IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Let's practice compilation – correction of the training

Langages et logiques – LaLog

Correction

Before starting the true project, we are going to start with a guided sequence. We are going to work with two languages $\mathcal{L}_1$ and $\mathcal{L}_2$. $\mathcal{L}_1$ is a simplified language to express basic boolean formula and $\mathcal{L}_2$ is a simplified low level language for a simple machine.

1 The low level language

As we do not have real hardware to execute $\mathcal{L}_2$, we are going to define an abstract machine to support the execution of a $\mathcal{L}_2$ program.

This abstract machine can be described by:

- it manipulates only one bit
- it has three registers R_A, R_B and R_C (of one bit)
- it has a memory of 16 bits (M_0 to M_{15}), this memory is initialized before running a program
- it supports the following operations
  - setting a register to either 0 or 1, $\mathtt{S}xb$ sets the register R_x to the value b
  - loading from memory to a register, $\mathtt{L}ix$ sets the register R_x to the value of M_i
  - computing the nand of two registers and putting the result in a register, $\mathtt{N}xyz$ puts R_x `nand` R_y in R_z where `nand` is defined by $1\,\mathtt{nand}\,1 = 0$ and $0\,\mathtt{nand}\,b = b\,\mathtt{nand}\,0 = 1$.
- a program is a sequence of the previous operations, running a program consists in executing its operation one by one until the last one

The informal semantics described above can be expressed mathematically using a small step semantics. Executing a program denoted $\mathcal{P}$ in the context of a memory $\mathcal{M}$ is done step by step by executing each of its operation while maintaining the values of the three registers denoted $\mathcal{R}$.

It is expressed by a set of rules of the form $\mathcal{M} \vdash \mathcal{R}, \mathcal{P} \rightarrow \mathcal{R}', \mathcal{P}'$ where $\mathcal{R}'$ is the resulting value of the registers and $\mathcal{P}'$ the remaining program.

Accessing the value M_i in memory is done by $\mathcal{M}(i)$ (we suppose that the memory is a function from $[\![0, 15]\!]$ to $[\![0, 1]\!]$). Similarly, accessing the value of the register R_x is done by $\mathcal{R}(x)$ (we suppose that the register is a function from $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ to $[\![0, 1]\!]$) and the register can be updated by the syntax $\mathcal{R}\{x \mapsto b\}$ meaning that $\mathcal{R}\{x \mapsto b\}(x) = b$ and if y is different from x, $\mathcal{R}\{x \mapsto b\}(y) = \mathcal{R}(y)$.

The semantics of the first operation is expressed by the rule:

$$(\mathtt{S})\ \mathcal{M} \vdash \mathcal{R}, \mathtt{S}xb\ \mathcal{P} \rightarrow \mathcal{R}\{x \mapsto b\}, \mathcal{P}$$

Exercise A

▷ **Question A.1:**
Explain this rule using plain words.

In the context of a memory $\mathcal{M}$, if the registers are $\mathcal{R}$ and first operation of a program is $\mathtt{S}xb$ while the rest is $\mathcal{P}$, the machine returns the updated register where x is equal to b and the other registers are unchanged and the rest of the program.

▷ **Question A.2:**
Give the rules for the semantics of load and nand.

$$\begin{cases} (\mathtt{L})\ \mathcal{M} \vdash \mathcal{R}, \mathtt{L}ix\ \mathcal{P} \rightarrow \mathcal{R}\{x \mapsto \mathcal{M}(i)\}, \mathcal{P} \\ (\mathtt{N})\ \mathcal{M} \vdash \mathcal{R}, \mathtt{N}xyz\ \mathcal{P} \rightarrow \mathcal{R}\{z \mapsto \mathcal{R}(x)\ \mathtt{nand}\ \mathcal{R}(y)\}, \mathcal{P} \end{cases}$$

▷ **Question A.3:**
Propose an OCaml implementation of the machine by at least defining a type `operation` for its operations and a function `step` expressing their semantics.

Encodings and data structures

To implement our machine, we have to choose some encodings:

- the bit the machine manipulate,
- the register identifiers,
- the memory locations.

For each of this elements, one may define an OCaml type to be able to define the operation type. In the code that follows, we do not give the concrete implementations of these types. It is an interface (a mli file) and the types remain abstracts.

```
(* Bit value *)
type bit
(* Register identifier *)
```

```
type reg_id
(* Memory location *)
type mem_id
(* Operations of the bit machine *)
type operation =
  | Set of reg_id * bit
  | Load of mem_id * reg_id
  | Nand of reg_id * reg_id * reg_id
```

Then, one has to implement the step function. For this, we need to choose data types for:

- A state of the machine which is the value associated to each register,
- The memory associating a location to a value,
- A program containing operations.

```
(* The registers *)
type state
(* The memory *)
type memory
(* A program *)
type program
```

Semantics interface

Once, they are defined, one can give the type of the `step` function:

```
(* The small step transition *)
val step: memory -> state * program -> (state * program) option
```

Here, we choose to return an option to encode the fact that an error may happen with the value `None`.
To be able to use this function, the user needs to be able to create a memory, an initial state and a program. And to be able to follow the reduction, it needs printing functions. To satisfy the specifications, the easier is to have a function creating the memory from a list of values and a program from a list of operations. This requires to be able to create bit values, register identifiers and memory locations.

```
(* Bit values *)
val z: bit
val o: bit
(* Register identifiers *)
val a: reg_id
val b: reg_id
val c: reg_id
(* A new memory location *)
val mem: int -> mem_id
(* An initial register state *)
val initial_state: state
(* A new memory *)
```

```
val new_memory: bit list -> memory
(* A new program *)
val new_program: operation list -> program
(* Is the program finished *)
val is_finished: program -> bool
(* Printing functions *)
val string_of_bit: bit -> string
val string_of_state: state -> string
val string_of_memory: memory -> string
val string_of_program: program -> string
(* Getting the result *)
val get_result: state -> bit
```

An example of usage

On the basis of this interface, one can propose the following usage:

```
let print_state_program (state,program) =
 Printf.printf "(%s, %s)\n" (string_of_state state) (string_of_program program)
let step_and_print memory program =
 let rec step_rec (state,program) =
  if is_finished program then
   Printf.printf "final result: %s\n" (string_of_bit (get_result state))
  else
   begin
    print_string " => ";
    match step memory (state,program) with
    | None -> print_endline "error"
    | Some (state',program') ->
     print_state_program (state',program');
     step_rec (state',program')
   end
 in
 print_state_program (initial_state,program);
 step_rec (initial_state,program)
let () =
 let initial_memory = new_memory [z;o;z;z;z;z;z;z;z;z;z;z;z;z;z;z] in
 let the_program = new_program [Load(mem 1,b); Set(c,o); Nand(b,c,a)] in
 print_newline ();
 step_and_print initial_memory the_program
```

that when executed prints:

```
([| 0 | 0 | 0 |], L 1 B; S C 1; N B C A)
 => ([| 0 | 1 | 0 |], S C 1; N B C A)
 => ([| 0 | 1 | 1 |], N B C A)
 => ([| 0 | 1 | 1 |], )
final result: 0
```

Semantics implementation

The last bit is the implementation:

```
(* Bit value *)
type bit = Zero | One
let z = Zero
let o = One
let nand x y = match (x,y) with
  | (One,One) -> Zero
  | _ -> One
let string_of_bit = function
  | Zero -> "0"
  | One -> "1"
(* Register identifier *)
type reg_id = A | B | C
let a = A
let b = B
let c = C
let string_of_reg_id = function
  | A -> "A"
  | B -> "B"
  | C -> "C"
(* Memory location *)
type mem_id = int
let mem x = x
type state = { ra: bit; rb: bit; rc: bit; }
let initial_state = { ra=z; rb=z; rc=z }
let get state = function
  | A -> state.ra
  | B -> state.rb
  | C -> state.rc
let update state reg value = match reg with
  | A -> { state with ra = value}
  | B -> { state with rb = value}
  | C -> { state with rc = value}
let get_result { ra; _ } = ra
let string_of_state state =
 Printf.sprintf "[| %s | %s | %s |]" (string_of_bit state.ra)
   (string_of_bit state.rb) (string_of_bit state.rc)
type memory = bit array
let new_memory bit_list =
 let size = List.length bit_list in
 match size with
  | 16 -> Array.of_list bit_list
  | i when i > 16 -> failwith "invalid memory initialization"
  | i when i < 16 -> Array.init 16 (fun n -> if n <= i then List.nth bit_list (n-1) else z)
  | _ -> failwith "Not reachable"
let string_of_memory memory =
```

```
  Array.mapi (fun i b -> Printf.sprintf "%i: %s" i (string_of_bit b)) memory
  |> Array.to_list
  |> String.concat ","
  |> Printf.sprintf "[|%s|]"
(* Operations of the bit machine *)
type operation =
  | Set of reg_id * bit
  | Load of mem_id * reg_id
  | Nand of reg_id * reg_id * reg_id
let string_of_op = function
  | Set(x,b) -> Printf.sprintf "S %s %s" (string_of_reg_id x) (string_of_bit b)
  | Load(i,x) -> Printf.sprintf "L %i %s" i (string_of_reg_id x)
  | Nand(x,y,z) ->
   Printf.sprintf "N %s %s %s" (string_of_reg_id x)
     (string_of_reg_id y) (string_of_reg_id z)
type program = operation list
let new_program x = x
let is_finished = function [] -> true | _ -> false
let string_of_program program = String.concat "; " (List.map string_of_op program)
let step memory = function
  | (_, []) -> None
  | (state, Set(x,b)::rest) -> Some (update state x b, rest)
  | (state, Load(i,x)::rest) -> Some (update state x memory.(i), rest)
  | (state, Nand(x,y,z)::rest) ->
   Some (update state z (nand (get state x) (get state y)), rest)
```

Another simpler (?) implementation

A very simple version follows. Beware it is very weakly typed and thus not very OCaml... It would be more idiomatic to have different types for the registers id, the bit values and the memory locations as in the previous implementation. Notice also that the code is fragile as we do not verify any limit.

```
let nand x y = match (x,y) with
  | (1,1) -> 0
  | _ -> 1
type operation =
  | Set of int * int
  | Load of int * int
  | Nand of int * int * int
let set regs x v =
 match regs, x with
  | (_,b,c), 1 -> (v,b,c)
  | (a,_,c), 2 -> (a,v,c)
  | (a,b,_), 3 -> (a,b,v)
  | _ -> failwith "wrong register id"
let get regs x =
 match regs, x with
```

```
  | (a,_,_), 1 -> a
  | (_,b,_), 2 -> b
  | (_,_,c), 3 -> c
  | _ -> failwith "wrong register id"
let step memory = function
  | (_, []) -> None
  | (regs, Set(x,b)::p) -> Some (set regs x b, p)
  | (regs, Load(i,x)::p) -> Some (set regs x memory.(i), p)
  | (regs, Nand(x,y,z)::p) -> Some (set regs z (nand (get regs x) (get regs y)), p)
```

2 The high level language and its compilation

Our high level language allows to define simple boolean expressions containing variables. The language is defined by:

$$F ::= \texttt{true} \mid \texttt{false} \mid x \mid F \wedge F \mid F \vee F$$

We suppose that this language AST is implemented as follows:

```
type t =
  | Var of string
  | Bool of bool
  | And of t * t
  | Or of t * t
```

During typing, a formula containing more than 16 variables is rejected and we build a mapping from variable names to memory locations denoted M. Therefore, if we denote R the register that should contain the result (the default being $\mathsf{R_A}$), the compilation rules have the form:

$$\mathsf{M}, \mathsf{R} \vdash \text{boolean expression} \rightsquigarrow \text{instructions sequence}$$

Exercise B

▷ **Question B.1:**
Give all the compilation rules.

$$(1)\ \mathsf{M}, \mathsf{R} \vdash \texttt{true} \rightsquigarrow \mathsf{SR1} \qquad (2)\ \mathsf{M}, \mathsf{R} \vdash \texttt{false} \rightsquigarrow \mathsf{SR0}$$

$$(3)\ \frac{x \in dom(\mathsf{M})}{\mathsf{M}, \mathsf{R} \vdash \texttt{Var}\ x \rightsquigarrow \mathsf{LM}(x)\mathsf{R}}$$

$$(4)\ \frac{\mathsf{M}, \mathsf{R_A} \vdash F_1 \rightsquigarrow is_1 \qquad \mathsf{M}, \mathsf{R_B} \vdash F_2 \rightsquigarrow is_2}{\mathsf{M}, \mathsf{R} \vdash \texttt{And}(F_1, F_2) \rightsquigarrow is_1 is_2 \mathsf{NR_A R_B RNRR}}$$

$$(5)\ \frac{\mathsf{M}, \mathsf{R_A} \vdash F_1 \rightsquigarrow is_1 \qquad \mathsf{M}, \mathsf{R_B} \vdash F_2 \rightsquigarrow is_2}{\mathsf{M}, \mathsf{R} \vdash \texttt{Or}(F_1, F_2) \rightsquigarrow is_1 is_2 \mathsf{NR_A R_A R_A NR_B R_B R_B NR_A R_B R}}$$

Notice that our compilation scheme is unsafe because $F_1 \wedge (F_2 \wedge (F_3 \wedge ...))$ requires a register

for F_1, a second for F_2 and the third for F_3. None register is left for the rest of the calculus. We are not going to explore a solution to the problem but notice that it means the compiler should limit the depth of a formula to be able to compute its value.

▷ **Question B.2:**
Give the compilation of $(\texttt{true} \vee x) \wedge (y \vee \texttt{false})$**.**

$\texttt{SR}_\texttt{A}1\ \texttt{LM}_0\texttt{R}_\texttt{B}\ \texttt{NR}_\texttt{A}\texttt{R}_\texttt{A}\texttt{R}_\texttt{A}\ \texttt{NR}_\texttt{B}\texttt{R}_\texttt{B}\texttt{R}_\texttt{B}\ \texttt{NR}_\texttt{A}\texttt{R}_\texttt{B}\texttt{R}_\texttt{A}\ \texttt{LM}_1\texttt{R}_\texttt{A}\ \texttt{SR}_\texttt{B}0\ \texttt{NR}_\texttt{A}\texttt{R}_\texttt{A}\texttt{R}_\texttt{A}\ \texttt{NR}_\texttt{B}\texttt{R}_\texttt{B}\texttt{R}_\texttt{B}\ \texttt{NR}_\texttt{A}\texttt{R}_\texttt{B}\texttt{R}_\texttt{B}\ \texttt{NR}_\texttt{A}\texttt{R}_\texttt{B}\texttt{R}_\texttt{A}\ \texttt{NR}_\texttt{A}\texttt{R}_\texttt{A}\texttt{R}_\texttt{A}$

▷ **Question B.3:**
Propose an OCaml implementation of compilation of a formula to the one bit machine.

```
open FormulaAst
let compile mem formula =
 let rec compile_rec register = function
  | Bool true  -> [Set(register,1)]
  | Bool false -> [Set(register,0)]
  | Var s      -> [Load(List.assoc s mem, register)]
  | And(f1,f2) ->
   let is1 = compile_rec 1 f1 in
   let is2 = compile_rec 2 f2 in
   is1 @ is2 @ [Nand(1,2,register);Nand(register,register,register)]
  | Or(f1,f2)  ->
   let is1 = compile_rec 1 f1 in
   let is2 = compile_rec 2 f2 in
   is1 @ is2 @ [Nand(1,1,1);Nand(2,2,2);Nand(1,2,register)]
 in
 compile_rec 1 formula
```