

```

1  module Eval3 (eval) where
2
3  import AST
4  import Control.Applicative (Applicative(..))
5  import Control.Monad      (liftM, ap)
6
7  -- Estados
8  type Env = [(Variable,Int)]
9
10 newtype StateErrorTick a = StateErrorTick { runStateErrorTick :: Env -> (Maybe (a,
11 Env),Int) }
12 --Agregamos un Int para contar la cantidad de operaciones. Si se produce un error
13 --contamos las operaciones hasta el error.
14
15 instance Monad StateErrorTick where
16     return x = StateErrorTick (\e -> (Just (x,e),0) )
17     StateErrorTick h >=> f = StateErrorTick (\e -> case h e of
18         (Nothing,n) -> (Nothing,n)
19         (Just (a,e'),n) -> let (ans ,n') = runStateErrorTick (f a) e'
20                             in (ans ,n' + n) )
21     -- h :: Env -> (Maybe (a, Env),Int)
22     -- f :: a -> StateErrorTick b
23
24 class Monad m => MonadTick m where
25     tick :: m ()
26
27 instance MonadTick StateErrorTick where
28     tick = StateErrorTick (\e -> (Just ( () , e ) , 1))
29
30
31 class Monad m => MonadState m where
32     lookfor :: Variable -> m Int
33     update  :: Variable -> Int -> m ()
34
35
36 instance MonadState StateErrorTick where
37     lookfor v = StateErrorTick (\s -> (Just (lookfor' v s, s),0) )
38         where lookfor' v ((u, j):ss) | v == u = j
39                                     | v /= u = lookfor' v ss
40     -- Suponemos que no se utilizan variables no declaradas en LIS
41     update v i = StateErrorTick (\s -> (Just ((), update' v i s),0))
42         where update' v i [] = [(v, i)]
43               update' v i ((u, _):ss) | v == u = (v, i):ss
44               update' v i ((u, j):ss) | v /= u = (u, j):(update' v i ss)
45
46 class Monad m => MonadError m where
47     throw :: m a
48
49 instance MonadError StateErrorTick where
50     throw = StateErrorTick (\e -> (Nothing,0))
51
52
53 -- Estado nulo
54 initState :: Env
55 initState = []
56
57 -- Evalua un programa en el estado nulo
58 eval :: Comm -> (Maybe Env,Int)
59 eval p = case (runStateErrorTick (evalComm p) initState) of
60     (Nothing,n) -> (Nothing,n)
61     (Just (_,e),n) -> (Just e,n)
62
63 -- Evalua un comando en un estado dado

```

```

64
65
66
67 evalComm :: (MonadState m, MonadError m, MonadTick m) => Comm -> m ()
68 evalComm Skip = return ()
69 evalComm (Let v n) = do n' <- evalIntExp n
70                       update v n'
71                       return ()
72
73 evalComm (Seq c1 c2) = do evalComm c1
74                           evalComm c2
75                           return ()
76
77 evalComm (Cond b ct cf) = do cond <- evalBoolExp b
78                             if cond then (do {evalComm ct ; return ()} )
79                             else (do {evalComm cf ; return ()} )
80
81 evalComm w@(While b c) = do cond <- evalBoolExp b
82                             if cond then evalComm (Seq c w)
83                             else return ()
84
85
86
87 evalIntExp :: (MonadState m, MonadError m, MonadTick m) => IntExp -> m Int
88 evalIntExp (Const x) = return x
89 evalIntExp (Var v) = lookfor v
90 evalIntExp (UMinus e) = do n <- evalIntExp e
91                           tick
92                           return (-n)
93 evalIntExp (Plus e1 e2) = do n1 <- evalIntExp e1
94                             n2 <- evalIntExp e2
95                             tick
96                             return (n1+n2)
97
98 evalIntExp (Minus e1 e2) = do n1 <- evalIntExp e1
99                             n2 <- evalIntExp e2
100                             tick
101                             return (n1-n2)
102
103 evalIntExp (Times e1 e2) = do n1 <- evalIntExp e1
104                             n2 <- evalIntExp e2
105                             tick
106                             return (n1*n2)
107
108 evalIntExp (Div e1 e2) = do n1 <- evalIntExp e1
109                             n2 <- evalIntExp e2
110                             if n2==0 then throw else do {tick ; return (div n1
111                             n2)}
112
113 -- Evalua una expresion entera, sin efectos laterales
114 evalBoolExp :: (MonadState m, MonadError m, MonadTick m) => BoolExp -> m Bool
115 evalBoolExp BTrue = return True
116 evalBoolExp BFalse = return False
117 evalBoolExp (Eq e1 e2) = do n1 <- evalIntExp e1
118                             n2 <- evalIntExp e2
119                             return (n1==n2)
120 evalBoolExp (Lt e1 e2) = do n1 <- evalIntExp e1
121                             n2 <- evalIntExp e2
122                             return (n1<n2)
123 evalBoolExp (Gt e1 e2) = do n1 <- evalIntExp e1
124                             n2 <- evalIntExp e2
125                             return (n1>n2)
126 evalBoolExp (And b1 b2) = do b1' <- evalBoolExp b1
127                             b2' <- evalBoolExp b2
128                             return (b1' && b2')
129 evalBoolExp (Or b1 b2) = do b1' <- evalBoolExp b1
130                             b2' <- evalBoolExp b2
131                             return (b1' || b2')

```

```
131 evalBoolExp (Not b)      = do b' <- evalBoolExp b
132                           return (not b')
133
```