Глава 5. Системы исполнения функциональных программ

5.3. Функциональные модели последовательных процессов

Рассмотрим простой язык императивного программирования без функций, переходов и сложных структур данных.

Выражения:

- Целые числа: `12  25  0`
- Простые переменные целого типа: `x  myInt`
- Унарные и бинарные операции с целыми результатами: `(x+12) * (y-1)`
- Логические константы и выражения (но не переменные): `not (x < 5)`

Операторы:

- Пустой оператор: `skip`
- Присваивание: `x := x+1`
- Последовательное исполнение: `begin s1; s2; ... end`
- Условный оператор: `if b then s1 else s2`
- Оператор цикла: `while b do s`

Программа начинает работу в состоянии, заданном совокупностью значений переменных (например, заданных оператором ввода данных), а результат программы – конечное состояние (например, выведенное в конце работы оператором вывода).

Пример программы.

Рассмотрим программу вычисления факториала.

```
begin f := 1;
      while n > 1 do begin
         f := f * n;
         n := n – 1
      end
end
```

В начальном состоянии существенно только значение переменной n; в конце работы результат определяется значением переменной f.

Прежде всего, переведем программу в вид, удобный для обработки.

```
data Expression = { представление выражений }
data Operator   = { представление операторов }
data Context    = { представление контекста переменных
}
```

Два способа обработки и исполнения программы:

- Интерпретация: `interpret :: Operator -> Context -> Context`
- Компиляция: `compile :: Operator -> (Context -> Context)`

На самом деле оба способа представлены одной и той же карринговой функцией!

Представление выражений и программ.

```
data Expression = Integral Int  |  Logical Bool
                | Variable String
                | Unary String Expression
                | Binary Expression String
                  Expression
data Operator   = Skip
                | Assignment String Expression
                | Sequence [Operator]
                | If Expression Operator
                  Operator
                | While Expression Operator
type Context    = [(String, Expression)]

interpret :: Operator -> Context -> Context
eval      :: Expression -> Context ->
             Expression
intrinsic :: String -> [Expression] ->
             Expression
eval v@(Integral n) _       =  v
eval v@(Logical b) _        =  v
eval (Variable x) ctx       =  assoc x ctx
eval (Unary op ex) ctx      =  intrinsic op [eval ex ctx]
eval (Binary e1 op e2) ctx  =  intrinsic op [eval e1 ctx, eval e2
ctx]
intrinsic "+"   [(Integral a), (Integral b)]  =  Integral
(a+b)
intrinsic "-"   [(Integral a)]                =  Integral
(-a)
intrinsic "and" [(Logical a), (Logical b)]    =  Logical (a &&
b)
```

Исполнение операторов.

```
replace :: String -> Expression -> Context ->
Context
replace x val  =  map (\(y,v) -> (y,if x == y then val else
v))
```

Например:

```
replace "f" (Integral 20) [("n", (Integral 4)), ("f", (Integral
5))]
```

Даст в результате `[("n", (Integral 4)), ("f", (Integral 20))]`

```
interpret  ::  Operator -> Context -> Context

interpret Skip ctx                 = ctx
interpret (Assignment x expr) ctx  = replace x (eval expr ctx) ctx
interpret (Sequence []) ctx        = ctx
interpret (Sequence (s:seq)) ctx   = interpret (Sequence seq) (interpret s
ctx)
interpret (If expr s1 s2) ctx      = case (eval expr ctx) of
                                        (Logical True)  -> interpret s1 ctx
                                        (Logical False) -> interpret s2 ctx
interpret oper@(While expr s) ctx  = case (eval expr ctx) of
                            (Logical True)  -> interpret oper (interpret s
ctx)
                            (Logical False) -> ctx
```

Пример компиляции и исполнения программы

```
begin f := 1;
      while n > 1 do begin
         f := f * n;
         n := n – 1
      end
end

program =
    Sequence
        [(Assignment "f" (Integral 1)),
         (While
             (Binary (Variable "n") ">" (Integral 1))
             (Sequence
                 [(Assignment "f" (Binary (Variable "f") "*" (Variable
"n"))),
                  (Assignment "n" (Binary (Variable "n") "-" (Integral
1)))]))
compile program                - функция преобразования контекстов
        ]

interpret program [("f", (Integral 0)), ("n", (Integral
3))]
                  [("f", (Integral 6)), ("n", (Integral 0))]
```