Shrimp

an interpreted imperative programming language that wants to believe

Lorenzo Loconte

Introduction

Shrimp is a very simple and didactic imperative programming language designed during the course of *Formal Methods for Computer Science* at Universita' degli Studi di Bari Aldo Moro. **Shrimp** uses an *eager evaluation strategy*. In order to ensure that, the interpreter executes the code using the *call by value* method.

Software Modules

The program is composed by three main components:

- The **parser**
- The **optimizer**
- The **interpreter**

The **parser** takes in input the source code and convert it into an intermediate representation. The intermediate representation have the structure of a n-ary tree having the non-terminals of the grammar as internal nodes and commands, identifiers and integer constants on the leaves.

The **optimizer** takes in input the intermediate representation given by the parser. The result of the optimizer is an *optimized* intermediate representation. It evaluates the constant expressions (both arithmetic and boolean) that might be present in the source code and replace them with the resulting constants. The optimizer also checks for empty command blocks and useless branch statements and removes (or optimize) them.

The **interpreter** execute the semantics present in an intermediate representation. The basic idea is to use a **state** (or environment) that collects the values of the variables during the execution of the program. The result of the interpretation is the resulting state, that is a set of ground integer assignments to the variables.

The Language Syntax

The syntax for the **Shrimp** programming language is a context-free grammar. So, it can be denoted using EBNF (Extended Backus Naur Form) as following:

```
Type ::= "int"
Integer ::= [0-9]+
Identifier ::= [a-zA-Z_]+
Program ::= "shrimp" Block
Block ::= [Command]*
Command ::= {Assignment | Branch | Loop}
Assignment ::= Identifier "=" ArithmeticExpr ";"
Branch ::= "if" "(" BooleanExpr ")" "then" Block
           ["else" Block] "end if" ";"
Loop ::= "while" "(" BooleanExpr ")" "do"
         Block "end while" ";"

ArithmeticExpr ::=
    ArithmeticTerm "+" ArithmeticExpr
  | ArithmeticTerm "-" ArithmeticExpr
  | ArithmetciTerm
ArithmeticTerm ::=
    ArithmeticFactor "*" ArithmeticTerm
  | ArithmeticFactor "/" ArithmeticTerm
  | ArithmeticFactor "%" ArithmeticTerm
  | ArithmeticFactor
ArithmeticFactor ::=
    Integer
  | Identifier
  | "-" ArithmeticExpr
  | "(" ArithmeticExpr ")"

BooleanExpr ::=
    BooleanTerm "or" BooleanExpr
  | BooleanTerm
BooleanTerm ::=
    BooleanFactor "and" BooleanTerm
  | BooleanFactor

BooleanFactor ::=
    "true"
  | "false"
  | "not" BooleanExpr
  | ArithmetciExpr "eq" ArithmeticExpr
  | ArithmetciExpr "neq" ArithmeticExpr
  | ArithmetciExpr "lt" ArithmeticExpr
  | ArithmetciExpr "gt" ArithmeticExpr
  | ArithmetciExpr "leq" ArithmeticExpr
  | ArithmetciExpr "geq" ArithmeticExpr
  | "(" BooleanExpr ")"
```

Some of the non-terminals of this context-free grammar are reported directly in Haskell. That is, I defined an *abstract syntax tree* that also represents the intermediate representation of a program. This intermediate representation will be the result of the parser. Moreover, the presence of an intermediate representation permits us to apply optimizations *at prior* respect to the interpretation step. The following code snippet contains the definition of the *abstarct syntax tree*.

```haskell
data ArithmeticExpr
  = Add ArithmeticExpr ArithmeticExpr
  | Sub ArithmeticExpr ArithmeticExpr
  | Mul ArithmeticExpr ArithmeticExpr
  | Div ArithmeticExpr ArithmeticExpr
  | Mod ArithmeticExpr ArithmeticExpr
  | Neg ArithmeticExpr
  | Constant Int
  | Identifier String
  deriving (Eq, Show)

data BooleanExpr
  = Boolean Bool
  | Not BooleanExpr
  | Or BooleanExpr BooleanExpr
  | And BooleanExpr BooleanExpr
  | Equal ArithmeticExpr ArithmeticExpr
  | NotEqual ArithmeticExpr ArithmeticExpr
  | Less ArithmeticExpr ArithmeticExpr
  | Greater ArithmeticExpr ArithmeticExpr
  | LessEqual ArithmeticExpr ArithmeticExpr
  | GreaterEqual ArithmeticExpr ArithmeticExpr
  deriving (Eq, Show)

data Command
  = Skip
  | Assignment String ArithmeticExpr
  | Branch BooleanExpr Block Block
  | Loop BooleanExpr Block
  deriving (Eq, Show)

type Block = [Command]
```

The Parser

The parser can be viewed as a function from a string to a list of pairs of values and strings (Graham Hutton).

```haskell
newtype Parser a = Parser {unwrap :: String -> [(a, String)]}
```

Note that the parser have a special function called `unwrap` that takes the function out from the parser. The parser is implemented in a "monadic" way. That is, I implemented the following interfaces: *monad plus* and *monad alternative* as suggested in *Monadic Parsing in Haskell* (Hutton & Meijer). In order to implement the monad interface I also implemented the *functor* and *applicative* interfaces as in the following code snippet.

```haskell
instance Functor Parser where
fmap f p = Parser (\cs ->
  [(f a, cs') | (a, cs') <- unwrap p cs])

instance Applicative Parser where
pure a = Parser (\cs -> [(a, cs)])
p <*> q = Parser (\cs -> concat
  [unwrap (fmap a q) cs' | (a, cs') <- unwrap p cs])
```

The functor implementation for the parser implements the fmap function, that is the application of a function on a wrapped parser. Moreover, the applicative implementation for the parser introduces both the function pure and the operator <*>. The pure function takes a simple value and wraps it into a parser. The <*> operator takes in input a function wrapped in a parser and another parser. The result is the application of the wrapped function onto the parser.

The next step is to implement the standard monad interface. Moreover, two more interfaces are implemented: MonadPlus and MonadAlternative. The plus operator defined on parsers concatenates the result of each one. Moreover, the MonadAlternative give us an operator <|> useful for combining parsers in a mutually exclusive way. The classes definition can be found in the following code snippet.

```
class (Monad m) => MonadPlus m where
zero :: m a
plus :: m a -> m a -> m a

class (MonadPlus m) => MonadAlternative m where
(<|>) :: m a -> m a -> m a
many :: m a -> m [a]
many m = some m <|> return []
some :: m a -> m [a]
some m = liftA2 (:) m (many m)
```

Note that the function liftA2 is the composition of fmap and the applicative operator <*>. The functions many and some are called *combinators* and are used to define the concept of repeated parsing. The many function refers to zero to any number of applications of a parser while the some function refers to at least one to any number of applications of a parser. The implementation for the parser of the bind operator and the other functions described before is the following.

```
instance Monad Parser where
return a = pure a
p >>= f = Parser (\cs -> concat
  [unwrap (f a) cs' | (a, cs') <- unwrap p cs])

instance MonadPlus Parser where
zero = Parser (const [])
p `plus` q = Parser (\cs -> unwrap p cs ++ unwrap q cs)

instance MonadAlternative Parser where
(<|>) p q = Parser (\cs ->
  case unwrap (p `plus` q) cs of
    [] -> []
    (x : _) -> [x])
```

The zero function defines what is an empty parser. Note that in this context an empty parser is a *failed* parser, that is a parser that results from a syntactical error. The plus function concatenates the results of two individual parsers. Using the monadic definition of parser, it permits us to easily build a parser for each structure of the programming language, without having to handle the eventual parsing errors individually. For example, it's possible to define a parser for symbols, identifiers and integers as in the following code snippet. Moreover, since the parser implements the MonadAlternative interface described above, the combination of multiple parsers is straightforward and easily **parallelizable** by the compiler.

```haskell
item :: Parser Char
item = Parser (\case "" -> []; (c : cs) -> [(c, cs)])

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do c <- item; if p c then return c else zero

token :: Parser a -> Parser a
token p = do space; v <- p; space; return v

space :: Parser String
space = many $ satisfy isSpace

identifier :: Parser String
identifier = token $ some $ satisfy isLetter

constant :: Parser Int
constant = read <$> token (some $ satisfy isDigit)

char :: Char -> Parser Char
char c = satisfy (c ==)
```

That is, I firstly defined a `item` function that reads a character from the string. Then I defined a function called `satisfy` that apply a constraint to the character read by the parser. This function returns an `zero` parser if the constraint is not satisfied. In the end, using both the combinators `many` and `some`, I built parsers for identifiers, constants and also spaces that are the main blocks of the context-free grammar already defined. In the same way, it's possible to define a parser for other constructs, such as keywords and symbols.

```haskell
keyword :: String -> Parser String
keyword cs = token $ word cs

word :: String -> Parser String
word [c] = do char c; return [c]
word (c : cs) = do char c; word cs; return (c : cs)

symbol :: Char -> Parser Char
symbol c = token $ char c
```

In order to build parsers for both arithmetic and boolean expression the `<|>` operator between parsers is used. So, I defined the parser for arithmetic expressions using other sub-parsers that are used in order to maintain the operators precedence.

```haskell
arithmeticExpr :: Parser ArithmeticExpr
arithmeticExpr = do
  a <- arithmeticTerm
    do symbol '+'; Add a <$> arithmeticExpr
    <|> do symbol '-'; Sub a <$> arithmeticExpr
    <|> do return a

arithmeticTerm :: Parser ArithmeticExpr
arithmeticTerm = do
  a <- arithmeticFactor
    do symbol '*'; Mul a <$> arithmeticTerm
    <|> do symbol '/'; Div a <$> arithmeticTerm
    <|> do symbol '%'; Mod a <$> arithmeticTerm
    <|> do return a
```

```haskell

arithmeticFactor :: Parser ArithmeticExpr
arithmeticFactor =
  do Constant <$> constant
  <|> do Identifier <$> identifier
  <|> do symbol '-'; Neg <$> arithmeticExpr
  <|> do symbol '('; a <- arithmeticExpr; symbol ')'; return a
```

The parser for boolean expressions is similar to the parser for arithmetic expressions and so it's omitted in this documentation. In the end we have a parser for every command described in the grammar. So, I defined a parser for the following commands: `skip`, `assignment`, `if-then-else` and `while-do`.

```haskell
assignment :: Parser Command
assignment = do
  d <- identifier
  symbol '='
  a <- arithmeticExpr
  symbol ';'
  return (Assignment d a)

branch :: Parser Command
branch = do
  keyword "if"
  symbol '('
  b <- booleanExpr
  symbol ')'
  keyword "then"
  c1 <- block
  do
    keyword "else"
    c2 <- block
    keyword "end if"
    symbol ';'
    return (Branch b c1 c2)
    <|> do
      keyword "end if"
      symbol ';'
      return (Branch b c1 [Skip])

loop :: Parser Command
loop = do
  keyword "while"
  symbol '('
  b <- booleanExpr
  symbol ')'
  keyword "do"
  c <- block
  keyword "end while"
  symbol ';'
  return (Loop b c)

skip :: Parser Command
skip = do
  keyword "skip"
  symbol ';'
  return Skip
```

It's important to notice that the grammar also allows for `if-then` statements (i.e. without the `else` command block). A parser that works on both `if-then-else` and `if-then` statements is implemented by combining the two individual parsers using the `<|>` operator. Moreover, using the `many` combinator and the `<|>` operator, the parsers for both multiple commands and a single command are straightforward.

```
block :: Parser Block
block = many command

command :: Parser Command
command = assignment <|> branch <|> loop <|> skip
```

The parser for the entire program is defined as in the following code snippet.

```
program :: Parser Block
program = do keyword "shrimp"; block

parse :: String -> Result (Block, String)
parse cs = case unwrap program cs of
  [] -> Error EmptyProgram
  [(b, cs)] -> Ok (b, cs)
```

The Optimizer

The *optimization* process is an intermediate step between the parsing and the interpretation of the program itself. Currently, the main optimization step is related to the execution of constant values. That is, if an expression in a loop is defined only on constant values, it's better to optimize the computation of that expression by replacing it with the result. This procedure is done before the interpretation of the program. For example, consider the following arithmetic expression, expressed in intermediate representation, that we wish to optimize.

```
let expr = Div (
    (Mul (Identifier "x") (Sub (Constant 5) (Constant 3)))
    (Add (Constant 9) (Constant 1)))
```

If we apply the optimization step to this arithmetic expression, we obtain the equivalent but more efficient arithmetic expression.

```
let expr' = Div (Mul (Identifier "x") (Constant 2)) (Constant 10)
```

The implementation of the *optimization* process in Haskell is straightforward, due to simple recursion functions. A very similar optimization process is also implemented on boolean expressions. The *optimization* process also includes a basic optimization on commands such as `skip`, `if-then-else` and `while-do`. First of all, all the `skip` commands are removed from the intermediate representation. Moreover, if the condition of a `if-then-else` command is always *true* then the entire statement is replaced with the first block of commands. In a similar way, if the condition is always *false* then the entire statement is replaced with the second block of commands. Furthermore, this approach is also used for `while-do` commands. That is, if the condition of a `while-do` command is always *false* then the entire statement is completely removed. However, if the condition is always *true* then an exception named *Infinite Loop* is raised. In other words, the optimizer is capable of detecting trivial infinite loops and prevents the interpretation of such programs.

The Interpreter

Finally, the obtained intermediate representation is interpreted by the interpreter. The interpreter scan the intermediate representation using *depth-first traversal* on the n-ary tree that compose the intermediate representation itself. Before introducing the execution of the interpreter on an intermediate representation, some useful data types are defined in order to handle errors.

```
data Exception
  = EmptyProgram
  | InfiniteLoop
  | DivisionByZero
  | UndeclaredVariable String
  deriving (Show)

data Result a = Ok a | Error Exception

instance Functor Result where
  fmap f (Ok v) = Ok (f v)
  fmap _ (Error e) = Error e

instance Applicative Result where
  pure v = Ok v
  (<*>) (Ok f) (Ok v) = Ok (f v)
  (<*>) (Error e) _ = Error e
  (<*>) _ (Error e) = Error e

instance Monad Result where
  (>>=) (Ok v) f = f v
  (>>=) (Error e) _ = Error e

exception :: Exception -> a
exception e = errorWithoutStackTrace $ show e
```

As one can see, the `Result` data type is a polymorphic type that can be either a `Ok` or a `Error`. The `Error` type also encapsulate an exception, one of the listed above. Moreover functor, applicative and monad interfaces are implemented in order to apply the needed operators directly on intermediate results. The main function that executes a block (i.e. a list of command) is reported in the following code snippet.

```
execute :: State -> Block -> State
execute s [] = s
execute s (Skip : cs) = execute s cs
execute s ((Assignment d a) : cs) =
  case evalArithmetic s a of
    Ok v -> execute s' cs
      where
        s' = insert d v s
    Error e -> exception e
execute s ((Branch b cs' cs'') : cs) =
  case evalBoolean s b of
    Ok True -> execute s (cs' ++ cs)
    Ok False -> execute s (cs'' ++ cs)
    Error e -> exception e
execute s (c@(Loop b cs') : cs) =
  execute s (Branch b (cs' ++ [c]) [Skip] : cs)
```

The implementation of the execution of a `while-do` resembles how it's evaluated in operational semantics, i.e. by "wrapping" it in a `if-then-else` command statement.

The implementation of the evaluation of arithmetic expressions make *heavily* use of the methods exposed by the applicative interface. Moreover, the `liftA2` function is used in order to make the evaluation functions implementation more compact.

```
evalArithmetic :: State -> ArithmeticExpr -> Result Int
evalArithmetic _ (Constant v) = Ok v
evalArithmetic s (Identifier d) =
  case search d s of
    Just v -> Ok v
    Nothing -> Error (UndeclaredVariable d)
evalArithmetic s (Add a1 a2) = liftA2 (+) v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Sub a1 a2) = liftA2 (-) v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Mul a1 a2) = liftA2 (*) v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Div a1 a2) = seqM2 safeDiv v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Mod a1 a2) = seqM2 safeMod v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Neg a) = negate <$> v
  where
    v = evalArithmetic s a
```

In this implementation, I also used the functions `safeDiv`, `safeMod` and `seqM2` which implementation is showed in the following code snippet. In other words, the `seqM2` is a custom function that implements binary sequencing, similarly to the `bind` operator.

```
safeDiv :: Int -> Int -> Result Int
safeDiv _ 0 = Error DivisionByZero
safeDiv u v = Ok (div u v)

safeMod :: Int -> Int -> Result Int
safeMod _ 0 = Error DivisionByZero
safeMod u v = Ok (mod u v)

seqM2 :: (Monad m) => (a -> b -> m c) -> m a -> m b -> m c
seqM2 f x y = join $ liftA2 f x y

join :: (Monad m) => m (m a) -> m a
join m = m >>= id
```

The implementation of the evaluation of boolean expressions is very similar to the one for arithmetic expressions, so it's omitted.

Conclusion

The monadic implementation of the parser give us a very simple way of concatenating and combining multiple parsers of sub-grammars. Also, it's very easy to extend the grammar of the language in order to include other commands or statements. Moreover, the use of an intermediate representation permits us to apply post-processing and optimizations and make the interpretation itself straightforward and more efficient.

Future works may include the introduction of other types for variables, such as boolean types. Another extension of this work consists of adding useful information about the parsing errors, i.e. missing tokens and relevant row and column locations in the source code where the error occurred. However, other improvements can be done in the optimization step. That is, one can implement more "aggressive" optimizations of the intermediate representation based on the expected state of memory of the program during its execution.

Running an example

First of all, navigate to the project directory. Then, open `ghci` and load the needed modules as following.

```
:load app/Main.hs src/Shrimp/Exception.hs src/Shrimp/Grammar.hs
      src/Shrimp/Optimizer.hs src/Shrimp/Utils.hs src/Shrimp/State.hs
      src/Shrimp/Interpreter.hs src/Shrimp/Parser.hs
```

Finally, run the main program and insert the path of a source file in the `examples` directory (for example `factorial.shr`).

```
*Main> main
The Shrimp Interpreter
Insert the path of the source file:
examples/factorial.shr
Memory state:
    i: 6
    n: 5
    x: 120
```