Future Generation Computer Systems 16 (2000) 753–769

# Abstract machine construction through operational semantics refinements

Frédéric Cabestre, Christian Percebois, Jean-Paul Bodeveix*

*IRIT, Université Paul Sabatier, 118, route de Narbonne, 31062 Toulouse Cedex4, France*

Accepted 24 May 1999

**Abstract**

This article describes the derivation of an abstract machine from an interpreter describing the operational semantics of a source language. This derivation process relies on the application of a set of gradual transformations to the interpreter written in a functional language. Through pass separation, the derivation process leads to the extraction of a compiler and an abstract machine from the transformed interpreter. © 2000 Elsevier Science B.V. All rights reserved.

*Keywords:* Abstract machine; Semantics; Pass separation; Compiler

## 1. Introduction

The increasing interest in abstract machines subsequent to their portability, advocated by the popularity of the Java language [1], raises the problem of their design. Generally, it needs an empiric survey of the semantics of the high-level language to study. In other words, one analyses the working of an interpreter embodying the operational semantics of the language to find the data structures and an emulator for an intermediate language constituting the abstract machine. Some propositions have been made aiming at organizing this conceptual study. Among the methods suggested, we put forward those proposing an interpreter or an operational semantics as starting point and a set of gradual transformations as principle. Section 2 discusses existing design methods. Section 3 presents the concepts supporting the derivation process. Section 4 illustrates the method through a concrete example.

## 2. Design methods of abstract machines

An abstract machine is composed of an interpreter for the intermediate language that it defines and of a *run-time environment* [2] on which relies the interpreter. In what follows, the instructions of the language interpreted by the abstract machine will be called *abstract instructions* and their interpreter will be called *emulator*. To design an abstract machine, we must conceive an abstract data type *environment*, a suitable instruction set and an emulator for this instruction set using the environment. In this section, we present studies connected to the design of an abstract machine and present the guiding lines of the derivation process.

### 2.1. Principles of existing methods

Existing methods rely on the principle of *pass separation* introduced by Jørring and Scherlis [3]. This

* Corresponding author.
*E-mail addresses:* cabestre@irit.fr (F. Cabestre), perceboi@irit.fr (C. Percebois), bodeveix@irit.fr (J.-P. Bodeveix)

section illustrates the application of this principle to the derivation of abstract machines. Methods depend on the formalism used to write the interpreter and on the considered source language. The starting point of these studies is a simple expression of the semantics of a language, its mechanisms being implicitly taken into account by the power of the implementation language. Then, these mechanisms are gradually described using lower level constructs. Consequently, data structures constituting the abstract machine appear. Finally, pass separation extracts a compiler, an emulator and abstract instructions.

#### 2.1.1. *Pass separation*

Pass separation [3] applies to an interpreter $I(p, d)$ of a source language, where $p$ is a user program and $d$ is the input data. The computation is split into two stages: the compile-time stage $C(p)$ performing as much work as possible with the static parameter $p$, and the run-time stage $E(c, d)$ using the result $c$ of the first stage and the data $d$. Thus, we have $I(p, d) = E(C(p), d)$, where $C$ is the compiler of the source program to some intermediate code and $E$ is the emulator of the intermediate code. Consequently, pass separation introduces a new data structure $c$ corresponding to abstract instructions, a compiler and an emulator.

#### 2.1.2. *Use of inference rules*

Concerning functional languages, Hannan and Miller [4] propose an ad hoc progressive transformation of an operational semantics described by inference rules over lambda-terms. The obtained result is an abstract machine defined by rewriting rules over the source language. Hannan [5] performs pass separation over these rewriting rules.

Sestoft [6] transforms, in only one stage, the natural operational semantics of a lazy functional language into an abstract machine, but the derivation steps are still ad hoc.

Diehl [7] introduces a specification formalism, the two level big-step semantics (`2BIG`), over first-order judgements. He defines and validates a set of progressive and automatic transformations leading to a compiler and an emulator. His pass separation step relies on Hannan's proposition. Furthermore, in [8], Diehl applies pass separation on evolving algebras [9]. However, they behave as abstract transition systems without any high-level control.

#### 2.1.3. *Denotational semantics*

All these approaches use natural semantics as specification language and rewrite rules as target language. Wand [10] uses a continuation-based denotational semantics applied on the derivation of an abstract machine for a procedural language. He introduces combinators to eliminate free variables from the semantics equations. Then, each combinator is replaced by a first order term, and so the interpreter becomes a compiler with these terms as target language. An emulator associates to each term its semantics defined by a combinator. However, all these steps are highly empiric.

### 2.2. *Overview of the derivation process*

The derivation presented in this paper shares the same fundamental principles as most of the works cited above, i.e. writing a first simple interpreter, giving a gradual clarification of its mechanisms and extracting an emulator and a compiler through pass separation. These works can be classified according to the specification formalism used to express the semantics of the source language [7]. Here, we consider an executable formalism for operational semantics description which is a functional language. Thus, we join Wand's approach [10] concerning the choice of the description formalism, altogether stating more precisely each transformation step. We use for our survey Caml [11], a dialect of ML.

Within a functional framework, the writing of the first interpreter uses lexical bindings, recursion and higher order functions. Then we ignore these capabilities to write new versions of the interpreter. The continuation passing style is used to specify control flow in a uniform way. However, the power of lexical binding still allows a high-level description of the data flow similar to recursion for the control flow. This mechanism must be eliminated to obtain a low-level machine code, thus producing the data structures of the run-time environment. Abstract instructions and their emulator are then extracted through pass separation.

## 3. Concepts supporting the derivation process

The derivation process uses techniques issued from compilation, partial evaluation (binding-time analysis) and program transformation (pass separation). Its objective is to extract an abstract machine from an interpreter supposed to be written in a functional language. An abstract machine is defined by an abstract data type, called the run-time environment, an intermediate language, and an emulator for this language using the abstract data type. In this section, we briefly introduce partial evaluation and binding-time analysis.

The objective of partial evaluation is to specialize a program using a known part of its input. Consider a program $P$ taking as inputs $s$ and $d$, where $s$ is the statically known part of its inputs and $d$ its dynamic part. Specializing $P$ means building a program $P_s$ such that $P_s(d) \equiv P(s, d)$. This equation must be related to the one of pass separation as in both cases, the goal is to separate static and dynamic parts of the input. However, pass separation defines an intermediate code used to express $P_s$. The common part between partial evaluation and pass separation, named *binding-time analysis*, is in fact the preprocessing phase of offline partial evaluators.

Binding-time analysis is an important phase of offline partial evaluators preceding specialization. It consists in annotating (i.e. underlining) the source program to distinguish static from dynamic constructions among applications, abstractions, conditionals and fixed points. For example, given that `x` is statically known, we get $\underline{\texttt{fun}}\,\texttt{z} \rightarrow \texttt{x} \pm ((\underline{\texttt{fun}}\,\texttt{y} \rightarrow \texttt{y} \pm \texttt{succ x})\texttt{z})$. The evaluation of dynamic operators by the partial evaluator produces code that will be executed at run-time while static operators are interpreted by the partial evaluator itself. For this purpose, the analysis classifies the parameters of each function into static or dynamic. In the following, we only consider this classification.

More elaborated versions also distinguish partially static closures and expressions. Then two families of analysers must be mentioned: monovariant analysers associate a unique classification to each function; polyvariant analysers associate to them a finite set of classifications depending on their call point [12].

## 4. Design of abstract machines

We now present in detail the derivation steps leading to the definition of an abstract machine. As a demonstration example, we consider in this section an interpreter of arithmetic expressions. It includes a construction `try expr1 with expr2` allowing to catch exceptions raised during the computation, here division by zero. This expression returns the value of `expr1`, or the value of `expr2` if the evaluation of `expr1` raises an exception. We also consider a recursive operator `SUM(i,b,e)` which computes $\Sigma_{i=0}^{i=b-1} e_i$.

The successive versions of the interpreter use structural induction over an abstract data type `expression` with constructors for identifiers (`ID`), integers (`INT`), try-with constructs (`TRY`), binary operators (`ADD, SUB, MUL, DIV`), and the sum operation `SUM`.

### 4.1. Writing a first interpreter

The first stage consists in writing an interpreter for the language to implement. This interpreter is either written in continuation passing style if special control flow must be specified, or results from the transformation of a direct style interpreter into continuation passing style [13]. In our example, the first interpreter, described in Fig. 1, uses continuation passing style. This style is used to interpret the `TRY` operator without the help of the `try-with` exception mechanism of Caml as this high-level control structure must be eliminated. Furthermore, this style allows a more precise control of the data flow, which avoids data copying.

The function `eval1` has the following arguments:

- A term of the abstract syntax of type `expression`.
- A forward continuation `cont` associated to the normal progress of the computation. It returns an integer and has three arguments: an exception continuation, the value of the previous sub-expression and the list of variable bindings.
- an exception continuation `exs`, which takes as argument the list of variable bindings and returns an integer. It pursues the computation from the enclosing `try ... with ...` if an exception is raised.
- a list `bnds` of pairs (`name, value`) where `name` is the name of a variable identifier and `value` is the value associated to this variable i.e. an integer.

Let us see more in detail the various forms of terms to be interpreted:

- On an integer `INT`, we pursue the evaluation with the same exception continuation; the returned value, transmitted to the forward continuation, is the integer itself (line 1).

```
     let rec eval1 = function
1        INT(n) -> fun cont exs bnds -> cont exs n bnds
       | ID(i) ->
2          fun cont exs bnds -> cont exs (List.assoc i bnds) bnds
       | ADD(e1,e2) -> fun cont exs bnds -> eval1 e1
           (fun exs1 v1 bnds1 -> eval1 e2
             (fun exs2 v2 bnds2 ->
3              cont exs2 (v1 + v2) bnds2) exs1 bnds1) exs bnds
       | DIV(e1,e2) -> fun cont exs bnds -> eval1 e2
4          (fun exs2 v2 bnds2 -> if v2 = 0 then exs2 bnds2
             else eval1 e1 (fun exs1 v1 bnds1 ->
5                                cont exs1 (v1 / v2) bnds1)
                        exs2 bnds2) exs bnds
       | TRY(e1,e2) -> fun cont exs bnds ->
6          eval1 e1 (fun _ bnds1 -> cont exs bnds1)
7                   (fun bnds2 -> eval1 e2 cont exs bnds2) bnds
       | SUM(i,b,e) -> fun cont exs bnds -> sigma1 i e b cont exs bnds
       | ...
     and sigma1 i e b cont exs bnds =
8        if b <= 0 then cont exs 0 bnds
         else eval1 e (fun exs1 v1 bnds1 ->
9          sigma1 i e (b-1)
                  (fun exs2 v2 bnds2 ->
                     cont exs2 (v1+v2) bnds2) exs1 bnds)
                  exs ((i,b-1)::bnds);;
```

Fig. 1. The first interpreter `eval1`.

- On a variable `ID`, we pursue the evaluation with the value found in the bindings list (line 2).
- On `ADD(e1,e2)`, the sub-expression `e1` is evaluated; its continuation evaluates the second sub-expression `e2` whose continuation calls the initial continuation with the sum of the two intermediate results (line 3).
- On `DIV(e1,e2)`, the sub-expression `e2` is evaluated. If the result is equal to zero, the exception continuation is called (line 4); otherwise `e1` is evaluated and the continuation pursues the evaluation with the quotient of the two results (line 5).
- On `TRY(e1,e2)`, the first sub-expression `e1` is evaluated, but its exception continuation is updated; it signals that if a division by zero occurs, `e2` must be evaluated with the same continuation `exs` as the whole expression (line 7). However, if the evaluation of `e1` does not raise an exception, the evaluation must be pursued with the normal forward continuation `cont` which is saved through lexical binding (line 6).
- On `SUM(i,b,e)`, the function `sigma1` calls the forward continuation if the bound `b` has reached `0` (line 8). Otherwise, the body of the sum is evaluated in a new binding environment where the identifier `i` is linked to `b-1`. The forward continuation calls `sigma1` with `b-1` and sums the results (line 9).

Note that variable bindings remaining unchanged outside `sigma1` are transmitted from continuation to continuation, avoiding unnecessary duplications. The same holds for exception continuations. A more natural writing of this first interpreter, corresponding to an automatic translation of a recursive writing would lead to multiple copies of these data.

```
  let interp1 e bnds =
      eval1 e
1       (fun _ v _ -> v)
2       (fun _ -> print_string "Uncaught exception" ; 0)
        bnds;;
```

Fig. 2. The top level evaluation function `interp1`

The top level function `interp1` of Fig. 2, transmits to `eval1` two functions: a forward continuation (line 1) which takes three arguments, among which the result of the evaluation, and an exception continuation (line 2) which displays an error message.

### 4.2. Binding-time analysis

This stage performs binding-time analysis in order to identify variables whose values are known at compile-time and at run-time. We use a monovariant analysis to get a unique version of the code, thus making easier the reading of the paper. The analysis differentiates partially static functions from static functions, the first ones having free dynamic variables in their definition. In fact, such a function will be transformed into a pair made of the static part of the code and a dynamic environment. Among the techniques proposed by [12], we have used abstract interpretation on a lattice with the following elements:
- $\bot$ is the minimal element associated to unreachable expressions,
- `S` $>\bot$ corresponds to first order static data,
- `D` $>$ `S` corresponds to first order dynamic data,
- `SF[e]` $>\bot$ corresponds to static functional variables,
- `PsF[e]` $>$ `SF[e]` corresponds to partially static functional variables,
- `DF[e]` $>$ `PsF[e]` corresponds to dynamic functional variables,
- the maximal element $\top$ is unused as it corresponds to badly typed expressions.

Abstract interpretation starts with the top level evaluator where arguments are annotated as static or dynamic. On each call, the annotation of a formal parameter must be greater than or equal to that of the corresponding argument. A lambda-expression becomes partially static if it contains a free dynamic variable. A partially static parameter becomes dynamic if the termination of the function depends on the value of its dynamic arguments. The algorithm stops when a fixed-point is reached.

Within annotations `SF[e]`, `PsF[e]` and `DF[e]`, `e` is the set of labels of lambda-expressions that the variable can take as value. Hence, the annotations `SF[e]`, `PsF[e]` and `DF[e]` form a lattice ordered by set inclusion. The result of this analysis leads to the three following annotations:
- $\mathtt{v}^{D}$ designates a dynamic variable only known at run-time,
- $\mathtt{v}^{\mathtt{PsF}}$ designates a partially static closure whose code part is statically known and whose free variables are only known at run-time,
- $\mathtt{v}^{\mathtt{DF}}$ designates a dynamic function.

Thus, the values of unmarked variables are supposed to be known at compile-time. Supposing that `eval1` is called with compile-time known values `e`, `cont` and `exs`, the binding-time analysis annotates as partially static the continuations `cont`, `exs`, `exs1`, `exs2`, and as dynamic the bindings `bnds` and intermediate results `v1` and `v2`.

With respect to the function `sigma1` which calls itself with the same static parameters `i` and `e`, its functional parameters `cont` and `exs` become dynamic and are annotated `DF`.

### 4.3. Lexical bindings elimination

This stage eliminates the use of lexical bindings to access annotated variables as they depend on the run-time environment of the interpreted program. Thus, we have underlined lexically accessed annotated variables in the following code fragment. Note that the underlining used in Section 3 denotes dynamic basic constructions and not dynamic variable references.

```
|ADD(e1,e2) $\rightarrow$ fun cont$^{\mathsf{PsF}}$ exs$^{\mathsf{PsF}}$ bnds$^{D}$
$\rightarrow$
eval1 e1
  (fun exs1$^{\mathsf{PsF}}$ v1$^{D}$ bnds1$^{D}$ $\rightarrow$ eval1 e2
  (fun exs2$^{\mathsf{PsF}}$ v2$^{D}$ bnds2$^{D}$ $\rightarrow$
    $\underline{\texttt{cont}}$ exs2($\underline{\texttt{v1}}$+v2) bnds2) exs1 bnds1)
  exs bnds
```

Lexical links give access to the value of underlined variables as defined before the evaluation of `e2`. This mechanism allows accesses to previous states of the computation and thus must be eliminated.

Concerning the function `sigma1`, lexical access to the dynamic continuations `cont` and `exs` will also be eliminated.

The elimination of lexical bindings introduces new components of the run-time environment. In order to eliminate lexical bindings of annotated (i.e. underlined) data in a typed context, we use a variant of Reynolds's proposal [14] where only the environment part of partially static functions is transmitted to nested abstractions. Thus, the code part, as well as static data, remain lexically accessed. We proceed as follows:

- A recursive and generic type `gstack` is introduced with one parameter for each type of functional variable occurring in the code. In our example, we have two kinds of continuations (forward and exception), which leads to two parameters. The type `stack` is obtained as an instanciation of the generic type `gstack`.
- The generic type `gstack` is a sum type with a variant for each abstraction having free annotated variables. The variant associated to an abstraction contains the type of statically accessed annotated data where closures are represented by the type `gstack`. For simplicity, a variant with one entry of type `gstack` is not considered. Furthermore, variants of equal types are not duplicated.
- Each annotated functional variable becomes a pair built from the original variable and a `stack` variable. Then, for each application, the code part of the pair must be applied to its stack part in order to transmit lexical information.
- Each nested function is associated to its `stack` environment encapsulating free dynamic variables. Consequently, a new formal parameter of type `stack` is added to these functions and is bound when applied to their lexical environment.

In our example, we introduce the types of Fig. 3. The type `gstack` defines six variants. For example, we have `Empty` for abstractions without free dynamic variables, `PushInt` for (`fun exs2`$^{\mathsf{PsF}}$ `v2`$^{D}$ $\rightarrow$ ...) that has lexical access to the two annotated variables `v1` and `cont`, and `PushStack` for (`fun _`$\rightarrow$`cont exs`) which has lexical access to `cont` and `exs` and remaining ones for abstractions of the function `sigma1`.

The code of the function `eval1` is then transformed as shown in Fig. 4. Note that the functional variable `cont` becomes the pair (`cont,Stack cs`) when applied (line 1). Similarly, `exs` becomes (`exs,Stack xs`) (line 4). Free dynamic variables are transmitted by building closures (lines 2–3).

```
type bindings = (string * int) list;;

type ('a,'b) gstack =
    Empty
  | PushInt of int * ('a,'b) gstack
  | PushStack of ('a,'b) gstack * ('a,'b) gstack
  | PushCntInt of 'a * int * ('a,'b) gstack
  | PushCntBnd of 'a * int * bindings * ('a,'b) gstack
  | PushEx of 'b * ('a,'b) gstack;;

type stack2 = Stack of (fwcont2,excont2) gstack
and fwcont2 =
    stack2 -> (excont2 * stack2) -> int -> bindings -> int
and excont2 = stack2 -> bindings -> int;;
```

Fig. 3. The stack types.

```
   let rec eval2 = function
1     INT(n) -> fun (cont,cs) exs bnds -> cont cs exs n bnds
    | ID(i) -> fun (cont,cs) exs bnds ->
         cont cs exs (List.assoc i bnds) bnds
    | ADD(e1,e2) -> fun (cont,Stack cs) exs bnds ->
         eval2 e1 ((fun (Stack cs) exs1 v1 bnds1 ->
             eval2 e2
2              ((fun (Stack(PushInt(v1,cs))) exs2 v2 bnds2 ->
                   cont (Stack cs) exs2 (v1 + v2) bnds2),
3               (Stack(PushInt(v1,cs))))
               exs1 bnds1),
            Stack cs
              ) exs bnds
4   | TRY(e1,e2) -> fun (cont,Stack cs) (exs,Stack xs) bnds ->
         eval2 e1
           ((fun (Stack(PushStack(xs1,cs1))) _ bnds1 ->
               cont (Stack cs1) (exs,Stack xs1) bnds1),
            Stack(PushStack(xs,cs)))
           ((fun (Stack(PushStack(xs2,cs2))) bnds2 ->
               eval2 e2 (cont,Stack cs2) (exs,Stack xs2) bnds2),
            Stack(PushStack(xs,cs))) bnds
    | SUM(i,b,e) -> fun cont exs bnds -> sigma2 i e b cont exs bnds
    | ...
```

Fig. 4. Lexical bindings elimination in eval2.

An abstraction extracts its dynamic variables using pattern-matching on its new formal parameter Stack(PushInt(v1,cs)) (line 2). In the same way, sigma1 and interp1 are modified according to Figs. 5 and 6.

Fig. 7 states the equivalence between versions 1 and 2 of the interpreter and its auxiliary functions. The continuations of the first version are built from the continuations of the second version as in (fun ex b $\rightarrow$ cont ...), where cont is a second version continuation. Conversely, first version continuations must be built from second version continuations as in ((fun _ b1 $\rightarrow$ ex b1),StackEmpty), where ex is a first version continuation.

```
... and sigma2 i e b (cont,Stack cs) (ex,Stack xs) bnds =
  if b <= 0 then cont (Stack cs) (ex,Stack xs) 0 bnds
  else eval2 e
    ((fun (Stack(PushCntBnd(cont,b,bnds,cs1))) exs1 v1 bnds1 ->
      sigma2 i e (b-1)
        ((fun (Stack(PushCntInt(cont,v1,cs2))) exs2 v2 bnds2 ->
            cont (Stack cs2) exs2 (v1+v2) bnds2),
         Stack(PushCntInt(cont,v1,cs1)))
        exs1 bnds),
     Stack(PushCntBnd(cont,b,bnds,cs)))
    ((fun (Stack(PushEx(ex,xs1))) bnds1 -> ex (Stack xs1) bnds1),
     Stack(PushEx (ex,xs)))
    ((i,b-1)::bnds);;
```

Fig. 5. Lexical bindings elimination in sigma2.

```
let interp2 e bnds =
  eval2 e ((fun _ _ v _ -> v), Stack Empty)
    ((fun _ _ -> print_string "error";0), Stack Empty) bnds;;
```

Fig. 6. The new top level evaluation function `interp2`.

### 4.4. Normalization of the interpreter

The normalization step aims at gathering all dynamic data inside a unique data structure, the run-time environment. For this purpose, a sum type is introduced where the type of each variant is the cartesian product of the dynamic parameter types of the functions used by the interpreter. Then, access to individual data is now performed through the environment. Thus, the interpreter will have the type:

$$\textit{static data} \rightarrow \texttt{continuation} \rightarrow \texttt{environment} \rightarrow \texttt{answer},$$

where $\texttt{continuation} =$ $\textit{static data} \rightarrow \texttt{environment} \rightarrow \texttt{answer}$.

In our example, for the sake of simplicity, we only consider one variant. The environment defined in Fig. 8 is introduced as a record type including dynamic data detected by the binding-time analysis: the two continuations (`cont` and `exs`), the environment part of closures (`cs` and `xs`), the variable bindings (`bnds`), the loop upper bound (`b`) and the integer accumulator (`v`) which stores the result of previous computations (`v1` and `v2`).

The operators of the data type `environment`, having the current environment as parameter, are introduced as follows:

- For each call, an operator computes the environment representing its dynamic arguments.
- For each call where the function is dynamic, an operator performs the function call.
- For each `if` statement, an operator returns a boolean value corresponding to the condition of the test.

For example, for the node `INT(n)`, we introduce the operator `load` of Fig. 9 building an environment containing the value of the formal arguments of the call to `cont`, i.e. `cs`, `xs` and `n` (line 1). These arguments are either static (`n`) or extracted from the input environment by pattern-matching. Concerning the node `ADD`, the operator `add` extracts static data from the environment using pattern-matching (line 2). In `sigma2`, when `b <=0`, the continuation extracted from the environment is called. For this purpose, the operator `call` is introduced to make explicit function calls over functional fields (line 3). The definition of the operator `jumpc` illustrates the use of `call` (line 4).

The normalized interpreter including the so-defined environment access functions is given in Fig. 10. The associated top level is defined in Fig. 11. All dynamic

```
∀ (e:expression) (cont:fwcont2) (cs:stack2) (exs:excont2)
  (xs:stack2) (bnds:bindings) (i:string) (b:int):
  eval2 e (cont,cs) (exs,xs) bnds
  = eval1 e
      (fun ex b -> cont cs ((fun _ b1 -> ex b1),Stack Empty) b)
      (exs xs) bnds
∧
  sigma2 i e b (cont,cs) (exs,xs) bnds
  = sigma1 i e b
      (fun ex b -> cont cs ((fun _ b1 -> ex b1),Stack Empty) b)
      (exs xs) bnds
∧
  interp2 e bnds = interp1 e bnds
```

Fig. 7. Relation between old and new interpreters.

```
type environment = {
  cont: fwcont3;          (* forward continuation *)
  ex: excont3;            (* exception continuation *)
  cs: stack3;             (* execution stack *)
  xs: stack3;             (* backup of the execution stack *)
  bnds: bindings;         (* variable bindings *)
  b: int;                 (* loop upper bound *)
  v: int;                 (* accumulator *)
}
and stack3 = Stack of (fwcont3,excont3) gstack
and fwcont3 = environment -> int
and excont3 = environment -> int;;
```

Fig. 8. The type `environment`.

data is now encapsulated inside a unique variable `env` of type `environment` which is locally accessed. Note that the top level function `interp3` must build the initial environment.

Fig. 12 states the equivalence between versions 2 and 3 of the interpreter and its auxiliary functions. Arguments of second version functions are extracted from the environment passed to their third versions. The contents of the environment of type `stack` are recursively translated through a family of functions whose code is not given here. The same holds for the transformation of continuations.

### 4.5. $\eta$-Reduction of the interpreter

By $\eta$-reduction, this step eliminates all explicit references to the environment. For this purpose, we must consider sequence and selection control structures:

- For the sequence, we introduce a composition combinator, noted ++ and defined by: `let (++) f g env=call g (call f env)`.
- For each *test* function of a selection, a combinator is defined as follows:
  `let if_`*test* `true_cnt false_cnt env=`
  if (*test* `env`) `then call true_cnt env`
  `else call false cnt env`;;

After the introduction of these combinators, $\eta$-reduction eliminates all occurrences of parameters of type `environment`. For example, the node `INT(n)` containing the composition `(cont exs (load n env))` is transformed by folding the operator ++ into

`INT(n)` → `fun cont exs env`
→ ((`load n`) + +(`cont exs`))`env`

```
1  let load n env = {cont=env.cont; ex=env.ex; pc=env.pc;
     px=env.px; bnds=env.bnds; b=env.b; v=n};;

   let loadv i env = {cont=env.cont; ex=env.ex; pc=env.pc;
     px=env.px; bnds=env.bnds; b=env.b; v=List.assoc i env.bnds};;

2  let add {cont=cont; ex=ex; pc=PushInt(v1,pc); px=px;
            bnds=bnds; b=b; v=v} =
     {cont=cont; ex=ex; pc=pc; px=px; bnds=bnds; b=b; v=v+v1};;

3  let call f x = f x;;

4  let jumpc env = call env.cont env;;
```

Fig. 9. Some environment access functions.

```
let rec eval3 = function
    INT(n) -> fun cont exs env -> cont exs (load n env)
  | ID(i) -> fun cont exs env -> cont exs (loadv i env)
  | ADD(e1,e2) ->  fun cont exs env->
      eval3 e1
        (fun exs1 env1 ->
          eval3 e2 (fun exs2 env2 -> cont exs2 (add env2))
            exs1
            (pushi env1)) exs env
  | DIV(e1,e2) -> fun cont exs env -> eval3 e2
      (fun exs2 env2 ->
        if (zerop env2) then exs2 (restore env2)
        else eval3 e1
            (fun exs1 env1 -> cont exs1 (div env1)) exs2
            (pushi env2)) exs env
  | SUM(i,b,e) -> fun cont ex env ->
      sigma3 i e (initb b (cont getex) ex env)
  | ...
and
  sigma3 i e env = if endb env then jumpc (load 0 env)
    else eval3 e
      (fun exs1 env1 ->
        sigma3 i e (rest_pushi (fun env2 -> jumpc (ret_add env2))
                               (setex exs1 env1)))
      callex (nextb i env);;
```

Fig. 10. The normalized interpreter eval3.

```
let interp3 e bnds =
  eval3 e (fun _ env -> env.v)
    (fun env -> print_string "error";0)
    {cont=(fun env -> env.v); ex=(fun env -> env.v); cs=Empty;
     xs=Empty; bnds=bnds; b=0; v=0};;
```

Fig. 11. The top level interpreter interp3.

which rewrites by $\eta$-reduction into

$$\begin{aligned}\mathtt{INT(n)} &\rightarrow \mathtt{fun\,cont\,exs}\\ &\rightarrow (\mathtt{load\,n}) + +(\mathtt{cont\,exs})\end{aligned}$$

The systematic application of these transformations leads to the code described in Fig. 13.

The relation between the two versions can easily be expressed by the equalities of Fig. 14 and is checked by unfolding the introduced combinators.

However, the new interpreter may not terminate as a consequence of the introduction of the if_*test* family of combinators. Indeed, the if construct is not strict and only reduces one of the two components of the alternative, while the combinator reduces both. In our example, for if_zerop (line 1), the *else* case performs a recursive call to eval4, but with a smaller argument: e1 replaces DIV(e1,e2) and the call can terminate (line 2). Conversely, in sigma4, the combinator if_endb recursively calls sigma4 with the same static arguments i and e: the $\eta$-reduced interpreter loops (line 3). This problem is solved in the context of partial evaluation by inserting memoization points on dynamic tests [12]. Thus, labels and jumps

```
∀ (i:string) (e:expression) (cont:excont3->fwcont3) (exs:excont3)
(env:environment):
  eval3 e cont exs env =
    eval2 e
      (dxcont2cont cont,Stack(gstack3gstack2 env.cs))
      (dexc2exc exs,Stack(gstack3gstack2 env.xs))
      env.bnds
∧
  sigma3 i e env =
    sigma2 i e env.b
      (dcont2cont env.cont,Stack(gstack3gstack2 env.cs))
      (dexc2exc env.ex,Stack(gstack3gstack2 env.xs))
      env.bnds
∧
  interp3 e bnds = interp2 e bnds
```

Fig. 12. Relation between old and new interpreters.

```
   let rec eval4 = function
       INT(n) -> fun cont exs -> (load n) ++ (cont exs)
     | ID(i) -> fun cont exs -> (loadv i) ++ (cont exs)
     | DIV(e1,e2) -> fun cont exs -> eval4 e2
1         (fun exs2 -> if_zerop
             (restore ++ exs2)
2            (pushi ++ (eval4 e1 (fun exs1 -> div ++ (cont exs1))
                             exs2))) exs
     | SUM(i,b,e) -> fun cont exs ->
          (initb b (cont getex) exs) ++ (sigma4 i e)
     | ...
   and sigma4 i e =
       if_endb
          ((load 0) ++ jumpc)
          ((nextb i) ++
             (eval4 e
                (fun exs1 ->
                  (setex exs1) ++ (rest_pushi (ret_add ++ jumpc))
3                 ++ (sigma4 i e))
                callex));;
```

Fig. 13. The $\eta$-reduced interpreter eval4.

```
∀ (i:string) (e:expression) (cont:excont3->fwcont3)
(exs:excont3) (env:environment) (bnds:bindings):
  eval4 e cont exs env = eval3 e cont exs env
∧
  sigma4 i e env = sigma3 i e env
∧
  interp4 e bnds = interp3 e bnds
```

Fig. 14. Relation between old and new interpreters.

```
  exception UndefinedLabel of int;;
  (* generator for label numbers *)
  let labNum = ref 0;;
  (* associates to a label number its code *)
  let codeTable =
    Array.init 100 (fun i _ -> raise (UndefinedLabel i));;
  (* associates to a static parameter a label number *)
  let labelTable = Hashtbl.create 100;;

  (* performs the jump to the code associated to the label *)
1 let jump lab codeTable env = Array.get codeTable lab env;;

  let memo x f =
  try
    (* jump to the code memorized in the table at index  x *)
    jump (Hashtbl.find labelTable x) codeTable
  with Not_found ->
    (* (f x) is not yet computed *)
    let lab = !labNum in           (* allocates a new label *)
    incr labNum;
    Hashtbl.add labelTable x lab;  (* associates the label to x *)
    Array.set codeTable lab (f x); (* saves the resulting code *)
    (f x);;                        (* and returns it *)
```

Fig. 15. The memoization function `memo`.

are introduced to break infinite loops. For this purpose, we replace in the interpreter of Fig. 13 `sigma4` by `sigma4'` defined as follows:

$$\texttt{let sigma4}'\ \texttt{i e} = \texttt{memo(i,e)}$$
$$(\texttt{fun(i,e)} \rightarrow \texttt{sigma4 i e})$$

The function `(memo x f)` defined in Fig. 15 breaks the loop: recursive calls to `f` on the argument `x` are delayed until the next parameter of `f`, i.e. `env` is given. At compile-time, `memo` returns a jump to a label number associated to the static parameter `x`. This number corresponds to the index of the code segment executed by jump when the environment parameter is given, i.e. at run-time (line 1).

## 4.6. Pass separation

This stage splits the interpreter into a compiler and an emulator. Until now, the interpreter is a function which, given a program and a continuation, returns a function taking an environment as parameter. It must now return a sequence of abstract instructions which is the compiled code for the program. It can be transmitted to an emulator to get the final result of the computation. In other words, the initial type of the interpreter was

$$\textit{static data} \rightarrow \texttt{continuation}$$
$$\rightarrow \texttt{environment} \rightarrow \texttt{answer}$$

We split the interpreter into `comp` and `emul` with respective types:

$\texttt{comp} : \textit{static data} \rightarrow \texttt{continuation} \rightarrow$
$\textit{label table} \times \textit{abstract code}$
$\texttt{emul} : \textit{abstract code} \rightarrow \textit{label table} \rightarrow$
$\texttt{environment} \rightarrow \texttt{answer}$

The function `comp` compiles a program into abstract code, which is contained partly in the table defining the labels. The function `emul` interprets this abstract code using the table for jumps, and updates the environment to compute the result. We have performed pass separation and extracted a compiler and an emulator from the initial interpreter.

The method used here for pass separation consists in generalizing the interpreter `interp4` with respect

```
      let gcomp e load loadv (++) (+++) pushi ret_add rest_pushi
          restore initb nextb getex setex callex pushex popex add div
          if_zerop if_endb jump jumpc cont0 ex0 =
        let labNum = ref 0 in
        let codeTable = Array.create 100 jumpc in
        let labelTable = Hashtbl.create 100 in
        let memo x f = ... in
        let rec eval5 = ...
        and sigma5 i e =
          memo (i,e) (fun (i,e) ->
          (if_endb
             ((load 0) ++ jumpc)
             ((nextb i) ++
                (eval5 e
                   (fun ex1 ->
1                    ((setex ex1) +++ (rest_pushi (ret_add ++ jumpc)))
                       ++ (sigma5 i e))
                   callex))))
        in ((eval5 e (fun _ -> cont0) ex0),code_table)
```

Fig. 16. The generic compiler `gcomp`.

to all the introduced combinators: we get the generic compiler of Fig. 16, called `gcomp` having all the combinators as formal parameters, and the same body as `interp4`. The label table is declared local to the generic compiler and so is generalized at the same time: it will thus store abstract code.

Note that the generic composition combinator $++$, used with two different instanciations, leads to two different formal parameters: $++$ and $+++$. The occurrences of $++$ within the interpreter are then renamed depending on their type (line 1).

In such a way, the type `environment` no more occurs within the synthesized type of the generic function. The type `environment`$\rightarrow$`environment` is abstracted into a polymorphic Caml type `'a`. In the same way, the type `environment`$\rightarrow$`answer` is abstracted into `'b`. In our example, we get the generic compiler `gcomp` whose type is described in Fig. 17.

It remains to notice that the type of the generic compiler defines the signature of an abstract data type. Each type parameter defines a sort, and the type of each argument defines the signature of an operator. Names must be chosen for each sort and each operator. In our example, we introduce two types: `instruction` and `control` and their associated constructors, as defined in Fig. 18. This transformation amounts to replace each combinator by an associated abstract instruction as proposed by Wand [10], but here typing separates instruction from control.

We are now able to start pass separation. The compiler of Fig. 19 is obtained by instantiating the generic compiler `gcomp` with the operators of the introduced abstract data type.

### 4.7. Abstraction of the environment

Pass separation ends with the definition of the emulator which associates to each abstract instruction its corresponding combinator. As currently defined, some combinators store lambda-expressions within the environment. For example, `rest_pushi` stores a continuation in the field `cont`. Other combinators, for example `jumpc`, apply these functions through the function `call`. However, these higher order capabilities must be eliminated. For this purpose, we consider the types of the two different uses of the generic function `call`:

$$\texttt{call} : (\texttt{environment} \rightarrow \texttt{environment}) \rightarrow \texttt{environment} \rightarrow \texttt{environment}$$

$$\texttt{call} : (\texttt{environment} \rightarrow \texttt{int}) \rightarrow \texttt{environment} \rightarrow \texttt{int}$$

```
gcomp :
  expression ->
  (int -> 'a) ->            (* load *)
  (string -> 'a) ->         (* loadv *)
  ('a -> 'b -> 'b) ->       (* ++ *)
  ('a -> 'a -> 'a) ->       (* +++ *)
  'a ->                     (* pushi *)
  'b ->                     (* getex *)
  ('b -> 'a) ->             (* setex *)
  'b ->                     (* callex *)
  'a ->                     (* pushex *)
  ('b -> 'b -> 'b) ->       (* if_zerop *)
  ... ->
  'b * 'b array             (* result: code and label table *)
```

Fig. 17. Synthesized type of the generic compiler gcomp.

```
type instruction =            and control =
    I_LOAD of int                 I_GETEX
  | I_LOADV of string           | I_SEQ of instruction * control
  | I_COMP of instruction       | I_CALLEX
           * instruction        | I_IF_ZEROP of control * control
  | I_PUSHI                     | I_IF_ENDB of control * control
  | I_SETEX of control          | I_JUMP of int
  | ...                         | ...;;
```

Fig. 18. The abstract instruction set.

```
let comp e = gcomp e
    (fun i -> I_LOAD i) (fun i -> I_LOADV i)
    (fun i1 i2 -> I_SEQ (i1,i2))    (* ++ *)
    (fun i1 i2 -> I_COMP (i1,i2))   (* +++ *)
    I_PUSHI I_GETEX (fun i -> I_SETEX i) I_CALLEX I_PUSHEX
    ...;;
```

Fig. 19. The compiler comp.

The types environment→environment and environment→int have been, respectively, abstracted into instruction and control by the previous stage. Applying the same idea to call leads to the two following types:

instruction → environment
  → environment

control → environment → int

which are in fact the types of the functions associating to an abstract instruction its corresponding combinator. Thus, abstracting the environment can be obtained by redefining call as a call to the emulator. We are now ready to define a generic version of the emulator parameterized by the environment access and update functions get_x and set_x where x is an environment field. This emulator is described by Fig. 20. It encapsulates the definition of all the combinators and defines the two functions ins_emul and ctr_emul. Note the introduction of the two variants call (line 1) and call1 (line 2) corresponding to the two instanciations of the generic function call. Furthermore, the combinators no more access directly to environment fields: they use the abstract environment access functions get_x and set_x.

```
let gemul ins table get_cont get_ex get_cs get_xs get_bnds get_b
    get_v set_cont set_ex set_cs set_xs set_bnds set_b set_v =

  let rec load n env = set_v n env
  and jumpc env = call (get_cont env) env
  and (++) f g x = call g (call1 f x)
  and (+++) f g x = call1 g (call1 f x)
  and if_zerop tcnt fcnt env =
    if (zerop env) then call tcnt env else call fcnt env
  and jump lab env = call (Array.get table lab) env
1 and call ins = ctr_emul ins
2 and call1 ins = ins_emul ins
  and ...
  and ins_emul = function                and ctr_emul = function
      I_LOAD i -> load i                     I_GETEX -> getex
    | I_LOADV i -> loadv i                 | I_SEQ(i1,i2) -> i1++i2
    | I_COMP(i1,i2) -> i1 +++ i2           | I_CALLEX -> callex
    | I_PUSHI -> pushi                     | I_IF_ZEROP(i1,i2) ->
    | I_INITB(i,c,e) -> initb i c e        | I_JUMP l -> jump l
    | I_PUSHEX -> pushex                   | ...
    | ...                                in ctr_emul ins;;
```

Fig. 20. The generic emulator `gemul`.

```
gemul : control -> control array ->
        ('a -> control) ->                 (* get_cont *)
        ('a -> control) ->                 (* get_ex *)
        ... ->
        (control -> 'a -> 'a) ->           (* set_cont *)
        (control -> 'a -> 'a) ->           (* set_ex *)
        ... ->
        'a -> int
```

Fig. 21. The synthesized type of `gemul`.

The synthesized type of the generic emulator parameterized by environment access functions, given in Fig. 21, produces the type of the abstract environment. Functional fields are now typed by `control`.

Thus, the concrete run-time `environment` type is defined in Fig. 22. The type of its fields is obtained from the signature of the generic emulator.

Finally, Fig. 23 defines the emulator as an instanciation of the generic emulator with run-time environment access functions.

The characteristic equation of pass seperation can now be stated. Fig. 24 builds the initial environment and defines the interpreter as a composition of compilation and emulation phases.

```
type environment = {
    cont: control;
    ex: control;
    cs: (control,control) gstack;
    xs: (control,control) gstack;
    bnds: bindings;
    b: int;
    v: int;
  };;
```

Fig. 22. The run-time environment.

```
let get_cont env = env.cont;;
and get_ex env = env.ex;;
...
let set_cont cont env = {env with cont=cont};;
and set_ex ex env = {env with ex=ex};;
...
let emul ins table = gemul ins table
    get_cont get_ex get_cs get_xs get_bnds get_b get_v set_cont
    set_ex set_cs set_xs set_bnds set_b set_v;;
```

Fig. 23. The emulator `emul`.

```
let env0 bnds =
  {cont=I_CONT0; ex=I_EX0; cs=Empty; xs=Empty; bnds=bnds;
   b=0; v=0};;

let interp e bnds =
  let (ins,table) = comp e in emul ins table (env0 bnds);;
```

Fig. 24. Composing compiler and emulator.

### 4.8. Summary

Within a functional framework, the writing of the first interpreter uses lexical bindings, recursivity and higher order functions. Then, we do without these high-level capabilities to write the new versions of the same interpreter. For that, we provide an implementation of the previous functionalities using lower level constructs.

The continuation passing style is used to express the control flow uniformly. However, the power of lexical binding still allows a high-level description of the data flow similar to recursivity for the control flow. This mechanism is eliminated to obtain a lower level machine code, thus revealing the abstract machine internal structure. This stage is completed by the introduction of an execution environment encapsulating dynamic data. The abstract instructions and their emulator are extracted through pass separation.

To sum up, the derivation process follows the next steps:
- writing of a first interpreter,
- binding-time analysis and dynamic variables annotation,
- lexical binding elimination for dynamic variables,
- normalization of the interpreter and construction of the abstract data type `environment`,
- elimination of the environment through $\eta$-reduction,
- pass separation and extraction of the compiler,
- abstraction of the environment and extraction of the emulator.

## 5. Conclusion

We have presented a process for deriving an abstract machine from a source language whose operational semantics is defined by an interpreter written in a high-level functional language. It is based on a progressive transformation of this interpreter into a compiler and an emulator of abstract code. The initial interpreter is written using high-level constructions. Then it is transformed gradually so that it does not rely on the features of the implementation language. It leads to the emergence of data structures which will be the heart of the abstract machine.

Pass separation is then applied to the interpreter, aiming at splitting it into two complementary activities: on the one hand the compilation of the source language that produces the intermediate code, and on the other hand the interpretation of this intermediate code. Thus, we get a compiler and an emulator of abstract code.

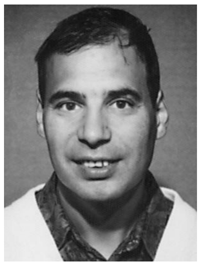
The formalism used in this paper is functional and higher order continuation-based as Wand's [10]. The guideline of the transformations is similar but the details differ in our framework as Wand's work is untyped.

We now plan to study the automatization of the presented study and its correctness. The first point concerns the choice of the functional language. The Caml type system is not powerful enough to manage typed variable bindings or to guarantee the coherence of the variants of the sum data types: a system with dependent types is necessary. The second point consists in defining an environment to make easier the mechanization of program transformations. At least, these transformations must be validated. These three requirements can be fulfilled using a proof assistant such as Coq [15].

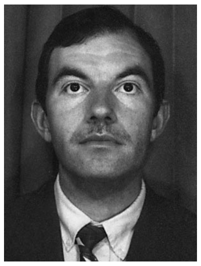
## References

[1] K. Arnold, J. Gosling, The Java Programming Language, Addison-Wesley, Reading, MA, 1996.
[2] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1987.
[3] U. Jørring, W. Scherlis, Compilers and staging transformations, in: Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida, 1986, pp. 86–96.
[4] J. Hannan, D. Miller, From operational semantics to abstract machines: preliminary results, in: M. Wand (Ed.), Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, ACM, New York, 1990, pp. 323–332.
[5] J. Hannan, Staging transformations for abstract machines, Partial Evaluation and Semantic Based Program Manipulation 26 (1991) 130–141.
[6] P. Sestoft, Deriving a lazy abstract machine, J. Funct. Programming 7 (1997) 3.
[7] S. Diehl, Semantics-directed generation of compilers and abstract machines, Ph.D. Thesis, University Saarbrücken, Germany, 1996.
[8] S. Diehl, Transformations of evolving algebras, in: LIRA'97 (VIII International Conference on Logic and Computer Science), Novi Sad, Yugoslavia, September 1997, pp. 51–57.
[9] Y. Gurevich, Evolving algebras: an introductory tutorial, Bull. Eur. Assoc. Theoret. Comput. Sci. 43 (1991) 264–284.
[10] M. Wand, Deriving target code as a representation of continuation semantics, ACM Trans. Programming Languages Syst. 4 (3) (1982) 496–517.
[11] X. Leroy, The Objective Caml System Release 1.03, INRIA, October 1996.
[12] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[13] G.D. Plotkin, Call-by-name, call-by-value and the λ-calculus, Theoret. Comput. Sci. 1 (1975) 125–159.
[14] J.-C. Reynolds, Definitional interpreters for higher order programming languages, in: ACM Conference Proceedings, 1972, pp. 717–740.
[15] G. Huet, G. Kahn, C. Paulin-Mohring, The Coq proof assistant —a tutorial, version 6.1, Tech. Rep. 204, INRIA, August 1997.

**Frédéric Cabestre** is a Ph.D. student working on the synthesis of abstract machines from operational semantics. His work is part of a project aiming at combining in the same framework objects, logic and concurrency. His main interests include functional programming, compilation techniques and partial evaluation.



**Christian Percebois** is professor of computer science at the University of Toulouse III, France, since 1992. His research interests include parallelism, functional, logic and object-oriented programming and abstract machines. Since 1996, he is mainly interested by multiset rewriting techniques in order to coordinate concurrent objects.



**Jean-Paul Bodeveix** is an old student of the Ecole Normale Superieure of Cachan and received a Ph.D. of Computer Science from the University of Paris-Sud in 1989. He is now an assistant professor of computer science at the University of Toulouse III, France. Current research interests include concurrency, object-oriented programming, logic programming, formal specifications and validation.