Domain specific language on automata: Design and Implementation problems

Loïc Fosse

Technical Report n^o0305, revision 0.20, 2nd November 2003

Even if *Domain Specific Languages* (DSL) are not well-known, they are quite used. Indeed, these languages, which are built especially for a domain or a problem, can be a powerful alternative to *General Purpose Languages* (GPL) for their domains. Our problem is to manipulate automata easily and quickly in order to test algorithms, whether the automaton is the goal or the tool of the algorithm. A DSL seems to be a good solution.

Two major points will be broached:

- First, DSL in general will be discussed. What are their weaknesses, their interests, why and when using one instead of standard languages like C++. A methodology for building such a language will be exposed, illustrated by experiences given by MAL, our DSL concerning automata.
- Next, MAL will be presented more precisely. Its syntax, its semantics were built in order to fit mathematical notations and way of thinking. It is one of the links between MAL and Vaucanson and other links will be shown. But design and implementation are two distinct parts, and if the language is a powerful one, its implementation is consequently harder. So difficulties will be exposed too.

Keywords
automata, language conception, domain specific languages (DSL), general purpose languages (GPL), domain

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
lrde@epita.fr – http://www.lrde.epita.fr

Contents

Chapter 1

Introduction

While automata are a mathematical concept, they are used in computer science for different purposes: text matching, transducers building... There are many libraries designed for manipulating automata easily, but in some cases one would like a more direct way to manipulate them.

We are looking for a tool which allows to write algorithms directly, without implementation concerns. If one wants to test an algorithm nowadays she must write it in a GPL[1] using an external library. This approach is indirect and linked to the language of the library and its implementation. It cannot provide us with the abstraction we are looking for.

An idea to solve this problem is to create a new language dedicated to automata. We wanted a language with a unique purpose: manipulating automata. Since the language domain is limited it is not necessary to build a fresh GPL. That is why we wanted to build a *Domain Specific Language*, or DSL, on automata, which will have a syntax near of the domain one, to make it easy to understand.

So we designed MAL[2], MAL for *Minimalist Automaton Language*. The purpose of MAL is to fill two gaps: it gives a language dedicated to automata, and a simple way to work on weighted automata. It was only designed for automata handling, so it has types according to the domain: semiring, series, etc. It gives easy ways to build objects used in automata theory like free monoids, or automata. For example, the correspondence between the set of transitions and the δ function can favorably be used to describe quickly an automaton. Since MAL is a DSL, it can perform checking according to its domain, and test the validity of an automaton described by the programmer or the possibility to build series on specific elements.

This report is divided in two parts. Chapter 2 and chapter 3 give an introduction to DSL in general. Chapter 4 and followings deal with MAL. Chapter 4 presents the domain of MAL and its basis: sets handling. Chapter 5 introduce complex structures, like monoid or series, all based on sets handling. Eventually, the chapter 6 give an example of use of MAL.

[1] General Purpose Language. Common languages are GPL: C++, Ocaml, Java, etc.

[2] MAL language is still in development, but a first interpreter will be available soon.

Chapter 2

Introduction to Domain Specific Languages

In this chapter a first definition of DSL is given. It is important to understand what DSL are, and what make them different from "standard languages". After giving a definition and their characteristics, we compare them to GPL, in order to know if DSL are a good solution for us.

2.1 A primary definition

> "A *Domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [Arie van Deursen, 2000]."

A *Domain Specific Language* (also called *Micro-language*, *Problem oriented language* or *Very high level language*) is a language which has a unique purpose: solving problems belonging to one domain. They are consequently very small and simple. Most DSL are more declarative than imperative. Indeed most of time only manipulated objects are described, not how to manipulate them. The execution of the program can be deduced from the context, from the domain. It may seem strange in our case since we want a language to write algorithms using automata, in fact there are parts of execution which can be deduced from our domain. One does not want to write how to build an automaton with a new state. She would rather write that she wants an automaton to which she adds a state.

An example can clarify that. *Make* can be viewed as a DSL dedicated to dependency management. When you write a Makefile you do not write how dependencies must be managed, you just describe them. Not only *Make* can be viewed as a DSL [Charles Consel, 1998]: TEX can be considered as a DSL for text writing, or *SQL* for database management.

Of course there are languages which were developed initially as DSL: *PLAN-P*, a language for programming application-specific protocols, or *GAL*, a *Graphic Adaptor Language* for programming video device drivers [Arie van Deursen, 2000].

2.2 Characteristics

It was above explained, a DSL is most of the time simple and small, because of the domain knowledge. It is declarative and must provide appropriate built-in abstractions and notations. An interest is to be able to write something as natural as possible.

A DSL does not need to be Turing complete. The domain knowledge can help finding the task to do, so this task can be inexpressible in the language. Or the domain is very strict and the

language does not need to be able to express everything. For example, if you consider *SQL* as a DSL, it is obvious you will not be able to do whatever you want. It seems difficult to write a factorial function using *SQL* only. It is possible, but the language was not designed for this purpose, and the resulting program is hard to read.

2.3 DSL specificities

Using a DSL offers lots of possibilities and facilities, which differentiate them from "standard" languages. Here are the main of them. Keep in mind that the purpose is that *a* DSL *might be used by a domain specialist, even if she is not a programmer.*

Notations

First of all, a DSL offers domain-specific notations. It is very important because that is what make a DSL interesting to use or not. For us it means the ability to describe an automaton like mathematics describe it.

Abstractions

A DSL must express things at the abstraction level of the domain. It is linked to the first point because it is notations that decide what can be expressed and how. And in the case of a DSL, one wants to work at the highest level of abstraction. It seems obvious, but it is difficult to apply this idea in a programming language which must take care of the computer "hidden" behind. More generally, the way of thinking, the philosophy of the language must be at the domain level.

Concision

Because the domain is known, a program written with a DSL is more concise. Expressing everything is not necessary: some points are implicit and can be found with few indications. For example, automaton transition sets can be described by a real set or by a function δ. One does not want different constructions for this two automaton definitions, nor having to explicit the conversion between the set and the function. This conversion is implicit and does not need to be written (c.f. 5.2).

Optimization

Based on the same idea of domain knowledge, optimizations can be made to increase performance of programs. Nevertheless this consideration is less important for us than the others since we are not looking for performance but for flexibility and expressivity. This point will further be broached.

Domain-specific checking

Domain-specific checking is an important point, maybe the most important for us. When someone writes an algorithm using a DSL, she expects checking in accordance with the domain.

A automaton described with:

$$States = \{1, 2, 3\} \tag{2.1}$$

$$Transitions = \{1 \rightarrow 2, 1 \rightarrow 4\} \tag{2.2}$$

is not valid, and this non-validity must be detected and pointed out. Generally speaking, a DSL should add constraints on manipulated objects, and these constraints should be checked after each action. It is one of the most important features provided by a DSL. It is an important point for us since our DSL is created to write and test quickly algorithms.

2.4 DSL weaknesses

Of course, DSL are not the ultimate solution and a programmer can turn toward more common GPL with a library. It can be the case if he does not want to invest in DSL design. Indeed, building a DSL can induce high cost of designing, implementing and maintaining. Moreover a new DSL is a new language and users must be educated. This education can be made easier by the syntax of the language, however an education is still necessary. For this points one can prefer to use a GPL, but that's loosing DSL syntax and checking system. Using a library forces to use the parent language syntax, and it is not suitable for our problem.

There is a compromise which permits a high abstraction and does not imply learning a new language: DSEL. A DSEL is a *Domain Specific Embedded Language*. It is in fact a subroutine library, but written in a functional language such as *Haskell*. According to Hudak, use of such a language allows to write the library in a declarative style [Backhouse, 2002]. So a DSEL is the union of the mother language and the library. It inherits of the host language features, so the library has no chance to evolve beyond its intended scope. And there is no cost of development and maintenance of a compiler or an interpreter.

But a DSEL does not provide neither domain specific optimizations nor error-checking, what was said very important for us.

2.5 Correct use of DSL

A DSL is adequate when a problem family must be solved, *i.e.* when a program family must be developed. A program family is a group of programs which work on the same domain, concern the same domain or are coded in the same way, using common constructs or principles. So before starting writing a new program, in order to know if a DSL should be a solution, one must ask oneself

> "Does the program to be developed address an isolated problem? Could it be a member of a future program family [DSL, 2001] ?"

If the response is *yes* then a DSL would be interesting to use.

Chapter 3

DSL design and implementation

Of course, the first step of DSL design is finding its domain and the scope of this domain. It is certainly the most important step of DSL creation. Neglecting this point can make the DSL go in a wrong way.

The second step, which will be discussed in the first section, is domain and family analysis. Next we will talk about DSL implementation, and finish this presentation with a discussion about DSL evolution.

3.1 Domain and family analysis

3.1.1 Domain analysis

To determine what must be in a DSL, the first idea is obviously to analyze the domain.

This approach has a big gap. Analyzing the domain permits to know commonalities between programs of this domain, so one can describe common structures and execution patterns, but it does not show differences that can appear between different programs of this domain. And these differences must be known to be able to offer generic structures (deduced from abstraction of differences) and tools to specify to which family the program belongs (possibility to specialize to a specific task, with several primitives). This is the reason why a family analysis is needed.

3.1.2 Family analysis

A program family is a set of programs which have commonalities, and concern the same domain. By studying a program family, one is able to find:

- Commonalities
- Variations (both concerning data and patterns)
- Objects manipulated
- Terminology

Commonalities

> "Commonalities are requirements or assumptions that are true for every member of the program family [Thibault, 1998]."

One reason to develop a DSL is reuse. By finding commonalities between programs of a family, one is able to build programming structures which can be very reusable:

- Language primitives. They are primitive types and operators. They are the basic elements of any program coded in the DSL, these elements are put together by the control structures.
- Control Structures. That is what permits to build huge constructs with the language primitives. They are common assemblies whose parts and use contexts may vary. It is the generalized notion of control structures found in GPL. In C the control structure `for` can be found, but in MAL one may want a control structure built in order to iterate on states: something like `on_each_state`.[1]
- Language execution model. It refers to the fixed part of the execution model. For example, in C the first function called is `main`. It is something not described in the program, it is inherent to the language. In *NewsClip*[2] one can find several entry points defined in the language called `begin`, `select` or `action`.

Variations

> "Variations define the boundary of the program family and each variation includes the range of values it varies over and the time at which the value is fixed [Thibault, 1998]."

Variations can be classified as data or behavior variations. To manipulate data variations, structures which allow these variations are needed. It is the same idea for behavior variations. So the language must provide operators and statements which permit to describe every behaviors when they are combined. Finally, the language should provide:

- Declaration constructs. The possibility of variations implies different possible values. Declaration constructs permit to give a name to these values, so to handle them.
- Predefined variables. They can be seen as global variables, which are defined by the language. Changing the value of one of these variables may change the behavior of the program.
- Primitive procedures. They are built-in procedures which imply variations in the program according to their parameters.
- Primitive parameters. They are basic values given as arguments to primitive procedures.

Objects

That's what is manipulated. A DSL was said to be at the same level of abstraction than the domain. The manipulated objects must be those used in the domain too. Used objects must have enough power to cover all possible variations. For our case automata must be generic enough to be defined on each wanted semiring.

Terminology

> "A dictionary of standard terms that are used in communications among developers [Thibault, 1998]."

Once the different abstractions of a language are defined, it could be a good idea to give them a name! Looking at the terminology permits to have familiar notations for the domain expert. In the case of MAL, we have tried to stick to mathematical notations.

[1] In fact, on_each_state does not exist in MAL but there is an equivalent.
[2] A language to specify filters for USENET News articles.

3.2 Possibilities to implement a DSL

Two possibilities are offered to us to implement the language: writing a compiler or writing an interpreter.

For our case, the latter has been selected. Even if a compiler produces more efficient programs, it is harder to develop. Compiling a program introduces an additional step before execution, whereas interpreter interprets and execute directly. As we explained before our first aim is not speed. An interpreter allows checking on each line, and is more flexible for testing, that's why our first future implementation of our language will be an interpreter, like many DSL.

3.3 Implementation orientation

Domain and family analysis make somebody "discover" commonalities, variations, objects and terminology. All of this is used to create the DSL, with its syntax, etc.

From the implementation point of view, it corresponds to the language's "interface", which is behind the abstract machine produced from objects and operations study. This abstract machine is the core of the language, it does the basic operations on basic objects, ignoring the "syntactic sugar".

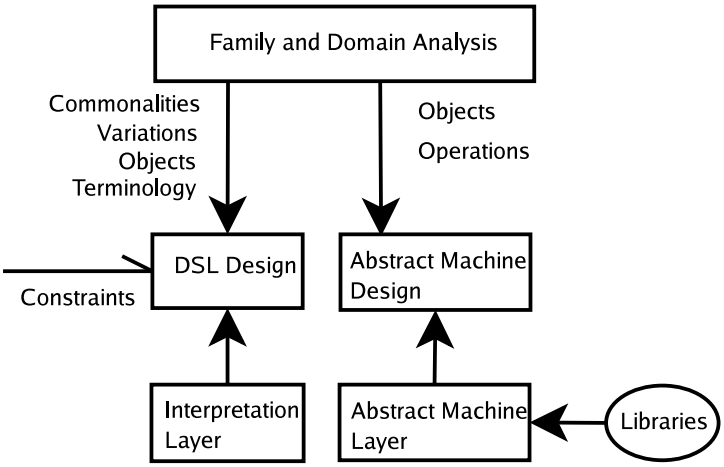

Figure 3.1: Global implementation structure

It induces two layers for implementing our interpreter [Thibault, 1998]. The interpret layer takes the language and gives simple objects and operations on them to the abstract machine, which finally executes what was asked. This structure for implementing a DSL is very common, because of its flexibility and reusability. It is also simpler to develop (Figure 3.1). Implementations of abstract machines can use specific libraries from the same domain. It can use several of them. Then the DSL becomes an "interface" between the different libraries and all programs which can be produced from them (Figure 3.2).

A remark concerning our choice about implementation: we decided to implement our DSL with an interpreter, which can be less efficient. By using lazy evaluation (on demand computation), this problem can be partially solved. It consists in not calculating anything until a result is asked. When a result is required, only what is needed is calculated. Such a method is massively used in Haskell, which is completely lazy. This will be indispensable for us if we want to manipulate "infinite" sets (this point will be exposed later).

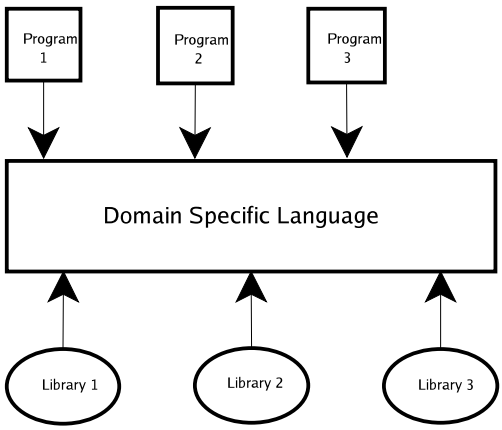

Figure 3.2: Interface between programs and libraries

3.4 DSL evolution

Lots of DSL which are a little popular have become GPL. Why? In fact, it is easy to understand: the more a DSL is popular, the more people are using it. And if there are many people using it, there will be many requests for new features, since a DSL does not implement all possible features. If developers answer to these requests, the DSL will move away from its original philosophy and will become a GPL.

It is not always a good thing, because the language may lose most of its characteristics: simplicity, concision, and it becomes more complex to maintain and develop. Performances can also decrease if the language go in a way which was not expected during its conception.

Chapter 4

Introduction to MAL: domain and language kernel

The main idea of MAL is of course automata manipulation. More precisely, it is weighted automata manipulation. Nowadays, there is not a lot of languages or libraries which deal with weighted automata, nothing exists to test quickly a new algorithm using weighted automata.

This chapter will present what we are looking for, what must be found in the syntax of MAL. It will describe the exact domain of MAL, and what features it provides. Basis of MAL, which consists in set handling, will then be exposed.

4.1 Domain of MAL

The purpose of MAL is

> Writing algorithms which manipulate automata easily and quickly, whether the automaton is labeled by letter or is weighted, in order to test them.

So we want the ability to write well-known algorithms quickly and to test written algorithms by looking at results or by measuring their speed. That induces a good "communication" with the programmer. A good "communication" is also needed concerning the checking system, in order to have functional algorithms quickly. Its syntax must be closely related to mathematical notations, to make MAL easily readable by a domain expert who is not necessarily a programmer.

In order to reach this purpose, it needs several features:

- High-level constructs
  - Sets (the basis of all automaton)
  - Algebraic structures (manipulated in algorithms concerning automata)
  - Automata
- Context
  - Algebraic context
- Execution directives

The context is the possibility to set which context the program is placed in. It is mainly a syntactic shortcut: it permits to specify the alphabet on which the automaton is defined, and this information have not to be rewritten. By specifying the context one can precise the alphabet, the semiring or the series set used in a program. Execution directives are used to describe what must be traced or timed in the execution of the algorithm.

It is mentioned above: sets are the basis of all automata. So sets are basis of the language. That is why sets handling is now exposed.

4.2 Foundations of MAL: sets handling

Manipulating automata means manipulating set. Indeed, an automaton is a sextuple describing states set, alphabet, transitions set, etc. So a many algorithms can be written in term of set manipulation.

An example is the pruned automaton. A pruned automaton is an automaton which only contains useful states, i.e. accessible and co-accessible states. Accessible states are those which can be reached from an initial state. Co-accessible are those from which a final state can be reached. In an set theoretic notation, it is written

$$accessible(A) = \{x \in states(A) / \exists y \in initial\_states(A), chain(x, y) \neq \emptyset\}$$

One of our purposes is to be able to write such an algorithm like below. We want to manipulate the states of an automaton as easy as possible. So we must be able to handle sets easily. It seems necessary to have a strong set theoretic base for our language. Moreover, the syntax for set theoretic manipulation must be the closest as possible to the mathematical one.

For this point, inspiration was found in Haskell notations. We want to be able to define a set by extension in a way near from the Haskell one:

```
[True, True, False]
[ 2 * n | n <- [2, 4, 7], isEven n, n > 3]
```

This Haskell code show how to write a list, but its syntax is very convenient for manipulating sets. The idea is now to generalize it for our purpose. The syntax produced permits to write sets by *extension* or by *intention*, i.e. one can describe the set by enumerating its elements, or by describing how to build it.

- By intention: `let X = {1, 2, 3}.`
- By extension: `let A = {elm <- X | elm % 2 = 1}.`

The second example must be read "let A the set of elm belonging to X as elm % 2 = 1".

We have introduced operators like `<-`. Here is a list of those used in MAL:

operators on sets

- Union ($\cup$): `\/`
- Exclusive union ($\sqcup$): `|_|`
- Intersection ($\cap$): `/\`
- Difference: `\`
- Cartesian product: `*`

operators on set elements

- Belongs to ($\in$): `<-`
- Does not belong to ($\notin$): `<+`

operator to describe a set

- By extension: `{...}`
- By intention: `{...|...}`

You may have noticed that a Cartesian product has also been introduced. It consequently introduces tuples. At this point of design, the syntax of tuples is known, using brackets and colons.

```
let T = (1, 'a', {1, 2}).
```

4.3 Type system

All of this is syntactic: there is no type consideration. Several solutions were studied to have a correct type system in MAL (c.f. appendix B.1), but the one chosen is a classic one. It is more natural for most people, simpler and more coherent. And as we do not want an error-prone language, as it was said earlier, a strong type system is needed.

4.3.1 Basic types

Four basic types can be found:

Integer relative integers: `1`, `42`, `51`, etc.

Boolean test values: `true` and `false`

Letter letter symbol: `'a'`, `'b'` or `'foo'`

Symbol a meta-type containing all: `'a'`, `1`, `true` or `{1, 2}`

Letter is the type corresponding to `string` in other languages. For MAL, the concept of *string* is useless[1] and the possibility of giving letter more complex name than `'a'`, `'b'` is pleasant. `Symbol` is a type which contains all elements of the language. There are few operations defined on it. It is used to allow writing

```
let X = {1, 2, 3}.
  value X: Set of Integer
let Y = {'v', 'c', 's', 'n'}.
  value Y: Set of Letter
let Z = X \/ Y.
  value Z: Set of Symbol
```

The meaning of writing this is to consider all like symbol, used as letter.[2]

There are also incremental types, which are types defined on other types:

Sets `Set of` *type*

Tuples *type1* `*` *type2*

We can now build a set, and having a correct type for it.

4.3.2 Type inference

We have seen in 4.3.1 how type inference is performed. Type inference is a key to have a language pleasant to write and to read. A type is found thanks to the construction of the object. If all elements of a set are `integers` then it is a set of integers. If there are elements of different types, the set will be a set of symbols.

Of course it is possible to specify the type if the system is not able to find it by itself or if one wants to restrict the use of a function. This is done using keyword `of`.

```
let (S of Symbol Set) = {1, 2}.
  value S: Set of Symbol
```

[1] Words are needed, but they are linked to free monoid structure and are described later.

[2] It will be remove from the language if type problems appears.

Chapter 5

Automata and Algebraic structures in MAL

In this chapter, we will look at more complex data types: automata and, more generally, algebraic structures. The way to build an automaton in MAL will be explained, and after having talked about control structures, the principle of algebraic structures handling will be shown.

5.1 Automata construction

A DSL on automata without automata constructs is incoherent. Nevertheless, automata are already possible in the current state of our presentation of the language.

Let an automaton be a sextuple [Sakarovitch, 2001],[Lombardy, 2001]:

$$\mathcal{A} = (\mathcal{Q}, A, \mathbb{K}, E, I, T)$$

where

$\mathcal{Q}$		set of states
A		alphabet
$\mathbb{K}$		semiring
E	$\subset \mathcal{Q} \times A \times \mathbb{K}^* \times \mathcal{Q}$	set of transitions
I	$\subset \mathcal{Q} \times \mathbb{K}^*$	set of initial states
T	$\subset \mathcal{Q} \times \mathbb{K}^*$	set of final states

So writing `(Q, A, K, E, I, T)` is describing an automaton. But we want distinction between sextuples and automata nevertheless: there is no checking done on tuple. That is why the notation <...> is introduced. It defines an automaton on the desired alphabet and semiring:

```
let A = <Q, A, K, E, I, T>.
```

This is in fact a sextuple, but with some constraints. These constraints can be applied after the event:

```
let N = (Q, A, K, E, I, T).
let A = <N>.
```

`A` is a automaton, and `N` a sextuple. Since the <...> notation only adds checking, `A` can be used anywhere `N` is used. It can be seen as a type hierarchy, where automaton is a child of tuples.

It also adds member access facilities. We can access states by writing

```
let S = AnAutomata.States.
  value S: Set of ...
```

It is the same with `Alphabet`, `Initial`, etc.

5.2 Functions and operators

To be able to write any algorithm of the automata domain, functions are needed. Recursive functions are also needed.

There is another idea in automata definition which has not yet been clearly exposed. An automaton is a sextuple containing a set of transitions $E \subset \mathbb{Q} \times A \times \mathbb{K}_* \times \mathbb{Q}$. But this set can be replaced by the δ function:

$$\delta : \mathbb{Q} \to A \times \mathbb{K}_* \times \mathbb{Q}$$

Our will is to use one instead of the other implicitly. So we should be able to write a value which is a function. This is done with the keyword `function`

```
(function x -> x + 1) 2.
  integer = 3
let test = function x -> x + 1.
  value test: Integer -> Integer
```

The isomorphism between functions and some kind of sets permits to write

```
let aset = {(0, 1), (1, 2), (2, 3)}.
  value aset: Set of Integer * Integer
let X = aset 1.
  value X: Set of Integer
```

where `X = {2}`. It returns a set: if `aset = {(0, 1), (0, 2), (2, 3)}`, asking for `aset 0` must return `{1, 2}`.

Now, let us apply this gor automata building. There are now two ways of defining an automaton. If we are working on letter automata:

```
let transitions_set = {(0, 1), (1, 2), (2, 3)}.
let transitions_fun = function x -> x + 1.
let automaton1 = <Q, A, transitions_set, I, T>.
let automaton2 = <Q, A, transitions_fun, I, T>.
```

We have said that there were some checking done on automata. For example all transitions must be on existing states. If it was applied here, the second writing would be wrong. The solution accepted now is to consider `transition_fun` as an infinite set and to restrict verifications when a set is not finite.

To deal with algebraic structures, operators are also needed. They are defined like functions with the keyword `operator`. To access the function corresponding to the operator, `[...]` was introduced. An example:

```
let + = operator x y -> x + y.
let u = 1 + 2.
let v = [+] 1 2.
```

5.3 Introducing imperative

Before broaching semirings or series, we will look at imperative structures in MAL. Lots of algorithms in automata theory are described in an imperative way. It is common to perform loops, variable assignments in algorithms concerning automata. That is why we need imperative structures. Three are introduced:

- The semicolon "`;`"
- `while`
- `for`

But having loops without variables is useless. So we decided to introduce variables in our language.

5.3.1 Variables

```
var X := {1, 2, 3}.
  variable X: Set of Integer
```

This defines a variable. Its type is decided at this moment. If it is not initialized, a programmer must precise the type, with the same syntax than in the general case. You can declare variable of any type, but once it is specified, it cannot change. Assignment is done with operator `:=`. An assignment has no type, and does not return anything.

Variables introduce side effects, that could seem strange since we are writing a language which is mostly functional, and which follow mathematical notations. But it is really essential since there are lots of algorithms concerning automata are written in a imperative way, with loops and variables.

5.3.2 Imperative structures

Semicolon

It is used to enumerate different actions to do. The type and value of a list of expressions separated by `;` are those of the last expression.

```
let x = (1; 2 + 3).
```

Here, `x` is an integer of value 5. In the future, a restriction could be to force expressions which are not ending the sequence to not return a value (*void* type).

While

The `while` looks like those one can find in other languages. The following expression is repeated until the argument of `while` becomes false.

```
let fct = function x ->
  var y = 0;
  var res = 0;
  while (y /= x)
  (
    res := res + y;
    y := y + 1
  );
  res.
  value fct: Integer -> Integer
```

For

The `for` keyword has not the same behavior in MAL than in GPL. If you want to a write something corresponding to the `for` of GPL, you must use a `while` loop, which is equivalent. In MAL, the `for` keyword permits to iterate on elements of a set, and evaluate the given expression for each element of this set. Its syntax is

```
for i <- E
  <expression>
```

But the order used to traverse the set is not known. Remember, we are handling true sets, then they are not sorted.

The order used to traverse can be specified:

```
  for i <- E with f
    <expression>
```

where `f` is an order function on `E`.

Here is a function which calculates the sum of the elements of a set.

```
let f = function x ->
  var res = 0;
  for i <- x
    res := res + i;
  res.
```

5.4 Dealing with algebraic structures

We now have tools to build algorithms, but we need structures on which algorithms are applied. The problem is to build *Monoid*, *Series*, etc. The free monoid on an alphabet containing `'a'` and `'b'` will produce objects like `""` or `"a.a.b"`. How to build such objects? What are their types?

In fact there is no *Monoid* or *Semiring* types in MAL, so no constructors, because these two structures are enriched `Set`, i.e. they are set to which operators and neutral elements are bound. To build such a set, just write:

```
let A = (X, +).
  value A: Set of ...
```

To build a Semiring (without distributivity information), write:

```
let A = (X, +:0 , *:1).
  value A: Set of ...
```

This sets can be handled like all other sets. But there is additional information on it, and then it can be used to specify the semiring of an automaton.

While *Monoid* and *Semiring* are represented by "simple" sets, *Free Monoids* and *Series* will insert new types: `Word on` *a type* and `Series`.

```
let A = {'a', 'b'}.
  value A: Set of Letter
let X = A*.
  value X: Set of Word over Letter
let Y = (A * {1, 2})*.
  value X: Set of Word over (Letter * Integer)
```

It builds a set of a new type: `Word`. An element of this set is a `Word`. In this set, you can find all the elements built with the Kleene Star: `""` (ϵ), `"a.b.a.a"`. It permits to declare a free monoid:[1]

```
let X = (A*, *).
  value X: Set of Word over Letter
```

Now let's construct a series:

```
let Astar = (X, *:Id).
let U = Set(Integer).
let K = (U, +:0, *:1)
let S = K << Astar >>.
  value S: Set of Series: (Word on Letter) -> Integer
```

[1] Syntax for operator used in Kleene Star is not yet defined.

The idea is the same: it builds a set of series, and we can work with its elements, or use it to define the context. Series will be very useful for the morphism that exists between them and automata. Nevertheless, the way of handling series is not syntactically specified.

When building structures such as series, there is some checking that is performed, as some checking is done when a tuple is declared as an automaton. That's why one must specify operators used in Monoid and Semiring. It informs about used operators. In a weighted automaton, operators on weights must be obligatorily known.

Other solutions were found to describe these structures, like the possibility to describe types in a recursive way. This possibility, which is the GPL's one, was dismissed for several reasons explained in appendix B.2.

5.5 Remarks

- The directives were not presented, but at this point of MAL development, they are not known. There will be two kinds of directives: one to specify context (alphabet, semiring), and another to tell the interpreter or the compiler what must be timed, or what must be traced.
- MAL handles infinite sets. That is why it will be implemented with lazy calculation. However, most of the infinite sets are present only for coherence, and for informations they give, not for what they contain.
- A example shows `Set(Integer)`. It builds the set of integers. It was exposed in 5.4, and was kept. It is an infinite set, but it exists only to be rigorous. At execution, it will not correspond to any action.
- The principal point which makes MAL a DSL is that building new types is impossible, and is not needed. New types are built with construct of Series and Free Monoid (with Kleene star).

Chapter 6

A use case of MAL

These algorithms are described with a specific context: they work on letter automata, and the alphabet have been specified by context directives. They are not syntactically fully correct (for transition writing), but the purpose of showing them is to give a first impression of what can be a program written in MAL.

6.1 Complete automaton

This algorithm completes an automaton. If a state has no outgoing transition on a letter, one is added, and go into a "well" state. The resulting automaton is the one given, with a new state, and with missing transitions added.

```
let complete = function A ->
  let NewStates = A.Sates |_| {'well'};
  <
    NewStates,
    A.Transitions \/
      {
        (x,a,y) <- NewStates * A.Alphabet * {'well'}
        | A.Delta(x, a) = {}
      },
    A.Initial,
    A.Final
  >.
```

6.2 Trim automaton

It defines the trim automaton, which is an automaton without useless transitions. A trim automaton is an automaton which have only accessible and co-accessible states.

```
let access = function A ->
  var reached = A.Inital;
  var old = {};
  while (s /= old)
  (
    old := reached;
    for i <- old
      for a <- A.Alphabet
        reached := reached \/ (A.Transitions i a)
```

```
  );
  reached.

// let coaccess = function A -> .....

let pruned = function A ->
  let SubStates = access(A) /\ coaccess(A);
  <
    SubStates,
    { (x,a,y) <- A.Transitions | x <- SubStates and y <- Substates},
    A.Initials /\ SubStates,
    A.Final /\ Subtsates
  >.
```

`access` is a function which return the set of accessible states. `coaccess`, the co-accessible function, is defined in the same way. The resulting automaton is the original with only accessible and co-accessible states. Transition set is redefined consequently.

Chapter 7

Conclusion

MAL is still in development, and its entire syntax is not defined. The handling syntax for rational expression is not decided, but this point will not be ignored a long time, because of the importance of rational expressions for automata. Syntax for directives and algebraic structures manipulation must also be fixed. And the type system is not definitive, and may change.

What is the future of MAL? First, all that was exposed in this report must be checked. Indeed, it seems coherent, but an incoherence can be still discovered. After this checking, and when all syntactic problems are solved, an interpreter will be written. Of course the language could continue to be developed: for example a system of modules could be inserted. The final purpose is offering an interpreter using libraries like *Vaucanson* which permits easy automata handling, and quick experiments on them.

Appendix A

Correspondences with VAUCANSON

A way of implementing such a language is to use libraries like *Vaucanson*. *Vaucanson* can be used *partially*, by using pre-constructed *Vaucanson* types, these types corresponding to types manipulated in MAL. For example:

```
let alphabet = {'a', 'b'}
let free_monoid = (A*, +).
```

would correspond to

```
using namespace small_alpha_letter;
Alphabet alphabet;
alphabet.insert('a');
alphabet.insert('b');
FreeMonoid<Alphabet> free_monoid(alphabet);
```

So we can use *Vaucanson* to implement types used in MAL.

The main problem of using *Vaucanson* is that all types which can be defined in MAL must have been instantiated in *Vaucanson*. It could be possible if new more flexible types are added to *Vaucanson*, but it will result in a big set of objects. Since MAL syntax is not yet defined, new things can appear, so thinking about using *Vaucanson* for implementation is maybe premature.

Appendix B

Some dismissed solutions

B.1 Dismissed type system

The first idea for type system was to use sets to define types. In mathematics, a type is a set. The element a is of type T if $a \in T$. A primitive set was defined, `atoms`, which is a set of `atom`, and which owns all the elements of the language. It can be compared to the `Object` class in Java. To build a new type, build a new set! For example

```
let Alphabet = {'a', 'b', 'c', 'd'}.
```

builds a new set named Alphabet, so a new type. If the type of `'a'` is asked, the system would respond it is an element of Alphabet. By writing

```
let Alpha2 = {'a', 'b'}.
```

one defines a subset of Alphabet. It build a hierarchy in types. `'a'` becomes an element of `Alpha2` and is still an element of `Alphabet`.

But this design has several serious problems:

- What is the type of the set? Is its type described by inclusion information? Or is it its own type? These informations are not enough to work on it, and it is not efficient.
- There is a problem of efficiency for type inference.
- There will be problem to describe new constructs like free monoid or semiring recursively described.[1]

B.2 Dismissed algebraic structures constructions

Recursive definition of sets

The idea was to allow the definition of sets in an inductive way. So the free monoid on `{'a', 'b'}` is the smallest set containing `"a"`, `"b"`, `""` and if x and y are in the free monoid, then x`.`y is in the set too. In other words, it is the smallest set containing `"a"`, `"b"`, `""` and which is stable by the `.` operator.

This idea was very interesting, because it allows to build anything one may want. Series and monoids are defined in this way and the only thing to do is to associate built set to operators. But the type of element of this sets is not defined. We are back to our first type problems!

[1] An idea was to extend the type system with algebraic structures, not only sets. Indeed, a type is not only a set, it is a set and operations defined on it. This matchs algebraic structures definition but such a system was to big too be used here. Remember: we are writing a DSL!

Recursive definition of types

The solution could be to define recursively types and not sets. But we need sets containing all elements from a type: defining a semiring on integers is to link set of integers with operators + and *. To be correct we must "convert" the type into a set, even if this conversion is only formal, and does not imply any action at execution.

In fact, with this system, types can be described as in Caml. But Caml is a GPL, and we are building a DSL! It is an interesting system, but it is too powerful. We do not need all this power! The chosen solution is giving constructors for all expected algebraic structures without offering the possibility to define new types for them.

Appendix C

Bibliography

[DSL, 2001] (2001). Domain specific languages - an overview. http://compose.labri.fr/doumentation/dsl/. IRISA/INRIA - LABRI.

[Arie van Deursen, 2000] Arie van Deursen, Paul Klint, J. V. (2000). Domain-specific languages: An annotated bibliography.

[Backhouse, 2002] Backhouse, K. (2002). D.phil thesis: Abstract interpretation od domain-specific embedded languages. Oxford University.

[Charles Consel, 1998] Charles Consel, R. M. (1998). Architecturing software using a methodology for language development. Technical report, IRISA / INRIA - University of Rennes 1, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France. http://www.irisa.fr/compose.

[Lombardy, 2001] Lombardy, S. (2001). Approche structurelle de quelques problèmes de la théorie des automates.

[Sakarovitch, 2001] Sakarovitch, J. (2001). éléments de théorie des automates.

[Thibault, 1998] Thibault, S. (1998). Domain-specific languages: Conception, implementation and application.