全体ゼミ

筑波大学 プログラム論理研究室
大石 純平

20150710

1　論文紹介します

読んだ論文

- A Gentle Introduction to Multi-stage Programming, Part II
- Walid Taha

論文紹介の目的

- Staging の技術を使って，interpreter の書き方を学ぶ．

2　概要

A Gentle Introduction to Multi-stage Programming の part I では，

1. インタプリタを書いて，その正当性をチェック
2. ステージングの annotation を加えることにより ステージインタプリタへ
3. ステージングの実装のパフォーマンスのチェック

の 3 つのステップからなるアプローチを紹介した．満足するステージングに到達するために，CPS 変換が必要である．

part I では，Lint [1] と呼ばれるシンプルな言語に焦点を当てていた．

part II では， Aloe と呼ばれる Scheme 言語のサブセット言語に焦点を当てる．そのインタプリタにおいては，型のタグを付けたり，外したりということが Over head となる．それを解決するために，CPS 変換，ステージングの技術を用いる．そのように，ステージングの知識のレパートリーを増やすことが目的である．

[1] Lint は， interger という 1 つの型のみを持ち，integer から integer への関数のみを持つような言語

3 Parsing

```
s-expression ::= integer | symbol | string | (s-expression*)
```

上記の s-expression syntax の式に対して， parse が成功すると，以下の様な OCaml の型の値が返ってくる．

```
type sxp =
| I of int (* Integer *) | A of string (* Atoms (symbols) *)
| S of string (* String *) | L of sxp list (* List *)
```

4 Interpreter

Aloe とは，bool, integer, string, mutable variable, mutable list, higher-order function を含む scheme のサブセット言語である．
型なしの言語のインタプリタを書くときに考慮すべき最初のことは，言語がサポートする値の種類を決定することである．

```
type dom = Bool of bool
         | Int of int
         | Str of string
         | Fun of int * (dom list -> dom)
         | Undefined
         | Void
         | Empty
         | Cons of dom ref * dom ref
```

- 3 base types
  - booleans
  - integers
  - strings
- function
  - 各関数の値は，それが期待する引数の個数を表す整数でタグ付けされる．
- Undefined と Void
  - Undefined は初期化されていない場所に対して使用される．
  - Void は 副作用のある計算から結果が存在しないことを示すものである．
- Empty と Cons
  - Empty は空の配列
  - Cons は配列

```
type var = Val of dom | Ref of dom ref
```

Aloe は scheme のサブセット言語なので，いくらかは変数を割り当てることができるが，全ての変数は割り当てることができない．例えば，Aloe の関数の引数は immutable である．
この違いを反映するために，全ての環境は，上記の型の値に名前をマップする必要がある．

```
let ext env x v = fun y -> if x=y then v else env y
```

ext : 環境の中の一つの変数に対して新しい binding を与える関数

```
let rec lext env xl vl =
  match xl with
  | []    -> env
  | x::xs -> match vl with
             | [] -> raise (Error "Not enough arguments")
             | y::ys -> lext (ext env x y) xs ys
```

lext: 一気に，環境の中の複数の変数に対して，binding を与える関数

5 Concrete Syntax

```
I is the set of integers
S is the set of strings
A is the set of symbols ("A" for "Atomic")

U ::= not|empty?|car|cdr
B ::= + | - | * | < | > | = | cons | set-car! | set-cdr!
E ::= true | false | empty | I | "S" | A | (U E) | (B E E)
      | (cond(E E)(elseE))|(set!AE)|(andE E*)|(orE E*)
      | (begin E E*) | (lambda (A*) E) | (E E*)

P ::= E | (define A E)P | (define (A A*) E)P
```

- I : integer 数字のシークエンスとして定義されている．
- S : string 文字のシークエンス
- A : symbol (atom) スペースの存在しない文字のシークエンス
- U : unary operator (単項演算子)
- B : binary operator (二項演算子)
- E : expression bool 値，空のリストを表す empty が含まれていて，integer, string, symbol を埋め込む．
- P : program プログラムは，式，または，変数や関数定義のネストされた配列

で，ここで，注意することとしては，symbol に関しては曖昧性があることである．例えば， true っていう expression は E の定義の true ってやつと A っていうやつのいずれかに一致する．このような理由から，文字列の導出は常に，E の定義の一番左のやつから検討していく．

6 The Interpreter for Expressions

- eval : 式 (expression) に対するインタプリタ（まあインタプリタというよりは評価器）
- peval : プログラムに対するインタプリタ

6.1 eval

```
let rec eval e env =
  try (match e with
      ...)
  with
      Error s -> (print_string ("\n"^(print e)^"\n");
                  raise (Error s))
```

... の中身については，全部書くとアレなので，いくつかピックアップして説明する．

6.2 Lambda Abstraction

```
| L [A "lambda" ; L [S x] ; e] ->
       Fun (1, fun l -> match l with [v] -> eval e (ext env x (Val v)))
```

1 つの引数の場合．

Fun タグが付いた値を返す．その Fun タグが付いた値は 1 つ目はそれが期待する引数の個数を表し（この場合は 1）．2 つ目は OCaml のラムダ抽象であり，それは，1 つの要素のみからなるリストを引数と取る．その OCaml のラムダ抽象の結果は，Aloe のラムダ抽象の body である式 e を評価した結果である．この式の評価は，環境 env のもとで，name x から値 Val v へのマッピングによって生じる．ラムダ抽象の引数を 1 つに固定させるという事実を反映するために，v にタグ Val を付ける．

```
| L [A "lambda" ; L axs ; e] ->
     let l = List.length axs in
     Fun (l, fun v ->
               eval e (lext env
                  (List.map (function A x -> x) axs)
                  (List.map (fun x -> Val x) v)))
```

引数が複数の場合

6.3 Function Application

```
     | L [e1; e2] ->
       let (1,f) = unFun (eval e1 env) in
       let arg = eval e2 env in
       f [arg]
```

1 つの引数の場合

パターンマッチでは，e1 は 1 つの引数を持つ関数であることを前提としている．まず最初の let statement では，式 e1 を評価し，Fun タグを除去する．第一成分が 1 であり，第二成分が f である．次の let statement では，e1 の引数である e2 を評価し arg としている．最後に 1 引数関数である f にシングルトンリスト [arg] を渡す．

```
     | L (e::es) ->
     let (i,f) = unFun (eval e env) in
     let args = List.map (fun e -> eval e env) es in
     let l = List.length args
     in if l=i
     then f args
     else raise (Error ("Function has "^(string_of_int l)^
                 " arguments but called with "^(string_of_int i)))
```

引数が複数の場合
　まず，e を評価し，それから es の要素全てに対して評価を行っていき，その結果を args(dom list) とする．それから args の個数と e の期待する引数の個数とが一致していれば，関数適用を行い，それ以外の場合は，エラーが発生する．

6.4 peval

```
let rec peval p env =
  match p with
    | [e1] -> eval e1 env
    | (L [A "define"; A x; e1])::p ->
        let r = ref Undefined in
        let env' = ext env x (Ref r) in
        let v = eval e1 env' in
        let _ = (r := v)
        in peval p env'
    | (L [A "define"; L ((A x)::xs); e1])::p ->
        peval (L [A "define"; A x;
                  L [A "lambda" ; L xs ; e1]]::p) env
    | _ -> raise (Error "Program form not recognized")
```

7 Converting into Continuation-Passing Style (CPS)

```
eval : sxp -> (string -> var) -> dom
```

CPS 変換する前の eval

```
keval : sxp -> (string -> var) -> (dom -> dom) -> dom
```

CPS 変換後の eval
　継続を表す関数を加える．元のコードの返り値を全て継続に渡す．

7.1 interpreter

7.1.1 keval

```
let rec keval e env k =
  try
    (match e with
     ...)
  with Error s -> (print_string ("\n"^(print e)^"\n");
                   raise (Error s))
```

7.1.2 Lambda Abstraction

```
| L [A "lambda" ; L axs ; e] ->
let l = List.length axs in
    k (Fun (l, fun v ->
    keval e (lext env
                (List.map (function A x -> x) axs)
                (List.map (fun x -> Val x) v))
      (fun r -> r)))
```

ポイントは，keval への 継続として，恒等関数を渡しているところである．これは，

8 Staging the CPS-Converted Interpreter

ここでは，CPS 変換したインタプリタを staging する際に発生する問題があるので，その問題について説明する．

以下では，いわゆる environment classifier parameter を型から除いて説明する．例えば，('a, int) code は int code とする．

```
keval : sxp -> (string -> var) -> (dom -> dom) -> dom
```

は以下のようにする．

```
seval : sxp -> (string -> svar) -> (dom code -> dom code) -> dom code
```

–

```
type var = Val of dom | Ref of dom ref
```

は以下のようにする．

```
type svar = Val of dom code | Ref of (dom ref) code
```

8.0.3 ここで，staging anotation の復習

- bracket は 計算を遅らせる
- escape は bracket を外す．
- run は code の評価を行う．

8.1 Cases That Require Only Staging Annotations

```
| A "true"   -> k .<Bool true>.
| I i -> k .<Int i>.
| A x ->
(match env x with
 | Val v -> k v
 | Ref r -> k .<(Runcode.run .~r)>.)
| L [A "not"; e1] ->
seval e1 env (fun r ->
      k .<Bool (not(unBool .~r))>.)
```

継続に渡すものの外側を bracket に囲ませれば良い．

8.2 Lambda Abstraction

一つの例外を除いて，ラムダ抽象については，braket と escape を追加するだけで，stage 化される．

ラムダ抽象の引数の数は static に決まっている．なので，static なパラメータリストから，個々のパラメータを抽出するコードを生成する．

これは，引数の数を知っているという事実を利用して，引数のリストを eta-expanding することによって，達成できる．

```
eta_list : int -> 'a list code -> 'a code list

let eta_list l v =
  let rec el_acc l v a =
    if l<=0 then [] else .<List.nth .~v a>. :: (el_acc (l-1) v (a+1))
  in el_acc l v 0

keta_list : int -> 'a list code -> ('a code list -> 'b code) -> 'b code

let keta_list l v k =
  let rec el_acc l v a k =
    if l<=0
    then k []
    else
      .<match .~v with
    | x::xs ->
      .~(el_acc (l-1) .<xs>. (a+1) (fun r ->
          k (.<x>. :: r)))
    | _ -> raise (Error "")>.
  in el_acc l v 0 k

     | L [A "lambda" ; L axs ; e] ->
     let l = List.length axs
     in k .<Fun (l, fun v ->
           .~(keta_list l .<v>. (fun r->
               seval e (lext env
                           (List.map (function A x -> x) axs)
                           (List.map (fun x -> Val x) r))
                 (fun r -> r))))>.
```

8.3 Function Application

```
lift_list : 'a code list -> 'a list code

let rec lift_list l =
  match l with
  | [] -> .<[]>.
  | x::xs -> .< .~x :: .~(lift_list xs)>.

| L (e::es) ->
     seval e env (fun r1 ->
         .<let (i,f) = unFun .~r1
           in .~(kmap (fun e -> seval e env) es (fun r2 ->
           let args = r2 in
               let l = List.length args
```

```
in .<if l= i
then let r = f .~(lift_list args) in .~(k .<r>.)
else raise (Error ("Function has "^(string_of_int l)^
                        " arguments but called with "^
                        (string_of_int i)))>.))>.)
```

9 The Interpretation of a Program as Partially Static Data Structures

In almost any imaginable language that we may consider, there are many programs that contain computation that can be performed before the inputs to the program are available. In other words, even when we ignore the possibility of having one of the inputs early, programs themselves are a source of partially static data. If we look closely at the way we have staged programs in the previous section, we notice that we made no attempt to search for or take advantage of such information. A standard source of partially static information is closed expressions, meaning expressions that contain no variables, and therefore, contain no unknown information. Some care must be taken with this notion, because some closed programs can diverge. Another, possibly more interesting and more profitable type of partially static information that can be found in programs in untyped languages is partial information about types. This information can be captured by the data type tags that allow the runtime of an untyped language to uniformly manipulate values of different types. Because the introduction and the elimination of such tags at runtime can be expensive, reducing such unnecessary work can be an effective optimization technique.

9.1 A Partially Static Type for Denotable Values

部分的に static に型情報を利用しやすくするために sdom という型を導入する.

具体的には，dom code を生成するのでなく，stage 用の型 sdom を生成するようにする.

```
type sdom =
  | SBool of bool code
  | SInt of int code
  | SStr of string code
  | SFun of int * (sdom list -> sdom)
  | SUndefined
  | SVoid
  | SEmpty
  | SCons of (dom ref) code * (dom ref) code
  | SAny of dom code
```

基本的には，static に値を知っているときに，上記コードコンストラクタに対応するタグを push できる．SAny というコンストラクタは，タグの追加に対して，static な情報を持っていな場合を表している．特別なケースとして，Cons が挙げられる．それぞれの Cons cell は，mutable なので，ここで検討している同じ技術 (タグの情報を push する) は ref コンストラクタに対しては使用できない.

In essence, this type allows us to push tags out of the code constructor when we know their value statically. The last constructor allows us to also express the case when there is no additional static information about the tags (which was what we assumed all the time in the previous interpreter). An important special case above is the case of Cons. Because

each of the components of a cons cell is mutable, we cannot use the same techniques that we consider here to push information about tags out of the ref constructor. A side effect of this type is that case analysis can become somewhat redundant, especially in cases in which we expect only a particular kind of data. To minimize and localize changes to our interpreter when we make this change, we introduce a matching function for each tag along the following lines:

この種の副作用によって，case 節において，特定の種類のデータを期待する時に冗長な変更が必要となる．各タグについて次のようなマッチング関数を用意する．

```
let matchBool r k =
   let m = "Expecting boolean value" in
   match r with
   | SBool b -> k b
   | SAny c -> .<match .~c with Bool b -> .~(k .<b>.)
                | _ -> raise (Error m)>.
   | _ -> k .<raise (Error m)>.
```

It is crucial in the last case that we do not raise an error immediately. We return to this point in the context of lazy language constructs. We also change the type for values stored in the environment as follows:

```
type svar = Val of sdom | Ref of dom ref code
```

繰り返すが，タグの情報を ref コンストラクタからは引っ張ってくることはできない．副作用として，reference cell に格納されている値を変更してしまう事がある．

9.2 Refining the Staged Interpreter

9.2.1 The Easy Cases

```
      | A "true"   -> k (SBool .<true>.)
      | A "empty"  -> k SEmpty
      | I i -> k (SInt .<i>.)
      | S s -> k (SStr .<s>.)
```

これらのケースは，dom code から， sdom へ戻り値の型を変更すればよい．

```
  (match env x with
    | Val v -> k v
    | Ref r -> k (SAny .<(! .~r)>.))
```

Ref r のケースに注目する．sdom 型でなく，dom code 型を持っているため，static なタグの情報は r には存在しない．なので，SAny というタグを使用して，対応している．

直感的に言うと，de-referencing が実行時に発生するので，その値のタグを静的に知る簡単な方法はないということです．

9.2.2 Reingroducing Information

Ground value はタグの static な情報の唯一のソースではない．一般的に言うと，最も primitive な計算の結果のタグが static に知られている．

例えば，次のように論理否定のケースを絞り込むことができる．

```
      | L [A "not"; e1] ->
          xeval e1 env (fun r ->
          matchBool r (fun x ->
          k (SBool .<not .~x>.)))
```

Knowing that the tag always has to be SBool in the rest of the computation allows us to make sure that this tag does not occur in the generated code when this code fragment occurs in a context that expects a boolean.

同様に，タグの情報は Aloe の他のすべての primitive な operation によって 再導入される．

9.2.3 Strictness and Laziness

```
| L ((A "and") :: es) ->
     let rec all l k =
       (match l with
        | [] -> k .<true>.
        | x::xs ->
        seval x env (fun r ->
            .<if unBool .~r
            then .~(all xs k)
            else .~(k .<false>.)>.))
     in all es (fun r -> k .<Bool .~r>.)
```

前

```
| L ((A "and") :: es) ->
          let rec all l k =
            (match l with
              | [] -> k .<true>.
              | x::xs ->
                  xeval x env (fun r ->
                  matchBool r (fun x ->
                    .<if .~x
                      then .~(all xs k)
                      else .~(k .<false>.)>.)))
         in all es (fun r -> k (SBool r))
```

後

```
let matchBool r k =
   let m = "Expecting boolean value" in
   match r with
   | SBool b -> k b
   | SAny c -> .<match .~c with Bool b -> .~(k .<b>.) | _ -> raise (Error m)>.
   | _ -> k .<raise (Error m)>.
```

if unBool を使う代わりに matchBool を使うと，最後のケースに注意深くならないと，正しい結果を返さないだろう．

特に，もしそのケースになれば直ちに例外を発生し，(and true 7) というような式は fail となる．

そのようなことは，あまりに厳密に，型を評価して確認したために起こる．

それを評価した時にだけ，エラーを発生させる code fragment を返すことで，正しいセマンティクスが保存されていることを確認する．

10 coclusion

主に Part1 での技術を用いて，higher-order な関数と複数の型を持つ言語に対してのインタプリタの実装を示した．その際，特に動的言語に対して有用な最適化の技術を用いた．また，合理的なスピードアップが，ステージングを使用することにより，可能となった．この実装は，最も効果的なステージングインタプリタを目指してはいない．