‘Galculator’

Functional Prototype of a Galois-connection Based Proof Assistant

Paulo F. Silva José N. Oliveira
CCTC, University of Minho, Braga, Portugal
{paufil,jno}@di.uminho.pt

Abstract

Galculator is the name of the prototype of a proof assistant of a special brand: it is solely based on the algebra of Galois connections. When combined with the *pointfree* transform and tactics such as the *indirect equality* principle, Galois connections offer a very powerful, generic device to tackle the complexity of proofs in program verification. The paper describes the architecture of the current *Galculator* prototype, which is implemented in Haskell in order to steer types as much as possible. The prospect of integrating the *Galculator* with other proof assistants such as e.g. Coq is also discussed.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Verification, Theory, Languages, Design

Keywords Galois connections, Point-free notation, Haskell, GADT, DSL, Proof Assistant

1. Introduction

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

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

It is often the case that practical application of tools is hindered by the underlying theory itself, whenever this is too “heavy” for the problem at hands. Let us consider a simple example: we want to prove the correctness of the following Haskell function[1]

$$x \text{ `}div\text{`} \ y \mid x < y = 0 \\ \qquad\qquad \mid x \geqslant y = (x - y) \text{ `}div\text{`} \ y + 1$$

which computes whole division, for non-negative x and positive y. A standard proof would involve some kind of induction (e.g. structural induction, fixpoint induction). However, what we have above is *code* — where is the *specification*?

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

$$c = x \div y \ \Leftrightarrow \ \langle \exists\, r \ : \ 0 \leqslant r < y : \ x = c \times y + r \rangle \qquad (1)$$

and the other *explicit*

$$x \div y = \langle \bigvee \ z \ :: \ z \times y \leqslant x \rangle \qquad (2)$$

where notation $\bigvee$ means the largest of a set of numbers.

Checking the correctness of the given Haskell code against implicit definition (1) in the Coq proof assistant (Coquand and Huet 1988) entails a number of steps which are described in (Bertot and Castéran 2004; Almeida 2008). Still for the same purpose, one might dive into real number arithmetics by defining $x \div y$ to be $(x - (x \ \mathbf{mod}\ y))/y$, and exploiting the properties of the *modulo* operator.

Correctness verification assumes that both specification and implementation are available before proofs take place. A different, more constructive alternative would be to *calculate* the implementation from the specification itself. In the current example, it can be observed that the following Galois connection (Ore 1944) arises from (2),

$$z \times y \leqslant x \ \Leftrightarrow \ z \leqslant x \div y \qquad (y > 0) \qquad (3)$$

assuming x, y, z universally quantified over natural numbers. Note how this property matches with (2): fixing x and y and reading (3) as an implication from left to right, this already tells us that $x \div y$ is the largest z such that $z \times y \leqslant x$ holds.

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

$$a - b \leqslant c \quad \Leftrightarrow \quad a \leqslant c + b \qquad (4)$$

which explains subtraction over the integers (another operator used in the algorithm). We can put these two connections together by restricting (3) to non-negative integers, keeping $y \neq 0$. We reason:

[1] Throughout this text, we use `lhs2TeX` (Hinze and Löh 2008) for typesetting symbols and code in Haskell.

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

PPDP'08, July 15–17, 2008, Valencia, Spain.

$$
\begin{array}{ll}
& z \leqslant x \div y \\
\Leftrightarrow & \quad \{ \text{ Galois connection (3) assuming } x \geqslant 0, y > 0 \} \\
& z \times y \leqslant x \\
\Leftrightarrow & \quad \{ \text{ cancellation, thanks to (4) } \} \\
& z \times y - y \leqslant x - y \\
\Leftrightarrow & \quad \{ \text{ distribution law } \} \\
& (z-1) \times y \leqslant x - y \\
\Leftrightarrow & \quad \{ \text{ (3) again, assuming } x \geqslant y \} \\
& z - 1 \leqslant (x-y) \div y \\
\Leftrightarrow & \quad \{ \text{ (4) again } \} \\
& z \leqslant (x-y) \div y + 1
\end{array}
$$

That is, every natural number z which is at most $x \div y$ (for $x \geqslant y$) is also at most $(x-y) \div y + 1$ and vice versa. We conclude that the two expressions are the same

$$x \div y = (x-y) \div y + 1 \tag{5}$$

thus calculating the second clause of the *div* function. Concerning the first, we assume $x < y$ and reason in the same style:

$$
\begin{array}{ll}
& z \leqslant x \div y \\
\Leftrightarrow & \quad \{ \text{ (3) and transitivity, since } x < y \ \} \\
& z \times y \leqslant x \ \wedge \ z \times y < y \\
\Leftrightarrow & \quad \{ \text{ since } y \neq 0 \ \} \\
& z \times y \leqslant x \ \wedge \ z \leqslant 0 \\
\Leftrightarrow & \quad \{ \ z \leqslant 0 \text{ entails } z \times y \leqslant x, \text{ since } 0 \leqslant x \ \} \\
& z \leqslant 0
\end{array}
$$

This time we get $x \div y = 0$ under the same principle which supported clause (5), which is known as the principle of *indirect equality* (Aarts et al. 1992):

$$a = b \quad \Leftrightarrow \quad \langle \forall\, x \ ::\ x \leqslant a \Leftrightarrow x \leqslant b \rangle \tag{6}$$

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

This simple (non inductive) proof shows the calculational power of Galois connections operated via indirect equality, which are applicable to arbitrarily complex problem domains. 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.

The appreciation of such wide applicability and potential for program reasoning has led the authors of the current paper to embark on a project whose main aim is the design and implementation of a proof assistant — the *'G'alculator* — solely based on Galois connections, their algebra and associated tactics such as (6) above.

Galculator does not intend to be a classical theorem prover (TP) because it is not usable in arbitrary proofs: only in those where adjoint operators of Galois connections participate. However, it can be very useful when used together with a "host" TP such as e.g. Coq (Bertot and Castéran 2004).

The prototype of this tool is currently being implemented in Haskell by resorting to generalized algebraic data types (GADTs). Elsewhere we have given a brief overview of this tool from a term rewriting perspective (Silva and Oliveira 2008). In the current paper we give a detailed account of the design, which is a full-fledged example of the use of state-of-the-art Haskell technology (Peyton Jones 2003), incorporating GADTs, existential data types, parsing combinators, strategic term rewriting combinators and polymorphic type representation. Sources and examples of application are available from the project's website (`www.di.uminho.pt/research/galculator`).

1.1 Structure of the paper

Section 2 gives a short introduction to the use of the Haskell type system in the implementation of domain specific languages (DSLs). Then a brief account of the theory behind the *Galculator*'s proof strategies is presented: Galois connections in Section 3 and the pointfree transform in Section 4. Section 5 describes the implementation of the *Galculator* prototype in some detail. Section 6 contains a summary of the related work. Finally, Section 7 concludes and points to future work.

2. Brief overview of Haskell

This section provides a brief introduction to Haskell, ranging from simple notation conventions to the advanced features of its type system that will be mentioned in the rest of this paper. This includes a brief explanation of the potential of using functional programming in the implementation of domain specific languages.

2.1 Haskell

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

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

$$
\begin{array}{l}
id :: a \to a \\
id\ x = x
\end{array}
$$

meaning that it "ignores" the type of its argument. It is parametric because a will be instantiated with the actual type of the argument at run-time (for instance, $id\ True$ will instantiate a to $Bool$).

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

$$
\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
\quad (\equiv), (\not\equiv) :: a \to a \to Bool
\end{array}
$$

defines two class functions: $\equiv$ for equality and $\not\equiv$ for inequality. Every instance of the Eq class must provide an implementation of the two functions (in fact, only one of them is needed because the other is just its negation). Type classes allow for generic code. For instance, we can define a generic function which removes duplicates from a list which works with any type:

$$nub :: Eq\ a \Rightarrow [a] \to [a]$$

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

Algebraic data types (ADTs) are the mechanism used in Haskell in order to declare new data types. An ADT declaration specifies

how inhabitants of the type can be built, i.e., its constructors, like in an algebraic definition. ADTs subsume parameterized, union, enumeration and recursive types in just one device, for instance

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

where Nil is the empty list and $Cons$ the list append constructor function.

Generalized abstract data types (GADTs) provide an extension to this device. They extend the capabilities of ADTs by introducing a new syntax for declarations where constructor types are explicitly spelt out. For instance, $List\ a$ will be written

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

using GADTs notation. What is this useful for?

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

$$\begin{array}{l}\mathbf{data}\ Type\ a\ \mathbf{where}\\ \quad Int\ \ :: Type\ Int\\ \quad List\ \ :: Type\ a \to Type\ [a]\\ \quad \cdot \times \cdot :: Type\ a \to Type\ b \to Type\ (a, b)\\ \quad \ldots\end{array}$$

If ADTs were used, the return type of all the constructors above would be bound to $Type\ \ a$; with GADTs each such type is restricted to a more *precise* type. This means that a is no longer parametric; it has become an *index type* which reflects the type of the term built. Moreover, this example introduces another feature of GADTs: the use of *singleton* (or *representation*) types. As (Sheard et al. 2005) put it, *"every singleton type completely characterizes the structure of its single inhabitant, and the structure of a value in a singleton type completely characterizes its type"*. As we will see in the sequel, this property allows for a *reflection* mechanism on the Haskell type system, and to introduce dynamic typing in a static context.

Another important feature of the Haskell type system are *existential types*. They allow us to introduce arbitrary types into type definitions, hiding them from the enclosing context. A traditional example is the definition of a list with elements of different types. We can define an existentially quantified type as follows:

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

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

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

written in Haskell pseudo-code. Using the T type a heterogeneous list can be built; however, values cannot be taken outside the constructor because that would break static type safety. Moreover, no operation can be performed because the type is too general. However, using type classes the existentially quantified variable can be constrained. For instance, if we restrict the quantified types to be instances of the $Show$ class (this provides a method $show :: Show\ a \Rightarrow a \to String$ for building string representations) we can define $\forall a.\ Show\ a \Rightarrow MkT'\ a$. A heterogeneous list of this type can be traversed in order to obtain string representations of the elements.

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

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

2.2 Monads

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

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

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

$$\begin{array}{l}return\ a \gg\!= f = f\ a\\ m \gg\!= return = m\\ (m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \to f\ x \gg\!= g)\end{array}$$

Although all instances of $Monad$ must instantiate both functions, very different effects can be achieved by changing the definitions. Thus, the semantics of the program depends of the underlying monad. Haskell provides a $\mathbf{do}$ notation similar to an imperative programming style as syntactical sugar for successive binds.

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

Of special importance in this work is the use of $MonadPlus$ and $MonadOr$. Although providing different behaviors they are many times confused. Currently, only $MonadPlus$ is part of the Haskell standard libraries but there is a discussion for reformulating the structure in order to introduce $MonadOr$. (More details are available from the Haskell wiki.)

Both of them provide a $mzero$ operator for modeling failure. However, they handle failure in different ways. $MonadPlus$ uses $mplus$ operator which obeys the *left-distribution* law: $(s\ `mplus`\ r) \gg\!= t = (s \gg\!= t)\ `mplus`\ (r \gg\!= t)$. This specifies a backtracking behavior, where all the possible combinations are tried. The *Parsec* library of parsing combinators (Leijen and Meijer 2001) to which we resort in Section 5.5 uses $MonadPlus$.

$MonadOr$ defines a $morelse$ operator which obeys the *left catch* law: $return\ a\ `morelse`\ r = return\ a$. This specifies a left biased behavior: the second argument is only tried if the first one fails.

2.3 GADTs and Domain Specific Languages

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

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

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

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

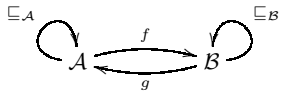
Functional languages are particularly apt to EDSL development thanks to their natural support for ADTs and the availability of generalized algebraic data types (GADTs), which offer new possibilities for EDSL implementation. While ADT data constructors only keep term information, GADT's constructors add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

3. Galois connections

As already explained, the mathematical concept of a Galois connection is the essence of the *Galculator*. This section presents an overview of Galois connections and their algebraic properties. 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 $b \in \mathcal{B}$:

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

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

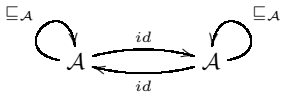


which we in-line in text by writing $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}) \xleftarrow{(f,g)} (\mathcal{B}, \sqsubseteq_{\mathcal{B}})$. Both notations always represent the source domain of the lower adjoint on the left. As we shall see, the arrow notation emphasizes the categorial structure of Galois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

Galois connections have several important properties which relate them to the underlying ordered structures, of which Table 1 gives a summary. (See (Backhouse 2004) 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 (see details in section 5.3). Let us see some of these combinators.

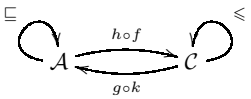
The simplest of all Galois connections is the identity,



Property	*Description*
$f\ a \sqsubseteq_B b \Leftrightarrow a \sqsubseteq_A g\ b$	*"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.*

where id is the polymorphic identity function mentioned in section 2.1. 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



(Note the reverse composition order in which adjoints compose.) Composition is an associative operation and the identity Galois connection is its unit. Thus Galois connections form a category.

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. That is, from $(\mathcal{A}, \sqsubseteq) \xleftarrow{(f,g)} (\mathcal{B}, \preceq)$ one builds the converse connection $(\mathcal{B}, \succeq) \xleftarrow{(g,f)} (\mathcal{A}, \sqsupseteq)$.

Moreover, every relator $\mathcal{F}$ [3] that distributes through binary intersections preserves Galois connections (Backhouse and Backhouse 2004). 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 e.g. $(\mathcal{A}, \sqsubseteq) \xleftarrow{(\delta, \sqcap)} (\mathcal{A} \times \mathcal{A}, \sqsubseteq \times \sqsubseteq)$ (where δ is the diagonal function $\delta a = (a, a)$) which captures the definition of greatest lower bounds in partial orders:

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

Back to Section 1, it is easy to see that the two connections (3,4) involved in our introductory example can be pictured as, respectively, $(\mathbb{N}, \leqslant) \xleftarrow{((\times y),(\div y))} (\mathbb{N}, \leqslant)$ (for nonzero y) and $(\mathbb{Z}, \leqslant) \xleftarrow{((-b),(+b))} (\mathbb{Z}, \leqslant)$.

Notations $(\times y)$, $(\div y)$, etc. call for an explanation: since the operations in equations (3,4) are binary, in order to form Galois connections one of their arguments must be fixed, so that they

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

Pointwise	Pointfree
$\langle \exists\, c \;::\; bRc \wedge\ cSa \rangle$	$b(R \circ S)a$
$\langle \forall\, x \;::\; xRb \Rightarrow\ xSa \rangle$	$b(R \setminus S)a$
$\langle \forall\, x \;::\; bRx \Rightarrow\ aSx \rangle$	$b(S/R)a$
$bRa \wedge\ cSa$	$(b,c)\langle R,S\rangle a$
$bRa \wedge\ dSc$	$(b,d)(R \times S)(a,c)$
$bRa \wedge\ bSa$	$b(R \cap S)a$
$bSa \vee bSa$	$b(R \cup S)a$
$(f\ b)R(g\ a)$	$b(f^\circ \circ R \circ g)a$
$b = a$	$b\ id\ a$
$True$	$b\ \top\ a$
$False$	$b\ \bot\ a$
$\langle \forall\, a,b \;::\; bRa \Rightarrow\ bSa \rangle$	$R \subseteq S$
$\langle \forall\, a,b \;::\; bRa \Leftrightarrow\ bSa \rangle$	$R = S$
$\langle \forall\, a \;::\; aRa \rangle$	$id \subseteq R$

Table 2. Sample of PF-transform rules.

become unary functions on the other argument. In general, given binary operator θ, one defines two unary *sections* [4] $(a\theta)$ 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.

4. Pointfree transform

The overall operation of the *Galculator* is based on transforming and rewriting terms involving adjoints of Galois connections. As is well-known in term rewriting, one must be very careful about variables: free and bound variables make substitutions tricky.

We overcome this complexity by transforming variable-level logical formulæ into *pointfree* formulæ involving binary relations only. In this *pointfree transform* (PF-transform for short) (Tarski and Givant 1987; Bird and de Moor 1997; Oliveira and Rodrigues 2004) variables are abstracted from formulæ in the same way Backus develops his algebra of programs (Backus 1978). The main difference stays in the fact that we are transforming logical formulæ while Backus was doing so for functional terms only [5].

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

$$b(R \circ S)c \quad \Leftrightarrow \quad \langle \exists\, a \;::\; b\ R\ a\ \wedge\ a\ S\ c \rangle \tag{8}$$

(Notation $b\ R\ a$ means that pair (b,a) is in R.) This means that variables are only needed in functional sections of shape $(a\theta)$ and (θb), as explained above.

A brief overview of the PF-transform is presented in Table 2. One particular rule of the PF-transform which is specially helpful in removing variables from expressions is

$$(f\ b)\ R\ (g\ a) \quad \Leftrightarrow \quad b(f^\circ \circ R \circ g)a \tag{9}$$

[4] This terminology is taken from functional programming, where sections are a very popular programming device (Peyton Jones 2003).

[5] See (Tarski and Givant 1987) for the theoretical foundations of this transform. The idea of encoding predicates in terms of relations was initiated by De Morgan in the 1860s and followed by Peirce who, in the 1870s, found interesting equational laws of the calculus of binary relations (Pratt 1992). The pointfree nature of the notation which emerged from this embryonic work was later further exploited by Tarski and his students (Tarski and Givant 1987). In the 1980's, Freyd and Ščedrov developed the notion of an *allegory* (a category whose morphisms are partially ordered) which finally accommodates the binary relation calculus as special case (Freyd and Ščedrov 1990).

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 (9) to both sides of (7) yields, for all suitably typed a, b

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

which leads to PF relational *equality*

$$f^\circ \circ \sqsubseteq_{\mathcal{B}} \quad = \quad \sqsubseteq_{\mathcal{A}} \circ g \tag{10}$$

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.

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

$$f = g \quad \Leftrightarrow \quad \preceq \circ f = \preceq \circ g \tag{11}$$

(or, equivalently, $f = g \ \Leftrightarrow\ f^\circ \circ \preceq\ = g^\circ \circ \preceq$) instantiates indirect equality can be easily checked by putting variables back via (9).

Switching to PF-terms makes the operation of the *Galculator* a lot easier. In fact, the pointfree representation can be regarded as an extension to relations of the combinatory logic approach to functional notation (Turner 1979).

For example, let us see how the calculation in the introduction is actually performed inside the *Galculator*: first of all, equations (3,4) become families of PF-equalities

$$(\times y)^\circ \circ \leqslant \quad = \quad \leqslant \circ (\div y) \tag{12}$$
$$(-b)^\circ \circ \leqslant \quad = \quad \leqslant \circ (+b) \tag{13}$$

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

$$\begin{array}{ll} & \leqslant \circ (\div y) \\ = & \quad \{\ \text{Galois connection (12) assuming } y > 0\ \} \\ & (\times y)^\circ \circ \leqslant \\ = & \quad \{\ \text{cancellation, thanks to (13) — steps omitted}\ \} \\ & ((-y) \circ (\times y))^\circ \circ \leqslant \circ (-y) \\ = & \quad \{\ \text{distribution law}\ \} \\ & ((\times y) \circ (-1))^\circ \circ \leqslant \circ (-y) \\ = & \quad \{\ \text{distribution of converse through composition}\ \} \\ & (-1)^\circ \circ (\times y)^\circ \circ \leqslant \circ (-y) \\ = & \quad \{\ (12)\ \text{again}\ \} \\ & (-1)^\circ \circ \leqslant \circ (\div y) \circ (-y) \\ = & \quad \{\ (13)\ \text{again}\ \} \\ & \leqslant \circ (+1) \circ (\div y) \circ (-y) \end{array}$$

From this, the *Galculator* uses tactic (11) to infer equality.

5. *Galculator*

This section describes the *Galculator* prototype, starting from its basic design principles and general architecture and proceeding to the technical details of the implementation.

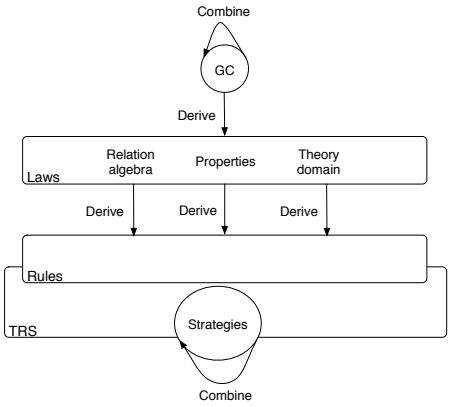


Figure 1. Design principles of the *Galculator* prototype

5.1 Design principles

Galois connections are the *Galculator*'s main building block. They are combined as needed, forming arbitrarily complex new connections from the existing ones. From each Galois connection the *Galculator* derives its properties as given by Table 1 which, together with laws from relation algebra (Bird and de Moor 1997) and algebraic properties of the particular domain of the problem being solved, form the set of *laws* of the system. In order to represent all these concepts (Galois connections, relation algebra, particular theory domains), several embedded DSLs are defined.

Galculator proofs are transformations of the abstract representation of the equality being proved. These transformations are made according to the equalities enabled by the laws of the system. However, laws are objects arising from the theoretical level; they cannot be applied to representations. Thus, a mechanism is defined for deriving functional applications of the available laws in the form of *rewrite rules*. The application of such rules in performed by a strategic term rewrite system (TRS).

Basic rewrite strategies can be combined in order to build more complex ones, according to the complexity of the problem. Moreover, with the same set of rules several different rewrite systems can be easily built and tried.

A summary of the design principles which guide the *Galculator* prototype is given in Figure 1.

5.2 Architecture

The *Galculator* is divided in several logical modules. Below we give an overview of the system before presenting a more technical description.

Interpreter. The command line interpreter provides for interactive user interfacing. Several options are offered: loading modules, exploiting Galois connection algebra, checking expressions and doing proofs. Currently, these are performed in interactive proof mode. At each step, the user can choose a rule derived from the set of laws available from the system. Rules can be applied using the strategies built from the combinators provided by the term rewriting system. The system offers hints about the applicable rules in the current proof step. At the end, a complete proof log, with all equational steps and justifications is made available.

Parser. Several domain specific languages (DSL) are available in order to express the concepts in use: Galois connections, relations, orders, functions and so on. For each one a parser was implemented using parsing combinators (Leijen and Meijer 2001). This technique makes it easy to use a DSL inside another simply by calling the respective parsing combinators. 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 the Coq (Bertot and Castéran 2004) theorem prover. (See Section 7.)

Type inference. Types are useful in finding errors, not only in programs but also in proofs: they give insight in some misleading details that are often overlooked. The *Galculator* prototype is a typed environment with its own type system, based on the Haskell type system using a type representation. Altogether, the user is released from having to provide explicit types in expressions.

The *Galculator* type system supports parametric polymorphism. The type representation is extended with support for type variables. However, it is not possible to rely on the type system of the host language alone in order to account this addition. Thus a unification mechanism on type variables has been implemented based on the Hindley-Milner algorithm (Milner 1978). Polymorphism is useful for deriving the so-called *free-theorems* of functions (Wadler 1989; Backhouse and Backhouse 2004), a kind of commutative property enjoyed by polymorphic functions solely inferred from their types.

Term rewriting system. The core of *Galculator* is its term rewriting system (TRS), whose rules (derived from the theory explained in Section 3) are applied to terms in order to build proofs. The system uses the flexibility of *strategies* (Lämmel and Visser 2003) and their combinatorial properties in order to build more complex proof strategies. Moreover, since the whole system is typed, the TRS is also typed, allowing for type directed rewriting rules.

Property inference. Galois connections are specified by their types (sets on which they are defined), the preorders involved and the adjoint functions. This component derives the properties stated in Table 1 from the starting specification and adds them to the system.

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

5.3 Representation

The concepts used in the system (relations, functions, orders, Galois connections) are represented by GADTs. As mentioned in Section 2.3, GADTs naturally induce an associated language. Thus, in fact, we are defining very small DSLs which relate to each other.

Types. The following data type encompasses the most used types in the domains we want to use our tool:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Type}\ a\ \textbf{where} \\
\quad \mathit{One} \quad :: \mathit{Type}\ \mathit{One} \\
\quad \mathit{Bool} \quad :: \mathit{Type}\ \mathit{Bool} \\
\quad \mathit{Char} \quad :: \mathit{Type}\ \mathit{Char} \\
\quad \mathit{String} :: \mathit{Type}\ \mathit{String} \\
\quad \mathit{Int} \quad :: \mathit{Type}\ \mathit{Int} \\
\quad \mathit{Float} \quad :: \mathit{Type}\ \mathit{Float} \\
\quad \mathit{List} \quad :: \mathit{Type}\ a \to \mathit{Type}\ [a] \\
\quad \mathit{Set} \quad :: \mathit{Type}\ a \to \mathit{Type}\ (\mathit{Set}\ a) \\
\quad \mathit{Maybe} :: \mathit{Type}\ a \to \mathit{Type}\ (\mathit{Maybe}\ a) \\
\quad \cdot \times \cdot \quad :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b) \\
\quad \cdot + \cdot \quad :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a + b) \\
\quad \cdot \rightharpoonup \cdot \quad :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a \rightharpoonup b)
\end{array}
$$

$$
\begin{array}{l}
Ord \quad :: Type\ a \to Type\ (PO\ a) \\
Fun \quad :: Type\ a \to Type\ b \to Type\ (a \leftarrow b) \\
Rel \quad :: Type\ a \to Type\ b \to Type\ (a \leftrightarrow b) \\
GC \quad :: Type\ a \to Type\ b \to Type\ (GC\ a\ b)
\end{array}
$$

$$
\begin{array}{ll}
\textbf{type}\ One \quad = () & \text{-- Unitary type} \\
\textbf{type}\ b \leftarrow a = a \to b & \text{-- Functions} \\
\textbf{data}\ b \leftrightarrow a & \text{-- Relations} \\
\textbf{data}\ PO\ a & \text{-- Pre-orders} \\
\textbf{data}\ GC\ b\ a & \text{-- Galois connections}
\end{array}
$$

We deviate from the usual Haskell representation for functions (we use $\leftarrow$ instead of $\rightarrow$) because the right to left arrow is visually more consistent with function composition which is the main connector of our pointfree calculus.

This representation works with a closed universe of types. So, parametric polymorphism is not possible. In type polymorphism type variables which range over the universe of types are allowed. In order to deal with parametric polymorphism we must enrich our type representation with another constructor which represents type variables:

$$
\begin{array}{l}
\textbf{data}\ Var \\
\textbf{type}\ Name = String \\
\textbf{data}\ Type\ a\ \textbf{where} \\
\quad \ldots \\
\quad TVar :: Name \to Type\ Var
\end{array}
$$

This means that all type variable representations have the same type (Var). Using $Type\ a$ does not work because it would not be possible to define a type equality mechanism over it. The drawback of the use of Var is that Haskell type inference mechanism fails to unify Var with any type, like it would do with a normal Haskell type variable. Thus, we cannot solely rely on the Haskell type-system and thus have to create our own unification mechanism (Section 5.4).

Combinators. The pointfree calculus presented in Section 4 presents a set of relational combinators. The relation calculus is the basis of all proofs once Galois connections are encoded in the pointfree style (recall Equation 10). Thus, we represent the combinators of Table 2 using constructors of a GADT. Here are some of the defined combinators:

$$
\begin{array}{l}
\textbf{data}\ R\ r\ \textbf{where} \\
\quad \cdot^{\circ} \quad :: R\ (b \leftrightarrow a) \to R\ (a \leftrightarrow b) \\
\quad \cdot \circ_{\cdot} \cdot :: Type\ b \to R\ (c \leftrightarrow b) \to R\ (b \leftrightarrow a) \to R\ (c \leftrightarrow a) \\
\quad \cdot \times \cdot :: R\ (b \leftrightarrow a) \to R\ (d \leftrightarrow c) \to R\ ((b,d) \leftrightarrow (a,c)) \\
\quad \ldots
\end{array}
$$

Given relations r and s with the right types, r° denotes the converse of r; $r \circ_b s$ denotes the composition of r with s where b is the common type; $r \times s$ denotes the product of two relations. Note the use of a type annotation $Type\ b$ in the definition of the composition operator. This is necessary during traversals of the representation in order to get the common type back. (This type is existentially quantified and otherwise it could not be known.)

Type lifting. Functions and orders are particular cases of relations. Thus, it should be possible to use them wherever a relation can; however their types do not match. Two embedding are defined for the purpose:

$$
\begin{array}{l}
\omega_o(\cdot) :: R\ (PO\ a) \ \to R\ (a \leftrightarrow a) \\
\omega_f(\cdot) :: R\ (b \leftarrow a) \to R\ (b \leftrightarrow a) \\
\quad \ldots
\end{array}
$$

$\omega_o(o)$ turns order o into a relation; $\omega_f(f)$ makes the function f a relation.

Sections. As explained in Section 3, many adjoints arise as sections of functions. We provide two sectioning operators:

$$
\begin{array}{l}
{}_{::\cdot}\cdot :: Type\ b \to R\ b \to R\ (a \leftarrow (b,c)) \to R\ (a \leftarrow c) \\
\cdot_{::\cdot} :: Type\ c \to R\ c \to R\ (a \leftarrow (b,c)) \to R\ (a \leftarrow b) \\
\ldots
\end{array}
$$

Given f a function with the right type and v a value with type t, ${}_{v::t}f$ denotes the left section of f; $f_{v::t}$ denotes the right section of f. Like in the composition operator, type annotations are again needed in order to retain existentially quantified types.

Application. Recall from Section 4 that variables are only needed (at PF-level) to express functional sections of shape $(a\theta)$ and (θb). We introduce them in our representation for this purpose:

$$
\begin{array}{l}
\cdot\ \$.\ \cdot :: Type\ b \to R\ (a \leftarrow b) \to R\ b \to R\ a \\
VAL :: Name \ \to Type\ a \to R\ a \\
\ldots
\end{array}
$$

$f \$_t v$ denotes the application of function f to a term v of type t; $VAL\ a\ t$ denotes a variable with name a and type t.

Galois connections. Our representation of Galois connections collects their adjoint functions and respective preorders. Moreover, the operations of Galois connection algebra are also provided:

$$
\begin{array}{ll}
GVar & :: Name \to R\ (b \leftarrow a) \to R\ (a \leftarrow b) \\
 & \quad \to R\ (PO\ b) \to R\ (PO\ a) \to R\ (GC\ a\ b) \\
GId & :: R\ (GC\ a\ a) \\
GComp & :: Type\ b \to R\ (GC\ c\ b) \to R\ (GC\ b\ a) \\
 & \quad \to R\ (GC\ c\ a) \\
GConv & :: R\ (GC\ b\ a) \to R\ (GC\ a\ b) \\
\ldots &
\end{array}
$$

Given two adjoint functions f and g of a Galois connection and fo and go their associated preorders, $GVar\ n\ f\ g\ fo\ go$ denotes Galois connection $(\mathcal{A}, go) \xleftarrow{(f,g)} (\mathcal{B}, fo)$ with name n; GId represents the identity Galois connection; $GComp\ g\ g'$ represents the composition of Galois connections g and g' with appropriated types; and $GConv\ g$ represents the converse connection of g.

Theories. Other constructors that represent operators in several domains (functions, integer arithmetics, real arithmetics and some other) are also defined. These will not be addressed in this paper for space economy.

5.4 Type equality and type unification

One advantage of using explicit type representations is that equality can be computed at run-time, allowing for the introduction of dynamic typing mechanisms in a static environment (Baars and Swierstra 2002). For this, a GADT definition is used:

$$
\textbf{data}\ Equal\ a\ b\ \textbf{where}\ Eq :: Equal\ a\ a
$$

This is called a witness type because it can only be built if the types are equal. This is ensured by the type checker by analysing the types of the type indexes. Moreover, thanks to this witness the type-checking mechanism can recognize values of the two types as interchangeable.

Given two type representations, the teq function tries to build an Eq witness of their equality: if the types are equal the witness is returned; otherwise it fails. Using GADTs computing type representation equality reduces to computing syntactical equality between data constructors:

$$
\begin{array}{l}
teq :: MonadPlus\ m \Rightarrow Type\ a \to Type\ b \to m\ (Equal\ a\ b) \\
teq\ Int\ Int = return\ Eq \\
teq\ (a \times b)\ (a' \times b') = \textbf{do}
\end{array}
$$

```
  Eq ← teq a a'
  Eq ← teq b b'
  return Eq
...
teq _ _ = mzero
```

However, implementing type equality over type variable representation leads to the question: when are two type variables of the same type? A type variable representation is just a placeholder, it can be replaced by another type representation. Using polymorphic type representations we cannot rely on the type checker in order to infer that types are equal. We have to implement a type unification mechanism which helps the type-checker to infer the types correctly.

Type unification can be easily solved with a unification algorithm such as the well-known Hindley-Milner type inference mechanism (Milner 1978). The algorithm receives a system of equations stating the supposed equalities between types. If some of the equalities do not hold, e.g., trying to unify integers with Booleans, the algorithm fails. Otherwise, a set of substitutions is returned with mappings from variables into the type which they should be instantiated to. One property of this algorithm is that if it succeeds, it returns the most general unifier. Function

$$\eta :: MonadPlus\ m \Rightarrow [Equation] \rightarrow m\ [Substitution]$$

implements this algorithm. Equations and substitutions are synonyms for the same type which is just a pair of type representations existentially quantified:

```
data Constraint where
  Unify :: Type a → Type b → Constraint
type Equation   = Constraint
type Substitution = Constraint
```

5.5 Parsing

Embedded DSLs imply that the user knows how to use the host language (write modules, compile files and so on). Thus, in order to allow the user to specify expressions in a textual format a parser was developed using the parsing combinators of the *Parsec* library (Leijen and Meijer 2001).

For each constructor of our representation a parsing combinator is defined. This combinatorial style makes it easy to embed DSLs inside others: all it is needed is to use the corresponding parser in a composable approach. While building ASTs using ADTs is almost straightforward, representations using GADTs pose some problems because their index type is only known at run-time: it is dependent of the input. This is circumvented by resorting to an existential data type to hide the type index

```
data Covert t = ∀x. Hide (t x)
```

maintaining static safeness. *Covert* is a common pattern (Sheard et al. 2005) where t can be parameterized with the data type we want to use. For instance, for building type representations:

```
type TypeBox = Covert Type
```

This can be manipulated in a type-safe manner provided the encapsulated value never escapes the scope of its quantification.

However, when trying to parse relational representations, for instance, hiding the index type is not enough. If we define a data type to encapsulate the representation and a parsing combinator,

```
type RBox = Covert R
parseR :: Parser RBox
```

when the result of the parsing function is used, for instance, by another parsing combinator in order to build a more complex term, the index type escapes from its scope and the compiler cannot guarantee type safeness of the code.

The solution is to add an explicit type representation sharing the same index type with the expression representation, changing the type of *parseR* accordingly:

```
data Exists singleton term =
  ∀t. Exists (singleton t) (term t)
type RType = Exists Type R
parseR :: Parser RType
```

Although the exact index type is not known, the type-checker knows that it must reflect the type representation (because it is a singleton type), being sufficient to ensure the static type safeness. So, for instance, the parsing combinator for the converse operator:

```
parseCONV :: Parser RType
parseCONV = do
  reserved "CONV"
  Exists (Rel t t') r ← parseR
  return (Exists (Rel t' t) (r°))
```

Since explicit type annotations are needed, the user has to provide them or the system has to infer them. Thanks to the unification mechanism, we just have to generate the equations. Polymorphic operators need to get fresh variable names in order to denote their type variable representation; other operators just have to be provided with the type representation corresponding to their types. An example of an operator that is polymorphic in its argument, but not in its result is *bang* (function that takes any value to the only inhabitant of the unitary type):

```
parseFBang :: Parser RType
parseFBang = do
  reserved "FBang"
  tid ← getFresh
  return (Exists (Fun One (TVar tid)) bang)
```

getFresh gets always fresh variables names from an infinite stream of identifiers due to lazy evaluation.

Unification is only needed when parsing relational combinators in which some variables must be equal. Composition is a good example of this:

```
parseCOMP :: Parser RType
parseCOMP = do
  reserved "COMP"
  Exists (Rel t3 t2) r1  ← parseR
  Exists (Rel t2b t1) r2 ← parseR
  constr   ← η [Unify t2 t2b]
  Hide t1' ← typeRewrite constr t1
  Hide t2' ← typeRewrite constr t2
  Hide t3' ← typeRewrite constr t3
  r1'      ← rCast constr (Rel t3' t2') r1
  r2'      ← rCast constr (Rel t2' t1') r2
  return (Exists (Rel t3' t1') (r1' ∘t2' r2'))
```

After parsing the two relational expressions $r1$ and $r2$ we have to make sure that they have a common type in order to be composable. Thus, a type equation is solved using unification and the set of substitutions of type variables is applied using *typeRewrite* (more details in Section 5.8). Next, the representation must also reflect type substitutions. Since constructors in GADTs retain the associated type information, a kind of type-safe "cast" is needed in order to reflect the new types. *rCast* is essentially an identity function where an expression of type r is transformed in an expression of type t, if types are compatible:

$$rCast :: MonadPlus\ m \Rightarrow [Substitution] \rightarrow Type\ t \rightarrow R\ r \rightarrow m\ (R\ t)$$

The need for the list of substitutions is justified by the relational combinators which have type annotations, like composition, where the substitutions have to be also applied for consistency.

Finally, the parsing function returns the newly built term with the respective type annotation. Since all computations are performed in a $MonadPlus$ context, if any of them fails, the all whole parsing function will fail.

5.6 Laws and rules

Recall from Section 5.1 that term *Law* refers to expressions at the theoretical level, while term *Rule* denotes functions applicable by the rewriting system. We use an equational approach to proofs, in which laws express equalities ($A = B$) or inequalities ($A \subseteq B$) between expressions. From an equality $A = B$ two rules can be inferred, one in each direction of rewriting: $A \rightarrow B$ and $B \rightarrow A$. This corresponds to the application of the Leibniz principle: if two objects are equal we can interchange them keeping the validity of the enclosing expression. Inequalities do not obey to this principle; they are handled by monotonicity: if $A \subseteq B$, E is an expression containing A and E' is the expression E with B substituted for A, then $E \subseteq E'$ [6]. Although two rewrite rules can be inferred from an inequality (cf. the converse orderings), in order to keep the system simple we only consider that $A \subseteq B$ induces a rule $A \rightarrow B$. The other direction can be easily achieved by inverting the direction of the calculus.

One advantage of using equational reasoning is that it is type preserving, i.e., expressions of both sides of equalities (or inequalities) have the same type. Our representation for laws reflects this fact, using an existentially quantified index type:

$$\begin{aligned}&\textbf{data}\ Law\ \textbf{where}\\ &\quad EQUIV :: Meta \rightarrow Type\ a \rightarrow R\ a \rightarrow R\ a \rightarrow Law\\ &\quad IMPL\ \ :: Meta \rightarrow Type\ a \rightarrow R\ a \rightarrow R\ a \rightarrow Law\end{aligned}$$

$EQUIV$ represents equality ($A = B$) and $IMPL$ represents inequality ($A \subseteq B$). The $Meta$ argument holds meta-information about the law: name, kind and possibly other useful information.

Following our design principles, laws can be specified in a purely declarative level using a textual notation from which the parser builds corresponding Law representations; which, in turn, are automatically converted into rewriting rules. A rewriting rule is defined as a polymorphic function, with type and expression representations as arguments:

$$\textbf{type}\ Rule = \forall a.\ Type\ a \rightarrow R\ a \rightarrow Rewrite\ (R\ a)$$

where $Rewrite$ is a monad that deals with effects during rewriting. (See more details in Section 5.8). The function that, from a law representation returns a rewriting function (rule), is defined as follows:

$$\begin{aligned}&getRule :: Law \rightarrow Rule\\ &getRule\ (EQUIV\ m\ t1\ r1\ r2)\ t2\ r = \textbf{do}\\ &\quad rcns \leftarrow rConstraint\ r1\ r && \text{-- 1}\\ &\quad cns \leftarrow \eta\ ([Unify\ t1\ t2] +\!\!+ rcns) && \text{-- 2}\\ &\quad Hide\ t1' \leftarrow typeRewrite\ cns\ t1 && \text{-- 3}\\ &\quad r' \leftarrow rCast\ cns\ t1'\ r && \text{-- 4}\\ &\quad r1' \leftarrow rCast\ cns\ t1'\ r1 && \text{-- 4}\\ &\quad r2' \leftarrow rCast\ cns\ t2\ r2 && \text{-- 5}\\ &\quad guard\ (r1' \equiv r') && \text{-- 6}\\ &\quad successEquiv\ m\ t2\ r\ r2' && \text{-- 7}\end{aligned}$$

[6] Note, however, that unrestricted substitutions inside sections of upper-adjoints may lead to anomalous behaviour due to their antitonicity. How to restrict such behaviour is subject of on-going research.

The general principle of this function is that given an argument expression r and its type $t2$ it will try to match these with the left hand side of the law $r1$ and its corresponding type $t1$. If they are compatible the right hand side of the law is returned; otherwise, the function fails.

Each step of $getRule$ are explained below. It should be noticed that we are working in the context of $MonadPlus$, meaning that if one of the steps fails, $getRule$ also fails.

1. Type equations are generated by the $rConstraint$ function from comparing the two expressions we are trying to match. This is necessary in order to ensure that type annotations inside data constructors are correctly unified. For instance, when trying to match $\cdot \circ_{(TVar\ \texttt{"a"})} \cdot$ and $\cdot \circ_{Int} \cdot$, an equation $Unify\ (TVar\ \texttt{"a"})\ Int$ should be generated. In case the two expressions do not match, $rConstraint$ fails.
2. Type representations of the argument and law are unified, together with the equations generated in the previous step. The unification failure means that types are incompatible and no rule can be derived. Otherwise, a set of substitutions is returned.
3. The substitutions obtained in the previous step are applied to the type representation of the law.
4. The type-safe cast function is applied to the two expressions we want to compare. This is needed because these have to be of the same type.
5. The type-safe cast function is applied to the right hand side of the law in order to make it possible to return a value with the right type $t2$, since our system is type preserving.
6. The argument expression and left hand side of the law, once casted, are compared for equality.
7. The $successEquiv$ function deals with the details of the $Rewrite$ monad (for instance, adds a successful rewriting to a proof log). Otherwise, it could be just $return\ r2'$, meaning that it is possible to rewrite r into $r2'$, both having type $t2$.

The implementation for $IMPL$ is completely analogous, the unique difference being the use of $successImpl$ in the last step. The inverse rewrite rule (the right hand side by the left hand side of the law) is obtained through a similar function $getRuleInv$, the only difference being the inversion of variables.

5.7 Galois connections properties

Galculator exploits Galois connection algebra and properties. These properties are equational laws (Table 1) representable in our system. What is needed is a way of automatically derive the properties from the definitions.

For each property, a function is defined, receiving the representation of a Galois connection together with its type representation. For instance, the pointfree version of the shunting property is generated by function:

$$\begin{aligned}&gcShunting :: Type\ (GC\ a\ b) \rightarrow R\ (GC\ a\ b) \rightarrow Law\\ &gcShunting\ (GC\ a\ b)\ (GVar\ nm\ ladj\ uadj\ lord\ uord) =\\ &\quad EQUIV\ meta\ (Rel\ a\ b)\\ &\qquad ((\omega_o(uord)) \circ_a (\omega_f(uadj)))\\ &\qquad (((\omega_f(ladj))^\circ) \circ_b (\omega_o(lord)))\\ &\quad \textbf{where}\\ &\qquad meta = Meta\ (\texttt{"Shunting: "} +\!\!+ nm)\ (Just\ SHUNT)\\ &gcShunting\ (GC\ a\ b)\ GId = \ldots\end{aligned}$$

The cancellation laws use the $IMPL$ constructor because they are not equalities:

$$\begin{aligned}&gcCancelUpper :: Type\ (GC\ a\ b) \rightarrow R\ (GC\ a\ b) \rightarrow Law\\ &gcCancelUpper\ (GC\ a\ b)\ (GVar\ nm\ ladj\ uadj\ \_\ uord) =\end{aligned}$$

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

Table 3. Strategic combinators implemented in the *Galculator* TRS.

$$
\begin{array}{l}
\quad IMPL\ meta\ (Rel\ a\ a) \\
\quad\quad (\omega_o(uord)) \\
\quad\quad ((\omega_o(uord)) \circ_a ((\omega_f(uadj)) \circ_b (\omega_f(ladj)))) \\
\quad \textbf{where} \\
\quad\quad meta = Meta\ (\texttt{"Cancellation: "} +\!\!+ nm)\ (Just\ CANC) \\
gcCancelUpper\ (GC\ a\ b)\ GId = \ldots
\end{array}
$$

5.8 Term rewriting system

The term rewriting system (TRS) puts the rules derived from Galois connections to work.. As already mentioned, it is based on strategic techniques (Visser and Benaissa 1998; Cirstea et al. 2001).

Strategic term rewriting uses simple basic strategies and combinators in order to build arbitrarily complex strategies in a declarative style. Strategies can be reused and combined in different ways in order to obtain different TRS. In the context of *Galculator*, the use of rewrite strategies on proofs can be compared to the use of tactics on traditional theorem provers.

Our TRS is not only strategic but also typed, allowing for type-dependent rewriting. The first framework combining strategic programming and strong typing in the functional paradigm was *Strafunski* (Lämmel and Visser 2003).

Two strategic TRS have been implemented: one for expressions and another for types. Although they use similar strategies, they are slightly different.

Expressions. The definition of rewriting rules for expressions was already presented in Section 5.6. In fact, $Rule$ is an instance of the general type of rewrite strategies

$$\textbf{type}\ GenericM\ m = \forall a.\ Type\ a \to R\ a \to m\ (R\ a)$$

parameterized by the $Rewrite$ monad which is defined using monad transformers in order to combine failure, non-determinism and a proof log. However, $GenericM$ can be instantiated differently in order to deal with different kinds of effects.

Table 3 summarizes the strategy combinators that are defined. Except the traversal combinators (all and one), the other are just renamings for monadic operations (recall Section 2.2). *nop* and $\triangleright$ are the return and bind operators of the monad class; $\bot$ and $\oplus$ the $mzero$ and $mplus$ operators of $MonadPlus$; $\oslash$ the $morelse$ operator of $MonadOr$.

Special mention should go to the fact that two different choice operators have been defined: one for *left-biased* choice and another for *non-deterministic* choice. The latter ($\oplus$) is based on the $MonadPlus$ monad, thus obeying the left-distribution law. The use of non-deterministic choice is important when all different paths should be tried but it comes with a performance penalty since the search space expands.

The left-biased choice combinator ($\oslash$) is based on the $MonadOr$ and thus obeys the left catch law: the right strategy is only tried if the first one fails. This provides a mechanism for cutting down unnecessary cases and restricting the search space.

The cost of traversal operators (*all*, *one*) having to deal with boilerplate code is minimized by using a *spine* representation inspired on (Hinze et al. 2006):

$$
\begin{array}{l}
\textbf{data}\ Typed\ a\ \textbf{where} \\
\quad (:|) \quad :: Type\ a \to R\ a \to Typed\ (R\ a) \\
\textbf{data}\ Spine\ a\ \textbf{where} \\
\quad Constr :: a \to Spine\ a \\
\quad (\diamond) \quad :: Spine\ (a \to b) \to Typed\ a \to Spine\ b \\
fromSpine :: Spine\ a \to a \\
fromSpine\ (Constr\ c) \quad = c \\
fromSpine\ (f\ \diamond\ (\_ :|\ a)) = (fromSpine\ f)\ a \\
toSpine :: Type\ a \to R\ a \to Spine\ (R\ a) \\
\ldots
\end{array}
$$

The $Spine$ data type is used in order to build a standard representation for constructors: $Constr$ is used for constructors without arguments, $\diamond$ is used for constructors with arguments. The $fromSpine$ function maps the spine representation back to actual expressions; $toSpine$ builds a spine representation from a given expression (the corresponding type representation is needed also). Changes in expression representation thus only affect $toSpine$.

The implementation of the *all* and *one* combinators boils down to defining how to traverse $Constr$ and $\diamond$ and use $toSpine$ and $fromSpine$ to map expressions between representations. This makes the implementation of the strategies independent of the representation and reduces the cost of changing.

More complex strategies have been defined using the basic combinators

$$
\begin{array}{ll}
try\ s & = s \oslash nop \\
many\ s & = (s \triangleright many\ s) \oslash nop \\
many1\ s & = s \triangleright many\ s \\
once\ s & = s \oslash one\ (once\ s) \\
topdown\ s & = s \triangleright all\ (topdown\ s) \\
bottomup\ s & = all\ (bottomup\ s) \triangleright s \\
innermost\ s & = all\ (innermost\ s) \triangleright try\ (s \triangleright innermost\ s)
\end{array}
$$

For instance, if we have a rule that associates any operator to the left, say $assocLeft$, and we want to associate an expression to the left we just have to use $innermost\ assocLeft$.

Types. The rewriting system for types uses the same principles and combinators as the TRS for expressions. However, it is not type preserving because changing type representations will imply having a different type in the end.

In order to make the type system consider all rewritings as type preserving, the new type is hidden using another existential data type (Cunha et al. 2006):

$$
\begin{array}{l}
\textbf{data}\ View\ a\ \textbf{where} \\
\quad View :: Type\ b \to View\ (Type\ a)
\end{array}
$$

Thus, the rewrite rule for type will have the type

$$
\begin{array}{l}
\textbf{type}\ Rule = \forall a.\ \forall m.\ MonadPlus\ m \\
\qquad\qquad \Rightarrow Type\ a \to m\ (View\ (Type\ a))
\end{array}
$$

The result of the rewriting is exactly of the same type, but with the new type hidden inside $View$. Since it is existentially quantified it cannot be used outside the scope of the quantification. However, we can pass it to other existential types, using the $Covert$ pattern:

$$
\begin{array}{l}
view2Box :: View\ (Type\ a) \to TypeBox \\
view2Box\ (View\ t) = Hide\ t
\end{array}
$$

Now, it can be manipulated providing that the result never falls outside an existentially quantified type. Using this approach, the function

$$typeRewrite :: MonadPlus\ m \Rightarrow [Substitution] \rightarrow Type\ t \rightarrow m\ TypeBox$$

receives a list of substitutions to be applied to a type t. Since substitutions on types are obviously not type-preserving, the result has to be encapsulated inside a $TypeBox$.

6. Related work

Galois connections in Coq. Reference (Pichardie 2005) presents a representation of Galois connections in Coq (Coquand and Huet 1988) developed in the context of work on *abstract interpretation*. Adjoints are defined over complete lattices (a stricter requirement than in the general theory). Proofs of the general properties that Galois connections enjoy are defined in order to be executed in Coq. 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 discharge proof obligations about adjoints prior to loading these into our system.

2LT. The core of the *Galculator* is inspired on the *2LT* system (Cunha et al. 2006). 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 (Cunha and Visser 2007) 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. Note that *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 (Oliveira 2007).

PF-ESC. This tool, which performs *pointfree extended static checking* (Necco et al. 2007) and is also inspired on *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. An advantage of using properties is that the system is more flexible in so far as allowing for new kinds of relation. Moreover, the type-lifts of our approach are not needed. However, predicate functions which calculate the properties of expressions are required in order to apply certain transformations. This makes the system not extensible because rewrite equations must be hard-wired into functions. Since the *Galculator* is based on types, predicate function are not needed and the rewrite rules can be purely declarative. Moreover, the representation used in the *Galculator* is statically safer, since incorrect constructions are not allowed.

Proof processor system. The authors of (Bohorquez and Rocha 2005) advocate the use of the calculational approach proposed by Dijkstra and Scholten in teaching discrete maths. Based on the *E* logical calculus, a tool was developed in Haskell to exploit equational proofs written in the *Z* notation (Spivey 1989). The system helps the user 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.

7. Concluding remarks

The *Galculator* is a proof assistant which implements an innovative approach to theorem proving, different from what is traditional in the field: it is solely based on Galois connections, their algebra and associated tactics. Thanks to Galois connection algebra, it builds new connections from old thanks to a number of combinators enabling such constructions on the fly. It is based on the tactic of indirect equality, which fits naturally with Galois connections.

This paper gives an account on the development of a prototype of the tool written in Haskell. Most of the techniques employed are not new, they are applications of work reported elsewhere in the literature. Arguably, the extension of the type representation in order to support polymorphic type representations and the support for unification is new; we are not aware of any other implementation, although it may exist.

To the best of our knowledge, the *Galculator* is the first proof engine ever to combine and calculate directly with PF-transformed Galois connections. We regard its current prototype as a non-trivial illustration of the power of functional programming advanced features for building prototypes of complex systems. It manages to combine many often publicized distinctive features of functional languages: GADTs, existential data types, combinatory approaches (parsing, rewriting), the support for embedded DSLs, computations as monads, higher-order functions and some other. Specially, the use of GADTs and existential data types allows mixing static and dynamic typing in a powerful way, making it possible to guarantee the static safeness of objects whose type will only be known at runtime, stepping into the power of dependent typing.

We decided to develop our own rewriting system instead of using another rewriting engine, e.g. Stratego (Visser 2001) or Maude (M. Clavel and Meseguer 2000), mainly because the typed behavior of the system would be lost. Moreover, translating *Galculator* laws into a foreign rewriting engine would require 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, state information may be required to implement some of the future features. This is easily accomplished by using monad transformers.

The tool is still in the prototype stage, thus many ideas are still left to be explored 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.

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 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 prover: it works only with well-defined situations involving Galois connections. Combined with other theorem provers it can behave like a specialized *add-on* component able to discharge proofs wherever terms involve adjoints of known connections. Currently, we are working on integrating the *Galculator* with Coq (Silva et al. 2008), either following the *believing* or the *skeptical* approach, as discussed by (Delahaye and Mayero 2005) in

integrating Coq with Maple. In the first case, *Galculator*-proven assertions are added as axioms. In the second, the idea is to define tactics in Coq which exploit Galois connection properties and invoke the *Galculator* in order to use the built-in strategies to prove the the correctness of the steps. The resulting proofs are then replayed using reflection to build trusted proofs in Coq.

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 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.

Thanks are also due to the anonymous referees for insightful comments which improved the quality of the original submission.

The first author is supported by the *Fundação para a Ciência e a Tecnologia* under grant number `SFRH/BD/19195/2004`.

References

C. Aarts, R.C. Backhouse, P. Hoogendijk, E. Voermans, and J. van der Woude. A relational theory of datatypes. Available from `www. cs. nott. ac. uk/~rcb/ papers`, December 1992.

J.C. Almeida. Program verification in Coq, 2008. Lecture notes for MAP-I course on *Program Semantics, Verification, and Construction* available from `http://twiki.di.uminho.pt`.

A.I. Baars and S.D. Swierstra. Typing dynamic typing. In *ICFP '02: Proc. 7th ACM SIGPLAN Int. Conf. Functional Programming*, pages 157–166. ACM Press, 2002.

K. Backhouse and R.C. Backhouse. Safety of abstract interpretations for free, via logical relations and Galois connections. *Sci. Comput. Program.*, 51(1-2):153–196, 2004.

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

J.W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8): 613–641, 1978.

Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer, 2004.

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

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.*, pages S3C/1–S3C/8, 2005.

J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104, New York, NY, USA, 2002. ACM.

H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In Furio Honsell, editor, *FOSSACS, ETAPS'2001*, LNCS, pages 166–180, Genova, Italy, April 2001. Springer-Verlag.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 73(2/3), 1988.

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).

A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. In FM'06 , volume 4085 of *LNCS*, pages 284–289. Springer-Verlag, Aug. 2006.

D. Delahaye and M. Mayero. Dealing with algebraic expressions over a field in coq using maple. *J.Symb.Comput.*, 39(5):569–592, 2005.

J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

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

R. Hinze and A. Löh. Guide2lhs2tex (for version 1.13), February 2008.

R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In *Proc. 8th Int. Symp. on Functional and Logic Programming*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.

D. Jackson. *Software abstractions: logic, language, and analysis*. The MIT Press, Cambridge Mass., 2006. ISBN 0-262-10114-9.

M. P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *AFP'95*, volume 925 of *LNCS*, pages 97–136. Springer, 1995.

R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. PADL'03*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

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.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

C. Necco, J.N. Oliveira, and J. Visser. Extended static checking by strategic rewriting of pointfree relational expressions. Technical Report FAST:07.01, CCTC Research Centre, University of Minho, 2007.

J.N. Oliveira. *Transforming Data by Calculation*. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE 2007 Proceedings*, pages 139–198, July 2007.

J.N. Oliveira and C.J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In *MPC'04*, volume 3125 of *LNCS*, pages 334–356. Springer, 2004.

O. 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.

D. Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés*. PhD thesis, Univ. de Rennes, 2005.

V. Pratt. Origins of the calculus of binary relations. In *Proc. of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 248–254, Santa Cruz, CA, 1992. IEEE Computer Soc.

T. Sheard, J. Hook, and N. Linger. GADTs + Extensible Kinds = Dependent Programming. Technical report, Portland State University, 2005. URL `http://www.cs.pdx.edu/ sheard`.

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

P.F. Silva, J.C. Almeida, and J.N. Oliveira. Galculator meets Coq. Technical report, CCTC, University of Minho, 2008. (In preparation).

J.M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989. C.A. R. Hoare.

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

D. Turner. A new implementation technique for applicative languages. *Software-Practice & Experience*, 9:31–49, 1979.

E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *RTA*, volume 2051 of *LNCS*, pages 357–361. Springer, May 2001.

E. Visser and Z. Benaissa. A Core Language for Rewriting. In C. Kirchner and H. Kirchner, editors, *WRLA'98*, volume 15 of *ENTCS*, Pont-à-Mousson, France, September 1998. Elsevier Science.

P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice, France*, 1990.

P. Wadler. Theorems for free! In *Proc. of 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept. 1989*, pages 347–359. ACM Press, New York, 1989.