Basics of *interpretation*

with Haskell as the metalanguage

Ralf Lämmel
Software Language Engineering Team
University of Koblenz-Landau
http://www.softlang.org/

Interpretation for a simple expression language

```
▶ evaluate (Pred (If (IsZero Zero) (Succ Zero) Zero))
Left 0
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ evaluate (Pred TRUE)
Left *** Exception: ... Irrefutable pattern failed for pattern ...
```

N.B.:

- An interpreter is a metaprogram which executes or evaluates object programs.
- The expression language at hand is also referred to as **BTL** — Basic TAPL Language — where TAPL is a reference to Pierce's textbook 'Types and programming languages'.

Informal baseline for BTL interpretation

```
▶ evaluate (Pred (If (IsZero Zero) (Succ Zero) Zero))
Left 0
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ evaluate (Pred TRUE)
Left *** Exception: ... Irrefutable pattern failed for pattern ...
```

- *TRUE*: Evaluates to *True*.
- *FALSE*: Evaluates to *False*.
- *Zero*: Evaluates to *0*.
- *Succ e*: Evaluates to *n*+1, if *e* evaluates to *n*.
- *Pred e*:
  - Evaluates to *n*-1, if *e* evaluates to *n* and *n*>0.
  - Evaluates to 0, if *e* evaluates to 0,
- *IsZero e*: Evaluates to a Boolean for a test for *'0'*.
- *If e0 e1 e2*: Evaluates like a normal conditional.

The interpreter for BTL

```
▶ evaluate (Pred (If (IsZero Zero) (Succ Zero) Zero))
Left 0
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ evaluate (Pred TRUE)
Left *** Exception: ... Irrefutable pattern failed for pattern ...
```

```
type Value = Either Int Bool
```

```
evaluate :: Expr → Value
evaluate TRUE = Right True
evaluate FALSE = Right False
evaluate Zero = Left 0
evaluate (Succ e) = Left (n+1) where Left n = evaluate e
evaluate (Pred e) = Left (n − if n==0 then 0 else 1) where Left n = evaluate e
evaluate (IsZero e) = Right (n==0) where Left n = evaluate e
evaluate (If e0 e1 e2) = evaluate (if b then e1 else e2) where Right b = evaluate e0
```

Interpretation for **BIPL** — Basic Imperative Programming Language

```
-- // Compute quotient q and remainder r for dividing x by y
-- q = 0; r = x; while (r >= y) { r = r - y; q = q + 1; }
euclideanDiv :: Stmt
euclideanDiv =
  Seq (Assign "q" (IntConst 0)) (Seq (Assign "r" (Var "x"))
      (While
        (Binary Geq (Var "r") (Var "y"))
        (Seq (Assign "r" (Binary Sub (Var "r") (Var "y")))
             (Assign "q" (Binary Add (Var "q") (IntConst 1))))))
```

```
▶ execute euclideanDivision (fromList [("x", Left 13), ("y", Left 4)])
fromList [("q",Left 3),("r",Left 2),("x",Left 14),("y",Left 4)]
```

N.B.: Interpretation involves **stores**.

The interpreter for BIPL

```
-- Results of expression evaluation
type Value = Either Int Bool
-- Stores as maps from variable names to values
type Store = Map String Value
```

```
-- Execution of statements
execute :: Stmt → Store → Store
execute Skip m = m
execute (Assign x e) m = insert x (evaluate e m) m
execute (Seq s1 s2) m = execute s2 (execute s1 m)
execute (If e s1 s2) m = execute (if b then s1 else s2) m where Right b = evaluate e m
execute (While e s) m = execute (If e (Seq s (While e s)) Skip) m

-- Evaluation of expressions
evaluate :: Expr → Store → Value
evaluate (IntConst i) _ = Left i
evaluate (Var x) m = m!x
evaluate (Unary o e) m = uop o (evaluate e m)
evaluate (Binary o e1 e2) m = bop o (evaluate e1 m) (evaluate e2 m)

-- Interpretation of unary operators
uop :: UOp → Value → Value
uop Negate (Left i) = Left (negate i)
uop Not (Right b) = Right (not b)
```

Interpretation for **BFPL** — Basic Functional Programming Language

```
-- factorial :: Int -> Int
-- factorial x = if ((==) x 0) then 1 else ((*) x (factorial ((-) x 1)))
-- main = print $ factorial 5
factorial :: Program
factorial = ([(
  "factorial",
  (([IntType], IntType),
   (["x"],
     If (Binary Eq (Arg "x") (IntConst 0))
        (IntConst 1)
        (Binary Mul
             (Arg "x")
             (Apply "factorial" [Binary Sub (Arg "x") (IntConst 1)])))))],
  (Apply "factorial" [IntConst 5]))
```

```
▶ evaluate factorial
Left 120
```

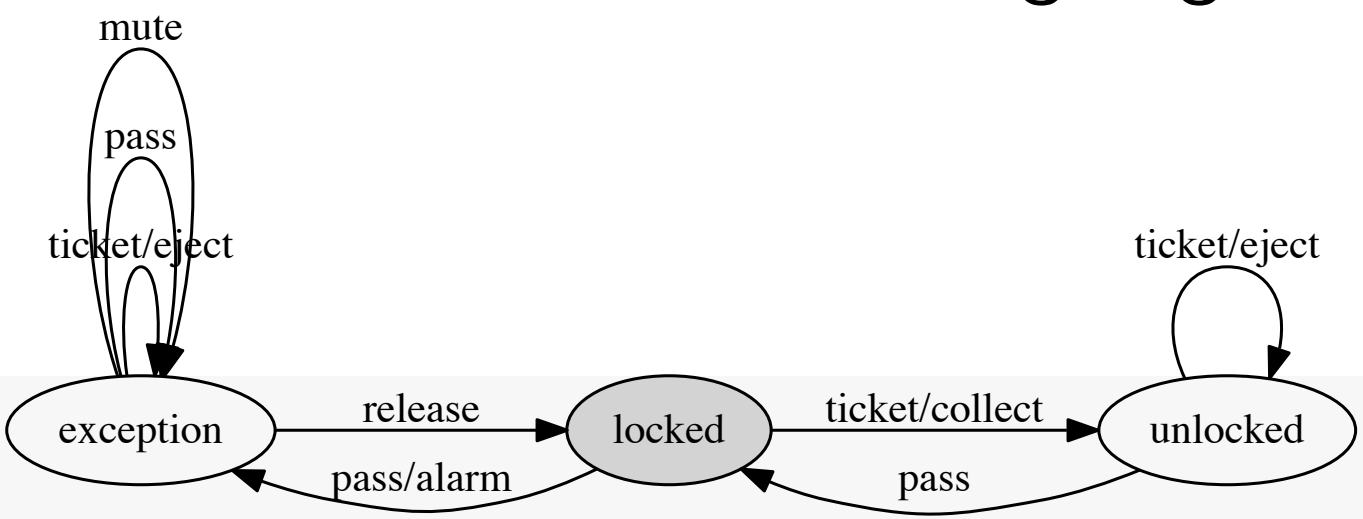
N.B.: Interpretation involves **environments**.

The interpreter for BFPL

```
-- Results of expression evaluation
type Value = Either Int Bool
-- Environments as maps from argument names to values
type Env = Map String Value
```

```
-- Evaluation of a program's main expression
evaluate :: Program → Value
evaluate (fs, e) = f e empty
 where
  -- Evaluation of expressions
  f :: Expr → Env → Value
  f (IntConst i) _ = Left i
  f (BoolConst b) _ = Right b
  f (Arg x) m = m!x
  f (If e0 e1 e2) m = f (if b then e1 else e2) m where Right b = f e0 m
  f (Unary o e) m = uop o (f e m)
  f (Binary o e1 e2) m = bop o (f e1 m) (f e2 m)
  f (Apply x es) m = f body m'
    where
      Just (_, (xs, body)) = lookup x fs
      vs = map (flip f m) es
      m' = fromList (zip xs vs)
```

Interpretation for **FSML** — Finite State Machine Language




```
turnstileFsm :: Fsm
turnstileFsm = Fsm [
  State True "locked" [
    (Transition "ticket" (Just "collect") "unlocked"),
    (Transition "pass" (Just "alarm") "exception") ],
  State False "unlocked" [
    (Transition "ticket" (Just "eject") "unlocked"),
    (Transition "pass" Nothing "locked") ],
  State False "exception" [
    (Transition "ticket" (Just "eject") "exception"),
    (Transition "pass" Nothing "exception"),
    (Transition "mute" Nothing "exception"),
    (Transition "release" Nothing "locked") ]
]
```

A turnstile FSM in a metro system

Expected I/O behavior of FSM interpretation

```
-- Sample input for sample FSM
sampleInput :: Input
sampleInput =
  [
    "ticket", -- Regular insertion of a ticket in locked state
    "ticket", -- Irregular insertion of a ticket in unlocked state
    "pass", -- Regular passage of turnstile in unlocked state
    "pass", -- Irregular attempt to pass turnstile in locked state
    "ticket", -- Irregular insertion of a ticket in exceptional state
    "mute", -- Mute exceptional state alarm
    "release" -- Return from exceptional to locked state
  ]

-- Expected output
sampleOutput :: Output
sampleOutput = ["collect", "eject", "alarm", "eject"]
```

```
▶ simulate turnstileFsm sampleInput == sampleOutput
True
```

N.B.: Interpretation commences in a **step-wise** manner.
Also, FSML is **domain-specific modeling language**.

The interpreter for FSML

```
-- FSM simulation starting from initial state
simulate :: Fsm → Input → Output
simulate (Fsm ss) xs = snd (foldl makeTransition (getInitial, []) xs)
  where
    -- Look up initial state
    getInitial :: StateId
    getInitial = ini
      where [State _ ini _] = [ s | s@(State initial _ _) ← ss, initial ]

    -- Process event; extent output
    makeTransition :: (StateId, Output) → Event → (StateId, Output)
    makeTransition (source, as) x = (target, as ++ maybeToList a)
      where (Transition _ a target) = getTransition source x

    -- Look up transition
    getTransition :: StateId → Event → Transition
    getTransition sid x = t
      where
        [t] = [ t | t@(Transition x' _ _) ← ts, x == x' ]
        [(State _ _ ts)] = [ s | s@(State _ sid' _) ← ss, sid == sid' ]
```

Online resources

YAS (Yet Another SLR (Software Language Repository))

http://www.softlang.org/yas

YAS' GitHub repository contains all code.

See languages BTL, BIPL, BFPL, and FSML.

There are Haskell directories with, for example, interpreters.

The Software Languages Book

http://www.softlang.org/book

The book discusses interpretation in detail.

Other related subjects:

operational and denotational semantics.