Práctica 6
Excepciones y Continuaciones en la Máquina $\mathcal{K}$

Sandra del Mar Soto Corderi Edgar Quiroz Castañeda

22 de noviembre de 2019

1. La Maquina $\mathcal{K}$

1.1. Marcos

Para modelar la máquina abstracta $\mathcal{K}$, en necesario tener una estrucutra que guarde los cómputos pendientes, llamados marcos. Hay un marco por cada posible cómputo pendiente de algún operador.

```
-- | Tipo de marcos vacíos
type Pending = ()

-- | Tipo de marco
data Frame = SuccF Pending
           | PredF Pending
           | ...
           | RaiseF Pending
           | HandleF Pending Identifier Expr
           | ContinueFL Pending Expr
           | ContinueFR Expr Pending
          deriving (Eq)
```

- `instance Show Frame`

  Sólo se tomó un el inicio del nombre del operador correspondiente, remplazando el cómputo pendiente con un guión `-`.

```
-- | Show para marcos
instance Show Frame where
    show ex =
    case ex of
        (SuccF _) -> "suc(-)"
        (PredF _) -> "pred(-)"
        ...
        (RaiseF _) -> "raise(-)"
        (HandleF _, x, e2) -> "handle(-, " ++ (show x) ++ ", " ++ (show e2) ++ ")"
        (ContinueFL _ e) -> "continue(-, " ++ (show e) ++ ")"
        (ContinueFR e _) -> "continue(" ++ (show e) ++ ", -)"
```

1.2. Estados

Ahora que hay que tener un registro de los cómputos pendientes, los estados al momento de evaluar expresiones cambian. Hay tres posibles estados: evaluando, terminando, y error.

```
-- | Tipo para estados
data State = E (Memory, Stack, Expr) | R (Memory, Stack, Expr)
      | P (Memory, Stack, Expr)
```

- `instance Show State`

  En los ejemplos en la descripción de la práctica, los estados se mostraban directamente como están definidos.

  Así que para mantener este formato de manera sencilla, sólo se agregó que los estados derivaran su instancia de `Show` de su definición.

```haskell
-- | Tipo para estados
data State = E (Memory, Stack, Expr) | R (Memory, Stack, Expr)
  | P (Memory, Stack, Expr)
  deriving (Show)
```

2. Excepciones y Continuaciones

- eval1
- evals
- eval

3. Integración

- frVars

```haskell
-- | Obteniendo variables libres de una expresion(AddFL _ e) -> "add(-, " ++ (show e) ++ ")"
frVars :: Expr -> [Identifier]
frVars ex =
        case ex of
                ...
                (Void) -> []
                (Seq e f) -> union (frVars e) (frVars f)
                (While e f) -> union (frVars e) (frVars f)
                (Raise e) -> frVars e
                (Handle e i f) -> union (frVars e) ((frVars f) \\ [i])
                (LetCC i e) -> (frVars e) \\ [i]
                (Continue e f) -> union (frVars e) (frVars f)
                (Cont s) -> []
```

- subst

```haskell
    -- | Aplicando substitucion si es semanticamente posible
subst :: Expr -> Substitution -> Expr
subst ex s@(y, e') =
        case ex of
                ...
                (Seq e f) -> Seq (st e) (st f)
                (While e f) -> While (st e) (st f)
                (Raise e) -> Raise (st e)
                (Handle e x f) ->
                        if x == y || elem x (frVars e')
                                then st (alphaExpr ex)
                        else Handle e x (st f)
                (LetCC x e) ->
                        if x == y || elem x (frVars e')
                                then st (alphaExpr ex)
                        else LetCC x (st e)
                (Continue e f) -> Continue (st e) (st f)
                (Cont a) -> Cont a
                where st = (flip subst) s
```

- alphaEq

```haskell
-- | Dice si dos expresiones son alpha equivalentes
alphaEq :: Expr -> Expr -> Bool
...
alphaEq (Seq e1 e2) (Seq f1 f2) = (alphaEq e1 f1) && (alphaEq e2 f2)
alphaEq (While e1 e2) (While f1 f2) = (alphaEq e1 f1) && (alphaEq e2 f2)
```

```
alphaEq (Raise e) (Raise f) = alphaEq e f
alphaEq (Handle e1 x e2) (Handle f1 y f2) = (alphaEq e1 f1) && (alphaEq e2 (subst f2 (y, (V x))))
alphaEq (LetCC x e) (LetCC y f) = (alphaEq e (subst f (y, (V x))))
alphaEq (Continue e1 e2) (Continue f1 f2) = (alphaEq e1 f1) && (alphaEq e2 f2)
alphaEq (Cont s1) (Cont s2) = s1 == s2
alphaEq _ _ = False
```

Dificultades

Nos fue difícil la implementación del eval1 ya que algunas definiciones nos causaban confusión y la longitud del código era bastante larga, por lo que era fácil que hubieran errores en el código.Igual fue complicado manejar los casos que manipulaban las continuaciones y la memoria a la vez.