The interpreter of a simple functional language

Letterio Galletta

A high level interpreter

- A high level interpreter directly executes code instructions without any translation steps
- These interpreters are best suited to build Domain Specific Languages (DSLs) rather the general purpose programming languages
- We prefer simplicity and low-cost implementation to execution efficiency
- Three things to consider when implementing an interpreter:
  1. How to store data
  2. How to track symbols, e.g., program variables
  3. How to execute instructions

Note: there is a very strong relationship between a language semantics and a language interpreter!

The interpreter of fun

- A simple functional language: integers, booleans, functions, conditional and identifier declarations
- No side effects
- We use an environment to track the value of variables: a map from variables to values
- No translation into intermediate language: the interpreter recursively walks the AST to execute each construct

Note: actually, our interpreter implements a big step operational semantics

See the file fun.ml

The fun language: syntax (in math)

$$
\begin{array}{rcll}
x, f & \in & \mathit{Id} & \\
n, n_1, n_2 & \in & \mathit{Num} & \\
e \in \mathit{Exp} & ::= & n & \text{integer literal} \\
& | & x & \text{identifier} \\
& | & \textbf{if } e_1 \textbf{ then } e_1 \textbf{ else } e_2 & \text{conditionals} \\
& | & e_1 \diamond e_2 & \diamond \in \{+, -, *\} \text{ primitive operators} \\
& | & \textbf{let } x = e_1 \textbf{ in } e_2 & \text{declarations} \\
& | & \textbf{fun } f\ x = e_1 \textbf{ in } e_2 & \text{function declarations} \\
& | & e_1\ e_2 & \text{function application}
\end{array}
$$

The fun language: syntax (in code)

```
type expr =
  | CstI of int
  | Var of string
  | If of expr * expr * expr
  | Prim of string * expr * expr
  | Let of string * expr * expr
  | Letfun of string * string * expr * expr  (* (f, x, fBody, lBody) *)
  | Call of expr * expr
```

The fun language: semantic domains (in math)

$v \in \mathit{Value} = \mathbb{Z} \cup \mathit{Closure}$	expressible and denotable values
$\rho \in \mathit{Env} = \mathit{Id} \to \mathit{Value}$	environments
$c \in \mathit{Closure} = \mathit{Id} \times \mathit{Id} \times \mathit{Exp} \times \mathit{Env}$	closures

The fun language: semantic domains (in code)

```
(** Environments: associative lists *)
type 'v env = (string * 'v) list

(*
 Expressible and Denotable values.
 A runtime value is an integer or a function closure
*)
type value =
  | Int of int
  | Closure of string * string * expr * value env
```

The fun language: semantic rules 1 (in math)

The semantic is defined by a relation (function) $\cdot \vdash \cdot \to \cdot \subseteq \mathit{Env} \times \mathit{Exp} \times \mathit{Value}$

$$\frac{}{\rho \vdash n \to n} \qquad \frac{}{\rho \vdash x \to \rho(x)} \qquad \frac{\rho \vdash e_1 \to n_1 \qquad \rho \vdash e_2 \to n_2}{\rho \vdash e_1 \diamond e_2 \to n_1 \odot n_2}$$

$$\frac{\rho \vdash e_1 \to v_1 \qquad \rho[x \mapsto v_1] \vdash e_2 \to v_2}{\rho \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \to v_2}$$

The fun language: semantic rules 1 (in code)

The semantic is defined by the function `eval`

```
let rec eval (e : expr) (env : value env) : value =
    match e with
    | CstI i -> Int i
    | Var x  -> lookup env x
    | Prim(ope, e1, e2) ->
      let v1 = eval e1 env in
      let v2 = eval e2 env in
      match (ope, v1, v2) with
      | ("*", Int i1, Int i2) -> Int (i1 * i2)
      ...
    | Let(x, eRhs, letBody) ->
      let xVal = eval eRhs env in
      let letEnv = (x, xVal) :: env in
      eval letBody letEnv
```

The fun language: semantic rules 2 (in math)

The semantic is defined by a relation (function) $\cdot \vdash \cdot \rightarrow \cdot \subseteq \mathit{Env} \times \mathit{Exp} \times \mathit{Value}$

$$\frac{\rho \vdash e_1 \rightarrow 1 \qquad \rho \vdash e_2 \rightarrow v_2}{\rho \vdash \textbf{ if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rightarrow v_2} \qquad \frac{\rho \vdash e_1 \rightarrow 0 \qquad \rho \vdash e_3 \rightarrow v_3}{\rho \vdash \textbf{ if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rightarrow v_3}$$

The fun language: semantic rules 2 (in code)

The semantic is defined by the function `eval`

```
let rec eval (e : expr) (env : value env) : value =
    ...
    | If(e1, e2, e3) ->
      begin
      match eval e1 env with
       | Int 0 -> eval e3 env
       | Int _ -> eval e2 env
       | _     -> failwith "eval␣If"
      end
    ...
```

The fun language: semantic rules 3 (in math)

The semantic is defined by a relation (function) $\cdot \vdash \cdot \to \cdot \subseteq \mathit{Env} \times \mathit{Exp} \times \mathit{Value}$

$$\frac{c = (f, x, e_1, \rho) \qquad \rho[f \mapsto c] \vdash e_2 \to v_2}{\rho \vdash \ \textbf{fun}\ f\ x\ = e_1\ \textbf{in}\ e_2 \to v_2}$$

$$\frac{\begin{array}{c}\rho \vdash e_1 \to (f, x, e, \rho') \qquad \rho \vdash e_2 \to v_2 \\ \rho'[f \mapsto (f, x, e, \rho'), x \mapsto v_2] \vdash e \to v\end{array}}{\rho \vdash e_1\ e_2 \to v}$$

The fun language: semantic rules 3 (in code)

The semantic is defined by the function `eval`

```
let rec eval (e : expr) (env : value env) : value =
    ...
    | Letfun(f, x, fBody, letBody) ->
      let bodyEnv = (f, Closure(f, x, fBody, env)) :: env in
      eval letBody bodyEnv
    | Call(eFun, eArg) ->    let fClosure = eval eFun env in
      begin
      match fClosure with
      | Closure (f, x, fBody, fDeclEnv) ->
        let xVal = eval eArg env in
        let fBodyEnv = (x, xVal) :: (f, fClosure) :: fDeclEnv
        in eval fBody fBodyEnv
      | _ -> failwith "eval␣Call:␣not␣a␣function"
      end
```