O mică introducere în Haskell 98

Paul Hudak
Universitatea Yale
Departamentul de Informatică

John Peterson
Universitatea Yale
Departamentul de Informatică

Joseph H. Fasel
Universitatea din California
Laboratorul Național Los Alamos

Octombrie 1999

Drepturi de autor
(c) 1999 Hudak Paul, John Peterson şi Fasel Joseph

Permisiunea se acordă, în mod gratuit, oricărei persoane care o obține o copie a „Gentle Introduction in Haskell "(textul acesta), să se ocupe de text fără restricții, inclusiv fără limitarea drepturilor de a folosi, copia, modifica, îmbina, publica, distribui, sublicenția, şi / sau vinde copii din text, şi de a permite persoanelor cărora textul modificat le este oferit este să facă acest lucru, sub rezerva următoarei condiți: notificarea dreptului de autor de mai sus şi obligația ca această notificare privind drepturile să fie inclusă în toate copiile sau porțiunile substanțiale ale textului.

1 Introducere

Scopul nostru când am scris acest tutorial nu era de a învăța programare, nici măcar de a preda programare funcționala. 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 în 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 Introducerea cărții lui Bird's despre Programare funcțională [1] sau cartea lui **Davis O Introducere în sisteme de programare funcțională - Utilizarea limbajului Haskell** [2]. Pentru un studiu util al limbajelor de programare funcționale şi aspectelor tehnice, incluzând aici unele dintre principiile de proiectare ale limbajelor folosite şi în Haskell, a se vedea [3].

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

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

Acestea nu sunt încă parte formalizată 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 potrivit din Raportul Haskell pentru detalii. Vă sugerăm, însă, să citiți detalile abia după ce ați parcurs această introducere în întregime.

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]) conțin(e) o mulțime de exemple utile de cod Haskell. Si vă recomandăm la o lectură atentă a lor îndată ce terminați acest tutorial. Acest lucru nu numai că vă va oferi o imagine a felului cum decurge programarea în Haskell şi cum arată programele reale, dar, de asemenea, vă 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. Paginile în limba Romåna sunt la adresa: http://www.haskell.org/haskellwiki/Ro/Haskell

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

Haskell este un limbaj de programare puternic tipizat

1. Tipurile sunt omniprezente, iar ca nou-venit este cel mai bine să deveniți conştienți de la început de puterea fantastică şi de complexitatea sistemului de tipuri din Haskell (n.tr. 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 dificilă de optică. Pentru cei familiarizați cu Java, C, Modula, sau chiar ML, adaptarea ar trebui să fie mai uşoară, dar nu imediată deoarece sistemul de tipuri din Haskell este diferit şi este mai bogat decât tot ce ați întâlnit până acum. În orice caz, programarea

tipizată face parte din experiența programării în 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 nişte seturi de valori.) Exemplele de expresii includ: 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 denotă valori, expresile de tip sunt termeni sintactici care denotă valorile de tip (numite doar „tipuri"). Exemple de expresii de tip includ următoarele tipuri atomice *Integer* (întregi de orice mărime), *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 clasă - 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 primă clasă (ele nu pot fi transmise ca argumente pentru funcții, nici returnate ca rezultate, nici plasate în structurile de date). Tipurile, într-un anume sens, descriu valori, şi asociația dintre o valoare şi tipul său se numeşte *specificație de tip* sau *semnatură*.

Utilizând 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". Funcțiile Haskell sunt în mod normal definite de o serie de ecuații. De exemplu, funcția "inc" poate fi definită de o singura ecuație:

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

O ecuație este un exemplu de declarație. Un alt fel de declarație este cea de tip, cu care putem preciza că o funcție anume, *inc* de exemplu, este de la Intregi la Intregi:

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

Vom discuta definițiile funcțiilor, pe larg, în capitolul 3.

În scop pedagogic, când vrem sa exprimam “e1 redus la altă expresie sau valoare e2” vom scrie:

e1 => e2.

De exemplu:

inc(inc 3) => 5

Sistemul Haskell defineste precis relația dintre tip şi valoare(vezi Raport $ 4.1.3)

Sistemul Haskell asigură o tipizare precisă statică, sigură, iar programatorul nu poate încălca tipul în nici un fel. De exemplu nu putem aduna doua caractere ca în expresia : ‘a’+’b’, asa cum este posibil în C. Principalul avantaj al tipizării statice este bine de ştiut de pe acum: Toate tipurile de erori sunt detectate din timp, la compilare. Nu toate erorile tastate pot fi recunoscute de sistemul de tipuri, cum ar fi expresia 1/0 care se poate scrie, dar în urma evaluării rezultatul va fi o eroare de execuție. Totuşi sistemul de tipuri descoperă multe erori de program […] şi de asemeni permite generarea unui cod mult mai eficient (de exemplu se elimină teste care în LISP ar fi fost necesare).

Acest sistem de tipuri asigură corectitudinea datelor utilizatorului. De fapt sistemul de tipuri din Haskell este destul de puternic pentru a permite scrierea oricărui fel de comandă. Scrierea tipului funcției *inc* este o idee bună, atâta timp cât tipul de semnătură furnizată are o formă clară, compactă, iar analiza ei automată ajută la descoperirea erorilor de programare.

[Cititorul va nota că am scris cu majusculă tipurile specifice cum ar fi *Integer* şi *Char*, dar nu am folosit majuscula pentru valori, cum era *inc*. Aceasta nu este doar o convenție ci o regulă de sintaxă implementată ca atare în Haskell. De fapt şi alte caractere mari sau mici contează pentru Haskell: foo, f0o şi f00 sunt identificatori distincți].

2.1 Tipurile polimorfe (polimorfice)

Tipul polimorfic descrie, la modul general, o 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 întregilor [1,2,5] sunt toate dintr-o aceeaşi familie (totuşi [2,'b'] nu este un exemplu bun deoarece 2 şi b nu fac parte din acelaşi tip *a*).

[Identificatorii cum ar fi „a” deasupra sunt numiți *variabile de tip* şi nu încep cu majuscule, ei deosebindu-se astfel de tipuri concrete, cum ar fi Int. În plus, deoarece Haskell are doar cuantificatori universali ∀ pentru tipuri nu este nevoie de o scriere explicită a cuantificatorului universal ∀ şi scriem [a] ca în exemplele de mai sus. Cu alte cuvinte, implicit, toate variabilele de tip sunt cuantificate universal.]

Listele sunt frecvent folosite în structura funcțiilor şi sunt un mijloc bun de a explica principiile polimorfismului.

Lista [1,2,3] scrisă aşa în Haskell, este o prescurtare pentru 1 : (2: (3:[])), unde [ ] este o lista vidă şi „:” este operatorul *cons* care inserează primul argument în lista care este cel de-al doilea. Operatorul „:” asociind implicit la dreapta putem scrie şi: „1:2:3:[]”. (n.tr. Nu uitati că “:” este asociativ la dreapta nu la stânga cum sunt alți operatori aritmetici.)

Ca un exemplu de funcție definită de utilizatorul care operează cu “:”, considerăm o problemă clasică, aflarea numărului de elemente dintr-o lista:

```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs
```

Definiția este aproape evidentă. Putem citi ecuațiile astfel: „lungimea listei vide este 0 şi lungimea listei ce conține ca prim element pe „x” şi coda „xs” este unu plus lungimea lui xs. (folosirea grupului xs este pluralul lui x şi ar trebui citit ca atare). Intuitiv, acest exemplu evidențiază un instrument important din Haskell care trebuie explicat: *pattern matching*-ul (n.tr. potrivirea de şabloane).

Părțile stângi ale ecuațiilor conțin şablonul [ ] şi respectiv şablonul (x:xs). La evaluarea funcției parametrii actuali oferiți acesteia (aici lista de măsurat) vor fi potriviți rând pe rând cu aceste şabloane, în ordinea de sus în jos. Acolo unde se potrivesc prima oară, acea ecuație furnizează partea dreaptă: formula rezultatului. Notați şi că [ ] se potriveste doar listei vide, pe cstângad x:xs se potriveşte oricărei liste care are mai multe elemente, identificând pe x cu primul element şi pe xs cu restul listei.

Dacă potrivirea şabloanelor are loc (în partea stângă) a unei anume ecuații, partea dreaptă este evaluată şi ea oferă valoarea funcției. Calculul se opreşte aici, chiar dacă ar mai exista alte potriviri. Dacă parametrii nu se potrivesc cu o ecuație, se trece la ecuația următoare iar dacă toate ecuațiile eşuează, atunci rezultatul este o eroare pe care sistemul Haskell o semnalează.

Definirea funcțiilor prin potrivire de şabloane este des folosită în Haskell, iar utilizatorul trebuie să devină familiarizat cu o varietate de şabloane (n.tr. cum este şablonul listelor (h:t) , şablonul perechii (x,y) etc.) des întâlnite. Vom dezbate această problemă în capitolul 4.

De asemenea, merită notat că funcția care calculează lungimea unei liste este un prim exemplu de *funcție polimorfică*. Ea poate fi aplicată unei liste care conține 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 încă doua funcții polimorfice (des utilizate pe liste) care pot fi folosite de dumneavoastră mai târziu. Funcția *head* returnează primul element din listă, iar funcția *tail* returnează lista celorlalte elemente.

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

Studiind tipurile polimorfice, 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 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 îmbunătățeste expresivitatea iar tipurile deduse automat reduc grijile programatorului în materie de declarații de tipuri.

Tipul principal al unei funcții sau al unei expresii este cel mai mic tip suficient de general să conțină 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 precis definit, prea limitativ. 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 tip, ***data*****,** pe care o prezentăm în continuare cu ajutorul exemplelor.(§4.2.1)

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

data Bool = False | True

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

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

sau (constructorii fiind aleşi de programator şi nefiind cuvinte rezervate):

data Color = Rosu | Galben | Verde | Albastru | 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 dar cu două argumente.

data Point a = Pt a a

Din cauza constructorului de date cu două argumente de tip a, 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** şi **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. Cuvântul **Point** poate fi văzut deci ca un *constructor de tip unar*, din moment ce din tipul **t** el construieşte un nou tip, **Point t.**

În acelaşi fel, revăzând lista de exemple date anterior, observăm că [ ] 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 şi u** , **t->u** este tipul funcțiilor care duc elementele de tip **t** în elemente de tip u .

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

Pt 2.0 3.0 :: Point Float (n.tr. Citeşte-l pe "::" ca "este de tipul")
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' şi 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 ține 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 declarații 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 inclusiv recursive, cum sunt, spre exemplu, arborii binari:

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

Aici noi am definit un tip de arbore binar polimorfic ale cărui elemente sunt ori noduri frunze ce conțin o valoare de tip a, ori noduri interne (Branch) ce conțin (aici e elementul recursiv al definiției) 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 funcțiile speciale Branch şi Leaf:

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

Odată cu acest exemplu de arbore avem definit un exemplu de tip suficient de complex pentru a permite scriea unor funcții (recursive) interesante care îl prelucrează. De exemplu, să presupunem că vrem să definim o funcție **fringe** care returnează o listă a tuturor elementelor din frunzele unui arbore de la stânga la dreapta. De obicei ajută să scrii întâi tipul noii funcții; în acest caz observăm că tipul ar trebui să fie Arbore a -> [a]. Aceasta înseamnă că **fringe** este o funcție polimorfică. Ea poate, pentru orice tip a, să transforme arbori de *a*-uri în liste de *a*-uri. Rezultă de aici o definiție convenabilă:

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

Aici ++ este un operator infixat care concatenează două liste (definiția sa completa va fi dată în Sectiunea 9.1). La fel ca şi exemplul cu lungimea dat mai devreme, funcția fringe este definită folosind potrivirea de şabloane, însă aici observăm şabloane care folosesc constructori definiți de utilizator: Frunza şi Ramura. [Observam că parametrii formali sunt usor de identificat fiind cei care încep 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 declarație de tip care asociaza noul nume cu descrierea tipului (vezi Raport 4.2.2). Iată 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 dău noi nume tipurilor deja existente. Ca de exemplu, tipul Persoana -> Nume este echivalent cu (Sir, Adresa) -> Sir. Noile nume sunt adesea mai scurte decât cele ale tipurilor cu care sunt sinonime, dar acesta nu este singurul scop al sinonimelor de tip; ele pot imbunătăți citirea programelor facându-le mai usor de memorat; bineințeles, exemplul de mai sus subliniaza acest lucru. Putem să dăm nume noi chiar şi 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 şi caracterele. Deasemenea am arătat cum pot fi realizate tipuri noi definite de utilizator. Pe langa sintaxa aparte, ne intrebăm dacă sunt şi alte deosebiri intre tipurile predefinite din sistemul Haskell şi cele definite apoi de utilizator ? Răspunsul este *nu*. Sintaxa modificată (specifică lor) este pentru comoditate (este mai elegant când scrii liste de forma [1,2,3] să scrii tipul lor **[Integer]** şi nu **[] Integer**), pentru consistență cu convenția istorică, pentru compatibilitate cu vechile programe dar nu are importanța semantică.
Putem sublinia acest aspect luând în considerare cum ar arăta declarațiile de tip pentru aceste tipuri predefinite dacă ni s-ar fi permis să folosim în definirea lor o sintaxă specială. Ca exemplu, tipul Caracter ar fi putut fi scris astfel :

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

Aceste nume de contructori de date nu sunt corecte sintactic; pentru a le corecta ar trebui să scriem ceva de genul :

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

Chiar dacă acesti constructori sunt mai concisi, sunt destul de ciudați pentru a reprezenta caractere.
În orice caz, a scrie în acest mod ne ajută să înțelegem sintaxa specială a tipurilor predefinite, deosebita de a tipurilor utilizator. **Observam acum şi că tipul Caracter este doar un tip enumerat format dintr-un numar mare de contructori fără argumente.** A ne gândi la tipul Char în acest mod clarifică faptul că putem folosi potrivirea şabloanelor de caractere în definiția funcțiilor, aşa cum ne aşteptăm să putem face acest lucru pentru orice contructor de tip (n.tr. ca în exemplul cu zilele săptămânii sau exemplele cu tipul Bool)
Acest exemplu vă invață deasemenea utilizarea comentariilor în Haskell; introduse prin caracterele -- Toate caracterele de după ele până la sfârşitul liniei

sunt ignorate. Haskell permite comentariile pe mai multe rânduri care sunt de forma {-. . .-} şi pot fi plasate oriunde în program. (2.2).
Similar, am putea defini Int (intregi cu interval de valori fixat) şi Integer ca fiind :

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

unde -65532 şi 65532, să zicem, sunt întregii fixați de precizie maximă şi minimă pentru o implementare dată. Int este un tip enumerat doar ceva mai mare decât Caracter, dar este totuşi un tip finit ! În contrast, pseudo-codul cu '…' pentru tipul Integer este menit să exprime o enumerație infinită.

Tuplurile sunt uşor de explicat considerand o sintaxă specială si urmând 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 declarație de mai sus defineşte un tip tuplu de o lungime anume, cu paranteze ( ) jucând un rol atât în sintaxa expresiei (pe post de constructor de date) cat şi în sintaxa expresiei tipului (pe post de constructor de tip). Punctele de după ultima declarație sunt menite să exprime un număr infinit de astfel de declarații, reflectând faptul că limbajul Haskell permite să folosiți tupluri de orice lungime.

Listele sunt deasemenea uşor de explicat şi mai interesant decât atât, sunt concepute ca tip recursiv:

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

Acum putem vedea clar ce am scris mai devreme despre liste: [] este lista vidă, iar “:” este operatorul infixat *cons*.

2.4 Tipurile constructorilor nu sunt speciale

Rețineți: Constructorul de listă (:) are proprietatea de asociativitate la dreapta deoarece lista [1, 2, 3] trebuie să fie şi 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 infixați se pot folosi în declarații de date și sunt deosebiți de către sistem de operatorii infixați datorită faptului că *ei trebuie să înceapă cu un “:”* - restricție sintactică.
Ajuns în acest punct, cititorul ar trebui să noteze cu grijă diferențele dintre tupluri şi liste, așa cum reies din definiția de mai sus. în particular, observăm:

- natura recursivă a tipului listei unde elementele sunt omogene ca tip şi că lista este de o lungime arbitrară, nefixată, precum și

– natura non-recursivă a unui t-uplu (cu un t particular) ale cărui elemente sunt heterogene (de tipuri diferite) şi formează o dată compusă cu lungimea fixă. (evident, un t-uplu are t elemente.) Regulile sintactice pentru scrierea de t-upluri şi 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 aibă acelaşi tip t, care nu este altul decât tipul elementelor listei **.**

2.4.1 Liste definite descriptiv şi secvențe aritmetice

Ca şi în cazul dialectelor Lisp, listele sunt omniprezente în Haskell la fel ca în alte limbaje funcționale , însă mai există încă multe lucruri de adaugat despre acest subiect scrierea listelor. În afară de constructorii de liste despre care am discutat, Haskell prevede o expresie cunoscută ca *"list comprehension" (n.tr. Mulțimi ordonate definite descriptiv)* explicată cel mai bine prin exemplul:

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

Această expresie poate fi citită intuitiv ca " lista tuturor f x pentru care x este provenit din lista xs." Notația similară cu notațiile matematice nu este o coincidență. Subexpresia x <- xs este numită *generator (generator), și se pot folosi mai mulți , nu numai unul* , ca în descrierea:

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

Mulțimea ordonata de mai sus (cu duplicate, eventual) descrie produsul cartezian a doua liste xs şi ys. Elementele sunt selectate ca și cum generatoarele ar fi imbricate (eng. "nested") din stânga în dreapta. Astfel, dacă xs este [1,2] şi ys este [3,4], rezultatul este [(1,3), (1,4), (2,3), (2,4)].
În afară de generatoare, expresile boleane numite gărzi (*guards)* sunt de asemenea folosite aici. Ordinea gărzilor contează la generarea elementelor! De exemplu, iată o definiție a algoritmului de sortare, cunoscută de toți:

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

Pentru a promova folosirea listelor, Haskell foloseste o sintaxă specială pentru *secvențele artmetice* care sunt cel mai bine explicate de următoarea 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, …        (secvența infinită)
```

Vom discuta mai multe despre secvențe aritmetice la Secțiunea 8.2, iar despre "liste infinite" în 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 înseamnă că putem utiliza funcțiile 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, şi într-adevăr aşa este. În această secțiune, comentăm unele aspecte particulare ale funcțiilor din Haskell.

Pentru început, considerăm această 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”, şi este echivalentă cu (add e1) e2, deoarece aplicările funcțiilor sunt asociative la stânga. Cu alte cuvinte, aplicând *add* unui argument se produce o noua funcție care este aplicată 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 şi că -> asociază la dreapta. Intr-adevăr utilizând funcția *add*, putem defini funcția “inc” într-un mod diferit de cel folosit anterior:

```
inc          =      add 1
```

Acesta este un exemplu al *aplicației parțiale (partial application)* a unei funcții (curried), și este de asemenea (singura) cale prin care o funcție poate să fie returnată ca o valoare. Să examinăm un caz clasic în care o funcție primește o altă funcție ca argument. Exemplul este așa 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 decât orice operator infixat şi astfel partea dreaptă a celei de a doua ecuații se interpretează ca (f x) : (map f xs).] Funcția map este polimorfică iar tipul ei indică clar că primul său argument este o funcție. Observați de asemenea că cele doua a-uri trebuie să fie înlocuite cu același tip (la fel şi b-urile). Ca un exemplu, prin folosirea lui map, putem incrementa toate elementele dintr-o listă:

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

Aceste exemple demonstrează că funcțiile sunt elemente de primă clasă în Haskell. Lucrul este evident atunci când sunt utilizate funcții ca argumente ale altor funcții, acestea din urmă fiind numite de obicei funcții de ordin superior *(higher-order)*.

3.1 Lamda Abstracții

În loc de a folosi ecuații pentru a construi funcții nominale (funcții cu nume), putem, ca alternativă, să descriem funcțiile în mod anonim "printr-o lambda abstracție" (n.tr. "**(**" chiar seamănă cu litera grecească lambda). De exemplu, o funcție echivalentă cu inc ar putea fi scrisă ca (\ x -> x +1). Citește: "funcția care duce pe x în x+1" . În mod similar, funcția *add* este echivalentă cu (\ x -> (\ y -> x + y)). Abstracțiile imbricate cum este lambda expresia de mai sus, pot fi scrise folosind notația prescurtată echivalentă (\ x y -> 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 echivalări în alt subcapitol. În general, având în vedere că x are tipul t1 şi y – expresia de dupa -> are tipul t2 , rezulta că (\ x-> y) are tipul t1-> t2.

3.2 Operatori în Haskell

În Haskell operatori sunt cu adevărat funcții, şi pot fi, asemeni funcțiilor, definiți de succesiuni de ecuații. Ca exemplu dăm definiția 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 funcții ale căror nume sunt identificatori, secvente alfanumerice care incep cu o litera şi putr conține _ şi ' (vezi Raport 2.4). Haskell nu are operatori prefixați, cu excepția lui minus (-), care este atât prefixat cât şi infixat.]

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

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

3.2.1 Secțiunile

Deoarece, în Haskell operatorii sunt cu adevărat funcții, este logic ca şi ei să se poată aplica parțial, (atunci când primesc de exemplu doar un argument din doi).

În Haskell aplicarea parțială a unui operator se (mai) numeşte *secțiune*. De exemplu, sunt echivalente notațiile:

```
(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 formă de funcție, echivalentă lui. O asemenea funcție se foloseste de exemplu atunci când il transferam pe *plus* unei funcții de ordin superior:

map (+) [1,2,3]

(cititorul ar trebui să verifice că această expresie returnează prin evaluare o listă de funcții, de exemplu tastând: :t map (+) [1,2,3]). Este, de asemenea necesar atunci când specificăm 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 funcții add şi inc pot avea definițiile:

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

Putem transforma uşor un operator într-o valoare funcțională, dar putem merge pe calea inversă ? Da. Folosiți 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 în acest fel, când le găsiți intr-un program. Un exemplu este căutarea unui element pe o listă *elem membru lista*; expresia x 'elem` xs poate fi citită intuitiv ca predicatul de apartenență „x aparține lui xs”.

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

În acest punct, cititorul poate fi de-a dreptul derutat: Haskell are atât de multe moduri de a defini o funcție! Este bine de ştiut că decizia de a furniza aceste notații vine în parte din convențiile „istorice”, şi în parte din preocuparea pentru coerență .

3.2.2 Declarațiile de prioritate

O declarație de prioritate poate fi scrisă pentru orice operator din Haskell sau pentru constructori (nota bene: inclusiv operatorii obținuti din funcțiile obişnuite, cum este `elem`). Această declarație specifică un nivel de prioritate de la 0 la 9 (9 fiind cel mai puternic; doar aplicarea funcțiilor se presupune a avea un nivel

de prioritate de 10), precum şi ordinea asocierii: la stânga, la dreapta, sau non-asociativ. De exemplu, pentru operațiile (++) şi (.) declarațiile sunt:

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

Ambele specifică asociativitate la dreapta (infixr),cu un nivele de prioritate de 5, respectiv 9. Asociativitatea la stânga este specificată prin *infixl*, iar non-asociativitatea prin *infix (fără l sau r – right sau left)*. De asemenea, pentru mult de un operator putem folosi aceeaşi declarație infix. În lipsa unui *infix* pentru un un operator, sistemul Haskell consideră implicit *infixl 9*. (A se vedea în Raportul Haskell la $ 5.9 pentru cunoaşterea detaliată a regulilor de specificare a asociativității.)

3.3 Funcțiile sunt nestricte

Să presupunem că bot este dat de ecuația:

```
bot = bot
```

Cu alte cuvinte bot este o expresie a cărei evaluare nu se termina. La modul abstract, notam cu _|_ valoarea unei expresii a cărei evaluare nu se termină (citeste 'bottom'). Expresiilor care produc erori la execuție, cum este 1/0 li se atribuie, şi lor, convențional, această valoare. O asemenea eroare nu este evitabilă, tratabilă: programele nu continuă dincolo de locul apariției unei asemenea erori. Aceasta spre deosebire de erorile produse de sistemul de I/O, cum este eroarea *end-of-file*, care sunt reparabile şi pot fi tratate într-o altă manieră. (De fapt aceste erori de I/O nu sunt propriuzis erori ci mai curând excepții. Despre excepții, găsiți mai multe în Secțiunea 7.)

O *funcție f se spune că este strictă* dacă aplicată fiind unei expresii care nu se termină, de asemenea nu reuşeste să se termine. Cu alte cuvinte, f este strictă dacă valoarea lui f bot este _|_. Situația comună, cazul celor mai multe limbaje de programare, este că au TOATE funcțiile stricte (n.tr. ceea ce înseamnă că o eroare opreşte orice funcție în care apare.) În Haskell este ceva mai bine decât atât. Nu toate funcțiile sunt stricte. Ca exemplu, considerați funcția const1, constanta 1, definită prin:

```
const1  x  =  1
```

Ei bine, în Haskell, valoarea lui *const1 bot* este 1. Operațional vorbind, deoarece *const1* nu are nevoie de valoarea argumentului său; nu va încerca să-l evalueze, şi nu va intra în acel calcul fără sfârsit. Din acest motiv, funcțiile nestricte sunt numite şi funcții leneşe (eng. lazy function), şi se spune că îşi evaluează argumentele în mod leneş (eng. lazy) ori la nevoie (eng. "by need").

Deoarece în Haskell erorile şi neterminarea evaluării sunt echivalente, din punct de vedere semantic, argumentul / paragraful, de mai sus este valabil şi pentru erori. De exemplu, const1 (1/0) de asemenea are valoarea 1.

Funcțiile nestricte sunt utile de asemenea intr-o serie de alte situații. Un avantaj major al lor este că ele eliberează programatorul de grija privind ordinea de evaluare. Expresii (n.tr. cum sunt funcțiile recursive) mari consumatoare de timp de calcul pot fi date ca argumente unor funcții fără grija că ele vor fi calculate şi atunci când nu este nevoie de ele. Alt avantaj este faptul că se pot concepe şi sunt posibile structuri de date infinite din acelaşi motiv. (n.tr. care de asemenea se parcurg sau se generează până unde este nevoie şi atât.)

Un alt mod de a explica, poate mai bine, ideea de funcții nestricte, ca în Haskell, ar fi să ne gândim că = este o definiție nu o atribuire, cum este în limbajele tradiționale. Citim şi ințelegem o asemenea declarație de forma:

```
v = 1/0
```

ca "numim v pe 1/0" în loc de "calculează 1/0 şi stochează rezultatul în v". Numai dacă valoarea definită de v ar fi necesară se va încerca împărțirea prin zero. Dar prin ea însăşi, declarația nu implică nici o tentativă de a se face un calcul (n.tr. Cam ca #define v 1/0 din C care defineste o substituție). Pe urmă nu uitați că programarea cu atribuiri cere o grijă a ordonării acestor atribuiri: funcționarea programului depinde evident de ordinea în care atribuirile sunt efectuate. Definițiile, pe de altă parte sunt mult mai simple: ele pot fi a şezate în orice ordine fără a strica funcționarea programului, fără a-i altera sensul.

3.4. Structuri de date "infinite"

Unul dintre avantajele naturii nestricte a funcțiilor din Haskell este faptul că şi constructorii (n.n. de date) sunt funcții nestricte, deoarece constructorii sunt doar o categorie aparte de funcții – distinctiv fiind faptul că ei pot fi folosiți în scrierea şabloanelor pentru *pattern-matching*.) De exemplu, constructorul pentru liste "cons", notat (:) este şi el nestrict, ceea ce vom folosi în exemplul următor:

```
unitati                          = 1 : unitati
```

Poate mai interesantă ar fi funcția numereDeLa:

```
numereDeLa n                     = n : numereDeLa (n+1)
```

Astfel *numereDeLa n* este lista infinită a întregilor începând cu n. Cu ajutorul ei putem defini, de exemplu, lista infinită a pătratelor numerelor naturale:

```
patrate             = map (^2) (numereDeLa 0)
```

(Observați: am folosit o secțiune, o aplicare parțială a operatorului binar ridicare la putere, ^ care a primit la început doar un argument.) Lista fiind infinită, în practică ne asteptăm să o folosim extragând din ea doar o porțiune finită. Si există o serie de funcții standard în Haskell care pot fi folosite în acest scop: **take, takeWhile, filter** şi altele. Limbajul Haskell include un set larg de funcții şi tipuri predefinite care formează aşa-zisul “Standard Prelude” - biblioteca standard. Listingul bibliotecii Standard Prelude este inclus în anexa din Raportul Haskell; căutați porțiunea numită “Prelude Listing” unde veți găsi o mulțime de funcții care procesează liste. De exemplu, *take* extrage primele n elemente dintr-o listă:

```
take 5 patrate                  => [0,1,4,9,16]
```

Definiția listei **unitati** de mai inainte este un exemplu de *lista circulară*. În majoritatea cazurilor caracterul “lazy” al funcțiilor are impact pozitiv asupra eficienței programelor. […]
Un alt exemplu de folosire a recursivității , şirul lui Fibonacci, poate fi calculat eficient prin folosirea următoarei secvențe infinite:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib)]
```

unde **zip** este o funcție din Standard Prelude care returnează perechi de elemente luate de pe ambele liste, conform algoritmului:

```
zip (x:xs) (y:ys)       = (x,y) : zip xs ys
zip xs ys               = []
```

Observați că fib, ca şi alte liste infinite, este definită recursiv, ca şi cum şi-ar “înghiți singură coada”. Si intr-adevăr, am putea desena o imagine a acestui calcul aşa cum se vede în Figura 1.

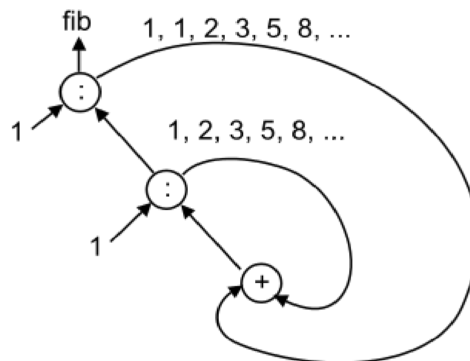

Figura 1: Sirul lui Fibonacci

Pentru a vedea o altă aplicație a listelor infinite, citiți Secțiunea 4.4. (cap. 4.4)

3.5 Funcția Error

Haskell are o funcție integrată numită **error** al cărei tip este **String->a**. Această funcție este oarecum ciudată: Tipul ei arată ca şi cum ar întoarce o valoare de tip polimorfic despre care nu ştie nimic, de vreme ce nu primeşte niciodată o valoare de acest tip ca argument!
De fapt *există* o valoare comună tuturor tipurile: _|_. Într-adevăr, semantic aceasta este exact valoarea care este întotdeauna întoarsă de **error** (Amintiți-vă că toate erorile au valoarea _|_). Oricum, ne putem aştepta ca o implementare rezonabilă să afişeze argumentul de tip String primit de **error** pentru depanare. Aşadar această funcție este folositoare când dorim să terminăm un program sau când un parametru primit a fost greşit. De exemplu, adevărata definiție a funcției **head** luată din Standard Prelude este:

```
head  (x:xs)        =  x
head   [ ]          =  error "head{PreludeList}: head [ ]"
```

4.Expresii Case şi Pattern Matching (potrivire de şabloane)

Mai devreme am dat câteva exemple de potrivire de şabloane în definirea funcțiilor --- de exemplu **length** şi **fringe**. În această secțiune vom examina mai în detaliu procesul de potrivire de şabloane (§3.17).

Şabloanele nu sunt valori de primă clasă; există un set fixat de diferite tipuri de şabloane. Am văzut deja câteva exemple de şabloane de constructori de date; atât **length** cât şi **fringe** definite mai devreme sunt construite folosind astfel de şabloane, primul folosind constructorul unui tip predefinit (liste), ultimul folosind tipul definit de utilizator (arbori). Într-adevăr, potrivirea de şabloane este permisă folosind orice constructori de tip, definiți de utilizator sau nu. Printre aceştia se includ n-uple, stringuri, numere, caractere, etc. Ca exemplu, iată o funcție *contrived* care potriveşte un n-uplu de constante.

```
contrived  : :  ([a],  Char,  (Int,  Float),  String,  Bool)  -> Bool
contrived       ([ ],    'b',     (1, 2.0),      "hi",    True)   = False
```

Acest exemplu demonstrează deasemenea că încuibărirea şabloanelor este permisă (la o adâncime arbitrară).

Tehnic vorbind, *parametrii formali* sunt asemenea şabloanelor doar că ei nu eşuează niciodată în potrivirea cu o valoare. Ca o consecință a potrivirii cu succes, parametrul formal este legat la valoarea cu care a fost potrivit. Din acest motiv şabloanele dintr-o ecuație nu pot include mai multe apariții ale aceluiaşi parametru formal (o proprietate numita *liniaritate* §3.17,§3.3,§4.4.2).

Şabloanele de felul parametrilor formali, care nu dau niciodată greş la potrivire, se spune că sunt *irefutabile*, în contrast cu şabloanele *refutabile* care pot eşua la potrivire. Şablonul folosit în exemplul cu funcția *contrived* de mai sus este refutabil. Mai sunt încă trei tipuri de şabloane irefutabile, iar două dintre ele le vom prezenta acum (celălalt îl vom amâna până la Secțiunea 4.4).

Şabloanele-@.

Câteodată este convenabil să dăm un nume unui întreg şablon necesar pentru folosirea în partea dreaptă a unei ecuații. De exemplu, o funcție care duplică primul element dintr-o listă poate fi scrisă aşa:

```
f  (x : xs)                = x : x : xs
```

(Amintiți-vă că „:” asociază la dreapta.) Țineți minte că x:xs apare şi ca şablon în partea stângă, dar şi ca expresie în partea dreaptă. Pentru a îmbunătăți lizibilitatea programului, ar fi de preferat să scriem x:xs o singură dată, lucru pe care îl putem obține folosind *şabloanele*-@ după cum urmează:

```
f  s@(x : xs)          = x : s
```

Tehnic vorbind, şabloanele-@ întotdeauna reuşesc cu succes să se potrivească, deşi sub-şablonul (în acest caz x:xs) ar putea, desigur, să eşueze.

Jockeri. Altă situație des întâlnită este potrivirea cu o valoare despre care nu ne interesează nimic, (n.n. pe care nu o vom folosi în calcule.). De exemplu, funcțiile head şi tail definite în Secțiunea 2.1 pot fi rescrise astfel:

```
head  (x : _)          = x
tail    (_ : xs)         = xs
```

în care am făcut vizibil faptul că nu ne interesează care este o anumită parte a intrărilor. Fiecare jocker se potriveşte independent cu orice ar fi, dar spre deosebire de un parametru formal, niciunul nu se leagă de nimic; din acest motiv pot coexista mai mult de unul într-o aceeaşi ecuație.

4.1 Semantica potrivirii de şabloane

Până acum am discutat despre şabloanele individuale: sunt potrivite, cum unele sunt refutabile, altele sunt irefutabile, etc. Dar cum decurge procesul în ansamblu? În ce ordine sunt încercate potrivirile? Dar dacă nu reuşeşte niciuna? Această secțiune se adresează acestor întrebări.

Potrivirea şabloanelor *poate eşua,* poate *reuşi* sau *diverge*. O potrivire reuşită leagă parametrii formali din şablon de valori. Divergența apare când o valoare necesară şablonului conține o eroare (_|_). Procesul de potrivire

decurge de sus în jos, şi de la stânga la dreapta. Eşecul unui şablon oriunde într-o ecuație înseamnă eşecul întregii ecuații, şi apoi este încercată următoarea ecuație. Dacă toate ecuațiile eşuează, valoarea funcției de aplicare este _|_, şi are ca rezultat o eroare în timpul execuției.

De exemplu, dacă [1, 2] este potrivit cu [0, **bot**], atunci 1 eşuează să se potrivească cu 0, aşa că rezultatul este o potrivire *eşuată*. (Amintiți-vă că **bot**, definit mai devreme, este o variabilă legată la _|_). Dar dacă [1, 2] este potrivit cu [bot, 0], atunci potrivirea lui 1 cu **bot** produce divergență (adică _|_).

Cealaltă derogare de la acest set de reguli este că şabloanele (de nivel înalt - exterioare) pot avea de asemenea o gardă booleană, ca în această definiție a unei funcții care ne furnizează semnul unui număr:

```
sign x | x > 0        = 1
       | x == 0       = 0
       | x < 0        = -1
```

Țineți minte că pentru un acelaşi şablon pot fi indicate mai multe gărzi, tratând mai multe cazuri. Ca şi şabloanele, gărzile sunt evaluate de sus în jos. Prima care este evaluată la True (Adevărat) produce o potrivire cu succes şi alege astfel formula de calcul a rezultatului.

4.2 Un exemplu

Ordinea ecuațiilor poate avea un efect subtil asupra rulării funcțiilor. De exemplu, să considerăm această definiție a funcției take:

```
take   0    _         =  []
take   _    []        =  []
take   n    (x:xs)    =  x : take (n-1) xs
```

şi această versiune puțin modificată (primele 2 ecuații au fost inversate)

```
take1  _    []        =  []
take1  0    _         =  []
take1  n    (x:xs)    =  x : take (n-1) xs
```

Acum observați următorul set de apeluri:

```
take    0    bot     =>  []
take1   0    bot     =>  ⊥

take    bot  []      =>  ⊥
take1   bot  []      =>  []
```

Se poate observa că take este mai bine definit în ceea ce priveste al doilea argument, iar take1 este mai bine definit în ceea ce priveste primul sau argument. Este dificil de spus în acest caz care definiție este mai buna. Rețineți însă faptul ca în anumite aplicații se poate sa apară o diferență. (Biblioteca “Standard Prelude” include o definiție similara funcției take).

4.3. Expresii Case

Succesiunea şabloanelor oferă o metodă de a trata mai multe cazuri ce se disting prin proprietățile structurale ale unei valori. În multe situații nu e de dorit definiriea unei funcții ori de cate ori avem nevoie de un caz, dar pană acum s-a aratat doar cum se utilizează şabloanele în definițiile funcțiilor. Instrucțiunea Case din Haskell ne ofera o soluție suplimentară pentru rezolvarea problemei. De altfel, folosirea şabloanelor în definițiile funcțiilor este explicată în Raport pe baza expresiilor Case, care sunt considerate a fi mai simple.

O definiție a unei funcții de forma:

$$
\begin{aligned}
&f\ p_{11}\ \ldots\ p_{1k} = e_1\\
&\ldots\\
&f\ p_{n1}\ \ldots\ p_{nk} = e_n
\end{aligned}
$$

unde fiecare p_{ij} este un sablon este echivalenta semantic cu:

$$
\begin{aligned}
f\ x1\ x2\ \ldots\ xk = \text{case } (x1, \ldots, xk)\ \text{ of }\ &(p_{11}, \ldots, p_{1k}) \to e_1\\
&\ldots\\
&(p_{n1}, \ldots, p_{nk}) \to e_n
\end{aligned}
$$

unde xi sunt identificatori noi. (Pentru o traducere mai generală, vedeți 4.2.2)
De exemplu, definiția funcției take prezentata mai sus este echivalentă cu:

```
take  m  ys             =  case (m, ys)  of
                             (0, _ )          ->  []
                            (  _, [])         ->  []
                            ( n, x:xs)        ->  x : take (n-1) xs
```

Un principiu care nu a fost explicat mai devreme este că pentru corectitudine, toate tipurile din partea dreapta a unor expresii Case sau a unor seturi de ecuații ce apar în definiția unei funcții trebuie sa fie (toate) compatibile; mai precis toate trebuie să aibă un tip comun.

Regulile de utilizare a şabloanelor pentru expresii Case sunt la fel ca şi cele pentru definiții de funcții, deci nu este nimic nou aici, în afara de faptul că expresiile Case sunt mult mai convenabile la folosit. Intr-adevar, există o situație de folosire a expresiilor Case şi este atât de des întâlnită, încât are o sintaxă specială: expresia condițională. În Haskell, expresiile condiționale au următoarea formă familiară:

if e_1 then e_2 else e_3

care este forma scurtă pentru:

case e_1 of True -> e_2
False -> e_3

Din acest exemplu ar trebui să fie clar faptul că e_1 va trebui să aibă tipul Bool, iar e_2 şi e_3 trebuie să aibă acelaşi tip. Cu alte cuvinte, if-then-else este văzut ca o funcție cu tipul Bool->a->a->a.

4.4 Sabloane lazy (eng. leneşe)

Mai există un tip de şabloane utilizat în Haskell. Este numit şablon lazy (amânat) şi are forma ~*pat*. Sabloanele lazy sunt irefutabile: potrivirea unei valori v cu ~*pat* are succes întotdeauna, indiferent de *pat*. În termeni operaționali, dacă un identificator din *pat* este folosit mai târziu în partea dreaptă a unei expresii, va fi legat de acea parte a valorii care ar rezulta dacă v s-ar potrivi cu *pat* şi cu _|_ în caz contrar.

Sabloanele lazy sunt folositoare în situații în care structuri infinite de date sunt definite recursiv. De exemplu listele infinite sunt instrumente excelente la scrierea programelor de simulare. În acest context listele infinite sunt deseori numite *fluxuri*. Să considerăm cazul simplul al simulării interacțiunii dintre un server şi un client, unde clientul trimite o secvență de interogări serverului, iar serverul răspunde fiecarei interogări cu un (tip anume de) răspuns. Această situație este arătată în figura 2. (De notat faptul că clientul primeşte un mesaj inițial ca argument.) Folosind fluxuri pentru a simula secvența mesajelor, codul Haskell corespunde următoarei diagrame a sistemului :

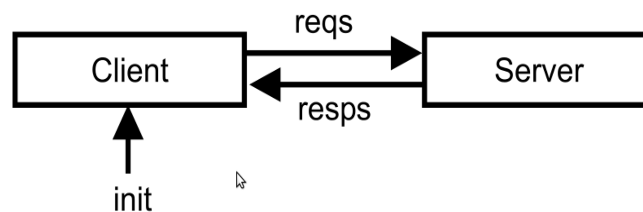

Figura 2.

şi este:

```
reqs                    =  client init resps
resp                    =  server reqs
```

Aceste ecuații recursive sunt o transcriere cuvânt cu cuvânt a diagramei. Să presupunem în continuare că schemele serverului şi clientului arată astfel:

```
client init  (resp:resps)  =  init : client ( next resp) resps
server       (req:reqs)    =  process req : server reqs
```

(va urma)
Ha$kell
3 aprilie 2011