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>