Terceira Prova de 2013.1 — Linguagens de Programação

Fabio Mascarenhas

07 de Agosto de 2013

A prova é individual e sem consulta. Responda as questões na folha de respostas, a lápis ou a caneta. Se tiver qualquer dúvida consulte o professor.

Nome: _____

DRE: _____

Questão:	1	2	3	4	Total
Pontos:	2	3	3	2	10
Nota:					

1. (2 pontos) Uma linguagem com funções de primeira classe não precisa de `let` como uma operação primitiva. Uma expressão `let` pode ser açúcar sintático para a criação e chamada de uma função anônima:

```
let <nome> = <e1> in      (fun (nome)
  <e2>                =>     <e2>
end                           end)(<e1>)
```

   Os casos abaixo dão o fragmento do tipo algébrico `Exp` das expressões da linguagem acima resposável pelas expressões `let`, declarações de função e aplicações de função:

```
case class Let(nome: String, e1: Exp, e2: Exp) extends Exp
case class Fun(param: String, corpo: Exp) extends Exp
case class Ap(fun: Exp, arg: Exp) extends Exp
```

   Dê o caso da função `desugar` para `Let`, que transforma uma expressão `let` em uma aplicação de uma função anônima.

2. (3 pontos) A transformação da questão 1 pode ser extendida facilmente para expressões `let` de múltiplas variáveis, em linguagens com funções de múltiplos parâmetros:

```
let <n1> = <e1>,       (fun (n1, ..., ni)
    ...,                  <ec>
    <ni> = <e1> in  =>  end)(<e1>,
  <ec>                        ...,
end                           <ei>)
```

   Os casos abaixo dão o fragmento do tipo algébrico `Exp` das expressões da linguagem acima resposável pelas expressões `let`, declarações de função e aplicações de função:

```
case class Let(nomes: List[String], es: List[Exp], ec: Exp) extends Exp
case class Fun(params: List[String], corpo: Exp) extends Exp
case class Ap(fun: Exp, args: List[Exp]) extends Exp
```

Dê o caso da função `desugar` para `Let`, que transforma uma expressão `let` em uma aplicação de uma função anônima.

3. (3 pontos) O código abaixo é o trecho da função `eval` do interpretador de uma linguagem funcional com funções de primeira classe de um parâmetro, responsável pela aplicação dessas funções:

```
case Ap(fun, arg) => fun.eval(env) match {
  case FunV(fenv, param, corpo) =>
    corpo.eval(env + (param -> arg.eval(fenv))
  case _ => sys.error("não é uma função")
}
```

O código acima possui dois bugs relacionados ao escopo de variáveis; mostre um trecho de código (use a sintaxe de *fun*, e pode assumir a presença de expressões aritméticas e uma expressão `let` que funciona corretamente) que exercita os bugs, dizendo qual o resultado correto e qual o dado pelos bugs, depois reescreva o código acima para corrigir os bugs.

4. (2 pontos) O trecho de Scala abaixo é de um interpretador para *microc*, escrito usando manipulação direta da memória ao invés da abstração de *ações*:

```
case Soma(e1, e2) => mem => {
  val (v1, mem1) = e1.eval(funs)(env)(mem)
  val (v2, mem2) = e2.eval(funs)(env)(mem)
  (v1, v2) match {
    case (NumV(n1), NumV(n2)) => (NumV(n1 + n2), mem2)
    case _ => sys.error("soma precisa de dois números")
  }
}
```

O trecho apresenta um bug de linearidade no uso da memória; escreva um programa *microc* que mostra o problema, dizendo qual o resultado do programa no interpretador com o bug e qual o resultado correto (assuma que o restante do interpretador está correto), depois reescreva o trecho para eliminar o bug de linearidade.

BOA SORTE!