Pocaml Project Proposal

by Leo (flq2101), Yunlan (yl4387), Yiming (yf2484), Peter (jc4883)

1. Introduction

Pocaml is the "poor man's OCaml". It tries to incorporate the main features of OCaml, such as the type system, functions, I/O, and much of the same syntactic sugar, while also including a standard library for common use cases. A few of tentative features in Pocaml includes algebraic data types, type inference and many functions from the OCaml's standard library. The challenge of this project will be to compile Pocaml down to LLVM.

Part of our goal is that everything valid in our language can also be compiled by an OCaml compiler, in the same way C code can be compiled by a C++ compiler. In the beginning, we will limit the language to a small number of types, so that it is easier to handle and debug. As we proceed with the project, we are interested in engineering features like type inference, algebraic types, etc., if time permits.

2. Code Snippets

```
let rec gcd (a : int) (b : int) : int =
    if b = 0 then a
    else gcd b (a mod b)

let print_gcd (a : int) (b : int) : () =
    print_endline (string_of_int (gcd a b))
```

This code snippet demonstrates how to implement the greatest common denominator algorithm in our language, as well as the function that prints out the solution after converting it to string. This is pretty much an OCaml implementation with type specification.

Our language supports the implementation of most simple algorithms, and specializes in arithmetic calculations and list manipulations.

3. Type System

Pocaml will have a Hindley-Milner type system and, ideally, type inference. We will have built-in types: Unit, Int, Bool, Char, Arrow (function).

4. Algebraic Data Types

Pocaml will aim to have ADT support. As a result, pattern matching should ideally be supported also. If this becomes too difficult or too time consuming, we have the fall back upon using lists and tuples, which also let users design their own data structures.

5. Functions

Ultimately, Pocaml will include type inference for functions. However, the first version of Pocaml will require explicit types. The goal for both the first and final versions is that all Pocaml functions can be run in a OCaml compiler. The basic syntax for a lambda expression is:

```
fun (num: int) : int-> num + 5
```

There will also be first-class functions, such as

```
let addFive (num: int) : int = num + 5
```

You can also write functions with multiple arguments like so:

```
let multiplyTwoNums (x: int) (y: int) : int = x * y
```

And much like OCaml, the above is just shorthand for:

```
let multiplyTwoNums (x: int) : int -> int -> int = fun _ (y: int) : int -> x*y
```

6. I/O

I/O for Pocaml will include functionality for reading and writing to files using the same syntax as OCaml. It will also be able to read from stdin and write to stdout in the same way as OCaml.

Writing to a file:

```
let oc = open_out "file.txt" in
    Printf.fprintf oc "Hello, world!\n";
    close_out oc;
```

Reading from a file:

```
let ic = open_in "file.txt" in
      let line = input_line ic in
        print_endline line;
        flush stdout;
        close_in ic
```

Reading a message from stdin and output to user to stdout:

```
let message = input_line stdin in
      print_endline message
```

7. Standard Library

Pocaml doesn't provide `for` and `while` loop constructs. Alternatively, Pocaml's standard library will implement common list operations such as `map` and `iter` using recursion to allow programmers to easily repeat the same set of procedures on a collection of elements or for a specified number of times.

In addition to `map` and `iterate`, common list operations such as `append`, `fold_left` and `fold_right` will also be supported.

```
let rec map (f: int -> int) = fun (lst : int list) ->
  match lst with
  | [] -> []
  | h::t -> f h :: map f t

let rec iter (f: int -> ()) = fun (lst: int list) ->
  match lst with
  | [] -> ()
  | h::t -> f h; iter f t


let rec append (list1: int list) (list2: int list) : (int list) =
  match list1 with
  | [] -> list2
  | h::[] -> h::list2
  | h::t -> h::append t list2

let rec fold_left (op: int -> int -> int) (init: int) (lst: int list) : (int)
=
  match lst with
  | [] -> init
  | h::t -> fold_left op (op init h) t

let rec fold_right (op: int -> int -> int) (lst: int list) (init: int) : (int)
=
  match lst with
  | [] -> init
  | h::t -> op h (fold_right op t init)
```

8. Built-in Code Structures

List

- List is one of the most commonly used data structures in functional programming languages, so we feel the need to incorporate special list syntax. We will support the following syntax:

    - empty list: `[]`
    - bracket-and-semicolon notation: `[1; 2; 3]`
    - :: cons notation: `(1 :: (2 :: (3 :: [])))`

Operators as functions

- We treat operators as normal functions that are evaluated as infix operators and provide built-in operators such as `=, <, >, +, -, *, /` which can be used in the same way as in ocaml. Since these operators are functions, they can also be used as prefix operators, such as (+) a b.

Control flow expressions

- Control flow expressions such as if and else are to be implemented as built-in structures so that the users do not need to define their own versions every time needing to use these features. However, since we are a functional language that is as pure as possible, we will largely forbid the use of mutable variables, and therefore, for and while loops.

Let expression and lambda expressions

- Similar to the use case in ocaml, we are going to create the let keyword as a default constructor for expressions. We use `->` notation to define anonymous functions.