O mica introducere in Haskell 98

Paul Hudak
Universitatea Yale
Departamentul de Informatică

John Peterson
Universitatea Yale
Departamentul de Informatică

Joseph H. Fasel
Universitatea din California
Laboratorul Naţional Los Alamos

Octombrie 1999

1 Introducere

Scopul nostru cand am scris acest tutorial nu era de a învăța programare, nici măcar de a preda programare functionala. Mai degrabă, el este destinat pentru a servi ca un fel de supliment al Raportului Haskell (The Haskell Report) [4], care este altfel o expunere tehnică destul de densă. Scopul nostru este de a oferi o introducere uşoară in limbajul Haskell pentru cineva care are experiență cu cel puțin un altul, de preferință un limbaj funcțional (Chiar dacă e doar un limbaj „aproape-funcțional ", cu elemente imperative , cum ar fi ML, LISP sau Scheme).

Dacă cititorul doreste să învețe mai multe despre stilul de programare funcțional, vă recomandăm Introducere cartii lui Bird's despre Programare funcțională [1] sau cartea lui **Davis O Introducere în sisteme de programare funcționale - Utilizarea Haskell** [2]. Pentru un studiu util al limbajelo de programare funcționale şi tehnice, incluzand aici unele dintre principiile de proiectare ale limbajelor folosite în Haskell, a se vedea [3].

Limbajul Haskell a evoluat semnificativ de la naşterea sa în 1987. Acest manual se ocupă de standardul Haskell 98. Versiunile mai vechi ale limbajului sunt în prezent nesemnificative;

Utilizatorii Haskell sunt încurajați să folosească Haskell 98. Există, de asemenea, multe extensii pentru Haskell 98 care au fost distribuite pe scară largă, fiind incluse in implementari ca Hugs si GHC.

Acestea nu sunt încă parte formalizata a limbajului Haskell şi nu sunt cuprinse în acest tutorial.Strategia noastră generală pentru introducerea elementelor de limbaj este aceasta: a motiva ideea, a da unele puncte de vedere, a da câteva exemple, iar apoi va trimitem la punctul din Raportul Haskell pentru detalii. Vă sugerăm, însă, sa cititi detalile abia dupa ce ati parcurs aceasta introducere in intregime.

2.2 VALORI, tipuri, precum şi alte bunătăți

Pe de altă parte, biblioteca Standard Prelude (în apendicele A al Raportului şi bibliotecile standard - găsite în Raport, vezi Biblioteca [5]) conține o mulțime de exemple utile de cod Haskell. Si va încurajam la o lectură atentă a lor indată ce terminati acest tutorial. Acest lucru nu numai ca va va oferi o imagine a felului cum decurge programarea in Haskell si cuma arata programele reale, dar, de asemenea, va vor familiariza cu setul de funcții şi tipuri standard din Haskell .

În cele din urmă, site-ul web Haskell, http://haskell.org, are o mulțime de informații despre limbajul Haskell şi implementările sale.

[Am luat, de asemenea, o serie de reguli de sintaxă si lexicale pentru a le introduce treptat ca exemple, şi sunt încadrate între paranteze, ca acest alineat. Acest lucru este în contrast evident cu organizare a Raportului, deşi raportul rămâne sursa de autoritate maxima pentru detalii (referințe, cum ar fi $ x2.1 "se referă la secțiunile din Raport).]

Haskell este un limbaj de programare tipizat puternic:

1. Tipurile sunt omniprezente, iar ca nou-venit este cel mai bine sa deveniti conştienți de la început de puterea fantastica şi de complexitatea sistemului de tipuri din Haskell (un sistem de tipuri polimorfice Hindley - Milner). Pentru cei a căror experiență este formata cu limbaje slab tipizate , cum ar fi Perl, Tcl, sau Scheme acest nou lucru poate fi o schimbare de optica dificila. Pentru cei familiarizați cu Java, C, Modula, sau chiar ML, adaptarea ar trebui să fie mai uşoara, dar nu imediata deoarece sistemul de tipuri din Haskell este diferit şi este mai bogat decât tot ce ati intalnit pina acum. În orice caz, programarea tipizata face parte din experiența programării in Haskell şi nu poate fi evitată.

Deoarece Haskell este un limbaj pur funcțional, toate calculele sunt realizate prin intermediul actului de evaluare a unor *expresii* (termeni sintactici) pentru a obține *valori* (entități abstracte pe care le considerăm ca fiind răspunsuri). Fiecare valoare are un tip asociat. (Intuitiv, ne putem gândi la tipuri ca la niste seturi de valori.) Exemplele de expresii include valorile atomice, cum ar fi întregul 5, caracterul 'a', şi funcția (\ X -> X + 1) , precum şi valori structurate, cum ar fi lista [1,2,3] şi perechea ('b',4).

Aşa cum expresiile denota valori, expresile de tip sunt termeni sintactici care denota valorile de tip (numite doar „tipuri"). Exemple de expresii de tip includ următoarele tipuri atomice *Integer* (întregi de orice marime), *Char* (caractere), *Integer-> Integer* (funcțiile de la Întregi la Întregi), precum şi tipuri structurate, de exemplu: *[Integer]* (liste de numere întregi omogene) şi *(Char, Integer)* (perechi formate de un caracter cu un întreg).

Toate valorile Haskell sunt de prima classa - ele pot fi transmise ca argumente pentru funcții, returnate ca rezultate, plasate în structurile de date, etc. Tipurile din Haskell, pe de altă parte, nu sunt de prima clasa. Tipurile, într-un anume sens, descriu valori, şi asociația dintre o valoare şi tipul său se numeşte *specificatie de tip* sau *semnatura*.

Utilizand exemplele de valori şi de tipuri de mai sus, vom scrie, după cum urmează, valorile şi tipurile lor:

```
5::Integer
'A'::Char
inc::Integer - > Integer
[1,2,3]::[Integer]
(4,'b'):: (Integer, Char)
```

Semnul "::" poate fi citit ca "de tipul" sau "are tipul".Functiile Haskell sunt in mod normal definite de o serie de ecuatii. De exemplu, functia "inc" poate fi definita de o singura ecuatie:

```
inc n =  n+1
```

O ecuatie este un exemplu de declaratie. Un alt fel de declaratie este cea de tip, cu care putem precisa o functie anume inc, de exemplu de la Intregi la Intregi:

```
inc :: Integer -> Integer
```

Vom dezbate functiile definite, pe larg, in capitolul 3.

In scop pedagogic, cand vrem sa exprimam “e1 redus la alta expresie sau valoare e2” vom scrie:

e1 => e2.

De exemplu:

inc(inc 3) => 5

Sistemul Haskell defineste precis relatia dintre tip si valoare(vezi Raport $ 4.1.3)

Sistemul Haskell asigura un tip static, sigur, iar programatorul nu incurca tipul in nici un fel. De exemplu nu putem aduna doua caractere ca in expresia : ‘a’+’b’, asa cum este scrisa. Principalul avantaj al tipizarii statice este bine de stiut de pe acum: Toate tipurile de erori sunt detectate la timp, la compilare. Nu toate erorile introduse pot fi recunoscute de sistem, cum ar fi expresia 1/0 care se poate introduce, dar in urma evaluarii rezultatul va fi o eroare de executie. Totusi sistemul descopera multe erori de program […] si de asemeni permite generarea unui cod mult mai eficient (de exemplu nu toate sarcinile sau testele sunt necesare).

Acest sistem de tipuri asigura corectitudinea datelor utilizatorului. De fapt sistemul de tipuri din Haskell este destul de puternic pentru a permite scrierea oricarui fel de comanda. Scrierea tipului functiei inc este o idee buna, atata timp cat tipul de semnatura furnixata are o forma clara,compacta, iar analiza ei ajuta la descoperirea erorilor de programare.

[Cititorul va nota ca am scrtis cu majuscula tipurile specifice cum ar fi *Integer* si *Char*, dar nu am folosit majuscula pentru valori, cum era *inc*. Aceasta nu este doar o conventie ci o regula de sintaxa implementata ca atare in Haskell. De fapt si alte caractere mari sau nici conteaza pentru Haskell: foo, f0o si f00 sunt identificatori distincti].

2.1 Tipurile polimorfe (polimorfice)

Tipul polimorfic exprima in general familie de tipuri . De exemplu: (∀ a)[a] este o familie de tipuri – listele de 'a'-uri -pentru fiecare tip de „a”. Lista de caractere ['a','b','c','d'], lista intregilor[1,2,5] sunt toate dintr-o aceeasi familie (totusi [2,'b'] nu este un exemplu bun deoarece 2 si b nu fac parte din acelasi tip “a”).

[Identificatorii cum ar fi „a” deasupra sunt numiti tip de variatie si nu sunt catalogati sa faca diferenta de un tip specific cum ar fi inc. In plus, deoarece Haskell are doar intrebari de tip universal nu este nevoie de o scriere explicita din afara intrebarilor universale si scriem [a] in exemplus de mai sus. Cu alte cuvinte, toate tipurile de variabile sunt intrebari universale.]

Listele sunt frecvent folosite in structura functiilor si sunt un mijloc bun de a explica principiile polimorfismului.

Lista [1,2,3] scrisa asa in Haskell, este o scurtatura pentru 1 : (2: (3:[])), unde [ ] este o lista vida si „:” este operatorul cons care insereaza primul argument in lista care este cel de-al doilea. Asociind implicit „:” la dreapta putem scrie si:

„1:2:3:[]”. (n.tr. Nu uitati ca “:” este asociativ la dreapta nu la stanga cum sunt alti operatori aritmetici.)

Ca un exemplu de functie definita de utilizatorul care opereaza cu “:”, consideram o problema clasica, aflarea numarului de elemente dintr-o lista:

```
length :: [a] -> Integer

length [] = 0
length (x:xs) = 1 + length xs
```

Definitia este aproape imediata. Putem citi ecuatiile astfel: „lungimea listei vide este 0 si lungimea listei ce contine ca prim element pe „x” si coda „xs” este unu plus lungimea lui xs. (folosirea grupului xs este pluralul lui x si ar trebui citit ca atare). Intuitiv, acest exemplu evidentiaza un aspect important din Haskell care trebuie explicat: *pattern matching*-ul.

Partile stangi ale ecuatiilor contin sablonul [ ] si respectiv sablonul (x:xs). La evaluarea functiei parametrii actuali oferiti acesteia (aici lista de masurat) vor fi potriviti rand pe rand cu aceste sabloane, in ordinea de sus in jos. Acolo unde se potrivesc prima oara, acea ecuatie furnizeaza partea dreapta, formula rezultatului. Notati si ca [ ] se potriveste doar listei vide, pe cand x:xs se potriveste oricarei liste care are mai multe elemente, identificand pe x cu primul element si pe xs cu restul listei.

Daca potrivirea sabloanelor are loc (in partea stanga) a unei anume unei ecuatii, partea dreapta este evaluata si ea ofera rezultatul functiei. Calculul se opreste aici, chiar daca ar mai exista alte potriviri. Daca parametrii nu se potrivesc la o ecuatie, se trece la ecuatia urmatoare iar daca toate ecuatiile esueaza, atunci rezultatul este o eroare pe care sistemul Haskell o semnaleaza.

Definirea functiilor prin potrivire de sabloane este des intalnita in Haskell, iar utilizatorul trebuie sa devina familiarizat cu o varietate de sabloane (n.tr. cum este sablonul listelor (h:t) , sablonul perechii (x,y) etc) des intalnite. Vom dezbate aceasta problema in capitolul 4.

De asemenea, merita notat ca functia care calculeaza lungimea unei liste este un prim exemplu de *functie polimorfica*. Ea poate fi aplicata unei liste care contine elemente de orice tip, de exemplu [Integer], [Char] sau de ce nu, [[Integer]].

```
length [1,2,3]  =>  3
length ['a','b','c'] =>  3
length [[1],[2],[3]] => 3
```

Aici mai pot fi amintite inca doua functii polimorfice (des utilizate pe liste) care pot fi folosite de dumneavoastra mai tarziu. Functia *head* returneaza primul element din lista, iar functia *tail* returneaza celelalte elemente.

Spre deosebire de *length*, funcțiile acestea nu sunt definite pentru toate valorile posibile ale argumentului lor. Puteti determina care este capul listei vide ? atunci când funcțiile de mai sus sunt aplicate unei liste vide sistemul Haskell semnaleaza o eroare de executie.

Studiind tipurile polimorfe , descoperim că unele tipuri sunt cu siguranță mai generale decât altele adică setul de valori pe care îl definesc este mai vast. De exemplu , tipul [a] este mult mai general decât tipul [Char]. Cu alte cuvinte, ultimul tip scris poate fi derivat din tipul scris anterior substitiund a cu Char. Luând în considerare ordinea prin care generalizăm tipurile , sistemul de tipuri Haskell posedă două proprietăți importante: prima , fiecare expresie bine definită cu siguranță va avea un tip unic, principal, (explicat mai jos) şi a doua : tipul principal poate fi dedus, automat(§ 4.13). În comparație cu un limbaj monomorfic cum ar fi C , cititorul va descoperi că polimorfismul imbunătățeste expresivitatea iar tipurile deduse automat reduc bataia de cap a programatorului in materie de declaratii detipuri.

Tipul principal al unei funcții sau al unei expresii este cel mai general tip care intuitiv " conține toate instanțele expresiei" . De exemplu, tipul principal al lui **head** este [a] ->a; . Tipurile [b]->a; a->a, sau chiar a (!!) sunt tipuri corecte , dar prea generale pe când ceva ca [Integer] ->Integer este prea exact . Existența tipurilor principale unice este o proprietate caracteristică a sistemului de tipuri *Hindley Milner,* ce constituie baza sistemului de tipuri din Haskell, ML, Miranda şi din alte câteva limbaje diferite (majoritatea funcționale).

2.2 Tipuri definite de utilizator

Putem defini propriile tipuri în Haskell folosind o declarație de tipul **data,** pe care o prezentam in continuare cu ajutorul unei serii de exemple.(§4.2.1)

Un tip important, predefinit, în Haskell este acela al valorilor de adevăr:

data Bool = False | True

Tipul definit mai sus este de tip Bool şi are exact două valori: True şi False. Bool este un exemplu de constructor de tip , iar True si False sunt constructori de date nulari - cu zero argumente – li se poate spune doar constructori.
Similar putem defini tipul unei culori:

data Color = Red | Green | Blue | Indigo | Violet

Atât Bool , cât şi Color sunt exemple de tipuri enumerate, deoarece sunt formate dintr-un număr finit de constructori de date nulari.

Urmează un exemplu cu un singur constructor de date .

data Point a = Pt a a

Din cauza constructorului cu un singur tip , un tip ca **Point** este deseori numit *tuple type (tip n-uplu utilizator)* din moment ce este doar un produs cartezian (în acest caz binar) a unor alte tipuri.

În contrast, tipurile data având constructori multipli cum ar fi **Bool** si **Color** se numesc tipuri reuniune .

Mai important, **Point** este un exemplu de tip polimorfic. Pentru orice tip **t** **Point t** defineşte tipul punctelor din plan ce folosesc **t** ca tip de coordonate. Cuvantul **Point** poate fi văzut deci ca un *constructor de tip unar*, din moment ce din tipul **t** el construieşte un nou tip, **Point t.**

In acelaşi fel, folosind lista de exemple dată anterior, vedem ca [ ] este de asemenea un constructor de tip . Orice tip **t** dat poate fi dat ca argument pentru constructorul tipului listelor, [ ], formându-se un nou tip **[t]**. Sintaxa Haskell-ului permite ca tipul **[] t** să fie scris sub forma **[t]**. Similar , -> este un constructor de tip : date fiind două tipuri **t** si **u** , **t->u** este tipul funcțiilor care duc elementele de tip **t** în elemente de tip b .

Observați că tipul constructorului de date binar **Pt** este **a->a->Point a** şi astfel următoarele tipizari pentru puncte se pot scrie asa:

```
Pt 2.0 3.0 :: Point Float
Pt 'a' 'b' :: Point Char
Pt True False :: Point Bool
```

Pe de altă parte o expresie cum ar fi **Pt' a' 1** este scrisă greşit deoarece ‘a’ si 1 sunt de tipuri diferite. Este important să facem deosebirea dintre folosirea unui constructor de date pentru a produce o valoare şi folosirea unui constructor de tip pentru a produce un tip; primul proces are loc în momentul rulării programului şi tine de felul cum definim prin expresii valori în Haskell, pe când ultimul are loc în momentul compilării şi este o parte din procesul sistemului Haskell de asigurare a corectitudinii scrierii programului: verificarea tipurilor.

[Constructorii de tip cum ar fi **Point** şi constructorii de date , ca **Pt**, se găsesc în spații de nume separate. Aceasta permite ca acelaşi nume să fie folosit atât pentru constructorii de tip cât şi pentru constructorii de date ; ca şi în exemplu următor :

```
data Point a = Point a a
```

Deşi asemenea declaratii pot părea un pic confuze la început, ele ajută la crearea unui legături evidente între un tip şi constructorul său de date .]

2.2.1 Tipuri recursive

Tipurile pot fi deasemenea recursive cum sunt tipurile de arbori binari:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Aici noi am definit un tip de arbore binar polimorfic al cărui elemente sunt ori noduri frunze ce conțin o valoare de tip a , sau noduri interne (branch) ce conțin (recursiv) doi subarbori..

Când citiți declarații de date ca aceasta , amintiți-vă din nou că Tree este un *constructor de tip* , pe când Branch şi Leaf sunt *constructori de date*. Pe lângă faptul că stabileşte o legătură între aceşti constructori , declarația de mai jos defineşte următoarele tipuri pentru functiile speciale Branch şi Leaf:

Branch :: **(Tree a) (Tree a) -> (Tree a)**
Leaf :: Tree a -> (Tree a)

Odata cu acest exemplu de arbore avem definit un tip suficient de complex pentru a permite scriea unor functii (recursive) interesante care il folosesc. De exemplu, sa presupunem ca vrem sa definim o functie fringe care returneaza o lista a tuturor elementelor din frunzele unui arbore de la stanga la dreapta. De obicei ajuta sa scrii intai tipul noii functii; in acest caz observam ca tipul ar trebui sa fie Arbore a -> [a]. Aceasta inseamna ca *fringe* este o functie polimorfica care, pentru orice tip a, transforma arbori de *a* in liste de *a*. Urmeaza de aici o definitie convenabila:

```
fringe                          : : Arbore a -> [a]
fringe (Frunza x)               = [x]
fringe (Ramura stanga dreapta)  = fringe stanga ++ fringe dreapta
```

Aici ++ este un operator infixat care concateneaza doua liste (definitia sa completa va fi data in Sectiunea 9.1). La fel ca si exemplul cu lungimea dat mai devreme, functia fringe este definita folosind potrivirea de sabloane, insa aici observam sabloane implicand constructori definiti de utiliZator: Frunza si Ramura. [Observam ca parametrii formali sunt usor de identificat fiind cei care incep cu litere mici.]

2.2 Sinonime de tip

Pentru comoditatea utilizatorului, Haskell ofera un mod de a defini tipuri sinonime; cum ar fi, de exemplu, nume sugestive pentru tipuri folosite foarte des. Tipurile sinonime sunt create folosind o declaratie de tip care asociaza noul nume cu descrierea tipului (vezi Raport 4.2.2). Iata mai multe exemple:

```
type Sir                        = [Caracter]
type Persoana                   = (Nume, Adresa)
type Nume                       = Sir
type Adresa                     = None | Adr Sir
```

Sinonimele de tip nu definesc noi tipuri, doar dau noi nume tipurilor deja existente. Ca de exemplu, tipul Persoana -> Nume este echivalent cu (Sir, Adresa) -> Sir. Noile nume sunt adesea mai scurte decat cele ale tipurilor cu care sunt sinonime, dar acesta nu este singurul scop al sinonimelor de tip; ele pot imbunatati citirea programelor facandu-le mai usor de memorat; bineinteles, exemplul de mai sus subliniaza acest lucru. Putem sa dam noi nume si tipurilor polimorfice:

```
type ListaAsociativa  a b       = [(a, b)]
```

2.3 Tipurile predefinite nu sunt speciale

Mai devreme am prezentat mai multe tipuri predefinite cum ar fi listele, t-uplurile, numerele intregi si caracterele. Deasemenea am aratat cum pot fi realizate tipuri noi definite de utilizator. Pe langa sintaxa aparte, ne intrebam daca sunt si alte deosebiri intre tipurile predefinite din sistemul Haskell si cele definite apoi de utilizator ? Raspunsul este *nu*. Sintaxa speciala este pentru comoditate (este mai elegant cand scrii liste de forma [1,2,3] sa scrii tipul lor [Integer] si nu [] Integer), pentru consistenta cu conventia istorica, pentru compatibilitate cu vechile programe dar nu are importanta semantica.
Putem sublinia acest aspect luand in considerare cum ar arata declaratiile de tip pentru aceste tipuri incluse daca ni s-ar fi permis sa folosim in definirea lor sintaxa speciala. Ca de exemplu, tipul Caracter ar putea fi scris astfel :

```
data Char    = ‘a’ | ‘b’ | ‘c’ | . . .          -- Nu este Cod
             | ‘A’ | ‘B’ | ‘C’ | . . .          -- Haskell valid!
             | ‘1’ | ‘2’ | ‘3’ | . . .
                  . . .
```

Acesti contructori de nume nu sunt valizi sintactic; pentru a-i corecta ar trebui sa scriem ceva de genul :

```
data Char    = Ca | Cb | Cc | . . .              -- Nu este Cod
                  | CA | CB | CC | . . .         --  Haskell valid!
                  | C1 | C2 | C3 | . . .
                  . . .
```

Chiar daca acesti constructori sunt mai concisi, sunt destule de ciudati pentru a reprezenta caractere.
In orice caz, a scrie in acest mod ne ajuta sa intelegem sintaxa speciala a tipurilor predefinite, deosebita de a tipurilor utilizator. **Observam acum si ca tipul Caracter este doar un tip enumerat format dintr-un numar mare de contructori fara argumente.** A ne gandi la tipul Char in acest mod clarifica faptul ca putem folosi potrivirea sabloanelor de caractere in definitia functiilor, asa cum ne asteptam sa putem face acest lucru pentru orice contructor de tip (n.tr.ca in exemplul cu zilele saptamanii sau exemplele cu tipul Bool)
Acest exemplu deasemenea va invata utilizarea comentariilor in Haskell; introduse prin caracterele -- Toate caracterele de dupa ele pana la sfarsitul liniei sunt ignorate. Haskell permite comentariile incapsulate care sunt de forma {-. . .-} si pot fi introduce oriunde (2.2).
Similar, am putea defini Int (intregi cu interval de valori fixat) si Integer ca :

```
data Int = -65532 | ... | -1 | 0 | 1 | ... | 65532        -- mai mult un pseudo-cod
data Integer = ... -2 | -1 | 0 | 1 | 2  ...
```

unde -65532 si 65532, sa zicem, sunt intregii fixati de precizie maxima si minima pentru o implementare data. Int este o enumeratie doar ceva mai mare decat Caracter, dar este inca finita! In contrast, pseudo-codul pentru Integer este menit sa exprime o enumeratie infinita.

Tuplurile sunt usor de definit urmand exemplul oferit de Integer :

```
data (a,b)          = (a,b)          -- un pseudo-cod
data (a,b,c)        = (a,b,c)
data (a,b,c,d)      = (a,b,c,d)
...                   …
```

Fiecare declaratie de mai sus defineste un tip tuplu de o lungime anume, cu paranteze ( ) jucand un rol atat in sintaxa expresiei (pe post de constructor de date) cat si in sintaxa expresiei tipului (pe post de constructor de tip). Punctele de dupa ultima declaratie sunt menite sa exprime un numar infinit de astfel de declaratii, reflectand faptul ca limbajul Haskell permite sa folositi tupluri de orice lungime.

Listele sunt deasemeni usor de explicat, si mai interesant decat atat, sunt recursive:

```
data [a]          =  []
                  | a : [a]          -- mai mult pseudo-cod
```

Acum putem vedea clar ce am scris mai devreme despre liste: [] este lista vida, si : este operatorul infixat *cons*.

2.4 Tipurile constructorilor nu sunt speciale

Retineti: Constructorul de lista (:) are proprietatea de asociativitate la dreapta astfel încat lista [1, 2, 3] trebuie să fie si este echivalentă cu lista 1:2:3:[].
Tipul constructorului [] este [a] iar tipul constructoruluilui (:) este a->[a]->[a].
Modul în care ":" este definit (aici) este conform cu sintaxa. Constructorii infixati se pot folosi in declaratii de date și sunt deosebiți de catre sistem de operatorii infixati datorita faptului că *ei trebuie să înceapă cu un ":"* .
Ajuns în acest punct, cititorul ar trebui sa noteze cu grija diferențele dintre tupluri si liste, așa cum reies din definiția de mai sus. în particular, observăm: natura recursivă a tipului listei unde elementele sunt omogene ca tip si ca lista este de o lungime arbitrară, nefixata, precum și natura non-recursivă a unui t-uplu (cu un t particular) ale carui elemente sunt heterogene (de tipuri diferite) si formeaza o data compusa cu lungimea fixa. (evident, un t-uplu are t elemente.) Regulile sintactice pentru scrierea de t-upluri si liste ar trebui să fie deasemenea clare cititorului acum:
Pentru (e1,e2, …, en),n≥2, dacă ei are tipul ti atunci tipul tuplului este (t1, t2, …, tn).
Pentru [e1,e2, …, en],n≥0, fiecare ei trebuie să aibe acelas tip t, care nu este altul decat tipul elementelor listei .

2.4.1 Liste definite descriptiv si segvente aritmetice

Ca si in cazul dialectelor Lisp, listele sunt omniprezente in Haskell la fel ca in alte limbaje funcționale , insa mai exista inca multe lucruri de adaugat despre acest subiect scrierea listelor. In afară de constructorii de liste despre care am

discutat, Haskell prevede o expresie cunoscută ca *“list comprehension” (n.tr. Multimi ordonate definite descriptiv)* explicată cel mai bine prin exemplul:

```
[ f x | x <- xs ]
```

Această expresie poate fi citita intuitiv ca “ lista tuturor f x pentru care x este provenit din lista xs.” Notatia similara cu notatiile matematice nu este o coincidența. Subexpresia x <- xs este numită *generator (generator), și se pot folosi mai multi , nu numai unul* , ca in descrierea:

```
[ (x,y) | x <- xs, y <- ys ]
```

Multimea ordonata de mai sus (cu duplicate, eventual) descrie produsul cartezian a doua liste xs si ys. Elementele sunt selectate ca și cum generatoarele ar fi imbricate (eng. “nested”) din stanga in dreapta. Astfel, dacă xs este [1,2] si ys este [3,4], rezultatul este [(1,3), (1,4), (2,3), (2,4)].

In afară de generatoare, expresile boleane numite gărzi (*guards)* sunt de asemenea permise aici. Locul gărzilor depinde de generarea elementelor. De exemplu, iata o definiție precisă al algoritmului de sortare cunoscuta de toți:

```
quicksort []        = []
quicksort (x:xs)    = quicksort [y | y xs, y<x ]
                      ++ [x]
                      ++ quicksort [y | y xs, y>=x ]
```

Pentru a susține necesitatea folosirii listelor, Haskell foloseste o sintaxa speciala pentru *secvențe artmetice* ce sunt cel mai bine explicate in urmatoare serie de exemple:

```
[1 .. 10]     →    [1,2,3,4,5,6,7,8,9,10]
[1,3 .. 10]   →    [1,3,5,7,9]
[1,3 .. 10]   →    [1,3,5,7,9, …        (secventa infinita)
```

Vom discuta mai multe despre secvențe aritmetice la Secțiunea 8.2, iar despre “liste infinite” in Secțiunea 3.4.

2.4.2 String-uri

Ca un alt caz de sintaxă a unor constructoril de tipuri, observăm că șirul de litere “hello” este de fapt prescurtarea listei de caractere [‘h’ ,’e’,’l’,’l’,’o’]. Intr-adevăr, tipul lui “hello” este String, unde String este tipul predefinit echivalent cu (ceea ce dădusem ca un exemplu de mai devreme):

```
type String        =[Char]
```

Acesta inseamnă că putem utiliza functiile predefinite pentru a opera asupra listelor polimorfice și pentru a opera cu string-uri. De exemplu:

```
“hello” ++ “world”    =>    “hello world”
```

3 Funcții

Deoarece Haskell este un limbaj funcțional, oricine se poate astepta ca funcțiile să joace un rol major, si într-adevăr asa este. In aceasta sectiune, comentăm aspecte ale funcțiilor din Haskell.

Pentru început, considerăm aceasta definiție a unei funcții care adună cele două argumente ale sale:

```
add             :: Integer -> Integer -> Integer
add x y         = x + y
```

Acesta este un exemplu de scriere a unei funcții cu argumente succesive (eng - “curried”, ro -”curryzate”) . O aplicare a funcției “add” are forma “add e1 e2”, si este echivalentă cu (add e1) e2, deoarece aplicările funcțiilor sunt asociative la stinga. Cu alte cuvinte, aplicand *add* unui argument se produce o noua funcție care este aplicata celui de-al doilea argument. Acesta este potrivit cu tipul *add*-lui: Integer -> Integer -> Integer care este de altfel echivalent cu Integer -> (Integer -> Integer); ceea ce ne spune si că -> asociază la dreapta. Intr-adevăr utilizand funcția *add*, putem defini funcția “inc” într-un mod diferit de cel folosit anterior:

```
inc     =   add 1
```

Acesta este un exemplu al *aplicației partiale (partial application)* a unei funcții (curried), și este de asemenea (singura/o) cale prin care o funcție poate sa fie returnata ca o valoare. Să examinam un caz clasic in care o functie primeste o alta funcție ca argument. Exemplul este asa numita funcție *map*:

```
map             :: (a->b) -> [a] -> [b]
map  f   []     =   []
map f  (x:xs)   =   f   x   :   map   f   xs
```

[Aplicarea funcției are mai mare prioritate decat orice operator infixat si astfel partea dreapta a celei de a doua ecuație se interpretează ca (f x) : (map f xs).] Funcția map este polimorfica iar tipul ei indica clar că primul sau argument este o funcție. Observati de asemenea ca cele doua a-uri trebuie sa fie inlocuite cu același tip (la fel si b-urile). Ca un exemplu, prin folosirea lui map, putem incrementa toate elementele dintr-o lista:

```
map (add 1)   [1,2,3]     =>     [2,3,4]
```

Aceste exemple demonstrează natura funcțiilor: in Haskell functiile sunt entitati de prima clasa in limbaj. Iar functiile care sunt folosite in acest mod – de mai sus, primind ca argumente alte functii - sunt numite functii de nivel superior *(higher-order).*

3.1 Abstractii

Aceste exemple demonstrează ca funcțiile sunt elemente de prtma clasa in Haskell. Lucrul este evident atunbci tunci când sunt utilizate functii ca argumente ale altor functii, acestea din urma fiind numite de obicei funcții de ordin superior.

3.1 Lamda Abstracti

În loc de a folosi ecuații pentru pentru a construi functii nominale (functii cu nume), putem, ca alternativa, descrie funcțiile in mod anonim "printr-un lambda abstractie”. De exemplu, o funcție echivalentă a inc ar putea fi scrisa ca (\ x -> x +1) . Citeste: “functia care duce pe x in x+1” . În mod similar, funcția *add* este echivalenta cu (\ x -> (\ y -> x + y)). Abstracțiile imbricate cum este lambda de

mai sus, pot fi scrise folosind notația prescurtata echivalenta (\ xy -> x + y). De fapt, ecuațiile:

```
inc x =          x+1
add x y =          x+y
```

sunt într-adevăr prescurtare pentru:

```
inc =          \x -> x+1
add =          \x y -> x+y
```

Vom avea mai multe de comentat despre astfel de echivalari in alt subcapitol. În general, având în vedere că x are tipul t1 şi y – expresia de dupa -> are tipul t2 , rezulta ca (\ x-> y) are tipul t1-> t2.

3.2 Operatorii în Haskell:

În Haskell operatori sunt cu adevărat funcții, şi pot fi, asemeni functiilor, definiti de succesiuni de ecuații. Ca exemplu dam definitia unui operator clasic de concatenare a listelor:

```
(++)                :: [a] -> [a] -> [a]
[] ++ ys            = ys
(x:xs) ++ ys        = x : (xs++ys)
```

[Lexical vorbind, în Haskell operatorii constau în întregime din simboluri, spre deosebire de functii ale caror nume sunt identificatori, secvente alfanumerice care incep cu o litera si putr contine _ si ' (vezi Raport 2.4). Haskell nu are operatori prefixati, cu excepția lui minus (-), care este atât prefixat cat şi infixat.]

Ca un alt exemplu, un operator important in Haskell pentru lucrul cu funcții este operția de compunere:

```
(.)             :: (b->c) -> (a->b) -> (a->c)
f . g           = \ x -> f (g x)
```

3.2.1 Sectiunile

Deoarece, în Haskell operatorii sunt cu adevărat funcții, este logic ca si ei să se poată aplica parțial, (atunci cand primesc de exemplu doar un argument din doi). În Haskell aplicarea parțială a unui operator se (mai) numeşte *secțiune*. De exemplu, sunt echivalente notatiile:

```
(x+)    ≡    \y -> x+y
(+y)    ≡    \x -> x+y
(+)     ≡    \x y -> x+y
```

Paranteze sunt obligatorii. Ultimul exemplu de *secțiune* de mai sus, în esență, este o transcriere a operatorului + într-o forma de funcție, echivalenta lui. O asemenea functie se foloseste de exemplu atunci cand il transferam pe *plus* unei functii de ordin superior:

```
map (+) [1,2,3]
```

(cititorul ar trebui să verifice că această expresie returnează prin evaluare o listă de funcții, de exemplu tastand: :t map (+) [1,2,3]). Este, de asemenea necesar atunci când specificam o semnătură de tip funcție, ca în exemplele de la (++) şi (.) date mai devreme.

Putem vedea acum că se confirma ceea ce adăugasem mai devreme: add este chiar (+) şi inc este chiar (+1) ! Într-adevăr, aceste functii add si inc pot avea definițiile:

```
inc     = (+ 1)
add     = (+)
```

Putem transforma usor un operator într-o valoare funcțională, dar putem merge pe calea inversa ? Da. Folositi ca funcție operatorul scris intre apostroafe inverse : (eng. backquotes). De exemplu, *x `adauga` y* este acelaşi apel ca *adauga x y*. Unele funcții se pot citi chiar mai bine in acest fel, cand le gasiti intr-un program. Un exemplu este cautarea unui element pe o lista *elem membru lista*; expresia x 'elem` xs poate fi citita intuitiv ca predicatul de apartenenta „x apartine lui xs".

[Există câteva reguli speciale în ceea ce priveşte secțiunile a se vedea (Raportul Haskell 3.5, 3.4).]

In acest punct, cititorul poate fi de-a dreptul derutat: Haskell are atât de multe moduri de a defini o funcție! Este bine de stiut ca decizia de a furniza aceste notații vine in parte din convențiile „istorice", şi in parte din preocuparea pentru coerență .

3.2.2 Declarațiile de prioritate

O declarație de prioritate poate fi data pentru orice operator din Haskell sau pentru constructori (inclusiv operatorii obtinuti din functiile obişnuite, cum este `elem`). Această declarație specifica un nivel de prioritate de la 0 la 9 (9 fiind cel mai puternic; aplicarea functiilor se presupune a avea un nivel de prioritate de 10), precum si ordinea asocierii: la stânga, la dreapta, sau non-asociativ. De exemplu, pentru operațiile (++) şi (.) declaratiile sunt:

```
infixr  5 ++
infixr  9 .
```

Ambele specifica asociativitate la dreapta (infixr),cu un nivele de prioritate de 5, respectiv 9. Asociativitatea la stânga este specificata prin *infixl*, şi non-asociativitatea prin *infix (fara l sau r – right sau left)*. De asemenea, pentru mult de un operator putem folosi aceeaşi declarație infix. In lipsa unui *infix* pentru un un operator, sistemul Haskell considera implicit *infixl 9*. (A se vedea in Raportul

Haskell la $ 5.9 pentru cunoaşterea detaliată a regulilor de specificare a asociativității.)

3.3 Funcțiile sunt Non-stricte

Să presupunem că bot este dat de ecuatia: x = x