主専攻実験[S-8] 関数プログラミング

最終課題

情報科学類 202113564 三村潤之介

これまで作成したプログラムと解答した課題を整理し、1 つのまとまったインタープリタ、型検査器およびコンパイラとして仕上げよ。

インタープリタ、型検査器、コンパイラを main.ml(レポート最下部に記載)にまとめて記述した。それぞれ、eval,tcheck,compile という式で実装した。インタープリタ及びコンパイラについては、1 つの exp 式を受け取って、その実行結果を示す式としてそれぞれ evaltop,docam を別に記述した。docam 式は、コンパイルと CAM 実行を続けて行う処理である。

上記のインタープリタおよびコンパイラに、ミニ OCaml 言語のプログラムとして 面白いものを与え、正しく動作するかチェックするとともに、性能(実行速度)を調べよ。 なお、例題として与えるプログラムは、エラーとなるものを必ず含むこと。

階乗を求めるプログラムを実装することを考えた。20 の階乗を求めるプログラムを以下のように exp 式で記述した。

```
let kaijou = LetRec("f","n",
               If(Eq(Var("n"), IntLit(1)), IntLit(1), Times(Var("n"),
App(Var("f"), Minus(Var("n"), IntLit(1))))),
               App(Var("f"), IntLit(20)))
```

インタープリタでの実行は以下のようになる。

```
# evaltop kaijou;;
- : value = IntVal 2432902008176640000
```

コンパイラ及び CAM での実行は以下のようになる。

```
# docam kaijou;;
- : cam_value = CAM_IntVal 2432902008176640000
```

この値は 20!となっていて、正確に計算できていることがわかる。

次に実行速度を調べる。
発展課題 7-5 で用いた time 式によって実行時間を測定する。

```
let time : (unit -> 'a) -> float =
  fun f ->
    let start = Sys.time () in
    let res   = f () in
```

```
  let end_  = Sys.time () in
  end_ -. start
```

実行するプログラムは、上記の階乗を求めるプログラムを変更し、100000!を求めるようにした。ただし、21 以上では正常に階乗を求められず、あくまで実行時間を測ることのみを目的としている。
また、OCaml で実行した時間も測定するため、階乗を求める exp 式をそのまま Ocaml 式で実装した。

```
let rec kai n =
 if n = 1 then 1
 else n * (kai (n - 1))
```

以下に順にインタープリタ、コンパイラ&CAM、OCaml での実行時間を示す。

```
# time (fun () -> evaltop kaijou);;
- : float = 0.119022000000000183
# time (fun () -> docam kaijou);;
- : float = 0.124832000000000498
# time (fun () -> kai 100000);;
- : float = 0.00261200000000005872
```

次に、コードのコンパイル及び CAM 実行の実行時間を示す。

```
# time (fun () -> compiletop kaijou);;
- : float = 1.19999999990128e-05

# let c = compiletop kaijou;;
val c : cam_code =
  [CAM_Closure
    [CAM_Ldi 1; CAM_Access 0; CAM_Eq;
     CAM_Test ([CAM_Ldi 1],
      [CAM_Ldi 1; CAM_Access 0; CAM_Sub; CAM_Access 1; CAM_Apply;
       CAM_Access 0; CAM_Mul]);
     CAM_Return];
   CAM_Let; CAM_Ldi 100; CAM_Access 0; CAM_Apply; CAM_EndLet]
# time (fun () -> transtop c);;
- : float = 0.134733999999999909
```

docam の実行時間のほとんどを CAM での実行時間が占めていることがわかる。
evaltop と docam は、実行するたびに実行時間が最大 20%ほど増減するため、どちらのほうがより短くなっているとは言えない。OCaml で実行したときの実行時間はそれらより短くなっていて、何らかの最適化処理が行われていると考えられる。
今回の実装では、コンパイラにおいて最適化処理を行うようにしていないため、このように差が出ない結果となったのだと考える。

次に、エラーとなるコードを与えた。上から順に、構文エラー、無限ループ、型エラーとなるコードを、階乗を求めるコードから変更して定義する。赤字で示している部分がエラーである。

```
let kaijou_structure_err = LetRec("f","n",
                    If(Eq(Var("n"), IntLit(1)), IntLit(1), Times(Var("n"),
App(Var("f"), Plus(Var("n"), IntLit(-1), IntLit(1)))),
                    App(Var("f"), IntLit(20)))

let kaijou_loop_err = LetRec("f","n",
                If(Eq(Var("n"), IntLit(1)), IntLit(1), Times(Var("n"),
App(Var("f"), Plus(Var("n"), IntLit(0)))),
                App(Var("f"), IntLit(20)))

let kaijou_type_err = LetRec("f","n",
                If(Eq(Var("n"), IntLit(1)), IntLit(1), Times(Var("n"),
App(Var("f"), Plus(Var("n"), BoolLit(false)))),
                App(Var("f"), IntLit(20)))
```

＜構文エラー＞

```
File "main.ml", line 366, characters 95-132:
366 |                     If(Eq(Var("n"), IntLit(1)), IntLit(1),
Times(Var("n"),    App(Var("f"),    Plus(Var("n"),    IntLit(-0),
IntLit(1)))),

Error: The constructor Plus expects 2 argument(s),
       but is applied here to 3 argument(s)
```

OCaml の#use の時点で引数の数の違いによるエラーなどは弾かれてしまうため、exp 式が定義できた段階で、構文エラーは起きていないと見ることができると考える。

＜無限ループ＞
上からインタープリタでの実行、CAM での実行を示す。

```
# evaltop kaijou_loop_err;
  ^CInterrupted.
# docam kaijou_loop_err;;
  ^CInterrupted.
```

どちらも、ターミナル上で中断しないと何も表示が出なくなってしまう。OCaml などでこういった処理があると、Stack overflow during evaluation (looping recursion?).というようにメッセージが出るため、こういった処理に近づけられると望ましいと考える。

＜型エラー＞
上からインタープリタでの実行、CAM での実行を示す。

```
# evaltop kaijou_type_err;;
Exception: Failure "integer value expected".
# docam kaijou_type_err;;
Exception: Failure "add error: integer expected".
```

どちらも型が整数でないことによるエラーであるとわかるメッセージが表示される。

本実験で学んだことについての考察および感想を記述せよ。 たとえば、このインタープリタは素朴でわかりやすい実装を優先したために、 効率がよくない点があるので、それについての改善案を思いつけば、 それを記述せよ。 また、関数型プログラム言語の利点と欠点について、自分の考えを述べよ。

実験を通して、インタープリタやコンパイラを作成したことで、OCaml がそういったものの実装に向いているような印象を得た。強力なパターンマッチングの機能を用いて、簡潔にインタープリタなどを記述できるのは、可読性だけでなく、初学者がその内部構造を学ぶのに適していると考える。
インタープリタの改善案について、最初に思いつくのが、末尾再帰である関数の最適化であるが、そもそもそれはコンパイラ上でコード最適化を行う際の話であったため、インタープリタ上ではどういった処理になるのか、検討がつかないでいる。

自分なりの工夫(インタープリタの拡張、型検査器の拡張、型推論器の実装、コンパイラの高速化など）を 1 つ以上すること。

行った独自の実装について、以下に箇条書きで示す。

- 型検査器の拡張(Let, Times, Greater)
- 型推論器の実装
- コンパイラの拡張(Minus,Times,Greater)

型検査器、コンパイラの拡張については、簡単な実装であるため詳細な説明は省く。型推論器の実装は、発展課題 7-5,7-6,7-7 についてのレポートで詳細を述べている。
個人的な感想としては、型検査器の拡張にて Let-rec 式も試みたが、実装できなかったことが心残りである。

ソースコードおよび実行結果

実行結果は本文にて対応する部分に記述しているため、省略する。ソースコード、およびテストプログラムは github リポジトリ上の 12 final/main.ml に記述した。テストプログラムはファイル下部 354~382 行である。その他は、レポートの作成にあたって必要となった式などである。
https://github.com/mimunojun/functional_prog