

# Referenciák tisztán funkcionális nyelvekben doktori értekezés tézisei\*

Diviánszky Péter

ELTE IK Informatika Doktori Iskola  
Az informatika alapjai és módszertana oktatási program  
vezető: Dr. Benczúr András

Programozási Nyelvek és Fordítóprogramok Tanszék

témavezető: Dr. Horváth Zoltán

2012.

---

\*The research was supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T037742, by GVOP-3.2.2.-2004-07-0005/3.0 ELTE IKKK and OMAA-ÖAU 66öu2, by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003)

# Bevezetés

## Hivatkozások behelyettesíthetősége

A funkcionális nyelvek sikerében jelentős szerepet játszik a hivatkozások behelyettesíthetősége.

A *hivatkozások behelyettesíthetősége* (angolul referential transparency) egy adott környezet, egy kifejezés és egy változó esetén azt jelenti, hogy a kifejezésben a változó kicserélhető a definíciójára anélkül, hogy a kifejezés jelentése megváltozna:

$$e \longrightarrow e[v := a] \quad \text{ha} \quad \{ \dots; v = a; \dots \}$$

Tisztán funkcionális programozási nyelvnek nevezhető az olyan programozási nyelv, amelyben mindig teljesül a hivatkozások behelyettesíthetősége, vagy egyszerű feltétel adható a hivatkozások behelyettesíthetőségére. A definíció alapján tisztán funkcionális nyelvek például a Haskell, a Clean<sup>1</sup> és a Miranda.

A hivatkozások behelyettesíthetőségének definíciójában a *kifejezés jelentése* fogalom külön magyarázatra szorul. Általában egy kifejezés jelentése alatt a kifejezés funkcionális viselkedését értjük. A funkcionális viselkedésbe nem tartozik bele a futási idő vagy a memóriahasználat, de beletartozik például az, hogy a program adott bemenetre milyen kimenetet ad.

A hivatkozások behelyettesíthetőségéhez szorosan kapcsolódó fogalmak a hivatkozási helyfüggetlenség, a függvénydefiníciók behelyettesíthetősége ( $\beta$ -redukció), az egyenlők behelyettesíthetősége (Leibniz szabály), a perzisztencia, a kiterjeszthetőség (extensionality), a determinisztikusság és a mellékhatásmentesség, amely fogalmak egymáshoz való viszonyát a [14] cikk tárgyalja.

A hivatkozások behelyettesíthetőségének előnyei:

- egyszerűbb programhelyesség bizonyítás, például egyenlőségi érvelés használata,
- kódoptimalizálási lehetőségek, parciális kiértékelés fordítás közben,
- nagyobb szabadság a kiértékelési stratégia megválasztásában, párhuzamos kiértékelés lehetősége.

---

<sup>1</sup>A Cleanben a hivatkozások behelyettesíthetőségét az egyszeres hivatkozásra vonatkozó típusozási szabályok korlátozzák.

## Kihívások a hivatkozások behelyettesíthetősége esetén

Különböző kihívások jelentkeznek a hivatkozások behelyettesíthetőségének betartásakor, ezek egyike a referenciák használatának megoldása, amivel a doktori értekezésemben foglalkozom.

## Megoldások összefoglalása

Referenciák használatára tisztán funkcionális nyelvekben a következő megoldások lehetségesek:

### 1. Tisztán funkcionális adatszerkezetek használata

**Rövid leírás** A véges leképezés adatszerkezettel[1] jól modellezhetők a referenciák.

**Előnyök** A programozónak nagy szabadsága van, mivel a tisztán funkcionális adatszerkezetek perzisztensek. Lehetőség van például az összes referencia értékének egy adott időpontban való megjegyzésére és későbbi visszaállítására.

**Hátrányok** A perzisztencia lehetetlenné teszi az adatszerkezetek destruktív módosítását, ami kizár sok hatékony implementációs lehetőséget. Például a (perzisztens) véges leképezés elemi műveleteinek hatékonysága csökken az elemszám növekedésével.

**Felhasználás az értekezésben** Az első tézisben definiáltam egy módosított véges leképezés adatszerkezet, ami hatékony bizonyos természetes feltételek mellett, amik teljesülnek ha referenciák modellezésére használjuk a leképezéseket.

### 2. A műveletek mellékhatásának elkülönítése

Ha elkülönítjük a mellékhatásos kifejezéseket a mellékhatásmentesektől, akkor biztonságosan ki tudjuk használni a hivatkozások behelyettesítésének az előnyeit legalább a mellékhatásmentes kifejezések esetén.

A mellékhatások elkülönítésének módjai:

#### (a) programhelyesség bizonyítással

**Rövid leírás** A programhelyesség bizonyítás során minden információ rendelkezésre áll ahhoz, hogy a kifejezések mellékhatásait vizsgáljuk. A referencia műveletek mellékhatásának elkülönítésére például a szétválasztási-logika[13] (angolul separation logic) használható.

**Előnyök** A programkód egyszerű marad, mert mellékhatásokra vonatkozó információ a programra vonatkozó állításokban és bizonyításokban van.

**Hátrányok** A hivatkozások behelyettesíthetősége komplex feltételhez kötött, ami többnyire a programkódtól elkülönítve jelenik meg, ezért a mellékhatások elkülönítése programhelyesség bizonyítással nem tekinthető a tisztán funkcionális programozás eszközének.

A helyességbizonyítás során kapott információ nehezen vehető igénybe a program fordításakor, például optimalizáció céljából.

Ha a szétválasztási logikát a típusrendszerrel fejezzük ki, akkor a mellékhatás-típusozással kapcsolatos hátrányok jelentkeznek.

**Felhasználás az értekezésben** Ezzel a módszerrel nem foglalkozok az értekezésben, mert a mellékhatások elkülönítését programhelyesség bizonyítással nem tekintem a tisztán funkcionális programozás eszközének.

(b) mellékhatás-típusozással

**Rövid leírás** A *mellékhatás-típusozás*[17][12] (angolul effect-typing) a kifejezések mellékhatásait a típusrendszer segítségével követi.

**Előnyök** A programkód egyszerű marad, mert mellékhatásokra vonatkozó információ a típusban van, ami esetleg automatikusan ki is következtethető.

**Hátrányok** A hivatkozások behelyettesíthetősége komplex feltételhez kötött, ami csökkenti a hivatkozások behelyettesíthetőségéből származó előnyöket, például nehezebb egyenlőségi érvelést végezni.

A mellékhatás-típusozási rendszerek meglehetősen komplexek, a nem mohó kiértékelési stratégiák különösen nehezen kezelhetők.

**Felhasználás az értekezésben** Bizonyos mellékhatások esetében egyszerű feltétel adható a hivatkozások behelyettesítéséhez, például referenciák egyenlőségvizsgálata esetén. Ekkor a mellékhatás-típusozás vagy hasonló módszer egyszerűen használható marad; ezt mutatom be a harmadik tézisben.

(c) egyediségi típusozással

**Rövid leírás** Az *egyediségi típusozás*[2] a mellékhatás-típusozás egy egyszerűsített változatának tekinthető, ahol az egyetlen lehetséges mellékhatás az, hogy egy értéket egy művelet során már felhasználtunk, és ezzel a mellékhatással fejezzük ki minden más mellékhatást.

**Előnyök** A kód olvasásakor – például egyenlőségi érvelés esetén – a hivatkozások behelyettesíthetősége feltétel nélkül alkalmazható.

A programkód egyszerű marad, mert a mellékhatásokra vonatkozó információ a típusban van, ami automatikusan kikövetkeztethető.

**Hátrányok** A hivatkozások behelyettesíthetősége komplex feltételhez kötött programírás – például programtranszformáció – esetén.

A típusrendszert alkalmassá kell tenni az egyediségi típusozásra, ami típusattribútumokkal történhet.

**Felhasználás az értekezésben** A második tézisben az egyediségi típusozás egy egyszerűsített változatát használom a referencia műveletek mellékhatásának elkülönítésére.

(d) monádokkal

**Rövid leírás** A monádok a mellékhatást a programkódban explicit módon jelzik, és megadják, hogy a részkifejezések mellékhatásából hogyan számítható ki az egész kifejezés mellékhatása [11] [16].

**Előnyök** A hivatkozások behelyettesíthetősége feltétel nélkül alkalmazható.

A monádok nem igényelnek speciális típusozást, a monadikus műveleteknek típusa típuskonstruktorokkal és univerzális kvantálással leírható.

**Hátrányok** A mellékhatásokat a programkódban is jelezni kell, és előfordul hogy ugyanannak az algoritmusnak meg kell adni egy mellékhatásokat kezelő és nem kezelő változatát is.

A hivatkozások behelyettesíthetősége teljesül ugyan, de a monádok használata indokolatlanul rögzítheti a kiértékelési sorrendet, így a helyességbizonyítás lépései kötöttebbek lesznek, valamint fordítóprogramnak kevesebb tere marad az optimalizációra, ami érdekes módon pont a hivatkozások be nem helyettesíthetőségének egy tünete.

Egymástól független mellékhatások kezelése monádokkal nehézkes.

**Felhasználás az értekezésben** A referenciák monádokkal való megvalósítása már kidolgozott, ezeket bemutatom az értekezésben.

Mind a négy módszernek<sup>2</sup> megvan az előnye és a hátránya is, és az alkalmazáshoz igazodva érdemes a leginkább megfelelő módszert kiválasztani.

---

<sup>2</sup>A mellékhatások elkülönítését programhelyesség bizonyítással nem tekinthető a tisztán funkcionális programozás eszközének, így négy módszer marad.

# Tézisek

A doktori értekezésemben megmutatom, hogy a referenciák a monadikus modell mellett milyen más konstrukciókkal válthatók ki tisztán funkcionális nyelvekben.

## Első tézis

### Referenciák hatékonyan kiválthatók véges leképezésekkel

*Definiáltam egy módosított véges leképezés adatszerkezetet, amely hatékonysága miatt alkalmas referenciák kiváltására.*

[6], [7] cikkek, [8] kapcsolódó cikk.

A véges leképezés adatszerkezet egy olyan asszociációs listának felel meg, amelyben a kulcs-érték párok sorrendje nem definiált. A referenciák kiválthatók véges leképezésekkel úgy, hogy a referenciák helyett kulcsokat használunk és a referenciák által hivatkozott értékeket egy, a kulcsokat tartalmazó véges leképezésben tartjuk.

Amennyiben referenciák kiváltására használjuk a véges leképezéseket, a következő feltételek általában teljesíthetőek:

- Bármely kulcs értékének írása után a kulcshoz tartozó korábbi értéket nem írjuk vagy olvassuk. Ez a feltétel teljesül referenciák esetén mivel ott nincs lehetőség a referencia írása előtti környezetre hivatkozni.
- A program futása során használt véges leképezések számára előre rögzített felső korlát van. Referenciák esetén ahol csak egyetlen, globális környezet áll rendelkezésre, aminek a feldarabolásával kapjuk a véges leképezéseket, amiknek a száma többnyire fordítási időben rögzített.

A módosított véges leképezések elemi műveletei, úgymint az írás, az olvasás, a kulcsok készítése és összehasonlítása konstans erőforrás igényűek, ha a fenti két természetes feltétel teljesül. Ez a tény alátámasztja hogy a módosított véges leképezések alkalmasak referenciák kiváltására.

Az elemi műveletek konstans erőforrásigényéből következik, hogy a program futása során használt kulcsok száma nem befolyásolja a műveletek erőforrás igényét. Ez a tulajdonság eltér a szakirodalomban ismert véges leképezés adatszerkezetektől, amikben az elemi műveletek hatékonysága egyenesen arányos a kulcsok számának a logaritmusával.

## Második tézis

### Általános interfész adható heapeknek

*Definiáltam egy általános heap interfészt mellyel lehetséges referenciák távoli létrehozása, heterogén és homogén heapek használata, osztott referenciák használata és a heap unió művelete, és ezen interfész hat különböző lehetséges implementációjának szemantikai vizsgálatát is elvégeztem.*

[10] cikk.

Heapnek nevezzük a memória absztrakt leírásában azt a részt, amely memóriahely-érték párok halmazaként képzelhető el. A heap tekinthető egy nem perzisztens véges leképezésnek is. Heapekkel kiváltható a referenciák használata tisztán funkcionális nyelvekben, ha a nyelv támogatja az egyediségi típusozást.

- Definiáltam egy általános heap interfészt. Az interfészben lehetővé tettem referenciák távoli létrehozását, heterogén és homogén heapek használatát, és referenciák megosztott használatát. Definiáltam a heap unió műveletet is.
- Megadtam 6 lehetséges heap modellt az általam definiált heap interfészhez, és ezeket össze is hasonlítottam.
- Megmutattam, hogy a Clean nyelvben használt heapek elláthatók az általam definiált interfésszel, és ezzel javítható a korábbi Clean heap interfésznek egy jelentős hibája.

## Harmadik tézis

### Biztonságossá tehető a referencia szerinti azonosságvizsgálat

*Kidolgoztam az algebrai adattípusok referencia szerinti azonosságvizsgálatát.*

[5] cikk, [3], [4] kapcsolódó cikkek.

Funkcionális nyelvekben az algebrai adattípusok implementációja többnyire referenciákkal történik: minden konstruktorból referenciák mutatnak a konstruktor paramétereire. Ezek a referenciák mintaillesztéssel olvashatók, a referenciák írása és azonosságvizsgálata viszont nem megengedett, mivel elrontaná a hivatkozások behelyettesíthetőségét.

Kibővítettem egy tisztán funkcionális nyelvet referencia szerinti egyenlőségvizsgálattal úgy, hogy egyszerű feltétel adható a hivatkozások behelyettesíthetőségére és teljesül a függvénydefiníciók behelyettesíthetősége, szemben a  $\nu$ -kalkulus[15] megoldásával. Ezeket az állításokat bizonyítottam is.

## Kapcsolódó publikációk

Folyóirat cikkek [9], [10].

Referált cikkek [5], [6].

Absztraktok [3], [4], [7], [8].

## Hivatkozások

- [1] Stephen Adams. Efficient sets: a balancing act. *Journal of Functional Programming* 3(4), pages 553–562, October 1993.
- [2] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.
- [3] Péter Diviánszky. Direct graph manipulation in functional languages. Presented at the Central-European Functional Programming School, Eötvös Loránd University, Budapest, Hungary, 4-16 July, 2005.
- [4] Péter Diviánszky. Substructural functional programming. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05*, page 3 pages. Trinity College, Dublin, 2005. Technical Report TCD-CS-2005-60.
- [5] Péter Diviánszky. Unique identifiers in pure functional languages. In Henrik Nilsson, editor, *Proceedings of the Seventh Symposium on Trends in Functional Programming (TFP)*, pages 84–98. The University of Nottingham, 2006.
- [6] Péter Diviánszky. Efficient implementation of linearly used finite maps. In Zoltán Horváth, László Kozma, and Viktória Zsók, editors, *Proceedings of the 10th Symposium on Programming Languages and Software Tools*, pages 199–213. Eötvös University Press, 2007.
- [7] Péter Diviánszky. Linearly used finite maps. In *2nd Central European Functional Programming School (CEFP 2007), Cluj Napoca, Romania, PhD workshop abstracts*, page 1 page, 2007.
- [8] Péter Diviánszky. Translating imperative algorithms to functional code with unique variable environments. In Sven-Bodo Scholz, editor, *Implementation and Application of Functional Languages, 20th International Symposium, IFL*



- 2008, pages 216–221, Hatfield, Hertfordshire, UK, September 2008. Technical Report No. 474, School of Computer Science, University of Hertfordshire.
- [9] Péter Diviánszky. Efficient implementation of linearly used finite maps. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:201–218, 2009.
  - [10] Péter Diviánszky. Non-monadic models of mutable references. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Third Summer School, CEFP 2009, Budapest, Hungary, May 2009 and Komárno, Slovakia, May 2009, Revised Selected Lectures*, volume 6299 of *Lecture Notes of Computer Science*, pages 147–187. Springer Verlag, 2010.
  - [11] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell.
  - [12] B. Lippmeier. *Type inference and optimisation for an impure world*. PhD thesis, Australian National University, 2009.
  - [13] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
  - [14] Harald Sondergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27(6):505–517, January 1990.
  - [15] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, 1994.
  - [16] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast: A functional semantics for the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36, New York, NY, USA, 2007. ACM.
  - [17] Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Brückner, editor, *ESOP 92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer, 1992.