FUN: a small functional language

Pedro Vasconcelos

February 27, 2023

Defining a small functional language

1. Pick an evaluation strategy:
   - applicative order (*call-by-value*) or normal order (*call-by-name*/*lazy evaluation*)
   - weak normal forms (no reductions inside λs)
2. Add primitive constants and operations:
   - numbers, arithmetic operations
   - booleans, characters, ...
3. Add data structures:
   - pairs, tuples, lists
   - user-defined algebraic data types

Plan

1. The FUN language

2. Operational semantics
   - Evaluator with substitutions
   - Evaluator with environments
   - Evaluator in CPS

Plan

1. The FUN language

2. Operational semantics
   - Evaluator with substitutions
   - Evaluator with environments
   - Evaluator in CPS

A small functional language

- Evaluation by aplicative order (*call-by-value*)
- Primitive integers
- Arithmetic operations $(+, -, \times)$
- Conditional expression **ifzero**
- Local definitions **let**. . . **in**. . .
- Recursive definitions using a fixed-point primitive **fix**
- A program is an expression with no free variables
- No other data types
- Pure: no input/output

Syntax of expressions

$$
\begin{array}{rcll}
e & ::= & x & \text{variables} \\
 & | & \lambda x.\, e & \text{abstraction} \\
 & | & e_1 \; e_2 & \text{aplication} \\
 & | & n & \text{integers} \\
 & | & e_1 + e_2 & \text{operations} \\
 & | & e_1 - e_2 & \\
 & | & e_1 \times e_2 & \\
 & | & \textbf{ifzero}\; e_0 \; e_1 \; e_2 & \text{conditional} \\
 & | & \textbf{let}\; x = e_1 \;\textbf{in}\; e_2 & \text{local definition} \\
 & | & \textbf{fix}\; e & \text{fixed-point}
\end{array}
$$

Example: conditions

$$\lambda x.\ \lambda y.\ \textbf{ifzero}\ x\ y\ x$$

Exemplo: higher-order

$$\begin{array}{l} \textbf{let } \mathit{twice} = \lambda f.\ \lambda x.\ f\ (f\ x) \\ \textbf{in } \mathit{twice}\ (\lambda x.\ x + 1)\ 42 \end{array}$$

Exemplo: recusive factorial

$$\begin{array}{l}\textbf{let } \mathit{fact} = \textbf{fix } \lambda f.\ \lambda n.\ \textbf{ifzero } n\ 1\ (n \times f\ (n-1)) \\ \textbf{in } \mathit{fact}\ 10\end{array}$$

Plan

1. The FUN language
2. Operational semantics
   - Evaluator with substitutions
   - Evaluator with environments
   - Evaluator in CPS

Operational semantics

Evaluation relation

$$e \Downarrow v \qquad\qquad \textit{e reduces to v}$$

Values: weak normal forms

$$\begin{array}{rcll} v & ::= & n & \text{integers} \\ & \mid & \lambda x.\, e & \text{closed terms}: \; FV(\lambda x.\, e) = \emptyset \end{array}$$

Evaluation with substitutions

Values and applications

$$\frac{}{n \Downarrow n} \qquad n \in \mathrm{Int} \tag{1}$$

$$\frac{}{\lambda x.\, e \Downarrow \lambda x.\, e} \qquad FV(\lambda x.\, e) = \emptyset \tag{2}$$

$$\frac{e_1 \Downarrow \lambda x.\, e' \qquad e_2 \Downarrow v \qquad e'[v/x] \Downarrow u}{e_1\ e_2 \Downarrow u} \tag{3}$$

Evaluation with substitutions

Conditional

$$\frac{e_0 \Downarrow 0 \qquad e_1 \Downarrow v}{\textbf{ifzero}\ e_0\ e_1\ e_2 \Downarrow v} \tag{4}$$

$$\frac{e_0 \Downarrow n \qquad e_2 \Downarrow v}{\textbf{ifzero}\ e_0\ e_1\ e_2 \Downarrow v} \quad n \in \mathrm{Int} \wedge n \neq 0 \tag{5}$$

Evaluation with substitutions

Primitive operations

$$\frac{e_1 \Downarrow n \qquad e_2 \Downarrow m}{e_1 + e_2 \Downarrow k} \quad n, m \in \mathrm{Int} \wedge k = n + m \tag{6}$$

Similar rules for $-$ e $\times$.

Evaluation with substitutions

Local definitions and fixed-point operator

$$\frac{(\lambda x.\, e_2)\; e_1 \Downarrow v}{\textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow v} \tag{7}$$

$$\frac{}{\textbf{fix } \lambda f.\, \lambda x.\, e \Downarrow \lambda x.\, e[(\textbf{fix } \lambda f.\, \lambda x.\, e)/f]} \tag{8}$$

Observations

- No reduction when no rule applies (e.g. if *e* is a free variable)
- Applications only replace variables by values
  - no free variables;
  - hence, no variable capture

$$((\lambda x\, M)\, N) \rightarrow_\beta M[N/x] \qquad \text{where } BV(M) \cap \underbrace{FV(N)}_{=\emptyset} = \emptyset$$

Abstract syntax in Haskell

```
data Term = Var Ident
          | Lambda Ident Term
          | App Term Term
          | Const Int
          | Term :+ Term
          | Term :- Term
          | Term :* Term
          | IfZero Term Term Term
          | Let Ident Term Term
          | Fix Term

type Ident = String  -- names
type Value = Term    -- special case of Term
```

Evaluator in Haskell

Defined by recursion over terms:

```
eval1 :: Term -> Value
```

Auxiliary functions:

```
apply :: Value -> Value -> Value            -- beta-reduction
subst :: Term -> Ident -> Value -> Term     -- substitution
primitive ::  (Int->Int->Int) -> Value -> Value -> Value
                                            -- aritthmetic operations
```

(Cue demo. . .)

Observations

- We use the operator `$!` to force evaluation of an argument

$$f \,\$!\, x = f\ x \text{ but } \textit{always} \text{ evaluates } x$$

- Otherwise: FUN would inherit the non-strict semantics of the meta-interpreter (Haskell).[1]

[1] cf. John Reynolds, *"Definitional Interpreters and Higher-Order Programming Languages"*.

Plan

Eliminating substitutions

Problem `eval1` performs traversals over expressions to replace variables one at a time

Solution instead of doing the substititions, we will record the assignments of values to free variables in an environment

Eliminating substitutions

Problem `eval1` performs traversals over expressions to replace variables one at a time

Solution instead of doing the substititions, we will record the assignments of values to free variables in an environment

Environment

Associations of values to variables:

$$\rho = [x_1 \mapsto v_1,\ x_2 \mapsto v_2,\ \ldots,\ x_n \mapsto v_n]$$

dom ρ	the domain ρ (i.e. a set of variables)
ρ_x	the value associated to x in environment ρ (if $x \in \text{dom } \rho$);
$\rho[x \mapsto v]$	the environment associating v to x and acting as ρ for other variables

Closures

If we don't perform substitutions λ-expressions may contain free variables; e.g.

$$\begin{array}{ll} \textbf{let } \mathrm{mult} = \lambda x.\, \lambda y.\, x \times y \qquad & \textbf{let } \mathrm{mult} = \lambda x.\, \lambda y.\, x \times y \\ \textbf{in } \mathrm{mult}\ 2 & \textbf{in } \mathrm{mult}\ 10 \end{array}$$

A functional result must therefore be a *pair* of a λ-expression and an environment:

$$(\lambda y.\, x \times y,\ [x \mapsto 2]) \qquad (\lambda y.\, x \times y,\ [x \mapsto 10])$$

This representation is called as closure.

Evaluator with environments

Evaluation relation

$$\rho \vdash e \Downarrow v$$

Values

$$\begin{array}{rcll} v & ::= & n & \text{integers} \\ & \mid & (\lambda x.\, e,\ \rho) & \text{closures} \end{array}$$

Evaluation with environments

Values and applications

$$\frac{}{\rho \vdash n \Downarrow n} \qquad n \in \mathsf{Int} \tag{9}$$

$$\frac{}{\rho \vdash x \Downarrow \rho_x} \qquad x \in \mathrm{dom}\ \rho \tag{10}$$

$$\frac{}{\rho \vdash \lambda x.\, e \Downarrow (\lambda x.\, e,\ \rho)} \tag{11}$$

$$\frac{\rho \vdash e_1 \Downarrow (\lambda x.\, e',\ \rho') \qquad \rho \vdash e_2 \Downarrow v \qquad \rho'[x \mapsto v] \vdash e' \Downarrow u}{\rho \vdash e_1\ e_2 \Downarrow u} \tag{12}$$

Evaluation with environments

Conditional and primitive operations

$$\frac{\rho \vdash e_0 \Downarrow 0 \qquad \rho \vdash e_1 \Downarrow v}{\rho \vdash \textbf{ifzero}\ e_0\ e_1\ e_2 \Downarrow v} \tag{13}$$

$$\frac{\rho \vdash e_0 \Downarrow n \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash \textbf{ifzero}\ e_0\ e_1\ e_2 \Downarrow v} \quad n \in \mathrm{Int} \wedge n \neq 0 \tag{14}$$

$$\frac{\rho \vdash e_1 \Downarrow n \qquad \rho \vdash e_2 \Downarrow m}{\rho \vdash e_1 + e_2 \Downarrow k} \quad n, m \in \mathrm{Int} \wedge k = n + m \tag{15}$$

Evaluation with environments

Fixed-point operator

$$\frac{}{\rho \vdash \textbf{fix}\ \lambda f.\, \lambda x.\, e \Downarrow v} \qquad v = (\lambda x.\, e, \textcolor{red}{?})$$

What should be the enviroment for this closure?

Evaluation with environments

Fixed-point operator

$$\frac{}{\rho \vdash \textbf{fix}\ \lambda f.\, \lambda x.\, e \Downarrow v} \qquad v = (\lambda x.\, e,\ \rho[f \mapsto v])$$

Operational interpretation: build a *cyclic* closure.

Implementing evaluation with environments

```
-- environments
type Env = [(Ident,Value)]

-- values
data Value = Int Int
           | Closure Term Env
           deriving Show
```

Env and Value are mutually recursive.

Implementing evaluation with environments (cont.)

```
-- empty environment
[] :: Env

-- add an entry to the enviroment
(:) :: (Ident,Value) -> Env -> Env

-- lookup the value for an identifier
lookup :: Ident -> Env -> Maybe Value
```

Evaluator in Haskell

```
eval2 :: Term -> Env -> Value
```

No need for substitutions.

(Cue demo…)

Observations

- Same as before: we use `$!` to force applicative evaluation order
- The Haskell interpreter diverges if we try to show a cyclic closure

Evaluator in CPS

Let us now define an evaluator in which the evaluation order is explicit using continuation passing style (CPS).

Continuations

A continuation is a function that represents the rest of the computation.

Using higher-order functions, we can re-write any function to another that takes its continuation as an explicit argumen.

Continuations (cont.)

Example: the factorial function

```
fact :: Integer -> Integer
```

Let as define `factCPS` such that

```
factCPS n k = k (fact n)
```

where the continuation is the argument `k`.

Since the result of factorial is an integer, continuations have the following type:

```
type Cont = Integer -> Integer
```

Continuations (cont.)

Direct style

```
fact :: Integer -> Integer
fact n
  | n>0 = n * fact (n-1)
  | otherwise = 1
```

Continuation-passing style (CPS)

```
factCPS :: Integer -> Cont -> Integer
factCPS n k
  | n>0 = factCPS (n-1) (\r -> k (n*r))
  | otherwise = k 1
```

Continuations (cont.)

To compute factorial we pass the *identity function* as inicial continuation:

```
> factCPS 10 id
3628800
```

Note that `factCPS` is *tail-recursive* but `fact` is not.

Evaluator in CPS

Let us re-write the evaluator with environments in CPS:

```
eval3 :: Term -> Env -> Cont -> Value
```

Since the results are of type `Value`, continuations have the following type:

```
type Cont = Value -> Value
```

Evaluator in CPS (cont.)

In the CPS evaluator, the evaluation order is explicit, e.g.:

```
eval3 (App e1 e2) env k
  = eval3 e1 env (\v1 ->  -- evaluate e1
    eval3 e2 env (\v2 ->  -- evaluate e2
    apply v1 v2 k))

eval3 (e1 :+ e2) env k
  = eval3 e1 env (\v1-> -- evaluate e1
    eval3 e2 env (\v2-> -- evaluate e2
    k $ primop (+) v1 v2))
```