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

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’.

---

<table>
<thead>
<tr>
<th>evaluate</th>
<th>(Pred (If (IsZero Zero) (Succ Zero) Zero))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>evaluate</th>
<th>(Pred TRUE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left *** Exception: ... Irrefutable pattern failed for pattern ...</td>
<td></td>
</tr>
</tbody>
</table>

---

**Recipe 5.1** (Interpretation for a given object language).

**Metalanguage** Pick the metalanguage for the interpreter. (This is going to be Haskell in our case.)
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 ...

```haskell
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
```

---

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
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

define 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

define 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

define uop :: UOp → Value → Value
uop Negate (Left i) = Left (negate i)
uop Not (Right b) = Right (not b)
N.B.: Interpretation involves environments.
The interpreter for BFPL

Results of expression evaluation

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

Environments as maps from argument names to values

```haskell
type Env = Map String Value
```

Evaluation of a program's main expression

```haskell
evaluate :: Program → Value
evaluate (fs, e) = f e empty
  where
    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)
```
5.1 Interpretation

Interpretation of binary operators

\[
\text{bop} \cdot \text{Value} \cdot \text{Value} \cdot \text{Value}
\]

\[
\text{bop Add (Left i1) (Left i2) = Left (i1+i2)}
\]

Exercise 5.6

(Parameterized procedures)

Intermediate level

Exercise 5.5 is extended as follows. A procedure declares variables for formal parameters and the statement form for calling a procedure includes a list of expressions as actual arguments.

5.1.4 Stepwise interpretation

Let us now discuss interpretation for finite state machines according to the fabricated FSML language; as discussed earlier. When compared to the previous examples, FSML is a modeling language rather than a programming language. Perhaps more importantly, interpretation of FSMs intrinsically calls for stepwise execution, as we will see in a second. Here is the recurring turnstile FSM represented as a Haskell term.

Illustration 5.9

(A FSM for a turnstile in a metro system)

Haskell module Language.FSML.Sample

\[
\text{turnstileFsm :: Fsm}
\]

\[
\text{turnstileFsm = Fsm [}
\]

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") ]

We expect that interpretation consumes events one by one. In each step, the corresponding action, if any, is produced as output and the machine makes a transition to the next state. Here is the informal semantics:

- To start the simulation, we need to determine the initial state of the FSM.

A turnstile FSM in a metro system
Expected I/O behavior of FSM interpretation

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

```haskell
-- 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 :: Stateld
    getInitial = ini
      where [State _ ini _] = [ s | s@State initial _ _) <- ss, initial ]

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

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

Exercise 5.7 (Interpretation for FSML without throwing).

Exercise 5.8 (A more abstract FSML syntax).

[Basic level] The interpreter 'throws' when attempting a transition to an undeclared state, when a given event is not handled in a given state or not handled deterministically. Revise the interpreter so that it returns Nothing of Haskell's Maybe type in these cases.

[Intermediate level] There is an opportunity for a more abstract Haskell-based abstract syntax for FSML. Please note that ids of the declared states must be distinct. Further, let us assume that only deterministic FSMs are permitted, which implies that
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.