C8

Un asamblor într-o coajă de nucă

Despre: Felul cum putem folosi împreună două instrumente puternice; noțiunea de functor din teoria categoriilor și limbajul Haskell 98 pentru a proiecta și implementa în scurt timp un assamblor adaptabil și universal, în sensul că prin modificarea morfismului care indică transformarea mnemonicilor în coduri rezultă imediat asamblorul altui limbaj.

Este greu de argumentat un răspuns la întrebarea ce este un asamblor. Definiția clasică afirmă că el este "compilatorul unui limbaj de asamblare". Deoarece șirul de numere produs de asamblor este format din coduri mașină, asamblorul este un compilator, în virtutea ideii că produc cod doar compilatoarele. Totuși unele părți ale unui asamblor sunt de natură interpretativă, de exemplu interpretorul care calculează valoarea expresiilor constante din limbajul de asamblare și plasează constantele rezultate în cod. (De regulă, această evaluare a expresiilor nu se face generând cod pentru ele și executându-l !). Dacă însă considerăm textul în limbaj de asamblare ca pe o gigantică expresie cu valori în *mulțimea listelor de coduri* atunci putem construi un evaluator pentru semantica acestor

expresii! Acesta va fi pe de o parte un interpretor de expresii iar pe de altă parte asamblorul limbajului, adică un program producător de cod, deci aparent un compilator.

Această afirmație nu este o noutate absolută, mai existând în literatura de specialitate lucrări în care compilarea (dar acolo este unui limbaj de nivel înalt) este văzută ca o evaluare (mai exact o evaluare parțială) a unei semantici bazate pe un functor. (Harrison, L. William; Kamin, N. Samuel; *Compilation as Partial Evaluation of Functor Category Semantics*, 1997). În paragrafele care urmează ne vom ocupa de construcția unui asamblor adaptabil, folosind tot un functor.

Adaptabilitatea se va obține prin *separarea informațiilor despre compilarea instrucțiunilor cu diverse mnemonici* de restul programului. Ele vor descrie doar un morfism. Un functor transformă acest morfism într-un asamblor. Morfismul se poate imediat redefini fără a mai face alte modificări în program și de aici rezultă adaptabilitatea. Câteva definiții sunt prezentate în continuare.

Fundamente

O definițe a noțiunii de categorie poate fi găsită în volumul semnat de Barr Michael și Wells Charles, *Toposes Triples and Theories*, (McGill University, 2002), la p.1-11:

O categorie C constă în două colecții, **Ob**(C) – a cărei elemente sunt numite **obiectele din C** și **Ar**(C) – **săgețile** sau **morfismele** categoriei C.

Fiecărei săgeți îi sunt asociate două obiecte numite respectiv **sursa** (**domeniul**) și **destinația** (**codomeniul**, în limba engleză termenul

target înseamnă țintă) săgeții. Morfismul cu sursa (originea) A *și destinația* B se notează cu $f : A \to B$. Dacă $f : A \to B$ și $g : B \to C$ sunt două morfisme atunci există în categoria C un al treilea morfism $g \circ f : A \to C$ numit compunerea lui g cu f. Această compunere nu mai este precizată prin nimic altceva. Ea se notează $g\,f$ în loc de $g \circ f$ atunci când nu există pericol de confuzie. Pentru fiecare obiect A din categoria C există un morfism asociat lui, notat id_A (uneori notat 1_A sau pur și simplu 1 în funcție de context), numit identitatea (sau morfismul identitate) al obiectului A. $\mathrm{id}_A : A \to A$ are asociată perechea de obiecte, sursă și destinație *(A,A)*.

Aceste obiecte și morfisme se conformează următoarelor axiome:

1. Pentru orice $f : A \to B$,

$$f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f$$

2. Pentru orice $f : A \to B$, $g : B \to C$, $h : C \to D$,

$$h \circ (g \circ f) = (h \circ g) \circ f$$

Deci o categorie constă în două colecții, una de obiecte și una de săgeți care verifică o serie de axiome.

În aceeași carte, la pagina 11 găsim definiția noțiunii de functor.

Ca și pentru alte feluri de obiecte matematice având o structuură și pentru categorii se definește o noțiune analoagă celei de morfism. Este natural să se definească un fel de morfism între categorii care să ducă obiectele în obiecte, săgețile în săgeți și să păstreze sursa, destinația, compoziția și identitățile (duce sursa în sursă, destinația în destinație, și fiecare morfism-identitate în alt morfism-identitate.) El se numește functor.

Dacă C și D sunt două categorii, un functor F : C ->D , este o aplicație care verifică:

1. Dacă $f : A \rightarrow B$ este un morfism din categoria C , atunci

$Ff : FA \rightarrow FB$ este un morfism în categoria D;

2. $F(\mathrm{id}_A) = \mathrm{id}_{FA}$ și

3. Dacă $g : B \rightarrow C$, atunci $F(g \circ f) = Fg \circ Ff$.

Un exemplu

Exemplul următor, care apare în multe cărți de specialitate, este reprodus după lucrarea semnată de Andrea Schalk, intitulată *Some notes on monads.*

Exemplul 1: Fie M : $Set \rightarrow Set$ un functor care duce fiecare mulțime (alfabet) A în mulțimea cuvintelor (A*) care se pot forma peste alfabetul A și a cărui acțiune asupra morfismelor este că pentru fiecare f : A -> B din Set avem Mf : MA -> MB dat de:

$$Mf(a_1 \bullet .. \bullet a_n) = f(a_1) \bullet .. \bullet f(a_n)$$

Functorul servește în continuare la construirea asamblorului adaptabil. Programul va fi scris tot în Haskell 98, versiunea revizuită din 2003.

Sursa unui asamblor, scris în 8 linii de cod

Ca prim exemplu prezentăm un mic asamblor pentru un limbaj de asamblare minuscul format din două mnemonici, NOP și RET cu codurile 0 și respectiv 201.

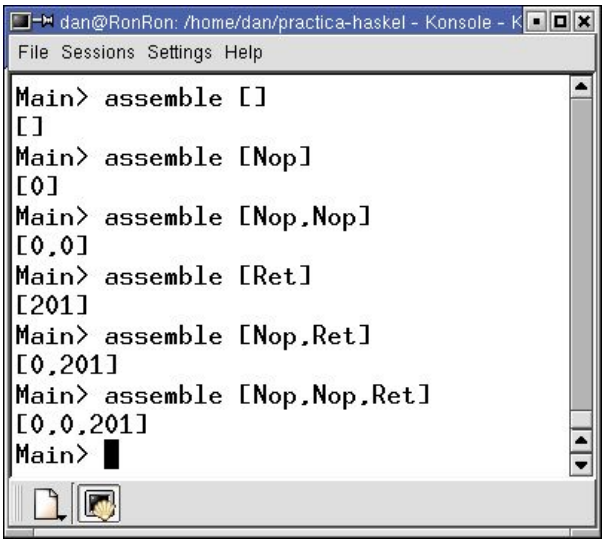
```
-- Un mic asamblor adaptabil scris in Haskell 98,
-- Dan Popa, 2005
```

```
data Instr = Nop | Ret
-- Arrow
f       :: Instr -> [Int]
f  Nop      = [ 0 ]
f  Ret      = [ 201 ]
f  _        = []
-- Functor
m          :: (Instr -> [Int]) -> [Instr] -> [Int]
m f []     = []
m f (a1:l) = f a1 ++ m f l
-- Assembler
assemble       :: [Instr] -> [Int]
assemble  x = m f x
```

Observație: Fiecare element din tipul **Instr poate** produce, teoretic, o listă de oricâte coduri mașină.

Implementarea și testarea

Interpretorul Hugs 98, al limbajului Haskell 98, a fost fost folosit pentru a rula programul de mai sus. (GHC 6.2, sau 6.4 – Glasgow Haskell Compiler este de asemenea de luat în considerare.) Sistemul de operare a fost un Mandrake Linux 8.2 (actualmente Mandriva), una din puținele distribuții care, în varianta de download, cuprinde și interpretorul Haskell 98 pe nume Hugs. Rezultatele testării asamblorului pot fi văzute în imaginea următoare:



Adaptarea asamblorului

Scopul următor de îndeplinit este adaptarea asamblorului precedent la un alt limbaj, cu instrucțiuni mai complexe. Limbajul ales ca exemplu este cel descris în capitolul 4 al volumului profesorului P.D. Terry, *Compilers and Compiler Generators.* Pentru construirea asamblorului este suficient să precizăm săgeata care duce mnemonicele în liste de coduri și să-i aplicăm același functor pentru a obține asamblorul. Noul program este:

```
-- Dan Popa  22.iunie.2005
-- Instructions -- Instructiunile de asamblat
data Instr   =NOP | CLA | CLC | CLX | CMC | INC | DEC |
              INX | DEX | TAX | INI | INH | INB | INA |
              OTI | OTC | OTH | OTB | OTA | PSH | POP |
              SHL | SHR | RET | HLT |
              LDA Int  | LDX Int  | LDI Int | LSP Int | LSI Int |
              STA Int  | STX Int  | ADD Int | ADX Int | ADI Int |
              ADC Int  | ACX Int  | ACI Int | SUB Int | SBX Int |
              SBI Int  | SBC Int  | SCX Int | SCI Int | CMP Int |
              CPX Int  | CPI Int  | ANA Int | ANX Int | ANI Int |
              ORA Int  | ORX Int  | ORI Int | BRN Int | BZE Int |
              BNZ Int  | BPZ Int  | BNG Int | BCC Int | BCS Int |
              JSR Int

-- Arrow f -- Morfismul de asamblare a fiecarei instructiuni
-- The semantics -- el ne da semantica de generare a codului
f     ::  Instr -> [Int]
f   NOP      = [ 00]
f   CLA      = [ 01]
f   CLC      = [ 02]
f   CLX      = [ 03]
f   CMC      = [ 04]
f   INC      = [ 05]
f   DEC      = [ 06]
f   INX      = [ 07]
f   DEX      = [ 08]
```

```
f   TAX      = [ 09]
f   INI      = [ 10]
f   INH      = [ 11]
f   INB      = [ 12]
f   INA      = [ 13]
f   OTI      = [ 14]
f   OTC      = [ 15]
f   OTH      = [ 16]
f   OTB      = [ 17]
f   OTA      = [ 18]
f   PSH      = [ 19]
f   POP      = [ 20]
f   SHL      = [ 21]
f   SHR      = [ 22]
f   RET      = [ 23]
f   HLT      = [ 24]
```

-- Double byte instr.- Instructiuni pe 2 bytes

```
f   ( LDA b )      = [ 25 , b ]
f   ( LDX b )      = [ 26 , b ]
f   ( LDI b )      = [ 27 , b ]
f   ( LSP b )      = [ 28 , b ]
f   ( LSI b )      = [ 29 , b ]
f   ( STA b )      = [ 30 , b ]
f   ( STX b )      = [ 31 , b ]
f   ( ADD b )      = [ 32 , b ]
f   ( ADX b )      = [ 33 , b ]
f   ( ADI b )      = [ 34 , b ]
f   ( ADC b )      = [ 35 , b ]
f   ( ACX b )      = [ 36 , b ]
f   ( ACI b )      = [ 37 , b ]
f   ( SUB b )      = [ 38 , b ]
f   ( SBX b )      = [ 39 , b ]
f   ( SBI b )      = [ 40 , b ]
f   ( SBC b )      = [ 41 , b ]
f   ( SCX b )      = [ 42 , b ]
f   ( SCI b )      = [ 43 , b ]
f   ( CMP b )      = [ 44 , b ]
f   ( CPX b )      = [ 45 , b ]
```

```
f   ( CPI b )       = [ 46 , b ]
f   ( ANA b )       = [ 47 , b ]
f   ( ANX b )       = [ 48 , b ]
f   ( ANI b )       = [ 49 , b ]
f   ( ORA b )       = [ 50 , b ]
f   ( ORX b )       = [ 51 , b ]
f   ( ORI b )       = [ 52 , b ]
f   ( BRN b )       = [ 53 , b ]
f   ( BZE b )       = [ 54 , b ]
f   ( BNZ b )       = [ 55 , b ]
f   ( BPZ b )       = [ 56 , b ]
f   ( BNG b )       = [ 57 , b ]
f   ( BCC b )       = [ 58 , b ]
f   ( BCS b )       = [ 59 , b ]
f   ( JSR b )       = [ 60 , b ]
f   _               = []

-- Functorul M -- functor M -- l inseamna lista elementelor
a2...an
m          :: (Instr -> [Int]) -> [Instr] -> [Int]
m f []     = []
m f (a1:l) = f a1 ++ m f l

-- Asamblorul final rezultat din m si f
-- The final assembler
assemble       :: [Instr] -> [Int]
assemble  x = m f x
```

Remarcă: așa cum se observă citind sursa programului, tot ce am avut de făcut a fost să scriem noile mnemonici și noile liste de coduri mașină. Functorul rămâne neschimbat. Iar el transformă și de această dată săgeata codificării mnemonicilor în asamblor.

Testarea noului asamblor

Am folosit exemplul 4.3 din aceeași carte deoarece în același capitol

există și sursa și codul produs prin asamblare din el.

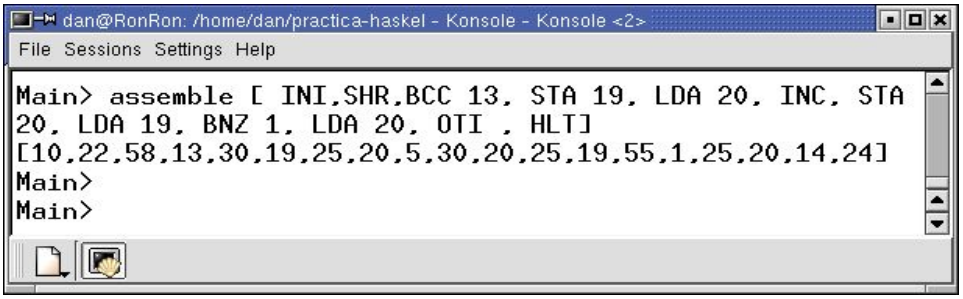
Sursa:

```
INI
SHR
BCC 13
STA 19
LDA 20
INC
STA 20
LDA 19
BNZ 1
LDA 20
OTI
HLT
```

Conform autorului, prof. Terry, trebuie să rezulte prin asamblare secvența de coduri:

10 22 58 13 30 19 25 20 5 30 20 25 19 55 1 25 20 14 24

Iar traducerea reușește,așa cum se vede din imagine:



Avantaj: Nu este deci nevoie să schimbăm întregul asamblor ci doar mnemonicele. Timpul de adaptare a unui asemenea asamblor este extrem de scurt. Cele două exemple de mai sus s-au încadrat împreună într-un timp de realizare mai mic de o oră.