Report on the Design of a *Galculator*

P.F. Silva and J.N. Oliveira

University of Minho, Braga, Portugal
paufil@di.uminho.pt and jno@di.uminho.pt

Abstract. This report presents the *Galculator*, a tool aimed at deriving equational proofs in arbitrary domains using Galois connections as the fundamental concept. When combined with the *pointfree* transform and the *indirect equality* principle, Galois connections offer a very powerful, generic device to tackle the complexity of proofs in program verification. We show how rewriting rules derived from the properties of the Galois connections are applied in proofs using a strategic term rewriting system which, in the current prototype, is implemented in Haskell. The prospect of integrating the *Galculator* with other proof assistants such as eg. Coq is also discussed.

Keywords: Galois connections, Equational Reasoning, Strategic Term Rewriting, *Haskell*.

1 Introduction

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

Let, however, a/b denote the *whole division* of two natural numbers a and b ($b \neq 0$). Does $(a/b)/c = a/(cb)$ 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 *implicit* definition $c = a/b \equiv \langle \exists\ r\ :\ 0 \leq r < a : a = cb{+}r \rangle$, be it *explicit* definition $a/b = \langle \bigvee\ c\ ::\ cb \leq a \rangle$ based on suprema ($\bigvee$) or be it defined by recursion on the natural numbers (the well-known algorithm based on repeated subtraction), leading to an inductive proof [1].

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

$$c \times b = a \quad \equiv \quad c = ab^{-1} \tag{1}$$

[1] The proof that the remainder is at most the divisor is not so simple to carry out [23] using the implicit definition in the Coq [8] proof assistant (under the *ring* tactics. Still in this alternative, the new problem can be mapped into the old by defining, back to real numbers, a/b to be $(a - (a \ \mathbf{mod}\ b))/b$, and exploiting the properties of the *modulo* operator.

$(b \neq 0)$ 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) can be weakened to inequalities,

$$c \times b \leq a \;\equiv\; c \leq a/b \tag{2}$$

thus obtaining a property valid in the natural numbers, for $b \neq 0$ [2]. With equivalence (2) in mind we can tackle the problem from a different perspective: for every natural number n,

$$\begin{array}{cl}
 & n \leq (a/b)/c \\
\equiv & \qquad \{ \text{ by (2) } \} \\
 & n \times c \leq a/b \\
\equiv & \qquad \{ \text{ (2) again } \} \\
 & (n \times c) \times b \leq a \\
\equiv & \qquad \{ \text{ multiplication is associative } \} \\
 & n \times (c \times b) \leq a \\
\equiv & \qquad \{ \text{ (2) again } \} \\
 & n \leq a/(c \times b)
\end{array}$$

That is, every natural number n at most $(a/b)/c$ is also at most $a/(c \times b)$, and vice versa. We conclude that the two expressions are the same.

A fundamental ingredient of 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 the whole division back into the expression by "shunting" in the opposite direction. But, above all, it is the rule of *indirect equality* [3]

$$a = b \;\equiv\; \langle \forall\, x \;::\; x \leq a \equiv x \leq b \rangle \tag{3}$$

which implicitly shapes the whole strategy of the proof [4].

[2] Note how this property matches with the *explicit* definition given earlier on: fixing a and b and reading (2) as an implication from left to right, this already tells us that a/b is the largest c such that $c \times b \leq a$ holds.

[3] We use notation $\langle \exists\, x \;:\; R : \; T \rangle$ meaning *there exists some x in the range R such that T holds.*

[4] 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. $A = B \;\equiv\; \langle \forall\, x \;::\; x \in A \equiv x \in B \rangle$ as opposed to, eg. $A = B \;\equiv\; A \subseteq B \;\wedge\; B \subseteq A$.

The structure of the calculational proof above was not accidental: equation (2) is an instance of an ubiquitous concept in mathematics, that of a Galois connection [20]. Galois connections relate pairs of functions between pre-ordered domains providing "shunting" laws between them. Functions which are inverses of each other form a special case of a Galois connection where both orders are the equality relation, as in (1). 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. Moreover, Galois connections form an algebra and can be combined to form arbitrarily complex connections.

Likewise, the *indirect equality* principle is also more general: two elements a and b in a partial order $(\mathcal{A}, \sqsubseteq)$ are the same if and only if they are related with the same objects:

$$\begin{aligned} a = b \quad &\equiv \quad \langle \forall\ x\ ::\ x \sqsubseteq a \equiv x \sqsubseteq b \rangle \\ &\equiv \quad \langle \forall\ x\ ::\ a \sqsubseteq x \equiv b \sqsubseteq x \rangle \end{aligned}$$

Clearly, these ingredients can be put together in order to solve more complex problems. Let, for instance, multiplication and whole division in (2) 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

$$X \cdot R \subseteq Y \equiv X \subseteq Y \,/\, R \tag{4}$$

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

$$b(R \cdot S)c \equiv \langle \exists\ a\ ::\ b\ R\ a\ \wedge\ a\ S\ c \rangle \tag{5}$$

and division:

$$c(S \,/\, R)a \equiv \langle \forall\ b\ :\ a\ R\ b :\ c\ S\ b \rangle \tag{6}$$

(References [1, 3] 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

$$(S \,/\, R) \,/\, U = S \,/\, (U \cdot R)$$

would be made along the very same steps as in inferring $(a/b)/c = a/(cb)$ 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

$$\langle \forall\ j\ :\ aUj :\ \langle \forall\ k\ :\ jRk :\ bSk \rangle \rangle \ \equiv\ \langle \forall\ k\ :\ \langle \exists\ j\ :\ aUj :\ jRk \rangle :\ bSk \rangle$$

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

This capability of dealing with identical structures despite their complexity makes Galois connections a very powerful 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 [5] quite simple.

The appreciation of all these advantages has led the authors of the current report to embark on a project whose main aim is the design and implementation of a rewriting system — the *'G'alculator* — solely based on Galois connections, their algebra and the associated tactics (such as indirect equality). The idea is to evaluate how far one can go in mechanical proofs solely relying on the Galois connection concept.

The purpose of this document is to report work on a prototype of this tool which uses strategic term rewriting [28] implemented in the Haskell functional programming language [15]. The rest of the report is organized as follows: Section 2 introduces Galois connections in more detail. Section 4 describes the architecture of the *Galculator*, including the use of the *pointfree* transform in the rewriting phase (Section 3) and a description of the strategies implemented (Section 4.2). Section 5 gives an account of related work. Finally, Section 6 concludes and suggests paths for future work.

2 Galois connections

This section presents an overview of Galois connections and their algebraic properties. Given two pre-ordered 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 $b \in \mathcal{B}$:

$$f\ a \sqsubseteq_{\mathcal{B}} b \quad \equiv \quad a \sqsubseteq_{\mathcal{A}} g\ b \tag{7}$$

Function f (resp. g) is referred to as the *lower adjoint* (resp. *upper adjoint*) of the connection. In this report we will display Galois connections using the graphical notation

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

which we in-line in text by writing arrow $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}) \xleftarrow{(f,g)} (\mathcal{B}, \sqsubseteq_{\mathcal{B}})$. 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. The right to left arrow is visually more consistent with function composition which will be heavily used through the rest of this report.

[5] The best known application of Galois connections in computer science is perhaps that of *abstract interpretation* [9, 22]. References [1–3] provide a far more expressive account of such applications, ranging over the predicate calculus, number theory, parametric polymorphism, etc.

Galois connections have several important properties, relating them to the underlying ordered structure, of which Table 1 gives a summary. (See [3] for a full account.) The main advantage of this rich theory is that once a concept is identified as adjoint of a Galois connection, all generic properties are inherited, even when the other adjoint is not known. For instance, every adjoint is monotonic; upper adjoints preserve top elements while lower adjoints preserve bottom-elements, and so on.

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. As we shall see, this is in fact central to the *Galculator* which provides several combinators of the Galois connections algebra. Let us see some of these combinators.

The simplest of all Galois connections is the identity, $(\mathcal{A}, \sqsubseteq) \xleftarrow{(id,id)} (\mathcal{A}, \sqsubseteq)$, where id is the (polymorphic) identity function such that $id\ x = x$, for all x. Two Galois connections $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ and $(\mathcal{B}, \preceq) \xleftarrow{(h,k)} (\mathcal{C}, \leq)$ with matching preorders can be composed, forming Galois connection

$$\begin{tikzcd}
\mathcal{A} \arrow[loop left, "\sqsubseteq"] \arrow[r, bend left, "h \cdot f"] & \mathcal{C} \arrow[loop right, "\leq"] \arrow[l, bend left, "g \cdot k"]
\end{tikzcd}$$

(Note the reverse composition order in which adjoints compose.) Composition is an associative operation and the identity Galois connection is its unit, thus forming a monoid structure.

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

Moreover, every relator $\mathcal{F}$ [6] that distributes through binary intersections preserves Galois connections [2]. Therefore, from $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ one infers, for every such relator, $(\mathcal{F}\mathcal{A}, \mathcal{F} \sqsubseteq) \xleftarrow{(\mathcal{F}f,\mathcal{F}g)} (\mathcal{F}\mathcal{B}, \mathcal{F} \preceq)$. This extends to binary relators such as, for instance, the *product* $A \times B$ which pairs elements of A with elements of B ordered by the pairwise orderings. Products also enable one to express more complex Galois connections such as eg. $(\mathcal{A}, \sqsubseteq) \xleftarrow{(\delta,\sqcap)} (\mathcal{A} \times \mathcal{A}, \sqsubseteq \times \sqsubseteq)$ (where δ is the diagonal function $\delta a = (a, a)$) which captures the definition of greatest lower bounds in partial orders:

$$a \sqsubseteq b \ \wedge \ a \sqsubseteq c \quad \equiv \quad a \sqsubseteq b \sqcap c$$

Summing up, it is now possible to explain, back to Section 1, our introductory example in terms of the algebra of connections sketched above: the example is

[6] Relators are the relational counterpart of functors. See eg. [1, 5] for details.

nothing but the composition of the two connections $(\mathbb{N}, \leq) \overset{((\times b),(/b))}{\longleftarrow} (\mathbb{N}, \leq)$ and $(\mathbb{N}, \leq) \overset{((\times c),(/c))}{\longleftarrow} (\mathbb{N}, \leq)$, where $\mathbb{N}$ denotes the set of natural numbers.

Notations $(\times b)$, $(/b)$, etc. call for an explanation: since the operations in equation (2) are binary, 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* [7] $(a\theta)$ 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. Thus, instead of having just one Galois connection, we build a family of Galois connections indexed by the frozen argument.

In this context, should $(\mathbb{Z}, \leq) \overset{((\times b),(/b))}{\longleftarrow} (\mathbb{Z}, \leq)$ be the extension of connection (2) to integers $\mathbb{Z}$ (with $b > 0$) and $(\mathbb{Z}, \leq) \overset{((+b),(-b))}{\longleftarrow} (\mathbb{Z}, \leq)$ the Galois connection

$$c + b \leq a \;\equiv\; c \leq a - b$$

Then, the outcome of the same calculation using the composition of the above Galois connections would wield equality

$$a/b - c \quad = \quad (a - cb)/b$$

Well, *not exactly* the same: distributivity of multiplication over addition would have been the property assumed about the lower adjoints (the "easy" ones), instead of associativity of multiplication [8].

This use of algebraic properties associated with the binary operators whose sections participate in Galois connections leads us into the explanation of how the *Galculator* actually works, which is the topic addressed in the next section.

3 Pointfree transform

As is well-known when transforming expressions, one must be very careful about variables: free and bound variables make substitutions tricky. However, often this complexity is not really needed. Many variables are not useful, they are just placeholders for connecting parts of the expressions. By removing these variables and introducing another connection mechanism, the calculus can be simplified.

A solution for this problem is the *pointfree transform* (PF-transform) [4, 26, 19]. By applying this technique, variables are abstracted from expressions, and composition (5) becomes the overall *glue* among terms. Once this transform is applied, variables are only needed in functional sections of shape $(a\theta)$ and

[7] This terminology is taken from functional programming, where sections are a very popular programming device [21].

[8] It should be stressed that two families of Galois connections are in fact used: the one given and, for the case $b < 0$, $(\mathbb{Z}, \leq) \overset{((\times b),(/b))}{\longleftarrow} (\mathbb{Z}, \geq)$. The proofs are, anyway, fully analogous.

Property	Description
$f\ a \sqsubseteq_B b \equiv a \sqsubseteq_A g\ b$	"Shunting rule"
$g\ (b \sqcap_B b') = g\ b \sqcap_A g\ b'$	Distributivity (UA over *meet*)
$f\ (a \sqcup_A a') = f\ a \sqcup_B f\ a'$	Distributivity (LA over *join*)
$a \sqsubseteq_A g\ (f\ a)$	Lower cancellation
$f\ (g\ b) \sqsubseteq_B b$	Upper cancellation
$a \sqsubseteq_A a' \Rightarrow f\ a \sqsubseteq_B f\ a'$	Monotonicity (LA)
$b \sqsubseteq_B b' \Rightarrow g\ b \sqsubseteq_A g\ b'$	Monotonicity (UA)
$g\ \top_B = \top_A$	Top-preservation (UA)
$f\ \bot_A = \bot_B$	Bottom-preservation (LA)

Table 1. Properties of Galois connections. Legend: UA — upper adjoint. LA — lower adjoint. Properties involving meet, join, top and bottom assume preorders $\sqsubseteq_A$ and $\sqsubseteq_B$ form lattice structures.

(θb), recall Section 2. One particular rule of the PF-transform which is specially helpful in removing variables from expressions is

$$(f\ b)\ R\ (g\ a) \equiv b(f^\circ \cdot R \cdot g)a \tag{8}$$

where f° denotes the converse of f. (In general, the converse of relation R, denoted R°, is such that $a(R^\circ)b$ holds iff bRa holds.)

It is easy to see that the application of (8) to both sides of (7) yields

$$a(f^\circ \cdot \sqsubseteq_{\mathcal{B}} \cdot id)b \quad \equiv \quad a(id \cdot \sqsubseteq_{\mathcal{A}} \cdot g)b$$

which leads to the relational *equality*

$$f^\circ \cdot \sqsubseteq_{\mathcal{B}} \quad = \quad \sqsubseteq_{\mathcal{A}} \cdot g$$

once variables are removed (and also because *id* is the unit of composition). So we can deal with expressions involving adjoints of Galois connections by equating terms without variables.

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 $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$ are partial orders. That

$$f = g \quad \equiv \quad \preceq \cdot f = \preceq \cdot g \tag{9}$$

(or, equivalently, $f = g \equiv f^\circ \cdot \sqsubseteq = g^\circ \cdot \sqsubseteq$) instantiates indirect equality can be easily checked by putting variables back via (8).

Switching to PF-terms makes the operation of the *Galculator* a lot easier. Let us then see how the calculation which motivated our introduction is actually performed inside the *Galculator*: first of all, equation (2) becomes a family of equalities

$$(\times b)^\circ \cdot \leq = \leq \cdot (/b) \tag{10}$$

indexed by b (assuming $b \neq 0$), where $(\times b)$ and $(/b)$ are the right section functions of multiplication and division, respectively. Then the following series of *equalities* are calculated:

$$
\begin{array}{ll}
 & \leq \cdot ((/c) \cdot (/b)) \\
= & \qquad \{ \text{ composition is associative } \} \\
 & (\leq \cdot (/c)) \cdot (/b) \\
= & \qquad \{ \text{ substitution of equals for equals (10) } \} \\
 & ((\times c)^{\circ} \cdot \leq) \cdot (/b) \\
= & \qquad \{ \text{ associativity again } \} \\
 & (\times c)^{\circ} \cdot (\leq \cdot (/b)) \\
= & \qquad \{ \text{ (10) again } \} \\
 & (\times c)^{\circ} \cdot ((\times b)^{\circ} \cdot \leq) \\
= & \qquad \{ \text{ associativity again } \} \\
 & ((\times c)^{\circ} \cdot (\times b)^{\circ}) \cdot \leq \\
= & \qquad \{ \text{ converse of composition: } (R \cdot S)^{\circ} = S^{\circ} \cdot R^{\circ} \ \} \\
 & ((\times b) \cdot (\times c))^{\circ} \cdot \leq \\
= & \qquad \{ \text{ multiplication is associative: } (a \times c) \times b = a \times (c \times b) \ \} \\
 & (\times (b \times c))^{\circ} \cdot \leq \\
= & \qquad \{ \text{ (10) again } \} \\
 & \leq \cdot (/(b \times c)) \\
:: & \qquad \{ \text{ indirect equality (9) } \} \\
 & (/c) \cdot (/b) = (/(c \times b))
\end{array}
$$

It is true that the pointfree notation makes expressions more cryptic. Although proofs are harder to understand, inside the *Galculator* they are much easier to rewrite, via simple equational rules and substitutions of equals by equals.

4 *Galculator*

We start by overviewing the architecture of the current prototype of the *Galculator*. Then we give an overview of the term rewriting system. For a more detailed description of the implementation see [24].

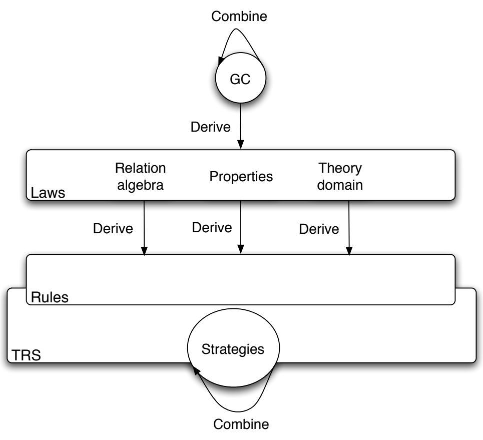

Fig. 1. Architectural components of the *Galculator* prototype

4.1 Architecture

The current prototype of the *Galculator* is implemented in *Haskell*. This language was chosen because of its representation capabilities due to a powerful data type system and its ability to perform symbolic manipulation.

Following a modular approach, the *Galculator* tool is divided into several components (see also Figure 1):

Interpreter. The command line interpreter provides for interactive user interfacing. Several options are offered: loading modules, exploiting Galois connections' algebra, checking expressions and doing proofs. Currently, proofs are interactive, with the system providing a list of applicable rules; the user can choose which one to use in each step. At the end, a complete proof log, with all equational steps and justifications can be obtained.

Parser. A domain specific language (DSL) was defined in order to express the concepts we use: Galois connections, relations, orders, functions and so on. The parser recognizes the user-defined input of the interpreter. Currently, the syntax is not much user friendly since it reflects closely the internal representation. The choice of a more definitive interface syntax has been deferred to a later phase in which the *Galculator* will interface with other theorem provers, namely Coq [8]. (See Section 6.)

Type inference. The *Galculator* is a typed environment. Besides being implemented in a typed language, it has its own type system in order to check for the correctness of expressions. Errors in expressions and proofs can be detected by the use of types. Altogether, the user is released from having to provide explicit types in expressions.

The *Galculator* type system supports parametric polymorphism. A unification mechanism on type variables has been implemented using the Hindley-Milner algorithm [12]. Polymorphism is useful for deriving the so-called *free-theorems* of functions [31, 2], a kind of commutative property enjoyed by polymorphic functions directly inferred from their types.

Term rewriting system. The core of *Galculator* is its term rewriting system (TRS) whose rewrite rules (derived from the theory explained in Section 2) are applied to terms in order to build proofs. The developed system uses the flexibility of strategies [15] and their combinatorial properties in order to build complex proof strategies. Moreover, since the whole system is typed, the TRS is also typed, making it possible to apply rewrite rules to expressions of a determined type. In fact, two TRS have been built: one deals with the expression representation, the other works on the internal type representation. More details are provided in Section 4.2.

Modules. The *Galculator* allows for grouping different theories into modules that can be loaded by the user. Each theory has its own concepts, partial orders, Galois connections and additional algebraic properties. Modules reduce the number of rules that have to be scanned in order to find the right one.

Property inference. Galois connections are specified by their type (sets on which they are defined), the pre-orders involved and the adjoint functions. This component derives the properties stated in Table 1 from the definition. The result are equational expressions recorded in the internal representation notation.

Rule inference. The equational laws expressed by our representation are purely declarative; they cannot be used in rewriting because they are not functions. Thus, we developed a rule inference engined developed which takes an equational expression and returns a rewrite function applicable by the TRS. This component ensures most of the genericity of *Galculator*.

GADTs. Symbolic systems implemented in functional languages use representations based on *algebraic data types* (ADTs). These allow for building typed representation of terms in a combinatorial manner. Moreover, ADTs have a close link with context-free grammars since their relationship with the corresponding abstract syntax tree is straightforward.

In the *Galculator* we have used *generalized* algebraic data types (GADTs) [13]. GADTs extend the notion of ADT by allowing restrictions of the domain of data constructors to certain types as well as the restriction of the result type. Thus the representation can contain more accurate restrictions on the valid expressions. This means that GADTs can represent the abstract syntax of languages plus some type restrictions, usually in the scope of semantical verification. Moreover, the technique used to encode type representation (GADTs are used as singleton types) makes it possible to define a type-safe equality between types.

Strategy combinator	**Symbol**
Always failing rule	$\perp$
Identity rule	nop
Sequential composition	$\triangleright$
Choice (non-deterministic)	$\oplus$
Choice (Left-bias)	$\oslash$
Map on all children	all
Map on one child	one

Strategy	**Definition**
try	$try(s) \Leftrightarrow s \oslash nop$
once	$once(s) \Leftrightarrow s \oslash one(once(s))$
many	$many(s) \Leftrightarrow (s \triangleright (many(s))) \oslash nop$
top-down	$topdown(\mathrm{s}) \Leftrightarrow \mathrm{s} \triangleright all(topdown(\mathrm{s}))$
bottom-up	$bottomup(\mathrm{s}) \Leftrightarrow all(bottomup(\mathrm{s})) \triangleright \mathrm{s}$
innermost	$innermost(\mathrm{s}) \Leftrightarrow all(innermost(\mathrm{s})) \triangleright (try(\mathrm{s} \triangleright innermost(\mathrm{s})))$

Table 2. Strategic combinators implemented in the *Galculator* TRS.

4.2 Details on the *Galculator* rewriting system

The term rewriting system (TRS) is the core of the *Galculator*. As already mentioned, it is based on strategic techniques.

Strategic term rewriting uses simple basic strategies and combinators in order to build arbitrarily complex strategies. This resembles the traditional pattern in functional programming of using combinators to solve problems in a modular way. Indeed, the implementation of rewriting strategies in the functional language *Haskell* is simple and elegant. This is because the rewriting system is specified in a declarative style, such as the equations themselves, making the rewriting strategies easily programmable. Moreover, equations and strategies can be reused and combined in different ways in order to obtain different TRS. Thus, strategic TRS allow for good separation of concepts, re-usability of definitions and great flexibility.

The possibility of allowing for non-confluent and non-terminating sets of equations is another advantage of using strategies. In the case of program calculi, equations can be used in any direction, thus resulting in a non-terminating system. In the case of the *Galculator*, most of the equations are directly derived from the Galois connections defined by the user, so the overall termination property is hard to ensure.

Stratego [29] and the *Rewriting Calculus* [7] are among the first strategic rewriting frameworks implemented. The first frameworks combining strategic programming and strong typing were *Strafunski* [15] in the functional paradigm and the *JJTraveler framework* [30, 14] in the object-oriented paradigm.

In order to make the implementation of the strategies independent of the representation, in the *Galculator* we use a *spine* representation [13]. The *spine* defines how to represent constructors with or without children. Additionally, two conversion functions are defined: from and to the spine representation. The strategies that make the traversal through the children of a node (*all*, *one*) use the spine representation in their definitions; the conversion functions allow the application to actual terms. This greatly reduces the amount of *boilerplate* code in the implementation of traversals. Moreover, although the representation has changed several times, the changes were restricted to the function that converts terms in their *spine* representation.

The basic strategy combinators implemented in the *Galculator* are summarized in Table 2. More elaborate strategies have been built using the basic ones (Table 2).

Special mention should go to the fact that two different choice operators have been defined: one for *left-bias* choice and another for *non-deterministic* choice. The left-biased choice combinator ($\oslash$) always returns the result of applying its left strategy if this succeeds; only in the case this fails is the right one tried. For instance, whenever used in combination with the strategic sequential composition in $(s \oslash r) \triangleright t$, if the application of s succeeds then t is tried; if t fails to apply then the overall strategy also fails; r is only tried in the case of failure of s.

The non-deterministic choice combinator ($\oplus$) allows for a backtracking behavior: any of the argument strategies can be chosen. Therefore, in $(s \oplus r) \triangleright t$ all the possible applications are tried. The use of non-deterministic choice is important when different paths should be tried but it comes with a performance penalty since the search space expands.

5 Related work

Galois connections in Coq. Reference [22] presents a representation of Galois connections in Coq [8] developed in the context of work on *abstract interpretation.* Adjoints are defined over complete lattices (a stricter requirement than in the general theory) and the fact that they form a connection has to be proved. Moreover, proofs of the general properties that Galois connections enjoy are defined in order to be executed in Coq. However, Galois connection algebra is not exploited in order to combine existing connections nor is it applied in proofs.

This work in a sense complements the *Galculator* approach since it can fulfill proof obligations about adjoints left to the user of our system.

2LT. The core of the *Galculator* is inspired on the *2LT* system [11]. 2LT 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 [10] and invariant preservation through data refinement.

Our representation technique and the rewriting strategies implemented were mostly influenced by this system, although the rewriting rules of *2LT* are defined using functions and therefore hard-wired into the system. Although *2LT* also uses a type representation, it does not support polymorphism. Moreover, *2LT* is not a prover: it calculates data and functional transformations using a correct-by-construction philosophy. Although 2LT does not rely on Galois connections explicitly, its underlying theory does so [18].

PF-ESC. This tool, which performs *pointfree extended static checking* [17] and is also inspired in *2LT*, uses the relation calculus to simplify PF-transformed proof obligations. Galois connections are used implicitly in the underlying calculus. Although it 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. The advantage of using properties is that the system is more flexible in so far as allowing for new kinds of relation. Moreover, no type-lifts are needed like in our approach. 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. Because 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. The authors of [6] 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 [25]. The system helps the user by detecting errors in proofs and suggesting valid deductive steps. Unlike our approach, this system does not provide type support and does not use Galois connections as a building block of the calculus implemented.

6 Concluding remarks

Despite being in its infancy, the *Galculator* already shows how Galois connection algebra, indirect equality and pointfree representation can be used together in an effective proof assistant. The pointfree representation can be regarded as an extension to relations of the combinatory logic approach to functional notation [27]. Strategic term rewriting provides the other ingredient of the approach, thanks to the support of GADTs.

We decided to develop our own rewriting system instead of using another rewriting engine, e.g. Stratego [28] or Maude [16],mainly because the typed behavior of the system would be lost. more effort than needed to implement the whole rewriting system. Since it is relies on Haskell monads its implementation is quite simple and extensible. For instance, adding state information may be in order to implement some of the future features.

The tool is still in the prototype stage, thus many ideas are still left to be explored and many improvements are likely to be made as more experimentation takes place. Some directions of the future work are as follows:

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.

User defined strategies. The user can apply the rewriting rules using some of the pre-defined global strategies. However, more flexibility can be achieved by allowing the user to define his/her own strategies by combining the existing ones. It would be interesting to study what kind of strategies can be used to solve certain problems and classify them accordingly.

Free-theorems. Exploiting free-theorems with Galois connections has been one of our objectives since the beginning of the *Galculator* project, specially because from [2] we know how to calculate free-theorems about Galois connections based on their types. Currently, some work has been done in this field but it is not fully satisfactory because the approach is not sufficiently generic, in particular concerning *relator* typed representation.

Integration with 'host' theorem provers. *Galculator* is not a general theorem prover: it works only with well-defined situations involving adjoints of Galois connection. Used together with other theorem provers it can behave like a specialized *add-on* component able to discharge proofs wherever terms involve adjoints of known GC s. Currently, we are working on integrating the *Galculator* with Coq [23]. The prospect of its integration with other proof assistants is also open.

Acknowledgments The authors thank José Bacelar Almeida for his experiments in the Coq [8] proof assistant related with the *Galculator*. Joost Visser has been a permanent source of inspiration. His help in bringing the *Galculator* to life is gratefully acknowledged.

The first author acknowledges support from the Portuguese *Science and Technology Foundation* under grant number `SFRH/BD/19195/2004`.

References

1. 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.
2. 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.
3. R.C. Backhouse. *Mathematics of Program Construction.* Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.
4. 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.

5. R. Bird and O. de Moor. Algebra of Programming. Series in Computer Science. Prentice-Hall International, 1997. C.A. R. Hoare, series editor.
6. J. Bohorquez 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, 7-9 July 2005.
7. H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In Furio 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.
8. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 73(2/3), 1988.
9. 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.
10. A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. *ENTCS*, 174(1):17–34, 2007. Proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006).
11. Alcino Cunha, José Nuno Oliveira, and Joost Visser. Type-safe two-level data transformation. In Jayadev Misra, Tobias Nipkow, and Emil 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, 2006.
12. J.R. Hindley. Basic Simple Type Theory. Number 42 in Cambridge Tracks in Theoretical Computer Science. Cambridge University Press, 1st edition, 1997.
13. 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.
14. 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.
15. 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.
16. P. Lincoln M. Clavel, S. Eker and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.
17. C.M. Necco, J.N. Oliveira, and J. Visser. Extended static checking by strategic rewriting of pointfree relational expressions. Draft of February 3, 2007.
18. J.N. Oliveira. *Transforming Data by Calculation.* In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE 2007 Proceedings*, pages 139–198, July 2007.
19. J.N. Oliveira and C.J. Rodrigues. Pointfree factorization of operation refinement. In *FM'06*, volume 4085 of *LNCS*, pages 236–251. Springer-Verlag, 2006.
20. Oystein Ore. Galois connexions, 1944. Trans. Amer. Math. Soc., 55:493-513.
21. S. Peyton Jones. Haskell 98: Language and libraries. *J. Funct. Program.*, 13(1):1–255, 2003.
22. David Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés.* PhD thesis, Universit'e de Rennes, December 2005.
23. P.F. Silva, J.C. Almeida, and J.N. Oliveira. Galculator meets Coq. Technical report, CCTC, University of Minho, 2008. (In preparation).

24. P.F. Silva and J. N. Oliveira. 'Galculator': Functional prototype of a Galois-connection based proof assistant. (submitted), April 2008.
25. J.M. Spivey. *The Z Notation — A Reference Manual.* Series in Computer Science. Prentice-Hall International, 1989. C.A. R. Hoare.
26. Alfred Tarski and Steven Givant. *A Formalization of Set Theory without Variables.* American Mathematical Society, 1987. AMS Colloquium Publications, volume 41, Providence, Rhode Island.
27. D. Turner. A new implementation technique for applicative languages. *Software-Practice & Experience*, 9:31–49, 1979.
28. 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 2001.
29. 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.
30. J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, 2001. Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications.
31. 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.