A HASKELL ÉS A CLEAN NYELV ÖSSZEHASONLÍTÓ ELEMZÉSE*

COMPARATIVE ANALYSIS OF HASKELL AND CLEAN

Hegedűs Hajnalka, heha@inf.elte.hu
Horváth Zoltán, hz@inf.elte.hu
Zsók Viktória, zsv@inf.elte.hu
Eötvös Loránd Tudományegyetem
Általános Számítástudományi Tanszék

Abstract

Both Haskell and Clean are purely functional languages, using lazy evaluation. Haskell is a relatively wide-spread language, so a great number of application and program library was written for it. It is also some kind of standard among functional languages. In contrary, Clean is a dynamically developing language which is continously increased with new language elements. One of its other advantages is that it has a compiler which is more efficient then most of the compilers for other purely functional languages. It also has a user-friendly developing environment. These are the reasons why the idea of designing a system which unifies the advantages of both languages arose. For the design of this it is essential to examine and compare the two languages.

Though the theoretical background of the two languages is quite similar, they have a lot of really different structures. They have radically different concepts to implement features which are not concidered really functional (like I/O). There are also elements which are present in one of the languages and missing from the other. This comparison is to describe the standard language elements of the two languages.

Összefoglaló

A Haskell és a Clean lusta kiértékeléssel dolgozó, tisztán funkcionális nyelvek. A Haskell viszonylag elterjedt nyelv, így rengeteg alkalmazás és programkönyvtár készült hozzá. A nyelv egyfajta szabványt jelent a tisztán funkcionális nyelvek körében. Ezzel szemben a Clean dinamikusan fejlődő, folyamatosan újabb elemekkel bővülő nyelv. Előnye az is, hogy a teljesen funkcionális nyelvek körében szokatlanul hatékony fordító áll mögötte, amelyhez egy felhasználóbarát fejlesztőkörnyezet is tartozik. Ennek megfelelően felmerült az igény olyan rendszer készítésére, amely kihasználja mindkét nyelv előnyös tulajdonságait. Ehhez elengedhetetlenül szükséges a két nyelv viszgálata, összehasonlítása.

Noha a két nyelv elméleti háttere nagyon hasonló, nyelvi eszközeik gyakran igen különbözőek (ilyenek például a teljesen funkcionális nyelvekben nehezen megvalósítható, de mégis szükséges elemek, mint az I/O kezelése). Vannak olyan nyelvi elemek is, amelyek ugyan az egyik nyelvben megvannak, de hiányoznak a másikból. Az alábbi összehasonlítás célja a szabványos nyelvi elemek vizsgálata.

* Készült az OTKA T037742 sz. pályázat támogatásával.

A HASKELL ÉS A CLEAN NYELV ÖSSZEHASONLÍTÓ ELEMZÉSE *

Hegedűs Hajnalka, heha@inf.elte.hu
Horváth Zoltán, hz@inf.elte.hu
Zsók Viktória, zsv@inf.elte.hu
Eötvös Loránd Tudományegyetem
Általános Számítástudományi Tanszék

1. Bevezetés

A funkcionális szemléletmód lehetővé teszi olyan programok írását, amelyek helyessége a matematikában alkalmazott módszerekkel könnyen bizonyítható, hiszen a funkcionális programozásban használt függvényfogalom a matematikai függvényfogalomnak felel meg, nincs megsemmisítő értékadás, így az objektumok értékei sem változnak. Ennek azonban ára van: az értékadás hiánya miatt sokminden nagyon körülményesen írható le. Ilyen például az I/O kezelése, ami elengedhetetlen a felhasználói felület elkészítéséhez. Sok funkcionális nyelv tervezője úgy döntött, hogy az ilyen problémákat egyszerűen megkerüli, azaz megengedi nem funkcionális elemek használatát is. Ezen nyelveken (pl. Scheme, Lisp) írt programok helyessége az imperatív funkcionális elemek miatt már nehezebben bizonyítható. Más funkcionális nyelvek tervezésénél fontos szempont volt, hogy az ilyen problémák megoldására is valamilyen funkcionális módszert találjanak. Ezeket a nyelveket nevezzük tisztán funkcionális nyelveknek. Ilyen a Haskell és a Clean is.

A két nyelv hasonló elméleti háttere miatt hasonló szerkezetű lett, vannak azonban olyan részeik, amelyek meglehetősen különbözőek. Ilyen például az előbb említett problémák megoldása is, ahol két egymástól teljesen eltérő alapötletet használtak a két nyelv tervezői.

Ezen cikk a Haskell és a Clean nyelvi elemei között von párhuzamot. Az összehasonlítás alapja főként a két nyelv definíciója ([1], [2]) volt. Az egyes nyelvi elemek közötti különbségek megértéséhez hasznosnak bizonyult a nyelvek összehasonlítása, ilyen összehasonlítás találunk például a [3] és a [4]-ben, illetve az [5]-ben. Terjedelmi okokból csak a két nyelv közötti lényeges különbségeket részletezzük, a kisebb különbségekkel, illetve a mindkét nyelvben egyformán meglévő elemekkel csak említés szintjén foglalkozunk.

2. Általános tulajdonságok

2.1. *Elnevezési konvenciók*

Néhány apró különbségtől eltekintve a két nyelv azonsítók készítésére vonatkozó szabályai megegyeznek. A Haskell azonban lehetőséget ad minősített nevek használatára is. Ezek akkor hasznosak, ha más modulokban definiált azonosítókat vegyesen szeretnénk használni.

* Készült az OTKA T037742 sz. és a Grid 01548 sz. pályázat támogatásával.

Előfordulhat ugyanis, hogy ugyanaz az azonosító több modulban is szerepel, esetleg más funkcióval. Például tegyük fel, hogy az `Egesz` nevű modulban definiáltunk egy `gyok` nevű függvényt, ami egész számok négyzetgyökét számítja ki, míg az `Egyenlet` modulban egy ugyanilyen nevű függvény egy másodfokú egyenlet gyökeit határozza meg. Ha egy harmadik modulból mindkét `gyok` függvényt használni szeretnénk, minősített neveket kell alkalmaznunk, mivel a fordító különben nem tudná eldönteni, hogy éppen melyik `gyok` függvényre gondolunk. A minősített név a következő alakú: *minősítő*`.`*azonosító*; a *minősítő* általában a modul neve, de megadhatunk más nevet is a modul importálásakor. A fenti példában tehát a `gyok` függvényre `Egesz.gyok` vagy `Egyenlet.gyok` néven kell hivatkozni.

2.2. *Megjegyzések*

Mindkét nyelvben szúrhatunk megjegyzéseket a program szövegébe. A megjegyzéseket Cleanben a `/*` és a `*/`, Haskellben a `{-` és a `-}` karaktersorozatok határolják. Egysoros megjegyzések megadása is lehetséges: Haskellben a sor `--`, Cleanben a `//` utáni része megjegyzésnek számít.

2.3. *Modulszerkezet*

Míg Haskellben csak egyfajta modul létezik, Cleanben definíciós és implementációs modulokat hozhatunk létre. A definíciós modulhoz általában tartozik egy implementációs modul is, ketten együtt alkotnak egy egészet, a definíciós modul interfészt nyújt az implementáiós modul felhasználói számára, ők csak azt látják a modulból, amit a definíciós modulban megadtunk.

3. Előre definiált típusok

3.1. *Egyszerű típusok*

Egész számok megadására mindkét nyelven az `Int` típust használhatjuk. A decimális számokon kívül oktális és hexadecimális számokat is magadhatunk. Haskellben van még egy egész típus, az `Integer`, ez annyiban különbözik az előbbitől, hogy tetszőlegesen sok számjegyből álló számokat is leírhatunk vele. Lebegőpontos számokhoz Cleanben a `Real`, Haskellben a `Float` és a `Double` típusokat használhatjuk.

A karakterek tárolására a `Char` típus szolgál. Mindkét nyelvben megadhatunk karaktert az oktális, illetve a hexadecimális kódjával, használhatjuk a C-ben megszokott speciális karaktereket is. Haskellben a vezérlőkarakterek (az első 32 ASCII karakter) megadhatóak néhány karakteres rövidítésekkel: például az előre törlés kódja a `'DEL'`.

3.2. *Összetett típusok*

Mindkét nyelven megadhatóak rendezett n-esek és listák. Használatuk és az őket kezelő beépített függvények lényegében azonosak, listák tekintetében azonban van egy kis különbség. Listákat ugyanis nemcsak elemeik felsorolásával lehet megadni, hanem úgynevezett generátorok segítségével is el lehet készíteni. Például a
`[x\\x<-[1..10] | x<>8]` kifejezés Cleanben, illetve

`[x|x<-[1..10], x/=8]` Haskellben is a következő eredményt adja:
`[1,2,3,4,5,6,7,9,10].`

Az `x<-[1..10]` rész az úgynevezett generátor, ennek segítségével készítünk egy már meglévő listából új listát. Az `x<>8`, illetve az `x/=8` a szűrő, ennek segítségével válogatjuk ki, hogy az új listában a régi lista mely elemeire van szükségünk. Cleanben mindig egyértelműen megmondjuk, hogy egy szűrő melyik genrátorhoz tartozik: a szűrőt a generátortól a `|` jel választja el. Megadhatunk több generátor - szűrő párt is, ezeket vesszővel vagy `&` jellel választjuk el a többi pártól. Haskellben a generátorokat és a szűrőket vegyesen adjuk meg, a fordítóprogram feladata eldönteni, hogy mi mihez tartozik.

Cleanben tömböket is használhatunk, elemeik elérése hatékonyabb. Ezek fix elemszámúak, tárolásuk hatékonyabb. Tömbökhöz is készíthetünk a listákéhoz hasonló generátorokat. Ez az adatszerkezet Haskellből kimaradt, de egyes Haskell kiterjesztésekben megjelenik. Ez az oka annak is, hogy a két nyelv `String` típusa különbözik: Haskellben karakterlista, Cleanben karakterek tömbje. Emiatt a két nyelv stringműveleteinek szerkezete is erősen eltér egymástól.

4. Felhasználó által definiált típusok

4.1. *Szinonímák és algebrai típusok*

Mindkét nyelven lehetséges a típusszinonímák készítése, de létrehozhatunk algebrai típusokat is. Ezek megadása a két nyelvben nagyon hasonló, Haskellben azonban kétféle algebrai adattípus létezik. Ezek között az egyetlen különbség az, hogy a `newtype` hatékonyabb kódot eredményez.

4.2. *Rekordok*

Megadhatunk rekordokat is, azaz olyan összetett adattípust, amelynek a mezői címkézettek. Rendezett n-esekben az összetevők helye kötött, rekordban felcserélhető a mezők sorrendje, a címkéjük azonosítja őket.

Valójában csak a Clean biztosít rekord típust, a Haskell csak lehetőséget ad az algebrai adattípus mezőinek címkézésére. Ez bizonyos esetekben rugalmasabbnak bizonyul a Clean-féle megoldásnál:

```
data Szemely    = Ferfi {nev::[Char], kor::Int}
                | No {nev::[Char]}
```

Ezzel valójában egy variáns rekordot hoztunk létre. Ekkor anélkül hogy előre tudnánk, hogy az aktuális `Szemely` férfi vagy nő, egyetlen utasítással beállíthatjuk a nevét `"Gabi"`-ra:

```
case x of Ferfi _ kor -> Ferfi "Gabi" kor
          No _    -> No "Gabi"
```

Hasonló konstrukció létrehozása Cleanben nagyon körülményes. Vannak azonban egyéb előnyei a Clean rekordoknak:

```
::Lakcim =      {    nev::String,
                     orszag::String,
                     iranyitoszam::Int,
                     varos::String,
                     utca::String,
                     hazszam::Int
```

```
                }
```

A rekordnak nem kell feltétlenül minden mezőjét kitölteni. Például, ha az `ember` rekordot már definiáltuk valahol, akkor az illető szomszédjának rekordját nem kell újra elölről kitölteni, felhasználhatjuk hozzá a már meglévő rekordot, elég csak azokat a mezőket megadni, ahol valamilyen eltérés van:

```
szomszed =      {     ember &     nev="Gipsz Jakab",
                hazszam=hazszam+1
                }
```

Ekkor a `szomszed` rekord minden mezője meg fog egyezni az `ember` rekordéval, kivéve a `nev` mezőt, illetve a `hazszam` mezőt, ami az eredeti házszámnál eggyel nagyobb lesz.

5. Függvények és operátorok

Mindkét nyelvben hasonlóan adhatunk meg és használhatunk függvényeket, illetve operátorokat. A típusdefiníciókban lehetőségünk van típusváltozók használatára, ezekre megszorításokat is adhatunk. A függvényeket általában prefix módon használjuk, de mindkét nyelv lehetőséget ad a függvény infix használatára is, ha annak van értelme. Ezzel ellentétben az operátorokat általában infix módon használjuk. A Haskell megengedi az operátorok prefix módon való használatát, a Clean azonban csak speciális helyeken: az osztályok definiálásakor és ezek példányosításakor. Operátoroknál meg kell adni az operátor precedenciáját (azt, hogy az adott operátor milyen erősen köt), illetve azt, hogy merre zárójelezzük.

Mindkét nyelv lusta kiértékeléssel dolgozik, azonban, ha ez szüséges kényszeríthetjük a szigorú kiértékelés használatát. Többé-kevésbé a mintaillesztési szabályok is megegyeznek.

6. Osztályok

Osztályokat azonos módon adhatunk meg a két nyelven. A Haskell azonban lehetőséget ad arra, hogy az osztály műveleteihez alapértelmezett definícót adjunk, sőt ezek a definíciók kölcsönösen hivatkozhatnak is egymásra. A következő példa a Haskell standard könyvtárából származik:

```
class Eq a where
    (==), (/=)        :: a->a->Bool
        -- Minimal complete definition:
        --       (==) or (/=)

    x /= y            = not (x==y)
    x == y            = not (x/=y)
```

Ez azt jelenti, hogy példányosításkor elég az egyik függvény (az `==` vagy a `/=`) implementációját megadni, a másikat a fordító automatikusan generálja. Cleanben ez nem lehetséges, viszont itt alapértelmezett példányt adhatunk meg az osztályokhoz: ha a környezetből nem dönthető el egyértelműen, hogy éppen melyik példányt kell használni, akkor a fordító az alapértelmezett példányt használja.

Mindkét nyelv lehetőséget ad arra, hogy osztályok megadásakor használjuk már meglévő osztályok műveleteit is. Haskellben azonban ezt algebrai adattípusok definiálásakor is megtehetjük:

```
data Szin = Voros | Zold | Kek
             deriving(Eq, Ord)
```

Ekkor erre az adattípusra külön példányosítás nélkül használhatjuk az `Eq` és az `Ord` osztály műveleteit.

7. Információelrejtés

Az információelrejtés Cleanben a 2. fejezetben már említett definíciós modulok segítségével történik. Haskellben ugyanezt a célt az export listák szolgálják: itt sorolhatjuk fel, hogy milyen függvények, illetve adattípusok legyenek a modulon kívül láthatóak. Az export lista a modul fejlécében helyezkedik el. Például:

```
module Uj ( fv1 ) where
fv1 x = x
fv2 x = 3 + x
```

Mindenki, aki az `Uj` modult importálja, csak a `fv1`-et fogja látni, a `fv2`-t nem. Ha nincs export lista, minden, amit a modulban definiáltunk látható kívülről.

Mindkét nyelv lehetőséget ad arra is, hogy egy modulból ne importáljunk mindent, csak azt, amire valóban szükségünk van.

8. Felhasználói interakció

A bevezetőben szó volt arról, hogy bizonyos problémák megoldása nehezen illeszthető be a funkcionális világba. A problémák gyökere az, hogy egy korábbi értéket megsemmisítő értékadás használata nem megengedett, így az objektumok értéke konstans. Gyakran szükség lenne azonban arra, hogy egyes objektumok értékét megváltoztassuk. Tipikus példája ennek az, amikor a program a felhasználótól kér be adatokat és ezekkel dolgozik. Egy másik gond az, hogy a kifejezések kiértékelési sorrendje nem kötött. Ez bizonyos esetekben (például többszöri kiiratás a képernyőre) hibás működést eredményezhet (a sorok nem olyan sorrendben kerülnek a képernyőre, ahogy ki akartuk iratni, hanem abban a sorrendben, ahogy a kiiratók függvények kiértékelődtek). Ezekre a problémákra kínálnak megoldást a következő módszerek.

8.1. *Clean: unique típus*

A Clean tervezői bizonyos korlátok között megengedik a destruktív értékadást: a speciális, úgynevezett unique objektumoknak újra értéket adhatunk, de csakis akkor, ha egyetlen helyen hivatkozunk rá. Ekkor ezen a helyen biztonságosan megváltoztathatjuk az objektum értékét, hiszen senki más nem használja a korábbi értékét többé. Azt, hogy nincs-e több hivatkozás az objektumra, a fordító ellenőrzi. Minden értékadáskor új nevet is kellene adnunk az objektumnak, de mivel több hivatkozás már nincs arra a névre, azaz az objektumra, ami a névhez tartozik, már nincs szükség, a nevet is újra felhasználhatjuk, innentől a név már az új objektumra mutat. Ezzel a módszerrel az imperatív stílushoz hasonlóan programozhatunk (amikor ez feltétlenül szükséges) anélkül, hogy a tisztán funkcionális programozás kereteit átlépnénk. Ezek használatához szükség van egyfajta szekvencializálásra is, hiszen különben nem lehetne tudni, hogy az újra felhasznált nevek melyik értékadás eredményére mutatnak.

8.2. *Haskell: monádok*

A monádok szintén egyfajta szekvencializációt valósítanak meg: a monádok egymás után fűzhetők, a füzérben a következő monád mindig meg kell, hogy várja az őt megelőző monád eredményét. A monád mindig egy osztály, aminek Haskellben négy művelete van. A `return`

művelet egyszerűen visszaadja az argumentumként kapott értéket és egy `fail` művelet egy hibaüzenet visszaadására alkalmas. Ezek segítségével monádokat konstruálhatunk egyszerű típusokból. Készíthetünk már meglévő monádokból is újakat a `>>=` és a `>>` művelet segítségével, ezek monádok összekapcsolására szolgálnak, a `>>=` fölhasználja az első monád által visszaadott értéket a `>>` pedig eldobja. A monádok produkálhatnak mellékhatásokat is, ellentétben a normál függvényekkel.

9. Irodalomjegyzék

[1] Rinus Plasmeijer and Marko van Eekelen. *Language Report for Concurrent Clean, version 1.3*. Technical Report CSI-R9816, Computing Science Institute, Faculty of Mathematics and Informatics, University of Nijmegen, 1988.

[2] Simon Peyton Jones, John Hughes, et al. *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language,* February 1999.

[3] Peter Achten. *Clean for Haskellers and vice versa*, January 2001. Draft.

[4] Simon Thompson. *Haskell, The Craft of Functional Programming*. Addison-Wesley, 1999.

[5] Rinus Plasmeijer et al. *Functional Programming in Clean*, July 1999. Draft.

[6] Pascal R. Serrarens and Marinus J. Plasmeijer. *Explicit Message Passing for Concurrent Clean.* In *Implementation of Functional Languages*, pages 229-245, 1998.

[7] P. Trinder, K. Hammond, H. Loidl, S. Jones, et al. *Strategy*. Journal of Functional Programming, 1998. (???)

[8] K. Hammond and G. Michaelson. *Research Directions in Functional Programming*, 1999.