A Rational Reconstruction of a System for Experimental Mathematics

Jacques Carette[1], William M. Farmer[1], and Volker Sorge[2]

[1]Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada
{carette,wmfarmer}@mcmaster.ca
http://www.cas.mcmaster.ca{˜carette,˜wmfarmer}

[2]School of Computer Science, University of Birmingham, UK,
V.Sorge@cs.bham.ac.uk, http://www.cs.bham.ac.uk/˜vxs

Abstract. In previous papers we described the implementation of a system which combines mathematical object generation, transformation and filtering, conjecture generation, proving and disproving for mathematical discovery in non-associative algebra. While the system has generated novel, fully verified theorems, their construction involved a lot of ad hoc communication between disparate systems. In this paper we carefully reconstruct a specification of a sub-process of the original system in a framework for trustable communication between mathematics systems put forth by us. It employs the concept of biform theories that enables the combined formalisation of the axiomatic and algorithmic theories behind the generation process. This allows us to gain a much better understanding of the original system, and exposes clear generalisation opportunities.

1 Introduction

Over the last decade several environments and formalisms for the combination and integration of mathematical software systems [2, 15] have been proposed. Many of these systems aim at a traditional automated theorem proving approach, in which a given conjecture is to be proved or refuted by the cooperation of different reasoning engines. However, they offer little support for experimental mathematics in which new conjectures are constructed by an interleaved process of model computation, model inspection, property conjecture and verification. And while for example the Theorema system [4] supports many of these activities, there are currently no systems available that provide, in an easy to use environment, the flexible combination of diverse reasoning systems in a plug-and-play fashion via a high level specification of experiments, despite some previous research in that direction [1, 3].

[8, 14] presents an integration of more than a dozen different reasoning systems—first order theorem provers, SAT solvers, SMT solvers, model generators, computer algebra, and machine learning systems—in a general bootstrapping algorithm to generate novel theorems in the specialised algebraic domain of quasigroups and loops. While the integration leads to provably correct results, the

integration itself was achieved in an ad-hoc manner, i.e., systems were combined and recombined in an experimental fashion with a set of custom-built bridges that not only perform syntax translations but also certain filtering functions.

In this paper we report on a rational reconstruction of an interesting subprocess of the bootstrapping algorithm, namely the generation of isotopy invariants (presented in Sec. 2), using the framework for trustable communication between mathematics systems that was put forth in [6]. It employs the concept of biform theories (Sec. 3) that enables the combined formalisation of the axiomatic and algorithmic theories behind the generation process. It turns out that it is surprisingly difficult to separate the syntactic, semantic, and algorithmic level of the current implementation. We present the formalisation in terms of the necessary semantic and syntactic concepts in Sec. 4 and of biform theories describing the actual computations in Sec. 5. The aim of this work is to expose the general principles behind the combination and communications of the single systems. It is currently only a purely theoretical reconstruction of the current implementation, and we do not have or even plan an implementation of the process in the framework of biform theories. However, the work should ultimately be used in the design of a flexible environment for experimental mathematics that enables a user to specify complex experiments on a high level without the need of detailed knowledge of the underlying logical relations and the particularities of the integrated systems.

Note that, as in [6], we abstract out the details of the idiosyncratic syntax of each of the systems. We use a uniform abstract syntax (in this case, we need 4 separate languages, each embedded in the other) for the specification. This allows us to abstract out the tedious engineering of transformations in and out of the actual systems. On the other hand, any transformation beyond trivial parsing and pretty-printing is explicitly specified.

The specification we present involves (at least) 3 levels of mathematical discourse: using the language of mathematics, we are specifying (syntactically) the semantics of a computer system which manipulates (the syntax of) mathematical theories, which are themselves inhabited by mathematical objects represented syntactically. Each level also possesses semantic models, which is ultimately what we want, but for computer manipulation, must be handled syntactically. Separating these languages cleanly is a difficult task – as the reader will soon witness.

2 Problem

The particular problem we are concerned with is the process of generating isotopy invariants for loops, which is part of the overall classification procedure presented in [14]. We give a brief, high level description of the procedure here. The more formal, mathematical details and definitions are presented in Sec. 4.

A loop is a quasigroup with identity, i.e., an algebraic structure $(L, \circ)$ satisfying the following two axioms: $\forall a, b.\ (\exists x. x \circ a = b) \wedge (\exists y. a \circ y = b)$ and $\forall x. x \circ e = e \circ x = x$. We say two loops $(L, \circ)$ and $(M, *)$ are *isotopic* to each other—or L is an *isotope* of M—if there are bijective mappings α, β, γ from

L to M such that, for all $x, y \in L$, $\alpha(x) * \beta(y) = \gamma(x \circ y)$ holds. A property P, that is preserved under the isotopy mapping (i.e., if P holds for L then it also holds for all its isotopes) is called an *isotopy invariant*. In our approach we generate universal identities, a particular type of isotopy invariants, presented by Falconer [9]. To find universal identities we have implemented the procedure presented in Fig 1.

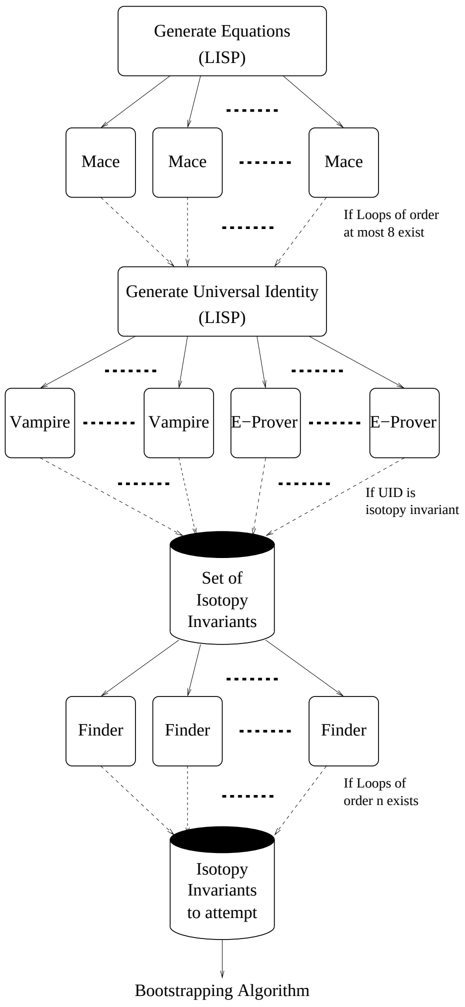

Fig. 1. Current implementation.

The basic idea of our procedure is to find identities (i.e., universally quantified equations) that hold for some loop, by first generating identities and then checking, which identity has a non-trivial loop satisfying it, using a model generator. All identities for which a loop exists are then transformed into derived identities (see Sec. 4 Def. 6). All derived identities for which we can prove, by means of a first order automated theorem prover, that they are invariant under isotopy are universal identities. Note that, for each universal identity, we show that it is an invariant under isotopy independently of the size of a loop. We can therefore reuse these universal identities in different classifications. Consequently, we collect universal identities in a pool of confirmed isotopy invariants, which we use in the overall bootstrapping algorithm. That is, during the classification of loops of a particular size n, we draw on the pool of invariants by first filtering them again using another model generator to generate loops of size n that satisfy the invariant. We then extract those invariants for which at least one loop of order n exists, and we use only these as potential discriminants. Note that the filter discards any invariants which cannot solve any discrimination problem, as no loop of size n satisfies the invariant property.

So far we have generated and verified $8,530$ isotopy invariants. These have been employed as one means, among others, to generate two novel classifications for loops of order 6 and 7 with respect to isomorphism. Observe, that we currently concentrate only on the subprocess for generating isotopy invariants and that the general bootstrapping algorithm is not the subject of the paper. For more details and results of the classification of quasigroups and loops with respect to both isomorphism and isotopism we refer the reader to [8, 14].

3 Biform Theories

The notion of a "biform theory" was introduced in [13] as the basis for FFMM, a Formal Framework for Managing Mathematics. Informally, a biform theory is simultaneously an axiomatic and an algorithmic theory. We present here a formulation of a biform theory that is simpler than the one given in [13].

General Logics A *general language* is a pair $L = (\mathcal{E}, \mathcal{F})$ where $\mathcal{E}$ is a set of syntactic entities called the *expressions* of L and $\mathcal{F} \subseteq \mathcal{E}$ is a set of expressions called the *formulas* of L. For example, if F is a first order language, then $L_F = (\mathcal{T} \cup \mathcal{F}, \mathcal{F})$ is a general language where $\mathcal{T}$ and $\mathcal{F}$ are the sets of terms and formulas of F, respectively. In the rest of this paper, let $L = (\mathcal{E}, \mathcal{F})$ be a general language.

A *general logic* is a set of general languages with a notion of logical consequence. In the rest of this paper, let $\mathbf{K}$ be a general logic. L is a *language* of $\mathbf{K}$ if it is one of the general languages of $\mathbf{K}$. If L is a language of $\mathbf{K}$ and $\Sigma \cup \{A\}$ is a set of formulas of L, then $\Sigma \models_{\mathbf{K}} A$ means A is a logical consequence of Σ in $\mathbf{K}$. For example, let $\mathbf{FOL}$ be a general logic representation of first order logic such that L is a language of $\mathbf{FOL}$ iff $L = L_F$ for some first order language F and $\Sigma \models_{\mathbf{FOL}} A$ means A is a logical consequence of Σ in first order logic.

An *axiomatic theory* in $\mathbf{K}$ is a pair $T = (L, \Gamma)$ where $L = (\mathcal{E}, \mathcal{F})$ is a language of $\mathbf{K}$ and $\Gamma \subseteq \mathcal{F}$. L is the *language* of T, and Γ is the set of *axioms* of T. A formula A of L is a *logical consequence* of T if $\Gamma \models_{\mathbf{K}} A$.

Transformers For $n \geq 0$, an n-ary *transformer in* L is a pair $\Pi = (\pi, \hat{\pi})$ where π is a symbol and $\hat{\pi}$ is an algorithm that implements a (possibly partial) function $f_{\hat{\pi}} : \mathcal{E}^n \rightarrow \mathcal{E}$. The symbol π serves as a name for the algorithm $\hat{\pi}$. There is no restriction on how the algorithm is presented. For example, it could be a a λ-expression in L or a program written in Lisp or Haskell (or even C or Java).

Let $\mathsf{dom}(\Pi)$ denote the domain of $\hat{\pi}$, i.e., the subset of $\mathcal{E}^n$ on which $f_{\hat{\pi}}$ is defined. Suppose $E_1, \ldots, E_n$ are expressions in $\mathcal{E}$. If $(E_1, \ldots, E_n) \in \mathsf{dom}(\Pi)$, the expression $\pi(E_1, \ldots, E_n)$ denotes the output of $\hat{\pi}$ when given $E_1, \ldots, E_n$ as input, i.e., it denotes $f_{\hat{\pi}}(E_1, \ldots, E_n) \in \mathcal{E}$ (and is thus *defined*). If $(E_1, \ldots, E_n) \notin \mathsf{dom}(\Pi)$, $\pi(E_1, \ldots, E_n)$ does not denote anything (and is thus *undefined*). The expression $\pi(E_1, \ldots, E_n)$ is not required to be in $\mathcal{E}$; it will usually be an expression of the metalanguage of L but not of L itself.

Example 1. Suppose $L_F = \{\mathcal{E}_F, \mathcal{F}_F\}$ is the general language corresponding to a first order language F. Let $\Pi = (\pi, \hat{\pi})$ be a unary transformer in L_F such that:

1. $\pi(E)$ is defined iff $E \in \mathcal{F}_F$.

2. If $\pi(A)$ is defined, it denotes a formula $B \in \mathcal{F}_F$ that is in prenex normal form and is logically equivalent to A.

That is, the algorithm $\hat{\pi}$ transforms any formula of L_F into a logically equivalent formula in prenex normal form. The expression $\pi(E)$ cannot be an expression in L_F (without some mechanism, such as Gödel numbering, for formalising the syntax of L_F in L_F itself). $\Box$

Example 2. Suppose $L_F = \{\mathcal{E}_F, \mathcal{F}_F\}$ is again the general language corresponding to a first order language F. Let $\mathit{\Pi} = (\pi, \hat{\pi})$ be a ternary transformer in L_F such that:

1. $\pi(E_1, E_2, E_3)$ is defined iff E_1 is a term of F, E_2 is a variable of F, and E_3 is a formula of F.
2. If $\pi(t, x, A)$ is defined, it denotes the result of simultaneously substituting t for each free occurrence of x in A.

That is, given t, x, A, the algorithm $\hat{\pi}$ transforms the formula A into the formula $A[x \mapsto t]$. Again the expression $\pi(E_1, E_2, E_3)$ cannot be an expression in $\mathcal{E}_F$. $\Box$

Example 3. Let **STT** be a general logic representation of simple type theory. Suppose $T = (L, \mathit{\Gamma})$ is an axiomatic theory of a complete ordered field in **STT** and that we have defined in T a type real of real numbers and the basic concepts of calculus such as limits, continuity, derivatives, etc. Let $\mathit{\Pi} = (\pi, \hat{\pi})$ be a unary transformer in L such that:

1. $\pi(E)$ is defined iff E is an expression of L of type $\mathsf{real} \rightarrow \mathsf{real}$.
2. If $\pi(E)$ is defined, it is an expression of L of type $\mathsf{real} \rightarrow \mathsf{real}$ that denotes the derivative of the function denoted by E.

That is, $\hat{\pi}$ is an algorithm that differentiates expressions that denote functions on the real numbers. $\Box$

An *algorithmic theory* is a pair $T = (L, \mathit{\Delta})$ where L is a general language and $\mathit{\Delta}$ is a set of transformers in L. L is called the *language* of T, and $\mathit{\Delta}$ is the set of *algorithms* of T. For more on transformers, see [12, 13].

Rules A *rule* in L is a pair $R = (\mathit{\Pi}, M)$ where:

1. $\mathit{\Pi} = (\pi, \hat{\pi})$ is an n-ary transformer in L.
2. M is a formula that uses π to relate the values of the inputs to $\hat{\pi}$ to the value of the output of $\hat{\pi}$.

The *transformer* of R, written $\mathsf{trans}(R)$, is $\mathit{\Pi}$, and the *meaning formula* of R, written $\mathsf{mean}(R)$, is M. The meaning formula M, which specifies the semantic relationship between the tuple of inputs and the output of the algorithm $\hat{\pi}$, will usually be an expression of the metalanguage of L but not of L itself. For each n-tuple $I = (E_1, \ldots, E_n)$ of inputs to $\hat{\pi}$, we assume that M reduces to a formula M_I of L which is called the *instance* of M with respect to I. An instance of M specifies the relationship between the values of a given tuple of input expressions and the value of the resulting output expression. Let $\mathsf{inst}(R)$ be the set of instances of M. M can often be conveniently expressed as a formula schema.

Example 4. Let $R = (\mathit{\Pi}, M)$ where:

1. $\mathit{\Pi} = (\pi, \hat{\pi})$ is the transformer in L_F given in Example 1.
2. M is the formula schema $A \equiv \pi(A)$ where A is a formula of L_F.

If A is the formula $p(c) \supset \forall x . q(x)$ (where c is a constant) and the result of applying $\hat{\pi}$ to (A) is $\forall x . p(c) \supset q(x)$, then $(p(c) \supset \forall x . q(x)) \equiv (\forall x . p(c) \supset q(x))$ is the instance of M with respect to (A). □

Example 5. Let $R = (\mathit{\Pi}, M)$ where:

1. $\mathit{\Pi} = (\pi, \hat{\pi})$ is the transformer in L_F given in Example 2.
2. M is the formula schema $(x = t \wedge A) \supset \pi(t, x, A)$ where t is a term, x is a variable, and A is a formula of L_F and t is free for x in A.

If t is a term, x is a variable, and A is $f(x, y) = g(x)$, then $(x = t \wedge f(x, y) = g(x)) \supset f(t, y) = g(t)$ is the instance of M with respect to (t, x, A). □

Example 6. Let $R = (\mathit{\Pi}, M)$ where:

1. $\mathit{\Pi} = (\pi, \hat{\pi})$ is the transformer in the language L of the theory T given in Example 3.
2. M is the formula schema $\mathsf{derivative}(E) = \pi(E)$ where E is of type $\mathsf{real} \rightarrow \mathsf{real}$. $\mathsf{derivative}$ is an expression of L of type $(\mathsf{real} \rightarrow \mathsf{real}) \rightarrow (\mathsf{real} \rightarrow \mathsf{real})$ that maps a function to its derivative. M thus asserts that the derivative of the function denoted by E is the function denoted by $\pi(E)$.

If E is $\lambda x : \mathsf{real} . x^2$, then $\mathsf{derivative}(\lambda x : \mathsf{real} . x^2) = (\lambda x : \mathsf{real} . 2 \cdot x)$ is the instance of M with respect to (E). □

For the sake of convenience, we will view a formula A of L as a (transformer-less) rule in L and assume that $\mathsf{trans}(A)$ is undefined, $\mathsf{mean}(A) = A$, and $\mathsf{inst}(A) = \{A\}$.

Biform Theories A *biform theory* in $\mathbf{K}$ is a pair $T = (L, \mathit{\Omega})$ where L is a language of $\mathbf{K}$ and $\mathit{\Omega}$ is a set of rules in L. ($\mathit{\Omega}$ may include formulas of L viewed as transformer-less rules.) L is the *language* of T, and $\mathit{\Omega}$ is the set of *axioms* of T.

T can be viewed as simultaneously both an *axiomatic theory* and an *algorithmic theory*. The *axiomatic theory of* T is the axiomatic theory $T_{\mathrm{axm}} = (L, \mathit{\Gamma})$ in $\mathbf{K}$ where $\mathit{\Gamma} = \bigcup_{R \in \mathit{\Omega}} \mathsf{inst}(R)$, while the *algorithmic theory of* T is the algorithmic theory $T_{\mathrm{alg}} = (L, \mathit{\Delta})$ where $\mathit{\Delta} = \{\mathsf{trans}(R) \mid R \in \mathit{\Omega} \text{ and } \mathsf{trans}(R) \text{ is defined}\}$.

The axioms of T—which are formulas and rules—are the background assumptions of T in an implicit form. The axioms of T_{axm}—which are formulas alone—are the background assumptions of T in an explicit form. A rule R in L is a *logical consequence* of T if, for all formulas $A \in \mathsf{inst}(R)$, A is a logical consequence of T_{axm}. Thus, the axioms of T are trivially logical consequences of T. Notice also that, since we are assuming that the formulas of L are rules in L, every logical consequence of T_{axm} is also a logical consequence of T.

4 Definitions

We now render the problem from Sec 2 precisely by giving the relevant formal definitions. To facilitate the formal specification as biform theories in Sec 5 we painstakingly distinguish between the semantics of the mathematical concepts, the languages necessary to express them, and the purely syntactic expression.

4.1 Semantic Concepts

Definition 1 *A* loop *is a non-empty set G together with a binary operation $\circ$ and a distinguished element $e \in G$ such that*

$$\forall a, b \in G.\ (\exists x \in G. x \circ a = b) \wedge (\exists y \in G. a \circ y = b) \quad \text{and} \quad \forall x \in G. x \circ e = e \circ x = x.$$

Definition 2 *Let G be a loop with binary operation $\circ$, then we can define two additional binary operations $/$ and $\backslash$ by*
1. $\forall x, y \in G. x \circ (x \backslash y) = y$ and $\forall x, y \in G. x \backslash (x \circ y) = y$
2. $\forall x, y \in G. (y/x) \circ x = y$ and $\forall x, y \in G. (y \circ x)/x = y$

Definition 3 *Let G, H be two loops with respective binary operation $\circ$ and $*$. We say G is* isotopic *to H if there are bijective mappings α, β, γ from G to H such that for all $x, y \in G$, $\alpha(x) * \beta(y) = \gamma(x \circ y)$ holds.*

Definition 4 *Let G be a loop and let P be a property that holds for G. We call P an* isotopy invariant *if P is preserved under the isotopy mapping (i.e., if P holds for G than it also holds for all its isotopes).*

Definition 5 *Let G be a loop with binary operation $\circ$. w is a word of G if it is a combination of elements of G with respect to the loop operation $\circ$. Let w_1, w_2 be two words in G, then $w_1 = w_2$ defines a* loop identity *if it holds for all elements of G.*

Definition 6 *Let G be a loop with binary operations $\circ, \backslash, /$. Given a word w in G, we define its corresponding* derived word *$\overline{w}$ by*
1. if $w = x, x \in G$, then $\overline{w} = x$;
2. if $w = u \circ v$, then $\overline{w} = (\overline{u}/y) \circ (z \backslash \overline{v})$, where $u, v, y, z \in G$.
Given a loop identity $w_1 = w_2$ of G, we define its corresponding derived identity *as $\overline{w_1} = \overline{w_2}$.*

Definition 7 *Let G be a loop with binary operations $\circ, \backslash, /$ and $\overline{w_1} = \overline{w_2}$ be a derived identity in G. Then $\overline{w_1} = \overline{w_2}$ is a* universal identity *if it is an isotopy invariant.*

4.2 Languages

In order to express the definitions of the syntactic concepts we give the necessary languages by stepwise extending the basic language of first order logic with equality (**FOL+EQ**). This will later enable us to define biform theories with minimal languages. Generally, we need a fair bit more machinery to define our various meaning functions; thus we will freely use Simple Type Theory (**STT**) [7, 10] as our general environment.

As a general typographical convention, we will underline all the symbols when we refer to the syntactic version of a symbol we already have in our semantics. We will not however underline variables, to ease (somewhat) the readability of the results. We assume that the reader is proficient enough in **FOL+EQ** and **STT** so that we do not need to repeat their syntactic definition here.

We first extend **FOL+EQ** to the language **Loop** by adding a binary function $\underline{\circ}$ and a constant $\underline{e}$. In a next step we add the two binary operations $\underline{\backslash}$ and $\underline{/}$ to **Loop** to obtain **Loop′**. In fact, in **Loop′**, we need *two* loops, so we in fact add $e_1, \circ_1, \backslash_1, /_1$ and $e_2, \circ_2, \backslash_2, /_2$ to **Loop′**. While these languages are sufficient to express the syntactic objects manipulated during the computation, we also need to express the meaning formulas for transformers in the language of a biform theory. This will necessarily be (at least) a second order logic, as we are quantifying over loops. It also is much easier to specify if we have access to a bit more machinery, such as lambda expressions and unique choice. We therefore use **STT**, as a superset of **Loop′**, for this purpose. Thus altogether we have the following sequence of languages: **FOL+EQ** $\subset$ **Loop** $\subset$ **Loop′** $\subset$ **STT**

4.3 Syntactic Concepts

Let $\mathcal{V}$ be a set of variables, with $x_i \in \mathcal{V}$. Let y, z be two new symbols not in $\mathcal{V}$ and let $\mathcal{V}' = \mathcal{V} \cup \{z, y\}$.

$\mathsf{Word} ::= \ x_i \mid \underline{e} \mid \mathsf{Word}\ \underline{\circ}\ \mathsf{Word}$
$\mathsf{Identity} ::= \underline{\forall} x.\mathsf{Word} \ \underline{=}\ \mathsf{Word}$
$\mathsf{Word}' ::= \ \ x_i \mid \underline{e} \mid (\mathsf{Word}'\ \underline{\backslash}\ y)\ \underline{\circ}\ (z\ \underline{/}\ \mathsf{Word}')$
$\mathsf{DerivedIdentity} ::= \underline{\forall} x.\mathsf{Word}' \ \underline{=}\ \mathsf{Word}'$

A Word is a word in the language of loops, composed of variables, an identity element $\underline{e}$ and an operation $- \ \underline{\circ}\ -$, where all variables of $\mathcal{V}$ are understood to be universally quantified. A Word' is a word in the extended language of loops, composed of variables, and identity element, operation and two new operators, $\underline{/}$ and $\underline{\backslash}$, where again variables ($\mathcal{V}'$) are universally quantified. Then $\mathsf{Identity}$ and $\mathsf{DerivedIdentity}$ are identities over the respective languages.

We also need syntactic representations of various axioms. For example, we have that

$\mathsf{CircAxm} \ \hat{=}\ \underline{\forall} a, b.\ (\underline{\exists} x.x\underline{\circ}a \ \underline{=}\ b)\underline{\wedge}(\underline{\exists} y.x\underline{\circ}y \ \underline{=}\ b)$ in **Loop**.
$\mathsf{IdAxm} \ \hat{=}\ (\underline{\forall} x.x\underline{\circ}e \ \underline{=}\ x)\underline{\wedge}(\underline{\forall} x.e\underline{\circ}x \ \underline{=}\ x)$ in **Loop**.
$\mathsf{DivLAxm} \ \hat{=}\ (\underline{\forall} x, y.x\underline{\circ}(x\ \underline{\backslash}\ y) \ \underline{=}\ y)\underline{\wedge}(\underline{\forall} x, y.x\ \underline{\backslash}\ (x\underline{\circ}y) \ \underline{=}\ y)$ in **Loop′**
$\mathsf{DivRAxm} \ \hat{=}\ (\underline{\forall} x, y.(y\ \underline{/}\ x)\underline{\circ}x \ \underline{=}\ y)\underline{\wedge}(\underline{\forall} x, y.(y\underline{\circ}x)\ \underline{/}\ x \ \underline{=}\ y)$ in **Loop′**.

These express respectively the axiom for $\circ$, the identity e, the left division $\backslash$ and the right division $/$.

Actually, to describe the full semantics, we need *two* copies of the above, for two different loops, whose components we'll naturally denote $(\underline{e_1}, \underline{\circ_1}, \underline{\backslash_1}, \underline{/_1})$ and $(\underline{e_2}, \underline{\circ_2}, \underline{\backslash_2}, \underline{/_2})$ respectively. Since we are in **STT**, we could have easily have written the above as functions from syntax to syntax, but that would have made the presentation too opaque. We also need to represent a *finite domain* syntactically. But this essentially amounts to creating n unique names.

We can then continue thus, for all the various concepts defined semantically in the previous section, which we use syntactically later (like bijective). We should also define the full syntax for a language of proofs (as the language for the output of one of our intermediate transformers below), but since the actual implementation ignores these proofs, it suffices to posit that this language exists.

5 Specification

We can view the generation of isotopy invariants from derived identities and their selection as possible discriminants for loops of a given size n as a sequence of single computational processes as displayed in Fig. 2. Each process accomplishes a different function in the overall computational process, e.g., is a source of equations, transforms expressions, or filters with respect to different criteria.

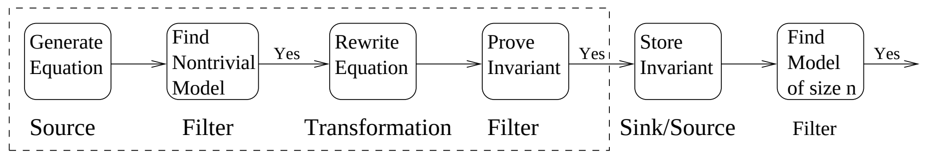

Fig. 2. Abstract view of the process.

For each module, we have a background (biform) theory that expresses the language and axioms necessary to describe the rules (and thus the inputs and outputs) encapsulated in that module. Using appropriate translations and interpretations, we can set up a communication channel (a connection in the language of [6]). We give detailed formal specifications for each process and their communications.

We can also give a specification for the global problem for generating universal identities in general, which corresponds to the dashed box in Fig. 2. This results in a transformer that operates on a more abstract level. We start by giving the formal specification for it, before going into the details of the component processes.

Concretely, for the remainder we let **K** be the general logic based on **STT** as general language. We then define the biform theory for the overall process consisting as axiomatic theory $T_{\mathrm{axm}} = (L, \Gamma)$ and algorithmic theory $T_{\mathrm{alg}} = (L, \Delta)$, where $L = \mathbf{STT}$, Γ contains all axioms defined in 4.1, and Δ contains one transformer $\Pi = (\pi, \hat{\pi})$.

1. $\pi() =$ `Generate_Universal_Identity`$()$ is always defined, as it takes no input. It returns a set of syntactical structures DerivedIdentity, as defined in Sec. 4.3

2. $\hat{\pi}$ is the implementation of `Generate_Universal_Identity` as presented in [14].
3. M is the meaning formula

$$
\begin{aligned}
&\forall U \in \mathtt{Generate\_Universal\_Identity}().U \in \mathsf{DerivedIdentity}\\
&\wedge Subst(\mathsf{CircAxm}, \underline{e_1}, \underline{\circ_1}, \underline{\backslash_1}, \underline{/_1}) \wedge Subst(\mathsf{IdAxm}, \underline{e_1}, \underline{\circ_1}, \underline{\backslash_1}, \underline{/_1})\\
&\wedge Subst(\mathsf{DivLAxm}, \underline{e_1}, \underline{\circ_1}, \underline{\backslash_1}, \underline{/_1}) \wedge Subst(\mathsf{DivRAxm}, \underline{e_1}, \underline{\circ_1}, \underline{\backslash_1}, \underline{/_1})\\
&\wedge Subst(\mathsf{CircAxm}, \underline{e_2}, \underline{\circ_2}, \underline{\backslash_2}, \underline{/_2}) \wedge Subst(\mathsf{IdAxm}, \underline{e_2}, \underline{\circ_2}, \underline{\backslash_2}, \underline{/_2})\\
&\wedge Subst(\mathsf{DivLAxm}, \underline{e_2}, \underline{\circ_2}, \underline{\backslash_2}, \underline{/_2}) \wedge Subst(\mathsf{DivRAxm}, \underline{e_2}, \underline{\circ_2}, \underline{\backslash_2}, \underline{/_2})\\
&\wedge Subst(U, \underline{e_1}, \underline{\circ_1}, \underline{\backslash_1}, \underline{/_1}) \wedge \exists \alpha, \beta, \gamma.\mathsf{bijective}(\alpha) \wedge \mathsf{bijective}(\beta) \wedge \mathsf{bijective}(\gamma)\\
&\wedge \forall x, y.\alpha(x)\underline{\circ_2}\beta(y) = \gamma(x\underline{\circ_1}y) \supset Subst(U, \underline{e_2}, \underline{\circ_2}, \underline{\backslash_2}, \underline{/_2}).
\end{aligned}
$$

In the formula M the predicate $Subst$ is shorthand for a syntactic replacement of the symbols $\underline{e}, \underline{\circ}, , \underline{\backslash}\ \underline{/}$ in U by their indexed counterpart to obtain two copies of loops. This can be implemented similar to the transformer for substitution in Example 2. However, we omit this level of detail here. Observe also that bijective is shorthand for the formulas representing the bijectivity property.

For the rest of this section, it is useful to remember that we use π for the *name* of the function (transformer) while $\hat{\pi}$ is its implementation. Whenever a transformer has a more natural name than π, we use that name instead.

5.1 Source: Generating Equation

The first step generates a set of identities. The language of the biform theory is $L = \mathbf{Loop}$ and the axiomatic theory is $T_{\rm axm} = (L, \{\})$. We do not need any axioms as the computation of this module is only a generation of constructs in our language and, therefore, not based on any logical consequences. In $T_{\rm alg} = (L, \Delta)$, Δ contains one transformer $\Pi = (\pi, \hat{\pi})$.

1. $\pi() =$ `Generate_Identity`$()$ is always defined, as it takes no input. It returns a set of syntactical structures Identity.
2. $\hat{\pi}$ is the implementation of `Generate_Identity` that corresponds to the grammar given for Identity in Sec. 4.3.
3. M is $\forall i \in \mathtt{Generate\_Identity}.i \in \mathsf{Identity}$.

Here M simply states that the elements of the generated set are indeed members of the syntactic class of identities.

5.2 Filter: Find Non-Trivial Model

The computation defined in this module is more complicated to formalise since we take a set of identities as input, for each one we try to generate a model of some specified size, and only output those identities for which we could successfully generate a model. Its formalisation can be achieved by using one transformer in the context of the other. We have a transformer modelling the general computation of a model generator that operates in the context of a transformer for the particular generation of models of loops satisfying given identities. In addition this latter transformer performs the actual filtering by discarding those identities for which no models exist.

The language of the biform theory is Loop and $T_{\text{axm}} = (L, \Gamma)$, where Γ contains the axioms given in definition 1. The algorithmic theory T_{alg} contains two transformers: $\Pi_1 = (\pi_1, \hat{\pi}_1)$ specifying the computation of the model generator and $\Pi_2 = (\pi_2, \hat{\pi}_2)$ specifying the filtering operation carried out on a set of identities.

1. $\pi_1(S, D) = \texttt{Generate\_Model}(S, D)$ is defined iff S is a set of syntactic formulas and D is a syntactic representation of the domain. It returns set of syntactic representation of models if one exists, i.e., for each constant element in the input a relation in D satisfying E.
2. $\hat{\pi}_1$ is the model generation process performed by some integrated model generator.
3. M is $\forall m \in \texttt{Generate\_Model}(S, D).(m \models S)$. Note that m is a model with interpretations of the symbols of S given as functions represented by sets of ordered pairs.

Given the transformer for general model generation, we now define the actual filter operation as:

1. $\pi_2(E, n) = \texttt{Filter\_Identities}(E, n)$ is defined iff E is a set of $\mathsf{Identity}$ n is a positive integer. It also returns a set of syntactical structures $\mathsf{Identity}$.
2. $\hat{\pi}_2$ is the implementation of `Filter_Identities` that generates the syntactic representation of the domain of size n given to π_1 and applies π_1 to each element in E.
3. M is $\forall i \in \texttt{Filter\_Identities}(E, n).\texttt{Generate\_Model}(\{i\} \cup \Gamma, \mathit{Dom}(n)) \neq \emptyset$.

Observe that in M above $\mathit{Dom}(n)$ is a schema specifying the set of domain elements computed in `Filter_Identities`, which could itself be specified using a transformer. Observe also that we have slightly simplified the formalisation, in that it only takes domains of one size, rather than a range of domain sizes.

We note that both transformers are very general and do not depend on the particular biform theory they live in. In particular `Generate_Model` works on any first order language regardless of the language of the biform theory. Here the theory serves as a way to parameterise the input to the transformer. `Filter_Identities` performs filtering with respect to model existence. Likewise it could filter with respect to non-existence of models or return computed models by interpreting them in the biform theory. Note that `Filter_Identities` is a total set-to-set function, and the underlying implementation is total as well; even `Generate_Model` is a total function (it always terminates), however it may return an *empty* result[1].

[1] In Haskell, we could say that `Generate_Model` belongs in the Maybe Monad.

5.3 Transformation: Rewrite Equations

This step rewrites loop identities to derived identities. The language of the biform theory is $L = \mathbf{Loop}'$ and the axiomatic theory is $T_{\mathrm{axm}} = (L, \Gamma)$ where Γ contains the axioms given in Definitions 1, 2, and 6. In $T_{\mathrm{alg}} = (L, \Delta)$, Δ contains one transformer $\Pi = (\pi, \hat{\pi})$.

1. $\pi(E) = \texttt{Generate\_Derived\_Identity}(E)$ is defined iff E is a set of $\mathsf{Identity}$. It returns a set of syntactical structures $\mathsf{DerivedIdentity}$.
2. $\hat{\pi}$ is the implementation of the rewrite system given in Def. 6 that performs syntactic rewriting of an $\mathsf{Identity}$ into a $\mathsf{DerivedIdentity}$.
3. M is $\forall i \in E. \exists! d \in \texttt{Generate\_Derived\_Identity}(E). \Gamma \models i \equiv d$.

5.4 Filter: Prove Invariant

Similar to the specification of the filter in Sec. 5.2 this process requires the combination of two transformers. One for the general proving process and one that performs the actual filtering.

We define the biform theory for $L = \mathbf{STT}$, with $\mathbf{Loop}'$ as a distinct sublanguage, $T_{\mathrm{axm}} = (L, \Gamma)$, where Γ contains all axioms defined in 4.1, and T_{alg} containing two transformers: $\Pi_1 = (\pi_1, \hat{\pi}_1)$ specifying the computation of the theorem prover and $\Pi_2 = (\pi_2, \hat{\pi}_2)$ specifying the filtering operation.

1. $\pi_1(A, C) = \texttt{Prove}(A, C)$ is defined iff A is a set of syntactic formulas in $\mathbf{Loop}'$ and C is a formula in $\mathbf{Loop}'$. It returns a syntactic representation of a proof if one exists.
2. $\hat{\pi}_2$ is the theorem proving process performed by some automated theorem prover. It takes the elements of A as assumptions and C as a conclusion.
3. M is $\mathit{Proves}(\texttt{Prove}(A, C), A \vdash C)$

Here the predicate Proves checks the correctness of the derivation. The formalisation of this predicate depends on the calculus of the integrated prover and is generally very lengthy to formalise. Since we are not interested in examining proofs but only their existence at this point, we do not go into any detail here.

Given the transformer for theorem proving, we now define the actual filter operation as:

1. $\pi_2(E) = \texttt{Filter\_Derived\_Identities}(E)$ is defined iff E is a set of expressions of the form $\mathsf{DerivedIdentity}$. It also returns a set of $\mathsf{DerivedIdentity}$.
2. $\hat{\pi}_2$ is the implementation of $\texttt{Filter\_Derived\_Identities}$ that generates an assumption set A from the syntactic representation of the elements of Γ computes for each element i of E $\texttt{Prove}(A, i)$. It returns a set of all elements in E for which a proof exists.
3. The meaning function M is of the same form as the meaning function of the $\texttt{Generate\_Universal\_Identity}$ transformer, with the exception that we now universally quantify over $U \in \texttt{Filter\_Derived\_Identities}(E)$.

Observe that the *Subst* is the symbol substitution transformer already used at the beginning of this section.

The result of this last transformer, the set of syntactic universal identities, can now be stored for later use. Choosing from this store is again achieved with a filter that uses model generation, which is similar to the transformer `Filter_Identities` above. Due to lack of space we omit the detailed formalisation of these processes here.

6 Conclusions

We have presented a first step towards a rational reconstruction of the classification procedure for finite algebras in [8, 14]. The use of biform theories enables us to express both axiomatic and algorithmic properties of the procedure while clearly distinguishing syntactic, semantic, and algorithmic levels. Although the generation of universal identities is only a small sub-process of the overall classification, its formalisation is surprisingly involved.

This is not really due to the design and current implementation of the algorithm, but rather because some of the operations necessarily intermix syntax and semantics. Keeping straight what has to be in **FOL+EQ**, what is safely in **STT**, and what is in fact in the meta-language is very difficult. For example when generating models with respect to a particular domain, the result is a semantic entity. Nevertheless, models have to be interpreted as syntax not only to express the meaning function but also in case the models are used in further computations and syntactic manipulations. This also has the effect that the given domain elements have to be incorporated into the language of the biform theory, which is currently not possible and subject to future work.

On the other hand, communication between the components is very simple, since it is all done via **FOL+EQ** or conservative extensions such as **Loop** and **Loop**$'$. The necessary interpretations [6] between these theories are straightforward, unlike the more general case where communication occurs amongst more disparate logics.

The current formalisation already exposes some generalisations. In particular many of the sub-processes can be expressed as a mixture of computation and filtering, where the computation is often independent of the particular theory. It also becomes apparent what is actual input and what has to be specified in the background theory of a process. This information could be exploited to design an environment enabling mathematical experimentation by combining systems on a high level, such that it is only necessary to specify input, output and parts of the background theory without interaction on the actual logical level.

Our formalisation is certainly not the only possible approach to reconstruct the generation process. Indeed we could view the entire process as a sequence of recursive generators and filters. E.g., the first three boxes in Fig. 2 could be combined in a single transformer that acts as a generator for the next filter. Comparing different specifications and combinations of transformers for the same process could expose possible optimisation opportunities for the overall process.

As we have already discussed, biform theories—particularly the meaning formulas of rules—are difficult to formalise in a traditional logic without the means to reason about syntax. The paper [11] illustrates how biform theories can be formalised in Chiron, a derivative of von-Neumann-Bernays-Gödel (NBG) set theory that directly supports reasoning about the syntax of expressions. After an implementation of Chiron is produced, we would like to use Chiron to fully formalize the work presented in this paper.

References

1. A. Armando and S. Ranise. From integrated reasoning specialists to “plug-and-play” reasoning components. In Calmet and Plaza [5], pages 42–54.
2. A. Armando and D. Zini. Towards Interoperable Mechanized Reasoning Systems: the Logic Broker Architecture. In *AI*IA Notizie Anno XIII (2000) vol. 3*, pages 70–75, Parma, Italy, 2000.
3. P. G. Bertoli, J. Calmet, F. Giunchiglia, and K. Homann. Specification and Integration of Theorem Provers and Computer Algebra Systems. In Calmet and Plaza [5], pages 94–106.
4. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4:470–504, 2006.
5. J. Calmet and J. Plaza, editors. *Proc. of AISC-98*, volume 1476 of *LNAI*. Springer Verlag, 1998.
6. J. Carette, W. M. Farmer, and J. Wajs. Trustable communication between mathematical systems. In *Proc. of Calculemus 2003*, pages 58–68, 2003.
7. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.
8. S. Colton, A. Meier, V. Sorge, and R. McCasland. Automatic generation of classification theorems for finite algebras. In *Proc. of IJCAR 2004*, volume 3097 of *LNAI*, pages 400–414. Springer Verlag, 2004.
9. E. Falconer. Isotopy Invariants in Quasigroups. *Transactions of the American Mathematical Society*, 151(2):511–526, 1970.
10. W. M. Farmer. STMM: A set theory for mechanized mathematics. *J. Autom. Reasoning*, 26(3):269–289, 2001.
11. W. M. Farmer. Biform theories in Chiron. In M. Kauers and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, LNAI. Springer, 2007 (this volume).
12. W. M. Farmer and M. von Mohrenschildt. Transformers for symbolic computation and formal deduction. In S. Colton, U. Martin, and V. Sorge, editors, *CADE-17 Workshop on the Role of Automated Deduction in Mathematics*, pages 36–45, 2000.
13. W. M. Farmer and M. von Mohrenschildt. An overview of a formal framework for managing mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.
14. V. Sorge, A. Meier, R. McCasland, and S. Colton. Automatic construction and verification of isotopy invariants. In *Proc. of IJCAR 2006*, volume 4130 of *LNAI*, pages 36–51. Springer Verlag, 2006.
15. J. Zimmer and M. Kohlhase. System Description: The MathWeb Software Bus for Distributed Mathematical Reasoning. In *Proc. of CADE-18*, 2002.