Galois

A Language for Proofs Using Galois Connections and Fork Algebras

Paulo F. Silva
CCTC, University of Minho, Braga
Portugal
paufil@di.uminho.pt

Joost Visser
Software Improvement Group
The Netherlands
j.visser@sig.nl

José N. Oliveira
CCTC, University of Minho, Braga
Portugal
jno@di.uminho.pt

Abstract

Galois is a domain specific language supported by the *Galculator* interactive proof-assistant prototype. *Galculator* uses an equational approach based on Galois connections with indirect equality as an additional inference rule. *Galois* allows for the specification of different theories in a point-free style by using fork algebras, an extension of relation algebras with expressive power of first-order logic. The language offers sub-languages to derive proof rules from Galois connections, to express proof tactics, and to organize axioms and theorems into modular definitions.

In this paper, we describe how the algebraic theory underlying the proof-method drives the design of the *Galois* language. We provide the syntax and semantics of important fragments of *Galois* and show how they are hierarchically combined into a complete language.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; D.3.2 [*Language Classifications*]: Specialized application languages

General Terms Theory, Languages, Design

Keywords Fork algebras, Galois connections, Proof assistant, DSL

1. Introduction

Domain-specific languages (DSLs) are little languages focussed on expressing the concepts of a specific domain (van Deursen et al. 2000). DSLs have been defined and implemented for domains such as text processing (e.g. LaTeX (Lamport 1986)), financial services (e.g. RISLA (van den Brand et al. 1996)), mark-up (e.g. HTML (Berners-Lee and Connolly 1995)), and many more. For mathematicians, DSLs are a standard instrument, in the sense that new mathematical notations are introduced continuously for capturing concepts and proofs in various theoretical domains. Usually, these DSLs are not supported by a compiler or other tools, as is the case for the DSLs mentioned above.

In this paper we will explore the development of a family of DSLs (named *Galois*) for a specific mathematical domain, *viz.* Galois connections (Ore 1944) and we will show the role of these DSLs in associated tool support for theorem proving and proof assistance. This tool, called *Galculator* (= **Gal**ois connection + cal**culator**) (Silva and Oliveira 2008a), takes Galois connections as primitives and exploits their algebraic properties in proofs. Basically, a Galois connection is a pair of functions with "good" preservation properties which connect two domains. 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 (this is known as "shunting"). Galois connections are pervasive across many domains of application (Erné et al. 1993), although they are mostly known in computer science because of their key role in the theory of abstract interpretation (Cousot and Cousot 1977).

Galculator uses an algebraic approach to theorem proving based on fork algebras (Frias et al. 2004), an extension of relation algebras (Tarski and Givant 1987) which has the same expressive and deductive power of first-order logic. Fork algebras require just one inference rule, namely substitution of equals by equals, allowing for equational proofs in a calculational style. This means that essentially there is no difference between *verification* (theorem proving) and *construction* (calculus).

During proofs by symbolic manipulation, sometimes it would be helpful to have some semantic information about the operators at hands. Galois connections provide for such a mechanism: in fact, they establish an implicit (semantic) definition of two functions, each one in terms of the other. Nevertheless, their "shunting" property can be used syntactically in equational proofs for changing the domain of the problem. We show how Galois connections can be integrated with fork algebras and effectively used in proofs by adding the so-called indirect equality principle as an inference rule.

Galois is a strongly typed language: expressions are well-formed only if they are well-typed. This often allows for error detection during reasoning while providing deeper insight about some definitions. This differs from most theorem provers which use types as propositions, exploiting the Curry-Howard isomorphism. Moreover, types can be used to ensure that certain hypothesis are met without having to discharge them explicitly, e.g., in the case of functions, the type ensures that simplicity and totality conditions are met.

Structure of the paper. Section 2 introduces a simple example to help the understanding of the concepts used in this paper. Sections 3 and 4 describe the theoretical background while the main contributions are presented in Sections 5 and 6. In Section 3, fork algebras are described as an extension of relation algebras. It is also discussed the theoretical foundations of the point-free transform which allows to express any first-order formula as a fork algebra expression without variables. Section 4 explains the basic theory

behind Galois connections and their algebra, as well as how to use them together with the indirect equality principle. The combination of Galois connections, indirect equality and fork algebras in proofs is introduced in Section 5. The syntax and semantics of the *Galois* language is presented in Section 6. Section 7 discusses some implementation issues of the language. Section 8 gives an account of the related work. Finally, Section 9 contains the main conclusions and a discussion of future work.

2. Motivating example

In this section, we will clarify the objectives of the *Galois* language by presenting a simple example of a proof about whole division. The explanation is introductory and very lightweight; later sections will provide for deeper understanding of the involved concepts.

Suppose that we want to prove the equality $(a \div b) \div c = a \div (c \times b)$, for $b, c \neq 0$. This proof is trivial when working with real number division (in a field, in general). However, in the domain of natural numbers, multiplicative inverses do not always exist. Looking closer at the properties of whole division we can see that the general property holding for real division can be weakened to inequalities,

$$c \times b \leqslant a \quad \Leftrightarrow \quad c \leqslant a \div b \qquad (b \neq 0) \qquad (1)$$

thus obtaining a property valid in the natural numbers. In fact, Equation (1) establishes a Galois connection between adjoint functions $(\times b)$ and $(\div b)$, for $b \neq 0$. Thus, the proof follows with a universally quantified variable n over natural numbers:

$$\begin{array}{ll} & n \leqslant (a \div b) \div c \\ \Leftrightarrow & \quad \{ \text{ by (1) } \} \\ & n \times c \leqslant a \div b \\ \Leftrightarrow & \quad \{ \text{ (1) again } \} \\ & (n \times c) \times b \leqslant a \\ \Leftrightarrow & \quad \{ \text{ multiplication is associative } \} \\ & n \times (c \times b) \leqslant a \\ \Leftrightarrow & \quad \{ \text{ (1) again } \} \\ & n \leqslant a \div (c \times b) \end{array}$$

That is, every natural number n at most $(a \div b) \div c$ is also at most $a \div (c \times b)$, and vice versa. By the indirect equality principle, the two expressions are equal. 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. $A = B \Leftrightarrow \langle \forall\, x :: x \in A \Leftrightarrow x \in B \rangle$[1] as opposed to, e.g. circular inclusion: $A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$.

This proof illustrates the essence of the use of Galois connections: a problem about whole division is transformed into a problem about multiplication. The fundamental step is the use of the associativity property of multiplication.

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.

[1] We use notation $\langle \forall\, x :\ R\ : T \rangle$ meaning *for all x in the range R such that T holds*. This notation is naturally generalized to other quantifiers.

```
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 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 Eq. (1) 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) using a proof strategy labelled by keyword `once`. Individual steps are combined with sequential composition and follow the same structure as the calculational version.

3. Relations

In this section, we start by presenting binary relations in a set-theoretical perspective. This is useful for giving a natural interpretation of fork algebras in terms of concrete relations. Then, we describe fork algebras as an extension of relation algebras and discuss their interpretation and expressiveness. Finally, the foundations of the point-free transform are presented.

3.1 Binary 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).

Given two sets, $\mathcal{A}$ and $\mathcal{B}$, a *binary relation* R is a subset of their Cartesian product $\mathcal{B} \times \mathcal{A} \stackrel{def}{=} \{\forall\, b, a :\ b \in \mathcal{B} \wedge a \in \mathcal{A}\ :\ (b, a)\}$.

We will write $\mathcal{B} \xleftarrow{R} \mathcal{A}$ to denote such a relation. When $\mathcal{A}$ and $\mathcal{B}$ coincide, the relation is said to be an *endorelation*.

We shall distinguish three special relations: the *empty relation* $\bot$ which does not relate any elements at 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 $\mathcal{B} \times \mathcal{A}$); and the *identity relation* id which relates equal to equal elements (consequently, an endorelation).

The operations on relations are standard extensions of the respective operations on the underlying set of pairs. Thus, the *converse* of a relation $\mathcal{B} \xleftarrow{R} \mathcal{A}$, denoted by $\mathcal{A} \xleftarrow{R^\cup} \mathcal{B}$, is defined as $(a,b) \in R^\cup \stackrel{def}{\Leftrightarrow} (b,a) \in R$. The *meet* (intersection) and *join* (union) of two relations $\mathcal{B} \xleftarrow{R} \mathcal{A}$ and $\mathcal{B} \xleftarrow{S} \mathcal{A}$, are defined, respectively, as $(b,a) \in R \cap S \stackrel{def}{\Leftrightarrow} (b,a) \in R \wedge (b,a) \in S$, and $(b,a) \in R \cup S \stackrel{def}{\Leftrightarrow} (b,a) \in R \vee (b,a) \in S$. 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 $(c,a) \in S \circ R \stackrel{def}{\Leftrightarrow} \langle \exists\, b \in \mathcal{B} :: cSb \wedge bRa \rangle$.

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., $R \subseteq S \stackrel{def}{\Leftrightarrow} \langle \forall\, b, a :: bRa \Rightarrow bSa \rangle$.

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

Functions. Functions are simple and total relations. A relation f is simple if and only if $f^\cup \circ f \subseteq id$ and total if and only if $id \subseteq f^\cup \circ f$. We use uppercase identifiers for general relations and lower case identifiers for the specific case of functions.

Orders. A preorder $\sqsubseteq$ is a reflexive ($id \subseteq \sqsubseteq$) and transitive ($\sqsubseteq \circ \sqsubseteq \subseteq \sqsubseteq$) relation. A partial order is an antisymmetric ($\sqsubseteq \cap \sqsubseteq^\cup \subseteq id$) preorder. Due to their non-symmetric behavior, the standard notation for orderings includes symbols such as $\sqsubseteq$, $\preceq$ and $\leqslant$.

3.2 Relation and fork algebras

Relation algebras. The formalization of relation algebras presented below follows Tarski and Givant (1987). However, our notation is different and follows the traditional notation used for binary relations. Moreover, like Priss (2006), we include all operators in the Boolean algebra reduct, while Tarski and Givant (1987) use a minimal signature.

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

$$(\mathcal{R}, \cup, \cap, \neg, \bot, \top) \text{ is a Boolean algebra} \tag{2}$$
$$r \circ (s \circ t) = (r \circ s) \circ t \tag{3}$$
$$r \circ id = r \tag{4}$$
$$(r^\cup)^\cup = r \tag{5}$$
$$(r \cup s) \circ t = r \circ t \cup s \circ t \tag{6}$$
$$(r \cup s)^\cup = r^\cup \cup s^\cup \tag{7}$$
$$(r \circ s)^\cup = s^\cup \circ r^\cup \tag{8}$$
$$r^\cup \circ \neg(r \circ s) \subseteq \neg s \tag{9}$$

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

Fork algebras. Several axiomatizations of fork algebra exist; the following is adapted from Frias et al. (2004). A *fork algebra* extends a relation algebra by adding a binary operator called *fork*, denoted by ∇, and the following axioms, for any $r, s, t, u \in \mathcal{R}$:

$$r \nabla s = (\pi_1^\cup \circ r) \cap (\pi_2^\cup \circ s) \tag{10}$$
$$(r \nabla s)^\cup \circ (t \nabla u) = (r^\cup \circ t) \cap (s^\cup \circ u) \tag{11}$$
$$\pi_1 \nabla \pi_2 \subseteq id \tag{12}$$

where $\pi_1 \stackrel{def}{=} (id \nabla \top)^\cup$ and $\pi_2 \stackrel{def}{=} (\top \nabla id)^\cup$ are *quasi-projections*. Using the fork operator, it is possible to define a binary product operator in relations $\times$ as

$$r \times s \stackrel{def}{=} (r \circ \pi_1) \nabla (s \circ \pi_2) \tag{13}$$

Interpretation. Relation and fork algebras can be interpreted in terms of proper relation algebras and proper fork algebras, respectively (Tarski and Givant 1987; Frias et al. 2004). These are algebras where the elements are binary relations and operations coincide with their set-theoretical counterparts. By establishing an isomorphism between a relation (corresp. fork) algebra and a proper relation (corresp. fork) algebra, a natural interpretation of the abstract symbols in terms of binary relation is established (Tarski and Givant 1987).

A completeness result (Frias et al. 2004) 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 true properties of binary relations.

Expressiveness. Tarski and Givant (1987) show that relation algebras are equivalent to a three variable fragment of first-order logic. The addition of the fork operator allows fork algebras to overcome lack of expressiveness of relation algebras. 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 correspondent derivation from axioms of fork algebra using equational reasoning. Frias et al. (2004) show that this result can be extended to other logics as well (non-classical), by using some extensions of the basic fork algebra.

3.3 Point-free transform

The idea of abstracting variables from terms has led to the so-called point-free style and the point-free transform (PF-transform for short) (Tarski and Givant 1987; Bird and de Moor 1997; Backus 1978; Oliveira and Rodrigues 2004). The basic principle is to establish an equivalence between a relation operator and its definition in set theory. Thus, we have

$$\langle \forall\, x, y :: x(A \cup B)y \Leftrightarrow xAy \vee xBy \rangle \tag{14}$$
$$\langle \forall\, x, y :: x(A \cap B)y \Leftrightarrow xAy \wedge xBy \rangle \tag{15}$$
$$\langle \forall\, x, y :: x(\neg A)y \Leftrightarrow \neg(xAy) \rangle \tag{16}$$
$$\langle \forall\, x, y :: x(A^\cup)y \Leftrightarrow yAx \rangle \tag{17}$$
$$\langle \forall\, x, y :: x(A \circ B)y \Leftrightarrow \langle \exists\, z :: xAz \wedge zBy \rangle \rangle \tag{18}$$
$$\langle \forall\, x, y :: xidy \Leftrightarrow x = y \rangle \tag{19}$$
$$\langle \forall\, x, y :: x \top y \Leftrightarrow \mathsf{true} \rangle \tag{20}$$
$$\langle \forall\, x, y :: x \bot y \Leftrightarrow \mathsf{false} \rangle \tag{21}$$
$$\langle \forall\, x, y, z :: (x,y)(A \nabla B)z \Leftrightarrow xAz \wedge yBz \rangle \tag{22}$$
$$\langle \forall\, x, y, z, w :: (x,y)(A \times B)(w,z) \Leftrightarrow xAw \wedge yBz \rangle \tag{23}$$

Additionally, extensional equality and inequality of relation is also defined

$$A = B \Leftrightarrow \langle \forall\, x, y :: xAy \Leftrightarrow xBy \rangle \tag{24}$$
$$A \subseteq B \Leftrightarrow \langle \forall\, x, y :: xAy \Rightarrow xBy \rangle \tag{25}$$

Frias et al. (2004) discuss that one can perform the PF-transform by manipulating a first-order formula until reaching the definitional

form and then replacing it by the corresponding relation operator. In alternative, a mapping can be used to algorithmically perform the translation; however, the result terms may not be amenable to use in further derivations (Frias et al. 2004).

Using the definitions, more complex translation rules that encompass recurring patterns can be derived. One particular rule which is specially helpful in removing variables from expressions involving functions is

$$\langle \forall\, b, a :: (f\, b)\ R\ (g\, a) \quad \Leftrightarrow \quad b(f^{\cup} \circ R \circ g)a \rangle \tag{26}$$

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. In fact, the point-free style is mostly based on composition which becomes the main "glue" among terms.

4. Galois connections and indirect equality

Galois connections' properties are useful for proofs. Nevertheless, they are mostly used for changing the domain of a problem by means of a "shunting" rule. This style is well-suited for proofs by indirect equality.

This section begins with an overview of Galois connections and their algebraic properties. Then, Section 4.2 introduces indirect equality and how it can be used with Galois connections in proofs.

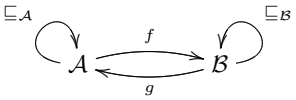
4.1 Galois connections

Recall the concept of a preorder (reflexive and transitive relation). 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 $b \in \mathcal{B}$:

$$f\ a \sqsubseteq_{\mathcal{B}} b \quad \Leftrightarrow \quad a \sqsubseteq_{\mathcal{A}} g\ b\,. \tag{27}$$

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

We can display Galois connections using the graphical notation introduced in (Silva and Oliveira 2008a)



which we in-line in text by writing $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}) \xleftarrow{(f,g)} (\mathcal{B}, \sqsubseteq_{\mathcal{B}})$. Both notations represent the source domain of the lower adjoint on the left.

Partial orders. In the above definition, Galois connections are required to be defined between preordered sets. However, partial orders (anti-symmetric preorders) are usually needed, since preorders are too "weak" for most applications such as indirect equality.

Sections and families of Galois connections. Although Galois connections' adjoints are unary functions, 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 $(a\theta)x = a\ \theta\ x$ and $(\theta b)y = y\ \theta\ b$, respectively. If Eq. (27) holds when we replace (θc) for f then $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}) \xleftarrow{(\theta c, \phi c)} (\mathcal{B}, \sqsubseteq_{\mathcal{B}})$ is a Galois connection. Thus, instead of having just one Galois connection, we build a family of Galois connections indexed by the frozen argument. The definition for the case of right section is analogous.

Building new connections. 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 *new* connections out of existing ones.

The simplest of all Galois connections is the identity, $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}) \xleftarrow{(id,id)} (\mathcal{A}, \sqsubseteq_{\mathcal{A}})$, where adjoints are instances of the polymorphic identity function id. Moreover, 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 $(\mathcal{A}, \sqsubseteq) \xleftarrow{(h \circ f, g \circ k)} (\mathcal{C}, \leqslant)$. (Note how adjoints compose in reverse order.) Composition is an associative operation and the identity Galois connection is its unit.

The particular case in which both orders are equalities boils down to both adjoints being isomorphisms (bijections). The converse combinator on Galois connections switches adjoints while inverting the orders, i.e., from $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ one builds the converse connection $(\mathcal{B}, \succeq) \xleftarrow{(g,f)} (\mathcal{A}, \sqsupseteq)$.

Moreover, every relator $\mathcal{F}$[3] preserves Galois connections. Therefore, from $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ one infers, for every such relator, $(\mathcal{F}\mathcal{A}, \mathcal{F}\sqsubseteq) \xleftarrow{(\mathcal{F}f, \mathcal{F}g)} (\mathcal{F}\mathcal{B}, \mathcal{F}\preceq)$. When $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$ are partial orders, the relator $\mathcal{F}$ must distribute through binary intersections for this property to hold (Backhouse and Backhouse 2004). This construction extends 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.

Algebra of Galois connections. Using the above operations, we can define an algebra of Galois connections as a tuple $(\mathcal{G}, \circ, id, ^{\cup})$ satisfying the following axioms, for any $g, h, j \in \mathcal{G}$:

$$g \circ (h \circ j) \quad = \quad (g \circ h) \circ j \tag{28}$$

$$g \circ id \quad = \quad g \quad = \quad id \circ g \tag{29}$$

$$(g \circ h)^{\cup} = h^{\cup} \circ g^{\cup} \tag{30}$$

$$(g^{\cup})^{\cup} = g \tag{31}$$

This structure is just a composition monoid $(\mathcal{G}, \circ, id)$ with a converse operation. Clearly, Galois connections are models for algebras of Galois connections, where the validity of these axioms when the elements of $\mathcal{G}$ are interpreted as Galois connections can be easily verified using the definitions.

Applications. Abstract interpretation (Cousot and Cousot 1977; Cousot 2001; Backhouse and Backhouse 2004) is perhaps the most well-known application of Galois connections, in which they are used to build analysis. Their algebra allows for combining analysis while ensuring, by construction, the preservation of certain properties. Another relevant area where Galois connections play a central role is that of formal concept analysis (Ganter and Wille 1999).

However, Galois connections are pervasive: references (Backhouse et al. 2002; Backhouse 2004; Erné et al. 1993; Denecke et al. 2004; Melton et al. 1986; Wang et al. 2008) give an extensive account of examples arising from several domains. Among such examples, applications can be found, e.g., in arithmetics, data refinement, temporal algebra, separation logic, weakest liberal precondition calculus and residuation theory. The later includes, for instance, residuated semigroups, formal languages, residuated lattices, Heyting algebras, Boolean algebras, regular algebras and relation algebras.

[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. (2000)

[3] Relators are the relational counterpart of functors. See e.g. (Aarts et al. 1992; Bird and de Moor 1997) for details.

4.2 Indirect equality

The principle of *indirect equality* (Aarts et al. 1992) allows for establishing equality between elements of a partial order by proving that each one is the maximum (or minimum) of the same set. Formally, for a poset $(\mathcal{A}, \sqsubseteq)$ and $a, b \in \mathcal{A}$

$$a = b \quad \Leftrightarrow \quad \langle \forall\, x \in \mathcal{A} :: x \sqsubseteq a \Leftrightarrow x \sqsubseteq b \rangle \tag{32}$$

The anti-symmetry of a partial order is fundamental to establish the equality. For an insightful discussion of the application of this principle to preorders see (Dijkstra 1991).

Indirect equality is the proof technique used in the example provided in Section 2. That simple (non inductive) proof shows the calculational power of Galois connections operated via indirect equality, which are applicable to arbitrarily complex problem domains. Note that we could use the composition of Galois connections to fuse the first two shunting steps (without the need of using the associativity steps in the point-free proof).

References (Aarts et al. 1992; Backhouse and Backhouse 2004; Backhouse 2004) provide an expressive account of such applications, ranging over the predicate calculus, number theory, parametric polymorphism, strictness analysis and so on. For a more elaborated description of proofs about whole division using indirect equality and Galois connection see (Silva and Oliveira 2008a,b).

5. Putting Galois connections and fork algebras together

In this section, all the ingredients will be put together to form a coherent 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.

As presented in Section 3.1, functions and orders are binary relations with certain properties. Moreover, from the enunciated completeness result 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 3.1, simplicity and totality are expressed, respectively, as $f^{\cup} \circ f \subseteq id$ and $id \subseteq f^{\cup} \circ f$.

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.

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.

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 (26) to both sides of (27) yields, for all suitably typed a, 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*

$$f^{\cup} \circ \sqsubseteq_{\mathcal{B}} \quad = \quad \sqsubseteq_{\mathcal{A}} \circ g \tag{33}$$

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.

Indirect equality. The *indirect equality* rule can also be formulated without variables thanks to the PF-transform. Consider two functions $\mathcal{B} \xleftarrow{f} \mathcal{A}$ and $\mathcal{B} \xleftarrow{g} \mathcal{A}$, where $(\mathcal{B}, \preceq)$ is a partial order. That

$$f = g \quad \Leftrightarrow \quad \preceq \circ f = \preceq \circ g \tag{34}$$

(or, equivalently, $f = g \Leftrightarrow f^{\cup} \circ \preceq = g^{\cup} \circ \preceq$) instantiates indirect equality can be easily checked by putting variables back via (26).

However, Eq. (34) 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 makes use of 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 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 $f = g$.

Sections. The absence of variables in point-free representation greatly simplifies the calculus and the implementation of a proof engine. However, as it was introduced in Section 4.1 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 ${}_a f$ 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 ${}_a f_b$.

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 ${}_a f_b$ is a constant, it can only appear where constants can, i.e., as a section of a binary function.

Point-free proofs. Using these principles, let us recall the proof of Section 2 The fundamental step of that proof is the use of the multiplication associative property which is expressed in point-free as :

$$(\times b) \circ (\times c) \quad = \quad \times (c \times b) \tag{35}$$

The reader can verify that this is in fact the case by applying the PF-transform defined in Section 3.3. We deviate from the convention of using subscripts to denote sections of functions since, in this example, it improves readability.

The proof would be performed by *Galculator* as

$$
\begin{array}{ll}
 & \leqslant \circ ((\div c) \circ (\div b)) \\
= & \quad \{ \text{ Associativity of composition (3). } \} \\
 & (\leqslant \circ (\div c)) \circ (\div b) \\
= & \quad \{ \text{ Galois connection (1) } \} \\
 & ((\times c)^{\cup} \circ \leqslant) \circ (\div b) \\
= & \quad \{ \text{ Associativity of composition (3). } \} \\
 & (\times c)^{\cup} \circ (\leqslant \circ (\div b)) \\
= & \quad \{ \text{ Galois connection (1). } \} \\
 & (\times c)^{\cup} \circ ((\times b)^{\cup} \circ \leqslant) \\
= & \quad \{ \text{ Associativity of composition (3). } \} \\
 & ((\times c)^{\cup} \circ (\times b)^{\cup}) \circ \leqslant \\
= & \quad \{ \text{ Contravariance of converse and composition (8). } \} \\
 & ((\times b) \circ (\times c))^{\cup} \circ \leqslant \\
= & \quad \{ \text{ Associativity of multiplication (35). } \} \\
 & (\times (c \times b))^{\cup} \circ \leqslant \\
= & \quad \{ \text{ Galois connection (1). } \} \\
 & \leqslant \circ (\div (c \times b)) \\
\therefore & \quad \{ \text{ Indirect equality (34). } \} \\
 & (\div c) \circ (\div b) = \div (c \times b)
\end{array}
$$

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 2.

6. The *Galois* language

In this section, the *Galois* DSL which underlies the design of the *Galculator* is presented. We only describe the subset of the language related with fork algebras, functions, orders and Galois connections introduced in the previous sections. The complete language has more modules in order to deal with definitions, rewriting and derivations of rules but they lay outside the scope of this paper.

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 its several sub-languages.

6.1 Language design

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. Some trade-offs were needed because it is hard to write symbols like $\top$ or $\bot$ in text; meaningful keywords are used instead. However, we will present just the abstract syntax and not the concrete language.

The complete language comprehends several modules or sub-languages concerning the different aspects of the *Galculator*. Its hierarchical structure is described in Fig. 1 where the shaded boxes correspond to the fragments of the language described in this paper. We start by giving a brief description of the other sub-languages:

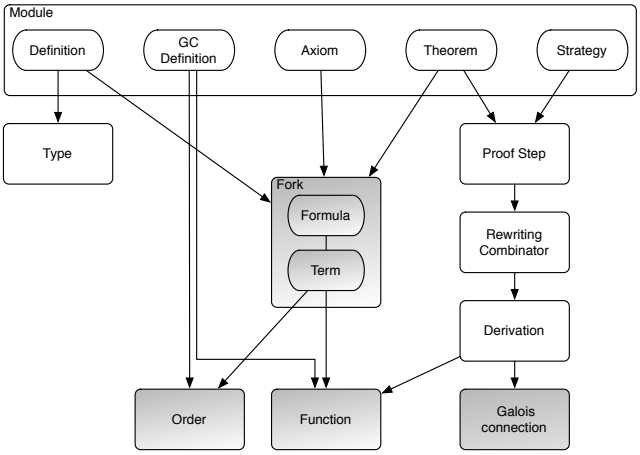


Figure 1. Structure of the sub-languages of the *Galois* DSLs. The arrows represent inclusion between languages; the shaded boxes are the fragments described in this paper.

Module. Allows for organizing each theory in its own module by grouping definitions of operations and Galois connections, axioms, theorems and proof strategies. Currently, modules are just namespaces, not allowing any kind of parametrization. Definitions are added to an environment mapping identifiers to definitions. A theorem must provide its proof which is verified when the module is loaded.

Type. A simple language for type declarations. The type system of *Galculator* is presented by Silva and Oliveira (2008a).

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 2008a) which allows for building more complex strategies from a small set of basic ones. The defined strategies are similar to the ones proposed by Stratego (Luttik and Visser 1997).

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.

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 it 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.
- Γ is an injective total function from variable names to variables. Like Σ we also use Γ in different contexts with different mean-

$$
\begin{array}{rl}
\textbf{Formula} & e ::= \mathit{Equal}\ t\ t \mid \mathit{Less}\ t\ t \\
\textbf{Term} & t ::= \mathit{Ident}\ i \mid \mathit{Var}\ v \mid \mathit{Id} \mid \mathit{Top} \mid \mathit{Bot} \mid \mathit{Pi1} \mid \mathit{Pi2} \\
& \mid \mathit{Neg}\ t \mid \mathit{Conv}\ t \mid \mathit{Meet}\ t\ t \mid \mathit{Join}\ t\ t \mid \mathit{Comp}\ t\ t \\
& \mid \mathit{Fork}\ t\ t \mid \mathit{Prod}\ t\ t \mid \mathit{Ord}\ o \mid \mathit{Fun}\ f \\
\textbf{Function} & f ::= \mathit{FunctionIdent}\ i \mid \mathit{FunctionVar}\ v \mid \mathit{FunctionId} \\
& \mid \mathit{FunctionComp}\ f\ f \\
& \mid \mathit{RightSection}\ f\ s \mid \mathit{LeftSection}\ s\ f \\
\textbf{Section} & s ::= \mathit{Const}\ c \mid \mathit{FunctConst}\ f\ c\ c \\
\textbf{Order} & o ::= \mathit{OrderIdent}\ i \mid \mathit{OrderVar}\ v \mid \mathit{OrderId} \\
& \mid \mathit{OrderConv}\ o \\
\textbf{GC} & g ::= \mathit{GCIdent}\ i \mid \mathit{GCVar}\ v \mid \mathit{GCId} \mid \mathit{GCConv}\ g \\
& \mid \mathit{GCComp}\ g\ g \\
\multicolumn{2}{l}{\textbf{Variable}\ v \quad \textbf{Identifier}\ i \quad \textbf{Constant}\ c}
\end{array}
$$

Figure 2. Abstract syntax of selected fragments of the *Galois* DSL.

ings. Thus, it can map variable names into type, relation, function, orders 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 the same variable is 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 5.

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 into a definition and its respective type, i.e., for an identifier i its definition is given by $\Sigma(i) = \mathit{expr}$ and its type is given by $\Sigma(i) = t$, meaning that $\mathit{expr} : t$ (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. 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. For types t_1 and t_2, we write $t_1 \times t_2$ to denote their product type; $t_1 \sim t_2$ to denote a relation between elements of types t_1 and t_2; $t_1 \leftarrow t_2$ to denote a function between elements of types t_1 and t_2; and $(t_1, \sqsubseteq_{t_1})$ to denote a partial order set t_1.

6.2 Syntax

The abstract syntax of *Galois* is presented in Fig. 2. Orders and functions are embedded in the fork language by Ord and Fun, 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 concrete 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 (2008a).

6.3 Semantics

The selected fragment of the *Galois* DSL has no operational meaning. The semantics of the language is almost straightforward for most of the constructors: they denote the corresponding operation in fork algebra.

We define a semantic function from the abstract syntax to fork algebras formulæ, taking an environment (Σ, Γ, Θ).

$$
\begin{array}{l}
\mathcal{C}^{Form}[\![\mathit{Equal}\ r1\ r2]\!](\Sigma,\Gamma,\Theta) \doteq \\
\qquad \mathcal{C}^{T}[\![r1]\!](\Sigma,\Gamma,\Theta) = \mathcal{C}^{T}[\![r2]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{Form}[\![\mathit{Less}\ r1\ r2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{T}[\![r1]\!](\Sigma,\Gamma,\Theta) \subseteq \mathcal{C}^{T}[\![r2]\!] \\
\mathcal{C}^{T}[\![\mathit{Ident}\ i]\!](\Sigma,\Gamma,\Theta) \doteq \Sigma(i) \quad \text{if} \quad i \in \mathit{dom}(\Sigma) \\
\mathcal{C}^{T}[\![\mathit{Var}\ v]\!](\Sigma,\Gamma,\Theta) \doteq \Gamma(v) \\
\mathcal{C}^{T}[\![\mathit{Id}]\!](\Sigma,\Gamma,\Theta) \doteq \mathit{id} \\
\mathcal{C}^{T}[\![\mathit{Top}]\!](\Sigma,\Gamma,\Theta) \doteq \top \\
\mathcal{C}^{T}[\![\mathit{Bot}]\!](\Sigma,\Gamma,\Theta) \doteq \bot \\
\mathcal{C}^{T}[\![\mathit{Pi1}]\!](\Sigma,\Gamma,\Theta) \doteq \pi_1 \\
\mathcal{C}^{T}[\![\mathit{Pi2}]\!](\Sigma,\Gamma,\Theta) \doteq \pi_2 \\
\mathcal{C}^{T}[\![\mathit{Neg}\ r]\!](\Sigma,\Gamma,\Theta) \doteq \neg\mathcal{C}^{T}[\![r]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{T}[\![\mathit{Conv}\ r]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{T}[\![r]\!](\Sigma,\Gamma,\Theta)^{\cup} \\
\mathcal{C}^{T}[\![\mathit{Meet}\ r1\ r2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{T}[\![r1]\!](\Sigma,\Gamma,\Theta) \cap \mathcal{C}^{T}[\![r2]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{T}[\![\mathit{Join}\ r1\ r2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{T}[\![r1]\!](\Sigma,\Gamma,\Theta) \cup \mathcal{C}^{T}[\![r2]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{T}[\![\mathit{Comp}\ r1\ r2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{T}[\![r1]\!](\Sigma,\Gamma,\Theta) \circ \mathcal{C}^{T}[\![r2]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{T}[\![\mathit{Fork}\ r1\ r2]\!] \doteq \mathcal{C}^{T}[\![r1]\!](\Sigma,\Gamma,\Theta) \nabla \mathcal{C}^{T}[\![r2]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{T}[\![\mathit{Prod}\ r1\ r2]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{T}[\![r1]\!](\Sigma,\Gamma,\Theta) \times \mathcal{C}^{T}[\![r2]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{T}[\![\mathit{Ord}\ o]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{O}[\![o]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{T}[\![\mathit{Fun}\ f]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{F}[\![f]\!](\Sigma,\Gamma,\Theta)
\end{array}
$$

We should notice that *Ord* and *Fun* have no semantics. They are only used for typing purposes.

The function language corresponds to the fragment of relation algebra closed for functions, i.e., when operators are applied to a function the result is still a function. The semantics of sections follows the principles of Section 5.

$$
\begin{array}{l}
\mathcal{C}^{F}[\![\mathit{FunctionIdent}\ i]\!](\Sigma,\Gamma,\Theta) \doteq \Sigma(i) \quad \text{if} \quad i \in \mathit{dom}(\Sigma) \\
\mathcal{C}^{F}[\![\mathit{FunctionVar}\ v]\!](\Sigma,\Gamma,\Theta) \doteq \Gamma(v) \\
\mathcal{C}^{F}[\![\mathit{FunctionId}]\!](\Sigma,\Gamma,\Theta) \doteq \mathit{id} \\
\mathcal{C}^{F}[\![\mathit{FunctionComp}\ f1\ f2]\!](\Sigma,\Gamma,\Theta) \doteq \\
\qquad \mathcal{C}^{F}[\![f1]\!](\Sigma,\Gamma,\Theta) \circ \mathcal{C}^{F}[\![f2]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{F}[\![\mathit{RightSection}\ f\ s]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{F}[\![f]\!](\Sigma,\Gamma,\Theta)_{(\mathcal{C}^{S}[\![s]\!](\Sigma,\Gamma,\Theta))} \\
\mathcal{C}^{F}[\![\mathit{LeftSection}\ s\ f]\!](\Sigma,\Gamma,\Theta) \doteq_{(\mathcal{C}^{S}[\![s]\!](\Sigma,\Gamma,\Theta))} \mathcal{C}^{F}[\![f]\!](\Sigma,\Gamma,\Theta) \\
\mathcal{C}^{S}[\![\mathit{Const}\ c]\!](\Sigma,\Gamma,\Theta) \doteq \Theta(c) \\
\mathcal{C}^{S}[\![\mathit{FunctConst}\ f\ s1\ s2]\!] \doteq_{(\Sigma,\Gamma,\Theta)} \mathcal{C}^{F}[\![f]\!](\Sigma,\Gamma,\Theta)_{(\Theta(s2))}
\end{array}
$$

The semantics of orders is also defined for relation operators closed for partial orders (cf. Section 5).

$$
\begin{array}{l}
\mathcal{C}^{O}[\![\mathit{OrderIdent}\ i]\!](\Sigma,\Gamma,\Theta) \doteq \Sigma(i) \quad \text{if} \quad i \in \mathit{dom}(\Sigma) \\
\mathcal{C}^{O}[\![\mathit{OrderVar}\ v]\!](\Sigma,\Gamma,\Theta) \doteq \Gamma(v) \\
\mathcal{C}^{O}[\![\mathit{OrderId}]\!](\Sigma,\Gamma,\Theta) \doteq \mathit{id} \\
\mathcal{C}^{O}[\![\mathit{OrderConv}\ o]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^{O}[\![o]\!](\Sigma,\Gamma,\Theta)^{\cup}
\end{array}
$$

Finally, the semantic function for Galois connections maps the abstract syntax into the algebraic operations defined in Section 4.1.

$$\mathcal{C}^G[\![\, GCIdent\ i \,]\!](\Sigma,\Gamma,\Theta) \doteq \Sigma(i) \quad \text{if} \quad i \in dom(\Sigma)$$
$$\mathcal{C}^G[\![\, GCVar\ v \,]\!](\Sigma,\Gamma,\Theta) \doteq \Gamma(v)$$
$$\mathcal{C}^G[\![\, GCId \,]\!](\Sigma,\Gamma,\Theta) \doteq id$$
$$\mathcal{C}^G[\![\, GCConv\ g \,]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^G[\![g]\!](\Sigma,\Gamma,\Theta)^{\cup}$$
$$\mathcal{C}^G[\![\, GCComp\ g1\ g2 \,]\!](\Sigma,\Gamma,\Theta) \doteq \mathcal{C}^G[\![g1]\!](\Sigma,\Gamma,\Theta) \circ \mathcal{C}^G[\![g2]\!](\Sigma,\Gamma,\Theta)$$

6.4 Typing

Figure 3 provides the typing rules for the fork algebra fragment of *Galois*. The typing rules for the other fragments of the language are omitted since they are easy to deduce from this example.

Types are important, not only to ensure that expressions are well-formed, but also to automatically discharge certain assumptions. For instance, suppose we have the following rule

$$f \circ \ldots \quad = \quad \ldots \tag{36}$$

which is valid only when f is a function. The corresponding expression in abstract syntax is $Equal\ (Comp\ (Fun\ (FunctionVar\ f))\ (\ldots))\ (\ldots)$. The typing rules ensure that the function variable f has a functional type, i.e., $FunctionVar\ f : t1 \leftarrow t2$. This means that any term t whose type is an instance of $t_1 \leftarrow t_2$ can instantiate the variable f. Otherwise, we would have to verify if t satisfied the simplicity and totality assumptions in order to use (36).

The type system of *Galculator* supports parametric polymorphism, meaning that universally quantified type variables can range through the universe of types. This means that if R is a relation and its type is $\forall t_1, t_2 . R : t_1 \sim t_2$, then it is defined between every possible types allowed by the system. However, if its type is $R : \mathbb{N} \sim \mathbb{N}$, then R defines a relation on natural numbers.

Moreover, *Galculator* provides a type inference mechanism (Silva and Oliveira 2008a) that infers the most general type of an expression. Therefore, the user does not have to explicitly provide the types of all expressions. This makes the type system non-intrusive and easier to work with: the user should only notice it when errors are detected.

7. Implementation notes

A detailed description of the implementation of *Galculator* can be found in (Silva and Oliveira 2008a). Here, we will just briefly explain the general principles concerning the representation of the language using the *Haskell* functional programing language (Peyton Jones 2003). Although we use *Galois* as the example language, the approach can be easily generalized to other typed language.

Representation using ADTs. Traditional approaches to the implementation of languages in *Haskell* resort to algebraic data types (ADTs) to represent abstract syntax. In our example, this means an almost direct transcription of the abstract syntax presented in Fig. 2:

$$\begin{aligned}
&\textbf{data}\ Expr\ = Equal\ Term\ Term\\
&\qquad\quad\ \mid Less\ \ Term\ Term\\
&\textbf{data}\ Term = Id\\
&\qquad\quad\ \mid Top\\
&\qquad\quad\ \mid Conv\ Term\\
&\qquad\quad\ \mid Meet\ \ Term\ Term\\
&\qquad\quad\ \ \ldots
\end{aligned}$$

where $Equal$, $Less$, Id, etc. are *data constructors*, i.e., the only functions capable of building values of the associated data type. Thus, $Equal$ is a function that takes two values of type $Term$ and returns a value of type $Expr$, i.e., $Term \rightarrow Term \rightarrow Expr$. Since constructor Id has no arguments it can be seen as a value of type $Term$. Note that the type of the returned value is always implicit in the declaration.

The above definition closely follows the abstract syntax but it does not include any type information. The type correctness of our language has to be verified using additional functions. Moreover, functions that manipulate the structure can introduce undetected typing errors. For instance, the typing rules for $Meet\ r1\ r2$ require that argument relations $r1$ and $r2$ have the same type. However, if their types are different, the *Haskell* compiler does not complain because it knows nothing about the typing rules of *Galois*: $r1$ and $r2$ are of type $Term$ and all seems correct.

Representation using GADTs. Generalized algebraic data types (GADTs) expand ADTs by allowing the explicit declaration of the signature of each data constructor. In our example, this means the translation not only of the *abstract syntax*, but also of the *typing information* presented in Fig. 3:

$$\begin{aligned}
&\textbf{data}\ R\ r\ \textbf{where}\\
&\quad Equal :: R\ (b \leftrightarrow a) \rightarrow R\ (b \leftrightarrow a) \rightarrow R\ (Expr\ (b \leftrightarrow a))\\
&\quad Less\ \ :: R\ (b \leftrightarrow a) \rightarrow R\ (b \leftrightarrow a) \rightarrow R\ (Expr\ (b \leftrightarrow a))\\
&\quad Id\ \ \ :: R\ (a \leftrightarrow a)\\
&\quad Top\ \ :: R\ (b \leftrightarrow a)\\
&\quad Conv\ :: R\ (b \leftrightarrow a) \rightarrow R\ (a \leftrightarrow b)\\
&\quad Meet\ :: R\ (b \leftrightarrow a) \rightarrow R\ (b \leftrightarrow a) \rightarrow R\ (b \leftrightarrow a)\\
&\quad \ldots
\end{aligned}$$

where $b \leftrightarrow a$ is the type of a relation between types b and a. Without entering into technical details, argument r is an index and not a parameter of data type R, i.e., it reflects the type of the term. For instance, for expression $Equal\ Top\ Top$, r assumes the type $Expr\ (b \leftrightarrow a)$. Since $Conv$ requires an argument whose index is $b \leftrightarrow a$, it is not possible to build the term $Conv\ (Equal\ Top\ Top)$ (because $Expr\ (b \leftrightarrow a)$ does not unify with $b \leftrightarrow a$).

Thus, using GADTs it is no longer possible to define $Meet\ r1\ r2$ when $r1$ and $r2$ have different types because now the *Haskell* compiler can verify *statically* the types of both relations. Moreover, the compiler enforces that rewriting functions do not introduce type errors. This guarantee is quite important in the construction of a proof assistant.

Reflection mechanism. Besides type safe behavior, another advantage of using GADTs in the implementation of the language is the possibility of building a reflection mechanism. This allows for accessing the type of each term during run-time, making it possible to have type-dependent behavior. This mechanism uses a type representation at the term level which is based on the possibility of defining singleton types (basically a bijection between terms and their respective types) using GADTs. Using regular ADTs, the type-safeness of this mechanism would not be possible to ensure statically by the type-checker.

The details about the implementation of the type representation mechanism are outside the scope of this paper but can be found in (Silva and Oliveira 2008a).

Front-end. Initially, *Galois* was an embedded DSL. However, the use of the type representation mechanism described above required the explicit declaration of the type of every expression. The addition of parametric polymorphism to the language increased this problem, since the correct type variables had to be supplied "by-hand". Thus, a front-end with a parser and a type inference mechanism was developed. In this case, the difficulty of building the front-end was greatly increased by the use of GADTs in the rep-

$$\frac{\Sigma,\Gamma,\Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma,\Gamma,\Theta \vdash r_2 : t_1 \sim t_2}{\Sigma,\Gamma,\Theta \vdash \mathit{Equal}\ r_1\ r_2 : t_1 \sim t_2} \qquad \frac{\Sigma,\Gamma,\Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma,\Gamma,\Theta \vdash r_2 : t_1 \sim t_2}{\Sigma,\Gamma \vdash \mathit{Less}\ r_1\ r_2 : t_1 \sim t_2}$$

$$\frac{\Sigma,\Gamma,\Theta \vdash \Sigma(i) : t}{\Sigma,\Gamma,\Theta \vdash \text{Ident}\ i : t} \qquad \frac{\Sigma,\Gamma,\Theta \vdash \Gamma(v) : t_1 \sim t_2}{\Sigma,\Gamma,\Theta \vdash \mathit{Var}\ v : t_1 \sim t_2} \qquad \frac{}{\Sigma,\Gamma,\Theta \vdash \mathit{Id} : t \sim t}$$

$$\frac{}{\Sigma,\Gamma,\Theta \vdash \mathit{Top} : t_1 \sim t_2} \qquad \frac{}{\Sigma,\Gamma,\Theta \vdash \mathit{Bot} : t_1 \sim t_2} \qquad \frac{}{\Sigma,\Gamma,\Theta \vdash \mathit{Pi1} : t_1 \leftarrow t_1 \times t_2} \qquad \frac{}{\Sigma,\Gamma,\Theta \vdash \mathit{Pi2} : t_2 \leftarrow t_1 \times t_2}$$

$$\frac{\Sigma,\Gamma,\Theta \vdash r : t_1 \sim t_2}{\Sigma,\Gamma,\Theta \vdash \mathit{Neg}\ r : t_1 \sim t_2} \qquad \frac{\Sigma,\Gamma,\Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma,\Gamma,\Theta \vdash r_2 : t_1 \sim t_2}{\Sigma,\Gamma,\Theta \vdash \mathit{Meet}\ r_1\ r_2 : t_1 \sim t_2} \qquad \frac{\Sigma,\Gamma,\Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma,\Gamma,\Theta \vdash r_2 : t_1 \sim t_2}{\Sigma,\Gamma,\Theta \vdash \mathit{Join}\ r_1\ r_2 : t_1 \sim t_2}$$

$$\frac{\Sigma,\Gamma,\Theta \vdash r : t_1 \sim t_2}{\Sigma,\Gamma,\Theta \vdash \mathit{Conv}\ r : t_2 \sim t_1} \qquad \frac{\Sigma,\Gamma,\Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma,\Gamma,\Theta \vdash r_2 : t_2 \sim t_3}{\Sigma,\Gamma,\Theta \vdash \mathit{Comp}\ r_1\ r_2 : t_1 \sim t_3} \qquad \frac{\Sigma,\Gamma,\Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma,\Gamma,\Theta \vdash r_2 : t_3 \sim t_2}{\Sigma,\Gamma,\Theta \vdash \mathit{Fork}\ r_1\ r_2 : (t_1 \times t_3) \sim t_2}$$

$$\frac{\Sigma,\Gamma,\Theta \vdash r_1 : t_1 \sim t_2 \qquad \Sigma,\Gamma,\Theta \vdash r_2 : t_3 \sim t_4}{\Sigma,\Gamma,\Theta \vdash \mathit{Prod}\ r_1\ r_2 : (t_1 \times t_3) \sim (t_2 \times t_4)} \qquad \frac{\Sigma,\Gamma,\Theta \vdash o : (t, \sqsubseteq_t)}{\Sigma,\Gamma,\Theta \vdash \mathit{Ord}\ o : t \sim t} \qquad \frac{\Sigma,\Gamma,\Theta \vdash f : t_1 \leftarrow t_2}{\Sigma,\Gamma,\Theta \vdash \mathit{Fun}\ f : t_1 \sim t_2}$$

Figure 3. Typing rules for the fork algebra fragment of the *Galois* DSL.

resentation. The details can also be found in (Silva and Oliveira 2008a).

Since the front-end is independent of the rest of the system, it can be integrated with other tools in the future. For this purpose, the concrete syntax tries to resemble the theoretical notation for fork algebras.

Summary. Algebraic types of functional languages seem naturally suited to implement algebraic DSLs given their syntactical similarity. Beside representing the abstract syntax of an algebraic DSL, GADTs can also encompass typing information. Thus, GADTs ensure that if a term can be built it is syntactically correct and well-typed.

The implementation could be done without resorting to GADTs, using the traditional ADTs. However, the statical type-safeness would be lost and we would have to built our own type verification mechanism.

8. Related work

aRa. *aRa* (Sinz 2000) is an automatic theorem prover for relation algebras. It has a front-end to translate relation algebraic formulæto Gordeev's Reduction Predicate Calculi logic. *aRa* implements a set of simplification rules and reduction strategies for this calculi in order to automatically derive proofs.

Like in our approach, formalization and translation follow Tarski and Givant (1987). However, *aRa* takes an alternative (in fact opposite) direction: relation algebraic formulæare translated to logical sentences and proved using logic, while we conduct our proofs in the algebraic setting. Moreover, the approach of *aRa* not including the fork algebra extension makes it less expressive than the language of *Galculator*. The integration of Galois connections is also absent.

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 formulæ. This is also performed in the *Galculator* system although it is not described in this paper (see (Silva and Oliveira 2008a) for a description of how the internal representation enforces type-safety).

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. However, *RELVIEW* only works with finite cases because relations have to be explicitly defined while *Galculator* can be used with infinite relations because they are defined abstractly.

RELVIEW is the concrete counterpart of the abstract algebraic perspective of *Galculator*. An interesting study would be to see how the two approaches relate and complement each other.

"Off-the-shelf" automated theorem provers. 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.

Since this work only uses relation algebra, it is restricted to a three-variable fragment of first-order logic, unlike our approach which uses fork algebras. Höfner and Struth (2008) mention that some relation operations are adjoints of Galois connections but this fact is not exploited in proofs. Moreover, their approach is not always equational because some proofs require the use of mutual inclusion, since indirect equality is not used.

The application of a similar approach to automatization of proofs in the *Galculator* is still open.

9. Concluding remarks

The approach taken in *Galculator* is innovative: 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 about software correctness.

In this paper, we have shown a DSL for a proof assistant. We started from the theoretical definitions and derived the syntax and semantics of the language as a natural consequence of the associated algebras. In fact, *Galois* is family of DSLs composed in a hierarchical structure that mirrors the relation between the theoretical concepts.

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 se-

mantic level can be operated syntactically in an equational approach, with the help of the introduction of indirect equality. Moreover, Galois connections and indirect equality can be accommodated in fork algebras to form a complete framework for equational point-free calculus. We also showed the usefulness of a strongly typed environment at several levels, even when not using an approach based on the Howard-Curry isomorphism.

Two remarks should be made about the use of Galois connections in *Galculator*. First, although fork algebra operators appear as adjoints in several Galois connections, these cannot be used to describe the properties of those operators. Fork algebras are the primitive theory of *Galculator*, and adjoints of Galois connections are defined as fork algebra terms. Thus, Galois connections cannot describe properties of the theory in which they are themselves defined. These properties can only be derived from the fork algebra axiomatization. Second, *Galculator* takes Galois connections as axioms. The user has the obligation to prove if the specification really defines a Galois connection.

Application. *Galois* is a formal language for mathematical reasoning which, instead of using logic, is based on fork algebras. In fact, as it was discussed, first-order logic and fork algebras are equivalent in terms of expressive and deductive power. However, substitution of equals by equals is the only inference rule needed to reason with fork algebras. Another advantage is that nested quantifications are eliminated, making some expressions simpler and proofs easier, since a single point-free step can encompass several logical steps. Bird and de Moor (1997) show that point-free reasoning based on relations can be very effective in solving several complex problems, although their approach is based in allegories (Freyd and Ščedrov 1990), a close but a bit different relation theory.

What distinguishes *Galculator* from other relational languages is the addition of Galois connections. These bring the ability of handling with situations where equivalent proofs about connected concepts have different difficulties. Moreover, this change of domain can be accommodated with fork algebras and easily expressed syntactically. The extra power introduced by Galois connections only requires the addition of indirect equality as an inference rule.

The *Galculator* can be used with any first-order theory, since these are expressible using fork algebras. However, this tool is not aimed directly at fork algebras: the *Galculator* should be mostly useful when the involved concepts are instances of Galois connections. Among several domains discussed in Sec. 4.1, possible targets of *Galculator* are abstract interpretation and formal concept analysis. For instance, Cousot (1999) shows how to design abstract semantics for abstract interpretations satisfying soundness requirements. This approach caters for correctness of the design by calculus and uses Galois connections as fundamental concept. His style of calculus resembles the one of *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.

The *Galculator* is still a prototype under development. More experiments are needed in order to assess its effectiveness as a proof assistant.

9.1 Future work

Mechanization of point-free transform. In Section 3.3, the equivalence between fork algebra formulæand first-order sentences is described. It would be interesting to automate such point-free transformation, like (Cunha et al. 2005) 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 in (Frias et al. 2004).

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*.

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.

Free-theorems. Exploiting free-theorems with Galois connections has been one of our objectives since the beginning of the *Galculator* project, 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 of free-theorems in *Galculator*. However, since the general theory of free-theorem regards types as relators, some work needs to be done to assess if it is possible to accommodate relators in the fork algebra framework. Otherwise, instead of fork algebras we must use tabular allegories (Bird and de Moor 1997) as the theoretical foundation of *Galculator*.

Evaluation of the language. The *Galois* language is very close to the corresponding mathematical language. Intuitively, its usefulness and usability should mirror the theoretical version. However, our experience with it does not allow us to take definite conclusions, yet. Further experimentation is needed to assess the strengths and weaknesses of the language. Moreover, it should be also compared with other approaches.

Acknowledgments

The authors thank the anonymous referees for insightful comments which improved the quality of the original submission.

This research was supported by FCT (the Portuguese Foundation for Science and Technology), in the context of the MATHIS Project under contract PTDC/EIA/73252/2006. The first author was supported by FCT under grant number SFRH/BD/19195/2004.

References

Chritiene Aarts, Roland Carl Backhouse, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes. Available from www.cs.nott.ac.uk/~rcb/papers, December 1992.

Kevin Backhouse and Roland Carl 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. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.

Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer-Verlag, April 2000.

Roland Carl Backhouse, Roy L. Crole, and Jeremy 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.

John 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.

Ralf Behnke, Rudolf Berghammer, Erich Meyer, and Peter Schneider. Relview - a system for calculating with relations and relational programming. *Fundamental Approaches to Software Engineering*, pages 318–, 1998. URL `http://www.springerlink.com/content/00ue9fk96952bfxx`.

T. Berners-Lee and D. Connolly. Hypertext Markup Language - 2.0. RFC 1866 (Historic), November 1995. URL `http://www.ietf.org/rfc/rfc1866.txt`. Obsoleted by RFC 2854.

R. Bird and O. de Moor. *AlgebraofProgramming*. Series in Computer Science. Prentice-Hall International, 1997. C.A. R. Hoare, series editor.

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.

Patrick Cousot. Abstract interpretation based formal methods and future challenges. In Reinhard Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001. ISBN 3-540-41635-8.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

Alcino Cunha, Jorge Sousa Pinto, and José Proença. A framework for point-free program transformation. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *IFL*, volume 4015 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005. ISBN 3-540-69174-X.

Klaus Denecke, Marcel Erné, and Shelly L. Wismath, editors. *Galois connections and applications*. Springer, 2004.

Edsger W. Dijkstra. Why preorders are beautiful. URL `http://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1102.PDF`. circulated privately, June 1991.

M. Erné, Jürgen Koslowski, A. Melton, and George E. Strecker. A primer on Galois connections. In Aaron 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.

P.J. Freyd and A. Ščedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990.

Marcelo F. Frias, Paulo A. S. Veloso, and Gabriel A. Baum. Fork Algebras: Past, Present and Future. *JoRMiCS*, 1:181–216, 2004.

B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin – Heidelberg, 1999.

Peter Höfner and Georg Struth. On automating the calculus of relations. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2008. ISBN 978-3-540-71069-1. URL `http://dblp.uni-trier.de/db/conf/cade/ijcar2008.html#HofnerS08`.

L. Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley Publishing Company, 5th edition, 1986.

Sebastiaan P. Luttik and Eelco 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.

William McCune. Prover9, June 2009. URL `http://www.cs.unm.edu/~mccune/prover9`.

A Melton, A Schmidt, D, and E Strecker, G. 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.

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.

Oystein Ore. Galois connexions, 1944. Trans. Amer. Math. Soc., 55:493-513.

S. Peyton Jones. Haskell 98: Language and libraries. *J. Funct. Program.*, 13(1):1–255, 2003.

Uta Priss. An FCA interpretation of relation algebra. In Rokia Missaoui and Jürg Schmid, editors, *ICFCA*, volume 3874 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2006. ISBN 3-540-32203-5.

Paulo F. Silva and José 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, 2008a. ACM. ISBN 978-1-60558-117-0. doi: http://doi.acm.org/10.1145/1389449.1389456.

P.F. Silva and J.N. Oliveira. Report on the design of a Galculator. Technical Report FAST:08.01, CCTC Research Centre, University of Minho, 2008b.

Carsten 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.

Alfred Tarski and Steven Givant. *A Formalization of Set Theory without Variables*. American Mathematical Society, 1987. ISBN 0821810413. AMS Colloquium Publications, volume 41, Providence, Rhode Island.

Mark van den Brand, Arie van Deursen, Paul Klint, Steven Klusener, and Emma van der Meulen. Industrial applications of asf+sdf. In Martin Wirsing and Maurice Nivat, editors, *AMAST*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer, 1996. ISBN 3-540-61463-X.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/352029.352035.

David von Oheimb and Thomas F. Gritzner. Rall: Machine-supported proofs for relation algebra. In William McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 1997. ISBN 3-540-63104-6.

Philip 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`.

Shuling Wang, Luís Soares Barbosa, and José Nuno Oliveira. A relational model for confined separation logic. In *TASE*, pages 263–270. IEEE Computer Society, 2008. ISBN 978-0-7695-3249-3.