSs. Let

Strategy combinator	**Symbol**
Identity strategy	nop
Always failing strategy	$\perp$
Sequential composition	$\triangleright$
Choice (non-deterministic)	$\oplus$
Choice (left-bias)	$\oslash$
Map on all children	all
Map on one child	one
Map on first child	first
Fixed point recursion	μ

Table 3.3: Primitive strategy combinators according to Luttik and Visser [1997] (adapted).

$s@t$ denote the application of a strategy s to a term t; the result of this application is called a *reduct*. A reduct can be another term, meaning that the strategy application was successful, or it can be an error value $\perp$ denoting failure.

The list of primitive strategy combinators as described in Luttik and Visser [1997] is presented in Table 3.3. The *identity strategy*, written nop, always succeeds returning its argument, $\mathrm{nop}@t \rightarrow t$. The *always failing strategy*, written $\perp$, fails for all input terms, $\perp@t \rightarrow \perp$. The *sequential composition strategy*, written $s \triangleright r$, succeeds if s succeeds returning a term t' and the application of r to t' succeeds. The *non-deterministic choice strategy*, written $s \oplus r$, succeeds if s or r succeed. The *left-biased choice strategy*, written $s \oslash r$, succeeds if s or r succeed, but r will be only tried if s fails.

All these strategies are only applicable at root terms. In order to build overall traversals, traversal over children (subterms) of a term is needed as introduced by all, one and first. The *mapping on all children strategy*, written $\mathrm{all}(s)$, succeeds if the application of s to all the children of the input term succeed. This strategy evaluates children terms from the left-most to the right-most. The *mapping on one children strategy*, written $\mathrm{one}(s)$, succeeds if the application of s succeeds for, at least, one child of the input term. The choice of such a child is non-deterministic. The *mapping on first child strategy*, written $\mathrm{first}(s)$, succeeds if the application of s succeeds for, at least, one child of the input term. The chosen child is the first one for which the strategy s succeeds when traversing children terms from the left-most to the right-most.

Finally, a *fixed point recursion* operator, written $\mu(s)$, is defined. The strategy s can be rewritten in the form $\langle\, v :: s\rangle$ where v is a free-variable of s. The fixed point operator replaces the strategy s for the free occurrences of the variable v in s. Thus, we have $\langle\mu\, v :: s\rangle@t = s[v := \langle\mu\, v :: s\rangle]@t$. The fixed point operator should be interpreted lazily; otherwise it would lead to a non-terminating rewrite system [Luttik and Visser, 1997].

Rewriting rules. Besides strategy combinators, basic strategies in the form of rewriting rules are needed. Rewriting rules can be applied to terms through the use of more complex strategies built with other combinators.

Basically, a rewriting rule of the form $t_l \rightarrow t_r$ can be applied to a term t if t is an instance of the left-hand side of the rule, i.e., one must find a substitution σ such that $\sigma(t_l) = t$. The returned term is an instance of the right-hand side of the rule, $\sigma(t_r)$. If the matching of the left-hand with the term is not possible the application fails.

3.4.1 Reduction semantics

After presenting the informal meaning of the strategies, we provide the complete reduction semantics adapted from [Luttik and Visser, 1997; Lämmel, 2003; Alves et al., 2005]. Each rule has a name together with a plus or a minus: name$^+$ identifies a rule which always succeeds, while name$^-$ identifies a rule which always fails. For a function symbol f, $f()$ represents a term without any sub-terms (i.e., a constant), while $f(d_1, \ldots, d_n)$ represents an n-ary constructor in which d_1, …, d_n are its sub-terms.

Identity strategy

$$\frac{}{\text{nop}\ @t \rightarrow t}\ \text{id}^+$$

Failure strategy

$$\frac{}{\bot @t \rightarrow \bot}\ \text{fail}^-$$

Sequential composition strategy

$$\frac{s_1@t \rightarrow t' \qquad s_2@t' \rightarrow t''}{(s_1 \rhd s_2)@t \rightarrow t''}\ \text{seq}^+$$

$$\frac{s_1@t \to \bot}{(s_1 \triangleright s_2)@t \to \bot}\ \text{seq1}^- \qquad \frac{s_1@t \to t' \qquad s_2@t' \to \bot}{s_1 \triangleright s_2@t \to \bot}\ \text{seq2}^-$$

Non-deterministic choice strategy

$$\frac{s_1@t \to t'}{s_1 \oplus s_2@t \to t'}\ \text{nd1}^+ \qquad \frac{s_2@t \to t'}{s_1 \oplus s_2@t \to t'}\ \text{nd2}^+$$

$$\frac{s_1@t \to \bot \qquad s_2@t \to \bot}{s_1 \oplus s_2@t \to \bot}\ \text{nd}^-$$

Left choice strategy

$$\frac{s_1@t \to t'}{s_1 \oslash s_2@t \to t'}\ \text{left1}^+ \qquad \frac{s_1@t \to \bot \qquad s_2@t \to t'}{s_1 \oslash s_2@t \to t'}\ \text{left2}^+$$

$$\frac{s_1@t \to \bot \qquad s_2@t \to \bot}{s_1 \oslash s_2@t \to \bot}\ \text{left}^-$$

Map on all children strategy

$$\frac{}{\text{all}(s)@f() \to f()}\ \text{all1}^+$$

$$\frac{s@d_1 \to d'_1 \qquad \dots \qquad s@d_n \to d'_n}{\text{all}(s)@f(d_1, \dots, d_n) \to f(d'_1, \dots, d'_n)}\ \text{all2}^+$$

$$\frac{\langle \exists\, i:\ 1 \leqslant i \leqslant n\ : s@d_i \to \bot \rangle}{\text{all}(s)@f(d_1, \dots, d_n) \to \bot}\ \text{all}^-$$

Map on one child strategy

$$\frac{\langle \exists\, i:\ 1 \leqslant i \leqslant n\ : s@d_i \to d'_i \rangle}{\text{one}(s)@f(d_1, \dots, d_i, \dots, d_n) \to f(d_1, \dots, d'_i, \dots, d_n)}\ \text{one}^+$$

$$\frac{}{\text{one}(s)@f() \to \bot}\ \text{one1}^-$$

$$\frac{s@d_1 \to \bot \qquad \dots \qquad s@d_n \to \bot}{\text{one}(s)@f(d_1, \dots, d_n) \to \bot}\ \text{one2}^-$$

Map on first child strategy.

$$\frac{s@d_1 \to \bot \qquad \dots \qquad s@d_{i-1} \to \bot \qquad s@d_i \to d'_i}{\mathrm{first}(s)@f(d_1,\dots,d_{i-1},d_i,\dots,d_n) \to f(d_1,\dots,d_{i-1},d'_i,\dots,d_n)}\ \text{first}^{+}$$

$$\frac{}{\mathrm{first}(s)@f() \to \bot}\ \text{first1}^{-}$$

$$\frac{s@d_1 \to \bot \qquad \dots \qquad s@d_n \to \bot}{\mathrm{first}(s)@f(d_1,\dots,d_n) \to \bot}\ \text{first2}^{-}$$

Rewriting rules strategy

$$\frac{\langle \exists\ \sigma :: \sigma(t_l) = t \wedge \sigma(t_r) = t' \rangle}{t_l \to t_r@t \to t'}\ \text{rule}^{+}$$

$$\frac{\langle \nexists\ \sigma :: \sigma(t_l) = t \rangle}{t_l \to t_r@t \to \bot}\ \text{rule}^{-}$$

3.4.2 Building compound strategies

Using the primitive strategy combinators presented in Table 3.3, more complex compound strategies can be defined. The following examples correspond to some useful strategies which are directly adopted in the design of *Galculator*.

Failure recovery. The following strategy always succeeds, even when its argument strategy fails. In this case, the identity strategy is used and the term is left unchanged:

$$\mathrm{try}\, s \quad \overset{def}{=} \quad s \oslash \mathrm{nop}$$

Repetition. The repetition strategies are useful when an argument strategy must be repeatedly applied until it fails:

$$\begin{aligned} \mathrm{many}\, s &\overset{def}{=} \langle \mu\ f :: (s \rhd f) \oslash \mathrm{nop} \rangle \\ \mathrm{many\_1}\, s &\overset{def}{=} s \rhd \mathrm{many}\, s \end{aligned}$$

Strategy many always succeeds since failures are always recovered (it could be defined as $\langle \mu\ f :: \mathrm{try}\,(s \rhd f) \rangle$ instead). Strategy $\mathrm{many\_1}$ to succeed requires that the application of the argument strategy must succeed at least once.

Partial traversal. Partial traversal strategies are useful to apply an argument strategy to just one sub-term of a tree:

$$\begin{aligned} \text{once}\, s &\overset{\mathit{def}}{=} \langle \mu\, f :: s \oplus \text{one}\, f \rangle \\ \text{once\_first}\, s &\overset{\mathit{def}}{=} \langle \mu\, f :: s \oslash \text{first}\, f \rangle\, \text{once\_first\_bu}\, s \qquad \overset{\mathit{def}}{=} \quad \langle \mu\, f :: \text{first}\, f \oslash s \rangle \end{aligned}$$

Strategy once non-deterministically chooses a sub-term from the tree for which the application of its argument strategy succeeds. Strategy once_first chooses the first sub-term of the tree for which the application of its argument strategy succeeds in top-down order. Strategy once_first_bu chooses the first sub-term of the tree for which the application of its argument strategy succeeds in bottom-up order.

Complete traversal. The complete structure top-down and bottom-up traversals of trees are defined as:

$$\begin{aligned} \text{topdown}\, s &\overset{\mathit{def}}{=} \langle \mu\, f :: s \triangleright \text{all}\, f \rangle \\ \text{bottomup}\, s &\overset{\mathit{def}}{=} \langle \mu\, f :: \text{all}\, f \triangleright s \rangle \end{aligned}$$

Fixed-point reduction. The two common fixed-point reduction strategies, the leftmost innermost and the leftmost outermost, are easily defined using the other strategies as:

$$\begin{aligned} \text{outermost}\, s &\overset{\mathit{def}}{=} \text{many}(\text{once\_first}\, s) \\ \text{innermost}\, s &\overset{\mathit{def}}{=} \text{many}(\text{once\_first\_bu}\, s) \\ \text{innermost}\, s &\overset{\mathit{def}}{=} \langle \mu\, f :: \text{all}\, f \triangleright \text{try}(s \triangleright f) \rangle \end{aligned}$$

The two definitions of innermost are equivalent, but the second version is more efficient than the first one.

3.4.3 Algebraic properties

The strategy combinators enjoy several interesting algebraic properties [Alves et al., 2005], some of which are presented below. It can be verified that these are consistent with the reduction semantics presented before.

Sequential composition. Sequential composition is associative, nop is its unit and $\perp$ is its zero:

$$\begin{aligned}
(r \triangleright s) \triangleright t &= r \triangleright (s \triangleright t) \\
\mathrm{nop} \triangleright s &= s \\
s \triangleright \mathrm{nop} &= s \\
\perp \triangleright s &= \perp \\
s \triangleright \perp &= \perp
\end{aligned}$$

Thus, $(\triangleright, \mathrm{nop})$ forms a monoid.

Non-deterministic choice. Non-deterministic choice is associative and $\perp$ is its unit, forming a monoid. Futhermore, sequential composition right distributes over non-deterministic choice:

$$\begin{aligned}
(r \oplus s) \oplus t &= r \oplus (s \oplus t) \\
\perp \oplus s &= s \\
s \oplus \perp &= s \\
(r \oplus s) \triangleright t &= (r \triangleright t) \oplus (s \triangleright t)
\end{aligned}$$

Left choice. Like non-deterministic choice, left-choice is also associative and $\perp$ is its unit. However, the right distributivity of sequential composition does not hold for left-choice. Instead, the left preservation of the identity strategy holds:

$$\begin{aligned}
(r \oslash s) \oslash t &= r \oslash (s \oslash t) \\
\perp \oslash s &= s \\
s \oslash \perp &= s \\
\mathrm{nop} \oslash s &= \mathrm{nop}
\end{aligned}$$

Maps on children. The maps on children (all, one and first) all preserve the identity and the failure strategies:

$$\begin{aligned}
\mathrm{all}\,\mathrm{nop} &= \mathrm{nop} \\
\mathrm{all}\,\bot &= \bot \\
\mathrm{one}\,\mathrm{nop} &= \mathrm{nop} \\
\mathrm{one}\,\bot &= \bot \\
\mathrm{first}\,\mathrm{nop} &= \mathrm{nop} \\
\mathrm{first}\,\bot &= \bot
\end{aligned}$$

3.5 Summary

Term rewriting is an important field related with many areas, and, in particular, with theorem proving and automated deduction. The close connection between TRSs and equational logic will be used in Chapter 8 to justify the approach taken in the *Galculator*.

The description given in this chapter deviates from the traditional ones, since it focused in strategic term rewriting systems. Strategic TRSs a have nice algebraic flavor and are related to what is sometimes called "tactics" in theorem proving. In strategic TRSs, strategies are no longer entangled with the environment and can be programmed just like rewriting rules can. Complex strategies can be build using combinators from a few basic strategies. The semantics of these strategies is provided since they will be important in the design of the *Galculator*.

Although the approach is not completely traditional, the classical subjects such as abstract reductions systems with the associated notions of termination and confluence, as well as equational logic and rewriting strategies, were also covered.

Chapter 4

Relation calculus

Relations are ubiquitous in Mathematics, Philosophy and even natural language, because of their intuitive nature. They describe connections between objects and their properties or among objects themselves. Augustus De Morgan, one of the founders of the relation algebra, describes relations [De Morgan, 1860, 1966] as:

> *When two objects, qualities, classes, or attributes, viewed together by the mind, are seen under some connexion, that connexion is called a relation.*

The idea of encoding predicates in terms of relations was initiated by De Morgan in the 1860s and followed by Peirce who, in the 1870s, found interesting equational laws of the calculus of binary relations. Other important contributions are due to Schröder, as explained by Pratt [1992]. The point-free nature of the notation which emerged from this embryonic work was later further exploited by Tarski and his students [Tarski, 1941; Tarski and Givant, 1987]. In the 1980's, Freyd and Ščedrov [1990] developed the notion of an *allegory* (a category whose morphisms are partially ordered) which finally accommodates the binary relation calculus as special case.

Looking closer to the subject, two major kinds of algebra based on relations exist: *relation* algebra and *relational* algebra. Although the names are very similar and can be easily confused, they are quite different. *Relational algebra* is due to Codd [1970] and is the base to the relational database model. The widely used database query language *SQL* [ISO, Nov. 1992] follows the operations defined in relational algebra. The expressive power of this algebra is equivalent to first-order logic.

Relation algebra is based on the De Morgan-Peirce-Schröder relation calculus, formalized by Tarski and his students. This algebra is weaker than relational algebra in terms of expressive power: it is only equivalent to a three variable fragment of first-order logic.

Although being different they share a common point: they use relations and their operations in order to abstract logical quantifiers and data elements.

In this chapter, we will start by introducing relations from a set-theoretical view point. Then, we will elaborate this view towards an algebraic approach, mostly based on the Tarski's view. The following step is to present *fork algebras*, an extension of relation algebra devised in order to surpass the limitations of expressiveness of relation algebra. We conclude by discussing the so-called *point-free-transform* [Tarski and Givant, 1987; Bird and de Moor, 1997]: a mapping from first-order logic into a fork algebra.

4.1 Relations

In this section, we introduce binary relations from a set-theoretical perspective. We restrict ourselves to binary relations, because of their simple yet powerful theory. Relations defined over an arbitrary number of objects can be reduced to the binary case by introducing a pairing operator. The study of binary relations is useful for giving a natural interpretation of fork algebras in terms of concrete relations.

The concepts introduced in this section are quite standard and can be found, e.g., in [Backhouse and Backhouse, 2004; Oliveira and Rodrigues, 2004].

Basic definitions. Given two sets, $\mathcal{A}$ and $\mathcal{B}$, a *binary relation* R can be defined as a subset of their Cartesian product $\mathcal{B} \times \mathcal{A} \overset{def}{=} \{\forall\, b, a : \ b \in \mathcal{B} \wedge a \in \mathcal{A} \ : (b, a)\}$. We will write $R \in \mathcal{B} \sim \mathcal{A}$ or $\mathcal{B} \xleftarrow{R} \mathcal{A}$ to denote such relation. When $\mathcal{A}$ and $\mathcal{B}$ coincide, the relation is said to be an *endo-relation*.

We shall distinguish three special relations: the *empty relation* $\perp$ which does not relate any elements all (corresponds to the empty subset of a Cartesian product); the *universal relation* $\top$ which relates every pair of elements (coincides with the Cartesian product of sets, $\mathcal{B} \times \mathcal{A}$); and the *identity relation* id which relates equal to equal elements (consequently, an endo-relation).

The operations on relations are standard extensions of the respective operations in the underlying set of pairs. Thus, the *converse* on a relation $\mathcal{B} \xleftarrow{R} \mathcal{A}$, denoted by $\mathcal{A} \xleftarrow{R^{\cup}} \mathcal{B}$, is defined as $(a, b) \in R^{\cup} \overset{def}{\Leftrightarrow} (b, a) \in R$. The *meet* (intersection) and

	Definition	
Property	**Point-wise**	**Point-free**
Reflexive	aRa	$id \subseteq R$
Coreflexive	$a'Ra \Rightarrow a' = a$	$R \subseteq id$
Symmetric	$a'Ra \Rightarrow aRa'$	$R \subseteq R^{\cup}$
Anti-symmetric	$a'Ra \wedge aRa' \Rightarrow a = a'$	$R \cap R^{\cup} \subseteq id$
Transitive	$a'Ra'' \wedge a''Ra \Rightarrow a'Ra$	$R \circ R \subseteq R$
Total	$a'Ra \vee aRa'$	$R \cup R^{\cup} = \top$

Table 4.1: Properties of endo-relations, both in point-wise and point-free style, for any endo-relation R on $\mathcal{A}$. In the point-wise definition, variables a, a' and a'' are universally quantified over $\mathcal{A}$.

join (union) of two relations $\mathcal{B} \xleftarrow{R} \mathcal{A}$ and $\mathcal{B} \xleftarrow{S} \mathcal{A}$, are defined, respectively, as $(b,a) \in R \cap S \overset{def}{\Leftrightarrow} (b,a) \in R \wedge (b,a) \in S$, and $(b,a) \in R \cup S \overset{def}{\Leftrightarrow} (b,a) \in R \vee (b,a) \in S$. When intermediate elements exist, relations can be composed. Thus, the composition of relations $\mathcal{C} \xleftarrow{S} \mathcal{B}$ and $\mathcal{B} \xleftarrow{R} \mathcal{A}$, denoted by $\mathcal{C} \xleftarrow{S \circ R} \mathcal{A}$, is defined as $(c,a) \in S \circ R \overset{def}{\Leftrightarrow} \langle \exists\, b \in \mathcal{B} :: cSb \wedge bRa \rangle$.

An ordering can be defined on relations, reflecting the subset ordering on sets of pairs. Thus, relation R is a *sub-relation* of S, denoted as $R \subseteq S$, if and only if all elements related by R and also related by S, i.e., $R \subseteq S \overset{def}{\Leftrightarrow} \langle \forall\, b, a :: bRa \Rightarrow bSa \rangle$.

Following the convention, we will often write bRa to denote that the pair (b,a) belongs to the relation R, i.e., $(b,a) \in R$.

Properties. Relations can be divided and classified according to their properties. Each additional property allows us to elaborate the concepts while preserving the underlying theory. This is a kind of common sense about the way things are developed in mathematics: what makes relations special is their wide scope of application solely based on a few basic properties.

Table 4.1 shows some important properties of endo-relations, both using the "traditional" point-wise definition and the equivalent point-free formulation. The justification for the point-free definition should become clear towards the end of this chapter. Based on these properties, important classes of relation can be distinguished, as graphically shown in Figure 4.1.

For the general case, Table 4.2 shows the definition of four of the most important

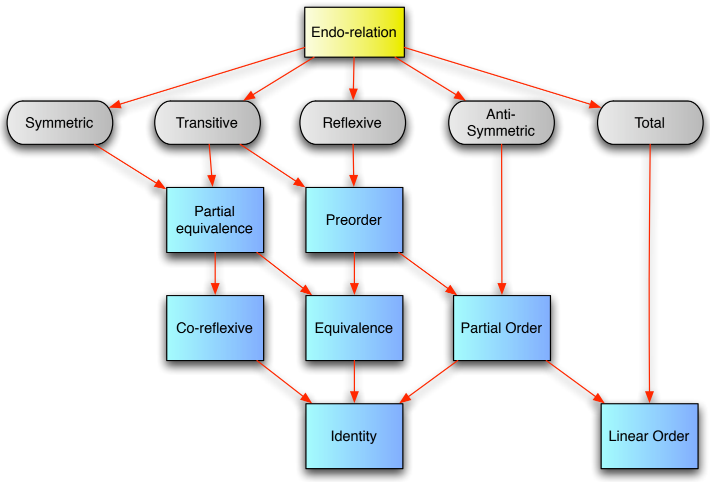


Figure 4.1: Endo-relation taxonomy.

	Definition	
Property	**Point-wise**	**Point-free**
Simple (functional)	$bRa \wedge b'Ra \Rightarrow b = b'$	$R \circ R^{\cup} \subseteq id$
Entire (total)	$\langle \exists\, b'' \in \mathcal{B} :: b''Ra \rangle$	$id \subseteq R^{\cup} \circ R$
Injective	$bRa \wedge bRa' \Rightarrow a = a'$	$R^{\cup} \circ R \subseteq id$
Surjective	$\langle \exists\, a'' \in \mathcal{A} :: bRa'' \rangle$	$id \subseteq R \circ R^{\cup}$

Table 4.2: Properties of relations, both in point-wise and point-free style, for any relation $\mathcal{B} \xleftarrow{R} \mathcal{A}$. In the point-wise definition, variables a and a' are universally quantified over $\mathcal{A}$ and variables b and b' are universally quantified over $\mathcal{B}$.

properties, which lead to the classification presented in Figure 4.2

Orders. *Orders* are special cases of endo-relation in which some properties hold, as the graphical representation of Figure 4.1 shows. In this text, two kinds of order play a special role: preorders and partial orders. A *preorder* $\sqsubseteq$ is a reflexive and transitive relation. A *partial order* is a anti-symmetric preorder. Due to their anti-symmetric behavior, the standard notation for ordering includes symbols such as $\sqsubseteq$, $\preceq$ and $\leqslant$.

Functions. *Functions* are another important special case of relations. This fact in emphasised by using the same notation for both cases. Thus, a function f is denoted as $\mathcal{B} \xleftarrow{f} \mathcal{A}$, where set $\mathcal{A}$ is the *domain* and set $\mathcal{B}$ is the *co-domain* of the function.

Figure 4.2 graphically shows that functions are simple and total relations, two no-

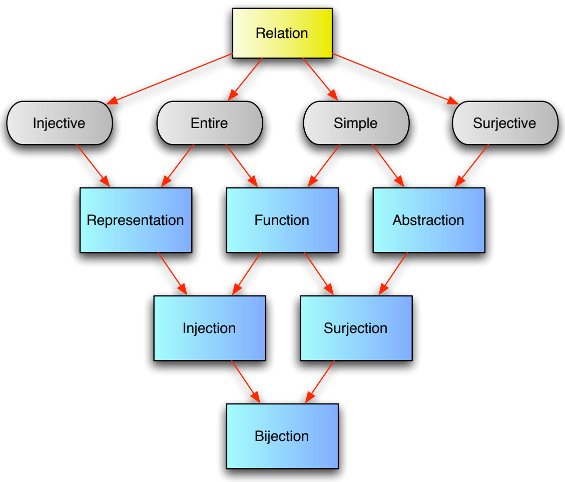


Figure 4.2: Relation taxonomy.

tions explained in Table 4.2. A function may also enjoy additional properties like injectivity and surjectivity. For instance, if these two properties both hold, the function is called a *bijection* or *isomorphism*. We use uppercase identifiers for general relations and lower case identifiers for the specific case of functions.

In the case of functions that share the same domain, the inclusion order on relations coincides with equality [Bird and de Moor, 1997]:

$$f \subseteq g \quad \Leftrightarrow \quad f = g \quad \Leftrightarrow \quad f \supseteq g \tag{4.1}$$

Lifted order relation. A *lifted order relation* inherits the underlying order of the co-domain of functions. Let $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{B} \xleftarrow{g} \mathcal{A}$ be two functions and $(\mathcal{B}, \preceq)$ be an *ordered set*. We define the lifted order $\dot{\preceq}$ on functions as,

$$f \mathrel{\dot{\preceq}} g \quad \overset{def}{\Leftrightarrow} \quad \langle \forall\, a \in \mathcal{A} :: f\ a \preceq g\ a \rangle \tag{4.2}$$

Remark about notation. Our notation for function types deviates from the standard mathematical practice, in the sense that the type of arguments (domain) appears on the right-hand side of the arrow, while the type of the results (co-domain) appears on the left-hand side of the arrow. This allows for a more natural and consistent use of function application and composition. Given functions $\mathcal{C} \xleftarrow{f} \mathcal{B}$ and $\mathcal{B} \xleftarrow{g} \mathcal{A}$, the their composition type is straightforward, i.e., $\mathcal{C} \xleftarrow{f \circ g} \mathcal{A}$ and clearly mirrors the definition

$$(f \circ g)\, a \overset{def}{=} f\,(g\, a)$$

for all $a \in \mathcal{A}$. Using the standard notation, the composition of functions $\mathcal{B} \xrightarrow{f} \mathcal{C}$ and $\mathcal{A} \xrightarrow{g} \mathcal{B}$ yields $\mathcal{A} \xrightarrow{f \circ g} \mathcal{C}$. Note that the argument, intermediate and result types appear in the opposite side with respect to the above definition.

Unlike the particular case of functions, general binary relations do not specify any direction of application because one of the arguments does not determine the other. All relation operations previously described handle pairs of values, being independent of any direction of application.

However, some direction is often implicitly assumed. Relations can be interpreted as non-deterministic, possibly partial, functions which given an argument return none or several related results [Oliveira and Rodrigues, 2004]. Thus, in order to maintain consistency, the same notation is used both for relation and function types. Using a conventional direction for relation application and composition does not impose any limitations because the converse operation can be used to switch the arguments of any relation. Moreover, such a practice also enforces the connection between relations and category theory, which will be exploited in Section 4.6. This convention is also adopted by Bird and de Moor [1997].

4.2 Boolean algebras

Boolean algebra was developed by George Boole as the algebraic counterpart of propositional logic.

Definition. A *Boolean algebra* is a tuple $(\mathcal{A}, \vee, \wedge, \neg, \mathsf{false}, \mathsf{true})$ such that, for all $a, b, c \in \mathcal{A}$ the following axioms are satisfied:

$$a \wedge b = b \wedge a \tag{4.3}$$

$$a \vee b = b \vee a \tag{4.4}$$

$$a \wedge (b \wedge c) = (a \wedge b) \wedge c \tag{4.5}$$

$$a \vee (b \vee c) = (a \vee b) \vee c \tag{4.6}$$

$$a \wedge (a \vee b) = a \tag{4.7}$$

$$a \vee (a \wedge b) = a \tag{4.8}$$

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \tag{4.9}$$

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) \tag{4.10}$$

$$a \wedge \neg a = \mathsf{false} \tag{4.11}$$

$$a \vee \neg a = \mathsf{true} \tag{4.12}$$

Constants true and false are two distinguished elements of $\mathcal{A}$; $\neg$ is an unary operation on $\mathcal{A}$ called *not* (or *complement*). The two binary operations on $\mathcal{A}$, $\wedge$ and $\vee$, are respectively called *and* (or *meet*) and *or* (or *join*). The set $\mathcal{A}$ is the *carrier* of the algebra.

Alternative definition (Huntington). The signature and set of axioms given above is somewhat redundant. It is possible to give an equivalent definition using an elegant axiomatization with just two operations and three axioms due to Huntington [1932][1]. The signature of the algebra is reduced to $(\mathcal{A}, \vee, \neg)$ and the only axioms are:

$$a \vee b = b \vee a \tag{4.13}$$

$$a \vee (b \vee c) = (a \vee b) \vee c \tag{4.14}$$

$$\neg(\neg a \vee b) \vee \neg(\neg a \vee \neg b) = a \tag{4.15}$$

All the operators and identities of the first axiomatization can be derived from the second one [Maddux, 1996].

In the presentation of relation algebra, it is usual to use the Huntington's axiomatization, like established by Tarski and Givant [1987]. However, the first axiomatization is better suited to proof assistants. All the usual operations are primitive and the axioms state important properties which they enjoy like commutativity, associativity or distributivity. This can make things simpler, avoiding the proof of additional theorems and the addition of more definitions. Nevertheless, for the sake of completeness the two versions are presented above.

Relation with lattice theory. Some of the names given to operations (*complement*, *join*, *meet*) resemble well-known concepts from lattice theory [Davey and Priestley,

[1]Many other axiomatizations exist, some of them using less axioms.

1990]. This is not a coincidence: from a Boolean algebra we can infer a partial order $\leqslant$, for all $a, b \in \mathcal{A}$, defined as $a \leqslant b \overset{def}{\Leftrightarrow} a = a \wedge b$ (or equivalently, $a \leqslant b \overset{def}{\Leftrightarrow} a \vee b = b$).

The least element of the order is false and true is the greatest element. Operations $\wedge$ and $\vee$ coincide, respectively, with the infimum and the supremum, with respect to $\leqslant$ ordering.

4.3 Relation algebras

The calculus of binary relations was started by De Morgan in a paper entitled: *On the Syllogism: IV; and on the Logic of Relations* [De Morgan, 1860]. But it was Peirce who greatly developed the subject. Peirce's insight was that the calculus of relations could be separated into two components, one logical and one relative (also called static and dynamic, respectively) [Pratt, 1992]. Each component consists of two binary operations called *disjunction* and *conjunction*, one unary operation called *negation* and two distinguished constants: *true* and *false*. Thus, for each operation in one component there is a corresponding one in the other component. Moreover, Peirce realized that the two components form a logic on their own [Pratt, 1992].

Definition. The notation and formalization of relation algebras presented below follows Tarski and Givant [1987]. The difference is that Tarski and Givant [1987] omit some operators from the algebraic signature and provide their definition using the other operators. We have decided to include all the operators in the signature like in [Priss, 2006a].

A *relation algebra* is a tuple $(\mathcal{R}, +, \cdot, ^{-}, 0, 1, \odot, ^{\smile}, \overset{\circ}{1})$ such that, for any $r, s, t \in \mathcal{R}$, the following axioms hold:

$$(\mathcal{R}, +, \cdot, ^{-}, 0, 1) \text{ is a Boolean algebra} \tag{4.16}$$

$$r \odot (s \odot t) = (r \odot s) \odot t \tag{4.17}$$

$$r \odot \overset{\circ}{1} = r \tag{4.18}$$

$$(r^{\smile})^{\smile} = r \tag{4.19}$$

$$(r + s) \odot t = r \odot t + s \odot t \tag{4.20}$$

$$(r + s)^{\smile} = r^{\smile} + s^{\smile} \tag{4.21}$$

$$(r \odot s)^{\smile} = s^{\smile} \odot r^{\smile} \tag{4.22}$$

Symbol	**Name**	**Arity**	**Static counterpart**	**Definition**
$\overset{\circ}{1}$	Identity	Constant	1	Primitive
$\overset{\circ}{0}$	Diversity	Constant	0	$\overset{\circ}{0}=\overset{\circ}{1}^{-}$
$^{\smile}$	Converse	Unary	$^{-}$	Primitive
$\odot$	Composition	Binary	$\cdot$	Primitive
$\oplus$	Relative sum	Binary	$+$	$r \oplus s = (r^{-} \odot s^{-})^{-}$

Table 4.3: Operations of the relative component of relation algebra.

$$r^{\smile} \odot (r \odot s)^{-} \quad \leqslant \quad s^{-} \tag{4.23}$$

where $r \leqslant s$ is defined as $r \cdot s = r$ (or, equivalently, $r + s = s$). We should notice that although points are omitted from definitions, *relation variables* (like r, s and t above) are used in definitions as placeholders for particular relations.

In this definition, we do not provide a relative counterpart for each static operator; the missing ones can be derived from the other. Table 4.3 gives a description of each operator of the relative component, the respective static operator.

In synthesis, a relation algebra is obtained by expanding the Boolean algebra $(\mathcal{R}, +, \cdot, ^{-}, 0, 1)$ with a monoid structure $(\mathcal{R}, \odot, \overset{\circ}{1})$ and a converse operation. As observed by Pratt [1993], it is interesting to note that the inner structure of the elements of relations is only used by the relative operations. For the static (Boolean) operations, relations are treated just like sets.

Proper relation algebra. In order to provide an interpretation for relation algebras, the concept of proper relation algebras is introduced, that is, algebras of binary relations over a set. A *proper relation algebra* (or relation set algebra) is an algebra $(\mathcal{R}, +, \cdot, ^{-}, 0, 1, \odot, ^{\smile}, \overset{\circ}{1})$ where 1 is an equivalence relation on the Cartesian product of the set $\mathcal{A}$ by itself, i.e., $1 \subseteq \mathcal{A} \times \mathcal{A}$, $\overset{\circ}{1}$ is defined as the set $\{\ x :\ (x, x) \in 1\ : (x, x)\}$, $\mathcal{R}$ is a set of binary relations equal to the powerset of 1, and the operations coincide with their set-theoretical counterparts (as defined in Section 4.1). Table 4.4 provides a summary of this correspondence.

Interpretation. Algebras with elements defined in set-theoretical terms (concrete algebras) can be used to interpret abstract equational classes (abstract algebras). We

Name	Operator	Binary relation
Union (join)	$+$	$\cup$
Intersection (meet)	$\cdot$	$\cap$
Negation (complement)	$^{-}$	$\neg$
Empty relation	0	$\perp$
Universal relation	1	$\top$
Composition	$\odot$	$\circ$
Converse	$^{\smile}$	$^{\cup}$
Diagonal (identity)	$\overset{\circ}{1}$	id
Equality	$=$	$=$
Inclusion	$\leqslant$	$\subseteq$

Table 4.4: Correspondence between proper relation algebra operators and binary relation operations.

informally define an *interpretation* as a mapping from abstract to concrete algebras. This mapping is used to ascribe a meaning to the abstract symbols and expressions.

A relation algebra is *representable* if it is isomorphic to a proper relation algebra [Tarski and Givant, 1987], i.e, its interpretation is an isomorphism.

It is interesting to note that not every interpretation of relation algebras is isomorphic to a proper relation algebra, as shown by Lyndon [1950]. Moreover, relation algebras admit other interpretation besides binary relations; for instance, they can be interpreted with respect to FCA (formal concept analysis) concepts [Priss, 2006a].

Important remark. In this document we will only consider representable relation algebras. Thus, we will drop the relation algebra notation and use the equivalent binary relation notation, whenever no ambiguity arises.

Expressiveness. The development of relation algebra aiming at providing an algebraic counterpart of first-order logic, just like the Boolean algebra is equivalent to propositional logic. The question is: are first-order logic and relation algebra equivalent in the means of expression and derivability?

To answer this important question, Tarski and Givant [1987] provide us with a simple formalism without variables, quantifiers or sentential connectives in which set theory and number theory can be developed. Statements become point-free equalities and the deductive system is very simple: the only inference rule is that of replacing equals by equals. They called this formalism $\mathcal{L}^{\times}$.

$\mathcal{L}^{\times}$ is a meta-system used to explore the properties of relation algebras. Tarski also defines two more formalisms: $\mathcal{L}$ of predicate logic and $\mathcal{L}^{+}$, an extension of $\mathcal{L}$ with the introduction of relational operators (at the meta-level)[2]. He then establishes a one-to-one correspondence[3] between $\mathcal{L}$ and $\mathcal{L}^{+}$. Using a fragment with only three-variables of $\mathcal{L}_3^{+}$, he establishes the most important result: there is a one-to-one correspondence between $\mathcal{L}^{\times}$ and a fragment with three-variables $\mathcal{L}_3$ of $\mathcal{L}$.

The conclusion is that the answer to our original question is negative. $\mathcal{L}^{\times}$ (and consequently relation algebra) can only be used to formalize first-order predicates with at most three variables. In spite of its weak expressive and proof power, Tarski shows that it can be used to formalize almost all known systems of set theory [Tarski and Givant, 1987].

Residuated structure. Backhouse [2004] proposes a different, although not complete, axiomatization of relation algebra, where left and right residuals are used. Residuation is a kind of division where not all the multiplicative inverses exist. The operation of whole division presented in Section 1.1.2 is an example of residuation.

The set theoretic definition of *left residual* or *left division* is, for all x, y of the correct type,

$$x(A\,/\,B)y \quad \overset{def}{=} \quad \langle \forall\, u :: xAu \Leftarrow yBu \rangle \tag{4.24}$$

and for the *right residual* or *right division* is, for all x, y of the correct type,

$$x(A\setminus B)y \quad \overset{def}{=} \quad \langle \forall\, u :: uAx \Rightarrow uBy \rangle \tag{4.25}$$

Left division, $T\,/\,S$, of relations T and S is the the greatest relation R such that $R \circ S \subseteq T$. Right division $R \setminus T$, of relations T and S is the greatest relation S such that $R \circ S \subseteq T$. This can be summarized by the two equations which follow

$$R \circ S \subseteq T \quad \Leftrightarrow \quad R \subseteq T\,/\,S \tag{4.26}$$

[2]The axioms of $\mathcal{L}^{+}$ can be seen as definitions of the relational operators in terms of predicate logic. This is one of the key ideas to the introduction of the point-free-transform as described in Section 4.5

[3]Tarski uses the concept of equipollence between theories in order to show the correspondence in means of expression and proof between them.

$$R \circ S \subseteq T \quad \Leftrightarrow \quad S \subseteq R \setminus T \tag{4.27}$$

Equations (4.26) and (4.27) closely resemble Equation (1.3) about whole division. The difference is that, since multiplication over natural numbers is commutative, right and left division coincide and we just need one equation to define whole division. Not surprisingly, both Equation (4.27) and Equation (4.26) establish a Galois connection.

The axiomatization provided by Tarski and Givant [1987] does not include left and right division as primitive operations because they can be easily defined in terms of the other. Thus, we have the definitions [Pratt, 1992]

$$R \,/\, S \overset{def}{=} \neg(\neg R \circ S^{\cup}) \tag{4.28}$$

$$R \setminus S \overset{def}{=} \neg(R^{\cup} \circ \neg S) \tag{4.29}$$

that can be derived from definitions (4.24) and (4.25).

4.4 Fork algebras

As pointed out by Tarski [1941] an expression like

$$\langle \forall\, x, y, z :: \langle \exists\, z :: xRu \wedge yRu \wedge zRu \rangle\rangle$$

is not expressible in his formalism because of the use of four variables. As explained by Veloso and Haeberer [1991], the main idea of Tarski's relation calculus is to use composition to simulate existential quantification. Therefore, the above expression can be transformed in order to enable composition to replace existential quantification

$$\langle \forall\, x, y, z :: \langle \exists\, z :: xRu \wedge uR^{\cup}y \wedge uR^{\cup}z \rangle\rangle$$

However, because variable u can only be used to compose two terms of the conjunction, the elimination of the existential quantifier is not possible.

Tarski noted that a pairing operation was missing in relation algebra [Tarski and Givant, 1987; van den Bussche, 2001] to allow treating pairs as primitive elements. This would give relation algebra an expressive power equivalent to first-order logic. Several methods were proposed (e.g, [van den Bussche, 2001]); one of the most promising and widely used is fork algebra [Veloso and Haeberer, 1991; Frias et al., 1995, 1997, 2004a]. Fork algebra is an extension of relation algebra which provides a kind of pairing operation, a fork, overcoming the problem of lack of expressiveness.

Definition. A *fork algebra* is a tuple $(\mathcal{R}, \cup, \cap, \neg, \bot, \top, \circ, ^{\cup}, id, \nabla)$ such that, for any $r, s, t, u \in \mathcal{R}$, the following axioms hold:

$$(\mathcal{R}, \cup, \cap, \neg, \bot, \top, \circ, ^{\cup}, id) \text{ is a relation algebra} \tag{4.30}$$

$$r \nabla s = ((id \nabla \top) \circ r) \cap ((\top \nabla id) \circ s) \tag{4.31}$$

$$(r \nabla s)^{\cup} \circ (t \nabla u) = (r^{\cup} \circ t) \cap (s^{\cup} \circ u) \tag{4.32}$$

$$(id \nabla \top)^{\cup} \nabla (\top \nabla id)^{\cup} \subseteq id \tag{4.33}$$

The binary operator ∇ is called *fork*; the other operators are defined in the same way as in relation algebra. Expressions $(id\nabla\top)^{\smile}$ and $(\top\nabla id)^{\smile}$ are *quasi-projections* and we will represent them by π_1 and π_2, respectively. Using this notation, equations (4.31) and (4.33) can be rewritten as:

$$r \nabla s = (\pi_1^{\cup} \circ r) \cap (\pi_2^{\cup} \circ s) \tag{4.34}$$

$$\pi_1 \nabla \pi_2 \subseteq id \tag{4.35}$$

Using the fork operator, it is possible to define a binary product operator[4] in relations $\times$ as

$$r \times s \overset{def}{=} (r \circ \pi_1) \nabla (s \circ \pi_2) \tag{4.36}$$

The axiomatization presented above is adapted from Frias et al. [2004a]. The axioms have been proved to be independent [Frias, 1998; Veloso, 1997].

Proper fork algebras. Like relation algebras can be interpreted in terms of proper relation algebras of binary relations, fork algebras can also be interpreted in terms of proper fork algebras. However, the approach is more contrived, thus we will only provide the idea and the main results. For a more detailed account see [Frias et al., 2004a].

The approach is based on a binary pairing function $\star$. The pairs do not have to coincide with the set-theoretical definition; the only requirement is that $\star$ must be

[4]Frias et al. [2004a] refer to this as the *cross* operator.

injective. A *pre proper fork algebra* is defined by extending a proper relation algebra with the pairing function $\star$ over a domain $\mathcal{U}$ and the fork operator ∇, where ∇ is induced by $\star$. A *proper fork algebra* is then defined as a reduct of the corresponding pre proper fork algebra, where the domain $\mathcal{U}$ and the function $\star$ are forgotten. Therefore, proper fork algebras hide the pairing operation, only exposing the fork operator.

Interpretation. Two important results arise from the representability of fork algebras by proper fork algebras. First, every abstract fork algebra is isomorphic to a proper fork algebra [Frias et al., 2004a, Theorem 1]. The consequence of this theorem is that a natural semantics in terms of binary relations can be given to the abstract fork algebra operators. In this interpretation, quasi-projections behave like projections of a pairing relation, by retrieving the individual components of pairs. The other operators have the standard interpretation used with relation algebras.

Second, let us denote a proper fork algebra by PFA and the set of fork algebra equational axioms by FAE. If $\mathcal{E} \cup \{e\}$ is a set of fork algebra equations, then,

$$\mathcal{E} \models_{\texttt{PFA}} e \quad \Leftrightarrow \quad \mathcal{E} \vdash_{\texttt{FAE}} e$$

This completeness result [Frias et al., 2004a, Theorem 2] yields that any property that holds for binary relations can be derived syntactically using the abstract operations and respective axioms of fork algebra. The converse is also true: syntactically valid derivations correspond to valid properties of binary relations.

Expressiveness. Unlike relation algebras, fork algebras have the same expressive power as first-order logic. Therefore, every first-order formula can be expressed as a point-free fork algebra term and every first-order sentence can be translated to an equation on point-free fork algebra terms. Moreover, for each derivation in first-order logic, there is a corresponding derivation from axioms of fork algebra using equational reasoning. The complete results and proofs can be found in [Frias et al., 2004a].

Other logical systems. Fork algebra encompasses the expressive power of the first-order logic; Frias et al. [2004a] show that this result can be extended to other logics, as well, by using some extensions of the basic fork algebra. This gives us a relational framework, where concepts of several different logics have a uniform representation in an equational theory, allowing for equational reasoning.

Frias et al. [2004a] describe some interpretations of non-classical logics in fork algebra. The general interpretation of *propositional modal logics* is established. Moreover, fork algebras are extended with a *closure* operator forming *closure fork algebras*; these are used to interpret *propositional dynamic logic*. A *choice* operator and an infinitary equational inference rules are added to closure fork algebras, forming the so-called ω-closure fork algebras. *First-order dynamic logic* has an interpretation in terms of these algebras.

4.5 Point-free transform

As is well-known in term rewriting, one must be very careful about variables: free and bound variables make substitutions tricky. This complexity can be overcome by transforming variable-level first-order logic formulæ into *point-free* formulæ involving binary relations only. As we have seen, Frias et al. [2004a] have shown this to be always possible thanks to completeness. In such a *point-free transform* (PF-transform for short) [Tarski and Givant, 1987; Bird and de Moor, 1997; Oliveira and Rodrigues, 2004; Oliveira, 2009] variables are abstracted from formulæ in the same way Backus [1978] develops his algebra of programs. The main difference stays in the fact that we are transforming first-order logic formulæ while Backus was doing so for functional terms only.

Once PF-transformed, formulæ involve binary relations only (R, S, etc.) and relational composition ($R \circ S$) becomes the main "glue" among terms:

$$b(R \circ S)c \quad \overset{def}{=} \quad \langle \exists\, a :: b\ R\ a \wedge a\ S\ c \rangle$$

Foundation. The idea of abstracting variables from terms is quite old so it is difficult to give the credits to someone in particular. However, a systematic approach is given by Tarski and Givant [1987] where formalism $\mathcal{L}^+$ is presented as a definitional extension of the $\mathcal{L}$ formalism of first-order logic. *Definitional extension* means that the extended formalism does not enrich the power of expression and proof of the original one. Thus, the axioms concerning the new operations (the relational operators of $\mathcal{L}^\times$) can be seen as definitions:

$$\langle \forall\, x, y :: x(A \cup B)y \ \Leftrightarrow\ xAy \vee xBy \rangle \tag{4.37}$$

$$\langle \forall\, x, y :: x(\neg A)y \;\Leftrightarrow\; \neg(xAy)\rangle \tag{4.38}$$
$$\langle \forall\, x, y :: x(A \circ B)y \;\Leftrightarrow\; \langle \exists\, z :: xAz \wedge zBy\rangle\rangle \tag{4.39}$$
$$\langle \forall\, x, y :: x(A^{\cup})y \;\Leftrightarrow\; yAx\rangle \tag{4.40}$$
$$A = B \;\Leftrightarrow\; \langle \forall\, x, y :: xAy \;\Leftrightarrow\; xBy\rangle \tag{4.41}$$

Equation (4.41) does not define any operator; it introduces extensional equality between relations. Basically, each definition is the set-theoretical interpretation of the corresponding relation operator.

Using the same method, we can introduce definitions for the other relational operators:

$$\langle \forall\, x, y :: x(A \cap B)y \;\Leftrightarrow\; xAy \wedge xBy\rangle \tag{4.42}$$
$$\langle \forall\, x, y :: x(A \oplus B)y \;\Leftrightarrow\; \langle \forall\, z :: xAz \vee zBy\rangle\rangle \tag{4.43}$$
$$\langle \forall\, x, y :: xidy \;\Leftrightarrow\; x = y\rangle \tag{4.44}$$
$$\langle \forall\, x, y :: x\top y \;\Leftrightarrow\; \mathsf{true}\rangle \tag{4.45}$$
$$\langle \forall\, x, y :: x\bot y \;\Leftrightarrow\; \mathsf{false}\rangle \tag{4.46}$$
$$\langle \forall\, x, y :: x(A \setminus B)y \;\Leftrightarrow\; \langle \forall\, z :: zAx \Rightarrow zBy\rangle\rangle \tag{4.47}$$
$$\langle \forall\, x, y :: x(A \,/\, B)y \;\Leftrightarrow\; \langle \forall\, z :: xAz \Leftarrow yBz\rangle\rangle \tag{4.48}$$
$$A \subseteq B \;\Leftrightarrow\; \langle \forall\, x, y :: xAy \Rightarrow xBy\rangle \tag{4.49}$$

The fork ∇ and product $\times$ operators of fork algebras are defined as:

$$\langle \forall\, x, y, z :: \star(x, y)(A \,\nabla\, B)z \;\Leftrightarrow\; xAz \wedge yBz\rangle \tag{4.50}$$
$$\langle \forall\, x, y, z, w :: \star(x, y)(A \times B) \star (w, z) \;\Leftrightarrow\; xAw \wedge yBz\rangle \tag{4.51}$$

Frias et al. [2004a] present two solutions to obtain a point-free relational term from a given first-order formula. In the first one, a mapping is used to perform an algorithmic translation. However, the result term may not be the most adequate to use in further derivations. The second one is similar to the Tarski's idea we presented above: manipulating an first-order formulæ until reaching the definitional form and then replacing it by the corresponding relation operator.

Related functional results. Using the definitions, more complex translation rules that encompass recurring patterns can be derived. One particular rule of the PF-transform which is specially helpful in removing variables from expressions is

$$\langle \forall\, b, a :: b(f^{\cup} \circ R \circ g)a \;\Leftrightarrow\; (f\, b)\; R\; (g\, a)\rangle \tag{4.52}$$

where f and g are functions. In fact, $\langle \forall\; b, a :: (f\, b)\; R\; (g\, a)\rangle$ is just a shorthand for $\langle \forall\; b, a :: \langle \exists\; b', a' :: b' f a \wedge b' R a' \wedge a' g a\rangle\rangle$ from which the above definition arises immediately by the application of the converse to f and the definition of composition twice.

Example (injectivity). As an example of the application of the PF-transform, let us verify the equivalence between the point-free and point-wise definitions of injectivity given in Table 4.2 of Section 4.1 (this calculus is adapted from [Oliveira, 2008]):

$$\begin{array}{ll}
 & \langle \forall\, b, a, a' :: bRa \wedge bRa' \Rightarrow a = a'\rangle \\
\Leftrightarrow & \quad \{\text{ Rules of quantification and converse (4.40) . }\} \\
 & \langle \forall\, a, a' :\; \langle \exists\, b :: aR^{\cup}b \wedge bRa'\rangle\; : a = a'\rangle \\
\Leftrightarrow & \quad \{\text{ Composition (4.39). }\} \\
 & \langle \forall\, a, a' :\; a(R^{\cup} \circ R)a'\; : a = a'\rangle \\
\Leftrightarrow & \quad \{\text{ Rules of quantification. }\} \\
 & \langle \forall\, a, a' :: a(R^{\cup} \circ R)a' \Rightarrow a = a'\rangle \\
\Leftrightarrow & \quad \{\text{ Identity (4.44). }\} \\
 & \langle \forall\, a, a' :: a(R^{\cup} \circ R)a' \Rightarrow a id a'\rangle \\
\Leftrightarrow & \quad \{\text{ Inclusion (4.49). }\} \\
 & R^{\cup} \circ R \subseteq id
\end{array}$$

The rules of quantification are summarized in Appendix B.

Example (reflexivity). Following the same approach as in the previous example, let us also verify the equivalence between the point-wise and point-free definitions of reflexivity given in Table 4.1 of Section 4.1:

$$
\begin{array}{ll}
 & \langle \forall\ a :: aRa \rangle \\
\Leftrightarrow & \qquad \{ \text{ Assuming a variable } a' \text{ equal to } a. \ \} \\
 & \langle \forall\ a, a' :: a = a' \Rightarrow aRa' \rangle \\
\Leftrightarrow & \qquad \{ \text{ Identity (4.44). } \} \\
 & \langle \forall\ a, a' :: a\,id\,a' \Rightarrow aRa' \rangle \\
\Leftrightarrow & \qquad \{ \text{ Inclusion (4.49). } \} \\
 & id \subseteq R
\end{array}
$$

Therefore, we can add the following equivalence to our collection of point-free transform rules:

$$id \subseteq R \quad \Leftrightarrow \quad \langle \forall\ a :: aRa \rangle \tag{4.53}$$

Example (lifted order relation). Recall the lifted order relation on functions defined in Section 4.1. Given two functions $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{B} \xleftarrow{g} \mathcal{A}$, where $(\mathcal{B}, \preceq)$ is an *ordered set*, the lifted order $\dot{\preceq}$ on function is defined as

$$f \mathrel{\dot{\preceq}} g \quad \overset{def}{\Leftrightarrow} \quad \langle \forall\ a \in \mathcal{A} :: f\ a \preceq g\ a \rangle \tag{4.54}$$

In order to calculate the equivalent point-free definition, the introduction of a shunting rule for functions is needed. Given a function f and two arbitrary relations R and S, the following holds:

$$R \subseteq f^{\cup} \circ S \quad \Leftrightarrow \quad f \circ R \subseteq S \tag{4.55}$$

More details about this equivalence will be provided in Section 6.3.

Let us calculate the point-free definition of the lifted order $\dot{\preceq}$:

$$
\begin{array}{ll}
 & f \mathrel{\dot{\preceq}} g \\
\Leftrightarrow & \qquad \{ \text{ Definition (4.54). } \} \\
 & \langle \forall\ a \in \mathcal{A} :: f\ a \preceq g\ a \rangle \\
\Leftrightarrow & \qquad \{ \text{ Related function results (4.52). } \}
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \forall\, a \in \mathcal{A} :: a(f^{\cup} \circ \preceq \circ g)a \rangle \\
\Leftrightarrow & \qquad \{ \text{ Reflexive relations (4.53). } \} \\
 & id \subseteq f^{\cup} \circ \preceq \circ g \\
\Leftrightarrow & \qquad \{ \text{ Shunting of functions (4.55). } \} \\
 & f \circ id \subseteq \preceq \circ g \\
\Leftrightarrow & \qquad \{ \text{ Unit of composition (4.18). } \} \\
 & f \subseteq \preceq \circ g
\end{array}
$$

Thus, we conclude that the lifted order $\dot{\preceq}$ can be alternatively defined as

$$f \mathrel{\dot{\preceq}} g \quad \overset{def}{\Leftrightarrow} \quad f \subseteq \preceq \circ g \tag{4.56}$$

4.6 Categories and allegories

To complete the discussion about relation calculus, we should give a look to its connection with category theory and allegory theory. Allegory theory arises as a generalization of categories when dealing with relations instead of functions. This introduction to categories and allegories is very basic: it only covers the definitions and concepts necessary for the understanding of the rest of the text. The interested reader can consult references [Mac Lane, 1971; Freyd and Ščedrov, 1990; Bird and de Moor, 1997; Rydeheard and Burstall, 1988] for complete details.

4.6.1 Categories

Category theory [Mac Lane, 1971] was developed to deal with mathematical structures in an abstract way. Instead of dealing with the structures directly, category theory studies the structure preserving functions (morphisms) between them. This parallels the point-free calculus presented before where variables are not relevant, only the relation between them. Thus, it is not surprising that the concepts studied before are instances of categories.

Definition. A *category* is a collection of *objects* and a collection of *arrows* (or *morphisms*) satisfying the following conditions:

1. Each arrow f has a *source* and a *target* object, denoted as $B \xleftarrow{f} A$, where A is its source and B is its target;

2. For each object A, there exists an *identity* arrow $A \xleftarrow{id_A} A$;

3. Every pair of arrows of the form $A \xleftarrow{f} B$ and $B \xleftarrow{g} C$ can be composed forming another arrow $A \xleftarrow{f \circ g} C$. The following must hold:

   (a) Composition is associative, i.e., for all $A \xleftarrow{f} B$, $B \xleftarrow{g} C$ and $C \xleftarrow{h} D$,
   $$f \circ (g \circ h) \quad = \quad (f \circ g) \circ h \,;$$

   (b) The identity arrow is the unit of composition, i.e., for all $B \xleftarrow{f} A$,
   $$id_B \circ f \quad = \quad f \quad = \quad f \circ id_A \,.$$

Examples of categories. The concepts of preorder, function and relation introduced before form categories. Table 4.6.1 summarizes how objects and arrows are instantiated together with the definition of the respective identity and composition.

Functors. A *functor* $\mathcal{F}$ is a mapping between categories, taking objects to objects and arrows to arrows, i.e., a functor is required to satisfy:

$$\mathcal{F}\,A \xleftarrow{\mathcal{F}\,f} \mathcal{F}\,B \quad \Leftarrow \quad A \xleftarrow{f} B \tag{4.57}$$

Additionally, functors must preserve identities and composition:

$$\mathcal{F}\,id_A \quad = \quad id_{\mathcal{F}\,A} \tag{4.58}$$
$$\mathcal{F}\,(f \circ g) \quad = \quad \mathcal{F}\,f \circ \mathcal{F}\,g \tag{4.59}$$

Functors can be seen as arrows in a category where objects are themselves categories. When the source and target categories of a functor coincide, it is called *endofunctor*.

Category of ordered sets (`Ord`)	
Objects	Elements of a set $\mathcal{P}$.
Arrows	For objects $p, q \in \mathcal{P}$ the *unique* arrow from p to q exists if and only if $p \sqsubseteq q$.
Identity	Reflexivity of the $\sqsubseteq$ preorder.
Composition	Transitivity of the $\sqsubseteq$ preorder.
Category of total functions (`Fun`)	
Objects	Sets.
Arrows	Total (set-theoretical) functions.
Identity	Identity function on a set.
Composition	Set-theoretical composition of functions with coinciding intermediate objects.
Category of relations (`Rel`)	
Objects	Sets.
Arrows	Relations—a relation $\mathcal{A} \xleftarrow{R} \mathcal{B}$ is a subset of the Cartesian product $\mathcal{A} \times \mathcal{B}$.
Identity	Identity relation on a set.
Composition	Relation composition with coinciding intermediate objects.

Table 4.5: Examples of categories.

4.6.2 Allegories

As Bird and de Moor [1997] put it, *"Allegories are to the algebra of relations as categories are to the algebra of functions."*. Allegories [Freyd and Ščedrov, 1990] are an extension of categories inspired by the `Rel` category. This section will mostly follow [Bird and de Moor, 1997].

Definition. An *allegory* [Freyd and Ščedrov, 1990; Bird and de Moor, 1997] extends a category with:

1. A partial *inclusion* order $\subseteq$ which allows for comparing arrows with the same source and target. Composition must be monotonic concerning this order.
2. A *meet* arrow $\cap$ for which the universal property of meet holds, i.e., given arrows $\mathcal{B} \xleftarrow{R} \mathcal{A}$ and $\mathcal{B} \xleftarrow{S} \mathcal{A}$, then, for all arrows $\mathcal{B} \xleftarrow{X} \mathcal{A}$,
$$X \subseteq (R \cap S) \quad \Leftrightarrow \quad X \subseteq R \wedge X \subseteq S \tag{4.60}$$
3. A *converse* arrow $^\cup$ which, for any arrow $\mathcal{B} \xleftarrow{R} \mathcal{A}$, is monotonic, contravariant and an involution, i.e.,
$$R \subseteq S \quad \Leftrightarrow \quad R^\cup \subseteq S^\cup \tag{4.61}$$
$$(R \circ S)^\cup \quad \Leftrightarrow \quad S^\cup \circ R^\cup \tag{4.62}$$
$$(R^\cup)^\cup \quad \Leftrightarrow \quad R \tag{4.63}$$
4. An additional axiom is needed to connect all the additional operations. This is known as the *modular law*:
$$(R \circ S) \cap T \quad \subseteq \quad R \circ (S \cap (R^\cup \circ T)) \tag{4.64}$$

Tabular allegories. Given a relation $\mathcal{A} \xleftarrow{R} \mathcal{B}$, a pair of functions $\mathcal{A} \xleftarrow{f} \mathcal{C}$ and $\mathcal{B} \xleftarrow{f} \mathcal{C}$ is called a *tabulation* of R if

$$R = f \circ g^\cup \quad \text{and} \quad (f^\cup \circ f) \cap (g^\cup \circ g) = id$$

If every arrow of a category has a tabulation, this is said to be *tabular*.

Tabulations were introduced because allegories admit models which are not set-theoretical relations [Bird and de Moor, 1997], making allegories closer to relations in terms of proof. Some proof about relations are not possible in allegories without this requirement. Thus, tabulations allow for proofs in allegories resembling point-wise derivations.

Relators. Relators are the relational counterpart of functors. Let $\mathtt{A} \xleftarrow{\mathcal{F}} \mathtt{B}$ be a functor, where A and B are tabular allegories. Then $\mathcal{F}$ is said to be a *relator* if $\mathcal{F}$ is monotonic, i.e., for all R and S,

$$\mathcal{F}\,R \subseteq \mathcal{F}\,S \quad \Leftarrow \quad R \subseteq S \tag{4.65}$$

An alternative formulation comes from a theorem [Bird and de Moor, 1997, Theorem 5.1] which states that a functor is a relator if and only if it preserves converse, i.e., for all R

$$(\mathcal{F}\,R)^{\cup} \quad = \quad \mathcal{F}\,R^{\cup}$$

Another lemma [Bird and de Moor, 1997, Lemma 5.1] states that relators preserve functions, i.e., given a relator $\mathcal{F}$ and a function f, $\mathcal{F}\,f$ is still a function.

4.7 Summary

The relation calculus took the first steps at the same time as formal logic and their development was mutual influenced. However, the success of logics overshadowed relations in such a way that only many year later people started to look at the potential of relation calculus again. In this chapter, the link between abstract relation algebras, concrete binary relations and logics was exploited. As it was explained, relations algebras are only equivalent to a three-variable fragment of first-order logic. Fork algebras solved this problem with the introduction of a pairing mechanism (the "fork"), making them equivalent in terms of expressive and deductive power to first-order logic. What makes relation and fork algebras so attractive is that the only needed inference rule is substitution of equals by equals. Thus, proofs can be conducted in an equational theory using a relatively small set of simple equational axioms.

Moreover, since the algebraic approach abstracts variables from expressions, it does not suffer of the problem of variable capture. This is known as the “point-free” style. The process of translating logical expression to point-free relational expression is known as “point-free transform” and it was also discussed in this chapter.

Chapter 5

Galois connections

Chapter 1 has already introduced the concept of a Galois connection and some of its nice properties. In this chapter, we will provide a deeper overview of the theory of Galois connections.

We start with a short historical perspective of the concept of Galois connection, derived from the original work of Évariste Galois (1811 – 1832). Then, several equivalent definitions of Galois connections are presented and some related concepts (such as pair algebras) are introduced. In Section 5.3 we provide necessary and sufficient conditions for the existence of adjoint functions. Section 5.4 will present the most important properties of Galois connections. Sections 5.5 and 5.6 will discuss an important feature of Galois connections: the ability to build new connections either from binary relations or from other existing Galois connections, leading to an algebraic approach. Finally, we will show the link between Galois connections and category theory.

5.1 Historical perspective

Évariste Galois (1811 – 1832) was a French mathematician born near Paris. Although the importance of his work lies in the theory of polynomial equations, he is mostly known for his almost legendary life. He performed badly at school and during his life no sign of genius was found in his work. He failed admission to the École Polytechique twice; the memoirs he submitted to the Academy of Sciences were lost or judged as *'incomprehensible'*. Because of his republican ideas together with several misunderstandings he was arrested and acquitted several times until he got actually condemned. The tragic hero role is completed with his death, consequence of a duel due to an affair with a woman [Stewart, 1989, 1992].

Although his work was never recognized during his short life, he is considered one of the founders of modern abstract algebra. The problem he took at hands was to determine which polynomial equations could be solved using radical expressions and those which cannot. A polynomial equation is an equation of the form

$$f(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_0 = 0$$

and a radical expression is built up using the operations of addition, subtraction, multiplication, division, and the nth roots (for $n = 2, 3, 4, \ldots$) on the coefficients a_i. For quadratic, cubic and quartic equations it was known that solutions using radicals are possible. However, the quintic (degree 5) equation was problematic; it took some time until Abel proved that the solution using radicals is not possible in general [Tignol, 2001].

The approach of Galois was more general: instead of having a prove for each degree he wanted to discover necessary and sufficient conditions under which the solution using radicals is possible. His idea was to exploit the symmetries of the solutions (also called zeros) of a polynomial equation. He observed that some solutions are naturally related, i.e., if we take any polynomial equation with rational coefficients valid for some of the zeros, the permutation of related zeros does not change the validity of the equation. Permutations have an identity; and the compositions of two valid permutations is still a valid permutation. Galois recognized permutations as a *group*, a concept that he invented [Stewart, 1989]. Besides relating the solution of an equation to a group (called the Galois group), he realized that the structure of this group determines if the equation is solvable by radicals or not [Galois, 1846, 1897].

Today's approach is more abstract but follows on the sames ideas. It involves field extensions instead of polynomials and automorphism groups instead of symmetry groups. It describes a relation between subfields of an extended field and subgroups of a group of automorphisms.

What today is known as classical Galois theory includes contributions, not only of Galois, but also by some other important mathematicians like Lagrange, Abel and Dedekind. The so-called "modern" Galois theory settled down in the 20th century thanks to the work of Birkhoff [1940] about *polarities*, Ore [1944] about *Galois connexions* and Schmidt [1953] about *Galois correspondences of mixed type*, better known as *adjunctions*. These works extended the Galois theory to order and lattice theory.

The original formulation was thus subsumed by these. Denecke et al. [2004] give a complete account of the history of Galois connections from its roots until the modern approaches.

5.2 Definitions

Galois connections are an important concept that naturally arises in many different fields. As often happens with notions central to mathematics, several different, but equivalent, definitions exist. In this section, we recall the concept of a preorder from Section 4.1 to introduce these definitions. Furthermore, we will try to provide some intuition about the concept.

Definition. Given two *preordered* sets $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ and $(\mathcal{B}, \sqsubseteq_{\mathcal{B}})$ and two functions $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{A} \xleftarrow{g} \mathcal{B}$, the pair (f, g) is a *Galois connection* if and only if, for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$:

$$f\,a \sqsubseteq_{\mathcal{B}} b \quad \Leftrightarrow \quad a \sqsubseteq_{\mathcal{A}} g\,b \tag{5.1}$$

Function f (resp. g) is referred to as the *lower adjoint* (resp. *upper adjoint*) of the connection[1].

Two kinds of definitions of Galois connections exist [Priestley, 2000]: an order-preserving and an order-inversing version. As we shall see, version (5.1) is the order-preserving one, where adjoint functions have asymmetric definitions. Later on, when polarities are introduced, we will present the order-inversing version in which there is no distinction between functions [Melton et al., 1986], thus being symmetric. However, both forms lead to the same results.

Partial orders. The above definition uses the weakest assumption to establishing a Galois connections: only preordered sets are required. However, this is not enough for several applications like, for instance, proofs based on the indirect equality principle. In these cases, anti-symmetry is fundamental to establish equalities, thus implying the

[1]Some authors use a different nomenclature. Lower and upper adjoint are sometimes called, respectively, left and right adjoint [Priestley, 2000], coadjoint and adjoint [Erné et al., 1993] or residuated and residual map [Melton et al., 1986].

use of partially ordered sets. In other cases, lattices or even complete lattices may be required.

In this presentation, we will try to provide results in their maximum generality. Only when stronger requirements are needed, we will require partially ordered sets or lattices in the definitions.

Notation. A standard notation for Galois connections does not exist. In this document we will display Galois connections using the graphical notation introduced in [Silva and Oliveira, 2008]

$$\begin{tikzcd}
\mathcal{A} \arrow[loop left, "\sqsubseteq_{\mathcal{A}}"] \arrow[r, bend left, "f"] & \mathcal{B} \arrow[loop right, "\sqsubseteq_{\mathcal{B}}"'] \arrow[l, bend left, "g"]
\end{tikzcd}$$

which we in-line in text by writing $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}) \xleftarrow{(f,g)} (\mathcal{B}, \sqsubseteq_{\mathcal{B}})$. Both notations always represent the source domain of the lower adjoint on the left. As we shall see, the arrow notation emphasizes the categorial structure of Galois connections, which are closed under composition and exhibit identity.

Sections and families of Galois connections. Galois connections' adjoints are unary functions: they only take one argument. Nevertheless, many important examples of Galois connections arise from binary operators. Therefore, in order to form Galois connections one of their arguments must be fixed, so that they become unary functions on the other argument. In general, given binary operator θ, one defines two unary *sections*[2], $(a\theta)$ and (θb), for every suitably typed a and b, such that $(a\theta)\, x = a\, \theta\, x$ (called the *left section*) and $(\theta b)\, y = y\, \theta\, b$ (called the *right section*), respectively. Thus, instead of having just one Galois connection, we build a family of Galois connections indexed by the frozen argument (a and b above).

The definition of Galois connection where adjoints are left sections becomes: given two binary functions $\mathcal{B} \xleftarrow{\theta} \mathcal{A} \times \mathcal{C}$ and $\mathcal{A} \xleftarrow{\phi} \mathcal{B} \times \mathcal{C}$, for a given $c \in \mathcal{C}$, the pair $(\theta c, \phi c)$ is a *Galois connection family* indexed by c, if and only if for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$:

$$(\theta c)\, a \sqsubseteq_{\mathcal{B}} b \quad \Leftrightarrow \quad a \sqsubseteq_{\mathcal{A}} (\phi c)\, b \tag{5.2}$$

[2]This terminology is taken from functional programming, where sections are a very popular programming device [Peyton Jones, 2003]. It is also used by Backhouse et al. [2002].

In our notation,

$$\begin{tikzcd}
\mathcal{A} \arrow[loop left, "\sqsubseteq_{\mathcal{A}}"] \arrow[r, bend left, "(\theta c)"] & \mathcal{B} \arrow[loop right, "\sqsubseteq_{\mathcal{B}}"] \arrow[l, bend left, "(\phi c)"]
\end{tikzcd}$$

This is the case for left sections; an analogous definition can be derived to right sections providing that domains of functions are correct.

Ore's definition. The definition we gave before is due to Schmidt [1953]. Equation (5.1) is quite elegant and easy to remember because of its symmetry. An alternative definition is due to Ore [1944] which defines Galois connections in terms of the monotonicity of adjoints and their cancellation properties. Thus, given two *preordered* sets $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ and $(\mathcal{B}, \sqsubseteq_{\mathcal{B}})$ and two functions $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{A} \xleftarrow{g} \mathcal{B}$, the pair (f, g) is a Galois connection if and only if the following conditions hold:

1. For all $a \in \mathcal{A}$ and $b \in \mathcal{B}$, $a \sqsubseteq_{\mathcal{A}} g\,(f\,a)$ and $f\,(g\,b) \sqsubseteq_{\mathcal{B}} b$;

2. f and g are monotonic.

The generalization for the case where adjoints are sections of binary operators is straightforward.

Hybrid definition. Another definition mixes monotonicity and the cancellation property of the upper adjoint with Equation (5.1) weakened to an implication. This definition is mostly useful to find the lower adjoint when the upper adjoint is a known function [Backhouse, 2000].

Let $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ and $(\mathcal{B}, \sqsubseteq_{\mathcal{B}})$ be two *preordered* sets and $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{A} \xleftarrow{g} \mathcal{B}$ two functions. The pair (f, g) is a Galois connection if and only if the following conditions hold:

1. g is monotonic;

2. For all $a \in \mathcal{A}$, $a \sqsubseteq_{\mathcal{A}} g\,(f\,a)$;

3. For all $a \in \mathcal{A}$ and $b \in \mathcal{B}$, $a \sqsubseteq_{\mathcal{A}} g\,b \Rightarrow f\,a \sqsubseteq_{\mathcal{B}} b$.

A dual version of this definition exists based on the monotonicity of f [Priestley, 2000].

Pair algebras. Hartmanis and Stearns [1964, 1966] introduced pair algebras, a related concept which provides a good way of further understanding Galois connections.

Given two *posets* $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ and $(\mathcal{B}, \sqsubseteq)$, a binary relation $\mathcal{B} \xleftarrow{R} \mathcal{A}$ forms a *pair algebra* if there exist functions $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{A} \xleftarrow{g} \mathcal{B}$ such that, for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$,

$$(b, a) \in R \quad \Leftrightarrow \quad f\,a \sqsubseteq_{\mathcal{B}} b \quad \text{and} \quad (b, a) \in R \quad \Leftrightarrow \quad a \sqsubseteq_{\mathcal{A}} g\,b \tag{5.3}$$

It is not difficult to see that the concepts are equivalent: a pair algebra establishes a Galois connection and a Galois connection connections defines a pair algebra [Backhouse and Backhouse, 2004]. Moreover, from Equation (5.3) we infer that the relation established by $f\,a \sqsubseteq_{\mathcal{B}} b$ is equal to the relation established by $a \sqsubseteq_{\mathcal{A}} g\,b$ [Backhouse, 2000].

Recalling our examples from Sections 1.1.1 and 1.1.2 concerning whole division, we would obtain a pair algebra R_c, such that, for all a, b and $c \in \mathbb{N}, c \neq 0$,

$$a(\times c) \leqslant b \quad \Leftrightarrow \quad (a, b) \in R_c \quad \Leftrightarrow \quad a \leqslant b(\div c)$$

Table 5.2 shows some values of the pair algebra R_c when we fix c to be equal to 3 and 7 (these values are arbitrary and this example would be valid for any instantiation of $c \neq 0$). This gives us an insight about the equivalence between the three relations and helps us understand the theoretical result [Backhouse, 2000, 2004] which states that given a pair algebra R we can define functions f and g, satisfying (5.3), for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$,

$$f\,a \overset{def}{=} \langle \textstyle\bigsqcap b : \ (a, b) \in R \ : b \rangle \tag{5.4}$$

$$g\,b \overset{def}{=} \langle \textstyle\bigsqcup a : \ (a, b) \in R \ : a \rangle \tag{5.5}$$

For instance, this means that from R_3 we can extract the definition of function $(\times 3)$ as $0 \times 3 = 0, 1 \times 3 = 3, 2 \times 3 = 6, \ldots$ and the function $(\div 3)$ as $0 \div 3 = 0, 1 \div 3 = 0, 2 \div 3 = 0, 3 \div 3 = 1, 4 \div 3 = 1, \ldots$

$a(\times 3) \leqslant b$	$a \leqslant b(\div 3)$	R_3	$a(\times 7) \leqslant b$	$a \leqslant b(\div 7)$	R_7
$0 \times 3 \leqslant 0$	$0 \leqslant 0 \div 3$	$(0,0)$	$0 \times 7 \leqslant 0$	$0 \leqslant 0 \div 7$	$(0,0)$
$0 \times 3 \leqslant 1$	$0 \leqslant 1 \div 3$	$(0,1)$	$0 \times 7 \leqslant 1$	$0 \leqslant 1 \div 7$	$(0,1)$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$1 \times 3 \leqslant 3$	$1 \leqslant 3 \div 3$	$(1,3)$	$1 \times 7 \leqslant 7$	$1 \leqslant 7 \div 7$	$(1,7)$
$1 \times 3 \leqslant 4$	$1 \leqslant 4 \div 3$	$(1,4)$	$1 \times 7 \leqslant 8$	$1 \leqslant 8 \div 7$	$(1,8)$
$1 \times 3 \leqslant 5$	$1 \leqslant 5 \div 3$	$(1,5)$	$1 \times 7 \leqslant 9$	$1 \leqslant 9 \div 7$	$(1,9)$
$1 \times 3 \leqslant 6$	$1 \leqslant 6 \div 3$	$(1,6)$	$1 \times 7 \leqslant 10$	$1 \leqslant 10 \div 7$	$(1,10)$
$1 \times 3 \leqslant 7$	$1 \leqslant 7 \div 3$	$(1,7)$	$1 \times 7 \leqslant 11$	$1 \leqslant 11 \div 7$	$(1,11)$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$2 \times 3 \leqslant 6$	$2 \leqslant 6 \div 3$	$(2,6)$	$2 \times 7 \leqslant 14$	$2 \leqslant 14 \div 7$	$(2,14)$
$2 \times 3 \leqslant 7$	$2 \leqslant 7 \div 3$	$(2,7)$	$2 \times 7 \leqslant 15$	$2 \leqslant 15 \div 7$	$(2,15)$
$2 \times 3 \leqslant 8$	$2 \leqslant 8 \div 3$	$(2,8)$	$2 \times 7 \leqslant 16$	$2 \leqslant 16 \div 7$	$(2,16)$
$2 \times 3 \leqslant 9$	$2 \leqslant 9 \div 3$	$(2,9)$	$2 \times 7 \leqslant 17$	$2 \leqslant 17 \div 7$	$(2,17)$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$3 \times 3 \leqslant 9$	$3 \leqslant 9 \div 3$	$(3,9)$	$3 \times 7 \leqslant 21$	$3 \leqslant 21 \div 7$	$(3,21)$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$10 \times 3 \leqslant 30$	$10 \leqslant 30 \div 3$	$(10,30)$	$10 \times 7 \leqslant 70$	$10 \leqslant 70 \div 7$	$(10,70)$
$10 \times 3 \leqslant 31$	$10 \leqslant 31 \div 3$	$(10,31)$	$10 \times 7 \leqslant 71$	$10 \leqslant 71 \div 7$	$(10,71)$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Table 5.1: Example of pair algebra R_c induced by whole division, for $c = 3$ and $c = 7$.

Explicit definition. If in Equations (5.4) and (5.5) we replace the pair algebra R by an equivalence given by (5.3), we get, for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$,

$$f\,a \overset{def}{=} \langle \bigsqcap\, b \in \mathcal{B} :\ a \sqsubseteq_{\mathcal{A}} g\,b\ : b \rangle \tag{5.6}$$

$$g\,b \overset{def}{=} \langle \bigsqcup\, a \in \mathcal{A} :\ f\,a \sqsubseteq_{\mathcal{B}} b\ : a \rangle \tag{5.7}$$

This result gives the explicit definition of one adjoint in terms of the other.

Perfect connections. Let $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ be a Galois connection. Then, the following are equivalent:

1. f is surjective;
2. g is injective;
3. $g\,b = \langle \bigsqcup\, a \in \mathcal{A} :\ f\,a = b\ : a \rangle$;
4. $f \circ g = id_{\mathcal{B}}$.

In this situation, $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ is said to be a *perfect Galois connection* (also called *Galois insertion* or *retraction*) [Priestley, 2000; Bělohlávek, 2000; Hankin, 2005].

An alternative formulation for perfect Galois connections [von Karger, 2000; Bělohlávek, 2000] arises by strengthening the other cancellation rule to equality instead. The following are equivalent:

1. f is injective;
2. g is surjective;
3. $f\,a = \langle \bigsqcap\, b \in \mathcal{B} :\ a = g\,b\ : b \rangle$;
4. $g \circ f = id_{\mathcal{A}}$.

Making perfect connections. From a Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$, we can always obtain a perfect Galois connection by enforcing that the lower adjoint is injective. This can be achieved by partitioning the domain of f in equivalence classes, each class containing the elements that share the same image, i.e., supposing $\mathcal{A}$ as the domain of f, elements a and a' of $\mathcal{A}$ belong to the same equivalence class if and only if

$f\,a = f\,a'$. Then, all elements of an equivalence class are represented by their infimum in the new perfect connection [Hankin, 2005].

We define the reduction operator $\mathcal{A} \xleftarrow{\varsigma} \mathcal{A}$ as, for all $a \in \mathcal{A}$,

$$\varsigma\, a \;\overset{def}{=}\; \langle \textstyle\bigsqcap a' \in \mathcal{A} :\; f\,a = f\,a' \;: a' \rangle \tag{5.8}$$

Then, $(\varsigma(\mathcal{A}), \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ is a perfect Galois connection, where $\varsigma(\mathcal{A})$ is the set resulting from the application of ς to all elements of $\mathcal{A}$.

Closure operators. Given a poset $(\mathcal{A}, \sqsubseteq)$, a *closure operator* is an endo-function $\mathcal{A} \xleftarrow{c} \mathcal{A}$ where for all $a, a' \in \mathcal{A}$,

$$a \;\sqsubseteq\; c\,a \tag{5.9}$$

$$a \sqsubseteq a' \;\Rightarrow\; c\,a \sqsubseteq c\,a' \tag{5.10}$$

$$c\,(c\,a) \;=\; c\,a \tag{5.11}$$

i.e., it is an increasing (5.9), monotonic (5.10) and idempotent (5.11) operator [Priestley, 2000; Erné et al., 1993]. An element $a \in \mathcal{A}$ is said to be closed if $c\,a = a$ and the set of closed elements of $\mathcal{A}$ is defined as

$$\mathcal{A}_c \;\overset{def}{=}\; \{\, a \in \mathcal{A} :\; c\,a = a \;: a\}$$

Interior operators. Given $\mathcal{A} \xleftarrow{i} \mathcal{A}$ monotonic, idempotent and decreasing, i.e., such that for all $a \in \mathcal{A}$,

$$i\,a \sqsubseteq a \tag{5.12}$$

i is said to be an *interior operator* [Erné et al., 1993]. An element $a \in \mathcal{A}$ is said to be open if $i\,a = a$ and the set of open elements of $\mathcal{A}$ is defined as

$$\mathcal{A}_i \;\overset{def}{=}\; \{\, a \in \mathcal{B} :\; i\,a = a \;: a\}$$

Closures and Galois connections. Let $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ be a Galois connection where $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$ are *posets*. We define two endo-functions $\mathcal{A} \xleftarrow{c} \mathcal{A}$ and

$\mathcal{B} \xleftarrow{i} \mathcal{B}$ as

$$c \overset{def}{=} g \circ f \tag{5.13}$$
$$i \overset{def}{=} f \circ g \tag{5.14}$$

Then, c is a closure operator on $\mathcal{A}$ and i is an interior operator on $\mathcal{B}$ [Priestley, 2000; Bělohlávek, 2000; Erné et al., 1993].

The set of *closed elements* of $\mathcal{A}$ is $\mathcal{A}_c \overset{def}{=} \{\ a \in \mathcal{A} :\ g\,(f\,a) = a\ :\ a\}$ and the set of *open elements* of $\mathcal{B}$ is $\mathcal{B}_i \overset{def}{=} \{\ b \in \mathcal{B} :\ f\,(g\,b) = b\ :\ b\}$. Posets $\mathcal{A}_c$ and $\mathcal{B}_i$ are isomorphic, the isomorphism functions being the lower and upper adjoint with the domain restricted to these sets, i.e., $f(\mathcal{A}_c) = \mathcal{B}_i$ and $g(\mathcal{B}_i) = \mathcal{A}_c$ [Erné et al., 1993; Priestley, 2000]. This means we have a new Galois connection,

$$\begin{tikzcd} \mathcal{A}_c \arrow[loop left, "="] \arrow[r, bend left, "f"] & \mathcal{B}_i \arrow[loop right, "="] \arrow[l, bend left, "g"] \end{tikzcd} \tag{5.15}$$

Closures from Galois connections. Conversely, every closure operator $\mathcal{A} \xleftarrow{c} \mathcal{A}$ is the composition of a lower adjoint $\mathcal{A}_c \xleftarrow{f} \mathcal{A}$ and an upper adjoint $\mathcal{A} \xleftarrow{g} \mathcal{A}_c$ of a Galois connection, i.e., $c \overset{def}{=} g \circ f$. The lower adjoint is defined as $f\,a \overset{def}{=} c\,a$, a mapping from the set $\mathcal{A}$ into its closure under c; the upper adjoint is just the trivial embedding $\mathcal{A} \xleftarrow{\rho_{\mathcal{A}}} \mathcal{A}_c$ of $\mathcal{A}_c$ into the set $\mathcal{A}$ [Priestley, 2000].

A similar result is valid for every interior operator $\mathcal{B} \xleftarrow{i} \mathcal{B}$, but the adjoints are inverted. The upper adjoint is defined as $g\,b \overset{def}{=} i\,a$ and the lower adjoint it the trivial embedding $\mathcal{B} \xleftarrow{\rho_{\mathcal{B}}} \mathcal{B}_i$ of $\mathcal{B}_i$ into the set $\mathcal{B}$.

5.3 Existence

In this section we will describe some results about necessary and sufficient conditions for the existence of a Galois connection. We start with a result for partially ordered sets which requires completeness, and then we transpose it to complete lattices. Finally, we discuss a more general result that holds in any partially ordered set.

Complete posets. Let $\mathcal{A}$ and $\mathcal{B}$ be *complete posets* and $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{A} \xleftarrow{g} \mathcal{B}$ be functions. Then [Backhouse, 2000],

1. f is a lower adjoint in a Galois connection if and only if f preserves existing suprema, i.e., for all subsets $\mathcal{A}' \subseteq \mathcal{A}$ if when $\bigsqcup_{\mathcal{A}} \mathcal{A}'$ exists then $\bigsqcup_{\mathcal{B}} f(\mathcal{A}')$ exists and $f(\bigsqcup_{\mathcal{A}} \mathcal{A}') = \bigsqcup_{\mathcal{B}} f(\mathcal{A}')$;

2. g is an upper adjoint in a Galois connection if and only if g preserves existing infima, i.e., for all subsets $\mathcal{B}' \subseteq \mathcal{B}$ if when $\bigsqcap_{\mathcal{B}} \mathcal{B}'$ exists then $\bigsqcap_{\mathcal{A}} g(\mathcal{B}')$ exists and $f(\bigsqcap_{\mathcal{B}} \mathcal{B}') = \bigsqcap_{\mathcal{A}} g(\mathcal{B}')$.

Complete lattices. This result can be modified if $\mathcal{A}$ and $\mathcal{B}$ are *complete lattices*. In a complete lattice, $\bigsqcup_{\mathcal{A}} A'$ and $\bigsqcap_{\mathcal{B}} \mathcal{B}'$ always exist, thus these existence conditions can be dropped [Priestley, 2000].

Posets. Melton et al. [1986, Theorem 2.6] present a necessary and sufficient condition for the existence of a Galois connection between *partially ordered sets* that drops the completeness requirement. This theorem is independent from the properties of functions and solely based on the properties of the underlying sets. We will not enunciate the theorem here but just describe its guidelines.

Let $\mathcal{A}$ and $\mathcal{B}$ be partially ordered sets. The theorem requires the existence of two isomorphic sets, $\mathcal{A}'$ and $\mathcal{B}'$, such that $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{B}' \subseteq \mathcal{B}$. $\mathcal{A}'$ must be a system of representatives for a partition of $\mathcal{A}$ such that each representative element is greater or equal to the represented elements; $\mathcal{B}'$ must be a system of representatives for a partition of $\mathcal{B}$ such that each representative element is smaller or equal to the represented elements. Additionally, the representative elements, both of $\mathcal{A}'$ and $\mathcal{B}'$, must maintain the original order of the represented elements.

5.4 Properties

From the definitions and existence conditions some of the properties of Galois connections can be inferred. In this section, we will explore some other important results.

Figure 5.1 summarizes the main properties of Galois connections.

Uniqueness. Uniqueness is an important property of Galois connections, that is, each adjoint uniquely determines the other [Priestley, 2000]. This means that Equations (5.6) and (5.7) explicitly show how a lower and an upper adjoint, respectively, are uniquely defined in terms of the other adjoint.

General properties

$f\,a \sqsubseteq_{\mathcal{B}} b \Leftrightarrow a \sqsubseteq_{\mathcal{A}} g\,b$	"Shunting rule"
$a \sqsubseteq_{\mathcal{A}} g\,(f\,a)$	Lower cancellation
$f\,(g\,b) \sqsubseteq_{\mathcal{B}} b$	Upper cancellation
$a \sqsubseteq_{\mathcal{A}} a' \Rightarrow f\,a \sqsubseteq_{\mathcal{B}} f\,a'$	Monotonicity
$b \sqsubseteq_{\mathcal{B}} b' \Rightarrow g\,b \sqsubseteq_{\mathcal{A}} g\,b'$	Monotonicity

Properties for partial orders

$f\,(g\,(f\,a)) = f\,a$	Semi-inverse
$g\,(f\,(g\,b)) = g\,b$	Semi-inverse

Distributivity properties (lattices)

$g\,(b \sqcap_{\mathcal{B}} b') = g\,b \sqcap_{\mathcal{A}} g\,b'$	Distributivity
$f\,(a \sqcup_{\mathcal{A}} a') = f\,a \sqcup_{\mathcal{B}} f\,a'$	Distributivity
$g\,\top_{\mathcal{B}} = \top_{\mathcal{A}}$	Top-preservation
$f\,\bot_{\mathcal{A}} = \bot_{\mathcal{B}}$	Bottom-preservation

Legend

f	Lower adjoint	g	Upper adjoint
$\top_{\mathcal{A}}$	Top element of $\mathcal{A}$, if it exists	$\bot_{\mathcal{A}}$	Bottom element of $\mathcal{A}$, if it exists
$\top_{\mathcal{B}}$	Top element of $\mathcal{B}$, if it exists	$\bot_{\mathcal{B}}$	Bottom element of $\mathcal{B}$, if it exists

$$a \sqcap_{\mathcal{A}} a' = a \quad \Leftrightarrow \quad a \sqsubseteq_{\mathcal{A}} a' \qquad\qquad b \sqcap_{\mathcal{B}} b' = b \quad \Leftrightarrow \quad b \sqsubseteq_{\mathcal{B}} b'$$
$$a \sqcup_{\mathcal{A}} a' = a' \quad \Leftrightarrow \quad a \sqsubseteq_{\mathcal{A}} a' \qquad\qquad b \sqcup_{\mathcal{B}} b' = b' \quad \Leftrightarrow \quad b \sqsubseteq_{\mathcal{B}} b'$$

for partial orders $\sqsubseteq_{\mathcal{A}}$ and $\sqsubseteq_{\mathcal{B}}$

Figure 5.1: Summary of the most important properties of Galois connections.

This uniqueness result is valid for a pair of adjoints in a certain preordered set. If we change the underlying order, different connections arise. As we shall see in the sections to follow, a lower adjoint of one connection can become the upper adjoint of another connection, or a polymorphic function can be the adjoint of several different operators depending of the underlying structure.

Preservation. From Ore's definition we already know that adjoints must be monotonic functions. But Galois connections have stronger preservation properties, as the existence result suggests. Thus, for Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$, f preserves existing suprema and g preserves existing infima [Priestley, 2000]. In particular, for the binary case this means that

$$\begin{aligned} f\,(a \sqcup_A a') &= f\,a \sqcup_B f\,a' \\ g\,(b \sqcap_B b') &= g\,b \sqcap_A g\,b' \end{aligned}$$

Moreover, if $\mathcal{A}$ and $\mathcal{B}$ have least and greatest elements, then

$$\begin{aligned} f\,\bot_{\mathcal{A}} &= \bot_{\mathcal{B}} \\ g\,\top_{\mathcal{B}} &= \top_{\mathcal{A}} \end{aligned}$$

Opposites. Let $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ be a Galois connection where $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$ are *posets*. The restriction of both adjoints to the images $g(\mathcal{B}) \subseteq \mathcal{A}$ and $f(\mathcal{A}) \subseteq \mathcal{B}$ is an isomorphism [Melton et al., 1994],

$$\begin{tikzcd} g(\mathcal{B}) \arrow[loop above, "="] \arrow[r, shift left, "f"] & f(\mathcal{A}) \arrow[loop above, "="] \arrow[l, shift left, "g"] \end{tikzcd}$$

This means that an element $a \in \mathcal{A}$ belongs to $g(\mathcal{B})$ if and only if $g\,(f\,a) = a$; and an element $b \in \mathcal{B}$ belongs to $f(\mathcal{A})$ if and only if $f\,(g\,b) = b$ [Melton et al., 1994]. Therefore, we can see that $g(\mathcal{B}) = \mathcal{A}_c$ and $f(\mathcal{A}) = \mathcal{B}_i$, respectively, the sets of closed elements of $\mathcal{A}$ and the set of open elements of $\mathcal{B}$, where closure and interior operators are defined as in (5.13) and (5.14).

The sets $f(\mathcal{A})$ and $g(\mathcal{B})$, besides being isomorphic, are also posets [Bělohlávek, 2001]. Moreover, if $\mathcal{A}$ and $\mathcal{B}$ are (complete) lattices, then so are $f(\mathcal{B})$ and $g(\mathcal{B})$ [Melton et al., 1994, 1986].

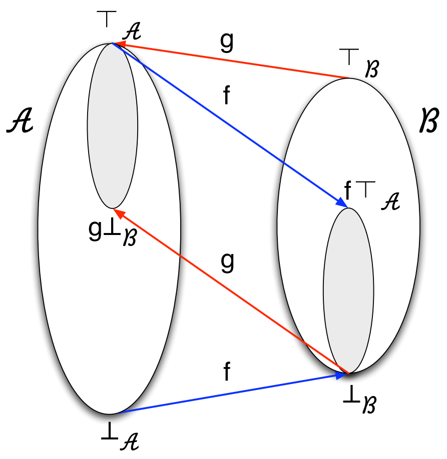


Figure 5.2: Opposites diagram.

Unity-of-opposites. The "unity-of-opposites" diagram (Figure 5.2) shows how sets are connected when both $\mathcal{A}$ and $\mathcal{B}$ have top and bottom elements. Backhouse [2004] explains the origin of this name which illustrates the behavior of Galois connections: the shaded sets are isomorphic (the unity) but $f(\mathcal{A})$ is a set of "small" elements and $g(\mathcal{B})$ is a set of "large" elements (the opposites). This is a consequence from the fact that one defines the lower adjoint as a minimum and the upper adjoint as a maximum. The diagram also helps to justify the choice of the names "lower" and "upper" for the adjoints.

Semi-inverses. From the unit-of-opposites property above, we know that if we have a Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$, then $f \circ g$ is the identity function on elements of the form $f\,a$, for all $a \in \mathcal{A}$; conversely, $g \circ f$ is the identity function on elements of the from $g\,b$, for all $b \in \mathcal{B}$ [Priestley, 2000]. Thus, we conclude the *semi-inverse* property of Galois connections: $f \circ g \circ f = f$ (i.e., $f\,(g\,(f\,a)) = f\,a$) and $g \circ f \circ g = g$ (i.e., $g\,(f\,(g\,b)) = g\,b$) already mentioned in Figure 5.1.

This property requires anti-symmetry, thus it does not hold for pre-orders. However, there is a weaker formulation for pre-orders: $f \mathrel{\dot{\preceq}} f \circ g \circ f \mathrel{\dot{\preceq}} f$ and $g \mathrel{\dot{\sqsubseteq}} g \circ f \circ g \mathrel{\dot{\sqsubseteq}} g$ [Erné et al., 1993] ($\dot{\sqsubseteq}$ and $\dot{\preceq}$ are lifted versions of pre-orders $\sqsubseteq$ and $\preceq$, respectively, cf. defined in Section 4.1).

Factorization. From a Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$, we know that the restriction of their adjoints to $f(\mathcal{A})$ and $g(\mathcal{B})$ yields an isomorphism between these sets.

Moreover, we know that $f(\mathcal{A}) = \mathcal{A}_c$ for closure operator $c \overset{def}{=} g \circ f$; and that $g(\mathcal{B}) = \mathcal{B}_i$ for interior operator $i \overset{def}{=} f \circ g$. Finally, we also know that every closure (interior) operator is a composition of two adjoints, where the lower (upper) adjoint is just the closure (interior) operator itself and the upper (lower) adjoint is an embedding function.

Therefore, $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ can be expressed as the composition of three Galois connections:

$$(\mathcal{A}, \sqsubseteq) \xleftarrow{(g \circ f, \rho_{\mathcal{A}})} (\mathcal{A}_c, \sqsubseteq), (\mathcal{A}_c, =) \xleftarrow{(f,g)} (\mathcal{B}_i, =) \text{ and } (\mathcal{B}_i, \preceq) \xleftarrow{(\rho_{\mathcal{B}}, f \circ g)} (\mathcal{B}, \preceq).$$

In fact, every Galois connection can be factored in this way [Erné et al., 1993].

Ordering. Suppose that $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f_1,g_1)} (\mathcal{B}, \preceq)$ and $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f_2,g_2)} (\mathcal{B}, \preceq)$ are Galois connections. Then we have that [von Karger, 2000]:

$$f_1 \mathrel{\dot{\preceq}} f_2 \quad \Leftrightarrow \quad g_2 \mathrel{\dot{\sqsubseteq}} g_1 \tag{5.16}$$

where $\dot{\preceq}$ and $\dot{\sqsubseteq}$ are the lifting of the underlying orders (recall the lifting of an order from Section 4.1).

Using these lifted orders we can define an order for Galois connections. Thus, we define an order relation $\trianglelefteq$ between Galois connections with the same domains, as

$$(\mathcal{A}, \sqsubseteq) \xleftarrow{(f_1,g_1)} (\mathcal{B}, \preceq) \;\trianglelefteq\; (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_2,g_2)} (\mathcal{B}, \preceq) \quad \text{if and only if} \quad f_1 \mathrel{\dot{\preceq}} f_2 \tag{5.17}$$

or, equivalently (by (5.16)), as

$$(\mathcal{A}, \sqsubseteq) \xleftarrow{(f_1,g_1)} (\mathcal{B}, \preceq) \;\trianglelefteq\; (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_2,g_2)} (\mathcal{B}, \preceq) \quad \text{if and only if} \quad g_2 \mathrel{\dot{\sqsubseteq}} g_1 \tag{5.18}$$

5.5 Building connections from relations

We already saw how to build a relation (pair algebra) from a Galois connection. However, in order to be a pair algebra the relation must obey some conditions. The constructions we will introduce below are generic and valid for *any* relation. They are always defined on sets of sets (power sets) and ordered by set inclusion (or reverse set inclusion).

Function image. Given two *sets* $\mathcal{A}$ and $\mathcal{B}$, and a function $\mathcal{B} \xleftarrow{h} \mathcal{A}$ we can build a new Galois connection $(\wp\mathcal{A}, \subseteq) \xleftarrow{(f,g)} (\wp\mathcal{B}, \subseteq)$ where the adjoints are defined as follows, for all $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{B}' \subseteq \mathcal{B}$:

$$f(\mathcal{A}') \overset{def}{=} \{\, b \in \mathcal{B} : \langle \exists\, a' : a' \in \mathcal{A}' : h\, a' = b \rangle : b \} \tag{5.19}$$

$$g(\mathcal{B}') \overset{def}{=} \{\, a \in \mathcal{A} : \langle \exists\, b' : b' \in \mathcal{B}' : h\, a = b' \rangle : a \} \tag{5.20}$$

f is called the *direct image* of $\mathcal{A}'$ under h, and g is called the *inverse image* of $\mathcal{B}'$ under h [Erné et al., 1993].

Moreover, if we define another function $\wp\mathcal{B} \xleftarrow{j} \wp\mathcal{A}$ as, for all $\mathcal{A}' \subseteq \mathcal{A}$,

$$j(\mathcal{A}') \overset{def}{=} \{\, b \in \mathcal{B} : g\,\{b\} \subseteq \mathcal{A}' : b \} \tag{5.21}$$

we obtain another Galois connection $(\wp\mathcal{B}, \subseteq) \xleftarrow{(g,j)} (\wp\mathcal{A}, \subseteq)$, where g is now the lower adjoint and j is the upper adjoint [Priestley, 2000].

Polarities. A *polarity* arises from a binary relation $\mathcal{B} \xleftarrow{R} \mathcal{A}$ defined on *sets* $\mathcal{A}$ and $\mathcal{B}$, and establishes a Galois connection $(\wp\mathcal{A}, \subseteq) \xleftarrow{(f,g)} (\wp\mathcal{B}, \supseteq)$ between the power set of $\mathcal{A}$ ordered by inclusion and the power set of $\mathcal{B}$ ordered by the "includes" order (or, equivalently, the dual of the power set of $\mathcal{B}$ ordered by inclusion).

We will use the notation of Priestley [2000] and denote the lower adjoint f by $^{\rhd}$ and the upper adjoint g by $^{\lhd}$. Thus, the adjoints are defined as, for all $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{B}' \subseteq \mathcal{B}$,

$$\mathcal{A}'^{\rhd} \overset{def}{=} \{\, b \in \mathcal{B} : \langle \forall\, a' \in \mathcal{A}' :: bRa' \rangle : b \} \tag{5.22}$$

$$\mathcal{B}'^{\lhd} \overset{def}{=} \{\, a \in \mathcal{A} : \langle \forall\, b' \in \mathcal{B}' :: b'Ra \rangle : a \} \tag{5.23}$$

and satisfy

$$\mathcal{A}'^{\rhd} \supseteq \mathcal{B}' \quad \Leftrightarrow \quad \mathcal{A}' \subseteq \mathcal{B}'^{\lhd} \tag{5.24}$$

Polarities correspond to the order-reversing version of Galois connections. In fact, the adjoints are monotonic in the dual order.

Polarities are important an important subject in their own because they are the basis of formal concept analysis [Ganter and Wille, 1999].

Axialities. Another way of obtaining a Galois connection from a relation is through *axialities* [Erné et al., 1993]. Given a relation $\mathcal{B} \xleftarrow{R} \mathcal{A}$ between *sets* $\mathcal{A}$ and $\mathcal{B}$, we build a Galois connection $(\wp\mathcal{A}, \subseteq) \xleftarrow{(f,g)} (\wp\mathcal{B}, \subseteq)$ where, for all $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{B}' \subseteq \mathcal{B}$,

$$f(\mathcal{A}') \overset{def}{=} \{\, b \in \mathcal{B} :\ \langle \exists\, a' \in \mathcal{A}' :: bRa' \rangle\ : b\} \tag{5.25}$$

$$g(\mathcal{B}') \overset{def}{=} \{\, a \in \mathcal{A} :\ \langle \forall\, b \in \mathcal{B} :\ bRa\ : b \in \mathcal{B}' \rangle\ : a\} \tag{5.26}$$

Unlike polarities, axialities are order preserving.

5.6 Building new connections from old

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. Let us see some of these combinators.

5.6.1 Basic combinators

Identity. The simplest of all Galois connections is the identity,

$$\begin{tikzcd}
\mathcal{A} \arrow[loop left, "\sqsubseteq_{\mathcal{A}}"] \arrow[r, bend left, "id"] & \mathcal{A} \arrow[l, bend left, "id"] \arrow[loop right, "\sqsubseteq_{\mathcal{A}}"]
\end{tikzcd}$$

where adjoints are instances of the polymorphic identity function id mentioned in Section 2.4.1.

Composition. Two Galois connections $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ and $(\mathcal{B}, \preceq) \xleftarrow{(h,k)} (\mathcal{C}, \leqslant)$ with matching *preorders* can be composed, forming Galois connection

$$\begin{tikzcd}
\mathcal{A} \arrow[loop left, "\sqsubseteq"] \arrow[r, bend left, "h \circ f"] & \mathcal{C} \arrow[l, bend left, "g \circ k"] \arrow[loop right, "\leqslant"]
\end{tikzcd}$$

(Note how adjoints compose in reverse order.) Composition is an associative operation and the identity Galois connection is its unit. Thus, Galois connections form a monoid structure.

Isomorphism. The particular case in which both orders are equalities boils down to both adjoints being isomorphisms (bijections), i.e.,

$$f\,a = b \quad \Leftrightarrow \quad a = g\,b \tag{5.27}$$

which we can write as

$$\begin{tikzcd} \mathcal{A} \arrow[loop left, "="] \arrow[r, bend left, "f"] & \mathcal{B} \arrow[loop right, "="] \arrow[l, bend left, "g"] \end{tikzcd}$$

This means that both functions f and g are simultaneously lower and upper adjoints. They are also inverses of each other, that is, $g = f^{\cup}$ and $f = g^{\cup}$.

In proofs by indirect equality, we use partial orders and not equalities. However, equality can be trivially replaced in (5.27) by any reflexive order, thus forming equivalent Galois connections. If $\sqsubseteq_{\mathcal{A}}$ and $\preceq_{\mathcal{B}}$ are reflexive orders defined on sets $\mathcal{A}$ and $\mathcal{B}$, respectively, among several possible combinations, for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$,

$$\begin{array}{lcl} f\,a \preceq_{\mathcal{B}} b & \Leftrightarrow & a \sqsubseteq_{\mathcal{A}} g\,b \\ g\,b \sqsubseteq_{\mathcal{A}} a & \Leftrightarrow & b \preceq_{\mathcal{B}} f\,a \end{array}$$

are Galois connections.

Converse. The converse combinator on Galois connections switches adjoints while inverting the orders. That is, from $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ one builds the converse connection $(\mathcal{B}, \succeq) \xleftarrow{(g,f)} (\mathcal{A}, \sqsupseteq)$.

5.6.2 Relators

Relators. Every relator $\mathcal{F}$ preserves Galois connections between *preordered sets*. Therefore, from $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ one infers, for every such relator, a new Galois connection $(\mathcal{F}\,\mathcal{A}, \mathcal{F} \sqsubseteq) \xleftarrow{(\mathcal{F}\,f, \mathcal{F}\,g)} (\mathcal{F}\,\mathcal{B}, \mathcal{F} \preceq)$.

When $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$ are *posets* instead, relator $\mathcal{F}$ is required to distribute through binary intersections, i.e., $\mathcal{F}(R \cap S) = \mathcal{F}\,R \cap \mathcal{F}\,S$, in order to preserve Galois connections [Backhouse and Backhouse, 2004]. The distributivity through binary intersection implies that $(\mathcal{F}\,\mathcal{A}, \mathcal{F} \sqsubseteq)$ and $(\mathcal{F}\,\mathcal{B}, \mathcal{F} \preceq)$ are also partial orders. If this condition does not hold for a relator $\mathcal{F}$, a weaker Galois connection is built, since

$(\mathcal{F}\,\mathcal{A}, \mathcal{F}\sqsubseteq)$ and $(\mathcal{F}\,\mathcal{B}, \mathcal{F}\preceq)$ are just preorders. Appendix A provides the proofs of these preservation properties of relators.

These results extend to binary relators such as, for instance, the *product* $\mathcal{A} \times \mathcal{B}$ which pairs elements of $\mathcal{A}$ with elements of $\mathcal{B}$ ordered by the pairwise orderings.

Forks. Let $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f_1, g_1)} (\mathcal{B}, \preceq)$ and $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f_2, g_2)} (\mathcal{C}, \leqslant)$ be Galois connections. Is it possible to build a new Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f, g)} (\mathcal{B} \times \mathcal{C}, \preceq \times \leqslant)$? A immediate candidate to the lower adjoint is $f \overset{def}{=} f_1 \bigtriangledown f_2$ where $\bigtriangledown$ is the fork combinator, i.e., $(f_1 \bigtriangledown f_2)\, a \overset{def}{=} (f_1\, a, f_2\, a)$. The definition of the upper adjoint is more difficult. Let us calculate it, for all $a \in \mathcal{A}$, $b \in \mathcal{B}$ and $c \in \mathcal{C}$,

Proof

$$
\begin{array}{cl}
 & (f_1 \bigtriangledown f_2)\, a \quad \preceq \times \leqslant \quad (b, c) \\
\Leftrightarrow & \qquad \{ \text{ Definition of } \bigtriangledown. \} \\
 & (f_1\, a, f_2\, a) \quad \preceq \times \leqslant \quad (b, c) \\
\Leftrightarrow & \qquad \{ \text{ Definition of } (\preceq \times \leqslant). \} \\
 & f_1\, a \preceq b \;\wedge\; f_2\, a \leqslant c \\
\Leftrightarrow & \qquad \{ \text{ Shunting. } \} \\
 & a \sqsubseteq g_1\, b \;\wedge\; a \sqsubseteq g_2\, c \\
\Leftrightarrow & \qquad \{ \text{ Universal property of } \sqcap. \} \\
 & a \sqsubseteq g_1\, b \sqcap g_2\, c
\end{array}
$$

□

Thus,

$$
f\, a \overset{def}{=} (f_1 \bigtriangledown f_2)\, a \tag{5.28}
$$

$$
g\,(b, c) \overset{def}{=} g_1\, b \sqcap g_2\, c \tag{5.29}
$$

In fact, this is the composition of the Galois connection obtained using the product relator $(\mathcal{A} \times \mathcal{A}, \sqsubseteq \times \sqsubseteq) \xleftarrow{(f_1 \times f_2, g_1 \times g_2)} (\mathcal{B} \times \mathcal{C}, \preceq \times \leqslant)$, and the infimum Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{(\triangle, \sqcap)} (\mathcal{A} \times \mathcal{A}, \sqsubseteq \times \sqsubseteq)$ where $\triangle$ is the doubling function: $\triangle\, a \overset{def}{=} (a, a)$ (i.e., $\triangle \overset{def}{=} id \bigtriangledown id$).

5.6.3 Function spaces

Post-composition. Galois connections can be lifted to function spaces, i.e., adjoints map functions to functions ordered by a lifted order relation.

Then, $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ is a Galois connection if and only if for all functions h and k according to the diagram,

$$\begin{tikzcd}
\mathcal{A} \arrow[loop left, "\sqsubseteq"] \arrow[rr, bend left=10, "f"] & & \mathcal{B} \arrow[loop right, "\preceq"] \arrow[ll, bend left=10, "g"] \\
& \mathcal{C} \arrow[ul, "h"] \arrow[ur, "k"'] &
\end{tikzcd}$$

the following holds,

$$f \circ h \mathrel{\dot{\preceq}} k \quad \Leftrightarrow \quad h \mathrel{\dot{\sqsubseteq}} g \circ k \tag{5.30}$$

In other words, $(\mathcal{A} \leftarrow \mathcal{C}, \dot{\sqsubseteq}) \xleftarrow{((f\circ),(g\circ))} (\mathcal{B} \leftarrow \mathcal{C}, \dot{\preceq})$ is a Galois connection if and only if $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ is a Galois connection. [Backhouse, 2000].

Pre-composition. If $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ is a Galois connection then, for all *monotonic* functions h and k according with the diagram,

$$\begin{tikzcd}
\mathcal{A} \arrow[loop left, "\sqsubseteq"] \arrow[rr, bend left=10, "f"] \arrow[dr, "h"] & & \mathcal{B} \arrow[loop right, "\preceq"] \arrow[ll, bend left=10, "g"] \arrow[dl, "k"'] \\
& \mathcal{C} \arrow[loop below, "\leqslant"] &
\end{tikzcd}$$

the following holds,

$$h \circ g \mathrel{\dot{\leqslant}} k \quad \Leftrightarrow \quad h \mathrel{\dot{\leqslant}} k \circ f \tag{5.31}$$

Thus, $(\mathcal{C} \leftarrow \mathcal{A}, \dot{\leqslant}) \xleftarrow{((\circ g),(\circ f))} (\mathcal{C} \leftarrow \mathcal{B}, \dot{\leqslant})$ is a Galois connection if $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ is a Galois connection, provided that $(\mathcal{C} \leftarrow \mathcal{A}, \dot{\leqslant})$ and $(\mathcal{C} \leftarrow \mathcal{B}, \dot{\leqslant})$ are sets of monotonic functions [Backhouse, 2000].

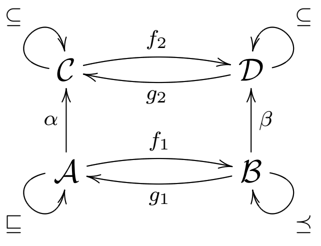
Lifting. By composing the previous constructions of Galois connections on function spaces, we get a new construction that is very important for building analyses in abstract interpretation [Cousot, 1999; Hankin, 2005].

From a Galois connection $(\mathcal{A},\sqsubseteq) \xleftarrow{(f_1,g_1)} (\mathcal{B},\preceq)$, using (5.31), we build another connection $(\mathcal{C} \leftarrow \mathcal{A}, \dot{\leqslant}) \xleftarrow{((\circ g_1),(\circ f_1))} (\mathcal{C} \leftarrow \mathcal{B}, \dot{\leqslant})$. The choice of $\mathcal{C}$ as co-domain is not accidental; this allow us to take another Galois connection $(\mathcal{C},\leqslant) \xleftarrow{(f_2,g_2)} (\mathcal{D},\subseteq)$ and, using (5.30), to build $(\mathcal{C} \leftarrow \mathcal{B}, \dot{\leqslant}) \xleftarrow{((f_2\circ),(g_2\circ))} (\mathcal{D} \leftarrow \mathcal{B}, \dot{\subseteq})$. In this way we can compose them, obtaining $(\mathcal{C} \leftarrow \mathcal{A}, \dot{\leqslant}) \xleftarrow{((f_2\circ)\circ(\circ g_1),(\circ f_1)\circ(g_2\circ))} (\mathcal{D} \leftarrow \mathcal{B}, \dot{\subseteq})$. By defining $f \stackrel{def}{=} (f_2\circ)\circ(\circ g_1)$ and $g \stackrel{def}{=} (\circ f_1)\circ(g_2\circ)$, this means that, for every *monotonic* functions $\mathcal{C} \xleftarrow{\alpha} \mathcal{A}$ and $\mathcal{D} \xleftarrow{\beta} \mathcal{B}$,

$$f\,\alpha \stackrel{def}{=} f_2 \circ \alpha \circ g_1 \tag{5.32}$$

$$g\,\beta \stackrel{def}{=} g_2 \circ \beta \circ f_1 \tag{5.33}$$

according to diagram:


$$\begin{tikzcd}
\mathcal{C} \arrow[loop left, "\subseteq"] \arrow[r, bend left, "f_2"] & \mathcal{D} \arrow[loop right, "\subseteq"] \arrow[l, bend left, "g_2"] \\
\mathcal{A} \arrow[u, "\alpha"] \arrow[loop left, "\sqsubseteq"'] \arrow[r, bend left, "f_1"] & \mathcal{B} \arrow[u, "\beta"'] \arrow[loop right, "\preceq"] \arrow[l, bend left, "g_1"]
\end{tikzcd}$$

5.6.4 Homomorphic image

Endo-functions. From a *preordered set* $(\mathcal{A},\sqsubseteq)$ and a function $\mathcal{A} \xleftarrow{h} \mathcal{B}$, a new preorder $\preceq$ on $\mathcal{B}$ can be defined as

$$\preceq \;\stackrel{def}{=}\; h^{\cup} \circ \sqsubseteq \circ h \tag{5.34}$$

as explained in Oliveira [2005]. Function h is a preorder homomorphism which lifts results from the $\sqsubseteq$-order to the $\preceq$-order. The homomorphism can be written as

$$h \circ \preceq \;=\; \sqsubseteq \circ h$$

corresponding to the point-free version of

$$a \preceq a' \quad \Leftrightarrow \quad h\,a \sqsubseteq h\,a'$$

Using this construction we can build new Galois connections from existing ones. First, we analyze the case in which adjoints are endo-functions.

Let $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{A}, \sqsubseteq)$ be a Galois connection in which $\sqsubseteq$ is a *preorder* and let $\mathcal{A} \xleftarrow{h} \mathcal{B}$ be a function. Suppose that $\mathcal{B} \xleftarrow{f'} \mathcal{B}$ and $\mathcal{B} \xleftarrow{g'} \mathcal{B}$ are h-homomorphic functions to lower adjoint f and upper adjoint g, respectively, i.e.:

$$h \circ f' \;=\; f \circ h \tag{5.35}$$

$$h \circ g' \;=\; g \circ h \tag{5.36}$$

Then, $(\mathcal{B}, \preceq) \xleftarrow{(f',g')} (\mathcal{B}, \preceq)$ is a Galois connection [Oliveira, 2005], according to diagram:

$$\begin{tikzcd}
\mathcal{A} \arrow[r, bend left, "f"] \arrow[loop left, "\sqsubseteq"] & \mathcal{A} \arrow[l, bend left, "g"] \arrow[loop right, "\sqsubseteq"] \\
\mathcal{B} \arrow[u, "h"] \arrow[r, bend left, "f'"] \arrow[loop left, "\preceq"] & \mathcal{B} \arrow[u, "h"'] \arrow[l, bend left, "g'"] \arrow[loop right, "\preceq"]
\end{tikzcd}$$

General case. In order to generalize to Galois connection in which adjoints are not limited to endo-functions, we need two homomorphic functions h and h', such as the diagram which follows:

$$\begin{tikzcd}
\mathcal{A} \arrow[r, bend left, "f"] \arrow[loop left, "\sqsubseteq"] & \mathcal{B} \arrow[l, bend left, "g"] \arrow[loop right, "\preceq"] \\
\mathcal{C} \arrow[u, "h"] \arrow[r, bend left, "f'"] \arrow[loop left, "\leqslant"] & \mathcal{D} \arrow[u, "h'"'] \arrow[l, bend left, "g'"] \arrow[loop right, "\subseteq"]
\end{tikzcd}$$

where the following equations must hold:

$$\leqslant \;\stackrel{\mathit{def}}{=}\; h^{\cup} \circ \sqsubseteq \circ h \tag{5.37}$$

$$\subseteq \;\stackrel{\mathit{def}}{=}\; h'^{\cup} \circ \preceq \circ h' \tag{5.38}$$

$$h' \circ f' = f \circ h \tag{5.39}$$
$$h \circ g' = g \circ h' \tag{5.40}$$

Then, as we prove in Appendix A, $(\mathcal{C}, \leqslant) \xleftarrow{(f',g')} (\mathcal{D}, \subseteq)$ is a Galois connection.

Partial orders. The previous construction is only valid, in general, for preorders. If we are dealing with *partial orders* the following question is important: in which conditions the derived order $\preceq$ as defined in (5.34) is a partial order, provided that anti-symmetry holds for $\sqsubseteq$?

As we prove in Appendix A, a function preserves anti-symmetry if it is injective. Therefore, we can use (5.34) to build partial orders provided that h is injective. Consequently, we can also use the previous homomorphic construction to lift existing Galois connections between posets.

5.6.5 Higher-order Galois connections

Backhouse and Backhouse [2004] provide an expressive account of how Galois connections and free-theorem about polymorphic functions can be combined to build new *higher-order Galois connections*, and how this provides for safe abstract interpretations. Here, we will just provide some of the results concerning the construction of new Galois connections.

Type expressions. Backhouse and Backhouse [2004] define the grammar of type expressions t as

$$\begin{array}{lll} t ::= & v & \text{Type variable} \\ \quad\;\; \mid & t'' \leftarrow t' & \text{Function type} \\ \quad\;\; \mid & \mathcal{F}(t_1, \ldots t_n) & n\text{-ary type relator} \end{array}$$

The use of type variables v allows for the use of parametric polymorphism. Basic types correspond to 0-ary type relators.

Assignments. The key of building higher-order Galois connections is the use of an assignment that extends individual variable assignments to type expressions.

Backhouse and Backhouse [2004] define a *variable assignment* as a function with domain the set of type variables, and divide it in two kinds, depending on the range. One kind assigns *posets* and the other assigns *relations on posets* to type variables. Thus, if V is a variable assignment and Σ_v is the set of type variables, $\mathit{poset} \xleftarrow{V} \Sigma_v$ is an assignment of the first kind and $\mathit{relations\ on\ posets} \xleftarrow{V} \Sigma_v$ is an assignment of the second kind.

Given two variable assignment V and W of the same kind, we can extend the assignment to arbitrary type expressions t by defining the following operator:

$$\begin{array}{rcl} [V,W]_v & \overset{def}{=} & V_v \\ [V,W]_{t''\leftarrow t'} & \overset{def}{=} & [V,W]_{t''} \leftarrow [W,V]_{t'} \\ [V,W]_{\mathcal{F}(t_1,\ldots,t_n)} & \overset{def}{=} & \mathcal{F}([V,W]_{t_1},\ldots,[V,W]_{t_n}) \end{array}$$

Higher-order Galois connections. Let $(\mathcal{A}_v, \sqsubseteq_v)$ and $(\mathcal{B}_v, \preceq_v)$ be *posets*. If we have a Galois connection $(\mathcal{A}_v, \sqsubseteq_v) \xleftarrow{(f_v,g_v)} (\mathcal{B}_v, \preceq_v)$ for each type variable v, then, for all type expressions t,

$$[f, g^{\cup}]_t^{\cup} \circ \preceq_t \quad = \quad \sqsubseteq_t \circ [g, f^{\cup}]_t \tag{5.41}$$

i.e., $(\mathcal{A}_t, \sqsubseteq_t) \xleftarrow{([f,g^{\cup}]_t,[g,f^{\cup}]_t)} (\mathcal{B}_t, \preceq_t)$ is a Galois connection, for each type expression t. This result follows from the instantiation of the variable assignment with a pair algebra induced by a Galois connection, for each type variable [Backhouse and Backhouse, 2004].

5.6.6 Algebra of Galois connections

The combinators we have presented may be seen as operators of an algebra. Thus, we have defined an algebra of Galois connections.

Definition. An *algebra of Galois connections* is a tuple $(\mathcal{G}, \circ, id, ^{\cup})$ satisfying the following axioms, for any $g, h, j \in \mathcal{G}$:

$$g \circ (h \circ j) \quad = \quad (g \circ h) \circ j \tag{5.42}$$

$$g \circ id = \quad g \quad = id \circ g \tag{5.43}$$

$$(g \circ h)^{\cup} = h^{\cup} \circ g^{\cup} \tag{5.44}$$

$$(g^{\cup})^{\cup} = g \tag{5.45}$$

This structure is just a composition monoid $(\mathcal{G}, \circ, id)$ with a converse operation.

Interpretation. Galois connections are models for algebras of Galois connections. The validity of these axioms when the elements of $\mathcal{G}$ are interpreted as Galois connections can be easily proved using the definitions.

5.7 Galois connections and categories

Galois connections relate themselves with categories in two ways: they are instances of adjunctions between categories of ordered sets; and they themselves form a category. Here, we briefly discuss this relation.

5.7.1 Galois connections as adjunctions

Adjunctions [Priestley, 2000]. An *adjunction* between categories $\mathtt{A}$ and $\mathtt{B}$ is a structure $(\mathcal{F}, \mathcal{G}, e, \epsilon)$, such that $\mathtt{B} \xleftarrow{\mathcal{F}} \mathtt{A}$ and $\mathtt{A} \xleftarrow{\mathcal{G}} \mathtt{B}$ are monotonic functors; for all $A \in \mathtt{A}$ and $B \in \mathtt{B}$, the arrows $\mathcal{GF}(A) \xleftarrow{e_A} A$ and $\mathcal{FG}(B) \xleftarrow{\epsilon_B} B$ exist; and the following conditions hold:

1. For arrows $A' \xleftarrow{u} A$ and $B' \xleftarrow{\varphi} B$, where $A, A' \in \mathtt{A}$ and $B, B' \in \mathtt{B}$ the following diagrams commute

$$\begin{tikzcd} A \arrow[r, "u"] \arrow[d, "e_A"'] & A' \arrow[d, "e_{A'}"] \\ \mathcal{GF}(A) \arrow[r, "\mathcal{GF}(u)"] & \mathcal{GF}(A') \end{tikzcd} \qquad \begin{tikzcd} B \arrow[r, "\varphi"] & B' \\ \mathcal{FG}(B) \arrow[u, "\epsilon_B"] \arrow[r, "\mathcal{FG}(\varphi)"] & \mathcal{FG}(B') \arrow[u, "\epsilon_{B'}"'] \end{tikzcd}$$

2. For $A \in \mathtt{A}$ and $B \in \mathtt{B}$, u and φ are associated in such a way that the following diagrams commute

$$\begin{tikzcd} A \arrow[r, "e_A"] \arrow[dr, "u"'] & \mathcal{GF}(A) \arrow[d, "\mathcal{G}(\varphi)"] \\ & \mathcal{G}(B) \end{tikzcd} \qquad \begin{tikzcd} B \arrow[r, "\epsilon_B"] \arrow[dr, "\varphi"'] & \mathcal{FG}(B) \arrow[d, "\mathcal{F}(u)"] \\ & \mathcal{F}(A) \end{tikzcd}$$

Interpretation. When in the above definition, categories `A` and `B` are categories of ordered sets, the adjunction is interpreted as a Galois connection, where the functors $\mathcal{F}$ and $\mathcal{G}$ are, respectively, the lower and upper adjoints, and u and φ are the orders [Priestley, 2000].

However, developing the theory on Galois connections in the categorical setting is much more complex than using order theory. Furthermore, not all properties valid for Galois connections hold for adjunctions. As Aarts et al. [1992] put it: *"To call a Galois connection an adjunction is just mathematical overkill!"*

5.7.2 Category of Galois connections

Galois connections form their own category if we take the following definitions:

Objects. Preordered sets or posets.

Arrows (morphisms). Galois connections between the preordered sets. The source is the domain of the lower adjoint and the target is the domain of the lower adjoint.

Identity arrow. The identity Galois connection.

Composition. The composition of Galois connections which is an associative operation and has the identity Galois connection as unit.

Functors. Functors in the Galois connections category are relators from other categories. When the objects are posets, relators must distribute over binary meet.

5.7.3 Galois connections and allegories

This sections explores how a category in which objects are *partial orders* and arrows are Galois connections can be extended to the more general concept of an allegory.

Inclusion. We define the inclusion order as the ordering of Galois connections $\trianglelefteq$. We must show that the composition of Galois connections is monotonic with respect to this order. Thus, if we have that

$$\alpha_1 = (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_1,g_1)} (\mathcal{B}, \preceq) \quad \trianglelefteq \quad \alpha_2 = (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_2,g_2)} (\mathcal{B}, \preceq)$$

and

$$\beta_1 = (\mathcal{B}, \preceq) \xleftarrow{(h_1,j_1)} (\mathcal{C}, \leqslant) \quad \trianglelefteq \quad \beta_2 = (\mathcal{B}, \preceq) \xleftarrow{(h_2,j_2)} (\mathcal{C}, \leqslant)$$

both hold, then

$$\alpha_1 \circ \beta_1 \quad \trianglelefteq \quad \alpha_2 \circ \beta_2$$

must hold either.

By definition of composition of Galois connections and of $\trianglelefteq$, this is equivalent to prove that

$$h_1 \circ f_1 \quad \dot{\leqslant} \quad h_2 \circ f_2$$

under assumptions $f_1 \mathrel{\dot{\preceq}} f_2$ and $h_1 \mathrel{\dot{\sqsubseteq}} h_2$.

Proof

$$\begin{array}{ll}
 & h_1 \circ f_1 \quad \dot{\leqslant} \quad h_2 \circ f_2 \\
\Leftrightarrow & \qquad \{ \text{ Definition of lifted order (4.56). } \} \\
 & h_1 \circ f_1 \quad \subseteq \quad \leqslant \circ\, h_2 \circ f_2 \\
\Leftarrow & \qquad \{ \text{ Transitivity of } \leqslant. \ \} \\
 & h_1 \circ f_1 \quad \subseteq \quad \leqslant \circ \leqslant \circ\, h_2 \circ f_2 \\
\Leftarrow & \qquad \{ \text{ Monotonicity of an adjoint. } \} \\
 & h_1 \circ f_1 \quad \subseteq \quad \leqslant \circ\, h_2 \circ \preceq \circ f_2 \\
\Leftarrow & \qquad \{ \text{ Assumption and definition of lifted order (4.56). } \} \\
 & h_1 \quad \subseteq \quad \leqslant \circ\, h_2 \\
\Leftrightarrow & \qquad \{ \text{ Assumption and definition of lifted order (4.56). } \} \\
 & \top
\end{array}$$

□

Meet. We define the meet operation $\Cap$ as, for all Galois connections g_1 and g_2,

$$g_1 \Cap g_2 = g_1 \quad \overset{def}{\Leftrightarrow} \quad g_1 \trianglelefteq g_2 \tag{5.46}$$

Therefore, we must show that, for all Galois connections x, g_1 and g_2, the universal property of meet holds (4.60), i.e.,

$$x \trianglelefteq g_1 \Cap g_2 \quad \Leftrightarrow \quad x \trianglelefteq g_1 \wedge x \trianglelefteq g_2 \tag{5.47}$$

Proof

$$\begin{array}{ll}
 & x \trianglelefteq g_1 \Cap g_2 \\
\Leftrightarrow & \qquad \{\ \text{Definition (5.46).}\ \} \\
 & x \Cap (g_1 \Cap g_2) = x \\
\Leftrightarrow & \qquad \{\ \text{Case analysis: } g_1 \Cap g_2 = g_1 \text{ and } g_1 \Cap g_2 = g_2.\ \} \\
 & x \Cap g_1 = x \wedge x \Cap g_2 = x \\
\Leftrightarrow & \qquad \{\ \text{Definition (5.46).}\ \} \\
 & x \trianglelefteq g_1 \wedge x \trianglelefteq g_2
\end{array}$$

□

Converse. We define the converse $^{\cup}$ as the standard converse operator for Galois connections. From the algebra of Galois connections we already know that it is an involution and that contravariance holds. Thus, we only must prove that converse is monotonic with respect to $\trianglelefteq$, i.e., for $\alpha_1 = (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_1,g_1)} (\mathcal{B}, \preceq)$ and $\alpha_2 = (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_2,g_2)} (\mathcal{B}, \preceq)$,

$$\alpha_1 \trianglelefteq \alpha_2 \quad \Leftrightarrow \quad \alpha_1^{\cup} \trianglelefteq \alpha_2^{\cup}$$

which by definition of converse and $\trianglelefteq$ is equivalent to prove that,

$$f_1 \mathrel{\dot{\preceq}} f_2 \quad \Leftrightarrow \quad g_1 \mathrel{\dot{\sqsupseteq}} g_2$$

Proof

$$\begin{array}{ll}
 & f_1 \mathrel{\dot{\preceq}} f_2 \\
\Leftrightarrow & \qquad \{\ \text{Equivalence (5.16).}\ \} \\
 & g_2 \mathrel{\dot{\sqsubseteq}} g_1
\end{array}$$

$\Leftrightarrow$ { Definition of converse. }

$g_1 \mathrel{\dot{\sqsupseteq}} g_2$

□

Modular law. Finally, all the introduced operations on Galois connections must satisfy the modular law (4.64), i.e.,

$$(\alpha_1 \circ \alpha_2) \Cap \alpha_3 \quad \trianglelefteq \quad \alpha_1 \circ (\alpha_2 \Cap (\alpha_1^{\cup} \circ \alpha_3)) \tag{5.48}$$

The proof follows, for all Galois connections x,

Proof

$x \trianglelefteq (\alpha_1 \circ \alpha_2) \Cap \alpha_3$

$\Leftrightarrow$ { Universal property of meet (5.47). }

$x \trianglelefteq \alpha_1 \circ \alpha_2 \ \wedge \ x \trianglelefteq \alpha_3$

$\Rightarrow$ { Monotonicity of composition with respect to inclusion. }

$x \trianglelefteq \alpha_1 \circ \alpha_2 \ \wedge \ \alpha_1^{\cup} \circ x \trianglelefteq \alpha_1^{\cup} \circ \alpha_3$

$\Leftrightarrow$ { Assuming $\alpha_1^{\cup} \circ \alpha_2 \trianglelefteq \alpha_3 \Leftrightarrow \alpha_2 \trianglelefteq \alpha_1 \circ \alpha_3$. }

$\alpha_1^{\cup} \circ x \trianglelefteq \alpha_2 \ \wedge \ \alpha_1^{\cup} \circ x \trianglelefteq \alpha_1^{\cup} \circ \alpha_3$

$\Leftrightarrow$ { Universal property of meet (5.47). }

$\alpha_1^{\cup} \circ x \trianglelefteq \alpha_2 \Cap (\alpha_1^{\cup} \circ \alpha_3)$

$\Leftrightarrow$ { Assuming again $\alpha_1^{\cup} \circ \alpha_2 \trianglelefteq \alpha_3 \Leftrightarrow \alpha_2 \trianglelefteq \alpha_1 \circ \alpha_3$. }

$x \trianglelefteq \alpha_1 \circ (\alpha_2 \Cap (\alpha_1^{\cup} \circ \alpha_3))$

$\therefore$ { By indirect inequality (see Section 8.3). }

$(\alpha_1 \circ \alpha_2) \Cap \alpha_3 \quad \trianglelefteq \quad \alpha_1 \circ (\alpha_2 \Cap (\alpha_1^{\cup} \circ \alpha_3))$

□

In the proof above, a kind of “shunting” property of Galois connections was assumed:

$$\alpha_1^{\cup} \circ \alpha_2 \trianglelefteq \alpha_3 \quad \Leftrightarrow \quad \alpha_2 \trianglelefteq \alpha_1 \circ \alpha_3 \tag{5.49}$$

The proof of this theorem is deferred to Appendix A.4. However, since we assume that objects are partial orders, (5.49) only holds in the particular case when α_1 is an isomorphism. Therefore, the modular law (5.48) only holds for Galois connections when α_1 is an isomorphism. Appendix A.4 discusses the typing issues that do not allow to further generalise (5.49). A similar argument can be applied to (5.48).

Thus, Galois connections do not satisfy all the axioms of an allegory because the modular identity does not hold, in general. However, the particular case when arrows are isomorphisms forms an allegory.

5.8 Summary

Galois connections are the principal concept behind the design of the *Galculator*. This chapter provided a theoretical introduction to the subject, together with a short historical overview. As usually happens with ubiquitous and important concepts in Mathematics, various distinct definitions of Galois connections exist, depending of the field of application. This chapter gave several of the different but equivalent definitions which appear in the literature.

Galois connections enjoy several interesting and important properties which were analyzed. One advantage of Galois connections is that once a concept is identified as an adjoint, it automatically enjoys all the general properties of Galois connections. This will be exploited in the design of the *Galculator* as a way of improving genericity.

Also of special importance on the design of the *Galculator* is the ability of building new Galois connections from existing ones. This allowed us to build an algebra of Galois connections which will be used by *Galculator*. Moreover, as we have shown, it is possible to define a category of Galois connections. The extension of this result to allegories is also discussed.

In the following chapter, several examples of Galois connections spreading through several different fields will be described. This will hopefully show their wide range of application, as well as the usefulness of their algebraic nature and properties.

Chapter 6

Examples and applications of Galois connections

This chapter provides an overview of some important examples of Galois connections and their applications. The list is not exhaustive but it should give a good insight about the ubiquity of Galois connections and their relevance in software design. More examples can be found in the literature about abstract interpretation [Cousot and Cousot, 1977; Cousot, 2001; Hankin, 2005; Backhouse and Backhouse, 2004] or in references such as [Backhouse et al., 2002; Backhouse, 2004; Erné et al., 1993; Denecke et al., 2004; Melton et al., 1986].

6.1 Abstract interpretation

Abstract interpretation [Cousot and Cousot, 1977] is, perhaps, the most well-known application of Galois connections in Computer Science. Below we provide an overview of the theory of abstract interpretation, mostly based on Cousot [1999] and Hankin [2005].

6.1.1 Description

A known limitation of static automatic approaches to program correctness is that many interesting properties are undecidable. However, we can often "forget" some details and approximate the original problem by a simpler one. Moreover, we can repeat this process until we reach a solvable problem. As Cousot [2001] puts it: *"The purpose of* *abstract interpretation is to formalize this notion of approximation in a unified framework.”*.

Approximation entails loss of information. The question is: which information should one ignore? Usually, the abstractions used in abstract interpretation over-approximate the concrete property and in this sense they are *safe*. That is, if the abstract property holds then the corresponding concrete property holds; if the abstract property does not hold we cannot conclude nothing about the concrete property. In this way, false positives are not allowed but false negatives can occur.

In a more general setting, abstract interpretation can be seen *“. . . as a theory for approximating sets and set operations as considered in set (or category) theory. . . ”* [Cousot, 2001]. Galois connections arise naturally as a device for building abstractions in a constructive way.

Concrete properties. General analysis must be valid for any program in a given programming language. Thus, we must use the semantics of the language, which is usually defined over a set $\mathcal{V}$ of values (states, traces, etc.). The semantics of a program specifies how it transforms the values of $\mathcal{V}$.

A *concrete property* P is the set of all values of $\mathcal{V}$ that satisfy the property, thus, $P \in \wp\mathcal{V}$.

Abstract properties. In order to approximate the domain of concrete properties, we use another (more abstract) domain $\mathcal{L}$. An *abstract property* $p \in \mathcal{L}$ corresponds to an approximation of concrete property $P \in \wp\mathcal{V}$.

Usually, the set $\mathcal{L}$ is a complete lattice $(\mathcal{L}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $\sqsubseteq$ is a partial order (referred to as the approximation order) which corresponds to abstract implication, $\bot$ is the false value, $\top$ is the true value, $\sqcup$ is the abstract disjunction and $\sqcap$ is the abstract conjunction [Cousot, 1999].

Concretization functions. Abstract properties are related to concrete properties by a monotonic *concretization function* $\wp\mathcal{V} \xleftarrow{\text{con}} \mathcal{L}$ such that for each $p \in \mathcal{L}$ it returns the corresponding $P \in \wp\mathcal{V}$. The monotonicity condition implies that, for all abstract properties $p, q \in \mathcal{L}$,

$$p \sqsubseteq q \quad \Rightarrow \quad \operatorname{con} p \subseteq \operatorname{con} q$$

In general, there is not an equivalent abstract property p of a concrete property P. Therefore, in order to be safe, p must over-approximate P, i.e., $P \subseteq \mathrm{con}\, p$, meaning that $\mathrm{con}\, p$ is weaker than P.

Abstraction functions. Is there an optimal over-approximation p of a concrete property P? Intuitively, an over-approximation is optimal if it is the smallest one which includes P, i.e., $\{\bigcap p : \ P \subseteq \mathrm{con}\, p \ : \mathrm{con}\, p\}$. This situation corresponds to an *abstraction function* $\mathcal{L} \xleftarrow{\mathrm{abs}} \wp\mathcal{V}$ which maps every $P \in \wp\mathcal{V}$ to its best over-approximation, $\mathrm{abs}\, P \in \mathcal{L}$.

In this way, abs and con are defined as for all $P \in \wp\mathcal{V}$ and $p \in \mathcal{L}$,

$$\mathrm{abs}\, P \ \overset{def}{=} \ \langle \bigsqcap p : \ P \subseteq \mathrm{con}\, p \ : p \rangle \tag{6.1}$$

$$\mathrm{con}\, p \ \overset{def}{=} \ \{\bigcup P : \ \mathrm{abs}\, P \sqsubseteq p \ : P\} \tag{6.2}$$

which corresponds to the explicit definition of a Galois connection. Thus, abs and con form a Galois connection,

$$\begin{tikzcd} \wp\mathcal{V} \arrow[loop left, "\subseteq"] \arrow[r, bend left, "\mathrm{abs}"] & \mathcal{L} \arrow[loop right, "\sqsubseteq"] \arrow[l, bend left, "\mathrm{con}"] \end{tikzcd}$$

such that, for all $P \in \wp\mathcal{V}$ and $p \in \mathcal{L}$,

$$\mathrm{abs}\, P \sqsubseteq p \quad \Leftrightarrow \quad P \subseteq \mathrm{con}\, p \tag{6.3}$$

From the properties of Galois connections, abs and con immediately satisfy the following important properties [Cousot, 1999], for all $P, Q \in \wp\mathcal{V}$ and $p \in \mathcal{L}$,

$P \subseteq Q \quad \Rightarrow \quad \mathrm{abs}\, P \sqsubseteq \mathrm{abs}\, Q$	abs preserves implication
$P \quad \subseteq \quad \mathrm{con}(\mathrm{abs}\, P)$	$\mathrm{abs}\, P$ over-approximates P
$\mathrm{abs}(\mathrm{con}\, p) \quad \sqsubseteq \quad p$	con controls the loss of information

When abs and con form a perfect Galois connection, the last property becomes

$$\mathrm{abs}(\mathrm{con}\, p) \quad = \quad p$$

meaning that no information is lost.

Representations and extractions. It is often the case that instead of an abstraction function we have a representation function $\mathcal{L} \xleftarrow{\beta} \mathcal{V}$, according to diagram

$$\begin{tikzcd}
\wp\mathcal{V} \arrow[rr, bend left, "\mathrm{abs}"] & & \mathcal{L} \arrow[ll, bend left, "\mathrm{con}"] \\
& \mathcal{V} \arrow[ul, "\{\cdot\}"] \arrow[ur, "\beta"'] &
\end{tikzcd}$$

We can define the abstraction and concretization functions from the representation function β, for all $\mathcal{V}' \subseteq \mathcal{V}$ and $l \in \mathcal{L}$,

$$\begin{aligned}
\mathrm{abs}\,\mathcal{V}' &\overset{def}{=} \langle \bigsqcup p \in \mathcal{V}' :: \beta\, p \rangle \\
\mathrm{con}\, l &\overset{def}{=} \{\, p \in \mathcal{V} :\ \beta\, p \sqsubseteq l\ : p \}
\end{aligned}$$

Another common situation occurs when $\mathcal{L} = (\wp\mathcal{A}, \subseteq)$ and we have an extraction function $\mathcal{A} \xleftarrow{\eta} \mathcal{V}$. Then, the abstraction and concretization functions are defined as, for all $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{V}' \subseteq \mathcal{V}$,

$$\begin{aligned}
\mathrm{abs}\,\mathcal{V}' &\overset{def}{=} \{\, v \in \mathcal{V}' :: \eta\, v \} \\
\mathrm{con}\,\mathcal{A}' &\overset{def}{=} \{\, a :\ \eta\, a \in \mathcal{A}'\ : a \}
\end{aligned}$$

6.1.2 Building new analysis

Even after abstracting from concrete properties $\wp\mathcal{V}$ to abstract properties $\mathcal{L}$, some properties remain undecidable or require too complex computations. This means that we must further abstract our domain $\mathcal{L}$ and/or combine it with other abstractions of $\wp\mathcal{V}$. We will call an *analysis* to the process of abstracting from a certain domain (Cousot refers to this as abstraction).

An important element of abstract interpretation is the possibility of building complex and powerful analysis by composing simpler ones. The literature about the field presents several constructions that preserve the best over-approximation properties of the basis analysis. In fact, it all boils down to combinational properties of Galois connections and the constructions we have described in Section 5.6.

Composition of analysis. Composition allows for the construction of analysis of increased abstraction provided that there exists a pair of abstraction and concretization functions between domains. This means that from a set of concrete properties $\mathcal{L}_0 = \wp\mathcal{V}$ it is possible to build an analysis on $\mathcal{L}_n$, if there exists $(\mathrm{abs}_i, \mathrm{con}_i)$ for $0 \leqslant i \leqslant n$, i.e.,

$$\begin{tikzcd}
\wp\mathcal{V} \arrow[r, shift left, "\mathrm{abs}_1"] & \mathcal{L}_1 \arrow[l, shift left, "\mathrm{con}_1"] \arrow[r, shift left, "\mathrm{abs}_2"] & \mathcal{L}_2 \arrow[l, shift left, "\mathrm{con}_2"] \arrow[r, shift left, "\mathrm{abs}_3"] & \cdots \arrow[l, shift left, "\mathrm{con}_3"] \arrow[r, shift left, "\mathrm{abs}_n"] & \mathcal{L}_n \arrow[l, shift left, "\mathrm{con}_n"]
\end{tikzcd}$$

This is a direct consequence of the fact that the composition of Galois connections is still a Galois connection.

Independent attribute method. The *independent attribute method* [Hankin, 2005] combines two analysis

$$\begin{tikzcd}
\mathcal{L}_1 \arrow[loop above, "\sqsubseteq_1"] \arrow[r, shift left, "\mathrm{abs}_1"] & \mathcal{M}_1 \arrow[loop above, "\preceq_1"] \arrow[l, shift left, "\mathrm{con}_1"] & \mathcal{L}_2 \arrow[loop above, "\sqsubseteq_2"] \arrow[r, shift left, "\mathrm{abs}_2"] & \mathcal{M}_2 \arrow[loop above, "\preceq_2"] \arrow[l, shift left, "\mathrm{con}_2"]
\end{tikzcd}$$

in just one defined over pairs of values

$$\begin{tikzcd}
\mathcal{L}_1 \times \mathcal{L}_2 \arrow[loop above, "\sqsubseteq_1 \times \sqsubseteq_2"] \arrow[r, shift left, "\mathrm{abs}_1 \times \mathrm{abs}_2"] & \mathcal{M}_1 \times \mathcal{M}_2 \arrow[loop above, "\preceq_1 \times \preceq_2"] \arrow[l, shift left, "\mathrm{con}_1 \times \mathrm{con}_2"]
\end{tikzcd}$$

where any possible relation between the pairs is lost. In fact, this is just the construction of Galois connections using the product relator (recall Sections 4.6 and 5.6).

Relational method. The *relational method* [Hankin, 2005] combines two analysis

$$\begin{tikzcd}
\wp\mathcal{A}_1 \arrow[loop above, "\subseteq"] \arrow[r, shift left, "\mathrm{abs}_1"] & \wp\mathcal{B}_1 \arrow[loop above, "\subseteq"] \arrow[l, shift left, "\mathrm{con}_1"] & \wp\mathcal{A}_2 \arrow[loop above, "\subseteq"] \arrow[r, shift left, "\mathrm{abs}_2"] & \wp\mathcal{B}_2 \arrow[loop above, "\subseteq"] \arrow[l, shift left, "\mathrm{con}_2"]
\end{tikzcd}$$

in just one

$$\begin{tikzcd}
\wp(\mathcal{A}_1 \times \mathcal{A}_2) \arrow[loop above, "\subseteq \times \subseteq"] \arrow[r, shift left, "\mathrm{abs}_{12}"] & \wp(\mathcal{B}_1 \times \mathcal{B}_2) \arrow[loop above, "\subseteq \times \subseteq"] \arrow[l, shift left, "\mathrm{con}_{12}"]
\end{tikzcd}$$

This is the composition of the independent attribute method, i.e., Galois connection $(\wp(\mathcal{A}_1) \times \wp(\mathcal{A}_2), \subseteq \times \subseteq) \xleftarrow{(\mathrm{abs}_1 \times \mathrm{abs}_2, \mathrm{con}_1 \times \mathrm{con}_2)} (\wp(\mathcal{B}_1) \times \wp(\mathcal{B}_2), \subseteq \times \subseteq)$, and the set relator construction we saw in Section 5.6.

The relation method preserves some information about the relation between values leading to more precise analysis than the independent attribute method.

Direct product method. The *direct product method* [Hankin, 2005] combines two analysis

$$\begin{tikzcd}
\mathcal{L} \arrow[loop above, "\sqsubseteq"] \arrow[r, bend left, "\text{abs}_1"] & \mathcal{M}_1 \arrow[loop above, "\preceq_1"] \arrow[l, bend left, "\text{con}_1"] & \mathcal{L} \arrow[loop above, "\sqsubseteq"] \arrow[r, bend left, "\text{abs}_2"] & \mathcal{M}_2 \arrow[loop above, "\preceq_2"] \arrow[l, bend left, "\text{con}_2"]
\end{tikzcd}$$

in just one sharing the same domain

$$\begin{tikzcd}
\mathcal{L} \arrow[loop above, "\sqsubseteq"] \arrow[r, bend left, "\text{abs}_{12}"] & \mathcal{M}_1 \times \mathcal{M}_2 \arrow[loop above, "\preceq_1 \times \preceq_2"] \arrow[l, bend left, "\text{con}_{12}"]
\end{tikzcd}$$

In fact, this is just the fork construction of Galois connections.

Direct tensor product method. The *direct tensor product method* [Hankin, 2005] combines two analysis like the direct product method but saves some relation between values like the relational method. In the case of two analysis defined over the power set relator

$$\begin{tikzcd}
\wp\mathcal{L} \arrow[loop above, "\subseteq"] \arrow[r, bend left, "\text{abs}_1"] & \wp\mathcal{M}_1 \arrow[loop above, "\subseteq"] \arrow[l, bend left, "\text{con}_1"] & \wp\mathcal{L} \arrow[loop above, "\subseteq"] \arrow[r, bend left, "\text{abs}_2"] & \wp\mathcal{M}_2 \arrow[loop above, "\subseteq"] \arrow[l, bend left, "\text{con}_2"]
\end{tikzcd}$$

they form a single analysis

$$\begin{tikzcd}
\wp\mathcal{L} \arrow[loop above, "\subseteq"] \arrow[r, bend left, "\text{abs}_{12}"] & \wp(\mathcal{M}_1 \times \mathcal{M}_2) \arrow[loop above, "\subseteq \times \subseteq"] \arrow[l, bend left, "\text{con}_{12}"]
\end{tikzcd}$$

We can see this method as the composition of the fork construction with the set relator construction of Galois connections.

Monotone function space method. The *monotone function space method* allows for approximating monotonic functions in abstract domains. Given two analyses,

$$\begin{tikzcd} \mathcal{L}_1 \arrow[loop above, "\sqsubseteq_1"] \arrow[r, bend left, "\mathrm{abs}_1"] & \mathcal{M}_1 \arrow[loop above, "\preceq_1"] \arrow[l, bend left, "\mathrm{con}_1"] & \mathcal{L}_2 \arrow[loop above, "\sqsubseteq_2"] \arrow[r, bend left, "\mathrm{abs}_2"] & \mathcal{M}_2 \arrow[loop above, "\preceq_2"] \arrow[l, bend left, "\mathrm{con}_2"] \end{tikzcd}$$

we can build another analysis, for all monotonic functions ϕ and φ

$$\begin{tikzcd} \mathcal{L}_1 \arrow[loop above, "\sqsubseteq_1"] \arrow[r, "\mathrm{abs}_1"] & \mathcal{M}_1 \arrow[loop above, "\preceq_1"] & \mathcal{L}_1 \arrow[loop above, "\sqsubseteq_1"] & \mathcal{M}_1 \arrow[loop above, "\preceq_1"] \arrow[l, "\mathrm{con}_1"'] \\ \mathcal{L}_2 \arrow[loop below, "\sqsubseteq_2"] \arrow[u, "\phi"] & \mathcal{M}_2 \arrow[loop below, "\preceq_2"] \arrow[l, "\mathrm{con}_2"] \arrow[u, "\mathrm{abs}_{12}(\phi)"'] & \mathcal{L}_2 \arrow[loop below, "\sqsubseteq_2"] \arrow[u, "\mathrm{con}_{12}(\varphi)"] \arrow[r, "\mathrm{abs}_2"'] & \mathcal{M}_2 \arrow[loop below, "\preceq_2"] \arrow[u, "\varphi"'] \end{tikzcd}$$

where abs_{12} and con_{12} are defined as

$$\begin{aligned} \mathrm{abs}_{12}\,\phi &\overset{\mathit{def}}{=} \mathrm{abs}_1 \circ \phi \circ \mathrm{con}_2 \\ \mathrm{con}_{12}\,\varphi &\overset{\mathit{def}}{=} \mathrm{con}_1 \circ \varphi \circ \mathrm{abs}_2 \end{aligned}$$

If we have a monotonic function $\mathcal{L}_1 \xleftarrow{\phi} \mathcal{L}_2$, then $\mathrm{abs}_{12}\,\phi$ is the best over-approximation of a (more abstract) function of type $\mathcal{M}_1 \leftarrow \mathcal{M}_2$. This corresponds to the lifted construction of Galois connections of Section 5.6.

6.1.3 Examples of analysis

Below we present just a few examples of analysis used in abstract interpretation, more specifically in Array Bound Analysis. This is a technique for statically determining whether an array index does not fall behind the limits (bounds) of the given array. These concrete examples are adapted from [Hankin, 2005] which shows how they can be combined to form a complete analysis.

Sign analysis. The *sign analysis* is a simple analysis on integers which only retains the information about their sign. From the extraction function defined as, for all $z \in \mathbb{Z}$,

$$\operatorname{sign} z \overset{def}{=} \begin{cases} - & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ + & \text{if } z > 0 \end{cases}$$

we get the following abstraction and concretization function, for all $\mathcal{Z} \subseteq \mathbb{Z}$ and $\mathcal{S} \subseteq \{-, 0, +\}$,

$$\begin{aligned} \operatorname{abs} \mathcal{Z} &\overset{def}{=} \{\, z \in \mathcal{Z} :: \operatorname{sign} z\} \\ \operatorname{con} \mathcal{S} &\overset{def}{=} \{\, z \in \mathbb{Z} : \ \operatorname{sign} z \in \mathcal{S} \ : z\} \end{aligned}$$

Thus, $(\wp\mathbb{Z}, \subseteq) \xleftarrow{(\text{abs,con})} (\wp\{-, 0, +\}, \subseteq)$ is a Galois connection.

Range analysis. The *range analysis* is identical to the sign analysis but more *precise* since more information is retained (the unit). The extraction function is, for all $z \in \mathbb{Z}$,

$$\operatorname{range} z \overset{def}{=} \begin{cases} <-1 & \text{if } z < -1 \\ -1 & \text{if } z = -1 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z = 1 \\ >+1 & \text{if } z > 1 \end{cases}$$

and the abstraction and concretization functions, defined as, for all $\mathcal{Z} \subseteq \mathbb{Z}$ and $\mathcal{R} \subseteq \{<-1, -1, 0, +1, >+1\}$,

$$\begin{aligned} \operatorname{abs} \mathcal{Z} &\overset{def}{=} \{\, z \in \mathcal{Z} :: \operatorname{range} z\} \\ \operatorname{con} \mathcal{R} &\overset{def}{=} \{\, z \in \mathbb{Z} : \ \operatorname{range} z \in \mathcal{R} \ : z\} \end{aligned}$$

form a Galois connection $(\wp\mathbb{Z}, \subseteq) \xleftarrow{(\text{abs,con})} (\wp\{<-1, -1, 0, +1, >+1\}, \subseteq)$.

Magnitude difference analysis. The *magnitude difference analysis* abstracts a pair of integers by the difference of their absolute values, according with the extraction function

$$\text{diff}(z_1, z_2) = |z_1| - |z_2|$$

defined for all $z_1, z_2 \in \mathbb{Z}$.

The abstraction and concretization functions defined as, for all $\mathcal{P} \subseteq \mathbb{Z} \times \mathbb{Z}$ and $\mathcal{Z} \subseteq \mathbb{Z}$,

$$\begin{aligned} \text{abs}\,\mathcal{P} &\overset{def}{=} \{\, z_1, z_2 :\ (z_1, z_2) \in \mathcal{P}\ : \text{diff}(z_1, z_2)\} \\ \text{con}\,\mathcal{Z} &\overset{def}{=} \{\, z_1, z_2 :\ \text{diff}(z_1, z_2) \in \mathcal{Z}\ : (z_1, z_2)\} \end{aligned}$$

form the $(\wp\mathbb{Z} \times \mathbb{Z}, \subseteq) \xleftarrow{(\text{abs},\text{con})} (\wp\mathbb{Z}, \subseteq)$ Galois connection.

Interval analysis. *Interval analysis* is a traditional analysis in which the complete lattice of intervals is used to represent potentially infinite domains of integers. Interval analysis can be used for Array Bound Analysis or for determining the sign of addition [Cousot, 1999; Hankin, 2005].

The set of integers is extended with top and bottom elements $\mathbb{Z} \cup \{-\infty, \infty\}$. An interval is either empty, $\perp$, or it is given by its limits

$$[z_1, z_2] \overset{def}{=} \{\, z \in \mathbb{Z} :\ z_1 \leqslant z \leqslant z_2\ : z\}$$

such that $z_1 \leqslant z_2$. The infimum and supremum operators of an interval i are defined as:

$$\inf i = \begin{cases} \infty & \text{if } i = \perp \\ z_1 & \text{if } i = [z_1, z_2] \end{cases} \qquad \sup i = \begin{cases} -\infty & \text{if } i = \perp \\ z_2 & \text{if } i = [z_1, z_2] \end{cases}$$

The inclusion order in intervals is then defined as, for all intervals i_1 and i_2, $i_1 \sqsubseteq_i i_2 \overset{def}{=} \inf i_2 \leqslant \inf i_1 \wedge \sup i_1 \leqslant \sup i_2$. Order $\sqsubseteq_i$ is a partial order since it is defined as a conjunction of two partial orders (ordering $\leqslant$ on integers).

From this we define the concretization and abstraction function which follows, for all intervals i and $\mathcal{Z} \subseteq \mathbb{Z}$,

$$\text{con}\, i \overset{def}{=} \{\, z \in \mathbb{Z} :\ \inf i \leqslant z \leqslant \sup i\ : z\}$$

$$\text{abs}\,\mathcal{Z} \quad \overset{def}{=} \quad \begin{cases} \bot & \text{if } \mathcal{Z} = \emptyset \\ [\inf z, \sup z] & \text{otherwise} \end{cases}$$

form a perfect Galois connection [Hankin, 2005]. $(\wp\mathbb{Z}, \subseteq) \xleftarrow{(\text{abs},\text{con})} (\mathsf{Int}, \sqsubseteq_i)$, where Int is the set of all intervals over $\mathbb{Z}$.

6.2 Formal concept analysis

Description. *Formal concept analysis* (FCA) [Ganter and Wille, 1999] is another important application of Galois connections. FCA intents to analyse the hierarchical structure and dependencies among objects and their attributes [Priestley, 2000]. Objects and attributes are, in this context, mathematical formal concepts as discussed by Priss [2006b].

One important feature of FCA is that it can be automatized. Several applications exist that extract the so-called *concept lattice*. Moreover, concept lattices can be visualized graphically using *Hass diagrams*, also known in FCA as *line diagrams*.

The applications of FCA spread over several fields: linguistics, software engineering, psychology, sociology, artificial intelligence, information retrieval, date mining and others [Priss, 2006b].

Ganter and Wille [1999] provide the mathematical foundations of FCA; [Priss, 2006b] is a more gentle introduction to the field which summarizes the applications and relevant bibliography about FCA.

Contexts. Let triple (G, M, R) denote a *context* where G is the set of *objects*, M is the set of *attributes* and $\mathcal{G} \xleftarrow{R} \mathcal{M}$ is a relation. $(g, m) \in R$ (or just gRm) means that *'object g has attribute m'* [Priestley, 2000].

Two functions, $^{\triangleright}$ and $^{\triangleleft}$, called, respectively, left and right polar, are defined as follows, for all $\mathcal{A} \subseteq G$ and $\mathcal{B} \subseteq M$,

$$\mathcal{A}^{\triangleright} \overset{def}{=} \{\, m \in \mathcal{M} : \langle \forall\, g \in \mathcal{A} :: gRm \rangle \ : m \}$$
$$\mathcal{B}^{\triangleleft} \overset{def}{=} \{\, g \in \mathcal{G} : \langle \forall\, m \in \mathcal{B} :: gRm \rangle \ : g \}$$

$^{\triangleright}$ and $^{\triangleleft}$ verify

$$\mathcal{A}^{\triangleright} \supseteq \mathcal{B} \quad \Leftrightarrow \quad \mathcal{A} \subseteq \mathcal{B}^{\triangleleft}$$

This means that $(\wp\mathcal{G}, \subseteq) \xleftarrow{(^{\triangleright}, ^{\triangleleft})} (\wp\mathcal{M}, \supseteq)$ is a Galois connection, corresponding to the polarity definitions of Section 5.5.

These functions have an intuitive meaning: $\mathcal{A}^{\triangleright}$ returns the set of attributes common to all objects in $\mathcal{A}$; $\mathcal{B}^{\triangleleft}$ returns the set of all objects which share all the atttibutes in $\mathcal{B}$ [Priestley, 2000].

Concepts. The pair $(\mathcal{A}, \mathcal{B})$, such that $\mathcal{A} \subseteq G$ and $\mathcal{B} \subseteq M$, is a *concept* if

$$\mathcal{A} = \mathcal{B}^{\triangleleft} \qquad \text{and} \qquad \mathcal{A}^{\triangleright} = \mathcal{B}$$

The set of all concepts ordered by $\subseteq \times \supseteq$ is referred to as a *concept lattice*.

6.3 Residuation

Residuation theory studies the interaction between ordered structures like lattices and semigroup operations. In this section we will not use the traditional definition of residuation in terms of preservation properties of principal ideals. Instead, we use Galois connections to specify the relation between the underlying order structure and the semigroup operators. Our intent is to show that some structures can be elegantly defined by requiring that a certain operator has an adjoint.

Residuated semigroups [Erné et al., 1993]. A *partially ordered semigroup* is a *poset* $(\mathcal{A}, \sqsubseteq)$ with a monotonic associative operation, $\otimes$. The partially ordered semigroup is said to be *residuated* if both right and left sections of $\otimes$ have upper adjoints, that is, $(\mathcal{A}, \sqsubseteq) \xleftarrow{((a\otimes),(a\backslash))} (\mathcal{A}, \sqsubseteq)$ and $(\mathcal{A}, \sqsubseteq) \xleftarrow{((\otimes a),(/a))} (\mathcal{A}, \sqsubseteq)$ are Galois connections. It turns out that another Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{((\backslash a),(a/))} (\mathcal{A}, \sqsupseteq)$ arises from the combination of these two connections as can be easily inferred from the definitions, for all a, b and $c \in \mathcal{A}$:

$$\begin{aligned} a \otimes b \sqsubseteq c \quad &\Leftrightarrow \quad b \sqsubseteq a \setminus c && (6.4)\\ a \otimes b \sqsubseteq c \quad &\Leftrightarrow \quad a \sqsubseteq c \,/\, b && (6.5)\\ a \setminus c \sqsupseteq b \quad &\Leftrightarrow \quad a \sqsubseteq c \,/\, b && (6.6) \end{aligned}$$

Sections $(a\backslash)$ and $(/a)$ are called, respectively, right and left division.

Complete residuated semigroups [Erné et al., 1993]. A complete partially ordered semigroup, also known as a *quantale*, is said to be *residuated* if and only if $\otimes$ distributes over arbitrary suprema, i.e., for all $a \in \mathcal{A}$ and $\mathcal{A}' \subseteq \mathcal{A}$, the following both hold,

$$\begin{aligned} a \otimes \bigsqcup \mathcal{A}' \quad &= \quad \langle \bigsqcup a' \in \mathcal{A}' :: a \otimes a' \rangle \\ \bigsqcup \mathcal{A}' \otimes a \quad &= \quad \langle \bigsqcup a' \in \mathcal{A}' :: a' \otimes a \rangle \end{aligned}$$

Commutative residuated semigroups. If the operation $\oplus$ is commutative left and right division coincide (see proof in Appendix A) and may be both denoted by $/$ as it is usual in arithmetics. Thus, we have the following Galois connections:

$$\begin{aligned} a \otimes b \sqsubseteq c \quad &\Leftrightarrow \quad b \sqsubseteq c \,/\, a \\ a \otimes b \sqsubseteq c \quad &\Leftrightarrow \quad a \sqsubseteq c \,/\, b \\ c \,/\, a \sqsupseteq b \quad &\Leftrightarrow \quad a \sqsubseteq c \,/\, b \end{aligned}$$

Whole division (dealt with in Section 1.1) is an example of this since multiplication is commutative.

Regular algebras. A *regular algebra* is a tuple $(\mathcal{L}, \leqslant, +, 0, \times, 1)$ such that $(\mathcal{L}, \leqslant, +, 0)$ is a complete, completely distributive lattice with least element 0; $(\mathcal{L}, \times, 1)$ is a monoid; and, $(a\times)$ and $(\times a)$ are lower adjoints, for all $a \in \mathcal{L}$ [Backhouse, 2004]. Thus, a regular algebra is an instance of a residuated semigroup.

Regular algebras are important in Computer Science because the monoid structure models composition and the complete distributive lattice models choice. Additionally, by requiring the existence of a certain fixed point, regular algebras also model the iteration of composition. Backhouse [2004] presents several interesting examples that are instances of regular algebras such as graphs, vectors and optimization problems.

Languages. Backhouse [2004] presents the concept of a language as a regular algebra.

Let $\mathcal{T}$ be a finite set of symbols called *alphabet*. Finite sequences of elements of $\mathcal{T}$ are called *words*; the *empty word* (length zero) is denoted by ϵ. There is a binary associative operator $\cdot$ called *concatenation* which forms new words by joining existing words. The empty word is the unit of concatenation, i.e., for all words w, $w \cdot \epsilon = w = \epsilon \cdot w$. Therefore, $(\mathcal{T}^*, \cdot, \epsilon)$ forms a monoid structure; $\mathcal{T}^*$ denotes the set of all words.

A *language* is a subset of $\mathcal{T}^*$; thus, the set $\mathcal{L}$ of all languages is just the power set of $\mathcal{T}^*$ and, consequently, a complete lattice ordered by set inclusion. Backhouse [2004] extends the monoid structure of $\mathcal{T}^*$ to the lattice $\mathcal{L}$ by defining the concatenation operation of words and languages, for all words w and languages $\mathcal{M}$,

$$w \cdot \mathcal{M} \quad \overset{def}{=} \quad \{\bigcup m : \ m \in \mathcal{M} \ : w \cdot m\}$$

and the concatenation of languages, for all languages $\mathcal{M}$ and $\mathcal{N}$,

$$\mathcal{M} \cdot \mathcal{N} \quad \overset{def}{=} \quad \{\bigcup m : \ m \in \mathcal{M} \ : m \cdot \mathcal{N}\}$$

The concatenation operator is residuated and thus verifies, for all languages $\mathcal{L}$, $\mathcal{M}$ and $\mathcal{N}$,

$$\begin{aligned} \mathcal{L} \cdot \mathcal{M} \subseteq \mathcal{N} \quad &\Leftrightarrow \quad \mathcal{M} \subseteq \mathcal{L} \setminus \mathcal{N} \\ \mathcal{L} \cdot \mathcal{M} \subseteq \mathcal{N} \quad &\Leftrightarrow \quad \mathcal{L} \subseteq \mathcal{N} \,/\, \mathcal{M} \\ \mathcal{N} \cdot \mathcal{M} \supseteq \mathcal{L} \quad &\Leftrightarrow \quad \mathcal{M} \subseteq \mathcal{L} \setminus \mathcal{N} \end{aligned}$$

Thus, $\mathcal{L} \setminus \mathcal{N}$ is the greatest language $\mathcal{M}$ such that $\mathcal{L} \cdot \mathcal{M} \subseteq \mathcal{N}$, and $\mathcal{N} \,/\, \mathcal{M}$ is the greatest language $\mathcal{L}$ such that $\mathcal{L} \cdot \mathcal{M} \subseteq \mathcal{N}$. More intuitively, $\mathcal{L} \setminus \mathcal{N}$ is the greatest language that when concatenated *after* $\mathcal{L}$ (suffix), the resulting language is still smaller or equal to language $\mathcal{N}$. In the same way, $\mathcal{N} \,/\, \mathcal{M}$ is the greatest language that when concatenated *before* $\mathcal{M}$ (prefix), the resulting language is still smaller or equal to language $\mathcal{N}$.

Backhouse [2004] presents another interesting example of a Galois connection concerning the word concatenation operator. For all words w and languages $\mathcal{L}$ and

$\mathcal{M}$,

$$w \cdot \mathcal{L} \subseteq \mathcal{M} \quad \Leftrightarrow \quad \mathcal{L} \subseteq \partial_w \mathcal{M}$$

where $\partial_w \mathcal{M}$ is called the w-derivative of $\mathcal{M}$ and is defined by

$$x \in \partial_w \mathcal{M} \quad \Leftrightarrow \quad w \cdot x \in \mathcal{M}$$

Boolean algebras. In von Karger [2000] a Boolean algebra is defined as a lattice $(\mathcal{B}, \leqslant, \wedge, \vee)$ with an operator $\neg$, such that it satisfies the following law, for all p, q and $r \in \mathcal{B}$,

$$p \wedge q \leqslant r \quad \Leftrightarrow \quad q \leqslant \neg p \vee r \tag{6.7}$$

i.e., $(\mathcal{B}, \leqslant) \xleftarrow{((p\wedge),(\neg p\vee))} (\mathcal{B}, \leqslant)$ is a Galois connection.

von Karger [2000] presents proofs based on the properties of Galois connections that the lattice has least and greatest elements, with their respective definition, that the lattice is distributive, and that double negation and contraposition both hold. He also proves the De Morgan's rules and the complement rule.

The duality principle ensures that instead of (6.7), the Galois connection

$$\neg p \wedge q \leqslant r \quad \Leftrightarrow \quad q \leqslant p \vee r$$

could be used.

Heyting algebras. Heyting algebras are generalizations of Booleans algebras. Like Boolean algebras model propositional logic, Heyting algebras model intuitionistic propositional logic, a kind of logic in which the law of excluded middle does not hold, in general [Priestley, 2000].

A *Heyting algebra* is a lattice $(\mathcal{B}, \leqslant, \wedge, \vee, \mathsf{true}, \mathsf{false})$ with greatest (true) and least (false) elements, and an operator $\Rightarrow$ such that, for every p, q and $r \in \mathcal{B}$, the following holds

$$p \wedge q \leqslant r \quad \Leftrightarrow \quad q \leqslant (p \Rightarrow r)$$

i.e., $(\mathcal{B}, \leqslant) \xleftarrow{((p\wedge),(p\Rightarrow))} (\mathcal{B}, \leqslant)$ is a Galois connection.

This definition is very similar to the definition of Boolean algebras above, mostly because in classical logic, $\neg p \vee q \;\Leftrightarrow\; p \Rightarrow q$. However, in intuitionistic logic this equivalence is weakened to $\neg p \vee q \;\leqslant\; p \Rightarrow q$ and negation is defined as $\neg p \overset{def}{=} p \Rightarrow \mathsf{false}$ where $\neg p$ is called a *pseudo-complement*. A Heyting algebra is a Boolean algebra in the particular case that, for all $p \in \mathcal{B}$, fact $p \Rightarrow \mathsf{false} \;\leqslant\; \mathsf{false} = p$ holds.

Residuated lattices. A *residuated lattice* [Ward and Dilworth, 1939] is a commutative residuated semigroup, with least (false) and greatest element (true) and an identity element (id) of the residuated operation. Therefore, $(\mathcal{L}, \wedge, \vee, \mathsf{false}, \mathsf{true})$ is a bounded lattice, $(\mathcal{L}, \otimes, id)$ is a commutative monoid and the left and right section of $\otimes$ are lower adjoints in a Galois connection [Bělohlávek, 2001].

Complete residuated lattices are used in fuzzy set theory to represent truth values. Thus, the "classical" relation $\mathcal{B} \xleftarrow{R} \mathcal{A}$ becomes a *fuzzy relation* $\mathcal{L} \xleftarrow{R} \mathcal{B} \times \mathcal{A}$, i.e., a function from pairs $(b, a) \in B \times A$ to values $l \in \mathcal{L}$ representing the grade of membership of (b, a) in relation R. Thus, the classical case occurs when $\mathcal{L} = \mathbb{B}$.

Bělohlávek [2000] uses fuzzy relations and fuzzy sets to define the concept of *fuzzy Galois connections* (or $\mathcal{L}$-Galois connections). In fact, when $\mathcal{L} = \mathbb{B}$ it boils down to the classical polarity definition.

Bělohlávek [2001] discusses how fuzzy Galois connections can be used to extend FCA for the case when the relation between the objects and concepts is *"non-sharp"*.

Relation algebras. As seen earlier on Chapter 4 relation algebras are an extension of Boolean algebras with the addition of the composition and converse operations, which yields a residuated semigroup structure. The interaction between the several operators and structures originates many Galois connections, some of which are listed below, for R, S and T of the correct types:

$$\begin{array}{rcl}
R \subseteq S \cap T & \Leftrightarrow & R \subseteq S \wedge R \subseteq T \\
R \cup S \subseteq T & \Leftrightarrow & R \subseteq T \wedge S \subseteq T \\
R \cap S \subseteq T & \Leftrightarrow & S \subseteq \neg R \cup T \\
\neg R \cap S \subseteq T & \Leftrightarrow & S \subseteq R \cup T \\
R \subseteq \neg S & \Leftrightarrow & \neg R \supseteq S \\
R^{\cup} \subseteq S & \Leftrightarrow & R \subseteq S^{\cup} \\
R \circ S \subseteq T & \Leftrightarrow & S \subseteq R \setminus T
\end{array}$$

$$R \circ S \subseteq T \quad \Leftrightarrow \quad R \subseteq T \,/\, S$$
$$R \setminus T \supseteq S \quad \Leftrightarrow \quad R \subseteq T \,/\, S$$

Two more Galois connections known as "shunting rules" [Bird and de Moor, 1997] exist, linking functions and relations. In fact, it can be shown that a relation f is a function if and only if, for all relations R and S of the correct types, any of the following Galois connections hold

$$f \circ R \subseteq S \quad \Leftrightarrow \quad R \subseteq f^{\cup} \circ S \tag{6.8}$$
$$R \circ f^{\cup} \subseteq S \quad \Leftrightarrow \quad R \subseteq S \circ f \tag{6.9}$$

6.4 Other examples

Injectivity. An *injectivity order* $\leqslant$ can be defined on relations [Oliveira, 2005]. This order measures the *"degree of injectivity"* (or, equivalently, if a relation is *"more or less defined"*).

The injectivity is a property of the *kernel* of a function. The kernel operator is defined, for all relations R, as

$$\ker R \quad \overset{def}{=} \quad R^{\cup} \circ R \tag{6.10}$$

Intuitively, the kernel relates elements of the domain which share the same images[1].

The injectivity order on relations is defined, for all relations R and S, as

$$R \leqslant S \quad \overset{def}{\Leftrightarrow} \quad \ker S \subseteq \ker R \tag{6.11}$$

where $R \leqslant S$ means that R is *less injective* than S. As expected, relation S is more injective than R because its kernel is smaller, i.e., there are less elements of the domain that share the same images. This definitions allows for comparing relations with different co-domains, provided that their domains are the same.

Since kernel is a function, and R and S can be seen as variables in the definition

[1]Recall from Section 4.1 that we extend the concept of *domain* and *co-domain* from functions to relations.

above, we can use the point-free transform to simplify (6.11):

$$
\begin{array}{ll}
 & \langle \forall\, R, S :: R \leqslant S \;\Leftrightarrow\; \ker S \subseteq \ker R \rangle \\
\Leftrightarrow & \qquad \{ \text{ Converse. } \} \\
 & \langle \forall\, R, S :: S \leqslant^{\cup} R \;\Leftrightarrow\; \ker S \subseteq \ker R \rangle \\
\Leftrightarrow & \qquad \{ \text{ Related functional results. } \} \\
 & \langle \forall\, R, S :: S \leqslant^{\cup} R \;\Leftrightarrow\; S(\ker^{\cup} \circ \subseteq \circ \ker) R \rangle \\
\Leftrightarrow & \qquad \{ \text{ Extensional equivalence. } \} \\
 & \leqslant^{\cup} \;\Leftrightarrow\; \ker^{\cup} \circ \subseteq \circ \ker \\
\Leftrightarrow & \qquad \{ \text{ Converse. } \} \\
 & \leqslant \;\Leftrightarrow\; (\ker^{\cup} \circ \subseteq \circ \ker)^{\cup} \\
\Leftrightarrow & \qquad \{ \text{ Contravariance and involution. } \} \\
 & \leqslant \;\Leftrightarrow\; \ker^{\cup} \circ \subseteq^{\cup} \circ \ker
\end{array}
$$

Thus, the injectivity order can be expressed as

$$\leqslant \quad \stackrel{def}{=} \quad \ker^{\cup} \circ \supseteq \circ \ker \tag{6.12}$$

This clearly instantiates the construction (5.34) introduced in Section 5.6.4, where $h := \ker$ and $\sqsubseteq := \supseteq$.

Using the homomorphic image construction, we can lift Galois connections defined on relations ordered by the inclusion order $\supseteq$, to Galois connections defined on relations ordered by the injectivity order $\leqslant$, provided that their adjoints are $\ker$-homomorphic. Let functions f and g be adjoints of a Galois connection defined on relations ordered by (converse) inclusion order $\supseteq$. Functions f' and g' are $\ker$-homomorphic to f and g, respectively, if by instantiating $h := \ker$ in (5.35) and (5.36), equations

$$\ker \circ f' \quad = \quad f \circ \ker \tag{6.13}$$

$$\ker \circ g' \quad = \quad g \circ \ker \tag{6.14}$$

hold. If this is the case, then f' and g' are adjoints of a Galois connection defined on relations ordered by injectivity $\leqslant$.

Let us try to establish a shunting rule for functions similar to (6.8) and (6.9) that works with the injectivity order. Thus, supposing that $f' := (\circ j^\cup)$ and $g' := (\circ j)$, we will use (6.13) and (6.14) to calculate f and g, and verify if these two functions are Galois connected. If this is the case, we can conclude that f' and g' are adjoints of a Galois connection.

We start by calculating function f:

$$
\begin{array}{ll}
 & \ker \circ f' = f \circ \ker \\
\Leftrightarrow & \qquad \{ \text{ Assuming } f' := (\circ j^\cup). \ \} \\
 & \ker \circ (\circ j^\cup) = f \circ \ker \\
\Leftrightarrow & \qquad \{ \text{ Relation extensional equality. } \} \\
 & \langle \forall\, R :: (\ker \circ (\circ j^\cup))\, R = (f \circ \ker)\, R \rangle \\
\Leftrightarrow & \qquad \{ \text{ Relation application. } \} \\
 & \langle \forall\, R :: \ker(R \circ j^\cup) = f\,(\ker R) \rangle \\
\Leftrightarrow & \qquad \{ \text{ Definition of kernel (6.10). } \} \\
 & \langle \forall\, R :: (R \circ j^\cup)^\cup \circ R \circ j^\cup = f\,(\ker R) \rangle \\
\Leftrightarrow & \qquad \{ \text{ Contravariance and involution of converse. Associativity of composition. } \} \\
 & \langle \forall\, R :: j \circ R^\cup \circ R \circ j^\cup = f\,(\ker R) \rangle \\
\Leftrightarrow & \qquad \{ \text{ Definition of kernel (6.10). } \} \\
 & \langle \forall\, R :: j \circ \ker R \circ j^\cup = f\,(\ker R) \rangle \\
\Leftrightarrow & \qquad \{ \text{ Variable introduction. } \} \\
 & \langle \forall\, R :: (j \circ X \circ j^\cup = f\, X) \ \wedge (X := \ker R) \rangle \\
\therefore & \qquad \{ \text{ Abstracting the variable. } \} \\
 & f\, X = j \circ X \circ j^\cup
\end{array}
$$

Using a similar calculus we can obtain the definition of function g. Thus, for all relations X and Y, f and g are defined as

$$ f\, X \overset{def}{=} j \circ X \circ j^\cup \tag{6.15} $$

$$ g\, Y \overset{def}{=} j^\cup \circ Y \circ j \tag{6.16} $$

Now, we must prove that functions f and g are Galois connected in the original preorder, i.e., that $f\,X \subseteq Y \Leftrightarrow X \subseteq g\,Y$ holds.

Proof

$$
\begin{array}{cl}
 & f\,X \subseteq Y \\
\Leftrightarrow & \qquad \{ \text{ Definition (6.15). } \} \\
 & j \circ X \circ j^{\cup} \subseteq Y \\
\Leftrightarrow & \qquad \{ \text{ Shunting of functions (6.8). } \} \\
 & X \circ j^{\cup} \subseteq j^{\cup} \circ Y \\
\Leftrightarrow & \qquad \{ \text{ Shunting of functions (6.9). } \} \\
 & X \subseteq j^{\cup} \circ Y \circ j \\
\Leftrightarrow & \qquad \{ \text{ Definition (6.16). } \} \\
 & X \subseteq g\,Y
\end{array}
$$

□

Thus, f is the lower adjoint and g is the upper adjoint of a Galois connection.

However, we should be careful because the injectivity order definition (6.12) uses the converse inclusion order. This means that we should consider the converse Galois connection instead, i.e., $g\,Y \supseteq X \Leftrightarrow Y \supseteq f\,X$, in which g is the lower adjoint and f is the upper adjoint. By the homomorphic construction, g' and f' are, respectively, the lower and upper adjoints of the new Galois connection. Thus, we conclude that, for all relations R and S, we have the Galois connection

$$R \circ j \leqslant S \quad \Leftrightarrow \quad R \leqslant S \circ j^{\cup}$$

which closely resembles connection (6.9), the only difference being the order.

This procedure can be exploited to obtain more Galois connections involving the injectivity order (see [Oliveira, 2005]), or extended to other orders.

Suprema and infima. Traditionally, suprema and infima are defined in terms of sets of upper and lower bounds [Priestley, 2000]. However, Backhouse [2000] defines suprema and infima in terms of the range of a function $\mathcal{B} \xleftarrow{f} \mathcal{A}$, where $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$ are partially ordered sets. This allows for the characterization of different kinds of infima and suprema by changing the domain of f (this is called the *shape* poset).

So let $(\mathcal{B} \leftarrow \mathcal{A}) \xleftarrow{\mathtt{K}} \mathcal{B}$ be the constant function defined as $(\mathtt{K}\, b)\, a \overset{def}{=} b$ that always returns the same value $b \in \mathcal{B}$. If the infimum of function f *exists* we write $\sqcap f$ to denote it, where $\mathcal{B} \xleftarrow{\sqcap} (\mathcal{B} \leftarrow \mathcal{A})$ is a function defined over functions. Similarly, a function $\mathcal{B} \xleftarrow{\sqcup} (\mathcal{B} \leftarrow \mathcal{A})$ returns the supremum of f if it exists.

If $\sqcap$ (resp. $\sqcup$) *exists* it is the lower (resp. upper) adjoint of a Galois connection in which the constant function is the other adjoint. Thus, for all $b \in \mathcal{B}$,

$$\mathtt{K}\, b \mathrel{\dot{\preceq}} f \quad \Leftrightarrow \quad b \preceq \sqcap f \tag{6.17}$$

$$\sqcup f \preceq b \quad \Leftrightarrow \quad f \mathrel{\dot{\preceq}} \mathtt{K}\, b \tag{6.18}$$

The particular case where $\mathcal{A} = \mathbb{B}$ boils down to the binary infimum and supremum operators, i.e., for all a, b and $c \in \mathcal{B}$,

$$\begin{aligned} a \sqcup b \sqsubseteq c \quad &\Leftrightarrow \quad a \sqsubseteq c \ \wedge \ b \sqsubseteq c \\ a \sqsubseteq b \sqcap c \quad &\Leftrightarrow \quad a \sqsubseteq b \ \wedge \ a \sqsubseteq c \end{aligned}$$

Although these equivalences do not seem like instances of Galois connections, we can recognized the fact after rewritten them as

$$\begin{aligned} \sqcup\,(a, b) \sqsubseteq c \quad &\Leftrightarrow \quad (a, b)\ (\sqsubseteq \times \sqsubseteq)\ \triangle\, c \\ a \sqsubseteq \sqcap\,(b, c) \quad &\Leftrightarrow \quad \triangle\, a\ (\sqsubseteq \times \sqsubseteq)\ (b, c) \end{aligned}$$

where $\triangle$ is the doubling function introduced in Section 5.6.2, defined as $\triangle\, a \overset{def}{=} (a, a)$, and the product order is defined as $(a, b)\ (\sqsubseteq \times \sqsubseteq)\ (c, d) \overset{def}{=} a \sqsubseteq c \ \wedge \ b \sqsubseteq d$.

Some properties of the binary supremum operator (and, by duality, of the binary infimum operator) immediately follow from the above definitions as a direct consequence of the properties of logical conjunctions, e.g., idempotence, commutativity and associativity:

$$\begin{aligned} x \sqcup x \quad &= \quad x \\ x \sqcup y \quad &= \quad y \sqcup x \\ (x \sqcup y) \sqcup z \quad &= \quad x \sqcup (y \sqcup z) \end{aligned}$$

A lattice is a poset for which (6.17) (or equivalently (6.18)) exists for all shape posets $\mathcal{A}$ [Backhouse, 2004].

An example of how infimum and supremum operators can be defined using Galois connections is the maximum and minimum operators on real numbers. In fact, for all x, y and $z \in \mathbb{R}$, we have the following Galois connections,

$$\begin{aligned} x \uparrow y \leqslant z \quad &\Leftrightarrow \quad x \leqslant z \ \wedge \ y \leqslant z \\ z \leqslant x \ \wedge \ z \leqslant y \quad &\Leftrightarrow \quad z \leqslant x \downarrow y \end{aligned}$$

Another example arises when natural numbers are ordered by the "divides" ordering, denoted as $\backslash$, in which $m \backslash n$ means that m exactly divides n, i.e., there exists $p \in \mathbb{N}$ such that $m \times p = n$. In this ordering, the *least common multiple*, lcm, is the supremum operator while the *greatest common divisor*, gcd, is the infimum operator. Therefore, for all m, n and $p \in \mathbb{N}$, we have the following Galois connections,

$$\begin{aligned} \mathrm{lcm}(m,n) \backslash p \quad &\Leftrightarrow \quad m \backslash p \ \wedge \ n \backslash p \\ p \backslash m \ \wedge \ p \backslash n \quad &\Leftrightarrow \quad p \backslash \mathrm{gcd}(m,n) \end{aligned}$$

Temporal algebra. Temporal logic is the general name given to logical systems that study the truth value of assertions through time, based on the principle that certain assertions are valid only for a limited period of time. Temporal logic has become important in computer science for the specification and verification of reactive computer programs [Pnueli, 1977].

von Karger [2000] gives an algebraic presentation of temporal logics, including linear temporal logic, interval temporal logic and duration calculus. The author uses an algebraic structure with two additional Galois connections (what he calls a *"Galois algebra"*) which adjoints are to be interpreted as the *next* and *previous* operators. See [von Karger, 2000] for more details.

Hash transpose. Oliveira and Rodrigues [2004] generalize the concept of functional transposition which is used transform relations into functions. If we apply the transpose operator Λ to a relation $\mathcal{B} \xleftarrow{R} \mathcal{A}$ we obtain a function $\wp\mathcal{B} \xleftarrow{f} \mathcal{A}$, such that

$$f = \Lambda R \quad \Leftrightarrow \quad (bRa \Leftrightarrow b \in f\, a)$$

or

$$f = \Lambda R \quad \Leftrightarrow \quad R = \in \circ f$$

Functional transposition is then extended to hash tables. For a fixed hash function $\mathcal{B} \xleftarrow{h} \mathcal{A}$, the hash-transpose of a set, represented by a co-reflexive relation $\mathcal{A} \xleftarrow{S} \mathcal{A}$ is a function $\wp\mathcal{A} \xleftarrow{t} \mathcal{B}$, such that $t = \Theta_h S$, where the hash-transpose operator $\Theta_h S$ is defined as

$$\Theta_h\, S \quad \overset{def}{=} \quad \Lambda(S \circ h^{\cup})$$

Another operator, which takes a hash table $\wp\mathcal{A} \xleftarrow{t} \mathcal{B}$ and returns the ("best") associated co-reflexive relation, is defined as

$$\Xi_h\, t \quad \overset{def}{=} \quad \mathit{rng}\,(\in \circ\, (t \,\dot{\cap}\, \Lambda(h^{\cup})))$$

They show that the pair (Θ_h, Ξ_h) forms a Galois connection

$$\Theta_h\, S \;\dot{\leqslant}\; t \quad \Leftrightarrow \quad S \subseteq \Xi_h\, t$$

which explains the representation of data collections by hash-tables and vice-versa.

Predicates. Backhouse [2000] gives several interesting examples of Galois connections that arise from predicates.

Let $(\mathcal{A}, \sqsubseteq)$ be a *poset* with greatest element $\top$ and let $\mathbb{B} \xleftarrow{p} \mathcal{A}$ be the predicate $p\,x \overset{def}{\Leftrightarrow} x \sqsubseteq a$, for some constant $a \in \mathcal{A}$. Denoting expression *"if b then a_1 else a_2 fi"* by $[a_1, a_2] \leftarrow b$, then, for all $x \in \mathcal{A}$ and $b \in \mathbb{B}$,

$$p\,x \Leftarrow b \quad \Leftrightarrow \quad x \sqsubseteq [a, \top] \leftarrow b$$

is a Galois connection.

Several other Galois connections of this kind arise, as shown in [Backhouse, 2004; Backhouse and Backhouse, 2004], namely:

$$\mathrm{d}_k\, m \Leftarrow b \quad \Leftrightarrow \quad m/([k, 1] \leftarrow b)$$

$$
\begin{array}{rcll}
s \in \mathcal{S} \Leftarrow b & \Leftrightarrow & \mathcal{S} \supseteq [\{s\},\emptyset] \leftarrow b & \\
(\mathcal{S} \neq \emptyset) \Leftarrow b & \Leftrightarrow & \mathcal{S} \subseteq [\emptyset,\mathcal{U}] \leftarrow b & \qquad\qquad \mathcal{S} \subseteq \mathcal{U}
\end{array}
$$

where $\mathbb{B} \xleftarrow{\mathrm{d}_k} \mathbb{N}$ is the predicate that tests if a number is divisible by $k \in \mathbb{N}$; $/$ is the *division* order on natural numbers (the converse of the *divides* ordering ($\backslash$) presented before), that is, m/n means that n divides m; and $s \in \mathcal{S} \overset{def}{=} \{s\} \subseteq \mathcal{S}$.

Weakest liberal precondition. Backhouse [2000] gives us another example of a Galois connection relating preconditions and post-conditions of program statements.

Let $s_2 \xleftarrow{S} s_1$ be a program statement from state s_1 to state s_2, and p and q two predicates on the state space of S. A *conditional correctness assertion* (also known as *Hoare triple* [Hoare, 1969]), $\{p\}S\{q\}$ means that if the predicate $p\,s_1$ holds, and S executes successfully, then $q\,s_2$ also holds. p is the *precondition* and q is the *post-condition* of S. This states the *conditional* (or *partial*) correctness of S because S may fail to terminate [Backhouse, 2000].

Dijkstra [1976] takes a different but related approach to total correctness based on predicate transformers. The standard *predicate transformer* is a function that maps post-conditions to preconditions [Back and von Wright, 1998]. In order to deal with possible non-termination, below we present a "liberal" version which ensures correctness whenever the program terminates (conditional correctness).

The *weakest liberal precondition*, $\mathrm{wlp}_S\ q$, is a predicate transformer that from a post-condition q returns the *weakest* precondition p capable of guaranteeing the conditional correctness of S, i.e.,

$$\{p\}S\{q\} \quad \Leftrightarrow \quad p \Rightarrow \mathrm{wlp}_S\ q$$

Conversely, the *strongest liberal post-condition*, $\mathrm{slp}_S\ p$, is a predicate transformer that from a pre-condition p returns the strongest post-condition q capable of guaranteeing the conditional correctness of S, i.e.,

$$\{p\}S\{q\} \quad \Leftrightarrow \quad \mathrm{slp}_S\ p \Rightarrow q$$

In fact, $\{p\}S\{q\}$ defines a pair algebra on predicates, and thus

$$\mathrm{slp}_S\ p \Rightarrow q \quad \Leftrightarrow \quad p \Rightarrow \mathrm{wlp}_S\ q$$

is a Galois connection.

Separation logic. Hoare logic allow us to reason about the correctness of imperative programs [Hoare, 1969]. However, it does not handle with structures such as linked lists, trees and so on. Reynolds [2002] describes *separation logic* as *"…an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure"*. Separation logic introduces commands for accessing and modifying shared structures and explicit allocation and deallocation of storage. Moreover, assertions are enriched with a separating conjunction and implication. Separating conjunction is specially important because it allows for assertions about disjoint parts of the heap.

Wang et al. [2008] propose *confined separation logic* as an extension to separation logic that is able to deal with dangling references. Confined separation logic enriches standard separation logic with three new variants of separating conjunction that describe the behavior of dangling references.

In [Wang et al., 2008], the authors prove that each type of conjunction of confined separation logic is a lower adjoint in a Galois connection, and that the respective upper adjoints are a form of implication. In particular, the standard separating implication is the upper adjoint of the separating conjunction. The properties of Galois connections are used to derive, for "free", the rules of separating conjunction and implication previously given in [Reynolds, 2002].

Data type refinement. The general theory of data type refinement [Oliveira, 1990; Alves et al., 2005] is not based on Galois connections. However, Galois connections arise naturally in certain situations that are sufficiently important to mention as examples of this ubiquitous concept in computer science.

Let us consider that data types are preordered sets. Thus, for two data types $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$

$$\begin{tikzcd}\mathcal{A} \arrow[r, bend left, "\mathrm{con}"] \arrow[r, phantom, "\leqslant"] & \mathcal{B} \arrow[l, bend left, "\mathrm{abs}"]\end{tikzcd}$$

such that,

$$\mathrm{abs} \circ \mathrm{con} = id_{\mathcal{A}}$$

we say that *"data type $\mathcal{B}$ implements, or refines data type $\mathcal{A}$"* [Alves et al., 2005]. abs is a surjective and simple relation called the *abstraction relation*; con is a injective and total relation called the *representation relation*.

In particular, whenever abs and con are *monotonic functions* which verify

$$id_{\mathcal{B}} \quad \sqsubseteq_{\mathcal{B}} \quad \mathrm{con} \circ \mathrm{abs}$$

then $(\mathcal{B}, \sqsubseteq) \xleftarrow{(\mathrm{abs},\mathrm{con})} (\mathcal{A}, \preceq)$ is a perfect Galois connection.

Oliveira [2008] provides a deeper explanation of the application of Galois connections to data type refinement.

6.5 Summary

This chapter provided some examples and applications of Galois connections, with special emphasis on abstract interpretation, probably the most well-known application of Galois connections. Abstract interpretation clearly shows the usefulness of the combinatory nature of Galois connections in the construction of complex analysis using more simple ones. The properties of Galois connections are also used to ensure that the preservation conditions are met.

However, many other examples were given such as their use in formal concept analysis or in data refinement. Also of special importance is the role of Galois connections in residuation theory and, in particular, their relevance in relation algebra.

This chapter closes the first part of this dissertation. Now we will move to the analysis of the design and implementation of the *Galculator*.

Part II

Galculator

Chapter 7

The language

This chapter presents *Galois*, the DSL which models the design of the *Galculator*. The emphasis will be on the algebraic nature of the language which is a consequence of the concepts which underpin the tool. Thus, only a subset of the complete DSL is described: the operational parts which deal with the execution of commands, derivation of rules and proofs are left out for economy of presentation.

For each module of the language (sub-language), we present its context-free grammar (lexical aspects and disambiguation issues are not discussed), its denotational semantics and its typing rules.

We start by introducing the language design principles that have guided the development of *Galois*. The following sections introduce the syntax, semantics and typing rules of the several sub-languages: types (Section 7.2); fork algebras (Section 7.3); functions (Section 7.4); orders (Section 7.5); Galois connections (Section 7.6); and modules (Section 7.7). Most of the material of this section follows [Silva et al., 2009].

7.1 Language design

Notation. *Galois* aims at taking advantage of the algebraic nature of the concepts being represented, maintaining the combinatory style of fork algebras and Galois connections. The notation tries to resemble the mathematical symbols, being as intuitive as possible. A decision was made of following the set-theoretical notation for relations instead of the fork algebra notation. The former is easier to represent in plain text and probably best known. However, some trade-offs were needed because it is hard to write symbols like $\top$ or $\perp$ in text; meaningful keywords are used instead.

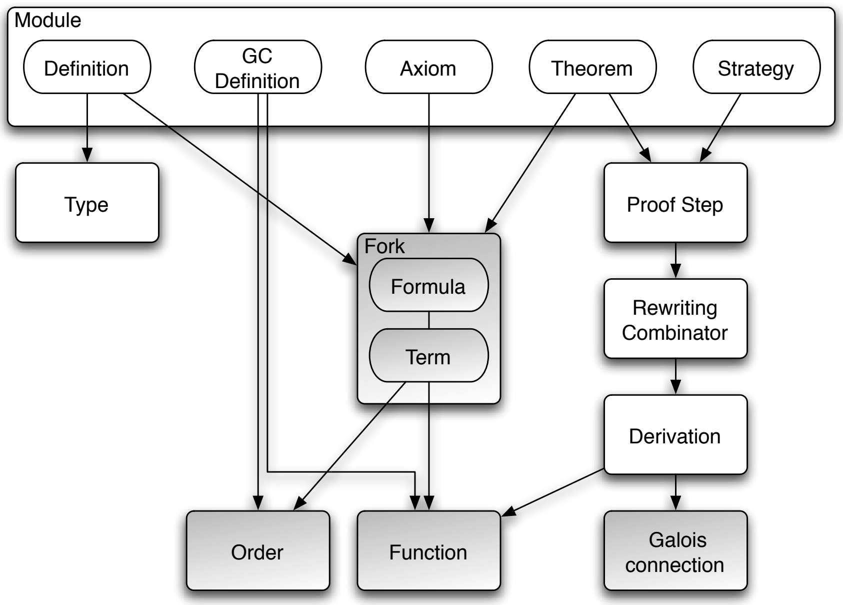


Figure 7.1: Structure of the sub-languages of the *Galois* DSLs. The arrows represent inclusion between languages; the shaded boxes are the fragments corresponding to the theoretical concepts introduced in the previous chapters.

Sub-languages. The complete language comprehends several sub-languages or modules concerning the different aspects of the *Galculator*. Its hierarchical structure is described in Fig. 7.1 where the shaded boxes correspond to the theoretical concepts introduced in the previous chapters. We start by giving a brief description of the sub-languages that are *not* discussed in this chapter:

Proof Step. Command language for handling proof steps, namely sequencing of single steps, finalization of proofs and the introduction of proofs by indirect equality.

Rewriting Combinator. The language of rewriting combinators allowing for the application of proof steps (rewriting rules) during proofs. *Galculator* uses a strategic term rewriting system [Silva and Oliveira, 2008] and this language closely follows its combinatory approach.

Derivation. Allows for the automatic derivation of equational properties from Galois connections and free-theorems from polymorphic functions. Once derived, these properties can be used as rewriting rules in proofs.

Syntax definition in SDF. The grammar of *Galois* is defined using the Syntax Definition Formalism (SDF) [Heering et al., 1989]. SDF is a formalism for specifying languages which offers modular design, lexical and context-free syntax and declarative disambiguation. It is not tied to a particular parsing technique, although a complete set of tools for grammar development using SDF exists [van den Brand et al., 2001].

In SDF, a production for a non-terminal B is written as $\mathsf{A} \longrightarrow \mathsf{B}$. Note that the non-terminal symbol of the production appears on the right-hand side which differs from most syntax formalisms. Terminal symbols are defined between quotes where backslash is the escape character. An optional symbol A is written as A?. An alternative between symbols A and B is written as $\mathsf{A} \mid \mathsf{B}$. Expression $\{\mathsf{A}\text{“;”}\}*$ represents a sequence of zero or more symbols A separated by symbol “;” .

SDF supports attributes in productions specified inside braces. In this text, we use the *cons* attribute which does not belong directly to SDF but is widely used by other tools to specify abstract syntax constructors.

Lexical variables. In the syntax definition we use the symbols Identifier, Variable and Constant which are part of the lexical grammar representing, respectively, identifiers of external definitions (references), variables and constant names.

Semantics. We specify the semantics of *Galois* by defining a semantic function from the abstract syntax to the denoted mathematical objects. Each semantic function takes an environment (Σ, Γ, Θ) where

- Σ is a mapping from identifiers into their respective definitions. In the next sections, we will abuse notation and use Σ in different contexts with different meanings: it can be a mapping from identifiers to fork terms, functions, orders or Galois connections. We could define a product of mappings, one for each concept, instead. However, it is always clear from the context which mapping is being used and this improves readability.

  Σ is a partial function, that is, it is not defined for all identifiers. Thus, before applying is to an identifier i, we must verify if it belongs to the domain of Σ: $i \in \mathrm{dom}(\Sigma)$. If this fails, the expression is meaningless.

  Notation $\{i \rightharpoonup e\} \uplus \Sigma$ is used to denote the extension of the partial function Σ. This is only valid when the identifier i does not belong to the domain of Σ. After the extension, the value of Σ for i becomes e, i.e., $\Sigma(i) = e$.

- Γ is an injective (total) function from variable names to variables. Like Σ we also use Γ in different contexts with different meanings. Thus, it can map variable names into type, relation, function, order or Galois connection variables.

  Being total means that it is defined for all variable names and that equal variable names are assigned to the same variable. Injectivity ensures that variables are not shared by different variable names.

- Θ is an injective function from constant names to a set of index constants. Thus, each constant name is associated with an index constant and an index constant cannot be shared by different constant names. The index constants are used to introduce sections of binary operations as introduced in Section 8.2.

Typing. We follow the traditional approach of presenting typing rules for each abstract syntax constructor based on the types of its components. An environment (Σ, Γ, Θ) is used: Σ is a mapping from identifiers to their types; Γ is an injective function from variable names to type variables; and Θ is an injective function from constant names to their types. In fact, we can take (Σ, Γ, Θ) as the same environment used before for semantics, by considering that Σ, Γ and Θ return a value and its respective type. For instance, Σ maps an identifier in a definition and its respective type, i.e., for an identifier i its definition is given by $\Sigma(i) = expr$ and its type is given by $\Sigma(i) = t$, meaning that $expr : t$ (notation $a : t$ means that *"a has type t"*). Thus, $\Sigma(i) : \Sigma(i)$, which we simply write as $i : \Sigma(i)$. When evaluating semantics, we only take the definition, while in typing rules we only take the typing information. Later on, we will explore this duality.

Using this overloading, we define the following as axioms:

$$\frac{}{\Sigma, \Gamma, \Theta \vdash i : \Sigma(i)} \qquad \frac{}{\Sigma, \Gamma, \Theta \vdash v : \Gamma(v)} \qquad \frac{}{\Sigma, \Gamma, \Theta \vdash c : \Theta(c)}$$

Σ and Γ are used in different contexts with different meanings in the same way as semantic functions. The meaning of type expressions used in the following sections is summarized in Table 7.2.

7.2 Types

Syntax. The language for type declarations is very simple. We define it in SDF as

Type expression	**Meaning**
1	Unitary data type (1 element)
$\mathbb{B}$	Boolean data type (2 elements)
$\mathbb{Z}$	Integer numbers
$\mathbb{R}$	Real numbers
$t_1 \times t_2$	Product of types t_1 and t_2
$t_1 + t_2$	Disjunct sum of types t_1 and t_2
$\wp(t)$	Power set of type t
$t^\star$	Sequence of type t
$t_1 \leftarrow t_2$	Function between types t_1 and t_2
$t_1 \sim t_2$	Relation between types t_1 and t_2
$t_1 \rightharpoonup t_2$	Map between types t_1 and t_2
$(t_1, \sqsubseteq_{t_1})$	Poset t_1 equipped with partial order $\sqsubseteq_{t_1}$
$(t_1, \sqsubseteq_{t_1}) \xleftarrow{(,)} (t_2, \sqsubseteq_{t_2})$	Galois connection between posets $(t_1, \sqsubseteq_{t_1})$ and $(t_2, \sqsubseteq_{t_2})$

Figure 7.2: Meaning of type expressions.

context-free syntax

"`(`" Type "`)`"	$\rightarrow$	Type	
Variable	$\rightarrow$	Type	{ *cons*("*TVar*") }
"`One`"	$\rightarrow$	Type	{ *cons*("*One*") }
"`Bool`"	$\rightarrow$	Type	{ *cons*("*Bool*") }
"`Char`"	$\rightarrow$	Type	{ *cons*("*Char*") }
"`String`"	$\rightarrow$	Type	{ *cons*("*String*") }
"`Int`"	$\rightarrow$	Type	{ *cons*("*Int*") }
"`Float`"	$\rightarrow$	Type	{ *cons*("*Float*") }
"`Maybe`" Type	$\rightarrow$	Type	{ *cons*("*Maybe*") }
"`Set`" Type	$\rightarrow$	Type	{ *cons*("*Set*") }
"`List`" Type	$\rightarrow$	Type	{ *cons*("*List*") }
Type "<-|" Type	$\rightarrow$	Type	{ *cons*("*Map*") }
Type "><" Type	$\rightarrow$	Type	{ *cons*("*Prod*") }
Type "-|-" Type	$\rightarrow$	Type	{ *cons*("*Either*") }
Type "<-" Type	$\rightarrow$	Type	{ *cons*("*Fun*") }

Type "<−>" Type	$\rightarrow$	Type	{ *cons*("*Rel*") }
Type "∼∼" Type	$\rightarrow$	Type	{ *cons*("*GC*") }
"`Ord`" Type	$\rightarrow$	Type	{ *cons*("*Ord*") }
"`Expr`" Type	$\rightarrow$	Type	{ *cons*("*Expr*") }

Semantics. In the definition which follows we avoid entering on complex details about types by considering them as sets of values together with possible invariant constraints. We omit Σ and Ω of the environment because they are not used.

$$\mathcal{C}^{Type}[\![TVar\ v]\!](\Gamma) \doteq \Gamma(v)$$
$$\mathcal{C}^{Type}[\![One]\!](\Gamma) \doteq 1$$
$$\mathcal{C}^{Type}[\![Bool]\!](\Gamma) \doteq \mathbb{B}$$
$$\mathcal{C}^{Type}[\![Char]\!](\Gamma) \doteq \{\text{'a',\ldots,'z','A',\ldots,'Z','0',\ldots,'9'}\}$$
$$\mathcal{C}^{Type}[\![String]\!](\Gamma) \doteq (\{\text{'a',\ldots,'z','A',\ldots,'Z','0',\ldots,'9'}\})^\star$$
$$\mathcal{C}^{Type}[\![Int]\!](\Gamma) \doteq \mathbb{Z}$$
$$\mathcal{C}^{Type}[\![Float]\!](\Gamma) \doteq \mathbb{R}$$
$$\mathcal{C}^{Type}[\![Prod\ t_1\ t_2]\!](\Gamma) \doteq \mathcal{C}^{Type}[\![t_1]\!](\Gamma) \times \mathcal{C}^{Type}[\![t_2]\!](\Gamma)$$
$$\mathcal{C}^{Type}[\![Either\ t_1\ t_2]\!](\Gamma) \doteq \mathcal{C}^{Type}[\![t_1]\!](\Gamma) + \mathcal{C}^{Type}[\![t_2]\!](\Gamma)$$
$$\mathcal{C}^{Type}[\![Maybe\ t]\!](\Gamma) \doteq 1 + \mathcal{C}^{Type}[\![t]\!](\Gamma)$$
$$\mathcal{C}^{Type}[\![Set\ t]\!](\Gamma) \doteq \wp(\mathcal{C}^{Type}[\![t]\!](\Gamma))$$
$$\mathcal{C}^{Type}[\![List\ t]\!](\Gamma) \doteq (\mathcal{C}^{Type}[\![t]\!](\Gamma))^\star$$
$$\mathcal{C}^{Type}[\![Map\ t_1\ t_2]\!](\Gamma) \doteq \mathcal{C}^{Type}[\![t_1]\!](\Gamma) \rightharpoonup \mathcal{C}^{Type}[\![t_2]\!](\Gamma)$$
$$\mathcal{C}^{Type}[\![Fun\ t_1\ t_2]\!](\Gamma) \doteq \mathcal{C}^{Type}[\![t_1]\!](\Gamma) \leftarrow \mathcal{C}^{Type}[\![t_2]\!](\Gamma)$$
$$\mathcal{C}^{Type}[\![Rel\ t_1\ t_2]\!](\Gamma) \doteq \mathcal{C}^{Type}[\![t_1]\!](\Gamma) \sim \mathcal{C}^{Type}[\![t_2]\!](\Gamma)$$
$$\mathcal{C}^{Type}[\![GC\ t_1\ t_2]\!](\Gamma) \doteq (t_1', \sqsubseteq_{t_1'}) \xleftarrow{(,)} (t_2', \sqsubseteq_{t_2'}) \quad \text{where}$$
$$t_1' = \mathcal{C}^{Type}[\![t_1]\!](\Gamma)$$
$$t_2' = \mathcal{C}^{Type}[\![t_2]\!](\Gamma)$$
$$\mathcal{C}^{Type}[\![Ord\ t]\!](\Gamma) \doteq (t', \sqsubseteq_{t'}) \quad \text{where}$$
$$t' = \mathcal{C}^{Type}[\![t]\!](\Gamma)$$
$$\mathcal{C}^{Type}[\![Expr\ t]\!](\Gamma) \doteq \mathcal{C}^{Type}[\![t]\!](\Gamma)$$

$Char$ corresponds to a set of characters and $String$ is a sequence of characters. $Ord\ t$ corresponds to a type equipped with a partial order. However, the type system does not enforce that this invariant holds. $GC\ t_1\ t_2$ corresponds to a Galois connection between two types equipped with partial orders. Again, the type system does not enforce the invariant. $Expr$ is an alias for the argument type; it is just used to ensure the correctness of some constructions.

7.3 Fork Algebras

Syntax. The syntax of *Galois* for the fork algebra operators is defined in SDF as

context-free syntax

Term "=" Term	→	Formula	{ *cons*("*Equal*") }
Term "<=" Term	→	Formula	{ *cons*("*Less*") }
"(" Term ")"	→	Term	
Identifier	→	Term	{ *cons*("*Ident*") }
Variable	→	Term	{ *cons*("*Var*") }
"Id"	→	Term	{ *cons*("*Id*") }
"Top"	→	Term	{ *cons*("*Top*") }
"Bot"	→	Term	{ *cons*("*Bot*") }
"Pi1"	→	Term	{ *cons*("*Pi1*") }
"Pi2"	→	Term	{ *cons*("*Pi2*") }
"∼" Term	→	Term	{ *cons*("*Neg*") }
Term "∗"	→	Term	{ *cons*("*Conv*") }
Term "/\" Term	→	Term	{ *cons*("*Meet*") }
Term "\/" Term	→	Term	{ *cons*("*Join*") }
Term "." Term	→	Term	{ *cons*("*Comp*") }
Term "/ ∗ \" Term	→	Term	{ *cons*("*Fork*") }
Term "><" Term	→	Term	{ *cons*("*Prod*") }
"Ord" "[" Order "]"	→	Term	{ *cons*("*Ord*") }
"Fun" "[" Function "]"	→	Term	{ *cons*("*Fun*") }

The order and function languages are embedded in the fork language by using the "`Ord`" and "`Fun`" keywords, respectively. Although lacking in elegance, this decision is a trade-off between the usual practice in mathematics of using an overloaded notation, and the ease of implementation. In fact, this makes the disambiguation of the grammar much simpler. Otherwise, a possible solution would be to extend the type system of *Galculator* to support some kind of type overloading [Silva and Oliveira, 2008].

Semantics. The semantics of the language is almost straightforward for most of the constructors: they denote the corresponding operation in fork algebra.

$$
\begin{aligned}
&\mathcal{C}^{Formula}[\![\mathit{Equal}\ r_1\ r_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Term}[\![r_1]\!](\Sigma,\Gamma,\Theta) = \mathcal{C}^{Term}[\![r_2]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Formula}[\![\mathit{Less}\ r_1\ r_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Term}[\![r_1]\!](\Sigma,\Gamma,\Theta) \subseteq \mathcal{C}^{Term}[\![r_2]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Term}[\![\mathit{Ident}\ i]\!](\Sigma,\Gamma,\Theta) \doteq \Sigma(i) \quad \text{if} \quad i \in \mathrm{dom}(\Sigma)\\
&\mathcal{C}^{Term}[\![\mathit{Var}\ v]\!](\Sigma,\Gamma,\Theta) \doteq \Gamma(v)\\
&\mathcal{C}^{Term}[\![\mathit{Id}]\!](\Sigma,\Gamma,\Theta) \doteq \mathit{id}\\
&\mathcal{C}^{Term}[\![\mathit{Top}]\!](\Sigma,\Gamma,\Theta) \doteq \top\\
&\mathcal{C}^{Term}[\![\mathit{Bot}]\!](\Sigma,\Gamma,\Theta) \doteq \bot\\
&\mathcal{C}^{Term}[\![\mathit{Pi1}]\!](\Sigma,\Gamma,\Theta) \doteq \pi_1\\
&\mathcal{C}^{Term}[\![\mathit{Pi2}]\!](\Sigma,\Gamma,\Theta) \doteq \pi_2\\
&\mathcal{C}^{Term}[\![\mathit{Neg}\ r]\!](\Sigma,\Gamma,\Theta) \doteq \neg\mathcal{C}^{Term}[\![r]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Term}[\![\mathit{Conv}\ r]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Term}[\![r]\!](\Sigma,\Gamma,\Theta)^{\cup}\\
&\mathcal{C}^{Term}[\![\mathit{Meet}\ r_1\ r_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Term}[\![r_1]\!](\Sigma,\Gamma,\Theta) \cap \mathcal{C}^{Term}[\![r_2]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Term}[\![\mathit{Join}\ r_1\ r_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Term}[\![r_1]\!](\Sigma,\Gamma,\Theta) \cup \mathcal{C}^{Term}[\![r_2]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Term}[\![\mathit{Fork}\ r_1\ r_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Term}[\![r_1]\!](\Sigma,\Gamma,\Theta) \nabla \mathcal{C}^{Term}[\![r_2]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Term}[\![\mathit{Comp}\ r_1\ r_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Term}[\![r_1]\!](\Sigma,\Gamma,\Theta) \circ \mathcal{C}^{Term}[\![r_2]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Term}[\![\mathit{Prod}\ r_1\ r_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Term}[\![r_1]\!](\Sigma,\Gamma,\Theta) \times \mathcal{C}^{Term}[\![r_2]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Term}[\![\mathit{Ord}\ o]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Ord}[\![o]\!](\Sigma,\Gamma,\Theta)\\
&\mathcal{C}^{Term}[\![\mathit{Fun}\ f]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Fun}[\![f]\!](\Sigma,\Gamma,\Theta)
\end{aligned}
$$

$\mathcal{C}^{Ord}$ and $\mathcal{C}^{Fun}$ are semantic functions defined for partial orders and functions, respectively, as explained in sections to follow.

Typing. The typing rules for fork algebra formulæ and terms are

$$\frac{\Sigma, \Gamma, \Theta \vdash r_1 : t \qquad \Sigma, \Gamma, \Theta \vdash r_2 : t}{\Sigma, \Gamma, \Theta \vdash \mathit{Equal}\ r_1\ r_2 : t} \qquad \frac{\Sigma, \Gamma, \Theta \vdash r_1 : t \qquad \Sigma, \Gamma, \Theta \vdash r_2 : t}{\Sigma, \Gamma \vdash \mathit{Less}\ r_1\ r_2 : t}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash \Sigma(i) : t}{\Sigma, \Gamma, \Theta \vdash \mathit{Ident}\ i : t} \qquad \frac{\Sigma, \Gamma, \Theta \vdash \Gamma(v) : t_1 \sim t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{Var}\ v : t_1 \sim t_2}$$

$$\frac{}{\Sigma, \Gamma, \Theta \vdash \mathit{Id} : t \sim t}$$

$$\frac{}{\Sigma, \Gamma, \Theta \vdash \mathit{Top} : t_1 \sim t_2} \qquad \frac{}{\Sigma, \Gamma, \Theta \vdash \mathit{Bot} : t_1 \sim t_2}$$

$$\frac{}{\Sigma, \Gamma, \Theta \vdash \mathit{Pi1} : t_1 \leftarrow t_1 \times t_2} \qquad \frac{}{\Sigma, \Gamma, \Theta \vdash \mathit{Pi2} : t_2 \leftarrow t_1 \times t_2}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash r : t_1 \sim t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{Neg}\ r : t_1 \sim t_2} \qquad \frac{\Sigma, \Gamma, \Theta \vdash r : t_1 \sim t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{Conv}\ r : t_2 \sim t_1}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma, \Gamma, \Theta \vdash r_2 : t_1 \sim t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{Meet}\ r_1\ r_2 : t_1 \sim t_2}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma, \Gamma, \Theta \vdash r_2 : t_1 \sim t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{Join}\ r_1\ r_2 : t_1 \sim t_2}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma, \Gamma, \Theta \vdash r_2 : t_2 \sim t_3}{\Sigma, \Gamma, \Theta \vdash \mathit{Comp}\ r_1\ r_2 : t_1 \sim t_3}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma, \Gamma, \Theta \vdash r_2 : t_3 \sim t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{Fork}\ r_1\ r_2 : (t_1 \times t_3) \sim t_2}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma, \Gamma, \Theta \vdash r_2 : t_3 \sim t_4}{\Sigma, \Gamma, \Theta \vdash \mathit{Prod}\ r_1\ r_2 : (t_1 \times t_3) \sim (t_2 \times t_4)}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash o : (t, \sqsubseteq_t)}{\Sigma, \Gamma, \Theta \vdash \mathit{Ord}\ o : t \sim t} \qquad \frac{\Sigma, \Gamma, \Theta \vdash f : t_1 \leftarrow t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{Fun}\ f : t_1 \sim t_2}$$

7.4 Functions

Syntax. The syntax of the function sub-language is a restriction of the fork algebra language to the operations for which functions are closed. It only introduces syntax to denote sections of binary functions:

context-free syntax

"(" Function ")"	$\rightarrow$	Function	
Identifier	$\rightarrow$	Function	{ *cons*("*FunctionIdent*") }
Variable	$\rightarrow$	Function	{ *cons*("*FunctionVar*") }
"Id"	$\rightarrow$	Function	{ *cons*("*FunctionId*") }
Function "." Function	$\rightarrow$	Function	{ *cons*("*FunctionComp*") }
Function "<" Section ">"	$\rightarrow$	Function	{ *cons*("*RightSection*") }
"<" Section ">" Function	$\rightarrow$	Function	{ *cons*("*LeftSection*") }
Constant	$\rightarrow$	Section	{ *cons*("*Const*") }
Function "<" Constant "," Constant ">"	$\rightarrow$	Section	{ *cons*("*FunctConst*") }

Semantics. The environment of the semantic function is similar to the one used before for fork algebra terms and formulas.

$$\mathcal{C}^{Fun}[\![\mathit{FunctionIdent}\ i]\!](\Sigma,\Gamma,\Theta) \doteq \Sigma(i) \quad \text{if} \quad i \in \mathrm{dom}(\Sigma)$$
$$\mathcal{C}^{Fun}[\![\mathit{FunctionVar}\ v]\!](\Sigma,\Gamma,\Theta) \doteq \Gamma(v)$$
$$\mathcal{C}^{Fun}[\![\mathit{FunctionId}]\!](\Sigma,\Gamma,\Theta) \doteq \ id$$
$$\mathcal{C}^{Fun}[\![\mathit{FunctionComp}\ f_1\ f_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Fun}[\![f_1]\!](\Sigma,\Gamma,\Theta) \circ \mathcal{C}^{Fun}[\![f_2]\!](\Sigma,\Gamma,\Theta)$$
$$\mathcal{C}^{Fun}[\![\mathit{RightSection}\ f\ s]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Fun}[\![f]\!](\Sigma,\Gamma,\Theta)_{(\mathcal{C}^{Sect}[\![s]\!](\Sigma,\Gamma,\Theta))}$$
$$\mathcal{C}^{Fun}[\![\mathit{LeftSection}\ s\ f]\!](\Sigma,\Gamma,\Theta) \doteq_{(\mathcal{C}^{Sect}[\![s]\!](\Sigma,\Gamma,\Theta))} \mathcal{C}^{Fun}[\![f]\!](\Sigma,\Gamma,\Theta)$$
$$\mathcal{C}^{Sect}[\![\mathit{Const}\ c]\!](\Sigma,\Gamma,\Theta) \doteq \Theta(c)$$
$$\mathcal{C}^{Sect}[\![\mathit{FunctConst}\ f\ s_1\ s_2]\!](\Sigma,\Gamma,\Theta) \doteq_{(\Theta(s_1))} \mathcal{C}^{Fun}[\![f]\!](\Sigma,\Gamma,\Theta)_{(\Theta(s_2))}$$

The subscript notation is used to denote sections of functions. Therefore, given a binary function $\mathcal{C} \xleftarrow{f} \mathcal{B} \times \mathcal{A}$, the right section of f is denoted as f_a for a value $a \in \mathcal{A}$, and the left section of f is denoted as ${}_bf$ for a value $b \in \mathcal{B}$. The simultaneous sectioning of both arguments of function f is denoted by ${}_bf_a$, for values $a \in \mathcal{A}$ and $b \in \mathcal{B}$, corresponding to a value of type $\mathcal{C}$. Sections of binary functions will be further discussed in Section 8.2.

Typing. The typing rules for functions are

$$\frac{\Sigma, \Gamma, \Theta \vdash \Sigma(i) : t_1 \leftarrow t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{FunctionIdent}\ i : t_1 \leftarrow t_2}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash \Gamma(v) : t_1 \leftarrow t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{FunctionVar}\ v : t_1 \leftarrow t_2}$$

$$\frac{}{\Sigma, \Gamma, \Theta \vdash \mathit{FunctionId} : t \leftarrow t}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash f_1 : t_1 \leftarrow t_2 \qquad \Sigma, \Gamma, \Theta \vdash f_2 : t_2 \leftarrow t_3}{\Sigma, \Gamma, \Theta \vdash \mathit{FunctionComp}\ f_1\ f_2 : t_1 \leftarrow t_3}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash f : t_1 \leftarrow t_2 \times t_3 \qquad \Sigma, \Gamma, \Theta \vdash s : t_3}{\mathit{RightSection}\ f\ s : t_1 \leftarrow t_2}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash f : t_1 \leftarrow t_2 \times t_3 \qquad \Sigma, \Gamma, \Theta \vdash s : t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{LeftSection}\ f\ s : t_1 \leftarrow t_3}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash \Theta(c) : t}{\Sigma, \Gamma, \Theta \vdash \mathit{Const}\ c : t}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash f : t \leftarrow t_1 \times t_2 \qquad \Sigma, \Gamma, \Theta \vdash \Theta(s_1) : t_1 \qquad \Sigma, \Gamma, \Theta \vdash \Theta(s_2) : t_2}{\Sigma, \Gamma, \Theta \vdash \mathit{FunctConst}\ f\ s_1\ s_2 : t}$$

7.5 Orders

Syntax. The syntax for the order sub-language is

context-free syntax

"(" Order ")"	$\rightarrow$	Order	
Identifier	$\rightarrow$	Order	{ *cons*("*OrderIdent*") }
Variable	$\rightarrow$	Order	{ *cons*("*OrderVar*") }
"`Id`"	$\rightarrow$	Order	{ *cons*("*OrderId*") }
Order "$*$"	$\rightarrow$	Order	{ *cons*("*OrderConv*") }

Semantics. The semantic function maps each abstract syntax constructor into the corresponding fork algebra operation. As explained later in Section 8.2, only operations which are closed for partial orders are defined.

$$\begin{aligned}
&\mathcal{C}^{Ord}[\![\mathit{OrderIdent}\ i]\!](\Sigma,\Gamma,\Theta) \doteq \Sigma(i) \quad \text{if} \quad i \in \mathrm{dom}(\Sigma)\\
&\mathcal{C}^{Ord}[\![\mathit{OrderVar}\ v]\!](\Sigma,\Gamma,\Theta) \doteq \Gamma(v)\\
&\mathcal{C}^{Ord}[\![\mathit{OrderId}]\!](\Sigma,\Gamma,\Theta) \doteq \mathit{id}\\
&\mathcal{C}^{Ord}[\![\mathit{OrderConv}\ o]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{Ord}[\![o]\!](\Sigma,\Gamma,\Theta)^{\cup}
\end{aligned}$$

Typing. The typing rules for orders are

$$\frac{\Sigma,\Gamma,\Theta \vdash \Sigma(i) : (t, \sqsubseteq_t)}{\Sigma,\Gamma,\Theta \vdash \mathit{OrderIdent}\ i : (t, \sqsubseteq_t)} \qquad \frac{\Sigma,\Gamma,\Theta \vdash \Gamma(v) : (t, \sqsubseteq_t)}{\Sigma,\Gamma,\Theta \vdash \mathit{OrderVar}\ v : (t, \sqsubseteq_t)}$$

$$\frac{}{\Sigma,\Gamma,\Theta \vdash \mathit{OrderId} : (t, \sqsubseteq_t)} \qquad \frac{\Sigma,\Gamma,\Theta \vdash o : (t, \sqsubseteq_t)}{\Sigma,\Gamma,\Theta \vdash \mathit{OrderConv}\ o : (t, \sqsubseteq_t)}$$

7.6 Galois Connections

Syntax. The syntax for Galois connections' combinators is as follows:

context-free syntax

"(" Galois ")"	$\rightarrow$	Galois	
Identifier	$\rightarrow$	Galois	{ *cons*("*GCIdent*") }
Variable	$\rightarrow$	Galois	{ *cons*("*GCVar*") }
"`Id`"	$\rightarrow$	Galois	{ *cons*("*GCId*") }
Galois "." Galois	$\rightarrow$	Galois	{ *cons*("*GCComp*") }
Galois "$*$"	$\rightarrow$	Galois	{ *cons*("*GCConv*") }

Semantics. The semantic function maps the abstract syntax into the algebraic operations defined in Section 4.4.

$$
\begin{aligned}
&\mathcal{C}^{GC}[\![\,\mathit{GCIdent}\ i]\!](\Sigma,\Gamma,\Theta) \doteq \Sigma(i) \quad \text{if} \quad i \in \mathrm{dom}(\Sigma)\\
&\mathcal{C}^{GC}[\![\,\mathit{GCVar}\ v]\!](\Sigma,\Gamma,\Theta) \doteq \Gamma(v)\\
&\mathcal{C}^{GC}[\![\,\mathit{GCId}]\!](\Sigma,\Gamma,\Theta) \doteq (t,\sqsubseteq_t) \xleftarrow{(id,id)} (t,\sqsubseteq_t)\\
&\mathcal{C}^{GC}[\![\,\mathit{GCConv}\ g]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{GC}[\![g]\!](\Sigma,\Gamma,\Theta)^{\cup}\\
&\mathcal{C}^{GC}[\![\,\mathit{GCComp}\ g_1\ g_2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{GC}[\![g_1]\!](\Sigma,\Gamma,\Theta) \circ \mathcal{C}^{GC}[\![g_2]\!](\Sigma,\Gamma,\Theta)
\end{aligned}
$$

Note that GGId denotes the identity Galois connection, in which both adjoints are the identity function defined over any poset $(t,\sqsubseteq_t)$.

Typing. The typing rules for Galois connections are

$$\frac{\Sigma,\Gamma,\Theta \vdash \Sigma(i) : (t_1,\sqsubseteq_{t_1}) \xleftarrow{(,)} (t_2,\sqsubseteq_{t_2})}{\Sigma,\Gamma,\Theta \vdash \mathit{GCIdent}\ i : (t_1,\sqsubseteq_{t_1}) \xleftarrow{(,)} (t_2,\sqsubseteq_{t_2})}$$

$$\frac{\Sigma,\Gamma,\Theta \vdash \Gamma(v) : (t_1,\sqsubseteq_{t_1}) \xleftarrow{(,)} (t_2,\sqsubseteq_{t_2})}{\Sigma,\Gamma,\Theta \vdash \mathit{GCVar}\ v : (t_1,\sqsubseteq_{t_1}) \xleftarrow{(,)} (t_2,\sqsubseteq_{t_2})}$$

$$\frac{}{\Sigma,\Gamma,\Theta \vdash \mathit{GCId} : (t,\sqsubseteq_{t}) \xleftarrow{(,)} (t,\sqsubseteq_{t})}$$

$$\frac{\Sigma,\Gamma,\Theta \vdash g : (t_2,\sqsubseteq_{t_2}) \xleftarrow{(,)} (t_1,\sqsubseteq_{t_1})}{\Sigma,\Gamma,\Theta \vdash \mathit{GCConv}\ g : (t_1,\sqsubseteq_{t_1}) \xleftarrow{(,)} (t_2,\sqsubseteq_{t_2})}$$

$$\frac{\Sigma,\Gamma,\Theta \vdash g_1 : (t_1,\sqsubseteq_{t_1}) \xleftarrow{(,)} (t_2,\sqsubseteq_{t_2}) \qquad \Sigma,\Gamma,\Theta \vdash g_2 : (t_2,\sqsubseteq_{t_2}) \xleftarrow{(,)} (t_3,\sqsubseteq_{t_3})}{\Sigma,\Gamma,\Theta \vdash \mathit{GCComp}\ g_1\ g_2 : (t_1,\sqsubseteq_{t_1}) \xleftarrow{(,)} (t_3,\sqsubseteq_{t_3})}$$

7.7 Modules

Galois allows for organizing each theory in its own module, by defining particular operations and Galois connections, and specifying axioms and theorems about them. Moreover, modules provide a mechanism of reuse of concepts, since definitions can be referenced elsewhere in other modules[1]. Currently, modules are just namespaces, not allowing any kind of parametrization.

[1]In the future, we plan to introduce explicit import declarations, thus introducing an hierarchical structure of modules.

Syntax. The syntax of a module is a sequence of zero or more definitions, axioms, theorems or Galois connections, separated by the ";" character. It should be noticed that the order does not matter and that different kinds of definitions can be mixed.

context-free syntax

{ (Definition | Axiom | Theorem | GaloisDef) ";" }∗	$\to$	Module
Identifier ":" Type (":=" Term)?	$\to$	Definition
		{ *cons*("*Definition*") }
"`Axiom`" Identifier ":=" Formula	$\to$	Axiom
		{ *cons*("*Axiom*") }
"`Theorem`" Identifier ":=" Formula Proof	$\to$	Theorem
		{ *cons*("*Theorem*") }
"`Galois`" Identifier ":=" Function Function Order Order	$\to$	GaloisDef
		{ *cons*("*GaloisDef*") }

In a definition, only the specification of its type is mandatory; the defining term is optional. This allows for the introduction of concepts and the respective characterization by means of their properties instead of an actual definition.

In a theorem, Proof specifies a sequence of proof steps, whose structure is left out of this document, as explained earlier on. Informally, proofs contain proof step information necessary for establishing the proof of the argument expression.

Semantics. Modules introduce definitions in a environment where they can be referenced. We further enrich the environment Σ, such as done before, in order to map identifier to axioms and theorems. Definitions can be added in any order to the environment but we must ensure that the identifier does not exist in the domain of Σ.

$$\mathcal{C}^{Mod}[\![\mathit{Definition}\ i\ \mathit{type}\ ()]\!](\Sigma,\Gamma,\Theta) \doteq \{i \rightharpoonup ()\} \uplus \Sigma \quad \text{if } i \notin \text{dom}(\Sigma)$$

$$\mathcal{C}^{Mod}[\![\mathit{Definition}\ i\ \mathit{type}\ \mathit{expr}]\!](\Sigma,\Gamma,\Theta) \doteq \{i \rightharpoonup e\} \uplus \Sigma \quad \text{if } i \notin \text{dom}(\Sigma)$$
$$\text{where} \quad e = \mathcal{C}^{Term}[\![\mathit{expr}]\!](\Sigma,\Gamma,\Theta)$$

$$\mathcal{C}^{Mod}[\![\mathit{Axiom}\ i\ \mathit{expr}]\!](\Sigma,\Gamma,\Theta) \doteq \{i \rightharpoonup e\} \uplus \Sigma \quad \text{if } i \notin \text{dom}(\Sigma)$$
$$\text{where} \quad e = \mathcal{C}^{Formula}[\![\mathit{expr}]\!](\Sigma,\Gamma,\Theta)$$

$$\mathcal{C}^{Mod}[\![\mathit{Theorem}\ i\ \mathit{expr}\ \mathit{proof}]\!](\Sigma,\Gamma,\Theta) \doteq \{i \rightharpoonup e\} \uplus \Sigma \quad \text{if } i \notin \text{dom}(\Sigma)$$
$$\text{and } \mathit{proof} \text{ is a proof of } e$$

$$\text{where} \quad e = \mathcal{C}^{Formula}[\![expr]\!](\Sigma, \Gamma, \Theta)$$

$$\mathcal{C}^{Mod}[\![\mathit{GaloisDef}\ i\ f_1\ f_2\ o_1\ o_2]\!](\Sigma, \Gamma, \Theta) \doteq \{i \rightharpoonup (f_1', f_2', o_1', o_2')\} \uplus \Sigma$$

$$\text{if } i \notin \mathrm{dom}(\Sigma)$$

$$\begin{aligned} \text{where} \quad f_1' &= \mathcal{C}^{Fun}[\![f_1]\!](\Sigma, \Gamma, \Theta) \\ f_2' &= \mathcal{C}^{Fun}[\![f_2]\!](\Sigma, \Gamma, \Theta) \\ o_1' &= \mathcal{C}^{Ord}[\![o_1]\!](\Sigma, \Gamma, \Theta) \\ o_2' &= \mathcal{C}^{Ord}[\![o_2]\!](\Sigma, \Gamma, \Theta) \end{aligned}$$

Typing. The typing rules for modules are

$$\frac{t = \mathcal{C}^{Type}[\![tp]\!](\Sigma, \Gamma, \Theta)}{\Sigma, \Gamma, \Theta \vdash \mathit{Definition}\ i\ tp\ ()\ : t} \qquad \frac{\Sigma, \Gamma, \Theta \vdash e : t \qquad t = \mathcal{C}^{Type}[\![tp]\!](\Sigma, \Gamma, \Theta)}{\Sigma, \Gamma, \Theta \vdash \mathit{Definition}\ i\ tp\ e : t}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash e : t}{\Sigma, \Gamma, \Theta \vdash \mathit{Axiom}\ i\ e : t} \qquad \frac{\Sigma, \Gamma, \Theta \vdash e : t}{\Sigma, \Gamma, \Theta \vdash \mathit{Theorem}\ i\ e\ p : t}$$

$$\frac{\Sigma, \Gamma, \Theta \vdash f_1 : t_2 \leftarrow t_1 \quad \Sigma, \Gamma, \Theta \vdash f_2 : t_1 \leftarrow t_2 \qquad \Sigma, \Gamma, \Theta \vdash o_1 : (t_1, \sqsubseteq_{t_1}) \quad \Sigma, \Gamma, \Theta \vdash o_2 : (t_2, \sqsubseteq_{t_2})}{\Sigma, \Gamma, \Theta \vdash \mathit{GaloisDef}\ i\ f_1\ f_2\ o_1\ o_2 : (t_1, \sqsubseteq_{t_1}) \overset{(,)}{\longleftarrow} (t_2, \sqsubseteq_{t_2})}$$

7.8 Summary

Several formal languages for mathematical reasoning exist. Although most of them are close to some kind of logic, a few relational languages exist. We could have used one of those languages as front-end for the *Galculator*. However, none of them completely meets our requirements: being simple and close to fork algebras; being strongly typed; being able to accommodate functions, orders and Galois connections easily. Thus, we decided to develop *Galois* a simple language that would satisfy this conditions. This chapter presented the details of the design of *Galois*, together with the syntax, semantics and typing rules of its relevant fragments. In the following chapter, we will see how declarations in *Galois* define an equational system which can be used in proofs. In Chapter 9, we will show how GADTs and existential data types can be used to implement the language maintaining its type safeness.

Chapter 8

Foundations of *Galculator*

This chapter provides the theoretical foundations of *Galculator*. We start by presenting the equational theory underlying our calculus. Moreover, we argue about the correctness of using a term rewriting system as the proof engine of *Galculator*.

Then, we show how Galois connections can be integrated with fork algebras and how indirect equality is formulated in a point-free setting.

Finally, we go back to the motivation given in the introduction of this dissertation and show how the proofs in there can be rewriting in a point-free style.

8.1 Equational theory

Recalling the notions introduced in Section 3.2, we start by discussing the equational theory used by *Galculator*. In this section, we restrict the *Galois* language to only fork algebra formulæ and terms (without the inclusion of function and orders). In the next sections, we will see that the other concepts are just extensions of the fork algebra language.

8.1.1 Fork algebras

Equational system. Using *Galois* concrete syntax we define the equational system $\mathcal{E}_{\text{Fork}}$ which will serve as the basis of *Galculator*.

```
Axiom Meet_comut  := r /\ s = s /\ r;
Axiom Meet_assoc  := (r /\ s) /\ t = r /\ (s /\ t);
Axiom Meet_absor  := r /\ (r \/ s) = r;
Axiom Meet_distr  := r /\ (s \/ t) = (r /\ s) \/ (r /\ t);
Axiom Meet_compl  := r /\ ~r = Bot;

Axiom Join_comut  := r \/ s = s \/ r;
Axiom Join_assoc  := (r \/ s) \/ t = r \/ (s \/ t);
Axiom Join_absor  := r \/ (r /\ s) = r;
Axiom Join_distr  := r \/ (s /\ t) = (r \/ s) /\ (r \/ t);
Axiom Join_compl  := r \/ ~r = Top;

Axiom Comp_assoc  := (r . s) . t = r . (s . t);
Axiom Comp_unit   := r . Id = r;
Axiom Comp_distr  := (r \/ s) . t = r . t \/ s . t;
Axiom Involution  := (r*)* = r;
Axiom Conv_join   := (r \/ s)* = r* \/ s*;
Axiom Ctrvariance := (r . s)* = s* . r*;
Axiom Interface   := r* . ~(r . s) <= ~s;

Axiom Fork        := r /*\ s = (Pi1* . r) /\ (Pi2* . s);
Axiom Fork_conv   := (r/*\s)* . (t/*\u) = (r* . t)/\(s* . u);
Axiom Fork_cancel := Pi1 /*\ Pi2 <= Id;

Axiom Pi1_def     := Pi1 = (Id /*\ Top)*;
Axiom Pi2_def     := Pi2 = (Top /*\ Id)*;
Axiom Prod_def    := r >< s = (r . Pi1) /*\ (s . Pi2)
```

The last three axioms are just definitions. They do not enrich the axiomatization and can be removed provided that the definition is used instead.

An identity $s \approx t$ is valid in $\mathcal{E}_{\text{Fork}}$ ($\mathcal{E}_{\text{Fork}} \vdash s \approx t$) if and only if it is derivable from $\mathcal{E}_{\text{Fork}}$ using the inference rules presented in Section 3.2 (reflexivity, symmetry, transitivity, substitution and closure under function symbols).

Inequations. Although some axioms appear as inequations rather than equations, we should recall from Section 4.3 that an inequation of the form `r <= s` is a shorthand for an equation of the form `r /\ s = r` (or equivalently `r \/ s = s`). Thus, we will also call equations (or identities) to inequations.

Reduction relation. The reduction relation inferred from $\mathcal{E}_{\text{Fork}}$ is defined as $s \rightarrow_{\mathcal{E}_{\text{Fork}}} t$ if and only if exists an identity $(l \approx r) \in \mathcal{E}_{\text{Fork}}$ such that $s[l'] = \sigma(l)$ and $t = s[\sigma(r)]$, for a substitution σ.

Interpretation. The semantic function $\mathcal{C}^{Term}$ defined in Chapter 7 is an interpretation of syntactical terms of *Galois* into fork algebra terms. Using this interpretation function, we can map every identity of $\mathcal{E}_{\text{Fork}}$ into an axiom of a fork algebra; thus, a fork algebra (`FA`) satisfies every identity of $\mathcal{E}_{\text{Fork}}$, and consequently, $\texttt{FA} \models \mathcal{E}_{\text{Fork}}$.

Semantic consequence. An identity $s \approx t$ is a semantic consequence of $\mathcal{E}_{\text{Fork}}$ ($\mathcal{E}_{\text{Fork}} \models s \approx t$) if and only if all models of $\mathcal{E}_{\text{Fork}}$ satisfy it. As we have seen above, fork algebras are a model of $\mathcal{E}_{\text{Fork}}$. What about other models of $\mathcal{E}_{\text{Fork}}$ which may arise if a different definition for $\mathcal{C}^{Term}$ is used? Clearly, any other model of $\mathcal{E}_{\text{Fork}}$ would obey the axioms of fork algebras. Thus, an identity of $\mathcal{E}_{\text{Fork}}$ which holds for fork algebras holds for any other model of $\mathcal{E}_{\text{Fork}}$; the converse may not be true.

Equational theory. The equational theory $\approx_{\mathcal{E}_{\text{Fork}}}$ induced by $\mathcal{E}_{\text{Fork}}$ is defined as

$$\approx_{\mathcal{E}_{\text{Fork}}} \quad \overset{def}{=} \quad \{\ s, t \in \text{Terms} :\ \mathcal{E}_{\text{Fork}} \models s \approx t\ :\ (s, t)\}$$

When we use the standard interpretation of $\mathcal{E}_{\text{Fork}}$ as fork algebras, $\approx_{\mathcal{E}_{\text{Fork}}}$ is equivalent to equality of fork algebra terms.

Word problem and decidability. As we have seen in Section 3.2, Birkhoff's theorem ensures that $\mathcal{E}_{\text{Fork}} \models s \approx t$ if and only if $\mathcal{E}_{\text{Fork}} \vdash s \approx t$, i.e., $\leftrightarrow_{\mathcal{E}_{\text{Fork}}}$ and $\approx_{\mathcal{E}_{\text{Fork}}}$ coincide. Thus, deciding if two fork algebra terms are equal $s = t$ reduces to the word problem, i.e., if it is possible to transform the term s into the term t using the reduction relation $\rightarrow_{\mathcal{E}_{\text{Fork}}}$. This means that a term rewriting system can be used in *Galculator* to prove equalities (theorems) in fork algebras. However, since the word problem is, in general, undecidable, *Galculator* is not able to prove all equalities of fork algebras.

From *Galois* to binary relations. In Section 4.4 we have seen the equivalence between syntactic derivations in fork algebras and properties holding for binary relations. Thus, by the above results, in *Galculator* we can syntactically derive any property that holds for binary relations; conversely, any valid derivation corresponds to a true property of binary relations.

8.1.2 Theories

In first-order logic, (first-order) theories are defined by adding specific axioms to the logical axioms. The language of the theory extends that of logic by defining non-logical symbols with a given signature.

Galois allows for the specification of theories based on fork algebras. Each theory defines its own symbols and axioms and induces an equational system $\mathcal{E}_{\text{Theory}}$.

Definitions. Definitions introduce distinguished relation symbols (identifiers) which serve to denote specific rather than arbitrary relations. In *Galois*, definitions are either opaque or transparent:

Opaque. Opaque definitions only introduce identifiers and types (signatures) for distinguished relations; an explicit definitions is not provided.

Transparent. Transparent definitions provide an explicit definition using a fork algebra term to an identifier and a type. Thus, they introduce an identity to the equational system, i.e., from a definition of kind `Ident : type := term` we get an identity $\mathcal{E}_{\text{Theory}} \cup \{Ident \approx term\}$.

Axioms. In *Galois*, the definition of an axiom of the form `Axiom A := a = b` concerning some theory introduces an identity $\mathcal{E}_{\text{Theory}} \cup \{a \approx b\}$. The user is responsible for ensuring the consistency of the axiomatization.

Equational system of *Galculator*. The equational system $\mathcal{E}_{\text{Galc}}$ of the *Galculator* is the conjunction of the equational system of fork algebras $\mathcal{E}_{\text{Fork}}$ with the equational system of all the defined theories $\mathcal{E}_{\text{Theory}}$, i.e., $\mathcal{E}_{\text{Galc}} = \mathcal{E}_{\text{Fork}} \cup \mathcal{E}_{\text{Theory}}$.

8.2 Putting Galois connections and fork algebras together

In this section, all the ingredients will be put together to form a consistent theory where proofs can be conducted. The main idea is to take orders and functions as particular cases of relations and accommodate them in the point-free fork algebra calculus [Silva et al., 2009].

As presented in Chapter 4, functions and orders are binary relations with certain properties. From the completeness result enunciated in Section 4.4 we know that an equivalence between the properties of binary relations and abstract fork algebra operations exists. Thus, functions and orders can be represented as fork algebra terms provided that their specific properties are taken as hypothesis. Then, the point-free transform is used to express Galois connections and indirect equality without variables using fork algebra formulæ.

Functions. When presenting theorems or statements to prove it is usual to just say that some relation f is a function. However, this implicitly adds both conditions about simplicity and totality of f to the hypothesis. Using the natural interpretation of binary relations as fork algebra terms presented in Section 4.4, simplicity and totality are expressed, respectively, as $f \circ f^{\cup} \subseteq id$ and $id \subseteq f^{\cup} \circ f$. Therefore, a function f in *Galois* implicitly adds two identities to the equational system $\mathcal{E} \cup \{f \circ f^{\cup} \subseteq id,\ id \subseteq f^{\cup} \circ f\}$.

The identity relation is also a function, as can be easily verified. Moreover, functions are closed under (relation) composition with the identity as unit, forming a monoid [Bird and de Moor, 1997]. The other fork algebra operations are not, in general, closed for functions.

Sections. The absence of variables in point-free representations greatly simplifies the calculus and the implementation of a proof engine. However, as it was introduced in Chapter 5, most interesting examples of Galois connections arise as sections of binary functions. This leads to a question: how to introduce sections of functions in fork algebras?

Our solution is a trade-off between simplicity and the purity of the point-free style. We introduce two sectioning operators, one for left sections and another for right sections, that take binary functions and turn them into unary functions by fixing one of the arguments. We denote the left and right sections of a binary function f by ${}_af$ and f_b, respectively. Sometimes, such as in the case of some associativity laws, both arguments must be fixed and the function becomes a constant: we introduce another operator to handle this case and denote it by ${}_af_b$.

The frozen arguments are constants and should be regarded as indexes. Thus, functions f_a and f_b are different because they have a different index, while functions with the same index are equal. This introduces some kind of name semantics for indexes that makes the implementation slightly more complicated but it is a fair compromise between power and simplicity. Moreover, since ${}_af_b$ is a constant, it can only appear where constants can, i.e., as a section of a binary function.

Orders. Indirect equality holds when working at least with partial orders. Like in the case of functions, we are implicitly adding the conditions about the reflexivity ($id \subseteq \sqsubseteq$), transitivity ($\sqsubseteq \circ \sqsubseteq \subseteq \sqsubseteq$) and antisymmetry ($\sqsubseteq \cap \sqsubseteq^{\cup} \subseteq id$) of an order $\sqsubseteq$ to the hypothesis of a statement. Like in the case of functions, an order $\sqsubseteq$ in *Galois* implicitly adds three identities to the equational system:

$$\mathcal{E} \cup \{id \subseteq \sqsubseteq,\ \sqsubseteq \circ \sqsubseteq \subseteq \sqsubseteq,\ \sqsubseteq \cap \sqsubseteq^{\cup} \subseteq id\}\ .$$

The identity relation is also a partial order. It corresponds to the equality ordering which relates objects when they are equal. Moreover, partial orders are closed under converse (this corresponds to the dual partial order).

Galois connections. Having established how functions and orders related with fork algebras, let us express Galois connections as point-free equalities. It is easy to see that the application of the PF transform for related functional results (4.52) to both sides of the definition of a Galois connection (5.1) yields, for all suitably typed a and b,

$$a(f^{\cup} \circ \sqsubseteq_{\mathcal{B}} \circ id)b \quad \Leftrightarrow \quad a(id^{\cup} \circ \sqsubseteq_{\mathcal{A}} \circ g)b$$

which leads to PF relational *equality*

$$f^{\cup} \circ \sqsubseteq_{\mathcal{B}} \quad = \quad \sqsubseteq_{\mathcal{A}} \circ g \tag{8.1}$$

once variables are removed (and also because the identity function id is its own converse and the unit of composition). So we can deal with logical expressions involving adjoints of Galois connections by equating the corresponding PF-terms without variables.

From a definition in *Galois* of the kind `Galois Foo := f1 f2 o1 o2`, where `f1`, `f2`, `o1` and `o2` are correctly defined, several implicit identities are added to $\mathcal{E}$: the function hypothesis for `f1` and `f2`; the order hypothesis for `o1` and `o2`; and an identity of the form of (8.1) ensuring the existence of the given Galois connection.

Since definitions of Galois connections are taken as axioms, the proof obligation that it really establishes a Galois connection is left to the user.

8.3 Indirect equality

Indirect equality is a powerful tool often used in proofs of lattice theory. However, its applications are often overlooked in other domains. In this section, we provide the basics about indirect equality and inequality, as well as the respective point-free versions.

Indirect inequality. Before introducing indirect equality, we will start by discussing the weaker and somewhat more intuitive notion of indirect inequality. Suppose that $(\mathcal{A}, \sqsubseteq)$ is a *preordered* set, and we want to prove that $a \in \mathcal{A}$ and $b \in \mathcal{A}$ are related, i.e., $a \sqsubseteq b$. When doing it directly is difficult, we can take advantage of a simple result arising from the reflexivity and transitivity of $\sqsubseteq$ [Dijkstra, 1991]: if $a \sqsubseteq b$ then all elements of $\mathcal{A}$ smaller or equal to a are also smaller or equal to b. The converse implication is also valid. This is called *indirect inequality* principle, and has two equivalent formulations: for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$,

$$a \sqsubseteq b \quad \Leftrightarrow \quad \langle \forall\, x :: b \sqsubseteq x \Rightarrow a \sqsubseteq x \rangle \tag{8.2}$$

$$a \sqsubseteq b \quad \Leftrightarrow \quad \langle \forall\, x :: x \sqsubseteq a \Rightarrow x \sqsubseteq b \rangle \tag{8.3}$$

Indirect equality. Let $(\mathcal{A}, \sqsubseteq)$ be a *partial order* instead. Anti-symmetry of $\sqsubseteq$ is important to establish equalities, usually by mutual inclusion, i.e., for $a \in \mathcal{A}$ and $b \in \mathcal{A}$, $a = b$ if and only if $a \sqsubseteq b$ and $b \sqsubseteq a$. Combining this with indirect inequality we have

$$\begin{array}{cl}
 & a = b \\
\Leftrightarrow & \qquad \{ \text{ Anti-symmetry of } \sqsubseteq. \ \} \\
 & a \sqsubseteq b \ \wedge \ b \sqsubseteq a \\
\Leftrightarrow & \qquad \{ \text{ Indirect inequality (8.2) and (8.3). } \} \\
 & \langle \forall\, x :: x \sqsubseteq a \Rightarrow x \sqsubseteq b \rangle \ \wedge \ \langle \forall\, x :: x \sqsubseteq b \Rightarrow x \sqsubseteq a \rangle \\
\Leftrightarrow & \qquad \{ \text{ Quantification rules. } \}
\end{array}$$

$$\begin{array}{ll} & \langle \forall\, x :: x \sqsubseteq a \Rightarrow x \sqsubseteq b \ \wedge\ x \sqsubseteq b \Rightarrow x \sqsubseteq a \rangle \\ \Leftrightarrow & \quad \{ \text{ Mutual implication (anti-symmetry of logical implication). } \} \\ & \langle \forall\, x :: x \sqsubseteq a \Leftrightarrow x \sqsubseteq b \rangle \end{array}$$

Thus, we get the formulation of the *indirect equality* principle (the second one is equivalent): for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$,

$$a = b \quad \Leftrightarrow \quad \langle \forall\, x :: x \sqsubseteq a \Leftrightarrow x \sqsubseteq b \rangle \tag{8.4}$$

$$a = b \quad \Leftrightarrow \quad \langle \forall\, x :: a \sqsubseteq x \Leftrightarrow b \sqsubseteq x \rangle \tag{8.5}$$

Therefore, a proof by indirect equality can be seen as the combination of two indirect inequality proofs by mutual inclusion.

Point-free indirect equality. The *indirect equality* rule can also be formulated without variables thanks to the PF-transform. Let us consider two functions $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{B} \xleftarrow{g} \mathcal{A}$, where $(\mathcal{B}, \preceq)$ is a *partial order* and $\mathcal{A}$ is a *set*. That

$$f = g \quad \Leftrightarrow \quad \preceq \circ f = \preceq \circ g \tag{8.6}$$

$$f = g \quad \Leftrightarrow \quad f^{\cup} \circ \preceq = g^{\cup} \circ \preceq \tag{8.7}$$

instantiate indirect equality can be easily checked by putting variables back via (4.52) (the PF transform for related functional results).

Point-free indirect inequality. Using the same reasoning, the *indirect inequality* rule also has a point-free formulation

$$f \mathrel{\dot{\preceq}} g \quad \Leftrightarrow \quad \preceq \circ f \subseteq \preceq \circ g \tag{8.8}$$

$$f \mathrel{\dot{\preceq}} g \quad \Leftrightarrow \quad g^{\cup} \circ \preceq \subseteq f^{\cup} \circ \preceq \tag{8.9}$$

where $\dot{\preceq}$ is the lifted order for functions. As in the point-wise definition, $\preceq$ is only required to be a *preorder.*

Indirect inequality and lifted orders. We should notice that Equation (8.8) is equivalent to Equation (4.56) defining the point-free lifted order for relations, i.e.,

$$f \mathrel{\dot{\preceq}} g \quad \overset{def}{\Leftrightarrow} \quad f \subseteq \preceq \circ g$$

The proof by mutual implication of this equivalence exploits the fact that $\preceq$ is a preorder and that composition is monotonic:

1. Implication $\preceq \circ f \subseteq \preceq \circ g \;\Rightarrow\; f \subseteq \preceq \circ g$ holds because $\preceq$ is reflexive, i.e., $id \subseteq \preceq$, and composition is monotonic;

2. Implication $f \subseteq \preceq \circ g \;\Rightarrow\; \preceq \circ f \subseteq \preceq \circ g$ holds because:

$$\begin{array}{ll}
& f \subseteq \preceq \circ g \\
\Rightarrow & \qquad \{ \text{ Monotonicity of composition. } \} \\
& \preceq \circ f \subseteq \preceq \circ \preceq \circ g \\
\Rightarrow & \qquad \{ \text{ Transitivity of } \preceq\text{, i.e., } \preceq \circ \preceq \subseteq \preceq\text{, and monotonicity of composition. } \} \\
& \preceq \circ f \subseteq \preceq \circ g
\end{array}$$

Application. Equation (8.6) is not a point-free equality but an equivalence between two point-free equalities. Like the substitution rule, it is a meta-level result and should be used as an inference rule of the system.

The usual application of indirect equality uses also the transitivity of equality. For instance, when trying to establish an equality $f = g$, a partial order is composed with one of the functions, e.g., $\sqsubseteq \circ f$. Then, the derivation follows by applying other laws using the substitution rule:

$$\begin{array}{ll}
& \sqsubseteq \circ f \\
= & \qquad \{ \dots \} \\
& \dots \\
= & \qquad \{ \dots \} \\
& \sqsubseteq \circ g
\end{array}$$

until the expression $\sqsubseteq \circ g$ is obtained. By transitivity of equality, $\sqsubseteq \circ f = \sqsubseteq \circ g$, and thus, by indirect equality, we conclude that $f = g$.

8.4 Proofs in point-free style

Whole division implementation. For example, let us see how the calculation in the introduction (Section 1.1.1) is actually performed inside the *Galculator*: first of all, equations (1.3), (1.4) become families of PF-equalities

$$(\times y)^{\cup} \circ \leqslant \;=\; \leqslant \circ (\div y) \tag{8.10}$$

$$(-b)^{\cup} \circ \leqslant \;=\; \leqslant \circ (+b) \tag{8.11}$$

indexed by y (assuming $y \neq 0$) and b, respectively, where $(\times y)$, $(\div y)$, $(-b)$ and $(+b)$ are the right section functions of multiplication, division, subtraction and addition, respectively. We deviate from the convention of using subscripts to denote sections of functions since, in this example, it improves readability. Then the following series of *equalities* are calculated:

$$\begin{array}{ll}
 & \leqslant \circ (\div y) \\
= & \quad \{ \text{ Shunting (8.10) assuming } y > 0. \} \\
 & (\times y)^{\cup} \circ \leqslant \\
= & \quad \{ \text{ Cancellation, thanks to (8.11) — steps omitted. } \} \\
 & ((-y) \circ (\times y))^{\cup} \circ \leqslant \circ (-y) \\
= & \quad \{ \text{ Distributivity. } \} \\
 & ((\times y) \circ (-1))^{\cup} \circ \leqslant \circ (-y) \\
= & \quad \{ \text{ Distributivity of converse through composition. } \} \\
 & (-1)^{\cup} \circ (\times y)^{\cup} \circ \leqslant \circ (-y) \\
= & \quad \{ \text{ Shunting (8.10). } \} \\
 & (-1)^{\cup} \circ \leqslant \circ (\div y) \circ (-y) \\
= & \quad \{ \text{ Shunting (8.11). } \} \\
 & \leqslant \circ (+1) \circ (\div y) \circ (-y)
\end{array}$$

From this, the *Galculator* uses indirect equality to infer equality:

$$(\div y) \;=\; (+1) \circ (\div y) \circ (-y)$$

Simple property about whole division. The corresponding point-free calculation of the proof of Section 1.1.2 is

$$
\begin{array}{ll}
& \leqslant \circ ((\div c) \circ (\div b)) \\
= & \quad \{ \text{ Associativity of composition. } \} \\
& (\leqslant \circ (\div c)) \circ (\div b) \\
= & \quad \{ \text{ Shunting (8.10). } \} \\
& ((\times c)^{\cup} \circ \leqslant) \circ (\div b) \\
= & \quad \{ \text{ Associativity of composition. } \} \\
& (\times c)^{\cup} \circ (\leqslant \circ (\div b)) \\
= & \quad \{ \text{ Shunting (8.10). } \} \\
& (\times c)^{\cup} \circ ((\times b)^{\cup} \circ \leqslant) \\
= & \quad \{ \text{ Associativity of composition. } \} \\
& ((\times c)^{\cup} \circ (\times b)^{\cup}) \circ \leqslant \\
= & \quad \{ \text{ Contravariance of converse and composition. } \} \\
& ((\times b) \circ (\times c))^{\cup} \circ \leqslant \\
= & \quad \{ \text{ Associativity of multiplication. } \} \\
& (\times (b \times c))^{\cup} \circ \leqslant \\
= & \quad \{ \text{ Shunting (8.10). } \} \\
& \leqslant \circ (\div (b \times c)) \\
\therefore & \quad \{ \text{ Indirect equality equality. } \} \\
& (\div c) \circ (\div b) = \div (c \times b)
\end{array}
$$

This proof in point-free style is equivalent to the previous one, although a little bit longer because the explicit use of the associativity steps. It also illustrates the proof format of *Galculator*: simple equational steps provided with clear justifications. In fact, this proof, tallies the example proof script given in Section 1.2.

8.5 Summary

This chapter discussed the theoretical foundations of the *Galculator*. The equational theory used by *Galculator* comes from the axioms of fork algebras together with the axioms of the specific theory being used. The Birkhoff's theorem ensures that a term rewriting system can be used to prove theorems in fork algebras, although this is, in general, an undecidable problem. Galois connections can be integrated with fork algebras by considering their point-free (equational) definition. Moreover, the point-free definition can be extended to indirect equality as well. We have shown how all concepts can be integrated together and used in calculational proofs. The examples presented in the introduction (Sections 1.1.1 and 1.1.2) were revisited and rewritten in this point-free calculational style which serves as foundation of the *Galculator*.

Chapter 9

Functional prototype

This chapter describes the *Galculator* prototype, starting from its basic design principles and general architecture and proceeding to the technical details of the implementation.

9.1 Design principles

Galois connections are the *Galculator*'s main building block. They are combined as needed, forming arbitrarily complex new connections from existing ones. From each Galois connection the *Galculator* derives its properties as given by Figure 5.1 which, together with fork algebra laws[1] and algebraic properties of the particular domain of the problem being solved, form the set of *laws* of the system. In order to represent all these concepts (Galois connections, fork algebra, particular domain theory), several embedded DSLs are defined. These embedded DSLs are *implementations* using GADTs of the *Galois* DSL *specified* in the previous chapters.

Galculator proofs are transformations of the abstract representation of the equality being proved. These transformations are made according to the equalities enabled by the laws of the system. However, laws are objects arising from the theoretical level; they cannot be applied to representations. Thus, a mechanism is defined for deriving functional applications of the available laws in the form of *rewrite rules*. The application of such rules is performed by a strategic term rewrite system (TRS).

Basic rewrite strategies can be combined in order to build more complex ones, according to the complexity of the problem. Moreover, with the same set of rules

[1]In this chapter, we shall refer to equalities in the theory level as laws.

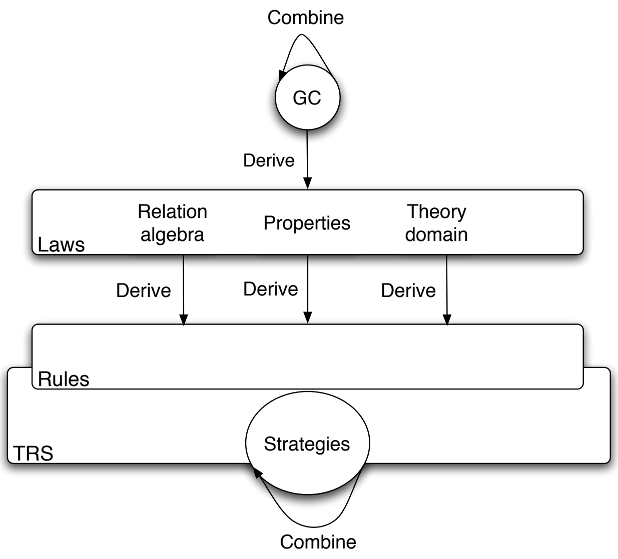


Figure 9.1: Design principles of the *Galculator* prototype.

several different rewrite systems can be easily built and tried.

A summary of the components and design principles which guide the *Galculator* prototype is given in Figure 9.1.

9.2 Architecture

The *Galculator* is divided in several logical modules. Below we give an overview of these before presenting a more technical description.

Interpreter. The command line interpreter provides for interactive user interfacing. Several options are offered: loading modules, exploiting Galois connection algebra, checking expressions and doing proofs. Currently, these are performed in interactive proof mode. At each step, the user can choose a rule derived from the set of laws available from the system. Rules can be applied using the strategies built from the combinators provided by the term rewriting system. The system offers hints about the applicable rules in the current proof step. At the end, a complete proof log, with all equational steps and justifications is made available.

Parser. Several domain specific languages (DSL) are available in order to express the concepts in use: Galois connections, relations, orders, functions and so on. For each one a parser was implemented using parsing combinators [Leijen and Meijer, 2001]. This technique makes it easy to use a DSL inside another simply by calling the respective parsing combinators.

Type inference. Types are useful in finding errors, not only in programs but also in proofs: they give insight in some misleading details that are often overlooked. The *Galculator* prototype is a typed environment with its own type system, based on the *Haskell* type system using a type representation. Altogether, the user is released from having to provide explicit types in expressions.

The *Galculator* type system supports parametric polymorphism. The type representation is extended with support for type variables. However, it is not possible to rely on the type system of the host language alone in order to account this addition. Thus a unification mechanism on type variables has been implemented based on the Hindley-Milner algorithm [Milner, 1978]. Polymorphism is useful for deriving the so-called *free-theorems* of functions [Wadler, 1989; Backhouse and Backhouse, 2004], a kind of commutative property enjoyed by polymorphic functions solely inferred from their types.

Term rewriting system. The core of *Galculator* is its term rewriting system (TRS), whose rules (derived from the theory explained in Chapter 5 and Chapter 8) are applied to terms in order to build proofs. The system uses the flexibility of strategies and their combinatorial properties in order to build more complex proof strategies. Moreover, since the whole system is typed, the TRS is also typed, allowing for type directed rewriting rules.

Property inference. Galois connections are specified by their types (sets on which they are defined), the pre-orders involved and the adjoint functions. This component derives the properties stated in Figure 5.1 from the starting specification and adds them to the system.

Rule inference. The equational laws expressed in our representation are purely declarative; they cannot be used in rewriting because they are not functions. Thus, we developed a rule inference engine which takes an equational expression and returns a rewrite function usable by the TRS. This component ensures most of the genericity of *Galculator*.

9.3 Representation

The concepts used in the system (relations, functions, orders, Galois connections) are represented by GADTs. As mentioned in Section 2.4.3, GADTs naturally induce an associated language. Thus, in fact, with GADTS we are defining very small DSLs which integrate with each other.

Types. The following data type encompasses the basic types in the domains we want to use our tool:

```
data Type a where
  One    :: Type One
  Bool   :: Type Bool
  Char   :: Type Char
  String :: Type String
  Int    :: Type Int
  Float  :: Type Float
  List   :: Type a → Type [a]
  Set    :: Type a → Type (Set a)
  Maybe  :: Type a → Type (Maybe a)
  · × ·  :: Type a → Type b → Type (a, b)
  · + ·  :: Type a → Type b → Type (a + b)
  · ⇀ ·  :: Type a → Type b → Type (a ⇀ b)
  Ord    :: Type a → Type (PO a)
  Fun    :: Type a → Type b → Type (a ← b)
  Rel    :: Type a → Type b → Type (a ∼ b)
  GC     :: Type a → Type b → Type (GC a b)
  Expr   :: Type a → Type (Expr a)
type One   = ()         -- Unitary type
type b ← a = a → b      -- Functions
data b ∼ a              -- Relations
data PO a               -- Partial orders
data GC b a             -- Galois connections
data Expr a             -- Expressions
```

We deviate from the usual *Haskell* representation for functions (we use $\leftarrow$ instead of $\rightarrow$) because the right to left arrow is visually more consistent with function composition which is the main connector of our point-free calculus. Of special interest are the PO and GC data types which will allows us to work with partial orders and Galois connections, respectively. Thus, type $PO\ a$ represents a type a which is equipped with a partial order forming a poset $(a, \sqsubseteq_a)$. Type $GC\ b\ a$ represents a Galois connection $(b, \sqsubseteq_b) \xleftarrow{(,)} (a, \sqsubseteq_a)$ between posets $(b, \sqsubseteq_b)$ and $(a, \sqsubseteq_a)$.

This representation works with a closed universe of types. So, parametric polymorphism is not possible. In type polymorphism type variables which range over the universe of types are allowed. In order to deal with parametric polymorphism we must enrich our type representation with another constructor which represents type variables:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Var} \\
\mathbf{type}\ \mathit{Variable} = \mathit{String} \\
\mathbf{data}\ \mathit{Type}\ a\ \mathbf{where} \\
\quad \ldots \\
\quad \mathit{TVar} :: \mathit{Variable} \rightarrow \mathit{Type}\ \mathit{Var}
\end{array}
$$

This means that all type variable representations have the same type (Var). Using $\mathit{Type}\ a$ does not work because it would not be possible to define a type equality mechanism over it. The drawback of the use of Var is that *Haskell* type inference mechanism fails to unify Var with any type, like it would do with a normal *Haskell* type variable. Thus, we cannot solely rely on the *Haskell* type-system and thus have to create our own unification mechanism (Section 9.4).

Combinators. The point-free calculus presented in Section 4.5 presents a set of relational combinators. The relation calculus is the basis of all proofs once Galois connections are encoded in the point-free style (recall Equation 8.1). Thus, we represent the combinators of Section 4.5 using constructors of a GADT. Here is how they are defined:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ R\ r\ \mathbf{where}} \\
\quad \cdot = \cdot & :: R\ a \rightarrow R\ a \rightarrow R\ (\mathit{Expr}\ a) \\
\quad \cdot \subseteq \cdot & :: R\ a \rightarrow R\ a \rightarrow R\ (\mathit{Expr}\ a) \\
\quad \mathit{Var} & :: \mathit{Variable} \rightarrow R\ (b \sim a)
\end{array}
$$

$$\begin{array}{ll}
id & :: R\ (a \sim a) \\
\bot & :: R\ (b \sim a) \\
\top & :: R\ (b \sim a) \\
\neg\, \cdot & :: R\ (b \sim a) \rightarrow R\ (b \sim a) \\
\cdot^{\cup} & :: R\ (b \sim a) \rightarrow R\ (a \sim b) \\
\cdot \cap \cdot & :: R\ (b \sim a) \rightarrow R\ (b \sim a) \rightarrow R\ (b \sim a) \\
\cdot \cup \cdot & :: R\ (b \sim a) \rightarrow R\ (b \sim a) \rightarrow R\ (b \sim a) \\
\cdot \circ_{\cdot} \cdot & :: \mathit{Type}\ b \rightarrow R\ (c \sim b) \rightarrow R\ (b \sim a) \rightarrow R\ (c \sim a) \\
\cdot \nabla \cdot & :: R\ (b \sim a) \rightarrow R\ (c \sim a) \rightarrow R\ ((b, c) \sim a) \\
\cdot \times \cdot & :: R\ (b \sim a) \rightarrow R\ (d \sim c) \rightarrow R\ ((b, d) \sim (a, c)) \\
\pi_1 & :: R\ (a \leftarrow (a, b)) \\
\pi_2 & :: R\ (b \leftarrow (a, b)) \\
\ldots &
\end{array}$$

The meaning of each operator should be clear since we use *lhs2TeX* to mirror the mathematical notation introduced in previous chapters. Note the use of a type annotation $\mathit{Type}\ b$ in the definition of the composition operator. This is necessary during traversals of the representation in order to get the common type back. This type is existentially quantified and otherwise it could not be known.

Type lifting. Functions and orders are particular cases of relations. Thus, it should be possible to use them wherever a relation can; however their types do not match. Two embeddings are defined for this purpose:

$$\begin{array}{l}
\omega_o(\cdot) :: R\ (PO\ a) \ \rightarrow R\ (a \sim a) \\
\omega_f(\cdot) :: R\ (b \leftarrow a) \rightarrow R\ (b \sim a) \\
\ldots
\end{array}$$

Expression $\omega_o(o)$ turns order o into a relation; $\omega_f(f)$ makes the function f a relation.

Functions. The representation of functions is given by a number of constructors:

$$\begin{array}{ll}
\mathit{FunctionVar} & :: \mathit{Variable} \rightarrow R\ (b \leftarrow a) \\
\mathit{FunctionId} & :: R\ (a \leftarrow a) \\
\mathit{FunctionComp} & :: \mathit{Type}\ b \rightarrow R\ (c \leftarrow b) \rightarrow R\ (b \leftarrow a) \rightarrow R\ (c \leftarrow a) \\
\ldots &
\end{array}$$

$\mathit{FunctionVar}\ \ a$ denotes a function variable with name a; FId denotes the identity function; $\mathit{FComp}\ t\ f\ \ g$ denotes the composition of functions f and g with intermediate type t.

Sections. As explained in Chapter 5, many adjoints arise as sections of functions. We provide the following sectioning operators:

$$\begin{array}{lll}
\cdot_{\cdot::\cdot} & ::\ \mathit{Type}\ c \to R\ (a \leftarrow (b, c)) \to R\ c \to R\ (a \leftarrow b) & \text{-- Right section} \\
{}_{\cdot::\cdot}\,\cdot & ::\ \mathit{Type}\ b \to R\ (a \leftarrow (b, c)) \to R\ b \to R\ (a \leftarrow c) & \text{-- Left section} \\
\mathit{Const} & ::\ \mathit{Constant} \to R\ a & \\
\cdot_{(\cdot::\cdot,\cdot::\cdot)} & ::\ \mathit{Type}\ b \to \mathit{Type}\ c & \\
 & \to R\ (a \leftarrow (b, c)) \to \mathit{Constant} \to \mathit{Constant} \to R\ a & \\
\ldots & &
\end{array}$$

Given a binary function f with the right type and v a value with type t, ${}_{v::t}f$ denotes the left section of f; $f_{v::t}$ denotes the right section of f. As happens with the composition operator, type annotations are again needed in order to retain existentially quantified types. $\mathit{Const}\ \ a$ represents a constant of name a and $f_{(c1::t1,c2::t2)}$ represents simultaneous right and left sections of a binary function f, whose sections $c1$ and $c2$ have, respectively, types $t1$ and $t2$.

We should notice the link between the sectioning operators at the function representation level and sections at the functional level. Operators ${}_{vb::t}f$ and $f_{vc::t}$ are representations of the *Haskell* sectioning operations for uncurried functions described in Section 2.4.1. Let us consider the corresponding objects at the functional level: $f :: R\ (a \leftarrow (b, c))$ is the representation of a function of type $f' :: (b, c) \to a)$; $vb :: R\ b$ is the representation of a value $vb' :: b$; and $vc :: R\ \ c$ is the representation of a value $vc' :: c$. Therefore, ignoring type annotations, the left section operator ${}_{vb::t}f$ represents $\mathit{curry}\ f'\ \ vb'$ and the right section operator $f_{vc::t}$ represents $\mathit{flip}\ \ (\mathit{curry}\ f')\ \ vc'$.

Orders. The representation of orders is given by a number of constructors:

$$\begin{array}{l}
\mathit{OrderVar}\ \ ::\ \mathit{Variable} \to R\ (PO\ \ a) \\
\mathit{OrderId}\ \ \ \ ::\ R\ (PO\ \ a) \\
\mathit{OrderConv} ::\ R\ (PO\ \ a) \to R\ (PO\ \ a) \\
\ldots
\end{array}$$

$\mathit{OrderVar}\ \ a$ denotes an order variable with name a; $\mathit{OrderId}$ denotes the identity order; $\mathit{OrderConv}\ o$ denotes the converse of order o.

Galois connections. Our representation of Galois connections puts together two adjoint functions and two partial orders, suitably typed. Moreover, the operations of Galois connection algebra are also provided:

$$\begin{array}{ll}
\mathit{GCVar} & :: \mathit{Variable} \to R\ (\mathit{GC}\ b\ a) \\
\mathit{GCId} & :: R\ (\mathit{GC}\ a\ a) \\
\mathit{GCComp} & :: \mathit{Type}\ b \to R\ (\mathit{GC}\ c\ b) \to R\ (\mathit{GC}\ b\ a) \to R\ (\mathit{GC}\ c\ a) \\
\mathit{GCConv} & :: R\ (\mathit{GC}\ b\ a) \to R\ (\mathit{GC}\ a\ b)
\end{array}$$

$\mathit{GCVar}\ \ a$ denotes a Galois connection variable with name a; GCId represents the identity Galois connection; $\mathit{GCComp}\ \ g\ \ g'$ represents the composition of Galois connections g and g' with appropriated types; and $\mathit{GCConv}\ \ g$ represents the converse connection of g.

9.4 Type equality and type unification

Type equality witness. An advantage of using explicit type representations is that equality can be computed at run-time, allowing for the introduction of dynamic typing mechanisms in a static environment [Baars and Swierstra, 2002]. For this, a GADT definition is used:

$$\mathbf{data}\ a = b\ \mathbf{where}\ \mathit{Eq} :: a = a$$

This is called a *witness type* because it can only be built if the types are equal. This is ensured by the type checker by analysing the types of the type indexes (recall Section 2.4.1). Moreover, thanks to this witness the type-checking mechanism can recognize values of the two types as interchangeable.

Type equality. Given two type representations, the teq function tries to build an Eq witness of their equality: if the types are equal the witness is returned; otherwise it fails. Using GADTs computing type representation equality reduces to computing syntactical equality between data constructors:

```
$teq :: MonadOr\ m \Rightarrow Type\ a \rightarrow Type\ b \rightarrow m\ (a = b)$
$teq\ Int\ Int = return\ Eq$
$teq\ (a \times b)\ (a' \times b') =$ do
  $Eq \leftarrow\ teq\ a\ a'$
  $Eq \leftarrow\ teq\ b\ b'$
  $return\ Eq$
...
$teq\ \_\ \_ = mzero$
```

However, implementing type equality over type variable representation leads to the question: when are two type variables of the same type? A type variable representation is just a placeholder, it can be replaced by another type representation. Using polymorphic type representations we cannot rely on the type checker in order to infer that types are equal. We have to implement a type unification mechanism which helps the type-checker to infer the types correctly.

Type unification. Type unification can be easily solved with a unification algorithm such as one presented in Section 3.2. The algorithm receives a system of equations stating the supposed equalities between types. If some of the equalities do not hold, e.g., trying to unify integers with Booleans, the algorithm fails. Otherwise, a set of substitutions is returned with mappings from variables into the type which they should be instantiated to. One property of this algorithm is that if it succeeds, it returns the most general unifier. Function

$$\eta :: MonadOr\ m \Rightarrow [Equation] \rightarrow m\ [Substitution]$$

implements this algorithm, where equations and substitutions are synonyms for the same type which is just a pair of type representations existentially quantified:

```
data Constraint where
  $\cdot :=: \cdot :: Type\ a \rightarrow Type\ b \rightarrow Constraint$
type Equation     = Constraint
type Substitution = Constraint
```

9.5 Parsing and type inference

Embedding DSLs implies that the user knows how to use the host language (writing modules, compiling files and so on). Thus, in order to allow the user to specify expressions in a textual format a parser was developed using the parsing combinators of the *Parsec* library [Leijen and Meijer, 2001]. However, as we shall see shortly, using GADTs implies the inference of the type of the expressions being parsed.

The first version of the *Galculator* prototype mixed parsing and type inference, as described by Silva and Oliveira [2008]. However, as the prototype evolved, this process was decoupled into several phases in order to deal with its increased complexity, as described in Section 9.5.3.

9.5.1 Fresh variable names

Prior to entering into the details of parsing and type inference, let us introduce a monadic construction to ensure production of fresh variable names. By *fresh* we mean an identifier that is unique in the system and that has never been used before.

We consider the concept of *stream* which can be seen as an infinite list of objects. A *Haskell* class is defined for representing streams

```
class Stream a v | a → v where
  headStr :: a → v
  tailStr :: a → a
```

This definitions uses a multi-parameter type class with a functional dependency $a \to v$. This means that a stream of type a uniquely determines the type of its objects v. Method $\mathit{headStr}$ returns the head of the stream while $\mathit{tailStr}$ returns the remaining of it.

Now we use Stream to define another class

```
class (Monad m, Stream s v) ⇒ MonadFresh s v m | m → s where
  getFresh :: m v
```

together with the implicit invariant that $\mathit{getFresh}$ should provide always different values. The functional dependency $m \to s$ states that a monad m uniquely determines the associated stream of objects s. Stream s ensures an infinite supply of objects.

The actual implementation of $\mathit{MonadFresh}$ in *Haskell* takes advantage of lazy evaluation and the capability of handling infinite lists.

9.5.2 First version

Note that in this section, the syntax of the parsed language is different from the specification of *Galois* and similar to the abstract syntax using GADTs. For each constructor of our representation a parsing combinator is defined. This combinatorial style makes it easy to embed DSLs inside others: all that is needed is to use the corresponding parser in a composable approach. While building ASTs using ADTs is almost straightforward, representations using GADTs pose some problems because their index type is only known at run-time: it is dependent on the input. This is circumvented by resorting to an existential data type to hide the type index

$$\mathbf{data}\ \mathit{Covert}\ t = \forall x.\ \mathit{Hide}\ (t\ x)$$

maintaining static safeness. Recall from Section 2.4.1 that *Haskell* notation resorts to the universal quantifier to introduce existential data types.

Covert is a common pattern [Sheard et al., 2005] where t can be parameterized with the data type we want to use. For instance, for building type representations:

$$\mathbf{type}\ \mathit{TypeBox} = \mathit{Covert}\ \mathit{Type}$$

This can be manipulated in a type-safe manner provided the encapsulated value never escapes the scope of its quantification.

However, when trying to parse relational representations, for instance, hiding the index type is not enough. If we define a data type to encapsulate the representation and a parsing combinator,

$$\mathbf{type}\ \mathit{RBox} = \mathit{Covert}\ R$$
$$\mathit{parseR} :: \mathit{Parser}\ \mathit{RBox}$$

wherever the result of the parsing function is used, for instance, by another parsing combinator in order to build a more complex term, the index type escapes from its scope and the compiler cannot ensure type safeness anymore.

The solution is to add an explicit type representation sharing the same index type of the expression representation, changing the type of parseR accordingly:

$$\mathbf{data}\ \mathit{Exists}\ \mathit{singleton}\ \mathit{term} = \forall t.\ \mathit{Exists}\ (\mathit{singleton}\ t)\ (\mathit{term}\ t)$$
$$\mathbf{type}\ \mathit{RType} = \mathit{Exists}\ \mathit{Type}\ R$$
$$\mathit{parseR} :: \mathit{Parser}\ \mathit{RType}$$

Although the exact index type is not known, the type-checker knows that it must reflect the type representation (because it is a singleton type), being sufficient to ensure static type safeness. Thus, for instance, the parsing combinator for the converse operator:

```
parseConv :: Parser RType
parseConv = do
  reserved "Conv"
  Exists (Rel t t') r ← parseR
  return (Exists (Rel t' t) (r∪))
```

Since explicit type annotations are needed, either the user has to provide these or the system has to infer them. Thanks to the unification mechanism, we just have to generate the equations. Polymorphic operators need to get fresh variable names in order to denote their type variable representation; other operators just have to be provided with the type representation corresponding to their types. An example of an operator that is polymorphic in its argument, but not in its result is *bang* (function that takes any value to the only inhabitant of the unitary type):

```
parseFBang :: Parser RType
parseFBang = do
  reserved "FBang"
  tid ← getFresh
  return (Exists (Fun One (TVar tid)) bang)
```

Function *getFresh* gets always fresh variables names from an infinite stream of identifiers due to lazy evaluation.

Unification is only needed when parsing relational combinators in which some variables must be equal. Composition is a good example of this situation:

```
parseComp :: Parser RType
parseComp = do
  reserved "Comp"
  Exists (Rel t3 t2) r1  ← parseR
  Exists (Rel t2b t1) r2 ← parseR
  subst       ← η [t2 :=: t2b]
  Hide t1' ← typeRewrite subst t1
  Hide t2' ← typeRewrite subst t2
```

$$\begin{aligned}
&\mathit{Hide}\ t3' \leftarrow \mathit{typeRewrite}\ \mathit{subst}\ t3\\
&r1' \qquad\ \leftarrow \mathit{safeCast}\ \mathit{subst}\ (\mathit{Rel}\ t3'\ t2')\ r1\\
&r2' \qquad\ \leftarrow \mathit{safeCast}\ \mathit{subst}\ (\mathit{Rel}\ t2'\ t1')\ r2\\
&\mathit{return}\ (\mathit{Exists}\ (\mathit{Rel}\ t3'\ t1')\ (r1' \circ_{t2'} r2'))
\end{aligned}$$

After parsing the two relational expressions $r1$ and $r2$ we have to make sure that they have a common type in order to be composable. Thus, a type equation is solved by unification and the set of type variable substitutions is applied using $\mathit{typeRewrite}$ (more details in Section 9.8). Next, the representation must also reflect type substitutions. Since constructors in GADTs retain the associated type information, a kind of type-safe "cast" is needed in order to reflect the new types. Function $\mathit{safeCast}$ is an embedding function where an expression of type r is transformed in an identical expression of type t, if types are compatible:

$$\mathit{safeCast} :: \mathit{MonadOr}\ m \Rightarrow [\mathit{Substitution}] \rightarrow \mathit{Type}\ t \rightarrow R\ r \rightarrow m\ (R\ t)$$

The need for the substitution list is justified by the relational combinators which have type annotations, like composition, where the substitutions have to be applied also for consistency.

Finally, the parsing function returns the newly built term with the respective type annotation. Since all computations are performed in a $\mathit{MonadOr}$ context, should any of them fail, the whole parsing function will fail.

9.5.3 Second version

The implementation described in the previous section was simplistic: its purpose was to illustrate how a GADT representation could be built from a textual definition. In fact, fusing parsing and type inference works for simple cases but can become a burden on maintenance when languages tend to grow. Moreover, in order to implement the whole language specified in Chapter 7 additional steps are needed. Therefore, the whole process was divided into four phases: parsing, verification, variable refreshing and type inference.

Parsing. The parsing phase still uses the *Parsec* library but takes advantage of its capability to build expression parsers. These overcome the traditional limitations of parsing combinators when dealing with left-recursive grammars.

The parser follows the specification of *Galois* given in Chapter 7. Following the usual approach, we take the abstract syntax annotations as constructors of an ADT in order to specify the AST of the language. This specification is very close to the DSL defined in the previous sections using GADTs. The difference is that the ADT representation does not store type information and does not enforce type correctness.

Verification. The specification of *Galois* requires that declarations of axioms, theorems or Galois connections are unique in a module. Furthermore, references must refer to declared identifiers.

The verification phase ensures that both these requirements are met; moreover, definitions are replaced for their references.

Variable refreshing. Several properties of rewriting systems are only valid when working with ground terms, i.e., without variables. The problem is that rules can capture variables of the term. This is avoided by ensuring that both sets of variables are disjoint [Gnaedig and Kirchner, 2009].

Since rewrite rules in *Galculator* are derived from equalities (as we shall see briefly), the problem of variable capture arises even among terms. The solution is to ensure that all variable names in a certain scope are fresh with respect to the whole module. For instance, every occurrence of a variable named `"a"` in an axiom declaration is replaced by a fresh variable. Occurrences of the same variable `"a"` in another axiom declaration are replaced by a different variable.

Type inference. Having ensured that the abstract representation meets the specification requirements, type inference can be performed so as to build another abstract representation based on GADTs. The type inference process follows the typing rules of *Galois* presented in Chapter 7.

Compared to the first version, which did not handle variables, the second implementation is also slightly more complex. A monomorphic restriction is applied to variables, i.e., their type is the same in every occurrence.

An inference function is defined for each non-terminal symbol. Thus, function

$$\begin{aligned}&\mathit{inferenceTerm} :: (\mathit{MonadOr}\ m, \mathit{MonadFresh}\ [\mathit{String}]\ \mathit{String}\ m)\\&\qquad \Rightarrow \mathit{Term} \rightarrow m\ (\mathit{RType}, \mathit{VarType})\end{aligned}$$

given a term returns its GADT representation together with the respective type representation ($RType$) and a $VarType$ defined as

$$\mathbf{type}\ VarType = [(Variable, TypeBox)]$$

Type $VarType$ is a environment which associates variable names to type representations. Since these are existential types, $TypeBox$ is used.

The $VarType$ result is used to keep track of variable types and pass this information around. Thus, $inferenceTerm$ is defined for relation variables as follows:

```
$inferenceTerm\ (Var'\ var) = \mathbf{do}$
  $t1 \leftarrow getFreshT; t2 \leftarrow getFreshT$
  $return\ (Exists\ (Rel\ t1\ \ t2)\ (Var\ var), [(var, Hide\ (Rel\ t1\ \ t2))])$
```

The type of a relation variable is $Rel\ t1\ \ t2$ where $t1$ and $t1$ are fresh type variables. At such a moment, it is not possible to say more about the type of var since only in the end of type inference a definitive type will be available in order to satisfy the monomorphic restriction. $[(var, Hide\ (Rel\ t1\ \ t2))]$ returns this association to the upper level.

As we did in the first version, let us see how to infer the types of converse and relational composition are inferred. The type inference of converse is now more complex:

```
$inferenceTerm\ (Conv\ r) = \mathbf{do}$
  $(Exists\ tr\ r', vars) \leftarrow inferenceTerm\ r$
  $t1 \leftarrow getFreshT; t2 \leftarrow getFreshT$
  $subst\ \ \ \ \ \leftarrow \eta\ [tr :=: Rel\ t1\ \ t2]$
  $Hide\ t1' \leftarrow typeRewrite\ subst\ t1$
  $Hide\ t2' \leftarrow typeRewrite\ subst\ t2$
  $r''\ \ \ \ \ \ \ \ \ \leftarrow safeCast\ subst\ (Rel\ t1'\ t2')\ r'$
  $return\ (Exists\ (Rel\ t2'\ t1')\ (r''^{\cup}),$
              $mapVariables\ subst\ vars)$
```

The difference is that of resorting to unification instead of *Haskell* pattern matching for the type tr of term r. In the first version, instead of returning tr, the result would be pattern matched againt $Rel\ t1\ \ t2$ directly. However, typing errors were very difficult to report because they looked like pattern matching errors. The second implementation allows for a possible future refinement: the introduction of type error messages by adding an error monad to the context of the function.

Since unification is used, we also need to use the *typeRewrite* and *safeCast* functions such as was explained in the first version of type inference for composition. The returned environment is the application of the function *mapVariables* to the sub-term environment. *mapVariables* is a function which applies a substitution to the type information associated with a variable identifier in a *VarType*. Type inference of converse does not change the types of relation variables, but the application of substitution is necessary to ensure that type variables in the environment are consistent with returned types.

The type inference for composition is now defined as

```
$inferenceTerm\ (Comp\ r1\ r2) = \mathbf{do}$
  $(Exists\ tr1\ r1', vars1) \leftarrow inferenceTerm\ r1$
  $(Exists\ tr2\ r2', vars2) \leftarrow inferenceTerm\ r2$
  $t1 \leftarrow getFreshT; t2 \leftarrow getFreshT; t3 \leftarrow getFreshT$
  $subst \quad \leftarrow \eta\ ([tr1 :=: Rel\ t3\ t2, tr2 :=: Rel\ t2\ t1]$
                 $+\!\!+\ toConstraints\ (vars1 +\!\!+ vars2))$
  $Hide\ t1' \leftarrow typeRewrite\ subst\ t1$
  $Hide\ t2' \leftarrow typeRewrite\ subst\ t2$
  $Hide\ t3' \leftarrow typeRewrite\ subst\ t3$
  $r1'' \quad \leftarrow safeCast\ subst\ (Rel\ t3'\ t2')\ r1'$
  $r2'' \quad \leftarrow safeCast\ subst\ (Rel\ t2'\ t1')\ r2'$
  $return\ (Exists\ (Rel\ t3'\ t1')\ (r1'' \circ_{t2'} r2''),$
          $mergeVariables\ subst\ vars1\ vars2)$
```

The main difference with respect to the first version is the use of environments with association of variables to types of both sub-terms. Such environments are concatenated so that function *toConstraints* takes this as argument and returns a list of constraints, one for each repeated identifier. For instance, $[(\texttt{"a"}, Hide\ t1), (\texttt{"a"}, Hide\ t2)]$ in the environment yields a constraint $[t1 :=: t2]$. This ensures that a variable will have the same type at every occurrence in the sub-terms. The generated constraints are added to the specific constraints of the composition operator and a most general unifier is computed over these.

The next steps are similar to the first version of type inference, except for the last one, where a new environment is returned using function:

```
$mergeVariables :: Substitution \to VarType \to VarType \to VarType$
$mergeVariables\ subst\ v1\ v2 = mapVariables\ subst\ (union\ v1\ v2)$
```

$mergeVariables$ takes the union of two $VarType$ environments and applies a substitution to the resulting environment. This ensures that the types of relation variables are correctly updated with the results of unification.

9.6 Laws and rules

Like parsing and type inference, also the derivation of rules from laws was implemented in two versions. The first one, reported by Silva and Oliveira [2008], is unable to deal with variables. However, the introduction of variables only requires the use of matching instead of equality. Both versions are presented below.

9.6.1 First version

Recall from Section 9.1 that term *Law* refers to expressions at the theoretical level, while term *Rule* denotes functions applicable by the rewriting system. From an equality $A = B$ two rules can be inferred, one in each direction of rewriting: $A \rightarrow B$ and $B \leftarrow A$. This corresponds to the application of the Leibniz principle: if two objects are equal we can interchange them keeping the validity of the enclosing expression.

Representing laws. An advantage of using equational reasoning is that it is type preserving, i.e., expressions of both sides of equalities (or inequalities) have the same type. Our representation for laws reflects this fact, using an existentially quantified index type:

$$\textbf{type}\ Law = Exists\ \ Type\ \ R$$

Thus, this type is essentially equal to $RType$ with the additional invariant (not statically enforced by the type system) that is only valid to arguments of the form $Exists\ (Expr\ \ t)\ (r1 = r2)$ and $Exists\ (Expr\ \ t)\ (r1 \subseteq r2)$.

Implementing rules. Following our design principles, laws can be specified in a purely declarative level using a textual notation from which the parser builds corresponding Law representations; which, in turn, are automatically converted into rewriting rules. A rewriting rule is defined as a polymorphic function, with type and expression representations as arguments,

$$\textbf{type}\ \mathit{Rule} = \forall a.\ \mathit{Type}\ a \to R\ a \to \mathit{Rewrite}\ (R\ a)$$

where *Rewrite* is a monad that deals with effects during rewriting. (See more details about this in Section 9.8). The function that, from a law representation returns a rewriting function (rule), is defined as follows:

```
getRule :: Law → Rule
getRule (Exists (Expr t1) (r1 = r2)) t2 r = do
  cns      ← rConstraint r1 r          -- 1
  subst    ← η ([t1 :=: t2] ++ cns)    -- 2
  Hide t1' ← typeRewrite subst t1      -- 3
  r'       ← safeCast subst t1' r      -- 4
  r1'      ← safeCast subst t1' r1     -- 4
  r2'      ← safeCast subst t2 r2      -- 5
  guard  (r1' ≡ r')                    -- 6
  successEquiv t2 r r2'                -- 7
```

The general principle of this function is that given an argument expression r and its type $t2$ it will try to match these with the left hand side of the law $r1$ and its corresponding type $t1$. If they are compatible the right hand side of the law is returned; otherwise, the function fails.

Each step of *getRule* is explained below, keeping in mind that it operates in the context of *MonadOr*: if one of the steps fails, *getRule* also fails.

1. Type equations are generated by the *rConstraint* function by comparing the two expressions one is trying to match. This is necessary in order to ensure that type annotations inside data constructors are correctly unified. For instance, when trying to match $\cdot \circ_{(TVar\ \texttt{"a"})} \cdot$ and $\cdot \circ_{Int} \cdot$, an equation $(TVar\ \texttt{"a"}) :=: Int$ should be generated. In case the two expressions do not match, *rConstraint* fails.

2. Type representations of the argument and law are unified, together with the equations generated in the previous step. The unification failure means that types are incompatible and no rule can be derived. Otherwise, a set of substitutions is returned.

3. The substitutions obtained in the previous step are applied to the type representation of the law.

4. The type-safe cast function is applied to the two expressions under comparison. This is needed because these have to be of the same type.

5. The type-safe cast function is applied to the right hand side of the law in order to make it possible to return a value with the right type $t2$, since our system is type preserving.

6. The argument expression and left hand side of the law, once casted, are compared for equality.

7. The $successEquiv$ function deals with the details of the $Rewrite$ monad (for instance, it adds a successful rewriting to a proof log). Otherwise, it could be just $return\ r2'$, meaning that it is possible to rewrite r into $r2'$, both having type $t2$.

The inverse rewrite rule (the right hand side by the left hand side of the law) is obtained through a similar function $getRuleInv$, the only difference being the inversion of variables.

9.6.2 Second version

Introducing variables only requires replacing equality by matching. The two different steps with respect to the previous version are signaled:

```
getRule :: Law → Rule
getRule (Exists (Expr t1) (r1 = r2)) t2 r = do
  cns       ← rConstraint r1 r
  subst     ← η ([t1 :=: t2] ++ cns)
  Hide t1'  ← typeRewrite subst t1
  r'        ← safeCast subst t1' r
  r1'       ← safeCast subst t1' r1
  r2'       ← safeCast subst t2 r2
  rsubst    ← rMatch r1' r'            -- 1
  let r2''  = rSubst rsubst t2 r2'     -- 2
  successEquiv t2 r r2''
```

The explanation of the two different steps is as follows:

1. Instead of testing $r1'$ and r' for equality one tries to match these. The function $rMatch$ implements expression matching which is a particular case of unification as it was discussed in Section 3.2. $rMatch$ returns a substitution σ such that $\sigma(r1') \equiv r'$, i.e., every variable of $r1'$ gets assigned a corresponding subexpression of r'.

   If the two expressions do not match, $getRule$ fails.

2. Function $rSubst$ applies the substitution obtained in the previous step to $r2'$, using the rewriting system defined on expressions (Section 9.8).

Such as in the previous version, the inverse rewriting rule is obtained simply by inverting variable order.

9.7 Galois connections properties

Galculator exploits Galois connection algebra and properties. These properties are equational laws representable in the system (recall Figure 5.1). What is needed is a way of automatically deriving properties from definitions.

For each property, a function is defined, receiving the representation of a Galois connection together with its type representation. For instance, the point-free version of the shunting property is generated by function:

```
$gcShunting :: (MonadFresh\ [String]\ String\ m, MonadOr\ m)$
        $\Rightarrow Type\ a \rightarrow R\ a \rightarrow m\ Law$
$gcShunting\ (GC\ b\ a)\ g = \mathbf{do}$
  $ladj \leftarrow lowerAdjoint\ g; uadj \leftarrow upperAdjoint\ g$
  $lord \leftarrow lowerOrder\ g;\ \ uord \leftarrow upperOrder\ g$
  $return\ \$\ Law\ (Expr\ (Rel\ a\ b))$
    $(((ladj^{\cup}) \circ_b (\omega_o(lord))) = ((\omega_o(uord)) \circ_a uadj))$
$gcShunting\ \_\ \_ = mzero$
```

The cancellation laws use the $\cdot \subseteq \cdot$ constructor because they are not equalities:

```
$gcCancellationUpper :: (MonadFresh\ [String]\ String\ m, MonadOr\ m)$
        $\Rightarrow Type\ a \rightarrow R\ a \rightarrow m\ Law$
$gcCancellationUpper\ (GC\ b\ a)\ g = \mathbf{do}$
```

$$\begin{aligned}
&\quad ladj \leftarrow lowerAdjoint\ g; uadj \leftarrow upperAdjoint\ g\\
&\quad uord \leftarrow upperOrder\ g\\
&\quad return\ \$\ Law\ (Expr\ (Rel\ a\ b))\\
&\qquad ((\omega_o(uord)) \subseteq ((\omega_o(uord)) \circ_a (uadj \circ_b ladj)))\\
&gcCancellationUpper\ \_\ \_ = mzero
\end{aligned}$$

The implementation uses the *upperAdjoint* and *lowerAdjoint* functions that from a Galois connection definition return, respectively their upper and lower adjoint representations. Also used are the *lowerOrder* and *upperOrder* functions that from a Galois connection definition return, respectively, the order representation associated with the lower and upper adjoints.

9.8 Term rewriting system

The term rewriting system (TRS) puts the rules derived from Galois connections to work. As already mentioned, it is based on strategic techniques (Section 3.4). Our TRS is not only strategic but also typed, allowing for type-dependent rewriting.

In the context of *Galculator*, the use of rewrite strategies on proofs can be compared to the use of tactics on traditional theorem provers [Paulson, 1983].

Two strategic TRS have been implemented: one for expressions and another for types. In spite of using similar strategies, they are slightly different.

Expressions. The definition of rewriting rules for expressions was already presented in Section 9.6. In fact, *Rule* is an instance of the general type of rewrite strategies

$$\mathbf{type}\ GenericM\ m = \forall a.\ Type\ a \rightarrow R\ a \rightarrow m\ (R\ a)$$

parameterized by the *Rewrite* monad which is defined using monad transformers in order to combine failure, non-determinism and a proof log. However, *GenericM* can be instantiated differently in order to deal with different kinds of effects.

Recall Table 3.3 which summarizes the defined strategy combinators. With the exception of the traversal (*all*, *one* and *first*) and the fixed point recursion combinators, the other are just renamings for monadic operations (recall Section 2.4.2). nop and $\triangleright$ are the return and bind operators of the monad class; $\perp$ and $\oplus$ the *mzero* and *mplus* operators of *MonadPlus*; $\oslash$ the *morelse* operator of *MonadOr*.

Special mention should go to the fact that two different choice operators have been defined: one for *left-biased* choice and another for *non-deterministic* choice. The latter ($\oplus$) is based on the $MonadPlus$ monad, thus obeying the left-distribution law. The use of non-deterministic choice is important when all different paths should be tried but it comes with a performance penalty since the search space expands.

The left-biased choice combinator ($\oslash$) is based on the $MonadOr$ and thus obeys the left catch law: the right strategy is only tried if the first one fails. This provides a mechanism for cutting down unnecessary cases and restricting the search space.

The cost of traversal operators (all, one, first) having to deal with boilerplate code is minimized by using a *spine* representation inspired on [Hinze et al., 2006]:

$$\begin{array}{l}
\textbf{data}\ Typed\ a\ \textbf{where} \\
\quad (:|)\quad :: Type\ a \to R\ a \to Typed\ (R\ a) \\
\textbf{data}\ Spine\ a\ \textbf{where} \\
\quad Constr :: a \to Spine\ a \\
\quad (\diamond)\quad :: Spine\ (a \to b) \to Typed\ a \to Spine\ b \\
fromSpine :: Spine\ a \to a \\
fromSpine\ (Constr\ c) \quad = c \\
fromSpine\ (f\ \diamond\ (\_ :|\ a)) = (fromSpine\ f)\ a \\
toSpine :: Type\ a \to R\ a \to Spine\ (R\ a) \\
\ldots
\end{array}$$

The $Spine$ data type is used in order to build a standard representation for constructors: $Constr$ is used for constructors without arguments, $\diamond$ is used for constructors with arguments. The $fromSpine$ function maps the spine representation back to actual expressions; $toSpine$ builds a spine representation from a given expression (the corresponding type representation is needed also). Changes in expression representation thus only affect $toSpine$.

The implementation of the all and one combinators boils down to defining how to traverse $Constr$ and $\diamond$ and use $toSpine$ and $fromSpine$ to map expressions between representations. This makes the implementation of the strategies independent of the representation and reduces the cost of changing.

The fixed point operator is easily defined as

$$rec\ s = s\ (rec\ s)$$

However, in *Haskell* it is easier to use recursion directly so this operator is not really necessary in the implementation.

More complex strategies have been defined using the basic combinators:

$$\begin{aligned}
&\mathit{try}\ s &&= s \oslash \mathit{nop}\\
&\mathit{many}\ s &&= (s \rhd \mathit{many}\ s) \oslash \mathit{nop}\\
&\mathit{many1}\ s &&= s \rhd \mathit{many}\ s\\
&\mathit{once}\ s &&= s \mid\mid\mid \mathit{one}\ (\mathit{once}\ s)\\
&\mathit{onceFirst}\ s &&= s \oslash \mathit{first}\ (\mathit{onceFirst}\ s)\\
&\mathit{topdown}\ s &&= s \rhd \mathit{all}\ (\mathit{topdown}\ s)\\
&\mathit{bottomup}\ s &&= \mathit{all}\ (\mathit{bottomup}\ s) \rhd s\\
&\mathit{innermost}\ s &&= \mathit{all}\ (\mathit{innermost}\ s) \rhd \mathit{try}\ (s \rhd \mathit{innermost}\ s)
\end{aligned}$$

Using the fixed point operator, the many strategy would be defined as

$$\mathit{many}\ s = \mathit{rec}\ (\lambda f \rightarrow ((s \rhd f) \oslash \mathit{nop}))$$

For instance, if we have a rule that associates any operator to the left, say $\mathit{assocLeft}$, and we want to associate an expression to the left we just have to use $\mathit{innermost}\ \mathit{assocLeft}$.

Types. The rewriting system for types uses the same principles and combinators as the TRS for expressions. However, it is not type preserving because changing type representations will imply having a different type in the end.

In order to make the type system consider all rewritings as type preserving, the new type is hidden using another existential data type [Cunha et al., 2006b]:

$$\mathbf{data}\ \mathit{View}\ a\ \mathbf{where}\ \mathit{View} :: \mathit{Type}\ b \rightarrow \mathit{View}\ (\mathit{Type}\ a)$$

Thus, the rewrite rule for Type will have the following type:

$$\mathbf{type}\ \mathit{Rule}\ m = \forall a.\ \mathit{Type}\ a \rightarrow m\ (\mathit{View}\ (\mathit{Type}\ a))$$

The result of the rewriting is exactly of the same type, apart from the new type hidden inside View. Since it is existentially quantified it cannot be used outside the scope of the quantification. However, we can pass it to other existential types, using the Covert pattern:

$$\begin{aligned}&\mathit{view2Box} :: \mathit{View}\ (\mathit{Type}\ a) \to \mathit{TypeBox}\\&\mathit{view2Box}\ (\mathit{View}\ t) = \mathit{Hide}\ t\end{aligned}$$

Now, it can be manipulated providing that the result never falls outside an existentially quantified type. Using this approach, the function

$$\mathit{typeRewrite} :: \mathit{MonadOr}\ m \Rightarrow [\mathit{Substitution}] \to \mathit{Type}\ t \to m\ \mathit{TypeBox}$$

receives a list of substitutions to be applied to a type t. Since substitutions on types are obviously not type-preserving, the result has to be encapsulated inside a $\mathit{TypeBox}$.

9.9 Free-theorems

9.9.1 Brief introduction

Free-theorems were introduced by Wadler [1989] based on Reynold's abstraction theorem [Reynolds, 1983]. A *free-theorem* is a parametric result stating a property enjoyed by any polymorphic function with a certain type. As Wadler [1989] explains it: *"The key idea is that types may be read as relations"*.

Type expressions. Parametric type expressions are defined as in Section 5.6.5:

$$\begin{array}{rcll} t & ::= & v & \text{Type variable}\\ & | & t'' \leftarrow t' & \text{Function type}\\ & | & \mathcal{F}(t_1, \ldots t_n) & n\text{-ary type relator}\end{array}$$

where $v \in \mathcal{V}$ and $\mathcal{V}$ is the set of type variables.

Free-theorem of type t. Suppose we have $\{R_v\}_{v\in\mathcal{V}}$, a $\mathcal{V}$-indexed family of relations, i.e., for each type variable $v \in \mathcal{V}$ we assign a relation R_v. A free-theorem for a parametrically polymorphic function of type t, is obtained by building a relation R_t such that $(f, f) \in R_t$. It should be noticed that the two f functions above are, in fact, different instantiations of the same polymorphic function [Backhouse and Backhouse, 2004]. Relation R_t is inductively defined on t as

1. If t is type variable $t = v$ then $R_t = R_v$, i.e., the v indexed relation of $\{R_v\}_{v\in\mathcal{V}}$.

2. If t is a function type $t = \mathcal{B} \leftarrow \mathcal{A}$ then $R_t = R_{\mathcal{B}} \leftarrow R_{\mathcal{A}}$, where $\leftarrow$ is known as the *Reynolds arrow*, a relation on functions defined as

$$f\,(R \leftarrow S)\,g \quad \overset{def}{=} \quad \langle \forall\, x, y :: (f\,x)R(g\,y) \Leftarrow xSy \rangle \tag{9.1}$$

   Intuitively, if input values are related by S then R relates the respective images through functions f and g.

   Using the point-free transform we get an equivalent formulation

$$f\,(R \leftarrow S)\,g \quad \overset{def}{=} \quad f \circ S \subseteq R \circ g \tag{9.2}$$

3. If t is a n-ary type relator $t = \mathcal{F}(t_1, \ldots, t_n)$, where $n \geqslant 1$, then $R_{\mathcal{F}(t_1,\ldots,t_n)} = \mathcal{F}(R_{t_1}, \ldots, R_{t_n})$.
4. If t is a 0-ary type relator $t = \mathcal{T}$ (basic type), then $R_{\mathcal{T}} = id_{\mathcal{T}}$, i.e., corresponds to the identity relation on $\mathcal{T}$.

An important feature of free-theorems is that they only depend on the type of the function, and not on the function itself. Thus, if two polymorphic functions share the same type, they obey the same free-theorem.

Example. The $head$ function that returns the first element of a list has type $a \xleftarrow{head} a^\star$, for all data types a. From this, we want to determine the relation R_t for $t = a \leftarrow a^\star$:

$$\begin{array}{cl} & R_{a \leftarrow a^\star} \\ \Leftrightarrow & \qquad \{ \text{ Function type — Rule (2) above. } \} \\ & R_a \leftarrow R_{a^\star} \\ \Leftrightarrow & \qquad \{ \text{ Relator type — Rule (3) above. } \} \\ & R_a \leftarrow {R_a}^\star \end{array}$$

where R_a is an arbitrary relation. By the free-theorem result we have that

$$\begin{array}{cl} & head\,(R_a \leftarrow {R_a}^\star)\,head \\ \Leftrightarrow & \qquad \{ \text{ Definition of Reynold's arrow operator (9.2). } \} \\ & head \circ {R_a}^\star \;\subseteq\; R_a \circ head \end{array}$$

When R_a is a function f, since inclusion and equality coincide for functions (Equation (4.1)), this boils down the $head$ natural property:

$$head \circ f^\star \quad = \quad f \circ head$$

Recall that * is a functor and $f^\star$ is equivalent to the function $map\ f$ of *Haskell* presented in Section 2.4.1. This property can be represented by the following commuting diagram:

$$\begin{tikzcd} a \arrow[d, "f"'] & a^\star \arrow[l, "head"'] \arrow[d, "f^\star"] \\ b & b^\star \arrow[l, "head"] \end{tikzcd}$$

for types a and b.

The $last$ function that returns the last element of a list has type $a \xleftarrow{last} a^\star$, for all data types a. Since $last$ and $head$ share the same polymorphic type, they both obey the same free-theorem. Thus, we have for any relation R that

$$last \circ R^\star \quad \subseteq \quad R \circ last$$

9.9.2 Implementation

In the following, we describe a simple implementation of a mechanism that derives a law corresponding to the free-theorem of a given polymorphic function.

Additional representation. First of all, we need to extend our representation with the Reynolds arrow operator

$$\begin{array}{l} \textbf{data}\ R\ r\ \textbf{where} \\ \quad \ldots \\ \quad \cdot \leftarrow \cdot :: R\ (b \sim a) \rightarrow R\ (d \sim c) \rightarrow R\ ((b \leftarrow d) \sim (a \leftarrow c)) \end{array}$$

If r and s are relations, $r \leftarrow s$ establishes a relation on functions.

Relation from types. The next step is to implement a function corresponding to the inductive definition of relation R_t for a type expression t. Thus, we have

$$\begin{array}{l} type2Rel :: (MonadFresh\ [String]\ String\ m, \\ \qquad\qquad\quad MonadState\ [(String, RType)]\ m, \end{array}$$

```
            $MonadOr\ m)$
          $\Rightarrow Type\ a \rightarrow m\ (RType)$
$type2Rel\ (TVar\ v) = maybe$
  $(\mathbf{do}\ r1 \leftarrow getFreshR$
      $t1 \leftarrow getFreshT; t2 \leftarrow getFreshT$
      $\mathbf{let}\ e = Exists\ (Rel\ t1\ t2)\ (Var\ r1)$
      $modify\ ((v, e){:})$
      $return\ e)$
  $return \circ lookup\ v =\!\!\ll get$
$type2Rel\ (Fun\ a\ b) = \mathbf{do}$
  $Exists\ (Rel\ t1a\ t2a)\ ra \leftarrow type2Rel\ a$
  $Exists\ (Rel\ t1b\ t2b)\ rb \leftarrow type2Rel\ b$
  $return\ (Exists\ (Rel\ (Fun\ t1a\ t1b)\ (Fun\ t2a\ t2b))\ (ra \leftarrow rb))$
$type2Rel\ (a \times b) = \mathbf{do}$
  $Exists\ (Rel\ t1a\ t2a)\ ra \leftarrow type2Rel\ a$
  $Exists\ (Rel\ t1b\ t2b)\ rb \leftarrow type2Rel\ b$
  $return\ (Exists\ (Rel\ (t1a \times t1b)\ (t2a \times t2b))\ (ra \times rb))$
$type2Rel\ One = return\ (Exists\ (Rel\ One\ One)\ Id)$
...
$type2Rel\ \_ = mzero'$
```

Instead of an indexed family of relations $\{R_v\}_{v \in \mathcal{V}}$ we use the infinite supply of relation variables given by *getFreshR*. For each type variable we lookup an environment $[(String, RType)]$ to see if there is any relation variable already assigned to it. If this is true, we just return the associated value. Otherwise, we ask for a new relation variable, and keep the association with the type variable in the environment.

The implementation for functions and relators is quite straightforward. However, to deal with other types such as $\cdot + \cdot$, *Maybe*, *List*, our relation representation has to be augmented with more relators. For basic types, we just show the definition for *One*; the other cases are similar.

Free-theorem. Using *type2Rel*, we can now define the function *free* which takes a type representation associated with a polymorphic function and returns the law expressing its free-theorem:

```
$free :: (MonadFresh\ [String]\ String\ m, MonadOr\ m)$
    $\Rightarrow Type\ (b \leftarrow a) \rightarrow R\ (b \leftarrow a) \rightarrow m\ Law$
$free\ (Fun\ b\ a)\ f = \mathbf{do}$
  $Exists\ (Rel\ (Fun\ trb\ tra)\ (Fun\ trb'\ tra'))\ (r1 \leftarrow r2) \leftarrow evalStateT$
      $(type2Rel\ (Fun\ b\ a))\ [\,]$
  $subst \quad \leftarrow \eta\ [Fun\ b\ a :=: Fun\ trb\ tra, Fun\ b\ a :=: Fun\ trb'\ tra']$
  $Hide\ b' \leftarrow typeRewrite\ subst\ b$
  $Hide\ a' \leftarrow typeRewrite\ subst\ a$
  $f' \qquad \leftarrow safeCast\ subst\ (Fun\ b'\ a')\ f$
  $r1' \qquad \leftarrow safeCast\ subst\ (Rel\ b'\ b')\ r1$
  $r2' \qquad \leftarrow safeCast\ subst\ (Rel\ a'\ a')\ r2$
  $return\ (Exists\ (Expr\ (Rel\ b'\ a'))\ (((\omega_f(f')) \circ_{a'} r2') \subseteq (r1' \circ_{b'} (\omega_f(f')))))$
$free\ \_\ \_ = mzero'$
```

9.10 Summary

This chapter presented the description of the actual *Galculator* prototype written in *Haskell*. The prototype uses many advanced features of functional programming, namely GADTs. Using GADTs, it is possible to build a reflexion mechanism which provides type information about term during run-time, allowing for type dependent behavior. This was explored in the implementation of free-theorems automatic derivation, although more work needs to be done on this subject. Moreover, the use of GADTs ensures that terms are well-typed by construction. The drawback is that parsing and building AST representations becomes more complex since type-inference on terms is needed. We have shown how existential data types can be used in this process.

We have extended the traditional type representation mechanism to allow for parametric polymorphism, i.e., the introduction of type variables. The drawback is the increased complexity since type equality has to be replaced by type unification.

Rewriting rules in *Galculator* are automatically generated from their definitions. This allows for a complete distinction between the theoretical concepts expressed in the *Galois* language and the operational application of the rule during proof in the assistant. The implementation and use of a strategic term rewriting system, whose strategies are available to the user, allows for great power and flexibility in the application of rules.

Chapter 10

Conclusions and future work

This chapter closes this dissertation by presenting conclusions, discussing contributions, comparing these with related work and suggesting directions for further work.

10.1 Main contributions

Approach. The approach taken in *Galculator* is innovative to the best of our knowledge: it is the first time that fork algebras and Galois connections are used together to perform formal proofs. Fork algebras allow for equational reasoning based on a very simple inference rule and without the problems usually associated with the manipulation of variables. Galois connections provide structure with nice algebraic properties and a mechanism of changing the domain of proofs. Fork algebras and Galois connections, together with the introduction of the indirect equality principle provide a powerful framework to tackle the complexity of proofs in general and about software correctness in particular.

All these concepts are integrated with an existing proof format amenable for either pencil-and-paper or computer assisted proofs. This format is suited for clearly stated equational and inequational proofs, in which each step is suitably justified. This style also allows for several levels of detail, ranging from formal to informal proofs.

Study of properties and application of Galois connections. This dissertation gathers the most important theoretical results on Galois connections from several different sources. A comprehensive study of the different approaches to combine existing Galois connections to form new Galois connections is presented. This leads to an *algebra* *of Galois connections*. The link with category theory is explored, not just in the traditional perspective of Galois connections scaled up to adjunctions of a category of pre-orders, but also in establishing a category of Galois connections. Furthermore, we have explored the possibility of extending Galois connections to allegories.

A survey of applications of Galois connections is presented. These range from the well-known ones such as abstract interpretation and formal concept analysis, to residuated structures, data refinement or temporal and separation logics.

***Galois* language.** A domain specific language have been developed to integrate all concepts and serve has a front-end to a proof-assistant based on Galois connections. The syntax and semantics of *Galois* is derived as a natural consequence of the algebraic nature of the theoretical definitions. In fact, *Galois* is a family of DSLs composed in a hierarchy which mirrors the structure of the theoretical concepts involved.

We aimed to provide some evidence of the importance of a mechanism such as Galois connections in the design of a language for formal reasoning. The connections between concepts at a semantic level can be operated syntactically in an equational approach, with the help of the introduction of indirect equality. We also showed the usefulness of a strongly typed environment at several levels, even when not using an approach based on the Curry-Howard isomorphism [Sørensen and Urzyczyn, 2006].

***Galculator* prototype.** The *Galculator* is a proof assistant which implements an innovative approach to theorem proving, different from what is traditional in the field: it is solely based on Galois connections, their algebra and associated tactics. Thanks to Galois connection algebra, it builds new connections from old thanks to a number of combinators enabling such constructions on the fly. It is based on the tactic of indirect equality, which fits naturally with Galois connections.

To the best of our knowledge, supported by submissions to conferences in the field [Silva and Oliveira, 2008; Silva et al., 2009], the *Galculator* is the first proof engine ever to combine and calculate directly with PF-transformed Galois connections.

Implementation techniques. Most of the techniques employed in the development of the *Galculator* are not new, as they are applications of work reported elsewhere in the literature. Arguably, the extension of the type representation in order to support polymorphic type representations and the support for unification is new; we are not aware of any other such implementation, although it may exist.

We regard the current prototype as a non-trivial illustration of the power of functional programming advanced features for building complex systems. It manages to combine many often publicized distinctive features of functional languages: GADTs, existential data types, combinatory approaches (parsing, rewriting), the support for embedded DSLs, computations as monads, higher-order functions and some other. Specially, the use of GADTs and existential data types allows for mixing static and dynamic typing in a powerful way, making it possible to guarantee the static safeness of objects whose type will only be known at runtime, stepping into the power of dependent typing [Martin-Löf, 1984].

10.2 Final remarks

Fork algebras and allegories. The axiomatization of relations described in this dissertation as basis for the *Galculator* is that of fork algebras [Frias et al., 2004b]. Fork algebras were used because of their clear relation with logics steaming from Tarski's work on relation algebras [Tarski and Givant, 1987]. The equivalence of fork algebras and first-order logic is clearly established in terms of expressive and deductive power. The so-called point-free-transform [Tarski and Givant, 1987; Bird and de Moor, 1997; Oliveira, 2009] connects the two worlds.

However, the design of *Galculator* does not exclude the use of other axiomatizations, since axioms concerning the language symbols are specified externally. This means that the axiomatization of allegories can be used instead. However, allegories admit models which are not set-theoretical relations [Bird and de Moor, 1997]. Only tabular and unitary allegories are close to relations in terms of proof. Tabulations allow for proofs resembling point-wise derivations when a "pure" point-free proof does not exist [Bird and de Moor, 1997]. Thus, unlike fork algebras, there is not a clear equivalence between allegories and some kind of logic.

Point-free style. Relations are used as modelling devices because they can easily accommodate failure, partiality and non-determinism. These features naturally lead to a point-free-style, because, as Bird et al. [2002] state *"Basically, one cannot assign to a variable 'the result of running program p on input x', because there may be no or many results."* Furthermore, the use of point-free relations makes substitutions easier because the capture of variables is not possible.

However, the effectiveness of the point-free-style greatly varies. In some cases, the complex logical quantification structure collapses leading to short and easy to read point-free expressions, which can be effectively used in proofs. However, in other cases, the equivalent point-free expression is cumbersome specially because the large use of fork and projection operators. This effect is extended to some proofs where many "bookkeeping" steps dealing with forks and projections are necessary. This seems the point-free equivalent version of the point-wise manipulation of variables.

Variables in point-wise style and forks and projections in point-free style assume the same role in expressions: they are the "plumbing" devices that feed values to specific operations. In point-free this feeding works like a real plumbing or an electrical wiring (despite the fact that in these two examples the feeding is a flow), in which there are replication and derivation points. The point-wise version works more like addressing, where values are delivered to points with the same identification (the variable name).

An interesting question is determining in which cases each style is more effective. Is there any pattern behind the effectiveness of each style? Currently, no definite answers exist to these questions meaning that more studies are needed about this subject.

Specific vs. general-purpose rewriting system. We decided to develop our own rewriting system instead of using another rewriting engine, e.g., *Stratego* [Visser, 2001a], *Maude* [Clavel et al., 2000] or *ELAN* [Borovanský et al., 1996]. The use of such a general-purpose rewriting engine is advisable when implementing a final version of the proof-assistant, mostly because of performance, but not at the prototyping stage. In the prototype, we want to explore types and operations on them, while ensuring correctness. It would be much more difficult to detect errors resorting to an external tool, mainly because part of the typed behavior of the system would be lost.

Moreover, translating *Galculator* laws into a foreign rewriting engine would require more effort than is needed to implement the whole rewriting system. Since this relies on *Haskell* monads, its implementation is quite simple and extensible. For instance, state information may be required to implement some of the future features. This is easily accomplished by using monad transformers.

10.3 Comparison with related work

After discussing the design of the *Galculator* prototype, this section compares it with the related work introduced in Section 2.1.

***aRa* [Sinz, 2000].** Like in our approach, formalization and translation in *aRa* follow Tarski and Givant [1987]. However, *aRa* takes an alternative (in fact, opposite) direction: relation algebraic formulæ are translated to logical sentences and proved using logic, while we conduct our proofs in the algebraic setting. Moreover, the expressive and deductive power of *aRa* is lower than that of *Galculator* because it uses relation algebra rather than fork algebra. The integration of Galois connections is absent and so does typing.

***RALL* [von Oheimb and Gritzner, 1997].** *RALL* is somewhat similar to *aRa*, so the comparison with *Galculator* is identical. The main difference is that, like *Galculator*, it checks for type-correctness of all formulæ.

***RELVIEW* [Behnke et al., 1998].** Unlike *RELVIEW*, which can only be used with finite concrete relations, *Galculator* can be used with infinite relations because they are defined abstractly. *RELVIEW* can be seen as the concrete counterpart of the abstract algebraic perspective of *Galculator*. An interesting experimental study would be to see in detail how the two approaches relate and complement each other.

[Höfner and Struth, 2008]. In this work, the *Prover9* automated theorem prover is employed to prove several theorems of relation algebra. However, since only relation algebras are used, it is restricted to a three-variable fragment of first-order logic, unlike our approach which uses fork algebra. Höfner and Struth [2008] mention that some relation operations are adjoints of Galois connections but this fact is not exploited in proofs. Indirect equality is not used either. Moreover, their approach in not always equational because some proofs require the use of mutual inclusion, since indirect equality is not used.

The results of this work are interesting concerning the automatization of proofs. The core of *Galculator* would benefit from following a similar approach.

***2LT* [Cunha et al., 2006b].** The core of the *Galculator* is inspired on the *2LT* (Two-level transform) system. Our representation technique and the rewriting strategies implemented were mostly influenced by the design of the *2LT* system, although the rewriting rules of *2LT* are defined using functions and therefore hard-wired into the system. Moreover, the *2LT* type representation mechanism does not support polymorphism. Although *2LT* does not rely on Galois connections explicitly, its underlying theory does so [Oliveira, 2008].

***PF-ESC* [Necco et al., 2007].** Although *PF-ESC* shares some common concepts with the *Galculator*, the two systems are different. The *PF-ESC* representation uses properties to classify relations while the *Galculator* uses the type representation itself. An advantage of using properties is that the system is more flexible in so far as allowing for new kinds of relation. Moreover, the type-lifts of our approach are not needed. However, predicate functions which calculate the properties of expressions are required in order to apply certain transformations. This makes the system not extensible because rewrite equations must be hard-wired into functions. Since the *Galculator* is based on types, predicate function are not needed and the rewrite rules can be purely declarative. Moreover, the representation used in the *Galculator* is statically safer, since incorrect constructions are not allowed.

Proof processor system [Bohórquez and Rocha, 2005]. This system, unlike our approach, does not provide type support and does not use Galois connections as a building block of the calculus implemented. Moreover, the input notation is *Z* which has a relational flavor but it differs from our "pure" relation calculus. However, this work shows how a system like *Galculator* can have potential interest for teaching activities.

Galois connections in Coq [Pichardie, 2005]. This work does not exploit the Galois connection algebra in order to combine existing connections nor it is applied in proofs. However, this approach in a sense complements the *Galculator* system since it can be used to discharge proof obligations about adjoints prior to loading these into our system.

10.4 Future work

The development of *Galculator* leaves several open questions which may be exploited in future work.

Integration with 'host' theorem provers. *Galculator* is not a general prover: it works only with well-defined situations involving Galois connections. Combined with other theorem provers it can behave like a specialized *add-on* component able to discharge proofs wherever terms involve adjoints of known connections.

Currently, we are working on integrating the *Galculator* with *Coq* [Silva et al., 2008], either following the *believing* or the *skeptical* approach, as discussed by Delahaye and Mayero [2005] in integrating *Coq* with *Maple*. In the first case, *Galculator*-proven assertions are added as axioms. In the second, the idea is to define tactics in *Coq* which exploit Galois connection properties and invoke the *Galculator* in order to use the built-in strategies to prove the correctness of the steps. The resulting proofs are then replayed using reflection to build trusted proofs in *Coq*.

The prospect of integrating the *Galculator* with other proof assistants is also open.

Mechanization of point-free transform. In Section 4.5, the equivalence between fork algebra formulas and first-order sentences is described. It would be interesting to automate such a point-free transformation, like Cunha et al. [2005] have done for functional programs, and incorporate it as a front-end for *Galculator*. Moreover, the point-free transform is not restricted to first-order classical logic; it can be extended to several non-classical logics as described by Frias et al. [2004b].

Automated proofs. Currently, the *Galculator* is used as a proof assistant where proofs are guided by the user. Some efforts have been made in order to automate proofs which exhibit recurrent patterns. However, the developed strategies can only deal with some of these patterns. More general strategies applicable to a wider range of problems are needed.

A possible approach is the one followed by Höfner and Struth [2008] which resorts to an automated theorem prover, namely *Prover9* (Section 2.1).

Free-theorems. Exploiting free-theorems with Galois connections has been one of the objectives of the *Galculator*, specially because from Backhouse and Backhouse

[2004] we know how to calculate free-theorems [Wadler, 1989] about Galois connections based on their types. We have introduced some basic support for free-theorems in *Galculator*. However, since the general theory of free-theorem regards types as relators, some work needs to be done to access if it is possible to accommodate relators in the fork algebra framework. Otherwise, instead of fork algebras we would have to use tabular allegories [Bird and de Moor, 1997] as the theoretical foundation of *Galculator*.

Extension of the type system. The type system of *Galculator* is limited although it supports parametric polymorphism. The set of basic types and type constructors is fixed and cannot be extended, i.e., the user cannot declare new types. This restricts the types of relations that can be declared.

Besides an extension mechanism with user declared types, it would be also interesting to explore more sophisticated type system features, such as overloading, type classes or even dependent types.

Evaluation of the language. The *Galois* language is very close to the corresponding mathematical language. Intuitively, its usefulness and usability should mirror to theoretical underpinning. However, our experience with the language does not allow us to draw definite conclusions, yet. Further experimentation is needed to access the strengths and weaknesses of the language. Moreover, it should be also compared with the DSLs of other approaches.

Application to abstract interpretation. Cousot [1999] shows how to design abstract semantics for abstract interpretations satisfying soundness requirements. This approach caters for correctness of the design by calculation and uses Galois connections as a fundamental concept. The style of calculus resembles the one used in *Galculator*, although it uses variables. Thus, the calculus of abstract semantics would be a natural application of *Galculator*. However, some work must be done in order to accommodate the calculus of Cousot [1999] in a relational setting.

Appendix A

Additional proofs

A.1 Sections of Galois connections

A.1.1 Commutative operators

Let $\mathcal{A} \xleftarrow{\theta} \mathcal{A} \times \mathcal{A}$ and $\mathcal{A} \xleftarrow{\otimes} \mathcal{A} \times \mathcal{A}$ be two binary operators on a *poset* $(\mathcal{A}, \sqsubseteq)$, such that their sections are Galois connected, i.e., for $a \in \mathcal{A}$ and for all $x, y \in \mathcal{A}$,

$$(a\theta)x \sqsubseteq y \quad \Leftrightarrow \quad x \sqsubseteq (a\otimes)y \tag{A.1}$$

If θ is commutative. Let us assume that θ is a commutative operator, i.e., $a\theta b = b\theta a$. This means that left and right sections of θ coincide:

$$(\theta b)a \quad = \quad (b\theta)a \tag{A.2}$$

Proof

$$\begin{array}{cl}
& x \sqsubseteq (a \sqcup a') \otimes y \\
\Leftrightarrow & \qquad \{ \text{ Shunting (A.1). } \} \\
& (a \sqcup a')\, \theta\, x \sqsubseteq y \\
\Leftrightarrow & \qquad \{ \text{ Commutativity of } \theta \text{ (A.2). } \} \\
& x\, \theta\, (a \sqcup a') \sqsubseteq y \\
\Leftrightarrow & \qquad \{ \text{ Lower adjoint distributes over } \sqcup. \} \\
& x\, \theta\, a \sqcup x\, \theta\, a' \sqsubseteq y
\end{array}$$

$$
\begin{array}{ll}
\Leftrightarrow & \{\ \sqcup\text{-universal.}\ \} \\
& x\,\theta\,a \sqsubseteq y \ \wedge\ x\,\theta\,a' \sqsubseteq y \\
\Leftrightarrow & \{\ \text{Commutativity of } \theta \text{ (A.2).}\ \} \\
& a\,\theta\,x \sqsubseteq y \ \wedge\ a'\,\theta\,x \sqsubseteq y \\
\Leftrightarrow & \{\ \text{Shunting (A.1).}\ \} \\
& x \sqsubseteq a \otimes y \ \wedge\ x \sqsubseteq a' \otimes y \\
\Leftrightarrow & \{\ \sqcap\text{-universal.}\ \} \\
& x \sqsubseteq a \otimes y \sqcap a' \otimes y \\
\therefore & \{\ \text{Indirect equality.}\ \} \\
& (a \sqcup a') \otimes y = a \otimes y \sqcap a' \otimes y
\end{array}
$$

□

Therefore, we conclude that if θ is commutative then

$$(a \sqcup a') \otimes y \quad = \quad a \otimes y \sqcap a' \otimes y \tag{A.3}$$

If $\otimes$ is commutative. Assuming $\otimes$ commutative:

$$(\otimes b)a \quad = \quad (b\otimes)a \tag{A.4}$$

Proof

$$
\begin{array}{ll}
& (a \sqcap a')\,\theta\,x \sqsubseteq y \\
\Leftrightarrow & \{\ \text{Shunting (A.1).}\ \} \\
& x \sqsubseteq (a \sqcap a') \otimes y \\
\Leftrightarrow & \{\ \text{Commutativity of } \otimes \text{ (A.4).}\ \} \\
& x \sqsubseteq y \otimes (a \sqcap a') \\
\Leftrightarrow & \{\ \text{Upper adjoint distributes over } \sqcap.\ \} \\
& x \sqsubseteq y \otimes a \sqcap y \otimes a' \\
\Leftrightarrow & \{\ \sqcap\text{-universal.}\ \} \\
& x \sqsubseteq y \otimes a \wedge x \sqsubseteq y \otimes a' \\
\Leftrightarrow & \{\ \text{Commutativity of } \otimes \text{ (A.4).}\ \}
\end{array}
$$

$$
\begin{array}{ll}
& x \sqsubseteq a \otimes y \wedge x \sqsubseteq a' \otimes y \\
\Leftrightarrow & \qquad \{ \text{ Shunting (A.1). } \} \\
& a \otimes x \sqsubseteq y \wedge a' \otimes y \sqsubseteq y \\
\Leftrightarrow & \qquad \{ \sqcup\text{-universal. } \} \\
& a\,\theta\,y \sqcup a'\,\theta\,y \sqsubseteq y \\
\therefore & \qquad \{ \text{ Indirect equality. } \} \\
& (a \sqcap a')\,\theta\,y = a\,\theta\,y \sqcup a'\,\theta\,y
\end{array}
$$

□

Therefore, we conclude that if $\otimes$ is commutative then

$$(a \sqcap a')\,\theta\,y \quad = \quad a\,\theta\,y \sqcup a'\,\theta\,y \qquad \text{(A.5)}$$

A.1.2 Residuated semigroups

Question. Let $(\mathcal{A}, \sqsubseteq, \otimes)$ be a residuated partially ordered semigroup. Thus, $\otimes$ is a monotonic associative operation, for which the following two Galois connections hold,

$$(a\otimes)^{\cup} \circ \sqsubseteq \quad \Leftrightarrow \quad \sqsubseteq \circ (a\backslash) \qquad \text{(A.6)}$$

$$(\otimes a)^{\cup} \circ \sqsubseteq \quad \Leftrightarrow \quad \sqsubseteq \circ (/a) \qquad \text{(A.7)}$$

Suppose that $\otimes$ is a commutative, i.e.,

$$(a\otimes) \quad = \quad (\otimes a) \qquad \text{(A.8)}$$

We want to prove that if $\otimes$ is commutative, then left and right division coincide, i.e.,

$$(a\backslash) \quad = \quad (/a) \qquad \text{(A.9)}$$

Proof

$$
\begin{array}{ll}
& \sqsubseteq \circ (a\backslash) \\
\Leftrightarrow & \qquad \{ \text{ Shunting (A.6). } \} \\
& (a\otimes)^{\cup} \circ \sqsubseteq
\end{array}
$$

$$\begin{array}{ll}
\Leftrightarrow & \quad \{ \text{ Commutativity of } \otimes \text{ (A.8). } \} \\
& (\otimes a)^{\cup} \circ \sqsubseteq \\
\Leftrightarrow & \quad \{ \text{ Shunting (A.7). } \} \\
& \sqsubseteq \circ (/a) \\
\therefore & \quad \{ \text{ Indirect equality. } \} \\
& (a\backslash) \;=\; (/a)
\end{array}$$

□

A.2 Relators preserve Galois connections

A.2.1 Preorders

Question. We want to prove that relators preserve Galois connections, i.e, for any relator $\mathcal{F}$ and any Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ between *preorders* $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$, $(\mathcal{F}\,\mathcal{A}, \mathcal{F} \sqsubseteq) \xleftarrow{(\mathcal{F}\,f, \mathcal{F}\,g)} (\mathcal{F}\,\mathcal{B}, \mathcal{F} \preceq)$ is also a Galois connection.

$\mathcal{F}$ **preserves functions.** We must first prove that $\mathcal{F}\,f$ and $\mathcal{F}\,g$ are functions. This holds trivially, because relators preserve functions. The general properties of relators are discussed in Section 4.6.

$\mathcal{F}$ **preserves preorders.** We must prove that, for any preorder $(\mathcal{A}, \sqsubseteq)$ and any relator $\mathcal{F}$, $(\mathcal{F}\,\mathcal{A}, \mathcal{F} \sqsubseteq)$ is also a preorder. A preorder is a reflexive and transitive relation, respectively,

$$id \;\subseteq\; \sqsubseteq \qquad \text{(A.10)}$$

$$\sqsubseteq \circ \sqsubseteq \;\subseteq\; \sqsubseteq \qquad \text{(A.11)}$$

First, we prove that $\mathcal{F} \sqsubseteq$ is reflexive:

Proof

$$
\begin{array}{ll}
 & id \subseteq \mathcal{F} \sqsubseteq \\
\Leftrightarrow & \qquad \{ \text{ Relators preserve identity. } \} \\
 & \mathcal{F}\, id \subseteq \mathcal{F} \sqsubseteq \\
\Leftarrow & \qquad \{ \text{ Monotonicity of relators. } \} \\
 & id \subseteq \sqsubseteq \\
\Leftrightarrow & \qquad \{ \text{ Assumption (A.10). } \} \\
 & \top
\end{array}
$$

□

Then, we prove that $\mathcal{F} \sqsubseteq$ is transitive:

Proof

$$
\begin{array}{ll}
 & \mathcal{F} \sqsubseteq \circ \mathcal{F} \sqsubseteq \\
= & \qquad \{ \text{ Relators preserve composition. } \} \\
 & \mathcal{F}\, (\sqsubseteq \circ \sqsubseteq) \\
\subseteq & \qquad \{ \text{ Transitivity of } \sqsubseteq \text{ (A.11). Monotonicity of relators. } \} \\
 & \mathcal{F} \sqsubseteq
\end{array}
$$

□

Thus, we conclude that $(\mathcal{F}\, \mathcal{A}, \mathcal{F} \sqsubseteq)$ is a preorder.

$\mathcal{F}$ preserves shunting. Assuming that $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ is a Galois connection, that is,

$$f^{\cup} \circ \preceq \quad = \quad \sqsubseteq \circ\, g \tag{A.12}$$

holds, we must prove that

$$(\mathcal{F}\, f)^{\cup} \circ \mathcal{F} \preceq \quad = \quad \mathcal{F} \sqsubseteq \circ\, \mathcal{F}\, g \tag{A.13}$$

Proof

$$
\begin{array}{ll}
& (\mathcal{F}\, f)^{\cup} \circ \mathcal{F} \preceq \\
= & \quad \{ \text{ Relators preserve converse. } \} \\
& \mathcal{F}\,(f^{\cup}) \circ \mathcal{F} \preceq \\
= & \quad \{ \text{ Relators preserve composition. } \} \\
& \mathcal{F}\,(f^{\cup} \circ \preceq) \\
= & \quad \{ \text{ Shunting (A.12). } \} \\
& \mathcal{F}\,(\sqsubseteq \circ\, g) \\
= & \quad \{ \text{ Relators preserve composition. } \} \\
& \mathcal{F} \sqsubseteq \circ\, \mathcal{F}\, g
\end{array}
$$

□

A.2.2 Partial orders

Question. Assuming that $(\mathcal{A}, \sqsubseteq)$ is a *partial order* and $\mathcal{F}$ a relator, under which conditions $(\mathcal{F}\,\mathcal{A}, \mathcal{F} \sqsubseteq)$ is also a partial order?

$\mathcal{F}$ preservation properties. As we have proved above, $\mathcal{F}$ preserves reflexivity and transitivity. Therefore, we just have to prove that $\mathcal{F}$ preserves anti-symmetry.

The anti-symmetry of a relation $\sqsubseteq$ is expressed as

$$\sqsubseteq \cap \sqsubseteq^{\cup} \quad \subseteq \quad id \tag{A.14}$$

We reason,

Proof

$$
\begin{array}{ll}
& \mathcal{F} \sqsubseteq \cap\, (\mathcal{F} \sqsubseteq)^{\cup} \subseteq id \\
\Leftrightarrow & \quad \{ \text{ Relators preserve converse. } \} \\
& \mathcal{F} \sqsubseteq \cap\, \mathcal{F} \sqsubseteq^{\cup} \subseteq id \\
\Leftrightarrow & \quad \{ \text{ Relators preserve identity. } \} \\
& \mathcal{F} \sqsubseteq \cap\, \mathcal{F} \sqsubseteq^{\cup} \subseteq \mathcal{F} id
\end{array}
$$

$$
\begin{array}{ll}
\Leftrightarrow & \quad \{ \text{ Assume } \mathcal{F} \sqsubseteq \cap \mathcal{F} \sqsubseteq^{\cup} = \mathcal{F}\,(\sqsubseteq \cap \sqsubseteq^{\cup}). \ \} \\
& \mathcal{F}\,(\sqsubseteq \cap \sqsubseteq^{\cup}) \subseteq \mathcal{F}\,id \\
\Leftarrow & \quad \{ \text{ Monotonicity of relators. } \} \\
& \sqsubseteq \cap \sqsubseteq^{\cup} \subseteq id \\
\Leftrightarrow & \quad \{ \text{ Assumption (A.14). } \} \\
& \top
\end{array}
$$

□

Thus, we conclude that $\mathcal{F}$ preserves anti-symmetry if it distributes over binary relation meet, i.e.,

$$\mathcal{F}\sqsubseteq \cap \mathcal{F}\sqsubseteq^{\cup} \quad = \quad \mathcal{F}(\sqsubseteq \cap \sqsubseteq^{\cup}) \tag{A.15}$$

A.3 Homomorphic image

A.3.1 Preorders

Question. Given a Galois connection $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ between *preordered sets* $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$, that is,

$$f^{\cup} \circ \preceq \quad = \quad \sqsubseteq \circ\, g \tag{A.16}$$

and two functions h and h' such that the following equalities hold

$$\leqslant \quad \overset{def}{=} \quad h^{\cup} \circ \sqsubseteq \circ\, h \tag{A.17}$$
$$\subseteq \quad \overset{def}{=} \quad h'^{\cup} \circ \preceq \circ\, h' \tag{A.18}$$
$$f \circ h \quad = \quad h' \circ f' \tag{A.19}$$
$$g \circ h' \quad = \quad h \circ g' \tag{A.20}$$

prove that $(\mathcal{C}, \leqslant) \xleftarrow{(f',g')} (\mathcal{D}, \subseteq)$ is a Galois connection, i.e.,

$$f'^{\cup} \circ \subseteq \quad = \quad \leqslant \circ\, g' \tag{A.21}$$

Proof

$$
\begin{array}{ll}
& f'^{\cup} \circ \subseteq \\
= & \qquad \{ \text{ Definition (A.18). } \} \\
& f'^{\cup} \circ h'^{\cup} \circ \preceq \circ h' \\
= & \qquad \{ \text{ Contravariance. } \} \\
& (h' \circ f')^{\cup} \circ \preceq \circ h' \\
= & \qquad \{ \text{ Homomorphism (A.19). } \} \\
& (f \circ h)^{\cup} \circ \preceq \circ h' \\
= & \qquad \{ \text{ Contravariance. } \} \\
& h^{\cup} \circ f^{\cup} \circ \preceq \circ h' \\
= & \qquad \{ \text{ Shunting (A.16). } \} \\
& h^{\cup} \circ \sqsubseteq \circ g \circ h' \\
= & \qquad \{ \text{ Homomorphism (A.20). } \} \\
& h^{\cup} \circ \sqsubseteq \circ h \circ g' \\
= & \qquad \{ \text{ Definition (A.17). } \} \\
& \leqslant \circ g'
\end{array}
$$

□

A.3.2 Partial orders

Question. Under which conditions, from a partial order $\sqsubseteq$ and a function h, can we build an order $\preceq$, defined as

$$\preceq \;\; \overset{def}{=} \;\; h^{\cup} \circ \sqsubseteq \circ h \qquad \text{(A.22)}$$

which is also a partial order? In other words, under which conditions does h preserve anti-symmetry?

Auxiliary result. To answer this question we need to introduce the modular identity rule

$$R \circ S \cap T \quad \subseteq \quad R \circ (S \cap R^{\cup} \circ T) \tag{A.23}$$

together with two of its corollaries [Backhouse, 2004]:

$$(R \circ S) \cap (R \circ T) = R \circ (S \cap T) \quad \Leftarrow \quad R^{\cup} \circ R \circ T \subseteq T \ \vee \ R^{\cup} \circ R \circ S \subseteq S \tag{A.24}$$

$$(S \circ R) \cap (T \circ R) = (S \cap T) \circ R \quad \Leftarrow \quad T \circ R \circ R^{\cup} \subseteq T \ \vee \ S \circ R \circ R^{\cup} \subseteq S \tag{A.25}$$

for all relations R, S and T with the right types.

Equations (A.24) and (A.25) are interesting because they express a distributivity property of composition through relation meet. However, the side-conditions are somewhat complicated and discharging these may lead to some lengthy derivations during a proof. Using some calculus, we can derive a stronger but more amenable formulation:

Proof

$$\begin{array}{ll}
 & R^{\cup} \circ R \circ T \subseteq T \vee R^{\cup} \circ R \circ S \subseteq S \\
\Leftarrow & \qquad \{ \text{ Monotonicity. } \} \\
 & R^{\cup} \circ R \subseteq id \vee R^{\cup} \circ R \subseteq id \\
\Leftrightarrow & \qquad \{ \text{ Idempotence. } \} \\
 & R^{\cup} \circ R \subseteq id
\end{array}$$

□

Proof

$$\begin{array}{ll}
 & T \circ R \circ R^{\cup} \subseteq T \vee S \circ R \circ R^{\cup} \subseteq id \\
\Leftarrow & \qquad \{ \text{ Monotonicity. } \} \\
 & R \circ R^{\cup} \subseteq id \vee R \circ R^{\cup} \subseteq S \\
\Leftrightarrow & \qquad \{ \text{ Idempotence. } \} \\
 & R \circ R^{\cup} \subseteq id
\end{array}$$

□

Although stronger, these conditions correspond to a formulation of two important properties of relations: $R^{\cup} \circ R \subseteq id$ means that R is injective and $R \circ R^{\cup} \subseteq id$ means that R is simple, i.e., a partial function (Section 4.1). Thus, we can formulate (A.24) and (A.25) as

$$(R \circ S) \cap (R \circ T) = R \circ (S \cap T) \quad \Leftarrow \quad R^{\cup} \circ R \subseteq id \tag{A.26}$$

$$(S \circ R) \cap (T \circ R) = (S \cap T) \circ R \quad \Leftarrow \quad R \circ R^{\cup} \subseteq id \tag{A.27}$$

Main proof. Back to the main proof, given an partial order $\sqsubseteq$ on $\mathcal{A}$, a function $\mathcal{A} \xleftarrow{h} \mathcal{B}$ and a preorder $\preceq$ defined as in (A.22), we derive in which conditions $\preceq$ is also anti-symmetric, i.e., in which circumstances h preserves anti-symmetry:

Proof

$$
\begin{array}{ll}
 & \preceq \cap \preceq^{\cup} \subseteq id \\
\Leftrightarrow & \quad \{ \text{ Definition (A.22). } \} \\
 & h^{\cup} \circ \sqsubseteq \circ h \cap (h^{\cup} \circ \sqsubseteq \circ h)^{\cup} \subseteq id \\
\Leftrightarrow & \quad \{ \text{ Contravariance and involution of converse. } \} \\
 & h^{\cup} \circ \sqsubseteq \circ h \cap (h^{\cup} \circ \sqsubseteq^{\cup} \circ h) \subseteq id \\
\Leftrightarrow & \quad \{ \text{ (A.27) with } R := h,\ S := h^{\cup} \circ \sqsubseteq \text{ and } T := h^{\cup} \circ \sqsubseteq^{\cup}. \\
 & \quad\ \ \text{The condition } h \circ h^{\cup} \subseteq id \text{ holds because } h \text{ is a function. } \} \\
 & (h^{\cup} \circ \sqsubseteq \cap\, h^{\cup} \circ \sqsubseteq^{\cup}) \circ h \subseteq id \\
\Leftrightarrow & \quad \{ \text{ (A.26) with } R := h^{\cup},\ S := \sqsubseteq \text{ and } T := \sqsubseteq^{\cup}. \\
 & \quad\ \ \text{The condition } (h^{\cup})^{\cup} \circ h^{\cup} \subseteq id \Leftrightarrow h \circ h^{\cup} \subseteq id \text{ holds because } h \text{ is a function. } \} \\
 & (h^{\cup} \circ (\sqsubseteq \cap \sqsubseteq^{\cup}) \circ h) \subseteq id \\
\Leftarrow & \quad \{ \text{ Anti-symmetry of } \sqsubseteq \text{ and monotonicity of composition. } \} \\
 & h^{\cup} \circ h \subseteq id \\
\Leftrightarrow & \quad \{ \text{ Definition. } \} \\
 & h \text{ is injective.}
\end{array}
$$

□

We conclude that h preserves anti-symmetry if it is an injective function.

A.4 Shunting of Galois connections

Recall the two equivalent definitions of the $\trianglelefteq$ ordering on Galois connections given in Section 5.4 (Equations (5.17) and (5.18)):

$$(\mathcal{A}, \sqsubseteq) \xleftarrow{(f_1, g_1)} (\mathcal{B}, \preceq) \ \trianglelefteq\ (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_2, g_2)} (\mathcal{B}, \preceq) \quad \Leftrightarrow \quad f_1 \mathrel{\dot{\preceq}} f_2 \tag{A.28}$$

$$(\mathcal{A}, \sqsubseteq) \xleftarrow{(f_1, g_1)} (\mathcal{B}, \preceq) \ \trianglelefteq\ (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_2, g_2)} (\mathcal{B}, \preceq) \quad \Leftrightarrow \quad g_2 \mathrel{\dot{\sqsubseteq}} g_1 \tag{A.29}$$

Question. Let α_1, α_2 and α_3 be Galois connections. We want to prove that Galois connections enjoy a kind of shunting property:

$$\alpha_1^{\cup} \circ \alpha_2 \trianglelefteq \alpha_3 \quad \Leftrightarrow \quad \alpha_2 \trianglelefteq \alpha_1 \circ \alpha_3 \tag{A.30}$$

Types. In order to the above expression be well-defined, Galois connections α_1, α_2 and α_3 must be correctly typed. Observing (A.30), the definition of $\trianglelefteq$ requires that the domains of the upper adjoints of α_2 and α_3 must coincide, although the domains of their lower adjoints can be different. Thus, for any *preorders* $(\mathcal{A}, \sqsubseteq)$, $(\mathcal{B}, \preceq)$ and $(\mathcal{C}, \leqslant)$, we have that

$$\alpha_2 = (\mathcal{B}, \preceq) \xleftarrow{(f_2,g_2)} (\mathcal{C}, \leqslant) \quad \text{and} \quad \alpha_3 = (\mathcal{A}, \sqsubseteq) \xleftarrow{(f_3,g_3)} (\mathcal{C}, \leqslant)$$

The difficulty arises because both Galois connection α_1 and its converse need to be composed. Observing the right-hand side of (A.30), the definition of composition and $\trianglelefteq$ requires that $\alpha_1 = (\mathcal{B}, \preceq) \xleftarrow{(f_1,g_1)} (\mathcal{A}, \sqsubseteq)$. Consequently, the converse Galois connection should be $\alpha_1^{\cup} = (\mathcal{A}, \sqsupseteq) \xleftarrow{(g_1,f_1)} (\mathcal{B}, \succeq)$. However, looking at the left-hand side of (A.30) we conclude that $\alpha_1^{\cup} = (\mathcal{A}, \sqsubseteq) \xleftarrow{(g_1,f_1)} (\mathcal{B}, \preceq)$. This means that, in order to (A.30) be well-defined, we must have $(\mathcal{A}, \sqsubseteq) = (\mathcal{A}, \sqsupseteq)$ and $(\mathcal{B}, \preceq) = (\mathcal{B}, \succeq)$, i.e., $\sqsubseteq$ and $\preceq$ must be symmetric.

Two cases should be distinguished, according with Figure 4.1:

1. When $\sqsubseteq$ and $\preceq$ are *preorders*, the additional symmetry requirement means that they must be *equivalence* relations;

2. When $\sqsubseteq$ and $\preceq$ are *partial orders*, the additional symmetry requirement means that they boil down to the identity order (equality). Thus, α_1 must be an isomorphism (Section 5.6).

From the point-free definition of α_1,

$$f_1^{\cup} \circ \sqsubseteq \quad = \quad \preceq \circ\, g_1 \tag{A.31}$$

we can apply converses $g_1^{\cup} \circ \preceq^{\cup} = \sqsubseteq^{\cup} \circ f_1$ and use the symmetry property of $\sqsubseteq$ and $\preceq$ to get the point-free definition for $\alpha_1^{\cup}$:

$$g_1^{\cup} \circ \preceq \quad = \quad \sqsubseteq \circ f_1 \tag{A.32}$$

Proof

$$\alpha_1^{\cup} \circ \alpha_2 \trianglelefteq \alpha_3$$

$\Leftrightarrow$ $\{$ Definition of $\trianglelefteq$ (A.29). f_1 is the upper adjoint of $\alpha 1^{\cup}$ (A.32). $\}$

$$g_3 \mathrel{\dot{\sqsubseteq}} f_1 \circ g_2$$

$\Leftrightarrow$ $\{$ Lifted order (4.56). $\}$

$$g_3 \subseteq \sqsubseteq \circ f_1 \circ g_2$$

$\Leftrightarrow$ $\{$ Shunting (A.32). $\}$

$$g_3 \subseteq g_1^{\cup} \circ \preceq \circ g_2$$

$\Leftrightarrow$ $\{$ Shunting of functions (6.8). $\}$

$$g_1 \circ g_3 \subseteq \preceq \circ g_2$$

$\Leftrightarrow$ $\{$ Lifted order (4.56). $\}$

$$g_1 \circ g_3 \mathrel{\dot{\preceq}} g_2$$

$\Leftrightarrow$ $\{$ Definition of $\trianglelefteq$ (A.29). g_1 is the upper adjoint of $\alpha 1$ (A.31). $\}$

$$\alpha_2 \trianglelefteq \alpha_1 \circ \alpha_3$$

$\square$

Thus, (A.30) holds when α_1 is defined on equivalence relations. Moreover, it holds in the particular case when α_1 is an isomorphism.

Appendix B

Quantifier rules

The quantifier rules we present here are taken from [Backhouse, 2004] which should be inspected for more details.

Symbols $\bigoplus$ and $\bigotimes$ denote, respectively, the quantifiers associated with the binary operators $\oplus$ and $\otimes$, both associative and commutative. $1_\oplus$ is the unit element of $\oplus$ and $1_\otimes$ is the unit element of $\otimes$.

B.1 Bound variables

Dummy renaming.

$$\langle \bigoplus j :\ R\ : T\rangle = \langle \bigoplus k :\ R[j := k]\ : T[j := k]\rangle \tag{B.1}$$

Nesting.

$$\langle \bigoplus j, js :\ R \wedge S\ : T\rangle = \langle \bigoplus j :\ R\ : \langle \bigoplus js :\ S\ : T\rangle\rangle \tag{B.2}$$

Rearranging.

$$\langle \bigoplus j, k :\ R\ : T\rangle = \langle \bigoplus k, j :\ R\ : T\rangle \tag{B.3}$$

B.2 Range part

Empty range.

$$\langle \bigoplus k : \ \mathsf{false} \ : T \rangle = 1_{\oplus} \tag{B.4}$$

One-point.

$$\langle \bigoplus k : \ k = e \ : T \rangle = T[k := e] \tag{B.5}$$

Splitting.

$$\langle \bigoplus k : \ P \ : T \rangle \oplus \langle \bigoplus k : \ Q \ : T \rangle = \\ \langle \bigoplus k : \ P \vee Q \ : T \rangle \oplus \langle \bigoplus k : \ P \wedge Q \ : T \rangle \tag{B.6}$$

Splitting (Idempotent case). If $\oplus$ is idempotent, then:

$$\langle \bigoplus k : \ P \ : T \rangle \oplus \langle \bigoplus k : \ Q \ : T \rangle = \langle \bigoplus k : \ P \vee Q \ : T \rangle \tag{B.7}$$

Splitting (General idempotent case). If $\oplus$ is idempotent, then:

$$\langle \bigoplus j : \ R \ : \langle \bigoplus k : \ S \ : T \rangle \rangle = \langle \bigoplus k : \ \langle \exists\, j : \ R \ : S \rangle \ : T \rangle \tag{B.8}$$

B.3 Trading

Trading.

$$\langle \bigoplus k \in \mathcal{K} : \ P \wedge Q \ : T \rangle = \langle \bigoplus k \in \{\, k \in \mathcal{K} :: P \} : \ Q \ : T \rangle \tag{B.9}$$

Trading.

$$\langle \bigoplus k : \ P \wedge Q \ : T \rangle = \langle \bigoplus k : \ Q \ : \texttt{if}\ P \rightarrow T \ \Box \ \neg P \rightarrow 1_{\oplus}\ \texttt{fi} \rangle \tag{B.10}$$

B.4 Term part

Rearranging.

$$\langle \bigoplus k :\ R\ : T_0 \oplus T_1 \rangle = \langle \bigoplus k :\ R\ : T_0 \rangle \oplus \langle \bigoplus k :\ R\ : T_1 \rangle \tag{B.11}$$

B.5 Distributivity

Distributivity. If f is a function such that $f\ 1_{\oplus} = 1_{\otimes}$ and $f\ (x \oplus y) = f\ x \otimes f\ y$, for all x and y, then:

$$f\ \langle \bigoplus k :\ R\ : T \rangle = \langle \bigotimes k :\ R\ : f\ T \rangle \tag{B.12}$$

Bibliography

C. Aarts, R. C. Backhouse, P. Hoogendijk, E. Voermans, and J. van der Woude. A relational theory of datatypes. Available from `www.cs.nott.ac.uk/~rcb/papers`, December 1992.

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

J. C. Almeida. Program verification in Coq, 2008. Lecture notes for the MAP-I course on *Program Semantics, Verification, and Construction* available from `http://twiki.di.uminho.pt`.

T. L. Alves, P. F. Silva, J. Visser, and J. N. Oliveira. Strategic term rewriting and its application to a VDM-SL to SQL conversion. In I. J. Hayes, J. Fitzgerald, and A. Tarlecki, editors, *FM 2005: Formal Methods: International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22*, volume 3582 of *LNCS*, pages 399–414. Springer, 2005. ISBN 3-540-27882-6.

T. L. Alves, P. F. Silva, and J. Visser. Constraint-aware schema transformation. In *Ninth International Workshop on Rule-Based Programming (RULE 2008)*, 2008.

D. Angluin. How to prove it. *SIGACT News*, 15(1), 1983. ISSN 0163-5700. URL `http://portalparts.acm.org/1010000/1008908/fm/frontmatter.pdf`.

F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *ICFP '02: Proc. 7th ACM*

SIGPLAN Int. Conf. Functional programming, pages 157–166. ACM Press, 2002. ISBN 1-58113-487-8. doi: http://doi.acm.org/10.1145/581478.581494.

R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

K. Backhouse and R. C. Backhouse. Safety of abstract interpretations for free, via logical relations and Galois connections. *Sci. Comput. Program.*, 51(1-2):153–196, 2004.

R. C. Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS8810, Department of Mathematics and Computing Science, University of Groningen, 1988.

R. C. Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, 1989.

R. C. Backhouse. Galois connections and fixed point calculus. In Backhouse et al. [2002], pages 89–148. ISBN 3-540-43613-8.

R. C. Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley & Sons, Inc., New York, NY, USA, 2003. ISBN 0470848820.

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

R. C. Backhouse, R. L. Crole, and J. Gibbons, editors. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures*, volume 2297 of *Lecture Notes in Computer Science*, 2002. Springer. ISBN 3-540-43613-8.

J. W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

R. Behnke, R. Berghammer, E. Meyer, and P. Schneider. RELVIEW — A system for calculating with relations and relational programming. *Fundamental Approaches to Software Engineering*, pages 318–321, 1998. URL `http://www.springerlink.com/content/00ue9fk96952bfxx`.

R. Bělohlávek. Fuzzy Galois connections and fuzzy concept lattices: from binary relations to conceptual structures. In *Discovering the world with fuzzy logic*, pages

462–494. Physica-Verlag GmbH, Heidelberg, Germany, 2000. ISBN 3-7908-1330-3.

R. Bělohlávek. Lattices of fixed points of fuzzy Galois connections. *Mathematical Logic Quarterly*, 47(1):111–116, Jan. 2001.

Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer, 2004. ISBN 3-540-20854-2 (hardcover).

R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computer Science*, volume 55 of *NATO ASI Series F*, pages 151–216. Springer-Verlag, 1988. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.

R. S. Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall International, 1998. C.A.R. Hoare Series Editor.

R. S. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C.A.R. Hoare Series Editor.

R. S. Bird, J. Gibbons, and S.-C. Mu. Algebraic methods for optimization problems. In Backhouse et al. [2002], pages 281–308. ISBN 3-540-43613-8.

G. Birkhoff. On the structure of abstract algebras. In *Proceedings of the Cambridge Philosophical Society*, volume 31, pages 433–454, October 1935.

G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, Rhode Island, revised edition, 1940.

J. Bohórquez and C. Rocha. Towards the effective use of formal logic in the teaching of discrete math. *Information Technology Based Higher Education and Training, 2005. ITHET 2005. 6th International Conference on*, pages S3C/1–S3C/8, July 2005. doi: http://dx.doi.org/10.1109/ITHET.2005.1560330.

P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4:35–50, 1996. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/S1571-0661(04)00032-5. First International Workshop on Rewriting Logic and its Applications (RWLW96).

F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20:10–19, 1987.

R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. doi: http://doi.acm.org/10.1145/581690.581698.

A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In F. Honsell, editor, *Foundations of Software Science and Computation Structures, ETAPS'2001*, Lecture Notes in Computer Science, pages 166–180, Genova, Italy, April 2001. Springer-Verlag.

H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. *Electr. Notes Theor. Comput. Sci.*, 86(4), 2003.

M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13 (6):377–387, June 1970.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 73(2/3), 1988.

P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

P. Cousot. Abstract interpretation based formal methods and future challenges. In R. Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001. ISBN 3-540-41635-8.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

A. Cunha, J. S. Pinto, and J. Proença. A framework for point-free program transformation. In A. Butterfield, C. Grelck, and F. Huch, editors, *IFL*, volume 4015 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005. ISBN 3-540-69174-X.

A. Cunha, J. N. Oliveira, and J. Visser. Type-safe two-level data transformation — with derecursivation and dynamic typing. Technical Report DI-PURe-06.03.01, Departamento de Informática, Universidade do Minho, Campus Gualtar, Braga, Portugal, March 2006a.

A. Cunha, J. N. Oliveira, and J. Visser. Type-safe two-level data transformation. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2006b. ISBN 3-540-37215-6. doi: http://dx.doi.org/10.1007/11813040_20.

B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Mathematical Textbooks. Cambridge University Press, 1990.

A. De Morgan. On the syllogism, no. IV, and on the logic of relations. *Transactions of the Cambridge Philosophical Society*, 10:331–358, 1860.

A. De Morgan. *"On the syllogism" and other logical writings*. Yale University Press, 1966. Collection of the most important works of Augustus De Morgan (1806—1871) on logic.

D. Delahaye and M. Mayero. Dealing with algebraic expressions over a field in Coq using Maple. *J. Symb. Comput.*, 39(5):569–592, 2005. doi: http://dx.doi.org/10.1016/j.jsc.2004.12.004.

K. Denecke, M. Erné, and S. L. Wismath, editors. *Galois connections and applications*. Springer, 2004.

N. Dershowitz. A taste of rewrite systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, pages 199–228, London, UK, 1993. Springer-Verlag. ISBN 3-540-56883-2.

E. W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice Hall, 1976.

E. W. Dijkstra. Why preorders are beautiful. URL `http://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1102.PDF`. circulated privately, June 1991.

E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972. Turing Award lecture.

E. W. Dijkstra and W. H. Feijen. *A Method of Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. ISBN 0201175363.

E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

M. Erné, J. Koslowski, A. Melton, and G. E. Strecker. A primer on Galois connections. In A. R. Todd, editor, *Papers on general topology and applications (Madison, WI, 1991)*, volume 704 of *Annals of the New York Academy of Sciences*, pages 103–125, New York, 1993. New York Acad. Sci.

J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007. ISBN 978-3-540-73367-6. doi: http://dx.doi.org/10.1007/978-3-540-73368-3_21.

P. J. Freyd and A. Ščedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990.

M. F. Frias. Independence of the axiomatization of fork. *Algebra Universalis*, 39(3): 211–215, 1998. doi: http://dx.doi.org/10.1007/s000120050076.

M. F. Frias, A. M. Haeberer, and P. A. S. Veloso. Fork algebras are representable. *Bulletin of the Section of Logic*, 24(2):64–75, 1995.

M. F. Frias, A. M. Haeberer, and P. A. S. Veloso. A finite axiomatization for fork algebras. *Logic Journal of the IGPL*, 5(3), 1997.

M. F. Frias, C. L. Pombo, and N. Aguirre. An equational calculus for Alloy. In J. Davies, W. Schulte, and M. Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2004a. ISBN 3-540-23841-7.

M. F. Frias, P. A. S. Veloso, and G. A. Baum. Fork Algebras: Past, Present and Future. *JoRMiCS*, 1:181–216, 2004b.

E. Galois. Memoire sur les conditions de résolutilité des équations par radicaux. *Journal de mathématiques pures et appliquées*, 1846.

E. Galois. *Oeuvres mathematiques d'Évariste Galois*. Gauthier-Villars et Fils, 1897.

B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin – Heidelberg, 1999.

I. Gnaedig and H. Kirchner. Termination of rewriting under strategies. *ACM Trans. Comput. Logic*, 10(2):1–52, 2009. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/1462179.1462182.

D. Gries and F. B. Schneider. Equational propositional logic. *Inf. Process. Lett.*, 53 (3):145–152, 1995. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(94)00198-8.

C. Hankin. Principles of program analysis. Lecture Notes of the International Summer School On Applied Semantics, Frauenchiemsee, Germany, September 2005.

J. Hartmanis and R. E. Stearns. Pair algebra and its application to automata theory. *Information and Control*, 7(4):485–507, 1964.

J. Hartmanis and R. E. Stearns. *Algebraic structure theory of sequential machines*. Series in Applied Mathematics. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1966.

J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF—Reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/71605.71607.

R. Hinze and A. Löh. Guide2lhs2tex (for version 1.13), February 2008.

R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In *Proc. 8th Int. Symp. on Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2006.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12 (10):576–580, 1969.

P. Höfner and G. Struth. On automating the calculus of relations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2008. ISBN 978-3-540-71069-1. doi: http://dx.doi.org/10.1007/978-3-540-71070-7_5.

R. R. Hoogerwoord. Formality works. *Information Processing Letters*, 77(2-4): 137–142, 2001. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/S0020-0190(00) 00210-6.

J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96. Springer, 1995. ISBN 3-540-59451-5.

E. V. Huntington. A New Set of Independent Postulates for the Algebra of Logic with Special Reference to Whitehead and Russell's Principia Mathematica. *Proceedings of the National Academy of Sciences of the United States of America*, 18(2):179–180, 1932. URL `http://www.pnas.org/content/18/2/179.short`.

G. Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3):323–343, 1992.

ISO. *Information Technology – Database languages – SQL.* Reference number ISO/IEC 9075:1992(E), Nov. 1992.

D. Jackson. *Software abstractions: logic, language, and analysis.* The MIT Press, Cambridge Mass., 2006. ISBN 0-262-10114-9.

M. P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming,* *First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer, 1995. ISBN 3-540-59451-5.

T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science, 2001. Proc. Workshop on Language Descriptions, Tools and Applications.

R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003. Also available as arXiv technical report cs.PL/0205018.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

R. Lämmel and J. Visser. Strategic polymorphism requires just two combinators! Technical Report cs.PL/0212048, arXiv, 2002. An early version was published in the informal pre-proceedings of IFL 2002.

R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

L. Lamport. How to write a proof. *The American Mathematical Monthly*, 102(7): 600–608, 1995. URL `http://www.jstor.org/stable/2974556`.

D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

V. Lifschitz. On calculational proofs. *Ann. Pure Appl. Logic*, 113(1-3):207–224, 2001.

S. P. Luttik and E. Visser. Specification of rewriting strategies. In *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing*. Springer-Verlag, 1997.

R. C. Lyndon. The representation of relational algebras. *Ann. of Math. (2)*, 51:707–729, 1950. ISSN 0003-486X.

S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

R. D. Maddux. Relation-algebraic semantics. *Theor. Comput. Sci.*, 160(1&2):1–85, 1996.

Z. Manna, S. Ness, and J. Vuillemin. Inductive methods for proving properties of programs. *Commun. ACM*, 16(8):491–502, 1973. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/355609.362336.

P. Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984. URL `http://www.ams.org/mathscinet-getitem?mr=769301`.

J. McCarthy. Checking mathematical proofs by computer. In *Proceedings Symposium on Recursive Function Theory*. American Mathematical Society, 1962.

W. McCune. Prover9, June 2009. URL `http://www.cs.unm.edu/~mccune/prover9`.

L. Meertens. Algorithmics – towards programming as a mathematical activity. In J. W. De Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North-Holland, 1986.

E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed. *Workshop on Revival of Dynamic Languages*, 2005. URL `http://research.microsoft.com/~emeijer/Papers/RDL04Meijer.pdf`.

A. Melton, D. A. Schmidt, and G. E. Strecker. Galois connections and computer science applications. In *Proceedings of a tutorial and workshop on Category theory and computer programming*, pages 299–312, New York, NY, USA, 1986. Springer-Verlag New York, Inc. ISBN 0-387-17162-2.

A. Melton, B. S. W. Schröder, and G. E. Strecker. Lagois connections — A counterpart to Galois connections. *Theor. Comput. Sci.*, 136(1):79–107, 1994.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

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,*, Brussels, January 1969. Scientific Affairs Division, NATO, NATO.

C. Necco, J. N. Oliveira, and J. Visser. Extended static checking by strategic rewriting of pointfree relational expressions. Technical Report FAST:07.01, CCTC Research Centre, University of Minho, 2007.

J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Asp. Comput.*, 2(1):1–23, April 1990.

J. N. Oliveira. Functional dependency theory made 'simpler'. Technical Report DI-PURe-05.01.01, DI/CCTC, University of Minho, Gualtar Campus, Braga, 2005. PUReCafé, 2005.01.18 [talk]; available from `http://wiki.di.uminho.pt/twiki/bin/view/Research/PURe/PUReCafe`.

J. N. Oliveira. Transforming data by calculation. In *GTTSE'07*, volume 5235 of *Lecture Notes in Computer Science*, pages 134–195. Springer, 2008.

J. N. Oliveira. Extended static checking by calculation using the pointfree transform. In A. Bove et al., editor, *LerNet ALFA Summer School 2008*, volume 5520 of *Lecture Notes in Computer Science*, pages 195–251. Springer-Verlag, 2009.

J. N. Oliveira and C. J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In MPC'04 *: Seventh International Conference on Mathematics of Program Construction, 12-14 July, 2004, Stirling, Scotland, UK (Organized in conjunction with AMAST'04)*, volume 3125 of *Lecture Notes in Computer Science*, pages 334–356. Springer, 2004.

O. Ore. Galois connexions, 1944. Trans. Amer. Math. Soc., 55:493-513.

D. Park. Fixpoint induction and proofs of program properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 59–78. Edinburgh University Press, 1969.

L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149, 1983.

S. Peyton Jones. Haskell 98: Language and libraries. *J. Funct. Program.*, 13(1):1–255, 2003.

D. Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés*. PhD thesis, Universit'e de Rennes, December 2005.

D. A. Plaisted. Equational reasoning and term rewriting systems. *Handbook of logic in artificial intelligence and logic programming*, 1:274–364, 1993.

A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

V. R. Pratt. Origins of the calculus of binary relations. In *Proc. of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 248–254, Santa Cruz, CA, 1992. IEEE Computer Soc.

V. R. Pratt. The second calculus of binary relations. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *MFCS*, volume 711 of *Lecture Notes in Computer Science*, pages 142–155. Springer, 1993. ISBN 3-540-57182-5.

H. A. Priestley. Ordered sets and complete lattices. In Backhouse et al. [2002], pages 21–78. ISBN 3-540-43613-8.

U. Priss. An FCA interpretation of relation algebra. In R. Missaoui and J. Schmid, editors, *ICFCA*, volume 3874 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2006a. ISBN 3-540-32203-5.

U. Priss. Formal concept analysis in information science. *Annual Review of Information Science and Technology*, 40(1):521–543, 2006b. doi: http://dx.doi.org/10.1002/aris.1440400120.

J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing 83*, pages 513–523, 1983.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002. ISBN 0-7695-1483-9.

D. E. Rydeheard and R. M. Burstall. *Computational Category Theory*. Series in Computer Science. Prentice Hall International, 1988. C.A.R Hoare Series Editor.

J. Schmidt. Beiträge zur Filtertheorie. II. *Math. Nachr.*, 10:197–232, 1953. ISSN 0025-584X.

T. Sheard, J. Hook, and N. Linger. GADTs + Extensible Kinds = Dependent Programming. Technical report, Portland State University, 2005. URL `http://www.cs.pdx.edu/~sheard`.

P. F. Silva and J. N. Oliveira. 'Galculator': functional prototype of a Galois-connection based proof assistant. In *PPDP '08: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 44–55, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-117-0. doi: http://doi.acm.org/10.1145/1389449.1389456.

P. F. Silva, J. C. Almeida, and J. N. Oliveira. Galculator meets Coq. Technical report, CCTC, University of Minho, 2008. (In preparation).

P. F. Silva, J. Visser, and J. N. Oliveira. Galois: A language for proofs using galois connections and fork algebras. In *PLMMS'09: The ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems*, 2009. To appear.

C. Sinz. System description: ARA - an automatic theorem prover for relation algebras. In D. McAllester, editor, *Automated Deduction CADE-17*, number 1831 in LNAI, pages 177–182, Pittsburgh, PA, June 2000. Springer-Verlag.

M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume Volume 149 (Studies in Logic and the Foundations of Mathematics). Elsevier Science Inc., New York, NY, USA, 2006. ISBN 0444520775.

J. M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989. C.A. R. Hoare.

I. Stewart. *Galois Theory*. Chapman & Hall, second edition, 1989.

I. Stewart. *The Problems of Mathematics*. Oxford University Press, second edition, 1992.

A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.

A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*. American Mathematical Society, 1987. ISBN 0821810413. AMS Colloquium Publications, volume 41, Providence, Rhode Island.

S. Thompson. *Haskell—The Craft of Functional Programming*. Addison-Wesley, 1st edition, 1996. ISBN 0-201-40357-9.

J.-P. Tignol. *Galois' Theory of Algebraic Equations*. World Scientific, 2001.

A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936. ISSN 0024-6115.

M. van den Brand, A. van Deursen, J. Heering, H. de Jonge, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC 2001)*, volume 2027 of *LNCS*. Springer-Verlag, 2001.

J. van den Bussche. Applications of Alfred Tarski's ideas in database theory. In *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, pages 20–37, London, UK, 2001. Springer-Verlag. ISBN 3-540-42554-3.

P. A. S. Veloso. On the independence of the axioms for fork algebras. *Bulletin of the Section of Logic*, 26(4):197–209, 1997.

P. A. S. Veloso and A. M. Haeberer. A finitary relational algebra for classical first-order logic. *Bulletin of the Section of Logic*, 20(2):52–62, 1991.

E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *LNCS*, pages 357–361. Springer, May 2001a.

E. Visser and Z. Benaissa. A Core Language for Rewriting. In C. Kirchner and H. Kirchner, editors, *Proc. International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *ENTCS*, Pont-à-Mousson, France, September 1998. Elsevier Science. doi: http://dx.doi.org/10.1016/S1571-0661(05)80027-1.

J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11): 270–282, 2001b. Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications.

B. von Karger. Temporal algebra. In Backhouse et al. [2002], pages 309–385. ISBN 3-540-43613-8.

D. von Oheimb and T. F. Gritzner. RALL: Machine-supported proofs for relation algebra. In W. McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 1997. ISBN 3-540-63104-6.

P. Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA*, pages 113–128, 1985.

P. Wadler. Theorems for free! In *Proc. of 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept. 1989*, pages 347–359. ACM Press, New York, 1989. URL `http://cm.bell-labs.com/cm/cs/who/wadler/papers/free/free.ps.gz`.

P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice, France*, 1990.

S. Wang, L. S. Barbosa, and J. N. Oliveira. A relational model for confined separation logic. In *TASE*, pages 263–270. IEEE Computer Society, 2008. ISBN 978-0-7695-3249-3.

M. Ward and R. P. Dilworth. Residuated lattices. *Trans. Amer. Math. Soc*, 45:335–354, 1939.

Wikibooks. Existentially quantified types, June 2009. URL `http://en.wikibooks.org/wiki/Haskell/Existentially_quantified_types`.

Index