

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 Máquina \mathcal{K}

1.1. Marcos

Para modelar la máquina abstracta \mathcal{K} , es necesario tener una estructura que guarde los cálculos pendientes, llamados marcos. Hay un marco por cada posible cálculo 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, reemplazando el cálculo 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 cálculos 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.

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

2. Excepciones y Continuaciones

■ eval1

Para esta función, había que traducir la función equivalente anterior que se usó para evaluar expresiones usando memoria. Hay diferentes casos por estado.

• E (Memory, Stack, Expr)

Para estas expresiones, si ya se llegó a un valor en todas las expresiones necesarias para determinar el operador, entonces se pasa al estado R (Memory, Stack, Expr) con el valor obtenido por el operador.

Si aún falta algo por evaluar, entonces se bota el marco actual y se mete el marco necesario para seguir con la evaluación.

```
eval1 (E (m, s, e)) =
  case e of
    -- valores
    (V _) -> R (m, s, e)
    ...
    -- operadores unarios
    (Succ e1) -> E (m, (SuccF ()):s, e1)
    ...
    -- operadores binarios
    (Add e1 e2) -> E (m, (AddFL ()) e2):s, e1)
    ...
    -- otros
    (If e1 e2 e3) -> E (m, (IfF ()) e2 e3):s, e1)
    (Let x e1 e2) -> E (m, (LetF x ()) e2):s, e1)
    (Letcc x e1) -> E (m, s, subst e1 (x, Cont s))
    -- error
    _ -> P (m, s, e)
```

• R (Memory, Stack, Expr)

Se tiene que regresar un valor. Dependiendo del valor y del cómputo pendiente en el tope de la pila de marcos, se realiza alguna operación.

Como evaluar la expresión resultante, devolver un valor, o manejar la memoria.

Estas operaciones fueron una traducción casi directa de las definiciones en la versión anterior de evals.

Si no se puede realizar ninguna operación, se propaga un error.

Se muestran algunos ejemplos de transición de este estado.

```
eval1 (R (mem, s, e)) =
  case e of
    ...
    (I m) ->
      case s of
        ((SuccF _) : s') -> R (mem, s', I (succ m))
        ...
        ((ContinueFR (Cont s') _) : s') -> R (mem, s', e)
        _ -> P (mem, s, Raise e)
    (L i) ->
      ...
      ((AssigFL _ e2) : s') -> E (mem, ((AssigFR e ()):s'), e2)
      ((AssigFR (L i) _) : s') ->
        case update (i, e) mem of
          Just mem' -> R (mem', s', Void)
          Nothing -> P (mem, s, Raise e)
```

```

...
(Void) ->
  case s of
    ...
    ((SeqF _ e2) : s') -> E (mem, s', e2)
    ...
(Cont st) ->
  case s of
    ...
    ((ContinueFL _ e2):s') -> E (mem, ((ContinueFR e ()):s'), e2)
    ((ContinueFR (Cont s'') _) : s') -> R (mem, s'', e)
    _ -> P (mem, s, Raise e)
...

```

Es la sección más larga de la función.

- P (Memory, Stack, Expr)

Hay dos casos.

Si no se está lidiando con errores, entonces sólo se sacan todos los valores de la pila de marcos hasta llegar al marco vacío. Llegar a este punto significa un error en tiempo de ejecución.

Si se está lidiando con errores (usando handle), entonces se desenvuelve el valor de error y se para a la expresión que se usará para recuperarse del error.

```

eval1 (P (mem, s, e)) =
  case s of
    -- lidiando con errores
    (HandleF _ x e1):s' ->
      case e of
        (Raise e1) -> E (mem, s', subst e1 (x, e1))
        _ -> P (mem, s, Raise e)
    -- dejando pasar errores
    (_:s') -> P (mem, s', e)

```

- evals

El propósito de esta función es dar pasos en la evaluación usando eval1 hasta llegar a un estado de fin de evaluación R (mem, [], v) o un estado de error P (mem, [], e).

En general, hay que vaciar la pila de marcos. Así que simplemente hay que evaluar el estado hasta que la pila de marcos esté vacía y la expresión esté bloqueada.

```

-- / Evaluar un estado exhaustivamente
evals :: State -> State
evals s =
  case eval1 s of
    s'@(E (_, [], e')) -> if blocked e' then s' else evals s'
    s'@(E (_, _, e')) -> evals s'
    s'@(R (_, [], e')) -> if blocked e' then s' else evals s'
    s'@(R (_, _, e')) -> evals s'
    s'@(P (_, [], e')) -> if blocked e' then s' else evals s'
    s'@(P (_, _, e')) -> evals s'

```

- eval

Utiliza evals para evaluar lo más posible una expresión. Si es un valor, se regresa. Si es otra cosa, se lanza un error.

```

eval :: Expr -> Expr
eval e =
  case evals (E ([], [], e)) of
    R (_, [], e') ->
      case e' of
        B _ -> e'
        I _ -> e'
        _ -> error "invalid final value"
    _ -> error "no value was returned"

```

3. Integración

Se tuvo que hacer modificaciones al módulo Syntax para incluir las expresiones concernientes a errores y continuaciones.

```
-- / Tipo para las expresiones
data Expr = ...
    | Raise Expr
    | Handle Expr Identifier Expr
    | Letcc Identifier Expr
    | Continue Expr Expr
    | Cont Stack
    | Error
    deriving (Eq)
```

- frVars
- subst
- alphaEq

Dificultades