

Property based class hierarchy of Vaucanson's Algebra module

Alex Hamelin

Technical Report n°0905, May 2009
revision 2110

In VAUCANSON, Finite State Machines are mathematically defined by an algebraic structure module called Algebra. Considering the algebraic mathematical definitions, however, the current design is inaccurate: some hierarchical relations are false (for example, the inheritance between semirings and monoids). Moreover, we are unable to add new algebraic structures easily.

Therefore, in order to give ALGEBRA more granularity in its algebraic concept definitions, it is necessary to rework its current structure by introducing a property based class hierarchy similar to the one presented in SCOOP. Using the mathematical operator and set properties to define algebraic structures, as opposed to a usual class hierarchy, we would be able to specialize algorithms more precisely thanks to structure property verifications, thus increasing Vaucanson's performance and expressiveness.

Le module de structures algébriques de VAUCANSON, ALGEBRA, sert de base à la définition mathématique des automates finis. Cependant la modélisation actuelle est inexacte du point de vue théorique: les relations d'héritages entre certaines classes sont fausses (l'héritage entre les semi-anneaux et les monoïdes en est le parfait exemple). D'autre part, nous ne pouvons facilement l'étendre avec de nouvelles structures algébriques.

Ainsi, afin de doter ALGEBRA d'une plus grande granularité dans sa définition des concepts algébriques, il est nécessaire de retravailler sa structure globale en introduisant un système de hiérarchie par propriétés similaire à celui présenté dans SCOOP. En se basant sur les propriétés des opérateurs et des ensembles mathématiques pour définir la nature des structures algébriques, et non sur une hiérarchie de classes classique, nous pourrions nous permettre une spécialisation plus précise des algorithmes grâce à la garantie de propriétés sur ces structures, entraînant ainsi un gain de performance et d'expressivité important au cœur de VAUCANSON.

Keywords

ALGEBRA, SCOOP, STATIC, Desing pattern, Element, Typelist



Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22

hamelin@lrde.epita.fr – TEXT

Copying this document

Copyright © 2009 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Contents

Introduction	4
1 Mathematical Definitions of some Algebraic Concepts	5
1.1 Variety	5
1.2 Universal Properties	5
1.3 Monoid	6
1.4 Group	6
1.5 Semiring	6
1.6 Ring	7
1.7 Skew Field	7
1.8 Field	7
2 Re-Designing ALGEBRA	8
2.1 Analysis of ALGEBRA design	8
2.1.1 ALGEBRA's history	8
2.1.2 ALGEBRA current design	9
2.2 Designing an algebraic concept hierarchy	10
2.2.1 Consequences of bad inheritance relations	10
2.2.2 Loss of expressiveness due to misplaced property	11
2.2.3 Need for new structure capabilities	12
3 SCOOP 2 implementation of ALGEBRA	14
3.1 SCOOP 2's interesting features for ALGEBRA	14
3.1.1 Implicit inheritance	14
3.1.2 Morphers	15
3.2 Property based class hierarchy	16
3.2.1 Abstraction Hierarchy	16
3.2.2 Hierarchies' Bridge	18
3.2.3 Implementation Hierarchy	19
3.3 Use of the Element design pattern with SCOOP 2	21
Conclusion and future work	23
4 Bibliography	26
A Current ALGEBRA Class Hierarchy	27

Introduction

VAUCANSON is a finite state machine manipulation platform for automata and transducers (Sakarovitch, 2003). As these *machines* strongly rely on *algebraic structures*, the VAUCANSON module handling them, ALGEBRA, must be powerful enough to manage several kinds of algebraic structures.

Current ALGEBRA, however, is facing limitations because of design choices: first, its hierarchical relations are not mathematically relevant. This prevents us from expressing some polymorphisms or extending ALGEBRA easily. Second, we lack of code safety when manipulating algebraic structures. As a consequence, it prevents us from specializing algorithms.

In order to solve these problems, we dismantled ALGEBRA and created a stand-alone ALGEBRA using SCOOP 2, a Static C++ Oriented Object Program paradigm (Géraud and Levillain, 2008). Thanks to our modifications, ALGEBRA will be able to determine implicit inheritance links between concrete algebraic structures and their abstractions. As this paradigm is based on static property checking, it ensures at compile time that concrete structures fulfill a set of properties and implicitly links them to their appropriate abstractions. We will acquire both ease of extensiveness and code safety.

The outline of this report is as follow: we will first remind some essential mathematical definitions for this report to be understood. We will then study and correct the current design of ALGEBRA. We will present the SCOOP 2 based implementation of ALGEBRA and how it should be integrated in VAUCANSON. Finally, we will propose future improvements based on the upcoming version of ALGEBRA.

Acknowledgments

I would like to thank Sylvain Lombardy, Jacques Sakarovitch, the VAUCANSON Group and especially Jérôme Galtier who introduced me to VAUCANSON's arcans and spoiled me with ideas and advice. I would like to thank Alexandre Duret-Lutz, Damien Lefortier and Warren Seine for their comments on the first draft of this report.

Chapter 1

Mathematical Definitions of some Algebraic Concepts

As automata strongly rely on algebra, the library module which manages it needs to be generic and efficient enough to allow the library designers to express whatever they need. As algebra is one of the main subject of this report, we must define some algebraic concepts for the reader to fully understands its content.

1.1 Variety

Universal Algebra is a mathematical domain which studies *algebraic concepts* and their attached properties, *e.g.* the Monoid Concept, opposed to actual *algebraic structures*, *e.g.* the $(\mathbb{N}, +, 0)$ monoid. It is a way to study Algebra in a general way.

As part of Universal Algebra, we can distinguish **Varieties** of algebras, *i.e.* classes of algebraic concepts fulfilling a given set of axioms (*e.g.* commutativity, associativity, ...). Therefore, there exist the class of groups, rings, ... We can underline two general Varieties:

- The **Group-Like** variety containing Monoid, Group, Magma, ...
- The **Ring-Like** variety containing Semiring, Ring, Field, ...

1.2 Universal Properties

Universal Properties generalize definitions of mathematical properties over several kinds of mathematical objects. Those are parts of the Category Theory, *i.e.* a formalism dealing in an abstract way with mathematical structures.

In this report, we will consider only two properties which are essential to the algebras we consider. The following definitions are not the exact one as the needed formalism is not what we are interested in:

- **Free Objects** are collections of words formed from an alphabet, *i.e.* generators exist for the set we study. When dealing with automata, the most commonly used algebraic structure is the *free monoid of strings* constructed over a given *alphabet of letters*.

- **Abelian Objects** are algebraic structures whose operation is commutative. A binary operation \cdot on a set \mathbb{S} is commutative if: $\forall(a, b) \in \mathbb{S}^2, (a \cdot b) = (b \cdot a)$

1.3 Monoid

A **monoid** (\mathbb{M}, \cdot, e) is a set \mathbb{M} equipped with a binary associative operation \cdot and an identity element e . This binary operation fulfills the following **monoid axioms**:

- Associativity: $\forall(a, b, c) \in \mathbb{M}^3, (a \cdot b) \cdot c = a \cdot (b \cdot c)$
- Neutral Element: $\exists e \in \mathbb{M}, \forall a \in \mathbb{M}, e \cdot a = a \wedge a \cdot e = a$

Monoids are part of the *Group-like* variety.

Example: $(\mathbb{N}, +, 0)$ is a monoid.

1.4 Group

A **group** (\mathbb{G}, \cdot, e) is a set \mathbb{G} equipped with a binary associative operation \cdot and an identity element e . This binary operation fulfills the following **group axioms**:

- Associativity: $\forall(a, b, c) \in \mathbb{G}^3, (a \cdot b) \cdot c = a \cdot (b \cdot c)$
- Neutral Element: $\exists e \in \mathbb{G}, \forall a \in \mathbb{G}, e \cdot a = a \wedge a \cdot e = a$
- Inverse Element: $\forall a \in \mathbb{G}, \exists b \in \mathbb{G}, a \cdot b = b \cdot a = e$

A group is an actual monoid admitting an inverse element for \cdot .

Example: $(\mathbb{Z}, +, 0)$ is a group.

1.5 Semiring

A **semiring** $(\mathbb{S}, +, \times, 0, 1)$ is a set \mathbb{S} equipped with two binary associative operations, usually $+$ and \times that fulfill the following **semiring axioms**:

- $(\mathbb{S}, +, 0)$ is an **abelian monoid**.
- $(\mathbb{S}, \times, 1)$ is a **monoid**.
- Right and left distribution:
 - $\forall(a, b, c) \in \mathbb{S}^3, a \times (b + c) = a \times b + a \times c$
 - $\forall(a, b, c) \in \mathbb{S}^3, (b + c) \times a = b \times a + c \times a$
- Absorbing Element: $\forall a \in \mathbb{S}, 0 \times a = 0 \wedge a \times 0 = 0$

This algebraic structure is part of the *Ring-Like* variety of algebras. Therefore, it cannot be considered or used in the same way as groups or monoids, even if they are related. The definition given above use *Group-like* structures but these could easily be replaced by a set of axioms over $+$ and \times .

Example: $(\mathbb{N}, +, \times)$ is a semiring.

1.6 Ring

A **ring** $(\mathbb{S}, +, \times, 0, 1)$ is a set \mathbb{S} equipped with two binary associative operations, usually $+$ and \times that fulfill the following **ring axioms**:

- $(\mathbb{S}, +, 0)$ is an **abelian group**.
- $(\mathbb{S}, \times, 1)$ is a **monoid**.
- Right and left distribution:
 - $\forall (x, y, z) \in \mathbb{S}^3, x \times (y + z) = x \times y + x \times z$
 - $\forall (x, y, z) \in \mathbb{S}^3, (y + z) \times x = y \times x + z \times x$
- Absorbing Element: This property can be deduced from $(\mathbb{S}, +, 0)$ being a group (*i.e.* $+$ being invertible). From now on, we will consider this property admitted.

The only difference between a *Ring* and a *Semiring* is the invertibility of its first operator $+$.
Example: $(\mathbb{Z}, +, \times)$ is a ring.

1.7 Skew Field

A **skew field** $(\mathbb{S}, +, \times, 0, 1)$ also known as **division ring** is a set \mathbb{S} equipped with two binary associative operations, usually $+$ and \times that fulfill the following **skew field axioms**:

- $(\mathbb{S}, +, 0)$ is an **abelian group**.
- $(\mathbb{S} \setminus \{0\}, \times, 1)$ is a **group**.
- Right and left distribution:
 - $\forall (x, y, z) \in \mathbb{S}^3, x \times (y + z) = x \times y + x \times z$
 - $\forall (x, y, z) \in \mathbb{S}^3, (y + z) \times x = y \times x + z \times x$

Example: $(\mathbb{H}, +, \times)$ is a skew field (\mathbb{H} being the quaternion set).

1.8 Field

A **field** is a set \mathbb{F} equipped with two binary associative operations, usually $+$ and \times that fulfill the following **field axioms**:

- $(\mathbb{S}, +, 0)$ is an **abelian group**.
- $(\mathbb{S} \setminus \{0\}, \times, 1)$ is an **abelian group**.
- Right and left distribution:
 - $\forall (x, y, z) \in \mathbb{S}^3, x \times (y + z) = x \times y + x \times z$
 - $\forall (x, y, z) \in \mathbb{S}^3, (y + z) \times x = y \times x + z \times x$

The difference between a *Field* and a *Skew Field* is the commutativity of its first operator $+$.
Example: $(\mathbb{R}, +, \times)$ is a field.

Chapter 2

Re-Designing ALGEBRA

ALGEBRA is a VAUCANSON module created to manage all kind of computations over mathematical sets. Because of its current limitations, however, it does not allow us to express all the mathematical diversity of algebra we may encounter when dealing with automata.

Currently, ALGEBRA is composed of three main algebraic concepts: Monoids, Semiring and Series. Even if they are sufficient to VAUCANSON, we may take advantage of more concepts. For instance, Vivien Delmon integrated an automata reduction algorithm working on fields which may also be specialized to work on groups (Delmon, 2009). But, as these concepts are not part of the current ALGEBRA design, this specialization is impossible.

Moreover, we have no guaranty of property correctness over the structures we manipulate. For instance, automata reduction algorithm is working on a field but is implemented using a semiring. We could improve code safety if we were able to check axioms over algebraic structures, making the programmer's life easier when manipulating ALGEBRA.

Finally, ALGEBRA current design is not mathematically correct. Some inheritance relations are false, therefore adding new concepts to the hierarchy is unnecessarily difficult.

We have to correct ALGEBRA design in order to give VAUCANSON architects new possibilities and more flexibility when dealing with algebraic structures. First, we need to identify the goals of ALGEBRA first designers. Based on this analysis, we will then correct ALGEBRA and provide a new class hierarchy.

2.1 Analysis of ALGEBRA design

2.1.1 ALGEBRA's history

VAUCANSON is aimed at scientific users and as a generic library, it has to allow its architects to express their needs in the easiest possible way. But as we previously said, the current design of ALGEBRA (See [Appendix A](#)) lacks of flexibility and accuracy. We may consider its evolution and understand the problems they faced while first designing ALGEBRA.

A previous work on ALGEBRA added *polynomial series, i.e. series* in the current VAUCANSON design (Michaël Cadilhac, 2006). They were facing the appearance of several plug classes due to the use of the design pattern Element (Régis-Gianas and Poss, 2003). Therefore, they introduced SCOOP (Burrus et al., 2003) as a response to generic programming problematics and presented a new design proposals of ALGEBRA for the monoid family (See [Figure 2.1](#)).

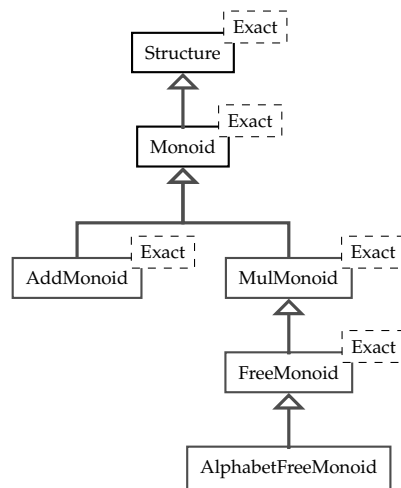


Figure 2.1: Proposal of an ALGEBRA design of the monoid family (Michaël Cadilhac, 2006).

We can see that two pseudo-classes appeared due to SCOOP and *Element*, **AddMonoid** and **MulMonoid**. Then at the moment, they were dealing with semiring integration into VAUCANSON and were clearly trying to split the both underlying monoids of a semiring. However, this method is not correct when trying to deal with several kind of algebraic structures. For example, let $(\mathbb{Z}, +, \times, 0, 1)$ and $(\mathbb{Z}, \min, +, -\infty, 0)$ two semirings. In the first case, $(\mathbb{Z}, +, 0)$ is an **AddMonoid**, but in the second case, it is a **MulMonoid**.

ALGEBRA evolved, acquired new capabilities (e.g. Free Monoid Product, Series, ...) which turned it into [Appendix A](#).

2.1.2 ALGEBRA current design

Someone can imagine many ways to design a hierarchy of algebraic structures. He may want to specialize monoids with different sub-types, define a semiring as an aggregation of two monoids or consider all classes as siblings and use adapters to simulate inheritance. But conceiving a design generic and powerful enough to express the mathematical diversity of algebras is really difficult. For us to understand the problems we are facing, we need to study ALGEBRA current design.

Current ALGEBRA can be divided in two parts (See [Figure 2.2](#)):

- The monoid family of algebraic concepts including **FreeMonoid**, ...
- The semiring family directly linked to **Monoid** through inheritance.

Considering this design, we can notice that ALGEBRA has two main problems:

- The first family of inheritance relations (*i.e.* monoid family) prevents us from expressing certain mathematical concepts easily. If we were to consider for instance Free Groups or other Free structures, we would have to introduce a concept redundancy inside ALGEBRA.
- The Semiring-Monoid relation is false for two reasons: first it is not mathematically relevant as monoids are not semirings, second it curbs our possibilities of **Algebraic Variety Polymorphisms**, as we cannot access both underlying monoids of a semiring with a single inheritance relation.

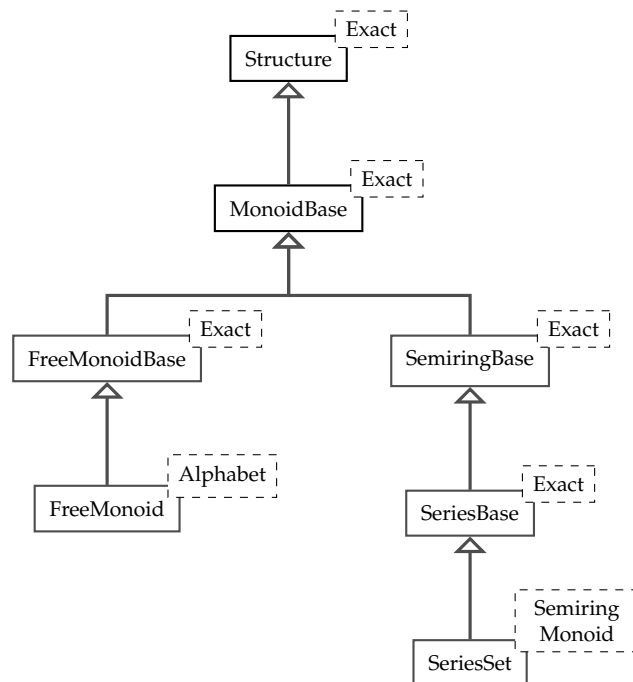


Figure 2.2: Current simplified ALGEBRA class hierarchy.

We will describe in the following sections how we constructed a proper abstract class hierarchy focusing on the previously described problems. Note that the inheritance between the *Structure* class and its children is used to group all the hierarchy under a common algebraic type.

2.2 Designing an algebraic concept hierarchy

2.2.1 Consequences of bad inheritance relations

Currently, *SemiringBase* inherits from *MonoidBase* while they both belongs to two distinguished varieties. Figure 2.3 illustrates the limitation we may encounter because of this choice:

```

// A semiring is composed of two underlying monoids. We here
// reproduce ALGEBRA's inheritance relation: a Semiring is a Monoid.
struct Monoid { /* ... */ };
struct Semiring : Monoid { /* ... */ };
5

// Here, the function foo only accepts as argument a monoid. One may,
// however, want to use a semiring (S,+,x,0,1) with foo by either
// considering the underlying monoids (S,+,0) or (S,x,1).
// Only using simple inheritance, it is impossible.
10 void foo(Monoid m) { /* ... */ }

```

Figure 2.3: One of our current design problem.

Let s be a semiring $(\mathbb{S}, +, \times, 0, 1)$. Because of the current type construction, it is impossible to extract both of the underlying monoids of s and consider them as valid arguments to function foo (i.e. $(\mathbb{S}, +, 0)$ and $(\mathbb{S}, \times, 1)$).

This **Algebraic Variety Polymorphism** was the first goal of ALGEBRA Architects as a semiring can clearly be used as a monoid. This single inheritance relation, however, does not allow us to access both of the underlying monoids.

So, we dismantled ALGEBRA by separating in distinguished branches monoids, semirings and series depending on their algebraic **variety** (See [Figure 2.4](#)).

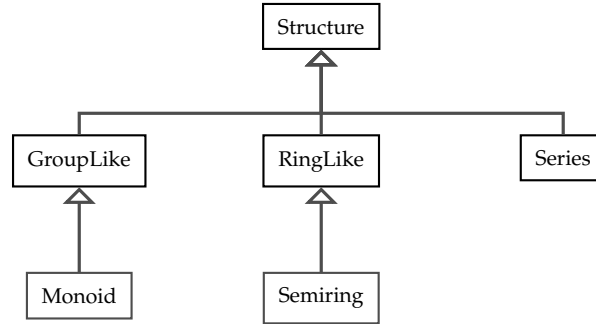


Figure 2.4: Algebraic Variety diagram.

This way we reestablish the mathematical accuracy of inheritance relations but we lose any kind of Variety Polymorphism. This will however be allowed using SCOOP's *morphers* (See [subsection 3.1.2](#)).

We can notice that we are working on *algebraic concepts* and not *algebraic structures*. For instance, we are not talking about the $(\mathbb{N}, +, 0)$ monoid but about the Monoid Variety.

2.2.2 Loss of expressiveness due to misplaced property

When dealing with category theory, we deal with universal properties. They are used to define abstract properties which can be applied to abstract objects.

In our case, we are only interested in defining properties over **Abstract Algebra** which is a generalized vision of classical algebra. It does not study concrete algebraic structures but more general concepts (like the group variety opposed to the $(\mathbb{Z}, +)$ group).

Talking about free monoids is in fact mixing the universal property *Freeness* with the abstract algebraic structure *Monoid*.

Thus we decided to separate the definition of *Freeness* from the abstract algebraic structures to avoid code redundancy and to be mathematically accurate (See [Figure 2.5](#)).

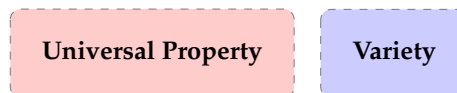


Figure 2.5: Universal Properties are separated from Algebraic Varieties.

As algebraic objects can be free **and** actual algebraic structures, we can note that multiple inheritance will be introduced into VAUCANSON. We may now add new universal properties

such as Abelianess which would be useful to describe more refined structures without being intrusive, thus avoiding code duplication. This modification allows a new kind of function specialization in VAUCANSON (See [Figure 2.6](#)).

```

struct Monoid { /* ... */};

struct Group { /* ... */};
5
struct FreeObject { /* ... */};

// ConcreteFreeMonoid is a concrete class which is both
// a FreeObject and a Monoid.
10 struct ConcreteFreeMonoid :
    Monoid,
    FreeObject
    { /* ... */};

// By using two times the same object as argument to an algorithm ,
// we will introduce a dispatch for multiple inheritance.
15 void
do_algorithm(FreeObject f, Monoid m) { /* ... */}

void
20 do_algorithm(FreeObject f, Group g) { /* ... */}

template<typename T>
void
25 algorithm(T obj)
{
    do_algorithm(obj, obj);
}

```

Figure 2.6: New possibility offered by separating universal properties from varieties.

2.2.3 Need for new structure capabilities

As exposed by Vivien Delmon in his work on automata reduction ([Delmon, 2009](#)), ALGEBRA lacks some algebraic concepts to specialize algorithm on more precise algebraic structures. Automata reduction is an algorithm working on fields or semirings with an division algorithm (e.g. $(\mathbb{R}, +, \times, 0, 1)$ and $(\mathbb{Z}, +, \times, 0, 1)$). As these concepts are not part of ALGEBRA, specialization is not possible. Moreover, this algorithm is currently being implemented as working on semirings, what could clearly be dangerous as the needed properties may not be fulfilled.

The simplest solution is to introduce new concepts in the design of ALGEBRA.

Thanks to all the modifications we applied to ALGEBRA, we now have a mathematically correct design with new algebraic concepts (See [Figure 2.7](#)).

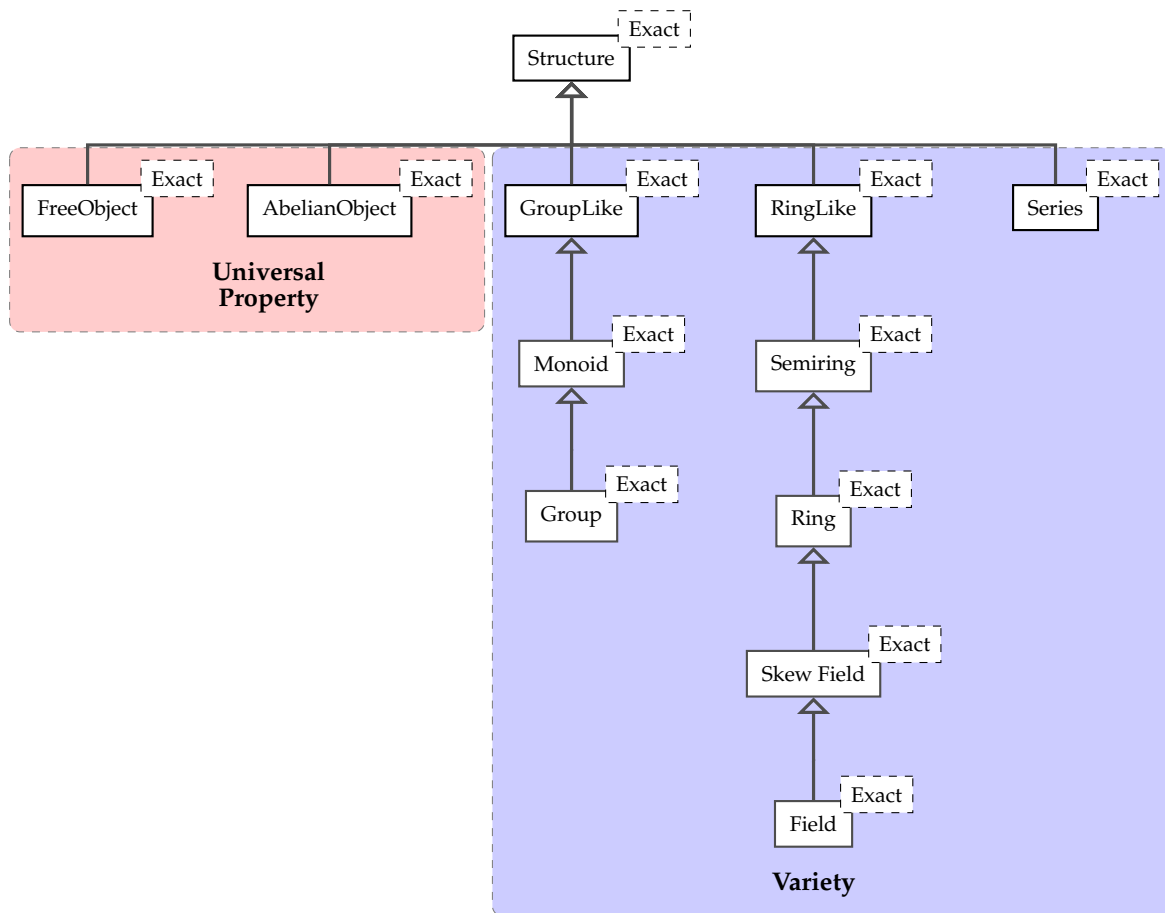


Figure 2.7: New ALGEBRA's Concept Hierarchy.

Two important problems remain and using traditional programming techniques, we cannot solve them while being efficient and generic:

- We have no check over properties algebraic structures may fulfill.
- We suppressed any kind of Algebraic Variety Polymorphism.

SCOOP 2 is a solution to these two problems.

Chapter 3

SCOOP 2 implementation of ALGEBRA

In the previous chapter we talked about ALGEBRA's needed modifications. Two problems however remained and for our new design to be usable, we need to solve them (See [subsection 2.2.3](#)).

SCOOP 2 ([Géraud and Levillain, 2008](#)) is a Static C++ Object Oriented Programming Paradigm created to allow static generic type transformations and implicit inheritance using type meta-data called properties.

Using SCOOP 2 features, we can solve both of the remaining problems without losing to performance. In the first part, we will briefly present SCOOP 2 and its possibilities. We will then describe the implementation of a SCOOP 2 based ALGEBRA. This stand-alone (*i.e.* not yet linked to VAUCANSON) does not implement any *morpher*. Finally, we will explain why the Element design pattern ([Régis-Gianas and Poss, 2003](#)) must be combined with SCOOP 2 in VAUCANSON.

3.1 SCOOP 2's interesting features for ALGEBRA

We have previously shown that two problems remained after our design changes. First, the necessity to have Variety Polymorphism, second, the need of static property checking for code safety.

SCOOP 2 is a paradigm using C++ created to express some kind of static class introspection to allow type transformations as a generic programming feature. This paradigm needs only a compiler complying the ISO/IEC C++ Standard to work and provides two libraries, STATIC and METALIC to make it easier to use.

METALIC supplies programmers a set of C++ metaprogramming tools (*e.g.* `static if`, `typedef lookup`, ...). It uses the `mlc` namespace.

Based on this library is constructed STATIC which provides a set of tools (*e.g.* macros, types, algorithms, ...) allowing an easier approach of SCOOP 2. It uses the `stc` namespace.

We will see the two main features that are interesting for ALGEBRA.

3.1.1 Implicit inheritance

Implicit inheritance is a way to compute through massive templating techniques the inheritance relations between a concrete class and its abstractions. Its functioning is based on **static prop-**

erty verifications. Given a set of *properties*, the way a type can be used, *i.e.* its interfaces, is computed at compile time. If a type fulfills a set of requirements needed by an interface, it will be linked to it.

A SCOOP 2 implicit hierarchy can be divided in three parts:

- A hierarchy of abstractions containing a set of interfaces for actual types. They specify a set of properties that must be fulfilled by their implementation. They hold no real information and could be compared to Java interfaces.
- A hierarchy of implementations containing concrete types which defines the actual methods, values, ... They specify a set of properties they fulfill and SCOOP 2 will automatically link them to their appropriate abstractions.
- A bridge between the two of them. It will act as a relation dispatcher using STATIC metalgorithms.

Using SCOOP 2, we will be able to, considering a set of property to be determined, link a concrete algebraic structure to its abstraction (link $(\mathbb{Z} \cup \{\infty\}, \min, +, -\infty, 0)$ to the semirings' abstraction). Using compile-time property checks, we will not only reduce the needed amount of work to add new algebraic structures because linking to abstractions will be automatic, but also guaranty property verifications which will allow safe algorithm specializations.

In our case, we can compare this process with mathematicians' proofs that a structure respect a given set of axioms. If a concrete algebraic structure fulfills the semiring axioms, then it is *proved* to be a semiring and can be used as one in VAUCANSON. The users of the library will be able to add new concrete types without entering VAUCANSON to understand the way it works.

3.1.2 Morphers

Morphers are the remedy to our last problem: low cost transformation of a type into other related ones (*e.g.* building from $(\mathbb{Z} \cup \{\infty\}, \min, +, -\infty, 0)$ both $(\mathbb{Z} \cup \{\infty\}, \min, -\infty)$ and $(\mathbb{Z} \cup \{\infty\}, +, 0)$).

They act as enriched adapters who can add or extract methods, change their interfaces, constructing types from several others, ...

Considering these features, morphers are a solution to our algebraic transformation problems. With their use, we give ALGEBRA the strength of type transformations, reducing code redundancy (See [Figure 3.1](#)).

```

typedef ConcreteZMinPlus<ZMin, ZPlus> z_min_plus_t;

z_min_plus_t structure1;

5 // We create here a monoid depending on the first substructure
  // of a more complex structure , a semiring for instance turned into
  // a monoid.
  casted_ringlike< z_min_plus_t, first_substruct > structure2(structure1);

```

Figure 3.1: Simple exemple of a Morpher use.

Given this set of tools and our design modifications, we can now implement a new version of ALGEBRA.

3.2 Property based class hierarchy

SCOOP ALGEBRA uses implicit inheritance in order to construct. Following the previously exposed requirements, we split our hierarchy in three parts:

- An Abstraction Hierarchy which defines our interfaces. It contains our algebraic concepts, *i.e.* Varieties and Universal Properties,
- An Implementation Hierarchy which contains actual data structures, *i.e.* $(\mathbb{N}, +, 0)$, $(\mathbb{Z}, *, 1)$, ..., and
- A bridge between the two hierarchies.

For each of these, we will expose the way ALGEBRA implement them.

3.2.1 Abstraction Hierarchy

First, based on what we explained in [section 2.1](#), we must distinguish two concepts in the abstract hierarchy:

- **Varieties** which represent general concepts of algebraic structures.
- **Universal Properties** which regroup a set of properties which can be applied to all varieties.

These two categories are orthogonal, *i.e.* they express unrelated requirements (*Freeness* and *Monoid* are two orthogonal concepts).

We must now establish the properties that will allow us to select the proper abstractions for a given concrete type. And as we are dealing with mathematical objects, we have formal definitions to help us. For instance, a monoid must fulfill a set of axioms: his operation must be associative and equipped with an identity element. If we enumerate all the axioms of [chapter 1](#), we obtain the set of properties for *GroupLike* and *Universal Properties* of [Figure 3.2](#).

```

// GroupLike properties
mlc_decl_typedef(op_has_unit);
mlc_decl_typedef(op_is_invertible);
mlc_decl_typedef(op_is_associative);

// Universal Properties
mlc_decl_typedef(op_is_commutative); // AbelianObject
mlc_decl_typedef(struct_is_free); // FreeObject

```

Figure 3.2: Properties used in ALGEBRA. `mlc_decl_typedef` is a macro defining types in METALIC scope.

We are also able to distinguish three concurrent concepts which are deduced from the variety of each abstract algebraic interface:

- **GroupLike** which contains Group and Monoid.
- **RingLike** which contains Semiring, Ring, Skew Field and Field.

- **Series** which is a variety on its own.

Based on these varieties, we can define four other properties to define more complex objects such as RingLike, and to compute inheritance relations between them (See [Figure 3.3](#)).

```

// Used to define to which variety is related a concrete class
mlc_decl_typedef( variety );

// Two of the RingLike properties are links to sub
5 // algebraic structures.
mlc_decl_typedef( first_substructure );
mlc_decl_typedef( second_substructure );
mlc_decl_typedef( op_mult_is_distributive );

```

Figure 3.3: RingLike properties.

Based on the definitions given in [chapter 1](#), we can easily deduce the hierarchy contained in [Figure 2.7](#) labeled by **Variety**. Each level of the hierarchy adds a method corresponding to the properties it fulfills (See [Figure 3.4](#)).

```

template<typename Exact>
struct Monoid : public virtual GroupLike<Exact>
{
// Implementation type defined by the
// linked concrete algebraic structure
// which will disappear thanks to Element.
typedef vcsn_type_of( Exact, value ) value_t;

// Retrives the unit of the operator
10 value_t
op_unit() const
{
return this->exact().impl_op_unit();
}
15 };

template<typename Exact>
struct Group : public virtual Monoid<Exact>
{
20 typedef vcsn_type_of( Exact, value ) value_t;

// Retrives the inverse of an Element
value_t
op_invert(const value_t& a) const
25 {
return this->exact().impl_op_invert(a);
}
};

```

Figure 3.4: Two linked Algebraic Varieties.

Dealing with **Universal Properties**, we must take into account a problem we encountered while implementing ALGEBRA's *Bridge* between *Abstractions* and *Implementations*. With our current design we need to create a class for each property (i.e. *FreeObject* and *AbelianObject*), but we also need to create opposite classes (i.e. *NonFreeObject* and *NonAbelianObject*) because of multiple inheritance issues (See [subsection 3.2.2](#)).

Also, *virtual inheritance* is here used to solve any problem of dreaded diamond, merging all the possible clones of the highest classes of our hierarchy (i.e. *stc::any* and *Structure*). The overhead of this inheritance is reduced thanks to the use of the Barton-Nackman trick ([John J. Barton, 1994](#)). As we directly cast an object to its correct concrete type, we do not have to pass through possible virtual tables to call its methods.

3.2.2 Hierarchies' Bridge

So as to link the Abstraction Hierarchy and the Implementation Hierarchy, we need an entry point from which every implementation class will inherit. It acts as a dispatcher using STATIC switch/case to choose the most suitable abstraction (See [Figure 3.5](#)).

```

5 // Case Tags
  struct switch_variety;
    struct switch_grouplike;
    struct switch_ringlike;
    struct switch_series;
  struct switch_freeness;
  struct switch_abelianness;

10 // Our dispatcher class, granting access for a concrete class
  // to its associated abstractions.
  template <typename Exact>
  struct structure_base :
    // Choose an associated abstraction
    public virtual switch_<switch_variety, Exact>::ret,

15    // Is or Is not a FreeObject
    public virtual switch_<switch_freeness, Exact>::ret,

    // Is or Is not an AbelianObject
20    public virtual switch_<switch_abelianness, Exact>::ret
  {
    structure_base()
    {
    }
25  };

```

Figure 3.5: Dispatcher class of ALGEBRA.

This structure is the key object of the static property check done by SCOOP 2. This is one of the reasons we wanted to use this paradigm. Thanks to this structure, we can guaranty that every properly constructed implementation class is linked only to its logical abstraction computing verifications over properties.

Note that the hierarchy may fail to compile if the same default type is given to several

switches. For instance, as there are default cases for static switches, if two abstraction were to be the same at the base of our implicit inheritance (*i.e.* *structure_base*), the compilation process would fail. Therefore, we need to introduce an opposite type for all the Universal Properties to prevent this from happening (*e.g.* *FreeObject* and *NonFreeObject*).

3.2.3 Implementation Hierarchy

Now that everything has been created, adding an implementation class to ALGEBRA is easy. A user does not have to enter ALGEBRA to add its own structures, he just has to create a new concrete type and to link it to *structure_base*.

The implementation is divided in two parts: first we need to define the properties our structure fulfills, second we have to implement its methods depending on the abstraction we will be linked to. [Figure 3.6](#) is an example implementing a concrete class of ALGEBRA.

```

// Implementation of ConcreteZPlusGroup
struct ConcreteZPlusGroup;

5 // Trait containing its super class.
template<>
struct super_trait_< ConcreteZPlusGroup >
{
    typedef structure_base< ConcreteZPlusGroup > ret;
10 };

// Specialization of vtypes, a trait holding
// the properties of our concrete class.
// Defining it this way is needed by STATIC.
15 template<>
struct vtypes< ConcreteZPlusGroup >
{
    typedef int value;

    typedef group_like_t variety;

    typedef mlc::bexpr_<true> op_has_unit;
    typedef mlc::bexpr_<true> op_is_invertible;

    typedef mlc::bexpr_<true> op_is_commutative;
25 };

struct ConcreteZPlusGroup : public structure_base< ConcreteZPlusGroup >
{
30     typedef ConcreteZPlusGroup self_t;
    typedef vcsn_type_of_(self_t, value) value_t;

    public:
    ConcreteZPlusGroup();
    ConcreteZPlusGroup(ConcreteZPlusGroup& a);

    bool
    impl_equal(const value_t& a, const value_t& b) const;

    value_t
    impl_op(const value_t& a, const value_t& b) const;

    value_t
    impl_op_invert(const value_t& a) const;
45     value_t
    impl_op_unit() const;
    private:
    const value_t unit_;
50 };

```

Figure 3.6: Example of a compatible concrete type with ALGEBRA, $(\mathbb{Z}, +, 0)$.

3.3 Use of the Element design pattern with SCOOP 2

Our design represents algebraic structures. Thus, using their **elements** is clearly not user-friendly (See [Figure 3.7](#)).

```

// We use the class ConcreteZPlusGroup for our example.
struct ConcreteZPlusGroup { /* ... */};

int
5 main()
  {
    ConcreteZPlusGroup z;

    // Returns -8
10 return z.invert(z.plus(3,5));
  }

```

Figure 3.7: Usage *as is* of SCOOP based ALGEBRA.

Even if we are manipulating a *Concrete Class* we are still working on an **abstract concept**.

When Olena first implemented SCOOP 2, they were manipulating concrete objects, *i.e.* Images. VAUCANSON on the other hand is using ALGEBRA as a way to express computation over automata and transducers. Thus, when we want to use mathematical objects as elements of algebraic structures, we must wrap them with some kind of adapter.

Element was first introduced in VAUCANSON to enable the orthogonal specialization of generic algorithms, depending on either an implementation or a concept while manipulating *elements* of a set represented by this concept ([Régis-Gianas and Poss, 2003](#)). Relying on the new design we presented, our stand-alone can't be used as is. Therefore, we need to adapt concrete types to communicate with Element to make the use of elements *natural*.

```

struct ConcreteZPlusGroup { /* ... */};

int
5 main()
  {
    Element<ConcreteZPlusGroup, int> x;
    Element<ConcreteZPlusGroup, int> y;

    x = 3;
    y = 5;
10 return x + y;
  }

```

Figure 3.8: Usage of SCOOP based ALGEBRA with Element.

[Figure 3.8](#) is an example of actual use interesting for two things. First, we easily manipulates elements thanks to Element. Second, we can specify other implementations for Concrete Classes (for instance, we could use for ConcreteZPlusGroup either *int* or a GMP type ([GNU Project, 2008](#))).

We now have to adapt *Concrete Classes* of ALGEBRA to manage the implementations of Element.

We must however keep in mind that *Concrete Classes* of ALGEBRA must be independent of the actual implementation of sets (e.g. *int*) or the use of Element would be useless.

Conclusion and future work

This overhaul of ALGEBRA was mainly focused on finding the most generic design VAUCANSON may need. So we implemented a SCOOP 2 based ALGEBRA working as a stand-alone module not linked to VAUCANSON. It contains more algebraic concepts than the current ALGEBRA and is more mathematically relevant. This will allow us to specialize algorithms more precisely, thus making the library more powerful.

Moreover, SCOOP 2 capabilities allow us to ensure code safety to VAUCANSON programmers and permit easy enrichment of new algebraic structures. The only drawback to be studied may be an impact on compilation time process.

We also demonstrated that *Element* was useful as we were not manipulating algebraic structures but their elements. A powerful wrapper over algebraic structures is required to fulfill our needs.

We now have to:

- Link Element and SCOOP ALGEBRA.
- Measure SCOOP 2 impact on VAUCANSON compilation process.

This work leads us to different possible enhancements we could bring to VAUCANSON.

Automated Concrete Type Construction

Andrei Alexandrescu introduced a process to construct types based on the use of typelists and multiple inheritance using recursive templates ([Alexandrescu, 2001](#)). A generic class B templated by a typelist TL inherits simultaneously from all the objects contained in TL .

This idea may lead us to a new way of constructing ALGEBRA's concrete types.

Lets consider a set of method containers carrying properties and a typelist constructed with these containers ([Figure 3.9](#)).

```

// Typelist definition. Used to carry types as potential elements
// of a class to be.
template<typename H, typename T>
5 struct TList
  {
    typedef H Head;
    typedef T Tail;
  };
10
// Represent an empty list necessary to construct
// one element only lists.
struct NullType;

15 // Operators carrying properties like is_commutative ,
// is_associative , ...
struct Plus { /* ... */ };

struct Mult { /* ... */ };
20

// A Typelist constructed using Plus and Mult
typedef TList<Plus, TList<Mult, NullType> > structure_t;

```

Figure 3.9: Typelist construction.

And now consider a *generic concrete type* A templated by this typelist. Thanks to Alexandrescu's method, we may be able to statically construct a concrete class only defining its operators. From this typelist we would extract properties, implementations and values.

Then adding new algebraic structures to ALGEBRA would only be a matter of typelist constructions or creating new operators and generic types if needed. Moreover, morpher creation may be simplified. We may just have to extract or add operators to typelists and use a generic concrete class.

```

// Operators carrying properties like op_is_commutative ,
// op_is_associative , ...
struct ZPlus { /* ... */ };
5
struct ZMult { /* ... */ };

// A Typelist representing the semiring (Z,+,x)
typedef TList<ZPlus, TList<ZMult, NullType> > z_plus_mult_t;
10

// We can retrieve its first monoid by deleting ZMult.
typedef Delete_<ZMult, z_plus_mult_t> first_monoid_t;

```

Figure 3.10: Retrieving the typelist of the first monoid of $(Z, +, \times, 0, 1)$.

BOOST MPL ([BOOST C++ Libraries, 2004](#)) and LOKI ([Alexandrescu, 2001](#)) both gives a typelist manipulation platform to do this kind of manipulations. We may however prefer a self-made implementation of typelists so as not to add more dependencies to VAUCANSON.

Of AUTOMATA needs for properties

Now that ALGEBRA is able to express a certain set of properties, would the same concept be useful to AUTOMATA, VAUCANSON module which manages *transducers* and *automata*?

If we were to rework AUTOMATA using SCOOP 2, we would have to completely rework the way algorithms work inside VAUCANSON to take advantage of such a modification. Currently, Automata contains two main classes, *Automata* and *Transducer*. We would have to identify ways to refined this hierarchy for this overhaul to be interesting. Then, we would have to study which algorithms need to be reworked to take into account this modification.

We may also consider compile time issues as *Automata* is the most manipulated structure in VAUCANSON and SCOOP 2 meta-algorithms are greedy when coming to compile time computation of templates. A solution may be the use of SCOOP 1.5, a *simplified* paradigm ([Seine, 2009](#)).

Chapter 4

Bibliography

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley.

Burrus, N., Duret-Lutz, A., Géraud, Th., Lesage, D., and Poss, R. (2003). A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, Anaheim, CA, USA.

Delmon, V. (2009). Automata reduction. CSI Seminar 0830, EPITA Research and Development Laboratory (LRDE).

Géraud, Th. and Levillain, R. (2008). A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'08)*, Paphos, Cyprus.

John J. Barton, L. R. N. (1994). *Scientific and Engineering C++: An Introduction With Advanced Techniques and Examples*. Addison Wesley.

Michaël Cadilhac, Robert Bigaignon, F. T. (2006). Remodeling vaucanson. CSI Seminar 0604, EPITA Research and Development Laboratory (LRDE).

Régis-Gianas, Y. and Poss, R. (2003). On orthogonal specialization in C++: dealing with efficiency and algebraic abstraction in Vaucanson. In Striegnitz, J. and Davis, K., editors, *Proceedings of the Parallel/High-performance Object-Oriented Scientific Computing (PSC; in conjunction with ECOOP)*, number FZJ-ZAM-IB-2003-09 in John von Neumann Institute for Computing (NIC), pages 71–82, Darmstadt, Germany. Also available here : <http://www.lrde.epita.fr/download/papers/poosc03-vaucanson.pdf>.

Sakarovitch, J. (2003). *Éléments de théorie des automates*. Vuibert informatique.

Seine, W. (2009). An implementation of the c++ container library with SCOOOL. CSI Seminar 0904, EPITA Research and Development Laboratory (LRDE).

BOOST C++ Libraries (2004). The BOOST MPL library. http://www.boost.org/doc/libs/1_39_0/libs/mpl/doc/index.html.

GNU Project (2008). GNU multiple precision. <http://gmpilib.org>.

Appendix A

Current ALGEBRA Class Hierarchy

