FUNKCIONÁLIS PROGRAMNYELVEK IMPLEMENTÁCIÓJA

Csörnyei Zoltán, *csz@maxi.elte.hu*
Nagy Sára, *saci@ludens.elte.hu*
Eötvös Loránd Tudományegyetem
Általános Számítástudományi Tanszék

Abstract

Functional languages have become the focus of much active research in recent years, but their acceptance has been delayed by the inefficiency of their available implementations when compared with more conventional languages. This situation has changed recently with the advent of rather fast implementations of new fuctional languages such as ML, Miranda, Haskell.

This article is about implementing functional programming languages. It describes how to translate a high-level functional language into an intermediate language, called the lambda calculus. After it shows a simple implementation of the lambda calculus using graph reduction.

1. Bevezetés

A programozók a programok többségét probléma-orientált magas szintû nyelveken írják. Ezek a programnyelvek három fõ csoportra oszthatók:

- *imperatív nyelvek*, amelyeknek alapja Neumann-modell struktúrája,

- *funkcionális nyelvek*, amelyek legfontosabb közös tulajdonságai a következõk:
  - nincs különbség az utasítás és a kifejezés között,
  - a nevek nem memóriahelyet azonosítanak, hanem a kifejezések neveit,
  - egy függvény lehet egy másik függvény paramétere és értéke is.

Egy funkcionális nyelven írt program végrehajtásának elve a *redukció*, amely a program kifejezéseit egyszerûbb kifejezésekkel helyettesíti, és a program végrehajtása akkor ér véget, ha a kifejezés tovább már nem egyszerûsíthetõ.

- *logikai programnyelvek*, amelyeknek alapja a predikátumkalkulus. A végrehajtási mechanizmusuk alapja a *rezolúció*, amelyet logikai tételek bizonyítására fejlesztettek ki.

További nyelvcsoportot alkotnak a hardware leíró nyelvek, az operációs rendszerek parancs nyelvei és azok a nyelvek, amelyeket szöveg és grafika leírására készítettek. Az *objektum orientált programnyelvek* nem tekinthetõk külön nyelvosztálynak, mivel mindegyik nyelvcsoportban találhatók olyan programnyelvek, amelyek az objektum orientált programozást támogatják.

2. A funkcionális nyelvek tulajdonságai

Az imperatív programnyelvek tulajdonságait elsõsorban a Neumann számítógépmodell architektúrája határozza meg, ehhez a modellhez az alapot pedig a Turing gép mûködésének matematikai modellje adja. Ennek az architektúrának nagy elõnye, hogy rendkívül egyszerû: lényegében két fajta információt használ: a gépi kódot reprezentáló *utasítást*, amelyet a gép központi egysége interpretál, és amely szabályozza a

számítógépben lezajló számítási folyamatot, és az *adatot*, amellyel az utasítások manipulálnak. Ez a modell könnyen realizálható olcsó hardware berendezésekkel.

Míg kezdetben a "magasszintû" programnyelveket egyszerûen csak arra a célra fejlesz tették ki, hogy velük a gépikódnál kényelmesebben írjanak le például matematikai kifejezéseket, addig késõbb a magasszintû nyelvek tervezésekor a cél az lett, hogy a nyelv alkalmas legyen a program szemantikájának mindenki által történõ megértésére és a program helyességének bebizonyítására. Azonban a kifejezések alacsonyabb szintre fordítása a fordító programok központi kérdése maradt. A funkcionális nyelvek pedig az egész programot egy nagy kiértékelendõ kifejezésnek tekintik.

2.1 A lambda-kalkulus

A funkcionális programnyelvek tehát olyan leíró nyelveknek tekinthetõk, amelyekkel értékek, függvények és ezek kapcsolata írható le. A *lambda-kalkulus*, amelyet már korábban kidolgoztak, kiválóan alkalmazható kiszámítható függvények formális leírására, ezért az elsõ funkcionális nyelvek alapja a lambda-kalkulus lett. Mivel minden lambda-kalkulussal definiálható függvény Turing-kiszámítható is, minden kiszámítható függvény leírható lambda-kalkulussal. A jól ismert LISP programnyelv is ezen alapul. Azonban hamar kiderült, hogy a lambda-kalkulus a típus-fogalommal nehezen bõvíthetõ, és az, hogy alkalmazása eléggé nehézkes.

Kifejlesztették a *kibõvített lambda-kalkulust*, amely már tartalmaz az adatokra vonatkozóan típus-, operátor- és literál-fogalmat, bár a programok fordítása során a kibõvített lambda-kalkulus programjait elõször az eredeti lambda-kalkulusra alakítják át, és ezt implementálják. A kibõvített lambda-kalkuluson alapuló nyelvek például az ML, Miranda és Haskell.

2.2 A függvény fogalma, polimorfikus függvények

Egy funkcionális nyelven írt program függvénydefiníciókból és egy kiszámítható kezdõ kifejezésbõl áll. Most a leírásban a Miranda nyelv függvényfogalmát használjuk.

Egy *függvénydefiníció* egy vagy több egyenlet, az egyenlet baloldalán a *függvény neve* és a *függvény argumentumai (formális paraméterei)* állnak, az egyenlet jobboldala pedig egy *kifejezés*. A formális paramétereket *változóknak* nevezzük. A változó hatásköre csak arra az egyenletre terjed ki, amelyikben paraméterként definiálták, míg a függvénynév hatásköre az egész program. A kifejezés lehet vagy egy érték, vagy egy formális paraméter, vagy egy *függvény applikáció*. A függvény applikációban egy függvényt egy kifejezésre, mint aktuális paraméterére alkalmazunk, és *f a* jelöli az *f* függvénynek az *a* kifejezésre történõ applikációját.

Egy függvénydefiníciót egy típusdefiníció elõzhet meg, amelyikkel a függvényben szereplõ kifejezések típusát lehet megadni. Egy típus lehet *elõredefiniált típus*, például *szám, karakter, boolean*, lehet *típuskonstruktorokkal* már meglévõ típusokból létrehozott új *komplex típus*, vagy *generikus típusváltozókat* tartalmazó komplex típus. Ha egy függvény típusdefiníciójában generikus típusváltozó van, akkor a függvényt *polimorfikus típusú függvénynek* nevezzük.

A kezdõ kifejezés az a kifejezés, amelynek meg kell meghatározni az értékét. A funkcionális program végrehajtása a kezdõ kifejezés értékének kiszámítását jelenti, a kiszámítás pedig a program függvénydefinícióit használja. A kiszámítás lépései a *redukciók*, a redukciós lépésekkel, a redukálási folyamat terminálásával a késõbbiekben foglakozunk.

2.3 Egyszerû funkcionális nyelvek

Míg az imperatív nyelvekben a névvel azonosított változók adatok tárolására szolgálnak, az adatokat pedig utasítások végrehajtásával változtatják meg, ez a tulajdonság funkcionális programnyelvekben *mellékhatásként* jelentkezik. Az imperatív programnyelvekben egy program mûködéséhez az utasítássorozatok dinamikus viselkedését kell ismerni, és elõfordulhat, hogy egy program különbözõ programfuttatásokra különbözõ eredményt ad, éppen a mellékhatások következményeként.

Mivel az imperatív program lényegében utasítások sorozata, a végrehajtás is *szekvenciális*, nehéz azokat a programrészeket detektálni, amelyek párhuzamosan végrehajthatók. A végrehajtás sebességét növelni a legegyszerûbb módszer az utasítások parallel végrehajtása, éppen ezért az utasítások szekvenciájának megadása az imperatív nyelvek nagy hátrányának tekinthetõ.

Azokat a programnyelveket, amelyekben az imperatív programnyelvre jellemzõ fenti tulajdonságok egyáltalán nem jelentkeznek, *egyszerû (purely) funkcionális nyelveknek* nevezzük. Ilyen nyelv például a Miranda, Haskell, nem egyszerû funkcionális nyelv például az ML, Lisp, Scheme.

2.4 Szigorú funkcionális nyelvek

A funkcionális nyelvekben két alapvetõ módszer alakult ki a függvények paramétereinek átadására:

- a függvény argumentuma a függvény hívásakor mindig kiértékelõdik, az ilyen nyelveket hívjuk *szigorú (strict) funkcionális nyelveknek.*
- függvény hívásakor a függvény argumentuma csak akkor értékelõdik ki, ha az argumentumra ténylegesen szükség van. Az ilyen programnyelvek a nem-szigorú funkcionális programnyelvek. A paraméterátadásnak ezt a módját *lusta (lazy) kiértékelésnek* nevezzük.

A két módszer használatában a lényeges különbség az, hogy a nem-szigorú nyelvek programjai akkor is szabályosan befejezõdhetnek, ha bennük egy függvény argumentumának kiszámítása végtelen ciklusba kerül, vagy hibajelzést ad. Szigorú funkcionális nyelv például az ML, Lisp, Scheme, nem-szigorú nyelv, azaz lusta paraméterkiértékelés van például a Miranda és a Haskell programnyelvekben. A Hope nyelvben pedig mindkét paraméterátadási módszer alkalmazására van lehetõség.

2.5 Magasabb rendû függvények

Ha egy függvénynek sem az argumentuma, sem az értéke nem lehet függvény, akkor a függvényt *elsõrendû függvénynek*, egyébként pedig *magasabb rendû függvénynek* nevezzük. A függvény applikációk balasszociatívak, így az $f\ a\ b...\ c$ applikáció az $(...(\ (\ f\ a\ )\ b\ )\ ...\ c)$ kiszámításnak felel meg. Elég egy-argumentumú függvényekkel foglalkozni, mivel többargumentumú függvényre az úgynevezett *curryzás* alkalmazható, azaz egy többargumentumú függvény felírható egyváltozós magasabb rendû függvények applikációjaként.

Egy függvény csak akkor hajtható végre, ha minden argumentuma rendelkezésre áll. A curryzás jelentõsége abban van, hogy egy n-változós függvény egy k-változós ($1 \leq k < n$) *paraméterezett magasabb rendû függvény* és egy n-k-változós függvény applikációjának tekinthetõ, azaz az elsõ paraméterezett függvény által nem kezelt paraméterek a másik függvény paraméterei lesznek. A curryzás felhasználásával, új argumentumok bevezetésével egy függvény általánosítása egyszerû módon elvégezhetõ.

2.6 Hivatkozási átlátszóság

A funkcionális nyelvek közel állnak a matematika nyelvéhez, a nevek használata konzisztens, a változók nem változnak, és egy nem feltétlenül ismert konstans értéket jelölnek. Ugyanaz a kifejezés, a mellékhatások hiánya miatt, mindig, a funkcionális program minden pontján, ugyanazt az értéket jelöli. Ez utóbbi tulajdonságot hívjuk *hivatkozási átlátszóságnak (referential transparency).*

Az $x = x + 1$ az imperatív nyelvekben azt jelenti, hogy az x értékét inkrementálni kell. Funkcionális nyelvekben viszont ez azt jelenti, hogy az x -t minden elõfordulási helyén helyettesíteni kell $x + 1$ -gyel.

Ez a tulajdonság lehetõvé teszi, hogy a matematikába n használt szimbólum-helyettesítést és a teljes indukciót a programnyelvben is használni lehet, és így egy függvény definíciója a funkcionális nyelven leírva azonos a definíció matematikai leírásával, vagy a két leírás között az eltérés minimális. Ha a matematikai leírás helyessége bizonyítható, akkor, kihasználva a hivatkozási átlátszóság tulajdonságot, a programnyelvi definíció helyessége ugyanazzal a módszerrel bizonyítható.

3. A funkcionális fordítóprogramok szerkezete

A fordítás menete a funkcionális nyelveknél jelentõsen eltér a nem-funkcionális nyelvek fordítási algoritmusától. Elsõ lépésben a funkcionális programot egy úgynevezett kiterjesztett lambda-kalkulusra fordítják, majd az így kapott programot transzformálják tovább a szigorú értelemben vett lambda-kalkulusra.

A lambda-kalkulusnak igen egyszerû a szintaxisa és a szemantikája is, ezért könnyû implementálni. Mivel a lambda-kalkulusba transzformált program lényegében egy kiszámítandó kifejezésnek felel meg, ezért természetes módon adódik, hogy ezt a kifejezést egy kifejezésfával reprezentálják. Így a program végrehajtása lényegében megfelel a kifejezésfa kiértékelésének. Ez pedig azt jelnti, hogy a fa egyes alkalmasan kiválasztott részfáit redukáljuk a nekik megfelelõ értékekre. Nincs kötött sorrendje az egyes redukciós lépéseknek, és a kiértékelés akkor ér véget, ha már nincs alkalmas részfa, amelyet redukálni lehetne.

A fent vázolt folyamatban az egyik fõ problémát a forrásprogramnak megfelelõ fa, illetve bizonyos hasonló részfák miatt általánosabban gráf felépítése jelenti. A másik probléma a redukálható részgráfok kiválasztása, illetve a redukció végrehajtása.

Mivel a redukciónak nincs kötött sorrendje, ezért elvi problémát okoz, hogy egy program futtatásának eredménye minden esetben ugyanaz lesz-e. Erre a késõbbiekben még kitérünk.

3.1 A lambda-kalkulus szintaxisa és szemantikája

Ahogy azt már korábban említettük egy funkcionális program egy lambda-kalkulus beli kifejezés kiértékelésének felel meg, ezért röviden ismertetjük a lambda-kalkulussal kapcsolatos legfontosabb fogalmakat.

Egy lambda-kalkulusban felírt kifejezés a következõk valamelyike lehet:

- konstans; (A lehetséges konstansokat elõre beépítetteknek tekintjük.)
- változó; (A változók tulajdonképpen csak szimbólumnevek.)
- két egymás után írt lambda-kifejezés; (Ez felel meg a függvény applikációnak. Ez azt jelenti, hogy az elsõ kifejezést egy függvénynek a másodikat ezen függvény aktuális argumentumának tekintve alkalmazzuk a függvényt erre az argumentumra. A currying miatt elég csak egy-argumentumú függvényekkel foglalkoznunk.)
- lambda-absztrakció. (A *lambda-absztrakció* egy név nélküli függvénydefiníciónak felel meg, szintaxisa: λ<változó>.<törzs> , ahol a törzs maga is egy lambda-kifejezés.)

Bár szigorú értelemben a lambda-kalkulusban nincsenek beépített függvények, azonban a gyakorlatban meg szokták engedni például az aritmatikai mûveletek használatát, amelyeket prefix formában kell beírni a kifejezésekbe.

A lambda-kifejezéseket a következõ négy konverziós szabály seg ítségével alakíthatjuk át velük szemantikusan ekvivalens lambda-kifejezésekké:
- α konverzió: változók átnevezése,
- β konverzió: függvény applikáció (lásd még alább),
- η konverzió: redundás lambda-absztrakciók eliminálása,
- δ konverzió: beépített függvények alkalmazása.

A lambda-absztrakció egy név nélküli függvényt jelöl, a szemantika pontosítása miatt még meg kell adnunk, hogy hogyan kell alkalmaznunk ezt a függvényt az argumentumára. A lambda-absztrakció törzsében a formális paraméternek tekintendõ változó szabad elõfordulásait kell makró szerûen helyettesíteni az aktuális argumentummal. Így a törzs egy példányát kapjuk.

Ha a lambda-absztraktciót alkalmazzuk az argumentumára, akkor β-*redukcióról,* ha pedig ezt visszafelé alkalmazzuk, akkor β-*absztrakcióról* beszélünk. A kettõt együtt nevezzük β-*konverziónak.*

A lambda-kifejezésben azok a részkifejezések az úgynevezett *redexek* , amelyek egyszerûbb alakra redukálhatóak. Ha nincs egy kifejezésben redex, akkor normál formában van. Nem minden kifejezésnek van normál formája, de ha létezik, akkor az egyértelmû és van olyan redukálási sorozat, amellyel ez a normál forma elõállítható.

3.2 A funkcionális program transzformációja lambda-kalkulusba

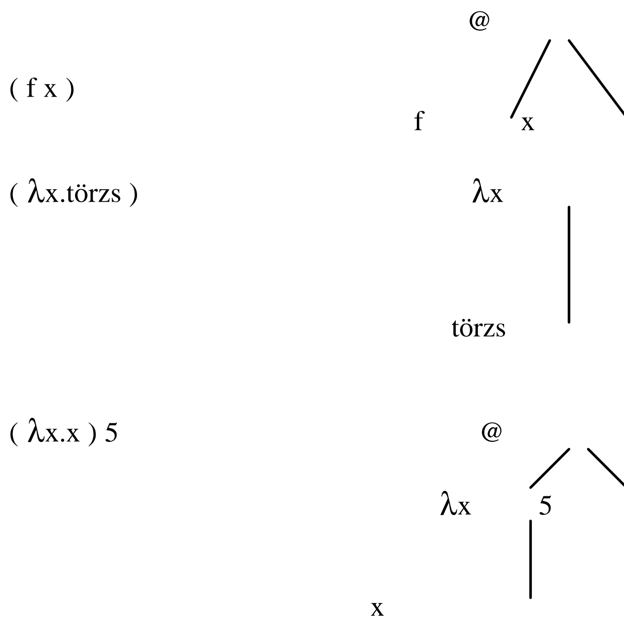
Egy funkcionális program két fõ részbõl áll: a függvények definícióiból, illetve egy kezdõ kifejezésbõl. Ha sikerül megadnunk, hogy miként kell egy függvény definíciót, illetve egy kifejezést lambda-kalkulusba transzformálni, akkor lényegében megadtuk a fordítóprogram magját. Ezenkívül még foglalkoznunk kell a *típus fogalom fordításával* , illetve a funkcionális programokban alkalmazott *mintaillesztéssel.* Ezek a problémák is megoldhatók alkalmas függvények úgynevezett *kombinátorok* segítségével.

A pontos transzformáció az [1]-ben megtalálható. Most csak felsorolunk néhány további problémát, amely a transzformáció során merül fel.
- Árfedés lehet a minták közt.
- Egymásba ágyazottak lehetnek a minták.
- Konstansokat kell illeszteni.
- Többargumentumosak is lehetnek a függvények.
- Esetszétválasztással is le het függvényt definiálni.
- Nem biztos, hogy minden eset definiálva van.
- Változó nevek ütközhetnek.

3.2 A lambda-kalkulus implementciója gráfredukcióval

Mint már említettük a lambda-kalkulusban megadott kifejezések gráffal ábrázolhatóak. Tekintsük az alábbi alapeseteket!

<u>lambda-kifejezés</u> <u>gráfos ábrázolás</u>



A gráfredukcióban a probléma a soronkövetkezõ redex kiválasztása. Ha létezik normál forma, akkor a legbaloldalabbi legkülsõ redexek redukálásával teljesen kiértékelhetjük a funkcionális program kezdõ kifejezésének megfelelõ gráfot, amely megfelel a program végrehajtásának.

3.4 Term újraíró rendszerek

Egy alternatív modell a *Term újraíró rendszer (TRS)* [3], amelyben a funkcionális program függvényei újraírási szabályoknak, a kezdõ kifejezés pedig egy redukálható termnek felel meg.

A TRS azonban általánosabb modell a lambda-kalkulusnál, például a nem-determinisztikusságot is le lehet írni vele. Ugyanakkor, mivel a TRS minden szabálya azonos prioritású, általában nincs egy elõre definiált explicit stratégia, amely megadná, hogy a redukciókat, azaz a TRS szabályait milyen sorrendben kell végrehajtani. A lambda-kalkulussal ellentétben, csak a TRS-k egy részhalmazára adható meg normalizáló stratégia, és bizonyítható, hogy csak az egyértelmû és nem-komparáló, azaz az ortogonális TRS *konfluens*, vagyis ha létezik normál forma, akkor az egyértelmûen meghatározható. Míg a lambda-kalkulusban a legbaloldalibb-legkülsõ redukciók sorozata vezetett el a normál formához, ortogonális TRS-ben a párhuzamos-legkülsõ redukciókkal lehet biztosítani a normál forma elérését.

A TRS-k jól használhatók funkcionális nyelvek implementációjára is, alkalmazásukkor egy lényeges probléma merül fel: nem biztosítható, hogy az azonos kifejezések kiszámítása csak egyszer történjen meg.

3.5 Gráf újraíró rendszerek

A TRS-ket általánosítva kapjuk a *Gráf újraíró rendszereket (GRS)* , amelyek konstans szimbólumokon és gráfpontok azonosító nevein értelmezett újraírási szabályokból állnak. A funkcionális program kezdõ kifejezésének egy speciális kezdõ gráf felel meg, amelynek gyökérpontja a redukálások alatt mindig a gráf gyökérpontja marad, a funkcionális program függvényei pedig a gráf újraírási szabályoknak felelnek meg.

A GRS-k öröklik a TRS-ekre vonatkozó redukálási tulajdonságokat, és belátható, hogy a GRS-k konfluens tulajdonságát az "önmagukat tartalmazó" redukálható kifejezések okozhatják. Így itt sem adható

meg tetszõleges GRS-re normalizáló stratégia, de speciális esetekben a párhuzamos-legkülsõ kifejezések redukálásai a gráf normál formájához vezetnek. A leglényegesebb különbség a két újraíró rendszer között az, hogy a GRS-kben az azonos kifejezések kiszámítása csak egyszer történik meg. Ezért alkalmasabbak a GRS-k a funkcionális nyelvek implementációjára.

Az implementáció általános módszere az, hogy a funkcionális programot TRS-re alakítják át, ebbõl a *lifting* mûvelettel GRS-t készítenek és gráfredukcióval a gráfot normál formára hozzák. A funkcionális nyelvek mintaillesztésében használt prioritás alkalmazható a gráf újraírási szabályainak alkalmazásakor is, az ilyen redukciós módszereket alkalmazó rendszereket *funkcionális GRS-knek (FGRS)* nevezik. Funkcionális programnyelvek implementációiban FGRS-ket használva a logikai és imperatív nyelvek nem-funkcionális tulajdonságai is, mint például a mellékhatás, könnyen megvalósíthatók.

4. Oktatási tapasztalatok

Az ELTE Általános Számítástudományi Tanszéke már évek óta oktat a programtervezõ matematikusoknak funkcionális nyelveket a program nyelvek- (Miranda), illetve mesterséges intelligencia sávokban (LISP). Azonban ezek fordítási kérdései még csak a doktoranduszi iskolában szerepeltek.

Mivel napjainkban, fõleg a párhuzamos programok írásának elõtérbe kerülése óta, egyre több figyelem fordítódik a funkcionális programnyelvekre, ezért tervezzük , hogy oktatásunkban nagyobb teret szentelünk a funkcionális nyelveknek. Megismertetnénk a hallgatókat további újfajta funkcionális programnyelvekkel mint az ML, Haskell, illetve nagy hangsúlyt fektetnénk ezen programnyelvek implementációjára is.

Irodalomjegyzék

[1] Simon L. Peyton Jones: The Implementation of Functional Programming Languages, Prentica Hall, 1987.

[2] Simon L. Peyton Jones - David R. Lester: Implementing Functional Languages, Prentica Hall, 1992.

[3] Rinus Plasmeijer - Marko van Eekelen:Functional Programming and Parallel Graph Rewriting, Addison-Wesley, 1993.

[4] Reinhard Wilhelm - Dieter Mauer: Compiler Design, Addison-Wesley, 1995.