Incremental Extension of a Domain Specific Language Interpreter

Olivier Michel[1] and Jean-Louis Giavitto[1]

IBISC - FRE 2873 CNRS & Université d'Évry, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France

Abstract. We have developed an interpreter for the domain-specific language MGS using OCAML as the implementation language. In this third implementation of MGS, we wanted to provide the end-user with easy incremental addition of new data structures and their associated functions to the language. We detail in this paper our solution, in a functional setting, which is based on techniques similar to those found in *aspect-oriented programming.*

1 Introduction

This work takes place in the MGS [11, 16] project[1] which develops new data and control structures for the modelization and simulation of *dynamical systems with a dynamical structure* [14]. These features are embedded in a simple functional language, called also MGS, which is used to model various physical and biological processes [30, 31, 13].

The adequacy of MGS to its application domain is achieved through the following three features:

1. it embeds a very rich family of data structures used for the representation of the states of dynamical systems;
2. it provides a very large set of functions operating on these data structures;
3. it offers a new way of specifying uniformly functions defined by case on arbitrary data structures, using topological rewriting [12].

An interpreter for the MGS language has been implemented in the OCAML [21, 28] language. The decisive advantages of OCAML for us were that (1) it provides both functional and object-oriented features in the same environment and (2) it produces very effective code [1, 2].

One of the main problems raised by the MGS project is the wish to offer easy incremental addition of new data structures and their associated functions to answer the needs expressed by the end-users. As a matter of fact, the initial release of the interpreter did only include the collection data types *sequences*, *sets* and *multisets.*
The current interpreter includes *arbitrary graphs*, *Voronoï tessalation*, *group*

[1] The MGS project is available at: `http://mgs.ibisc.univ-evry.fr`

```
type expr =
    Constant of value
  | Apply of expr * expr

and value =
    Int of int
  | Fun of (value -> value)

let print_val = function
  | Int i -> Printf.printf "%d\n" i
  | Fun x -> Printf.printf "<fun>\n"

let inc_val =
  Fun(function (Int i) -> Int (i + 1)
     |  _      -> failwith "bad arg")
```

```
let rec eval = function
  | Constant x     -> x
  | Apply (e1, e2) ->
      (match (eval e1) with
       |   Fun x -> x (eval e2)
       | _       -> failwith "apply: type error")

let inc_expr = Constant inc_val
let inc = Apply(inc_expr,
                Apply(inc_expr, Constant (Int 1)))

print_val (eval(inc))
```

Fig. 1. A simple and basic interpreter expressed in a *higher-order syntax* style in ML.

based fields [11] which generalize various kind of arrays, *gmaps* [22], *extensible records* and *maps*, *trees* defined by automata, and many other data types [29]. All the additional data structures (together with their operators) have been added incrementally using the techniques described in this paper.

Usually, the values handled in the *target language* (that is, the language to be implemented, here, MGS), are represented through a unified data structure in the *implementation language* (that is the language used to implement the target language, here, OCAML). We call this data structure the *value* data structure. Using OCAML as the implementation language, there are two choices for the *value* data structure:

1. it can be represented using a *sum type*, following a functional style,
2. or, it can be represented using a *class* following an object-oriented style.

Both approaches have some shortcomings, with respect to the requirement of incremental development. To summarize

1. in the functional approach, it is easy to add new functions but difficult to add new target data structures;
2. on the contrary, in the object-oriented approach, it is easy to add new target data structures but difficult to add new functions.

To overcome these drawbacks, we have developed an original technique, inspired from aspect programming techniques, that consists in weaving both the *value* data structure and their associated functions. This technique has the advantages of:

- allowing new target data structures to be added without modifying the already written implementation files of the interpreter,
- facilitating the addition of new target data structures and functions to the point that even end-users are able to increment the MGS interpreter.

The rest of the paper is organized as follows. We briefly describe the MGS language in the next section to give the reader an idea of the complexity raised by the implementation of the rich data types in the interpreter. Section 3 describes the functional and the object-oriented approach used to implement the *value* data structure and details the problem raised by its incremental evolution. The implementation of heavily overloaded target functions are presented in the next section. The software architecture of the final implementation code of the interpreter is sketched in section 5. Section 6 presents how the informations gathered along all implementation files are collected to generate the *value* data type and to implement the multiple dispatch of the target functions. The conclusion summarizes our approach and shortly reviews related works.

2 Functions and Values in the MGS Programming Language

We briefly discuss in this section the values manipulated in the MGS language and their associated functions. Our aim is to show that the technique presented in this paper is required to deal with its complexity and to allow an easy incremental addition of new data structures and their associated functions.

2.1 The Type Hierarchy of the MGS Programming Language

We briefly give in this section an incomplete description of the type hierarchy of the MGS programming language.

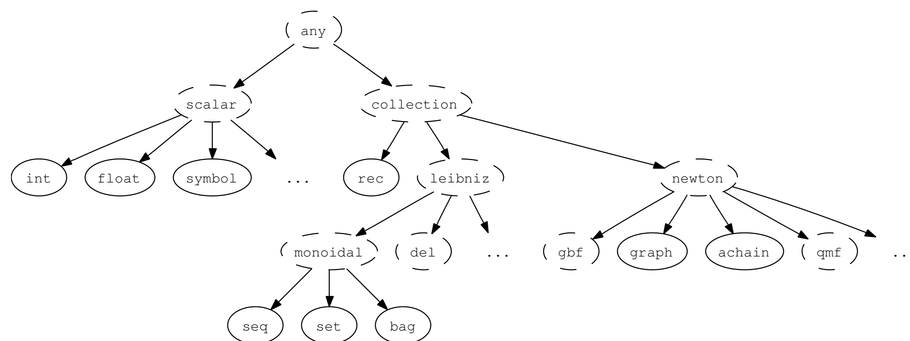

Fig. 2. The type hierarchy of the MGS language.

A graphical representation of the type hierarchy of MGS is given in figure 2. In MGS two main types of values are distinguished: the *scalar values* which are elementary constants and *collections* which allow to organize the values. Example of scalar values are integers, floats, symbols... Example of collection types are

sets, bag, Delaunay graphs, group-based fields [15], quasi-manifolds [22, 23]... Collection values can be any combination of collections and scalar values such as a bag containing symbols and sequences of integers.

In the following example, we define three values equal to collections: `v_seq` which consists in the *sequence* (like a `C` one-dimensional array) composed of a string value (`"str"`), a floating-point value (`3.5`), two integers values (`4, 4`), a boolean value (`true`) and the identity function (expressed as an anonymous lambda-calculus expression: `\x.x`) and the same elements organized as a *set* (`v_set`) and a *bag* (`bag`).

```
1   mgs> v_seq := "str", 3.5, 4, 4, true, (\x.x), seq:();;
2   ("str", 3.500000, 4, 4, true, [funct]):'seq
3
4   mgs> v_set := "str", 3.5, 4, 4, true, (\x.x), set:();;
5   (4, true, 3.500000, "str", [funct]):'set
6
7   mgs> v_bag := "str", 3.5, 4, 4, true, (\x.x), bag:();;
8   (4, 4, true, 3.500000, "str", [funct]):'bag
```

The comma operator is overloaded and used, following the context, to add an element to a collection, to merge two collections of the same type or to create a sequence composed of its elements. For most of the collection types, the empty collection type `xxx` is written `xxx:()` (for the example above, the empty collection for the *sequence* type is namely `seq:()`).

2.2 Functions For the Manipulation of Values

In `MGS`, most of the functions are overloaded to allow an easy handling of complex values. A collection value c has a type $\tau(\mu)$ where τ is the collection type (like `set`, `seq`, `bag`, ...) and μ is the type of the elements of the collection. To allow an easy handling of complex values, most built-in functions are overloaded so that user-defined functions can handle collections of any type $\tau(\mu)$ regardless of τ and μ. That property can be seen as a kind of *polytypism* [5, 19].

For example, the `size` functions, that returns the number of elements in the collection, can be applied to any collection:

```
1   mgs> size(v_seq);;
2   6
3
4   mgs> size(v_set);;
5   5
6
7   mgs> size(v_bag);;
8   6
```

Among all the polytypic functions, we have the classics `map`, `iter`, `fold`, `one_off`, `rest`, `member`... The interested reader should refer to `http://mgs.ibisc.univ-evry.fr/Online_Manual/Collections.html` for the detail of available functions defined on collection types.

2.3 A Short Example

MGS unifies the collection types together with the polytypic functions in a general rewriting scheme. Programs are written as a composition of *transformations*, a very expressive form of rewriting process [12, 13, 17, 30, 32] based on the neighborhood relationship exhibited by each collection type together with a general form of pattern matching.

The following MGS expression returns (if it exists) the Hamiltonian path in a graph G

```
1  trans Hamiltonian =
2    (s* as whole / (size(whole) == size('self)) => whole)
```

Pattern `s*` matches any *path* p (that is, a sequence of neighboring values) in G such that each element in p appears only once; the additional requirement that p is of the same size as G ensures that such paths are Hamiltonian. Of course, the complexity of the search remains, but the complexity of its expression is highly reduced.

3 The Implementation of the `value` Data Structure

3.1 The `value` Data Structure in a Functional Setting

In a functional setting, an evaluator consists in a function `eval` that, given an expression of type `expr`, returns a value of type `value`. A toy example of such an interpreter is given in figure 1.

In this example, the type `value` is restricted to integers and functions. The precise application area of MGS does not matter in this paper and detailing the handling of integers and integers operators should be enough to explain our approach.

Functions in the target language rely on the use of functions of the implementation language (see the example of the `inc_val` function at line 16 in figure 1). This mechanism of representing a target function by an implementation function lies at the heart of the *higher-order abstract syntax* [26, 7] approach. For the sake of simplicity, we do not detail here on how to implement user-defined functions. In the current MGS interpreter, this is achieved by using combinators to translate on-the-fly a user-defined lambda expression into a `Fun` value [6]. The same mechanism can be used in the OO approach presented below. With the higher-order syntax approach it is immediate to integrate existing libraries of functions as a predefined kernel of functions: predefined library functions are embedded using the `Fun` constructor. Note that the functions of the kernel have exactly the same status and implementation as the user-defined functions and so they can be arbitrarily mixed "for free" (e.g. using higher-order operators). In the rest of this paper we focus only on the handling of a set of predefined functional constants like `inc_val`.

If one wants to extend the interpreter with a new primitive, like the addition of integers, it only requires to define the corresponding constant

```
#include <iostream>
using namespace std;

struct value;

struct expr { virtual value& eval() =0; };

struct value : public expr {
   value& eval() { return *this; }
   virtual ostream& print(ostream& o) =0;
};

struct Number : public value {
   virtual Number& inc() =0;
};

struct Int : public Number {
   int val;
   Int(int n) : val(n) {}
   Number& inc() { return *(new Int(val + 1));}
   ostream& print(ostream& o) {return o << val
<< "\n";}
};

struct Fun : public value {
   virtual value& operator() (value&) =0;
   ostream& print(ostream& o)
                  {return o << "<fun>\n";}
};

struct Error : public value {
   char* msg;
   Error(char* s) : msg(s) {}
   ostream& print(ostream& o) {return o << msg
<< "\n";}
};

struct Apply : public expr {
   expr& fct;
   expr& arg;
   Apply (expr& f, expr& a) : fct(f), arg(a) {}

   value& eval() {
      if (Fun* f = dynamic_cast<Fun*>(&(fct.eval())))
 return (*f)(arg.eval());
      else
 return *new Error("apply: type error");
   }
};

struct Inc : public Fun {
   value& operator() (value& arg) {
      if (Number* a = dynamic_cast<Number*>(&arg))
 return a->inc();
      else
 return *new Error("bad arg");
   }
};

main()
{
   Int v(1);
   Inc incr;
   Apply tmp(incr, v);
   Apply(incr, tmp).eval().print(cout);
}
```

Fig. 3. A simple and basic interpreter expressed in an OO programming style.

```
1   let add_val =
2     Fun(function (Int v1) ->
3         Fun (function (Int v2) -> Int (v1 + v2)))
```

in a new file and to rely on separate compilation and linking to produce the new interpreter. The new function can be made available to the `MGS` programmer by registering the previous expression in the global environment under an adequate name.

So, it is straightforward to extend the library of available functions. On the contrary, if we want to extend the available `value` type, for example with floating-points values, we face several problems:

1. the type `value` must be extended accordingly, which implies to edit an existing file,
2. *all* functions defined by case on type `value` have to be updated to take into account the new case.

The second point requires to edit *all existing files* related to the `value` type. For instance, in the context of the `MGS` project, which represents 50k lines of `OCAML` code, spread in about 75 files, it would require a huge amount of work.

3.2 The `value` Data Structure in an Object-Oriented Framework

In a object-oriented (OO) framework, the sum type used in the functional approach is replaced by an *abstract class* whose derived classes represent all the cases. *Methods* are used to implement predefined target functions. The corresponding interpreter, in `C++`, is given in figure 3.

The `dynamic_cast<...>(...)` is used for downward casting a class to one of its derived classes in a safe way. Failure to downcast corresponds to type errors during evaluation of `MGS` expressions. `value` are defined as a subtype of `expression`. A class `Number` gathers all classes that admit numerical operations like incrementation. Initially, the only descendant of `Number` is `Int` which represents integers. Despite the syntactic differences, the OO `C++` code mimics closely the functional approach. The `eval` methods applies to any expression and is defined, case by case, on each derived subclasses. The real difference is that the cases are not gathered in one place but scattered in each derived classes. The evaluation of a value is always the identity and so it is defined at the level of the `value` class.

If one wants to extend the interpreter with a new data type, like floating-points values, it only requires to define the corresponding derived class

```
1   struct Float : public Number {
2      float val;
3      Float(float f) : val(f) {}
4
5      value& inc() { return *(new Float(val + 1.0)); }
6
7      ostream& print(ostream& o)
8      { return o << val << "\n"; }
9   };
```

in a new file and to rely on separate compilation and linking to produce the new interpreter.

So, it is straightforward to add new target data structures. On the contrary, if we want to extend the library of available functions, we have to add a virtual function to the mother-class `value` or one of its derived classes. This implies to edit the class value but also *all the derived classes* for which an implementation of the new method is relevant.

arity	number	min cases	average cases	max cases
1	100	1	3.43	24
2	93	1	5.77	40
3	22	1	2.4	14
4	4	1	1	1
5	0			
6	4	1	6	21
7	2	1	12	23

Fig. 4. Statistics summary of overloaded functions in `MGS`.

4 Implementing Overloading

The implementation of an incremental interpreter has also to face an additional problem if we provide to the end-user *overloaded target functions*. In the previous example, the function `inc` has a meaning for both integer and floating-points values. It would be very convenient to offer to the end-user an overloaded function acting on both types. This means that from an `MGS` identifier `inc` and the type of the arguments in an application, some *dispatch* mechanism must be used to call the correct implementation method or function. This problem is not negligible. In the `MGS` context, there are many overloaded functions: figure 4 gives the number, and distribution with respect to their arity, of overloaded target functions available to the end-user.

In the functional framework, the dispatch is easily provided for unary functions, using definition by cases through the pattern matching on the constructors of the `value` data type. In the OO framework, this is also easily achieved using virtual methods.

Things get more complex when we consider functions with multiple arguments. For example, consider the addition of two values. Pattern matching can still be used, but at the price of explicitly writing the Cartesian product of the `value` constructors. For example, in the current `MGS` interpreter, there are 24 available data types. So, overloading the addition comes at the cost of writing 576 cases. Obviously, most of the cases correspond to errors and are handled similarly. Even if this can be done using wild-cards in patterns, there is still a huge number of cases to be written.

In the OO framework, the extension of the overloading of a target function to multiple arguments requires *multiple dispatch* [18]. Multiple dispatch can be implemented (in languages with only single dispatch, like `C++` or `OCAML`) using auxiliary methods [25, item 31]. The number of these functions also grows exponentially with the number of arguments meaningful for the dispatch.

5 An "Incremental" Software Architecture for the `MGS` Interpreter

Our first design decision in `MGS` was to rely on the functional approach. As a matter of fact multiple dispatch is easier to implement in this framework. However, the problems raised in section 3.1 have still to be addressed. Our idea is to split the various cases of an overloaded function into multiple `OCAML` functions spread through the whole set of files. A pre-processing phase gathers all the defined functions and merges them into the actual implementation. A similar process is done for the various constructors of the `value` data type.

Splitting the definition into several files raises the problem of functional dependency. It is hopeless to force the developer to have a correct sequencing of the files when we want to enable at the same time the unconstrained addition of new data types and pieces of code. To solve this problem, we use a well-known technique of forward pointers that are correctly set at run-time (see for example [21, page 150]).

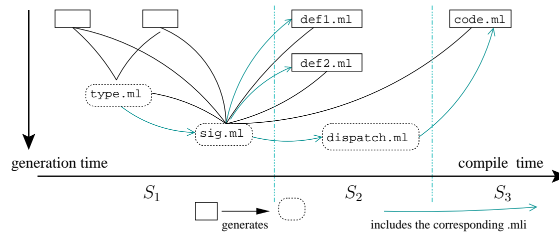

Fig. 5. Organisation of the code: the three phases S_1, S_2 and S_3 are given together with the exact date when each file is produced and the functions are made available.

We detail in the rest of this section the overall software organization through the description of a small example. We assume that the `value` data type is completely defined once and for all. Section 6.1 sketches how this data type can also be generated from informations gathered through all the code. The reader is supposed to be familiar with the `OCAML` language and its compilation tools.

5.1 Organization of the Code

The project consists in three set of files, S_1, S_2 and S_3. A *dispatch* will be computed from the definitions occurring in S_1 and S_2. After S_1, the *signature* of the dispatched functions are available (for the functions defined in S_2 and S_3). That is, the functions are called through a *diversion* mechanism. After S_2, the functions can be directly called since all dispatched functions are known *and initialized* after S_2. Then, the dispatch is effective and the *direct call* to the dispatched functions is possible. Figure 5 shows the clear timing of the operations occurring in the three phases and what files are used.

5.2 S_1: Basic Definitions

The files in S_1 are *definitions*, usually *types* and *functions*, that do not rely on other previous definitions and that will be used everywhere in the project.

It includes a file `types.ml` that defines the type of the values (the `value` type) that are going to be handled. All the functions that will handle values in the code will require to have access to this file. From the `.ml`, a `.mli` *include* file is produced by the `OCAML` compiler. Using this include file through the `open Types` directive all other files are able to define functions on `value`.

```
1  type value =
2      Int of int
3    | Float of float
```

5.3 The Diversion Mechanism: Generation of Forward and Signatures

At this point, it is necessary to give access to the overloaded functions, which raises two problems:

1. since the functions are defined incrementally, there is no global repertory of them;
2. these functions must be made available for code in S_2 and S_3 independently of their actual implementation localization.

These two problems are solved by scanning all implementation files to collect the various function names to generate a unique file `sig.ml` providing the implementation of the forwarding mechanism. The scanning is made possible by enforcing a specific syntax for the function names (see below).

For our example, the generated `sig.ml` file is

```
1   open Types;;
2
3   (* Signature declaration *)
4   let (add_forward : (value -> value -> value) ref) =
5     ref  (function _ -> failwith "unitialized add")
6
7   let (print_forward : (value -> unit) ref) =
8     ref  (function _ -> failwith "unitialized print")
9
10  Printf.printf "Setting the forward pointers\n"
11
12  let add x y = !add_forward x y
13  let print x = !print_forward x
```

The forward mechanism works as follows: an overloaded function `add` is a *wrapper* that applies the value of the imperative variable `add_forward`. This imperative variable is initialized with a dummy function raising an error. This variable will be set later with the correct function (see lines 20 and 21 of the file `dispatch.ml` in section 5.5).

5.4 S_2: Writing of Code

The files in the *second set* S_2 contains the implementation of the various cases of an overloaded function. Suppose that a unary function `XX` is overloaded on two types `p` and `q`. This suppose that the `value` data type has two constructors `P` and `Q` defined like

```
1   type value = ...
2   | P of type_p
3   ...
4   | Q of type_q
5   ...
```

Then the `MGS` implementers have only to provide two functions called `_XX_p` and `_XX_q` both of arity one. The argument of `_XX_p` is of type `type_p`. The naming convention is simple: the name of the constructor (which is constrained to always begin with a capital letter in `OCAML`) is used in small letter in the name of the function case.

The naming convention is straightforwardly extended to handle multiple arguments. A function definition:

$_XX_p_1_ \ldots _p_n$

represents the handling of the arguments of type `type_p`$_1$, ..., `type_p`$_n$ for the overloaded function `XX`. The types `type_p`$_i$ are arguments of constructors of the sum type `value`. Each constructor corresponds to a different `MGS` value type and we assume that the `type_p`$_i$ are all different, even if the implementation type are the same by using alias type declaration. This naming convention enables the scanning described in the previous section and the generation of the diversion functions `XX` and `XX_forward` as well as the dispatch function `__XX` described in the next section.

An example of two overloaded functions, `add` and `print` is given in the `def1.ml` file below:

```
1   open Types;;
2   open Sig;;
3
4   let _add_int_int i1 i2     = Int (i1 + i2)
5
6   let _print_int i1          = Printf.printf "%d" i1
7   let _print_float f1        = Printf.printf "%f" f1
```

Note that the definition of `add` is, at this point, not complete. Other cases are specified or will be specified in other files.

All functions are allowed to recursively call any overloaded function. For example, in another file `def2.ml`, the definition of `_add_float_int` uses the overloaded function `add`:

```
1   open Types;;
2   open Sig;;
3
4   let _add_int_float i1 f1   = Float
5                                (f1 +. (float_of_int i1))
6   let _add_float_float f1 f2 = Float (f1 +. f2)
7   let _add_float_int f1 i1   = add (Int i1) (Float f1)
```

Note however that `add` can only be effectively used once the wrapper has correctly been set at run-time. This means that, at this point, only function *definitions*, implying overloaded functions, can occur and *no actual function calls* to overloaded functions.

5.5 Generation of the Overloaded Functions

An overloaded function is implemented using pattern matching to dispatch to the several function cases. The implementation function corresponding to the overloaded function `XX` is called `__XX`. For our example, the generated `dispatch.ml` file is:

```
1   open Types;;
2   open Sig;;
3   open Def1;;
```

```
4   open Def2;;
5
6   let __add x y = match x, y with
7     |  (Int x0), (Int x1)     -> _add_int_int x0 x1
8     |  (Float x0), (Float x1) -> _add_float_float x0 x1
9     |  (Int x0), (Float x1)   -> _add_int_float x0 x1
10    |  (Float x0), (Int x1)   -> _add_float_int x0 x1
11
12  and __print x = match x with
13  | Int x0   -> _print_int x0
14  | Float x0 -> _print_float x0
15
16  Printf.printf "Setting the correct link\n"
17  flush Pervasives.stdout
18
19  Sig.add_forward   := __add
20  Sig.print_forward := __print
```

At the end of the file, the imperative variables used in the wrapper functions are set to their correct value, using the just defined `__XX` functions.

5.6 S_3: Using Dispatched Functions

At this point, all function cases have been gathered, the overloaded functions have been generated and can be used even in the initialization phase, on the contrary to the code in the S_2 set of files. In the MGS project, the files in S_3 corresponds to the implementation of transformations, the parsing, the top-level, etc.

To finalize our running example, the file `code.ml` below describes some possible use of the overloaded functions, `add` and `print`:

```
1   open Types;;
2   open Sig;;
3
4   print (add (Float 2.0) (Float 3.0))
5   print_newline()
6   print (add (Float 2.0) (Int 1))
7   print_newline()
8   print (add (Int 2) (Float 1.0))
9   print_newline()
10  print (add (Int 2) (Int 1))
11  print_newline()
```

5.7 Compilation and Execution of the Code

The compilation follows five phases to respect the code organization:

1. in a first phase, all the files in S_1 are compiled;

2. in a second phase, all the files of the project are scanned to automatically generate and compile the `sig.ml` file;
3. in a third phase, all files from S_2 are compiled (which additionally produces the include files required for `dispatch.ml`);
4. in a fourth phase, `dispatch.ml` is generated and compiled;
5. finally, files in S_3 are compiled and the final linking is done.

This process is fully automated by a `Makefile`. The compilation and the execution of our example gives:

```
ibisc 12 > make

ocamlc -c types.ml
ocamlc -c sig.ml
ocamlc -c def1.ml
ocamlc -c def2.ml
ocamlc -c dispatch.ml
ocamlc -c code.ml
ocamlc -o dsal types.cmo sig.cmo def1.cmo def2.cmo\
               dispatch.cmo code.cmo

ibisc 13 > dsal

Setting the forward pointer
Setting the correct link
5.000000
3.000000
3.000000
3
```

6 Weaving the Implementation Code

In this section, we sketch the automatic generation of the `type.ml`, `sig.ml` and `dispatch.ml` files.

6.1 Weaving the `value` Data Structure

In the same way that the function cases are split through several files, the various constructor of the `value` data type are split in several files. This enables to add a new data structure to `MGS` simply by providing a new file introducing the corresponding constructor. The precise syntax used for the constructor declaration does not matter. The first weaving tool scans all the source files to gather all the constructors related to the `value` type and generates the `types.ml` file.

6.2 Weaving the Dispatch on `value` Type

The second weaving tool gathers all the function cases spread among the source files to generate the overloaded functions. The dispatch mechanism presents some

subtleties. In the previous example, all the types used as the arguments of the constructors of the `value` type where incomparable. However, the situation is more complex in the implementation of MGS:

– wild-cards are required to handle within the same case function various argument types;
– there is a hierarchy of data types in the MGS language that is available to the developer of the MGS language.

A simple example of the last kind is the following: MGS values are split into *atomic* and *compound* values. Sometimes, cases functions are dispatched on this distinction, and not on the implementation type of the data structure. For example, the primitive function `size` returns `-1` on all atomic values and returns the number of elements in its argument in case of a compound value. Interior nodes of the MGS hierarchy type corresponds to several constructor in the `value` type. The type of the argument passed to the dispatched function is then `value` and not the argument type of a constructor.

Having *family* of types produces a hierarchy that has to be taken into account while generating the pattern matching of the overloaded functions. For example, a case on `_XX_int_int` has to appear before the case `_XX_int_atomic`. The partial order relationships between the MGS types is used to sort lexicographically the collected cases of an overloaded function.

A "catch-all" case is produced to handle "bad argument types" error. To avoid spurious warnings by the OCAML compiler, this case is produced only if required.

7 Conclusion

The software organization and the weaving tools described in this paper have been successfully used in the development of the MGS interpreter. This represents over 50k lines of OCAML code (there is also over 100k lines of C++ libraries to provide basic support for sophisticated data structures like Voronoï tessalation, G-Maps, Cayley graphs, ...). The 50k lines of OCAML files are scattered over 75 files. The scanning of these files is almost immediate and does not slow down the compilation process. It generates 225 overloaded functions. These results show that our approach is well suited to the development, in a functional setting, of large incremental projects.

One of the originality of this work is the application of aspect weaving techniques in the context of a functional language (OCAML). As far as we know, this is the first attempt to merge these two worlds to ease the implementation of a domain-specific language. Our approach relies only on a tailored software architecture, a dedicated `makefile`, some naming conventions and two external tools to parse and collect informations on the various data types entering in the `value` type and on the overloaded functions. It does not involves any changes on the OCAML compiler nor sophisticated typing techniques. It is therefore a lightweight solution to the problem of incrementally building an interpreter.

Related Works.

The various techniques implied have already been used in other contexts (for example, wrapper functions are used to overcome the impossibility to have recursively defined modules spun across multiple files) and the problem that we have tried to solve has been coined *the expression problem* in [34] (with an enlightening discussion in [35]). We briefly review, because of space limitation, some similar approaches.

Language Extensions. In [20] is proposed a specific design pattern called the *Extensible Visitor* which is a combination of functional and object-oriented programming methods while our approach is purely functional.

An aspect-oriented programming extension to `OCAML`, very similar to AspectJ [3], is proposed in [24]. It is a highly technical approach that uses the usual features of *join points*, *pointcuts* and *advices declarations* that leads to the definition of the `Aspectual Caml` language while our work do not change the language itself but consists in two additional tools to collect information and produce the dispatch files.

Extensible Interpreters. The conception of extensible interpreters has been considered for example in [33]. However, it requires sophisticated type inference techniques to be implemented that goes beyond standard `ML` type inference.

Multiple dispatch has been considered for overloaded functions in a functional language [4]. As for the previous work, it requires sophisticated types techniques.

Extensible sum data types[8, 9] (which is further extended in [10] by adding private row types to functors) have been proposed and are implemented in `OCAML`. They enable the incremental definition of the `value` data type and of the functions but at the cost of requiring a lot of wrap/unwrap functions that are done for free in our approach. Moreover, since with polymorphic variants a matching case can easily be forgotten in a function definition, we believe that this approach would be too error-prone on a large-scale development like the `MGS` language

Once again, a very technical solution is found in [27] by relying on modules and (higher-order) functors.

Acknowledgements.

The authors thank Julien Cohen of LINA – CNRS FRE 2729 for his comments on the paper.

References

1. Computer language shootout scorecard, June 2003. `http://dada.perl.it/shootout/craps.html`.
2. Gentoo : Intel® pentium® 4 computer language shootout, July 2006. `http://shootout.alioth.debian.org/gp4/index.php`.

3. AspectJ project. Available at http://www.eclipse.org/aspectj/.
4. Bourdoncle, F., and Merz, S. Type checking higher-order polymorphic multi-methods. In *Conference Record of POPL'97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), ACM SIGACT and SIGPLAN, ACM Press, pp. 302–315.
5. Cohen, J. *Intgration des collections topologiques et des transformations dans un langage fonctionnel.* PhD thesis, Université d'Évry, Dec. 2004.
6. Cohen, J. Interprétation par SK-traduction et syntaxe abstraite d'ordre supérieur. In *Journées Francophones des Langages Applicatifs (JFLA 2005)* (2005), O. Michel, Ed., INRIA, pp. 17–34.
7. Cohen, J. Interprétation par syntaxe abstraite d'ordre supérieur et traduction en combinateurs. *Technique et science informatiques* (2007). To appear.
8. Garrigue, J. Programming with polymorphic variants. In *Proc. of 1998 ACM SIGPLAN Wksh. on ML, Baltimore, MD, USA, 26 Sept. 1998*. Oct. 1998.
9. Garrigue, J. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering (FOSE)* (Nov. 2000).
10. Garrigue, J. Private row types: Abstracting the unnamed. In *APLAS* (2006), N. Kobayashi, Ed., vol. 4279 of *Lecture Notes in Computer Science*, Springer, pp. 44–60.
11. Giavitto, J.-L. A framework for the recursive definition of data structures. In *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)* (Montréal, Sept. 2000), ACM-press, pp. 45–55.
12. Giavitto, J.-L. Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)* (Valencia, June 2003), vol. LNCS 2706 of *LNCS*, Springer, pp. 208 – 233.
13. Giavitto, J.-L., Malcolm, G., and Michel, O. Rewriting systems and the modelling of biological systems. *Comparative and Functional Genomics 5* (Feb. 2004), 95–99.
14. Giavitto, J.-L., and Michel, O. Modeling the topological organization of cellular processes. *BioSystems 70*, 2 (2003), 149–163.
15. Giavitto, J.-L., Michel, O., and Cohen, J. Pattern-matching and rewriting rules for group indexed data structures. *ACM SIGPLAN Notices 37*, 12 (Dec. 2002), 76–87.
16. Giavitto, J.-L., Michel, O., Cohen, J., and Spicher, A. Computation in space and space in computation. Tech. Rep. 103-2004, May 2004. 22 p.
17. Giavitto, J.-L., and Spicher, A. *Systems Self-Assembly: multidisciplinary snapshots.* Elsevier, 2006, ch. Simulation of self-assembly processes using abstract reduction systems.
18. Ingalls, D. H. H. A simple technique for handling multiple polymorphism. In *OOPSLA* (1986), pp. 347–349.
19. Jeuring, J., and Jansson, P. Polytypic programming. In *Tutorial Text from 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, J. Launchbury, E. Meijer, and T. Sheard, Eds., vol. 1129 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996, pp. 68–114.
20. Krishnamurthi, S., Felleisen, M., and Friedman, D. P. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP* (1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer, pp. 91–113.
21. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. *The Objective Caml system, release 3.09.* INRIA, October 2005. available at http://caml.inria.fr/distrib/ocaml-3.09/.

22. LIENHARDT, P. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design 23*, 1 (1991), 59–82.
23. LIENHARDT, P. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal on Computational Geometry and Applications 4*, 3 (1994), 275–324.
24. MASUHARA, H., TATSUZAWA, H., AND YONEZAWA, A. Aspectual caml: an aspect-oriented functional language. In *ICFP* (2005), O. Danvy and B. C. Pierce, Eds., ACM, pp. 320–330.
25. MEYERS, S. *More Effective C++*. Addison Wesley, 1996.
26. PFENNING, F., AND ELLIOT, C. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988), pp. 199–208.
27. RAMSEY, N. ML module mania: A type-safe, separately compiled, extensible interpreter. *Electr. Notes Theor. Comput. Sci 148*, 2 (2006), 181–209.
28. REMY, D., AND VOUILLON, J. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems 4*, 1 (1998), 27–50.
29. SPICHER, A. *Transformation de collections topologiques de dimension arbitraire. Application la modélisation de systèmes dynamiques.* PhD thesis, Université d'Évry, 2006.
30. SPICHER, A., AND MICHEL, O. Using rewriting techniques in the simulation of dynamical systems: Application to the modeling of sperm crawling. In *Fifth International Conference on Computational Science (ICCS'05)* (2005), vol. I, pp. 820–827.
31. SPICHER, A., MICHEL, O., AND GIAVITTO, J.-L. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)* (Amsterdam, October 2004), vol. 3305 of *LNCS*, Springer.
32. SPICHER, A., MICHEL, O., AND GIAVITTO, J.-L. *Rewriting and Simulation - Application to the Modeling of the Lambda Phage Switch*, vol. Modélisation de systèmes biologiques complexes dans le contexte de la génomique. Genopole, 2006, ch. Modeling of the Lambda Phage Switch.
33. STEELE JR, G. L. Building interpreters by composing monads. In *POPL* (1994), pp. 472–492.
34. WADLER, P. The expression problem. Email to the Java Genericity mailing list, Dec. 1998.
35. ZENGER, M., AND ODERSKY, M. Independently extensible solutions to the expression problem. In *The 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)* (Long Beach, California, 2005), ACM.