프로그래밍 언어 개론
과제 보고서

[PL00]HW12_201601980_KimSeongHan

개발환경 : Ubuntu

제출일 (2021-05-21)

201601980 / 김성한(00분반)

<과제 설명>

Homework

• 재귀함수를 작성할 수 있는 RCFVAE을 읽을 수 있는 interpreter를 작성 하시오.

```
let rec interp_e (s : Store.t) (e : Ast.expr) : Store.value =
(* write your code *)
(* ID의 경우 FreezedV 여부를 확인하고 반환해야함 *)
(* App의 경우 RCFVAE가 함수 호출 시 항상 인자를 얼리도록 변경 *) (이론강의 23p)
(* RLetIn 에서 오류가 생길 경우 "Not a function: %a"*)

let interp (p : Ast.fvae) : Store.value =
(* write your code *)
```

Testcase : https://github.com/SeoyeonKang/UPL2021/tree/master/week12

[과제 제시 내용]

● 이번 과제는 지난주까지 진행했던 CFVAE interpreter에 재귀함수를 추가한 RCFVAE interpreter를 구현하는 것이다.

● CFVAE와 RCFVAE의 차이로는 재귀함수를 해석할 수 있다는 점이며 재귀함수를 읽기위해 Id 및 App에서의 Bigstep-Operataion이 바뀌게된다. 기존 CFVAE는 인자의 expression을 미리 계산하여 함수에 인자로 넘겨주었는데 재귀함수에서 해당방식을 사용할 경우 재귀함수는 무한루프를 돌게되고, 이는 곧 stack overflow를 야기하게 된다.

● 이를 막기 위하여 이번 RCFVAE에서는 Expression Freezing을 통해 표현식을 얼려두었다가 함수 호출시 필요한 인자만을 녹여 사용하는 방식을 채택하였으며, 이번 과제에서의 구현 내용에 해당된다.

<코드 구현 - 해결 방법>

k0s0a7@DESKTOP-MLPLCFD:

```
module F = Format

type expr =
  | Num of int
  | Add of expr * expr
  | Sub of expr * expr
  | Id of string
  | LetIn of string * expr * expr
  | RLetIn of string * expr * expr
  | App of expr * expr
  | Fun of string * expr
  | Lt of expr * expr

type fvae =
  | Prog of expr

let rec pp_e fmt e =
  match e with
  | Num n -> F.fprintf fmt "(Num %d)" n
  | Add (e1, e2) -> F.fprintf fmt "(Add %a %a)" pp_e e1 pp_e e2
  | Sub (e1, e2) -> F.fprintf fmt "(Sub %a %a)" pp_e e1 pp_e e2
  | Id x -> F.fprintf fmt "(Id %s)" x
  | LetIn (x, e1, e2) -> F.fprintf fmt "(LetIn %s %a %a)" x pp_e e1 pp_e e2
  | RLetIn (x, e1, e2) -> F.fprintf fmt "(RLetIn %s %a %a)" x pp_e e1 pp_e e2
  | App (e1, e2) -> F.fprintf fmt "(App %a %a)" pp_e e1 pp_e e2
  | Fun (p, e) -> F.fprintf fmt "(Fun %s -> %a)" p pp_e e
  | Lt (e1, e2) -> F.fprintf fmt "(Lt %a < %a)" pp_e e1 pp_e e2

let pp fmt (Prog e) =
  F.fprintf fmt "(Prog %a)" pp_e e
```

[ast.ml의 내부코드]

● 이번 주 실습에서 사용된 ast.ml이다.

● 지난 CFVAE와 다르게 재귀함수를 다루는 RLetIn expr이 추가되었으며 기존 LetIn과 비슷하게 string*expr*expr의 튜플형태를 가진다.

[store.ml의 내부코드]

● 이번 실습에서 사용된 store.ml의 내부코드이다.

● CFVAE와 다르게 함수 호출시 필요한 인자의 계산을 늦추기 위하여 사용된 FreezedV 가 추가되었으며, Ast.expr*t 의 타입을 가진다. 추가적으로 times 변수가 추가되었다.

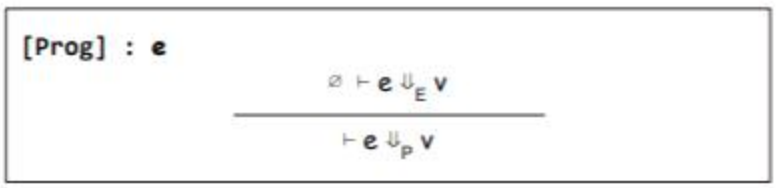
● 그 외의 것들은 지난 CFVAE와 동일하다.

```
(* practice & homework *)
let interp (p : Ast.fvae) : Store.value =
        match p with
        |Prog expr -> interp_e [] expr
```

[interpreter.ml 의 interp 함수]

● interp함수로 초기 인자로 들어온 Prog타입을 interp_e [] expr을 통해 interp_e함수를 호출하고 값을 반환받는 시작함수의 기능을 한다.

● Prog의 Bigstep-Operation은



으로 표현할 수 있으며 이는 빈 추상메모리에서 e를 계산한 결과가 v라고 가정하면 e 는 v로 계산하는 역할을 하게된다. 따라서 위의 interp함수에서 패턴매칭시 같은 기능을 수행하도록 구현하였다.

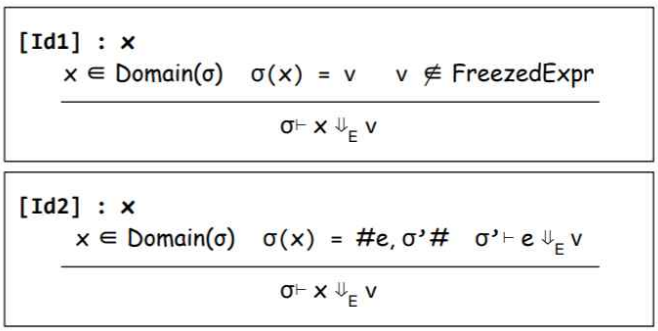
```
(* practice & homework *)
let rec interp_e (s : Store.t) (e : Ast.expr) : Store.value =
       match e with
       | Num n -> NumV n
       | Add (e1,e2) ->
                       begin
                               let exp1 = interp_e s e1 in
                               let exp2 = interp_e s e2 in
                               match exp1,exp2 with
                               | NumV n1, NumV n2 -> NumV(n1 + n2)
                               | _ , _ -> failwith (Format.asprintf "Invalid addition: %a + %a" Ast.pp_e e1 Ast.pp_e e2)
                       end
       | Sub (e1,e2) ->
                       begin
                               let exp1 = interp_e s e1 in
                               let exp2 = interp_e s e2 in
                               match exp1,exp2 with
                               | NumV n1, NumV n2 -> NumV(n1 - n2)
                               | _ , _ -> failwith (Format.asprintf "Invalid subtraction: %a - %a" Ast.pp_e e1 Ast.pp_e e2)
                       end
       | Id str ->
                       begin
                              let v = Store.find str s in
                              match v with
                              | FreezedV (e, s') -> (interp_e s' e)
                              | _ -> v
                       end
       | LetIn (str,e1,e2) -> interp_e (Store.insert str (interp_e s e1) s) e2
```

[interpreter.ml 의 interp_e 함수 초반부]

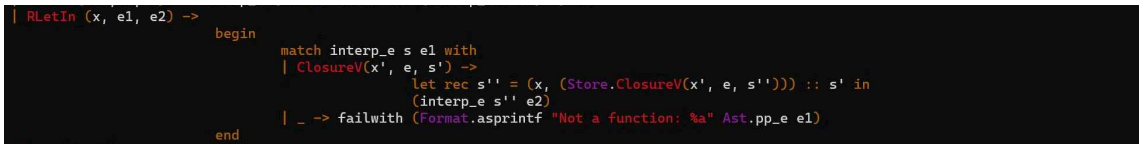
● interp_e 함수의 초반부는 지난 CFVAE와 크게 달라진 점이 없으며 Num, Add, Sub, Id, LetIn 패턴에 대한 처리를 하고 있다.

● 이번 RCFVAE에서는 Id 의 경우 기존 단순히 Store.find str s를 반환하던 것에서 Store.find str s 로부터 반환된 값을 v 에 저장하고 v에 대해 패턴매칭을 수행하여 만일 v가 FreezedV 타입인 경우 해당 FreezedV(e,s`) 에 대해 interp_e s` e를 진행한 값을 반환한다. 그 외의 NumV 혹은 ClosureV의 경우는 값인 v를 그대로 반환해준다.

● 이와 같이 구현함으로써 RCFVAE에서의 Bigstep-Operation



과 같은 동작을 할 수 있도록 하였다.



```
| RLetIn (x, e1, e2) ->
                    begin
                        match interp_e s e1 with
                        | ClosureV(x', e, s') ->
                                let rec s'' = (x, (Store.ClosureV(x', e, s''))) :: s' in
                                (interp_e s'' e2)
                        | _ -> failwith (Format.asprintf "Not a function: %a" Ast.pp_e e1)
                    end
```

[interpreter.ml 의 interp_e 함수 중반부 - RLetIn 패턴]

● 이번에 추가된 RLetIn 패턴의 Bigstep-Operation은

$$[\text{RLetIn}] : \text{let rec } x = e_1 \text{ in } e_2$$

$$\frac{\sigma \vdash e_1 \Downarrow_E \langle \lambda x'.e, \sigma' \rangle \quad \sigma'' = \sigma'[x \mapsto \langle \lambda x'.e, \sigma'' \rangle] \quad \sigma'' \vdash e_2 \Downarrow_E v_2}{\sigma \vdash \text{let rec } x = e_1 \text{ in } e_2 \Downarrow_E v_2}$$

이며 해당 Bigstep-Operation에 기술된 과정대로 구현되었다.

● 다만 기존에 store.ml에 선언한 Store.insert함수를 사용하지 않고 list의 앞부분에 삽입하는 :: 연산을 수행한 이유는 중간에서 추상메모리 s``에 대하여 (x,ClosureV(x`,e,s`))을 추가할 때 아직 s``이 정의되지 않았을 때 사용되었기에 s`` 은 Free Identifier로 인식되고 이를 방지하기 위하여 rec 키워드를 사용하게 되면 rec키워드의 조건 중 RHS(Right Hand Side)에 function application이 오면 안된다는 규칙에 위배되어 해당 :: 연산을 사용하도록 바꿔주었다.

● 이와 같이 RLetIn을 구현해 줌으로써 rec키워드를 사용하여 추상메모리 s``에 함수 자기 자신을 저장할 수 있으며 우리는 재귀함수로 활용할 수 있게 되었다.

● 이제 추가적으로 재귀함수 호출시 Stack Overflow를 방지하기 위해 App 패턴도 조금 변경이 필요하다.

```
| App (e1,e2) ->
                begin
                        match interp_e s e1 with
                        | ClosureV (x,e3,s') -> (interp_e (Store.insert x (Store.FreezedV(e2, s)) s') e3)
                        | _ -> failwith (Format.asprintf "Not a function: %a" Ast.pp_e e1)

                end

| Fun (str,e) -> ClosureV(str,e,s)
| Lt (e1,e2) ->
                begin
                        let v1 = interp_e s e1 in
                        let v2 = interp_e s e2 in
                        match v1,v2 with
                        | NumV n1, NumV n2 -> if n1 < n2 then ClosureV("x",Fun("y",Id("x")),[]) else ClosureV("x",Fun("y",Id("y")),[])
                        | _, _ -> failwith (Format.asprintf "Invalid less-than: %a < %a" Ast.pp_e e1 Ast.pp_e e2)
                end
```

[interpreter.ml 의 interp_e 함수 후반부]

● RLetIn 패턴으로 우리는 재귀함수를 해당 시점의 추상메모리 s``에 저장할 수 있었으며, Stack Overflow를 방지하기 위해 App의 동작도 조금 바꿔주어야 했다.

● 먼저 이번 RCFVAE에서의 App의 Bigstep-Operation은

$$[\text{App}] : e_1\ e_2 \quad (* \text{ Lazy Evaluation } *)$$
$$\frac{\sigma \vdash e_1 \Downarrow_E \langle \lambda x.e_3, \sigma' \rangle \quad \sigma'[x \mapsto \#e_2, \sigma\#] \vdash e_3 \Downarrow_E v_2}{\sigma \vdash e_1\ e_2 \Downarrow_E v_2}$$

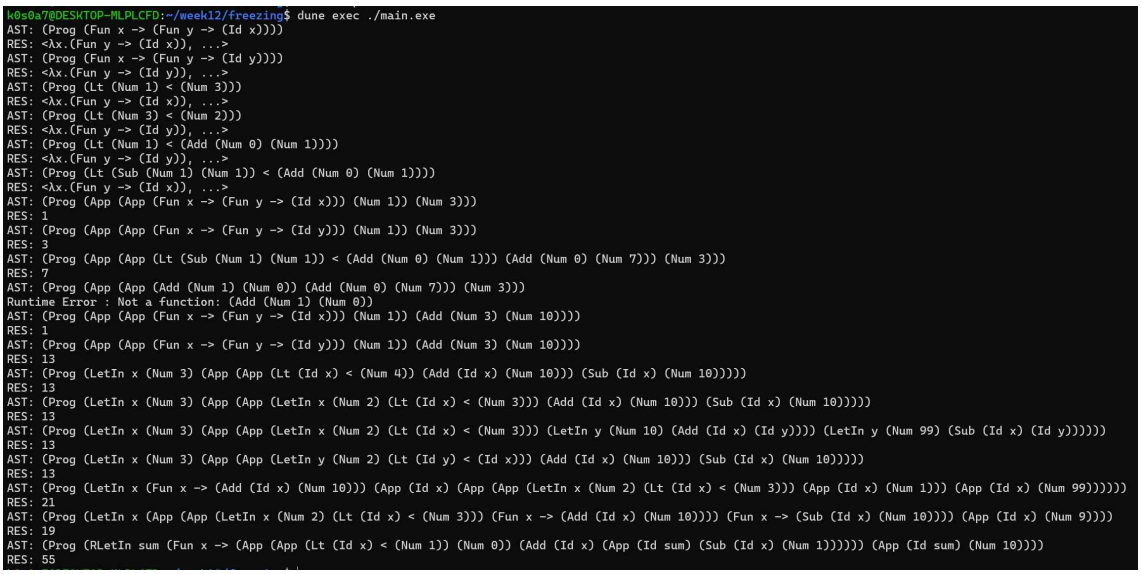
로 표현되며 이는 e1을 연산하여 얻은 ClosureV에서의 추상메모리 s`에 e2와 현재 시점의 추상메모리 s를 FreezedV로 저장하여 당장 연산을 진행하지 않고 해당 추상메모리 s`을 이용해 e3를 연산한 값인 v2를 리턴해주는 과정을 가진다.

● 이러한 Bigstep-Operation의 과정대로 해당 App패턴을 구성하였으며 s`에 FreezedV(e2,s)를 저장하여 e3연산시 인자의 계산을 늦춰줄 수 있었다.

● 이렇게 저장된 FreezedV(e2,s)는 Id 패턴에서 FreezedV와 매칭되어 연산된 값을 반환받고 해당함수는 그 값을 이용해 연산을 진행하게 된다.

● 이후 선언된 Fun, Lt 패턴은 지난 CFVAE와 동일하다.

<실행 결과 화면>



[main.exe 실행시 결과]

[main.ml 내부 작성된 TestCase 결과]

● main.exe의 실행결과와 main.ml내부 작성된 TestCase결과가 일치함을 확인할 수 있었다.

<새로 고찰한 내용 또는 느낀점>

● 이번 과제는 이론 강의를 들으면서 고민을 더 많이 했던 것 같다. 재귀함수라고 하면 질색을 하였는데 이번 학기 프로그래밍 언어 개론 강의를 수강하면서 재귀에 대해 조금씩 이해하게 되었음을 많이 느꼈다. 특히 이번 과제에서는 재귀함수를 사용할 수 있도록 interpreter를 직접 구현해 보면서 더 많이 생각해보았다. 이론 강의 마지막에 나오는 Fixpoint 부분에서도 이해해보려 안간힘을 썼다. 과제의 난이도는 교수님께서 강의 중에 RLetIn 패턴에 대한 코드도 미리 조금 보여주셨고, Bigstep-Operation에 대한 설명도 자세하게 해주셨기에 코드로 옮기는 데에는 크게 어렵지 않았지만 이론 내용을 이해하는데 조금 난이도가 있었다고 생각한다. 앞으로 남은 기간도 열심히 해야겠다.