Chapter 16

Compiling ASL to an abstract machine code

In order to fully take advantage of the static typing of ASL programs, we have to:

- either write a new interpreter without type tests (difficult, because we used pattern-matching in order to realize type tests);
- or design an untyped machine and produce code for it.

We choose here the second solution: it will permit us to give some intuition about the compiling process of functional languages, and to describe a typical execution model for (strict) functional languages. The machine that we will use is a simplified version the *Categorical Abstract Machine* (CAM, for short).

We will call CAM our abstract machine, despite its differences with the original CAM. For more informations on the CAM, see [8, 26].

16.1 The Abstract Machine

The execution model is a *stack machine* (i.e. a machine using a stack). In this section, we define in Caml the *states* of the CAM and its instructions.

A state is composed of:

- a *register* (holding values and environments),
- a *program counter*, represented here as a list of instructions whose first element is the current instruction being executed,
- and a *stack* represented as a list of code addresses (instruction lists), values and environments.

The first Caml type that we need is the type for CAM instructions. We will study later the effect of each instruction.

```
#type instruction =
#  Quote of int              (* Integer constants *)
```

```
#| Plus | Minus              (* Arithmetic operations *)
#| Divide | Equal | Times
#| Nth of int                 (* Variable accesses *)
#| Branch of instruction list * instruction list
#                             (* Conditional execution *)
#| Push                       (* Pushes onto the stack *)
#| Swap                       (* Exch. top of stack and register *)
#| Clos of instruction list   (* Builds a closure with the current environment *)
#| Apply                      (* Function application *)
#;;
Type instruction defined.
```

We need a new type for semantic values since instruction lists have now replaced abstract syntax trees. The semantic values are merged in a type `object`. The type `object` behaves as data in a computer memory: we need higher-level information (such as type information) in order to interpret them. Furthermore, some data do not correspond to anything (for example an environment composed of environments represents neither an ASL value nor an intermediate data in a legal computation process).

```
#type object = Constant of int
#            | Closure of object * object
#            | Address of instruction list
#            | Environment of object list
#;;
Type object defined.
```

The type `state` is a product type with mutable components.

```
#type state = {mutable Reg: object;
#              mutable PC: instruction list;
#              mutable Stack: object list}
#;;
Type state defined.
```

Now, we give the *operational semantics* of CAM instructions. The effect of an instruction is to change the state configuration. This is what we describe now with the `step` function. Code executions will be arbitrary iterations of this function.

```
#exception CAMbug of string;;
Exception CAMbug defined.
```

```
#exception CAM_End of object;;
Exception CAM_End defined.
```

```
#let step state = match state with
#  {Reg=_; PC=Quote(n)::code; Stack=s} ->
#                 state.Reg <- Constant(n); state.PC <- code
#
```

```
#| {Reg=Constant(m); PC=Plus::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(n+m); state.Stack <- s;
#                state.PC <- code
#
#| {Reg=Constant(m); PC=Minus::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(n-m); state.Stack <- s;
#                state.PC <- code
#
#| {Reg=Constant(m); PC=Times::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(n*m); state.Stack <- s;
#                state.PC <- code
#
#| {Reg=Constant(m); PC=Divide::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(n/m); state.Stack <- s;
#                state.PC <- code
#
#| {Reg=Constant(m); PC=Equal::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(if n=m then 1 else 0);
#                state.Stack <- s; state.PC <- code
#
#| {Reg=Constant(m); PC=Branch(code1,code2)::code; Stack=r::s} ->
#                state.Reg <- r;
#                state.Stack <- Address(code)::s;
#                state.PC <- (if m=0 then code2 else code1)
#
#| {Reg=r; PC=Push::code; Stack=s} ->
#                state.Stack <- r::s; state.PC <- code
#
#| {Reg=r1; PC=Swap::code; Stack=r2::s} ->
#                state.Reg <- r2; state.Stack <- r1::s;
#                state.PC <- code
#
#| {Reg=r; PC=Clos(code1)::code; Stack=s} ->
#                state.Reg <- Closure(Address(code1),r);
#                state.PC <- code
#
#| {Reg=_; PC=[]; Stack=Address(code)::s} ->
#                state.Stack <- s; state.PC <- code
#
#| {Reg=v; PC=Apply::code;
#          Stack=Closure(Address(code1),Environment(e))::s} ->
#                state.Reg <- Environment(v::e);
#                state.Stack <- Address(code)::s;
#                state.PC <- code1
#
```

```
#| {Reg=v; PC=[]; Stack=[]} ->
#                raise (CAM_End v)
#| {Reg=_; PC=(Plus|Minus|Times|Divide|Equal)::code; Stack=_::_} ->
#                raise (CAMbug "IllTyped")
#
#| {Reg=Environment(e); PC=Nth(n)::code; Stack=_} ->
#                state.Reg <- (try nth n e
#                              with Failure _ -> raise (CAMbug "IllTyped"));
#                state.PC <- code
#| _ -> raise (CAMbug "Wrong configuration")
#;;
step : state -> unit = <fun>
```

We may notice that the empty code sequence denotes a (possibly final) *return* instruction.

We could argue that pattern-matching in the Camlstep function implements a kind of dynamic typechecking. In fact, in a concrete (low-level) implementation of the machine (expansion of the CAM instructions in assembly code, for example), these tests would not appear. They are useless since we trust the typechecker and the compiler. So, any execution error in a real implementation comes from a *bug* in one of the above processes and would lead to memory errors or illegal instructions (usually detected by the computer's operating system).

16.2 Compiling ASL programs into CAM code

We give in this section a compiling function taking the abstract syntax tree of an ASL expression and producing CAM code. The compilation scheme is very simple:

- the code of a constant is `Quote`;
- a variable is compiled as an access to the appropriate component of the current environment (`Nth`);
- the code of a conditional expression will save the current environment (`Push`), evaluate the condition part, and, according to the boolean value obtained, select the appropriate code to execute (`Branch`);
- the code of an application will also save the environment on the stack (`Push`), execute the function part of the application, then exchange the functional value and the saved environment (`Swap`), evaluate the argument and, finally, apply the functional value (which is at the top of the stack) to the argument held in the register with the `Apply` instruction;
- the code of an abstraction simply consists in building a closure representing the functional value: the closure is composed of the code of the function and the current environment.

Here is the compiling function:

```
#let rec code_of = function
#  Const(n) -> [Quote(n)]
#| Var n -> [Nth(n)]
#| Cond(e_test,e_t,e_f) ->
#        Push::(code_of e_test)
#      @[Branch(code_of e_t, code_of e_f)]
#| App(e1,e2) -> Push::(code_of e1)
#              @[Swap]@(code_of e2)
#              @[Apply]
#| Abs(_,e) -> [Clos(code_of e)];;
code_of : asl -> instruction list = <fun>
```

A global environment is needed in order to maintain already defined values. Any CAM execution will start in a state whose register part contains this global environment.

```
#let init_CAM_env =
#  let basic_instruction = function
#        "+" -> Plus
#      | "-" -> Minus
#      | "*" -> Times
#      | "/" -> Divide
#      | "=" -> Equal
#      | s -> raise (CAMbug "Unknown primitive")
#  in map (function s ->
#          Closure(Address[Clos(Push::Nth(2)
#                                ::Swap::Nth(1)
#                                ::[basic_instruction s])],
#                  Environment []))
#        init_env;;
init_CAM_env : object list =
 [Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Plus]], Environment []);
 Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Minus]], Environment []);
Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Times]], Environment []);
Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Divide]], Environment []);
Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Equal]], Environment [])]
```

```
#let global_CAM_env = ref init_CAM_env;;
global_CAM_env : object list ref =
 ref
  [Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Plus]], Environment []);
  Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Minus]], Environment []);
 Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Times]], Environment []);
Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Divide]], Environment []);
Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Equal]], Environment [])]
```

As an example, here is the code for some ASL expressions.

```
#code_of (expression(lexer(stream_of_string "1;")));;
- : instruction list = [Quote 1]
```

```
#code_of (expression(lexer(stream_of_string "+ 1 2;")));;
```

```
- : instruction list =
 [Push; Push; Nth 6; Swap; Quote 1; Apply; Swap; Quote 2; Apply]
#code_of (expression(lexer(stream_of_string "(\\x.x) ((\\x.x) 0);")));;
- : instruction list =
 [Push; Clos [Nth 1]; Swap; Push; Clos [Nth 1]; Swap; Quote 0; Apply; Apply]
#code_of (expression(lexer(stream_of_string
#              "+ 1 (if 0 then 2 else 3 fi);")));;
- : instruction list =
 [Push; Push; Nth 6; Swap; Quote 1; Apply; Swap; Push; Quote 0;
  Branch ([Quote 2], [Quote 3]); Apply]
```

16.3 Execution of CAM code

The main function for executing compiled ASL manages the global environment until execution has succeeded.

```
#let run (Decl(s,e)) =
#  (* TYPING *)
#    reset_vartypes();
#    let tau =
#        try asl_typing_expr !global_typing_env e
#        with TypeClash(t1,t2) ->
#              let vars=vars_of_type(t1) @ vars_of_type(t2) in
#              print_string "ASL Type clash between ";
#              print_type_scheme (Forall(vars,t1));
#              print_string " and ";
#              print_type_scheme (Forall(vars,t2));
#              raise (Failure "ASL typing")
#           | Unbound s -> raise (TypingBug ("Unbound: "^s)) in
#    let sigma = generalise_type (!global_typing_env,tau) in
#  (* PRINTING TYPE INFORMATION *)
#    print_string "ASL Type of ";
#    print_string s; print_string " is ";
#    print_type_scheme sigma; print_newline();
#  (* COMPILING *)
#    let code = code_of e in
#    let state = {Reg=Environment(!global_CAM_env); PC=code; Stack=[]} in
#  (* EXECUTING *)
#    let result = try while true do step state done; state.Reg
#                 with CAM_End v -> v in
#  (* UPDATING ENVIRONMENTS *)
#    global_env := s::!global_env;
#    global_typing_env := sigma::!global_typing_env;
#    global_CAM_env := result::!global_CAM_env;
#  (* PRINTING RESULT *)
#    (match result
#     with Constant(n) -> print_int n
#        | Closure(_,_) -> print_string "<funval>"
#        | _ -> raise (CAMbug "Wrong state configuration"));
#    print_newline();;
run : top_asl -> unit = <fun>
```

Now, let us run some examples:

```
#(* Reinitializing environments *)
#global_env:=init_env;
#global_typing_env:=init_typing_env;
#global_CAM_env:=init_CAM_env;
#();;
- : unit = ()

#run (parse_top "1;");;
ASL Type of it is Number
1
- : unit = ()

#run (parse_top "+ 1 2;");;
ASL Type of it is Number
3
- : unit = ()

#run (parse_top "(\\f.(\\x.f x)) (\\x. + x 1) 3;");;
ASL Type of it is Number
4
- : unit = ()
```

We may now introduce the Z fixpoint combinator as a predefined function `fix`.

```
#begin
#  global_env:="fix"::init_env;
#  global_typing_env:=
#    (Forall([1],
#            Arrow(Arrow(TypeVar{Index=1;Value=Unknown},
#                        TypeVar{Index=1;Value=Unknown}),
#                  TypeVar{Index=1;Value=Unknown})))
#   ::init_typing_env;
#  global_CAM_env:=
#   (match code_of (expression(lexer(stream_of_string
#           "\\f.((\\x.f(\\z.(x x)z)) (\\x.f(\\z.(x x)z)));")))
#    with [Clos(C)] -> Closure(Address(C), Environment [])
#       | _ -> raise (CAMbug "Wrong code for fix"))
#   ::init_CAM_env
#end;;
```

```
Toplevel input:
>    with [Clos(C)] -> Closure(Address(C), Environment [])
>               ^
Warning: the variable C starts with an upper case letter in this pattern.
- : unit = ()


#run (parse_top
#    "let fact be fix (\\f.(\\n. if = n 0 then 1
#                          else * n (f (- n 1))
#                          fi));");;
ASL Type of fact is (Number -> Number)
<funval>
- : unit = ()

#run (parse_top
#  "let fib be fix (\\f.(\\n. if = n 1 then 1
#                            else if = n 2 then 1
#                                 else + (f(- n 1)) (f(- n 2))
#                                 fi
#                            fi));");;
ASL Type of fib is (Number -> Number)
<funval>
- : unit = ()

#run (parse_top "fact 8;");;
ASL Type of it is Number
40320
- : unit = ()

#run (parse_top "fib 9;");;
ASL Type of it is Number
34
- : unit = ()
```

It is of course possible (and desirable) to introduce recursion by using a specific syntactic construct, special instructions and a dedicated case to the compiling function. See [26] for efficient compilation of recursion, data structures etc.

Exercise 16.1 *Interesting exercises for which we won't give solutions consist in enriching according to your taste the ASL language. Also, building a standalone ASL interpreter is a good exercise in modular programming.*