C6

Transcrierea semanticii în do-notație

Despre: Semantică, do-notația din Haskell, transcrierea regulilor semantice în do-notație, condiții necesare ca do-notația să fie conformă cu programarea imperativă, legile monadei, definiția monadei

Descrierea semanticii unei simple adunări de termi într-un limbaj de expresii se poate face în mai multe moduri, depinzând de semantica utilizată.În acest capitol vom prezenta un instrument puternic pentru transcrierea semanticilor în limbajul funcțional Haskell: do-notația.
Exemplu de regulă semantică din lucrarea [Cio-96] (Cap. 3.2.3. pg 42) într-o notație ușor modificată:

```
 E |- t1 => a          E |- t2 => b
-------------------------------------------------
      E |- t1 + t2 => a add b
```

Paragraful din care am extras-o prezintă semantica dinamică bazată pe “environment” (trad. - mediul de calcul).

În aceea și lucrare găsim în capitolul dedicat semanticilor denotaționale a expresiilor aritmetice (Cap. 4.11, pg.98) următoarea formulă (notațiile pot fi ușor diferite):

```
A[[e1 + e2]] (M) = A[[e1]](M)  add A[[e1]](M)
```

Limbajul Haskell permite tranimplementarea acestor semantici chiar și în cazul prezenței efectelor laterale prin ceea ce se numește **do-notație.**

Informal, do-notația este o scriere cu aspect imperativ practicată în limbajul funcțional pur care este Haskell. (Unul dintre locurile unde este folosită este la scrierea programelor sau porțiunilor de programe care utilizează instrucțiuni de I/O.)

În realitate do-notația este un “syntactic-sugar” pentru notații realizate cu operatorii >>= și return având tipurile:

```
return :: a -> M a
>>= :: M a -> (a -> Mb) -> Mb
```

Formal se definește recursiv, (pentru orice variabile x, actiuni m și expresii / acțiuni e) ca fiind:

```
do { e } = e
do { m ; e} = m>>= (\ _ -> do {e} )
do { x <- m; e } = m >>= (\x -> do {e})
```

iar în unele lucrări cum ar fi: [Lab-**] apare și o a patra regulă:

```
do { let exp; e} = let exp in do {e}
```

Aplicație: Reguli semantice cum este cea de mai sus se scriu în

Haskell, în do-notație (ascunzând detaliile despre 'environment' în felul cum sunt definiți operatorii >>= și return):

```
do {  a <- t1 ;
      b <- t2 ;
      add a b }
```

Această descriere este “syntactic-sugar” pentru expresia echivalentă:

```
t1 >>= (\a->t2 >>= (\b -> add a b))
```

Suportul acestor calcule este o structură algebrică (despre care vom vedea că este chiar monadă) alcătuită dintr-un constructor M de tipuri (cu parametru) și două funcții polimorfice cu tipurile de forma (cunoscută):

```
return :: a -> M a
>>= :: M a -> (a -> Mb) -> Mb
```

care satisfac o serie de egalități fără de care do-notația nu s-ar conforma intuiției programatorului care utilizează limbaje imperative.

Ecuațiile de îndeplinit (care ar fi fost evidente la programarea imperativă) nu sunt satisfăcute automat de orice operatori >>= și return:

```
do { x<- return t ;
     f x   }              =        f t
do { x <- m ;
      return x  }         =        m
do {  x <- m ;                     do { y <- do {x <- m;
      do { y <- fx ;      =                    f x } ;
           g y } }                       g y }
```

Aplicând definiția do-notației și transcriindu-le în expresii

formate cu >>= și return se obțin imediat formele echivalente:

```
return t >>= (\ x -> f x)       = f t
m >>= (\x -> return x)          = m
```

și ultima:

```
 m >>= (\x -> (f x)   >>= (\y -> g y)  )       =
( m >>= (\x -> (f x))) >>= (\y -> g y)
```

Teoremă: Aceste condiții necesare pentru ca do-notația să funcționeze similar programării imperative sunt îndeplinite dacă și numai dacă structura formată de (M, >>=, return) este o monadă - așa cum a definit-o și utilizat-o Wadler:

Demonstrația este imediată: comparând relațiile de mai sus cu legile monadei din lucrarea lui Wadler se constată că cele 3 proprietăți intuitive necesare ale do-notației sunt exact legile monadei exprimate în do-notație (cu observația că la prima lege apare o variabilă suplimentară care se poate înlătura din ambii mebri).

6.1. Operatori de “lifting”

Surse variate inclusiv [Esp-95] indică drept revoluționară ideea pornită de la lucrările lui E.Moggi [...] de a considera interpretorul nu ca funcție de la termi și environment la valori ci ca o funcție de la termi și environment la valori monadice.

Acest lucru impune, în realizarea interpretoarelor care se bazează pe această idee, proiectarea calculelor din universul valorilor în universul monadei. Lucrările de specialitate numesc acest fenomen “lifting” sau “lifting monadic”. În [Esp-95], acest lucru este realizat folosind o serie de funcții de nivel superior

scrise în Scheme, care transformă operațiile ce returnează *valori* în operații care returnează *valori monadice*. Lucrarea [Esp-95], prezintă acești operatori de lifting (eng - “lifting operators”) în Cap 1.4.1 fig 1.1

```
(define ((lift-p1-a0) unit bind op) p1)
   (unit (op p1)))

(define ((lift-p0-a1 unit bind op) d1)
   (bind d1
      (lambda (v1)
         (unit (op v1)))))

(define ((lift-p0-a2 unit bind op) d1 d2)
   (bind d1
      (lambda (v1)
         (bind d2
            (lambda (v2)
               (unit (op v1 v2)))))))

(define ((lift-p1-a1 unit bind op) p1 d1)
   (bind d1
      (lambda (v1)
         (unit (op p1 v1)))))

(define ((lift-if unit bind op) d1 d2 d3)
   (bind d1
      (lambda (v1)
         (op v1 d2 d3))))
```

Notați că acești operatori de lifting acționează în cadrul creat de o monadă (mai exact de operatorii acesteia “unit” și “bind”) făcând lifting monadic operatorului oarecare “*op”*. Variabilele di, i=1..3 sunt variabilele libere din expresia următoare.

Transcriind în Haskell expresiile funcțiilor de mai sus se obțin următoarele expresii cu “bind” și “return” utilizabile și utilizate la realizarea interpretoarelor:

```
return (op p1)
d1>>= (\ v1 -> return (op v1))
d1 >>= (\v1 -> d2 >>= (\ v2 -> return (op v1 v2)))
d1 >>= (\ v1 -> return (op p1 v1))
d1 >>= (\ v1 -> op v1 d2 d3)
```

... care la rândul lor pot fi transcrise imediat (folosind definiția din orice manual de Haskell) în do-notație:

```
do { return (op p1) }
do { v1 <- d1 ; return (op v1) }
do { v1 <- d1 ; v2 <- d2 ; return (op v1 v2) }
do { v1 <- d1 ; return (op p1 v1) }
do { v1 <- d1 ; op v1 d2 d3 }
```

6.2. Observații și concluzii

Observație: În Haskell *nu este necesar să definim acești operatori de lifting* ei putând fi înlocuiți cu scriere directă a

semanticii în do-notație. Acest lucru recomandă odată în plus limbajul Haskell (deosebit de alte limbaje funcționale , el fiind înzestrat cu această do-notație) la scrierea de interpretoare (și compilatoare) adaptabile.

Concluzia întâi: Monada se poate defini și ca structura algebrică (M, >>=, return) pentru care do-notația din Haskell funcț ionează conform celor trei reguli.

Concluzia a II-a: Dacă dorim să exprimăm semanticile în do-notație structura algebrică de monadă este un instrument util și necesar (Interpretoarele fară monade au adaptabilitatea limitată.)

Concluzia a III-a: Do-notația existentă în Haskell recomandă acest limbaj ca platformă pentru realizarea de interpretoare adaptabile.