# Magasabbrendű funkcionális programozás

**Hanák Péter**

**hanak@inf.bme.hu**

**Irányítástechnika és Informatika Tanszék**

**OM Kutatás-Fejlesztési Helyettes Államtitkárság**

**Hanák Dávid**

**dhanak@inf.bme.hu**

**Számítástudományi és Információelméleti Tanszék**

**Csala Viktor**

**viktor@csala.net**

**AAM Technologies Kft.**

# BEVEZETÉS

# Bevezetés

## Az előadás felépítése

- 1. alkalom: a λ-kalkulus alapjai
- 2. alkalom: a λ-kalkulus, mint programozási nyelv
- 3. alkalom: Típusok bevezetése
  A Caml nyelvcsalád bevezetése a λ-kalkuluson keresztül

# Felhasznált irodalom

## A λ-kalkulus jelentős irodalommal bír, az előadás alapját a következő bevezető cikkek képezték:

- Greg Michaelson: An Introduction to Functional Programming Through Lambda Calculus
- John Harisson: Intorduction to Functional Programming (A cikk a lambda kalkuluson keresztűl vezeti le a Caml alapjait, így a továbbiakban is hasznos lehet, emellett *az előadás szerkezete a cikk szerkezetéhez illeszkedik*)
- Henk Barendregt, Erik Barendsen: Introduction to Lambda Calculus (erősen matematikai tárgyalás)

## További elolvasásra érdemes még

- Csörnyei Zoltán: A lambda-kalkulus (magyar nyelvű, az ELTE-n használt jogvédett oktatási anyag)

# A λ-KALKULUS ALAPJAI

# A λ-kalkulus története - 1

## Leibniz

- Univerzális nyelv, amelyen minden probléma megfogalmazható
- Általános megoldási metódus, amely minden univerzális nyelven megfogalmazott problémára megoldást talál → *„Entscheidungsproblem”*

## Hilbert 2. kérdése: Létezik-e teljes és konzisztens matematikai axióma rendszer?

## Russell, Whitehead: Principia Mathematica

- A matematika logikai leírásának kísérlete

## Gödel (1931): Modellelmélet segítségével bebizonyította, hogy Hilbert kérdésére a válasz nemleges!

# A λ-kalkulus története - 2

**Gödel „negatív” eredménye ellenére az univerzális megoldó eszköz keresése nyomán megszületett a „kiszámíthatóság” elmélete (theory of computability)**

- Turing: Turing-gép (többé-kevésbé a mai számítógépek őse, a Neumann architektúra lényegében véges Turing-gép közvetlen elérésű memóriával szalag helyett)
- Kleene: Rekurzív függvények elmélete
- *Church: λ-kalkulus*

**Church tézise szerint a három metódus ekvivalens és minden probléma leírható velük (ezt mai napig nem bizonyították formálisan).**

***Kleene és Church munkája tekinthető a funkcionális paradigma alapjának.***

# A λ-kalkulus formális leírása

## A formális szintaxis a következőképp néz ki:

- *< λ-kifejezés >* → *< változó >* |
  *<konstans>* |
  *< λ-absztrakció >* |
  *< applikáció >*
- *< λ-absztrakció >* → *( λ < változó > . < λ-kifejezés > )*
- *< applikáció >* → *( < λ-kifejezés > < λ-kifejezés > )*

## Az előadás során alkalmazott szabályok

- Függvény alkalmazás jelölése: *f(x)*, vagy *f x*
- A függvény applikáció balra köt: *f x y* jelentése *(f(x))(y)*
- *λ x.λ y.t[x, y]* helyett *λ x y.t[x,y]*-t írunk
- A λ-absztrakciót minden esetben a lehető leghosszabban kiterjesztjük jobbra, vagyis *λ x.x y* jelentése *λ x.(x y)* (ezt sok szerző nem teszi meg).

# Mit jelent a λ?

**Gyakorlatilag semmit…**

**A λ csak jelölési konvenció (pl. jellemzően francia szövegekben mai napig használják a *[a] t[a]* jelölést), létét egy „evolúciós” folyamatnak köszönheti:**

1. Russell és Whitehead a Principia Mathematica-ban a kalap jelölést használta: *t[â]*.
2. Church módosította a jelölést: *â.t[a]*, de ezt csak *∧a.t[a]* formában tudták az akkori nyomdászok megjeleníteni.
3. Végül egy újabb nyomdász kezében, valószínűleg tévedésből kialakult a végleges forma: *λa.t[a]*

# A λ jelölés előnyei

- Nagyon egyszerű eszközrendszerrel gyakorlatilag tetszőleges bonyolultságú matematikai elemek leírhatók és jellemezhetők (persze idővel az egyszerűségért az átláthatóság csökkenésével fizetünk, de mindez javítható szintaktikus édesítőszerek alkalmazásával, ld. Michaelson könyv).
- Matematikai szövegekben jellemzően az *f(x)* jelölést használják mind magára az *f* függvényre való hivatkozásra, mind az *f x*-re való alkalmazására. Ezt a kétértelműséget teljesen megszűnteti a λ jelölés.
- Erősen megkönnyíti a változók kezelését (bevezethetővé válik a kötött és a szabad változók fogalma, illetve a programozási nyelveknél jól ismert láthatóság (scoping)).

# A λ-kalkulus ereje

- Számítási képességét tekintve valószínűleg ekvivalens a Turing-géppel (Church tézis)
- Ennek megfelelően léteznek a Turing-gép problémáival (pl. megállási probléma) ekvivalens λ-kalkulus problémák.
- Russell paradoxon: az önmagát nem tartalmazó halmazok halmaza tartalmazza és nem is tartalmazza önmagát:
  $(R = \{x \mid x \notin x\} \Rightarrow R \in R \Leftrightarrow R \notin R)$
  λ-kalkulus megfelelője: $R = \lambda\, x.\neg(x\ x)$, így $R\ R = \neg(R\ R)$, már pedig ez pont ellentétes a negálás természetes jelentésével.
  *Church ezt felismerve vezette be a típusokat a λ-kalkulusba, ami kiküszöbölte az ellentmondást (ld. a 3. alkalom anyagát).*

# Szabad és kötött változók

- Szabad változók:

| | | | |
|---|---|---|---|
| $FV(x)$ | = | $\{x\}$ | változó |
| $FV(c)$ | = | $\emptyset$ | konstans |
| $FV(s\ t)$ | = | $FV(s) \cup FV(t)$ | applikáció |
| $FV(\lambda\ x.s)$ | = | $FV(s) - \{x\}$ | λ-absztrakció |

- Kötött változók:

| | | | |
|---|---|---|---|
| $BV(x)$ | = | $\emptyset$ | változó |
| $BV(c)$ | = | $\emptyset$ | konstans |
| $BV(s\ t)$ | = | $BV(s) \cup BV(t)$ | applikáció |
| $BV(\lambda\ x.s)$ | = | $BV(s) \cup \{x\}$ | λ-absztrakció |

- pl. $s = (\lambda\ x\ y.x)(\lambda\ x.z\ x)$, akkor $FV(s) = \{z\}$ és $BV(s) = \{x, y\}$

# Helyettesítés

- Intuitív megközelítésben egyszerű a helyettesítés, de nekünk most gépies metódust kell megalkotnunk.
- Pl. $(\lambda\, y.x+y)[y/x] = \lambda\, y.y + y$, márpedig nem ez az elvárt működés.
- Helyes szabályok:

$$
\begin{aligned}
x[t/x] &= t \\
y[t/x] &= y \text{ if } x \neq y \\
c[t/x] &= c \\
(s_1\, s_2)[t/x] &= s_1[t/x]\; s_2[t/x] \\
(\lambda\, x.s)\,[t/x] &= \lambda\, x.s \\
\boldsymbol{(\lambda\, y.s)\,[t/x]} &= \boldsymbol{\lambda\, y.(s[t/x]),\ ha\ x \neq y\ és\ x \notin FV(s)\ vagy\ y \notin FV(t)} \\
\boldsymbol{(\lambda\, y.s)\,[t/x]} &= \boldsymbol{\lambda\, z.(s[z/y][t/x])\ egyébként,\ ahol\ z \notin FV(s) \cup FV(t)}
\end{aligned}
$$

# Konverziók

## A λ-kalkulus három konverziós szabályon alapul:

- α-konverzió: $\lambda\, x.s \rightarrow_\alpha \lambda\, y.s[y/s]$ *feltéve, hogy* $y \notin FV(s)$
- β-konverzió: $(\lambda\, x.s)\ t \rightarrow_\beta s[t/x]$
- η-konverzió: $\lambda\, x.t\ x \rightarrow_\eta t$ *feltéve, hogy* $x \notin FV(t)$ (pl. $\lambda\, u.v\ u \rightarrow_\eta v$, de $\lambda\, u.u\ u\ \not\rightarrow_\eta u$)

# A λ-azonosság

- Két λ-kifejezés azonos (=), ha létezik az előző konverziók olyan véges sorozata, amely az egyiket a másikba viszi át:

$$s \rightarrow_\alpha t \; vagy \; s \rightarrow_\beta t \; vagy \; s \rightarrow_\eta t \Rightarrow s = t$$

$$T \Rightarrow t = t$$

$$s = t \Rightarrow t = s$$

$$s = t \; és \; t = u \Rightarrow s = u$$

$$s = t \Rightarrow s\,u = t\,u$$

$$s = t \Rightarrow u\,s = u\,t$$

$$s = t \Rightarrow \lambda\, x.s = \lambda\, x.t$$

- Két λ-kifejezés szintaktikailag megegyezik (≡), ha teljesen azonos (vagyis $\lambda\, x.x \equiv \lambda\, x.x$, de a $\lambda\, y.y \equiv \lambda\, x.x$ már nem igaz, annak ellenére, hogy $\lambda\, y.y = \lambda\, x.x$)
- A szintaktikai egyezőség gyengített, de sokkal hasznosabb változata az, ahol a szintaktikai egyezőség elérhető kizárólag α-konverzióval ($\equiv_\alpha$).
  Pl. $\lambda\, x.x \equiv_\alpha \lambda\, y.y$

# λ-redukció - 1

- A λ-azonosság fontos, de nem túl hasznos reláció, sokkal hasznosabb az aszimmetrikus változata a λ-redukció ($\rightarrow$):

$$s \rightarrow_\alpha t \; vagy \; s \rightarrow_\beta t \; vagy \; s \rightarrow_\eta t \Rightarrow s \rightarrow t$$

$$T \Rightarrow t \rightarrow t$$

$$s \rightarrow t \; és \; t \rightarrow u \Rightarrow s \rightarrow u$$

$$s \rightarrow t \Rightarrow s\ u \rightarrow t\ u$$

$$s \rightarrow t \Rightarrow u\ s \rightarrow u\ t$$

$$s \rightarrow t \Rightarrow \lambda\ x.s \rightarrow \lambda\ x.t$$

- A redukció (bár nevével ellentétben néha méret növekedéssel jár együtt) a λ-kifejezések szisztematikus kiértékelését jelenti.
- Amennyiben egy kifejezés redukciója α-konverziók kivételével tovább már nem folytatható, úgy a kifejezés elérte a *normál formáját*.

# λ-redukció - 2

- Kézenfekvő, hogy egy összetett λ-kifejezés redukálása többféleképp is elvégezhető, amelyek között lehet végtelen redukciós sorozat is.

  Pl. $(\lambda\, x.y)(\lambda\, x.x\ x\ x)(\lambda\, x.x\ x\ x) \rightarrow$

  $(\lambda\, x.y)(\lambda\, x.x\ x\ x)(\lambda\, x.x\ x\ x)\ (\lambda\, x.x\ x\ x) \rightarrow$

  $(\lambda\, x.y)(\lambda\, x.x\ x\ x)(\lambda\, x.x\ x\ x)\ (\lambda\, x.x\ x\ x)(\lambda\, x.x\ x\ x) \rightarrow$

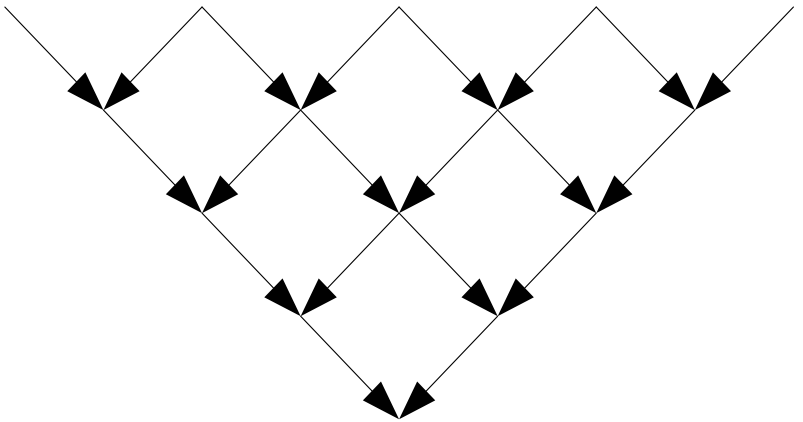
  ...

  de $(\lambda\, x.y)(\lambda\, x.x\ x\ x)(\lambda\, x.x\ x\ x) \rightarrow y$

- Tétel: ha $s \rightarrow t$ és t normál fomájú, akkor az a stratégia, amely minden esetben $s$ legkülső, legbaloldalibb redukálható részkifejezését redukálja véges redukciós sorozathoz vezet és normál formájú kifejezésben ér véget.

# Church-Rosser tétel

- Tétel: ha $t \rightarrow s_1$ és $t \rightarrow s_2$, akkor létezik olyan $u$ kifejezés, hogy $s_1 \rightarrow u$ és $s_2 \rightarrow u$.
- Következmény1: ha $t_1 = t_2$, akkor létezik $u$ kifejezés, hogy $t_1 \rightarrow u$ és $t_2 \rightarrow u$.



- Következmény2: ha $t = t_1$ és $t = t_2$, és $t_1$ és $t_2$ normál formájú, akkor $t_1 \equiv_\alpha t_2$.

# Kombinátorok

## A kombinátorok elmélete a λ-kalkulushoz hasonló elmélet, azonos számítási képességekkel:

- Kombinátornak tekinthető minden szabad változótól mentes λ-kifejezés. Ezeket zárt kifejezéseknek is nevezzük.
- 3 alap kombinátorral minden kifejezés kifejthető:

| | | | |
|---|---|---|---|
| I | = | λ x.x | („identity” – azonosság) |
| K | = | λ x y.x | (konstans) |
| S | = | λ f g x.(f x)(g x) | („sharing” – megosztás) |

- Sőt: I = S K A, ahol A bármilyen kombinátor lehet (rendszerint A-nak K-t választják, így I = S K K).
- A kombinátorok kicsit a gépi kód analógiájának felelnek meg a funkcionális világban.

# A λ-KALKULUS, MINT PROGRAMOZÁSI NYELV

# Első lépések

- Egy valós programozási nyelv elképzelhetetlen adatok nélkül.

  $\Downarrow$

  Az első lépésnek valamiféle adatreprezentáció kialakításának és az adatok végezhető alapvető műveleteknek kell lennie.
- Meg kell teremteni a programvezérlés valamilyen módját.
- Alapvető elemek:
  - igazságértékek
  - feltételes kifejezés
  - logikai operátorok
  - valamiféle számábrázolás, és a számokon végezhető műveletek
  - ismételt végrehajtás lehetősége
- Továbbiakban a definíció jelölése: $s \triangleq s'$, ami annyit jelent, hogy $s = s'$ definíció szerint.

# Igazságértékek

- A cél valamiképp a true és a false értékek definiálása λ-kifejezések segítségével.
- Természetes, hogy erre a célra elvben bármely két egymással nem egyenlő (λ-azonosságot tekintve) λ-kifejezés megfelelne.
- Célszerűségi okokból a következő definíciót használjuk:

$$true \triangleq \lambda x\, y.\, x$$
$$false \triangleq \lambda x\, y.\, y$$

- A fenti kifejezéseknek még nagyon fontos szerepe lesz a továbbiakban…
- Bebizonyítható, hogy ez a két λ-kifejezés semmilyen redukciós sorozattal sem vihető át egymásba (mindez intuitív úton is belátható, lévén egy pár első elemét kiválasztó függvény soha nem fog ugyanúgy viselkedni, mint a második elemet visszaadó kifejezés).

# Feltételes kifejezés

- A C '*c ? x : y*' konstrukcióját akarjuk megalkotni.
- Fontos, hogy nem feltételes utasításról, hanem tényleges kifejezésről beszélünk, így az „else" ág nem elhagyható (ez a legtöbb funkcionális nyelvben tényleg így is van).
- Ennek megfelelően:

$$\textit{if } E \textit{ then } E1 \textit{ else } E2 \quad \triangleq \quad E\ E1\ E2 \quad \text{(vagyis } \textit{(E(E1))(E2)}\text{)}$$

- Például:

$$\begin{aligned} \textit{if true then } E1 \textit{ else } E2 &= \textit{true } E1\ E2 \\ &= (\lambda x\ y.\ x)\ E1\ E2 \\ &= E1 \\ \textit{if false then } E1 \textit{ else } E2 &= \textit{false } E1\ E2 \\ &= (\lambda x\ y.\ y)\ E1\ E2 \\ &= E2 \end{aligned}$$

# Logikai operátorok

- Az igazságértékek logikai operátorok nélkül nem igazán használhatóak, de a feltételes kifejezés megteremtette a definiálásuk lehetőségét:

$$\begin{aligned} \textit{not } p &\triangleq \textit{if } p \textit{ then false else true} \\ p \textit{ and } q &\triangleq \textit{if } p \textit{ then } q \textit{ else false} \\ p \textit{ or } q &\triangleq \textit{if } p \textit{ then true else } q \end{aligned}$$

- Az igazságtáblák ellenőrzésével formálisan is bizonyítható ezek helyessége, de szükségtelen.

# Párok és többesek

- Minden felsorolás (lista vagy többes) tekinthető párok egymásba ágyazásának a következő módon: $(E_1, (E_2, (E_3, \ldots)))$ (ez egyfajta rekurzív konstrukció, ami az egyik legalapvetőbb funkcionális eszköz)
- Alapvető a párok ábrázolásának és kezelésének a képessége.
- Párok ábrázolása: $(E_1, E_2) \triangleq \lambda f.f\ E1\ E2$
- Destruktorok:

$$fst\ p \triangleq p\ true$$
$$snd\ p \triangleq p\ false$$

hiszen például:

$$\begin{aligned} fst\ (p, q) &= (p, q)\ true \\ &= (\lambda f.f\ p\ q)\ (\lambda x\ y.\ x) \\ &= (\lambda x\ y.\ x)\ p\ q \\ &= p \end{aligned}$$

- Többesekre (jelölés):

$$(p)_1 \triangleq fst\ p$$
$$(p)_i \triangleq fst\ (snd^{i-1}\ p)$$

# Curry-ing

- A párok kezelése lehetővé teszi a függvények hagyományos meghívását (*f x y z ...* helyett *f(x, y, z, ...)*)
- Az ehhez szükséges eszközök párokra:

$$CURRY\ f \triangleq \lambda x\ y.f(x, y)$$
$$UNCURRY\ g \triangleq \lambda p.g\ (fst\ p)\ (snd\ p)$$

- Többesekre:

$$CURRY_n\ f \triangleq \lambda x_1 \ldots x_n.f(x_1, \ldots, x_n)$$
$$UNCURRY_n\ g \triangleq \lambda p.g\ (p)_1 \ldots (p)_n$$

- Ezután $\lambda(x_1, \ldots, x_n).t$ írható az $UNCURRY_n\ (\lambda x_1 \ldots x_n.t)$ rövidítéseként

# Természetes számok

- Számábrázoláshoz a Chruch-számábrázolást használjuk, ami gyakorlatilag megegyezik a Turing-gépeknél tanulthoz, vagyis az unáris számokról van szó: 1, 11, 111, … (ez nem túl hatékony, de rendelkezik néhány nagyon kellemes tulajdonsággal).
- Ennek megfelelően: $n \triangleq \lambda f\, x.f^n\, x$,
- Például:

$$0 \triangleq \lambda f\, x.x$$
$$1 \triangleq \lambda f\, x.f\, x$$
$$2 \triangleq \lambda f\, x.f\, (f\, x)$$

- A számok ilyetén ábrázolásának megválasztása bár nem önkényes, de nem is kötelező, gyakorlatilag alkalmazható bármilyen hasonló tulajdonságokkal rendelkező λ-kifejezés (pl. a 0 lehet az azonosság függvény – $\lambda x.x$ és a minden egyes szám a $\lambda n\ s.(s\ false)\ n$ növekmény függvény megfelelő számú alkalmazásával definiálható).

# Műveletek természetes számokon - 1

- Nulla vizsgálat: $ISZERO\ n \triangleq n\ (\lambda x.false)\ true$

  vagyis $ISZERO\ 0 = (\lambda f\ x.x)\ (\lambda x.false)\ true$

  $= true$

  és $ISZERO\ (n+1) = (\lambda f\ x.f^{n+1}\ x)\ (\lambda x.false)\ true$

  $= (\lambda x.false)^{n+1}\ true$

  $= (\lambda x.false)((\lambda x.false)^{n}\ true)$

  $= false$

# Műveletek természetes számokon - 2

- Következő szám:

$$SUC \triangleq \lambda n\, f\, x.n\, f\, (f\, x)$$

vagyis

$$\begin{aligned} SUC\ n &= (\lambda n\, f\, x.n\, f\, (f\, x))\ (\lambda f\, x.f^n\ x) \\ &= \lambda f\, x.(\lambda f\, x.f^n\ x) f\, (f\, x) \\ &= \lambda f\, x.(\lambda f\, x.f^n\ x)(f\, x) \\ &= \lambda f\, x.f^n\ (f\, x) \\ &= \lambda f\, x.f^{n+1}\ x \\ &= n + 1 \end{aligned}$$

# Műveletek természetes számokon - 3

- Az előző szám már nehezebb (cél: $PRE\ 0 = 0$ és $PRE\ (n + 1) = n$)
- Trükk a párokra értelmezett *PREFN* deklarálása, ami eldobja *f* egy példányát $f^n$-ből a következőképp:

  $PREFN\ f\ (true, x) = (false, x)$

  és $PREFN\ f\ (false, x) = (false, f\ x)$

  vagyis $PREFN \triangleq \lambda f\ p.(false, if\ fst\ p\ then\ snd\ p\ else\ f\ (snd\ p))$
- Majd ezzel: $PRE \triangleq \lambda f\ x.snd(n\ (PREFN\ f)\ (true, x))$

  (a helyhiány miatt eltekintek a levezetéstől, mivel viszonylag hosszú, de mindenki kipróbálhatja).

# Műveletek természetes számokon - 4

- Összeadás:

$$m + n \triangleq \lambda f\, x.m\, f\, (n\, f\, x)$$

vagyis

$$\begin{aligned} m + n &= \lambda f\, x.m\, f\, (n\, f\, x) \\ &= \lambda f\, x.(\lambda f\, x.f^m\, x)(n\, f\ x) \\ &= \lambda f\, x.f^m\, (n\, f\, x) \\ &= \lambda f\, x.f^m\, ((\lambda f\, x.f^n\, x)\, f\, x) \\ &= \lambda f\, x.f^m\, ((\lambda x.f^n\, x)\, x) \\ &= \lambda f\, x.f^m\, (f^n\, x) \\ &= \lambda f\, x.f^{m+n}\, x \end{aligned}$$

# Műveletek természetes számokon - 5

- Szorzás: $m * n \triangleq \lambda f\, x.m\ (n\ f)\ x$

  vagyis

$$
\begin{aligned}
m * n &= \lambda f\, x.m\ (n\ f)\ x \\
&= \lambda f\, x.(\lambda f\, x.f^m\ x)\ (n\ f)\ x \\
&= \lambda f\, x.(\lambda x.(n\ f)^m\ x)\ x \\
&= \lambda f\, x.(n\ f)^m\ x \\
&= \lambda f\, x.((\lambda f\, x.f^n\ x)\ f)^m\ x \\
&= \lambda f\, x.(\lambda x.f^n\ x)^m\ x \\
&= \lambda f\, x.(f^n)^m\ x \\
&= \lambda f\, x.f^{n*m}\ x
\end{aligned}
$$

# Rekurzív függvények – bevezető

- Az ismételt végrehajtás alapvető programozási elem, erre a célra a funkcionális programozáson belül a rekurzió szolgál.
- Lévén a λ-kalkulus név nélküli függvényekkel dolgozik első gondolatra lehetetlennek tűnik a rekurzív kifejezések definiálása.
- Szerencsére ezt nem minden matematikus gondolta így és nem kis erőfeszítéssel megtalálták a megoldást (Curry publikálta először), ami egy előtét függvény alkalmazásával lehetővé teszi a rekurzió alkalmazását a λ-kalkulusban.
- Az előtét függvény neve **fix pont kombinátor**:
  Az *Y* zárt λ-kifejezést akkor nevezzük fix pont kombinátornak, ha minden *f* λ-kifejezésre $f\ (Y\ f) = Y\ f$
- A fix pont kombinátor egy *f* kifejezésre alkalmazva azt az *x* értéket adja vissza, amelyre $f(x) = x$.

# Rekurzív függvények – fix pont kombinátor

- Russell-paradoxon: $R = \lambda x.\neg(x\ x)$
  és $R\ R = \neg(R\ R)$
  vagyis $R$ a $\neg$ operátor fix pont kombinátora.
- Általánosítsuk $R$-t: $Y \triangleq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$
  amire
  $$\begin{aligned} Y\ f &= (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ f \\ &= (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \\ &= f\ (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \\ &= f\ (Y\ f) \end{aligned}$$
- Mivel az utolsó lépésben egy fordított β-redukciót hajtottunk végre, a fenti kifejezés programozási szempontból nem megfelelő (mivel ott csak a normális redukciós lépések megengedettek), így gyakorlati alkalmazásokban Turing után: $T \triangleq (\lambda x\ y.\ y\ (x\ x\ y))\ (\lambda x\ y.\ y\ (x\ x\ y))$.
- Továbbiakban $Y$-t használjuk, de a helyére bármelyik fix pont kombinátor használható (pl. £££££££££££££££££££££, ahol £ = $\lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$)

# Rekurzív függvények definiálása

- Példaként definiáljuk a faktoriális függvényt a következőképp:

  $$fact(n) = \text{if } ISZERO\ n \text{ then } 1 \text{ else } n * fact(PRE\ n)$$

- Írjuk át λ-kifejezésszerű formára:

  $$fact = \lambda n.\ \text{if } ISZERO\ n \text{ then } 1 \text{ else } n * fact(PRE\ n)$$
  $$= (\lambda f\ n.\ \text{if } ISZERO\ n \text{ then } 1 \text{ else } n * f(PRE\ n))\ fact$$

- Ha jól megnézzük látszik, hogy fact a

  $$F = \lambda f\ n.\ \text{if } ISZERO\ n \text{ then } 1 \text{ else } n * f(PRE\ n)$$

  függvény fix pontja, vagyis $fact = Y\ F$.

- Hasonlóan megoldható a kölcsönösen rekurzív függvények definiálása is.

# TÍPUSOK BEVEZETÉSE

# Típusok

- Mind logikai, mind programozási szempontból indokolt a típusok bevezetése.
- A logikai szempontot a Russell paradoxon megléte adja, ami gyakorlatilag azért fordulhat elő, mert megengedett a függvények önmagukra való alkalmazása $\Rightarrow$ ennek kiküszöbölése megszünteti a paradoxont, emiatt vezette be eredetileg Russell a típusokat a Principia Mathematica-ban.
- Programozási szempont összetettebb:
  - Már a FORTRAN is megkülönböztet típusokat, mert így hatékonyabb kódot tud előállítani (ugyanez igaz pl. a C mutató aritmetikájára).
  - A típusok megléte segít kiküszöbölni a programozási hibák jelentős részét.
  - A típusok használata több féle lehet: nincsenek típusok (BCPL, ISWIM, SASL, Erlang), gyenge típizálás (PL/I), futásidejű dinamikus típusellenőrzés (LISP), fordítás idejű erős tipizálás polimorfizmussal (ML).

# Típusos λ-kalkulus

- Típusok bevezetése a λ-kalkulusba viszonylag kézenfekvő dolognak tűnik, de az eredménye nagyon messzire visz.
- Alapötlet: minden λ-kifejezésnek legyen típusa, így *s* csak akkor alkalmazható *t*-re (*s t)*, ha a típusok megfelelően illeszkednek. Vagyis, ha *s* típusa $\sigma \rightarrow \tau$ és *t* típusa $\sigma$, akkor *s t* típusa $\tau$.
- Programozási nyelvek között ezt hívják erős tipizálásnak, vagyis *t*-nek $\sigma$ típusúnak kell lennie, nincs szó altípusokról vagy megfeleltetésről (ezzel szemben pl. a C nyelvben egy függvény double típusú paramétert vár, akkor elfogad float típusút is).
- Jelölésmód: $t : \sigma$ jelentése *t* típusa $\sigma$ (ez teljesen összhangban van a matematikában használt jelölésmóddal).
- A típusokra úgy tekintünk, mint értékhalmazokra, vagyis $t : \sigma \Rightarrow t \in \sigma$.

# Típusok halmaza

- A formális tárgyalás első lépéseként meg kell határoznunk, hogy pontosan mit értűnk típusokon.
- Feltételezzük, hogy rendelkezünk néhány primitív típussal (pl. bool, int, stb.).
- Vannak típus konstruktoraink, amelyekkel a primitív típusokból összetett típusokat hozhatunk létre, jelölésük $(\alpha_1, \ldots, \alpha_n)$con (pl. $\rightarrow$, $\times$ vagy list).
- Ennek megfelelően $Ty_C$ típus halmaza a $C$ típushalmazból a következő induktív szabályokkal állítható elő ($\alpha_i$ tetszőleges típust jelöl):

$$\frac{\sigma \in C}{\sigma \in Ty_C}$$

$$\frac{\forall i \in [1..n] : \alpha_i \in Ty_C}{(\alpha_i, \ldots, \alpha_n)con \in Ty_C}$$

# Church megközelítése

- Church a típusok bevezetésénél az explicit típusjelölést alkalmazta, vagyis minden kifejezéshez hozzárendelte a megfelelő típust.
- Az explicit típusmegadás szabályai a következők:

$$\frac{}{v : \sigma}$$

$$\frac{A\ c\ konstans\ típusa\ \sigma}{c : \sigma}$$

$$\frac{s : \sigma \rightarrow \tau\ és\ t : \sigma}{s\ t : \tau}$$

$$\frac{v : \sigma\ és\ t : \tau}{\lambda v.t : \sigma \rightarrow \tau}$$

# Curry megközelítése

- Curry a típusok kezelésére a teljesen implicit megoldást választotta, vagyis a kifejezések teljesen megegyeznek a nem típusos λ-kalkulus kifejezéseivel, vagy nincs egyáltalán típusuk, vagy van és akár többféle is lehet. Pl. a λx.x függvény típusa $\sigma \rightarrow \sigma$, ahol $\sigma$ tetszőleges típus lehet (mindig az adott környezet dönti el).
- Ebben az esetben a tipizálás mindig a környezettől függ, ami a változók típus hozzárendelésének véges halmaza.
- Formálisan: $\Gamma \vdash t : \sigma$, ahol $\Gamma$ elemei $v : \sigma$ változó típus összerendelések. $\Gamma$ lehet üres, akkor $\vdash t : \sigma$-t írunk.
- Feltételezzük, hogy $\Gamma$ nem tartalmaz ugyanarra a változóra ellentmondó típus hozzárendelést (tekinthető a változók halmaza és a típusok halmaza közötti függvénykapcsolatnak is).
- A továbbiakban ezt a tipizálást használjuk, mivel az ML nyelvek is ezt a fajta megoldást követik.

# Formális típus szabályok

- Az általunk használt tipizálás szabályai a következők:

$$\frac{v : \sigma \in \Gamma}{\Gamma \vdash v : \sigma}$$

$$\frac{A\ c\ konstans\ típusa\ \sigma}{\vdash c : \sigma}$$

$$\frac{\Gamma \vdash s : \sigma \to \tau\ és\ \Gamma \vdash t : \sigma}{\Gamma \vdash s\ t : \tau}$$

$$\frac{\Gamma \cup \{v : \sigma\} \vdash t : \tau}{\Gamma \vdash \lambda v.t : \sigma \to \tau}$$

- Pl. az azonosság függvény levezetése: a változók szabályából $\{x : \sigma\} \vdash x : \sigma$, majd az utolsó szabályból $\emptyset \vdash \lambda x.x : \sigma \to \sigma$.
- Churchnél ugyanez lehet $\lambda x : \sigma.(x : \tau) : \sigma \to \tau$ ...

# Típus megőrzés

- A típusok bevezetésének csak akkor van értelme, ha a redukció megőrzi egy kifejezés típusát. Ez bizonyíthatóan így van, most röviden felvázoljuk az η-redukcióra:
- Lemma 1: ha $\Gamma \vdash t : \sigma$ és $\Gamma \subseteq \Delta$, akkor $\Delta \vdash t : \sigma$.
  - Bizonyítás strukturális indukcióval mehet.
- Lemma 2: ha $\Gamma \vdash t : \sigma$, akkor $\Gamma_t \vdash t : \sigma$, ahol $\Gamma_t$ csak azokra a változókra tartalmaz hozzárendeléseket, amik szabadok $t$-ben (formálisan $\Gamma_t = \{x : \alpha \mid x : \alpha \in \Gamma \text{ és } x \in FV(t)\}$).
  - Ismét strukturális indukcióval bizonyítható.
- Tétel: ha $\Gamma \vdash t : \sigma$ és $t \rightarrow_\eta t'$, akkor $\Gamma \vdash t' : \sigma$ is fennáll.
  - A két lemma segítségével bizonyítható.
- Ugyanilyen módon belátható az α és a β-redukció is.

# Polimorfizmus

- A polimorfizmus és a túlterhelés (overloading) hasonló jelentésű fogalmak, mind a kettő nagy vonalakban azt jelenti, hogy egy kifejezésnek több típusa is lehet.
- A polimorfizmus pontos jelentése, hogy a kifejezés típusa minden olyan típus lehet, ami megfelel a megadott mintának (paraméteres polimorfizmus).
- A túlterhelés „ugyanahhoz” a kifejezéshez több strukturálisan nem összefüggő típust rendel hozzá (ad-hoc polimorfizmus).

# Let utasítás - előtekintés

- A kifejezések átláthatóságának érdekében érdemes bevezetni a lokális értékadásnak megfelelő „szintaktikai édesítőszert”.
- Ennek megfelelően:

$$let\ x = s\ in\ t \quad \triangleq \quad (\lambda x.t)\ s$$

- A fenti forma az ML nyelvek alapvető eleme, sőt a CAML alapvető definíciós eszköze (az SML val és fun utasításainak a CAML-ben a let felel meg).

# A let utasítás polimorfizmusa

- Tekintsük a *if (λx.x) true then (λx.x) 1 else 0* kifejezést:

$$\dfrac{\dfrac{\dfrac{\{x : bool\} \vdash x : bool}{\vdash \lambda x.x : bool \to bool} \quad \vdash true : bool}{\vdash (\lambda x.x)\ true : bool} \quad \dfrac{\dfrac{\{x : int\} \vdash x : int}{\vdash \lambda x.x : int \to int} \quad \vdash 1 : int}{\vdash (\lambda x.x)\ 1 : int} \quad \vdash 0 : int}{\vdash if\ (\lambda x.x)\ true\ then\ (\lambda x.x)\ 1\ else\ 0 : int}$$

- De *let I = λx.x in if I true then I 1 else 0* kifejezéshez, ami a *(λI.if I true then I 1 else 0) λx.x* kifejezésnek felel meg, nem rendelhető típus, mivel I típusa már a λ-absztrakcióban eldől.
- Emiatt a let-et, mint primitív utasítást vezetjük be:

$$\dfrac{\boldsymbol{\Gamma \vdash s : \sigma} \quad \Gamma \vdash t[s/x] : \tau}{\Gamma \vdash let\ x = s\ in\ t : \tau}$$

# A legáltalánosabb típus

- A Curry-féle típus megállapításra a polimorfizmusnál erősebb állítás is igaz, azaz minden kifejezéshez létezik egy legáltalánosabb típus (principal type), és a kifejezés minden lehetséges típusa ennek egyede.
- A tétel megfogalmazásához bevezetjűk a típusváltozókat és az azokon értelmezett helyettesítést:

$$\begin{aligned} \alpha_i[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] &= \tau_i \\ \beta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] &= \beta, \text{ ha } \alpha_i \neq \beta \ (1 \leq i \leq k) \\ (\alpha_1, \ldots, \alpha_n)con[\theta] &= (\alpha_1[\theta], \ldots, \alpha_n[\theta])con \end{aligned}$$

- Az alaptípusokat nullargumentumos konstruktorként kezeljük, pl. *()int*.
- Jelölés: $\sigma \preceq \sigma'$ jelentése $\sigma$ általánosabb típus, mint $\sigma'$.
- Tétel: Minden típusos kifejezéshez létezik egy legáltalánosabb típus, vagyis ha $t : \tau$, akkor létezik $\sigma$, hogy $t : \sigma$ és bármely $\sigma'$-re, ha $t : \sigma'$, akkor $\sigma \preceq \sigma'$.
  - A bizonyítás a Milner (v. Hindley-Milner) algoritmuson alapszik, amire az ML-ekben alkalmazott típus-meghatározás is épül.

# Erős normalizáció

- Tekintsük a korábban már mutatott példát:

$$((\lambda x.x\ x\ x)(\lambda x.x\ x\ x))$$
$$\rightarrow ((\lambda x.x\ x\ x)(\lambda x.x\ x\ x)(\lambda x.x\ x\ x))$$
$$\rightarrow (\ldots)$$

- A típusos kifejezések világában ez a fajta viselkedés nem létezik, ahogy azt a következő tétel kimondja.
- Tétel: minden tipizálható kifejezésnek létezik normál formája (ha itt vége lenne, akkor beszélnénk gyenge normalizációról) és minden lehetséges redukciós sorozat terminál.

# A típusos λ-kalkulus korlátai

- Az erős normalizáció jól hangzik, de ára van, mivel nagyon korlátozott azoknak a függvényeknek a száma, amit a típusos λ-kalkulus ki tud fejezni (sőt bebizonyítható, hogy a leírható kifejezések köre nagyban függ a választott számábrázolástól).
- Pl. nem termináló kifejezés írása alapvető feltétele a Turing-teljesség elérésének.
- A rekurzió sem megvalósítható a korábban felvázolt módon, mert a fix pont kombinátorok jellemzően nem tipizálhatóak.
- A Turing-teljesség érdekében bevezetünk egy polimorf rekurziós operátort:

$$Rec : ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau$$

és az ehhez tartozó redukciós szabályt:

$$Rec\ F \to F\ (Rec\ F)$$

# A CAML NYELVCSALÁD BEVEZETÉSE A λ-KALKULUSON KERESZTÜL

# Rövid ismertető

- A Caml Light és az OCaml (az utóbbi objektum orientált) az Edinborough ML-ből alakult ki, az ML nyelvjárásának tekinthető (a Standard ML csak nevében szabványos, ténylegesen még nem szabványosították).
- Keletkezési helye a francia INRIA.
- Teljesítményében összemérhető a C++-szal, köszönhetően a hatékony megvalósításának (a nyelv megvalósításának alapja Category Abstract Machine-nek nevezett eszköz – innen kapta a nevét is – megfelelően hangolt garbage collectionnal kiegészítve).
- Bővebben a Caml előadáson…

# A λ-kalkulus megfelelői a Caml-ben

- Névadás: `let <név> = <kifejezés>`
- Helyi értékadás: `let <név> = <kifejezés> in <kifejezés>` (ld. korábban)
- Rekurzív definíció: `let rec <név> = <rekurzív kifejezés>`
- Kölcsönösen rekuzív definíció:
  ```
  let rec <név> = <kölcsönösen rekurzív kifejezés>
  and <név> = <kölcsönösen rekurzív kifejezés>
  and …
  ```
- A λ-absztrakció:
  `function <paraméter> -> <kifejezés>` vagy
  `fun <paraméter> … <paraméter> -> <kifejezés>`
- ...