A shorter way between Vaucanson and its users

Louis-Noel Pouchet <louis-noel.pouchet@lrde.epita.fr>

LRDE seminar, June 23, 2004
http://vaucanson.lrde.epita.fr/

Copying this document

Introduction

The needs:

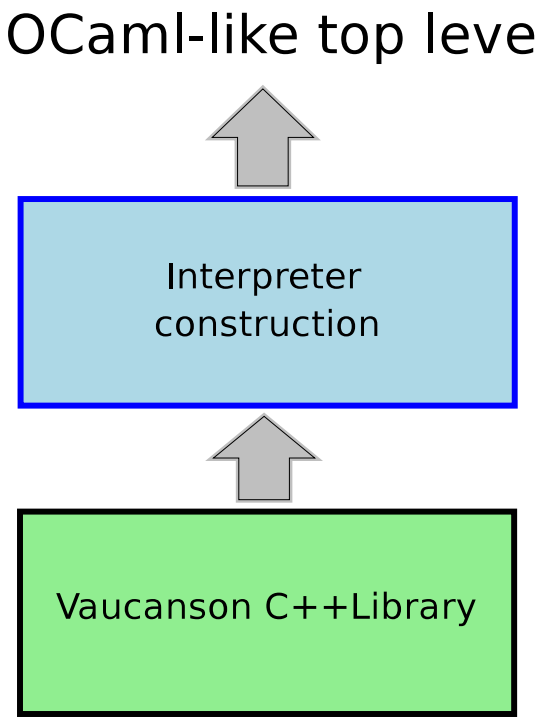
- get rid of annoying C++ syntax in Vaucanson[5],
- spare compilation time.

A solution:

- create an interpreter for Vaucanson,
- provide syntactic sugar for automaton manipulation.

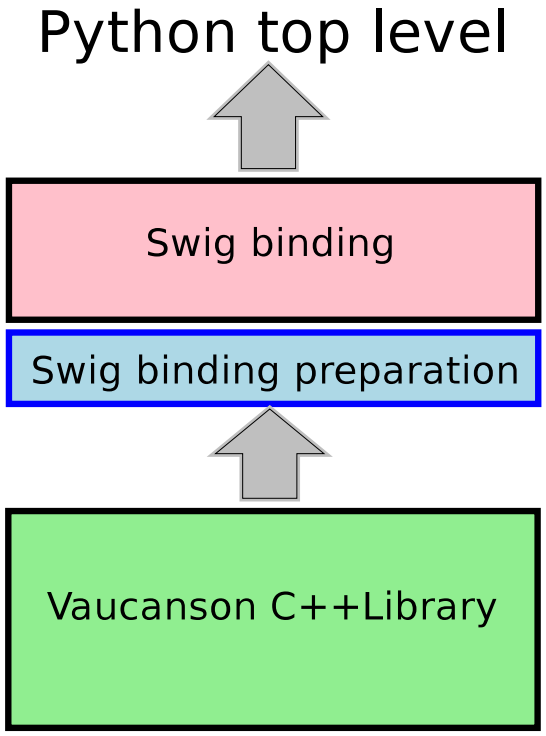
Swig-less interpreter

- Entire design of the interpreter,
- define its own syntax,
- need to instantiate all types.



Vaucanswig

- Based on Swig[3][1],
- Python syntax,
- need to instantiate all types.



This approach

- Based on Swig and OCaml,
- modified OCaml syntax,
- use generic type adapters.

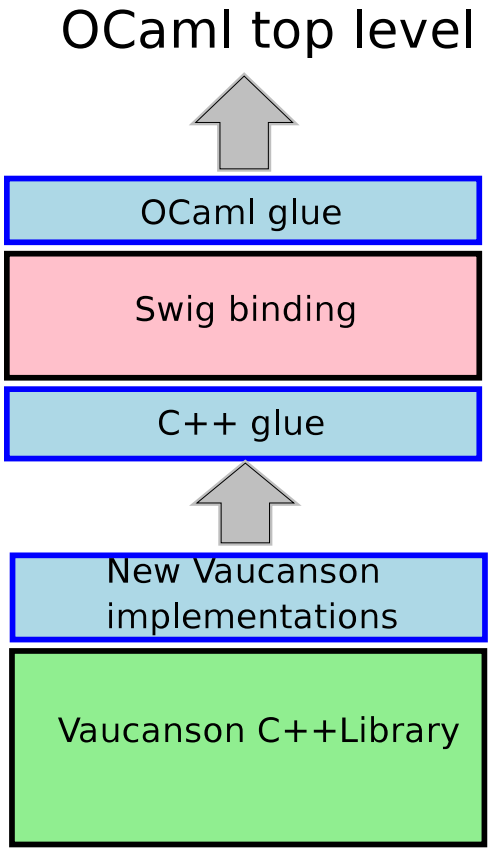


Table of Contents

A binding for Vaucanson

Our objectives:

- experiment algorithms in a dynamic environment,
- computations,
- fast and intuitive algorithm writing.

Using Swig

- Bind C / C++ code into various languages (Python, OCaml, PHP, . . .),
- actually only creates interfaces in destination languages,
- few parser limitations, that can be bypassed.

Bind to OCaml: why?

- Excellent interoperability with C / C++,
- interpreted and compiled,
- functional approach,
- known to be flexible,
- all the other advantages of the OCaml language. . .

Swig Vaucanson into OCaml [1/2]

- Define all the methods for each object,
- objects have a special type c_obj,
- can write object->method(arg1, arg2, . . .).

Swig Vaucanson into OCaml [2/2]

A code sample:

```
let alphabet = make_alphabet ("ab" to string);;
let automaton = new_boolean_automaton_t alphabet;;
let state1 = automaton->add_states();;
let state2 = automaton->add_states();;
automaton->add_letter_edge(state1,
                           state2,
                           (C_char 'a'));;
```

Extending the interface

Genericity for an automaton

Our purpose here is to get:

- various types of alphabets,
- various types of semirings.

$\Longrightarrow$ **Many kinds of multiplicity automaton can be generated.**

How we want to write it

- Use OCaml types,
- get genericity on input types (alphabets).

Typically, we would like to write:

```
let alphabet = ['a'; 'b'];;
let automaton = new_boolean_automaton_t alphabet;;

let alphabet2 = [('a', 1); ('b', 2)];;
let automaton2 = new_boolean_automaton_t alphabet2;;
```

Interoperability between C and OCaml

- OCaml closure call from C code,
- C code call in OCaml,
- accessors macros to internal structure,
- only one C type for every OCaml type: value (typedef for long).

The alphabet problem

Availability to use every letter type which can be defined in OCaml.

$\Longrightarrow$ **Provide a generic adapter for letters.**

- C++ letter class (one attribute containing the OCaml representation),
- OCaml method to print the letter.

$\Longrightarrow$ **A single entry point for the constructor: Letter(const value l)**

The semiring problem [1/2]

We would like to define semiring properties in OCaml and use them in Vaucanson.

$\Longrightarrow$ **Provide a generic adapter for numbers.**

- C++ number class (one attribute containing the OCaml representation),
- OCaml functions for all number operators.

The semiring problem [2/2]

To declare an operator function in OCaml (+ for Boolean semiring):

```
let add_ML a b = a || b;;
```

- No type specification, thanks to OCaml type inference,
- we only need to declare add, mul, identity, zero (and inf and eq for comparison) for general use of the semiring.
- like for letters, we need a printing function.

Some glue in C++

- New context in Vaucanson: `generic_automaton`,
- wrap some Vaucanson functions: explore the OCaml value to build the alphabet,
- wrap C++ function outputs to generate OCaml compliant values.

Now we have:

```
let alphabet = ['a'; 'b'];;
let automaton = new_automaton_t alphabet;;
```

The OCaml glue

- Provide some predefined semirings,
- handle properly the KRat expressions,
- provide an automaton constructor:

```
let automaton = build_automaton states semiring
                                alphabet edges
                                initials finals;;
```

Little summary

- Genericity on alphabets and semirings,
- OCaml types usable everywhere,
- wrappers for each wanted service,
- nice constructor for automata.

$\Longrightarrow$ **Let's extend the grammar!**

Grammar extension

- Extend syntax with Camlp4,
- files must be parsed with our grammar extension.

Automaton construction

Let an automaton be a six-tuple[4]:
automaton = $< Q, A, \mathbb{K}, E, I, T >$

$\Longrightarrow$ **We provide a similar constructor in the interpreter:**

```
let states = [0; 1; 2];;
let alphabet = ['a'; 'b'];;
let edges = [(0, 'a', 1); (1, 'b', 2)];;
let initials = [0];;
let finals = [2];;
let automaton = < states, boolean_semiring, alphabet,
                  edges, initials, finals >;;
```

Semiring construction [1/2]

Define a semiring by its calculus properties[2]:

```
let n_semiring = < (+):0, (*):1 >;;

let min a b = if a > b then b else a;;
let tropical_min_semiring = < min:infinity, (+):0 >;;
```

Notice that:

- you still need to define a `print_number` function and register it,
- type inference in OCaml permits to omit the type of numbers.

Semiring construction [2/2]

Support of the Kleene star:

```
let is_starable x = x == 0;;
let star x = 0;

let n_semiring = < (+):0, (*):1, is_starable, star >;;
```

Notice that:

- you may not need star functions,
- the interpreter tells you when star functions are needed.

Other extensions

Some extensions about algebraic structures are welcome:

- set union and intersection,
- element membership.

(Notice that sets are here implemented with OCaml list).

```
let sum = [0; 1; 2] |/ set2;;
let diff = set1 /| set2;;
let x = 4;;
let is_included = x <! set1;;
let not_included = x <+ set1;;
```

Review of our solution

What is good...

- Genericity over alphabets and semirings,
- syntax extension,
- fast compilation of the interpreter,
- code can be compiled / interpreted,
- it is implemented.

. . . and what is not

- Wilder automata are not supported,
- type checking is actually weak (loss of Vaucanson typing force),
- work with a subset of Vaucanson functionalities.

Conclusion

Future work:

- enforce type checking,
- add more sugar,
- add more services from Vaucanson,
- mechanize some processes.

References

[1] Swig 1.3 reference manual.

[2] Loic Fosse. Domain specific language on automata. Technical report, Epita Research and Development Laboratory, 2003.

[3] Raphael Poss. Liaison de bibliothèques a généricité statique avec un langage interprèté. Technical report, Epita Research and Development Laboratory, 2002.

[4] Jacques Sakarovitch. *Elements de théorie des automates*. 2003.

[5] Jacques Sakarovitch Sylvain Lombardy, Yann Regis-Gianas. Introducing vaucanson. 2004.

Questions