Verifica del Software

Project 17 (Interpreter, Call-by-name version)

Nicola Carlesso

UniPd - Informatica

November 2, 2020

Haskell

- Pure Functional Language
- Lazy Evaluation
- Pattern Matching
- Tail Recursion

The REC language grammar

- The grammar of REC language:
  $t ::= n \mid var \mid t_1 + t_2 \mid t_1 - t_2 \mid t_1 * t_2 \mid \text{ if } t_0 \text{ then } t_1 \text{ else } t_2 \mid f_i(t_1, \ldots, t_{ai})$
- The grammar of REC language with respect to the precedence of operations:

$$expr ::= term + expr | term - minus | term$$

$$minus ::= term - minus | term \qquad Es.(5-2)-1 \neq 5-(2-1)$$

$$term ::= factor * term | factor$$

$$factor ::= \mathrm{n} \mid \mathrm{var} \mid t_1 + t_2 | t_1 - t_2 | t_1 * t_2 | \text{if } t_0 \text{then } t_1 \text{else } t_2 | f_i(t_1, \ldots, t_{ai})$$

Further more:

$$func ::= var(varn)$$

$$n ::= \underline{undef}|0|1|2|\ldots$$

$$varn ::= var, varn|var$$

$$var ::= \text{“any character”}\, varch$$

$$varch ::= \text{“any character”} \mid \text{“any digit”} \mid \_ \mid varch|\epsilon$$

- The grammar of the input:

$$
\begin{aligned}
prog &::= funcn; expr; decn; \\
funcn &::= func, funcn | func \\
decn &::= dec, decn | dec \\
dec &::= var = expr
\end{aligned}
$$

```haskell
-- Parse the entire program
parseProg :: Parser Program
parseProg = do
    fd <- many parseFuncDec
    symbol ";"
    e <- parseExpr
    symbol ";"
    vs <- many parseVarDec
    symbol ";"
    return (fd, e, vs)
```

Example

$f_1(x_1, x_2) = x_1, f_2(x_1) = x_1 + 2, f_3() = f_3() + 1; 3 + f_1(x_1 + 2, y); x_1 = 2, x_2 = 3, y = \underline{undef};$

The Parser

([("f_1", [(Evar "x_1"), (Evar "x_2")], (Evar "x_1")), ("f_2", [(Evar "x_1")], (EOp (Evar "x_1") PL (Enum (Just 2)))), ("f_3", [], (EOp (EFunc "f_3" []) P (ENum 1)))], (EOp (ENum 3) P (EFunc "f_1" [(EOp (EVar "x_1") P (ENum 2)), (EVar "y")])), [("x_1", (Just 2)), ("x_2", (Just 3)), ("y", Nothing)])

```haskell
type Variable = String

data Expr
    = EVar Variable
    | ENum (Maybe Int)
    | ECond
        Expr
        Expr
        Expr
    | EOp
        Expr
        Op
        Expr
    | EFunc
        Variable
        [Expr]
    deriving Show

data Op = PL | MI | ML
    deriving Show

type Def = (Variable, (Maybe Int))
type VEnv = [Def]

type FuncDec= (Variable, [Variable], Expr)

type Program = ([FuncDec], Expr, VEnv)
```

The Interpreter

Initially a simple translation job ...

$$
\begin{aligned}
[\![n]\!]_{na} &= \lambda\varphi\lambda\rho.\ \lfloor n \rfloor \\
[\![x]\!]_{na} &= \lambda\varphi\lambda\rho.\ \rho(x) \\
[\![t_1 \ \mathbf{op}\ t_2]\!]_{na} &= \lambda\varphi\lambda\rho.\ [\![t_1]\!]_{na}\varphi\rho \ op_\perp \ [\![t_2]\!]_{na}\varphi\rho \\
& \quad \text{where } \mathbf{op} \text{ is } +, -, \text{ or } \times \\
[\![\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2]\!]_{na} &= \lambda\varphi\lambda\rho.\ Cond([\![t_0]\!]_{na}\varphi\rho, [\![t_1]\!]_{na}\varphi\rho, [\![t_2]\!]_{na}\varphi\rho) \\
[\![f_i(t_1, \ldots, t_{a_i})]\!]_{na} &= \lambda\varphi\lambda\rho.\ \varphi_i([\![t_1]\!]_{na}\varphi\rho, \ldots, [\![t_{a_i}]\!]_{na}\varphi\rho)
\end{aligned}
$$

$$\downarrow$$

```haskell
valueExpr :: ProgramParsed -> Maybe Int
valueExpr (funcn, expr, decn) = case expr of
    (EVar v)           -> valueVar decn v
    (ENum n)           -> n
    (EOp t1 op t2)     -> valueOp op (valueExpr (funcn, t1, decn)) (valueExpr (funcn, t2, decn))
    (ECond t0 t1 t2)   -> valueCond (valueExpr (funcn, t0, decn)) (valueExpr (funcn, t1, decn)) (valueExpr (funcn, t2, decn))
    (EFunc f params)   -> let (_, fun) = getFunc funcn f
                           in fun (valueParams funcn decn params)
```

The Functional

$$
\begin{aligned}
F(\varphi) \quad = \quad & (\lambda z_1, \ldots, z_{a_1} \in \mathbf{N}_\perp . \, [\![d_1]\!]_{na} \varphi \rho [z_1/x_1, \ldots, z_{a_1}/x_{a_1}], \\
& \qquad\qquad \vdots \\
& \lambda z_1, \ldots, z_{a_k} \in \mathbf{N}_\perp . \, [\![d_k]\!]_{na} \varphi \rho [z_1/x_1, \ldots, z_{a_k}/x_{a_k}]).
\end{aligned}
$$

$$\downarrow$$

```
functional :: [FuncDec] -> VEnv -> (FEnv -> FEnv)
functional [] _ = \_ -> []
functional ((name, params, exp):fs) venv = \fenv ->
    (name, (\inp -> valueExpr (fenv, exp, (replaceVars venv params inp)))) : (functional fs venv fenv)
```

The Knaster-Tarski-Kleene theorem

Theorem 4.37 Let f: $D \to D$ be a continuous function on the ccpo $(D, \sqsubseteq)$ with least element $\perp$. Then

$$\text{FIX } f = \bigsqcup\{ f^{\text{n}} \perp \mid \text{n} \geq 0 \}$$

defines an element of D and this element is the least fixed point of f.

Here we have used that

$$f^0 = \text{id, and}$$

$$f^{\text{n}+1} = f \circ f^{\text{n}} \text{ for n} \geq 0$$

$$\downarrow$$

```haskell
-- Use the tail recursive technique
rho :: (FEnv -> FEnv) -> Int -> (FEnv -> FEnv)
rho _ 0 = id
rho f k = \funcn -> rho f (k - 1) (f funcn)
```

```haskell
bottom :: Func
bottom = ("bottom", \_ -> Nothing)
```

The Tail Recursion

- Definition of Functional function:

$$\texttt{factorial 0 r} = \texttt{r}$$
$$\texttt{factorial n r} = \texttt{factorial (n - 1) (r * n)}$$

- Execution steps of an example:

$$\begin{aligned}\texttt{facAux 5 1} &= \texttt{factorial 4 5}\\ &= \texttt{factorial 3 20}\\ &= \texttt{factorial 2 60}\\ &= \texttt{factorial 1 210}\\ &= \texttt{factorial 0 120}\\ &= 120\end{aligned}$$

The main functions

```haskell
-- Use the tail recursive technique
interpreter' :: Int -> Program -> Int
interpreter' n input = let findF = findFix input
    in case (findF n) of
        Just n -> n
        Nothing -> interpreter' (n + 1) input
```

```haskell
-- Find fix points of FEnv whether they are request
findFix :: Program -> Int -> Maybe Int
findFix (funcn, t, decn) = \k -> let fix = rho (functional funcn decn) k [bottom]
                                 in valueExpr (fix, t, decn)
```