CSÖRNYEI ZOLTÁN

FUNKCIONÁLIS PROGRAMNYELVEK IMPLEMENTÁCIÓJA

I. rész

BUDAPEST, 2010

Előszó

Ez a jegyzet az Eötvös Loránd Tudományegyetem Informatikai Karán tartott *Funkcionális programnyelvek implementációja* című tantárgy előadásainak *első részét* tartalmazza.

Az anyag a λ-kalkulus és kombinátor logika ismeretére épül. Ezekkel a témakörökkel részletesen a [2] könyv foglalkozik, ezért ebben a jegyzetben ezeknek a kalkulusoknak a részletes tárgyalására nem térünk ki, csupán a kalkulusok alaptulajdonságait foglaljuk össze.

Ebben az első részben az implementáció klasszikus módszereit tárgyaljuk. Először kibővítjük a λ-kalkulust a mintákkal, let-, letrec- és case-utasítással, és tanulmányozzuk, hogy a funkcionális programok hogyan alakíthatók át a kibővített λ-kalkulus kifejezéseire. Ezután ezeket a kifejezéseket az egyszerű konstansos λ-kalkulus kifejezéseire transzformáljuk, és az ilyen alakú programok, azaz λ-kifejezések végrehajtása már csak a szokásos egyszerű redukciós lépések végrehajtását jelenti.

Külön foglalkozunk a mintaillesztés problémájával, és algoritmust adunk a mintaillesztés hatékony elvégzésére. Vizsgáljuk a listákat tartalmazó programok fordítását is, példán keresztül megmutatva, hogy a végtelen listák kezelése sem okoz problémát.

A jegyzet utolsó fejezetében azt vizsgáljuk, hogy a kibővített λ-kalkulus kifejezései hogyan alakíthatók át a kombinátor logika kifejezéseire. Ilyen kifejezésekkel a program végrehajtása még egyszerűbb, hiszen csak néhány kombinátor gyenge redukciós műveletét kell ismételgetni.

A téma könnyebb megértése és elsajátítása érdekében a könyvben nagyon sok példát adunk, a példák végét a □ jellel jelöljük.

A jegyzetben szereplő algoritmusok egy része Simon L. Peyton Jones és David R. Lester [5] és [6]-ban szereplő implementációval foglalkozó munkáján alapul. A kombinátor logikás alkalmazásokra a [4] néhány fejezete szolgálhat mintául.

A jegyzet tervezett *második része* a funkcionális programoknak a kifejezés- és gráf-újraíró rendszereken alapuló implementációját tárgyalja. Ezzel kapcsolatos témakör például [7] és [8]-ban található. A funkcionális programok fordításának módszereivel foglalkozik az [1] egy fejezete is.

Megjegyezzük, hogy részben a funkcionális programnyelvek fordításának speciális problémáival foglalkozik az *IFL Implementation and Application of Functional Languages* konferenciasorozat is, amelynek előadásai a *Lecture Notes in Computer Science* kiadványaiban jelennek meg.

* * *

Köszönetemet fejezem ki Diviánszky Péternek, a jegyzet lektorának hasznos tanácsaiért, rendkívüli gondossággal végzett alapos munkájáért.

Köszönöm a munkatársaknak és egyetemi hallgatóknak az észrevételeiket, ezek a jegyzet megírásához nagy segítséget nyújtottak. Külön köszönet illeti az Eötvös Loránd Tudományegyetem Informatikai Karát, amiért lehetővé tette ennek a jegyzetnek a megjelentetését.

Kérem, hogy észrevételeiket a `csz@inf.elte.hu` címre írják meg.

Budapest, 2010. november 2.

Csörnyei Zoltán

Jelölések

nyilak:

$\Longrightarrow$	programból kibővített λ-kalkulusba, vagy kibővített λ-kalkulusból kibővített λ-kalkulusba
$\rightsquigarrow$	kibővített λ-kalkulusból λ-kalkulusba
$\rightarrow, \leftrightarrow$	λ-kalkulus redukciója
$\rightarrow_w$	kombinátor logika (gyenge) redukciója
$\ldots^+$	egy vagy több ismétlés
$\ldots^*$	nulla, egy vagy több ismétlés

karakterek:

$x, y, \ldots$	változó
$E, F, \ldots$	kifejezés
$k, l, \ldots$	konstans
$p, q, \ldots$	minta
$s, \ldots$	összegkonstruktor
$t, \ldots$	szorzatkonstruktor

listák:

$xs, ys, \ldots$	lista
$x{:}xs, y{:}ys, \ldots$	x, y fejelemek, xs, ys a maradék részek
[ ]	üres lista
$[x_1, x_2, \ldots, x_n]$	$x_1, x_2, \ldots, x_n$ elemekből álló lista

Tartalomjegyzék

1. FEJEZET

Bevezetés

A λ-kalkulust Church definiálta 1932-33-ban, és ezzel a rendszerrel például a függvények kiszámíthatóságának elmélete könnyen leírhatóvá vált. Kleene megmutatta, hogy a kiszámítható függvények pontosan a λ-definiálható függvények, és a Turing-tétel a Turing-kiszámíthatóság és a λ-kiszámíthatóság ekvivalenciáját adta.

Az 1950-es években a számítógépek elterjedése, az első funkcionális programnyelvek megjelenése újból előtérbe helyezte a λ-kalkulus elméletét. Ekkor vált ismertté, hogy a a funkcionális nyelvű programok fordítására és végrehajtására a λ-kalkulus kiválóan alkalmazható, minden funkcionális program egy λ-kifejezésnek tekinthető, és a program végrehajtása a kifejezés legegyszerűbb formájának előállítását jelenti, ami a λ-kalkulus redukciós műveleteivel elvégezhető. A λ-kalkulus tehát az első funkcionális nyelv, egy olyan egyszerű funkcionális programnyelv, amelyre minden „magasszintű” funkcionális nyelven megírt program átalakítható. A λ-kalkulus redukcióinak interpretációja pedig a funkcionális programok végrehajtó rendszerét adja.

1924-ben Schönfinkel kidolgozott egy, még a λ-kalkulusnál is egyszerűbb rendszert, a kombinátor logikát, amiről szintén Kleene bizonyította be, hogy a λ-kalkulussal ekvivalens. Ezzel bizonyítottá vált a Turing-kiszámíthatóság, a λ-kalkulus és a kombinátor logika teljes egyenrangúsága. Így a funkcionális programok fordítására és végrehajtására az a módszer is alkalmazható, hogy a programot a kombinátor logika kifejezésére alakítjuk át és ezt a kifejezést redukáljuk.

2. FEJEZET

Kalkulusok és programok

2.1. A λ-kalkulus

A következőkben röviden összefoglaljuk a λ-kalkulus alaptulajdonságait. Először az *egyszerű λ-kifejezés*ek szintaktikáját adjuk meg.

Tegyük fel, hogy adott egy nem feltétlenül véges, egymástól páronként különböző változókat (szimbólumokat), a λ jelet, a pontot, valamint a nyitó- és csukózáró-jeleket tartalmazó halmaz. Ezen a halmazon, mint ábécén értelmezett λ-kifejezések a következő szavak:

$$
\begin{aligned}
\langle \lambda\text{-kifejezés} \rangle &::= \langle \text{változó} \rangle \\
&\mid \langle \lambda\text{-absztrakció} \rangle \\
&\mid \langle \text{applikáció} \rangle \\
\langle \lambda\text{-absztrakció} \rangle &::= (\lambda \langle \text{változó} \rangle . \langle \lambda\text{-kifejezés} \rangle) \\
\langle \text{applikáció} \rangle &::= (\langle \lambda\text{-kifejezés} \rangle \langle \lambda\text{-kifejezés} \rangle)
\end{aligned}
$$

A kifejezésekben a legkülső zárójelet elhagyjuk, és a kiírt zárójelek száma csökkenthető, ha feltesszük, hogy az absztrakciók jobbasszociatívak, az applikációk balasszociatívak, és az applikációnak mindig nagyobb a precedenciája, mint az absztrakciónak. Két λ-kifejezés szintaktikus *azonosságára* az $\equiv$ jelet használjuk.

Az absztrakciók a függvényeknek felelnek meg. A kifejezésben a pont előtti változó a függvény változója, a pont utáni kifejezés a függvény *törzse*. Azt mondjuk, hogy az absztrakció *köti* a változóját a kifejezés törzsében. A nem kötött változók a szabad változók. Ha egy kifejezésben nincs szabad változó, akkor a kifejezést *zárt kifejezésnek* nevezzük, és a zárt λ-kifejezéseket *kombinátoroknak* is mondjuk. Az E kifejezés szabad változóinak halmazát $FV(E)$-vel jelöljük.

A λ-kifejezések halmaza Λ, a zárt λ-kifejezések halmazának jele pedig a Λ^0.

Látható, hogy a λ-kalkulusban a függvények *magasabb rendű függvények*, és minden függvény egyváltozós. Többváltozós függvények a Curry nevéből származó *körrizés* műveletével alakíthatók át egyváltozós függvények kompozíciójává.

Ha az E λ-kifejezésben a *szabad x változót* mindenütt az F λ-kifejezéssel

helyettesítjük, akkor ezt a műveletet *helyettesítésnek* nevezzük, és a helyettesítéssel kapott λ-kifejezést $E[x := F]$-fel jelöljük. A helyettesítés a következő szabályokkal adható meg:

$$
\begin{aligned}
&1.\ x[y := G] && \equiv \begin{cases} G, & \text{ha } x \equiv y, \\ x, & \text{egyébként,} \end{cases} \\
&2.\ (E\,F)[y := G] && \equiv (E[y := G])(F[y := G]), \\
&3.\ (\lambda x\,.\,E)[y := G] && \equiv \begin{cases} \lambda x\,.\,E, & \text{ha } x \equiv y, \\ \lambda x\,.\,E[y := G], & \text{ha } x \not\equiv y \text{ és } x \notin FV(G), \\ \lambda x\,.\,E, & \text{egyébként.} \end{cases}
\end{aligned}
$$

A kifejezések átalakítási szabályait a λ-kalkulus *operációs szemantikája* határozza meg. Ezt a szemantikáját *konverziós szabályokkal* írjuk le, amelyek megadják, hogy egy λ-kifejezést hogyan lehet egy másik λ-kifejezésbe transzformálni.

Egy ilyen konverzió az α-konverzió, ezzel egy absztrakció változóját cserélhetjük egy másik változóra:

Ha az E-ben y nem szabad változó, azaz $y \notin FV(E)$*, akkor*
$\lambda x\,.\,E \leftrightarrow \lambda y\,.\,E[x := y]$.

Az α-konverzió alkalmazásával a helyettesítés 3. szabálya pontosabban is megadható:

$$
3.\ (\lambda x\,.\,E)[y := G] \equiv \begin{cases} \lambda x\,.\,E, & \textit{ha } x \equiv y, \\ \lambda x\,.\,E[y := G], & \textit{ha } x \not\equiv y \textit{ és } x \notin FV(G), \\ \leftrightarrow (\lambda z\,.\,E[x := z])[y := G] \equiv & \\ \qquad \lambda z\,.\,E[x := z][y := G], & \\ \qquad\qquad \textit{ha } x \not\equiv y \textit{ és } x \in FV(G). & \end{cases}
$$

A λ-absztrakció egy függvényt jelöl, és egy λ-absztrakcióból és egy λ-kifejezésből álló függvényapplikáció második tagja a függvény aktuális paramétere. A β-redukció megadja, hogy egy függvényt az aktuális paraméterére alkalmazva milyen eredményt kapunk:

Ha az $E[x := F]$*-ben az F szabad változói nem válnak az E kötött változóivá, akkor*
$(\lambda x\,.\,E)F \;\rightarrow\; E[x := F]$.

Ezeknek a redukcióknak, konverzióknak az ismeretében már tudjuk két kifejezés egyenlőségét is definiálni, az egyenlőséget az = jellel jelöljük.

Az E és F λ*-kifejezésekre* $E = F$*, ha*

- $E \equiv F$, *vagy*
- $E \overset{*}{\leftrightarrow} F$.

Ennek a fogalomnak az ismeretében már megadhatjuk az *egyszerű λ-kalkulus* definícióját:

Az egyszerű λ-kalkulus az egyszerű λ-kifejezések közötti olyan $E = F \quad (E, F \in \Lambda)$ *egyenlőségeket tartalmaz, amelyek a következő axiómák felhasználásával bizonyíthatók:*

I. $(\lambda x\,.\,E)F = E[x := F]$

II.i. $E = E$

II.ii. *ha* $E = F$, *akkor* $F = E$

II.iii. *ha* $E = F$ *és* $F = G$, *akkor* $E = G$

II.iv. *ha* $E = F$, *akkor* $EG = FG$

II.v. *ha* $E = F$, *akkor* $GE = GF$

II.vi. *ha* $E = F$, *akkor* $\lambda x\,.\,E = \lambda x\,.\,F$

Megjegyezzük, hogy az utolsó axiómát ξ-szabálynak nevezzük.

Egy kifejezés redukálása során gyakran szükség van még az η-konverzió alkalmazására is:

Ha az x az E-nek nem szabad változója, azaz $x \notin FV(E)$, *akkor*
$\lambda x\,.\,Ex \leftrightarrow E$.

Miután áttekintettük a λ-kalkulus redukciós szabályait, nézzük meg, hogy hogyan tudjuk egy kifejezés *normál formáját*, azaz tovább már nem redukálható alakját meghatározni. Ez a kérdés számunkra azért fontos, mert mint már említettük, a normál forma a funkcionális program futási eredményének fog megfelelni.

Könnyen belátható, hogy nem minden kifejezés hozható normál formára, ráadásul a normál forma elérésének lehetősége nem csak a kifejezéstől, hanem az alkalmazott redukálási stratégiától is függ. A gyakorlatban kétféle stratégiát szokás alkalmazni. Azt a redukálási stratégiát, amelyik a legbaloldalibb legkülső redukálható kifejezést, *redexet* redukálja, *normál sorrendű redukálási stratégiának*, amelyik a legbaloldalibb legbelső redexet redukálja, *applikatív sorrendű redukálási stratégiának* nevezzük.

A normál sorrendű redukálás jelentőségét a *II. Church–Rosser-tétel* adja meg:

A normál sorrendű redukálási stratégia normalizáló redukálási stratégia, azaz ha a normál forma létezik, akkor ez a stratégia a normál formát meghatározza.

Az applikatív sorrendű redukálási stratégia nem normalizáló, azaz előfordulhat,

hogy a normál formát akkor sem határozza meg, ha a normál forma létezik, viszont ha a normál formát megtalálja, akkor ezt általában kevesebb lépéssel teszi, mint a normál sorrendű stratégia.

Az *I. Church–Rosser-tétel* is a normál formákra vonatkozik, de számunkra, a funkcionális programok szempontjából, a tétel következményei, különösen a 2. következmény a fontos:

1. *Ha* $E_1 = E_2$*, és* E_2 *normál formában van, akkor* $E_1 \rightarrow^* E_2$.

2. *Minden λ-kifejezésnek legfeljebb egy normál formája van.*

3. *Ha E és F mindegyike normál forma, és* $E \not\equiv F$*, akkor* $E \neq F$.

A 2. következmény tehát azt mondja ki, hogy ha egy programnak van futási eredménye, akkor ez az eredmény egyértelmű.

Az egyszerű λ-kalkulusban nincsenek konstansok, és hiányoznak a matematikában, programozásban megszokott függvények is. Az egyes konstansokhoz, az egyes műveletekhez azonban találhatunk olyan λ-kifejezéseket, amelyekkel a helyes eredményeket kapjuk meg, azaz például a + 2 3 kifejezés esetén, ha az összeadás műveletének λ-kifejezésére applikáljuk a 2 és 3 számoknak megfeleltett λ-kifejezéseket, az applikációs λ-kifejezés redukálásával pontosan azt a λ-kifejezést kapjuk eredményül, ami az 5 számjegynek lett megfeleltetve.

A [2]-ben megtalálható a tréfás *„a típus csak illúzió”* állítás háttere, itt is látható, hogy *mindennek* egy vagy akár több λ-kifejezés is megfeleltethető, a típusok értékei, a típuskonstruktorok, a típusokon értelmezett műveletek mind az egyszerű λ-kalkulus λ-kifejezései.

A λ-kifejezésekkel definiált konstansokkal, műveletekkel a kifejezések kiértékelése, azaz a normál forma meghatározása azonban kissé nehézkes. A λ-kalkulus kifejezései könnyebben és kényelmesebben írhatók és olvashatók, ha a λ-kalkulus beépített, előre definiált konstansokat, és az ezeken a konstansokon értelmezett előre definiált függvényeket tartalmaz. Ezért a továbbiakban a számkonstansokat leíró λ-kifejezéseket a reprezentált számkonstanssal, a függvényeket a szokásos műveleti jelekkel jelölhetjük. A konstansokon értelmezett függvényeket *δ-függvényeknek* nevezzük. A δ-függvények nem csak a leírást egyszerűsítik, hanem a kifejezések átalakítását is gyorsítják, mivel redukciókból, konverziókból álló sorozatok végrehajtását teszik feleslegessé. Ezeket a rövid műveleteket *δ-redukcióknak* nevezzük:

Ha egy függvényapplikációban szereplő függvény egy δ-függvény és az aktuális paraméter konstans, akkor a függvényapplikáció helyettesíthető azzal az értékkel, amelyet a függvény a paraméterrel megadott pontban felvesz.

Így jutunk el a *konstansos λ-kalkulushoz*, az egyszerű λ-kalkulust tehát a konstan-

sokkal bővítjük:

⟨*konstans*⟩ ::= ⟨*számkonstans*⟩
| ⟨*logikai konstans*⟩
| ⟨*δ-függvény*⟩,

a konstansos λ-kalkulus kifejezései tehát a következők:

⟨*λ-kifejezés*⟩ ::= ⟨*változó*⟩
| ⟨*konstans*⟩
| ⟨*λ-absztrakció*⟩
| ⟨*applikáció*⟩
⟨*λ-absztrakció*⟩ ::= (λ⟨*változó*⟩ . ⟨*λ-kifejezés*⟩)
⟨*applikáció*⟩ ::= (⟨*λ-kifejezés*⟩⟨*λ-kifejezés*⟩)

2.2. A kombinátor logika

A Bevezetésben utaltunk rá, hogy a λ-kalkulus és a kombinátor logika ekvivalens, tehát a kombinátor logikára is ugyanazokat a fő eredményeket kapjuk, mint a λ-kalkulusra. A kombinátor logika kifejezéseinek szintaktikája a következő:

Tegyük fel, hogy adott egy nem feltétlenül véges, egymástól páronként különböző változókat (szimbólumokat), a K *és* S *konstansokat, valamint a nyitó- és csukózárójelet tartalmazó halmaz. Ezen a halmazon, mint ábécén értelmezett kombinátor logikai kifejezések a következő szavak:*

⟨*kifejezés*⟩ ::= ⟨*változó*⟩
| ⟨*konstans*⟩
| ⟨*applikáció*⟩
⟨*konstans*⟩ ::= K
| S
⟨*applikáció*⟩ ::= (⟨*kifejezés*⟩⟨*kifejezés*⟩)

A kifejezésekben a legkülső zárójelet most is elhagyhatjuk, az applikáció itt is *jobbasszociatív*, és két kifejezés szintaktikus *azonosságára* az ≡ jelet használjuk.

A kombinátor logika kifejezéseit röviden *CL-kifejezéseknek* nevezzük, és a *CL*-kifejezések halmazát $\mathcal{K}$-vel jelöljük.

Mivel a kombinátor logikában nincs absztrakció, most nem beszélhetünk szabad és kötött változókról, itt minden változó szabad.

A *helyettesítés* művelete így nagyon egyszerű:

1. $x[y := G] \quad \equiv \begin{cases} G, \text{ ha } x \equiv y, \\ x, \text{ egyébként,} \end{cases}$

2. $(EF)[y := G] \equiv (E[y := G])(F[y := G])$.

A kombinátor logika operációs szemantikája két konverziós szabályt tartalmaz, ezek a K és S kombinátorokra vonatkoznak.

$KEF \quad \rightarrow_w E,$
$SEFG \rightarrow_w EG(FG).$

Látható, hogy a redukciók csak akkor hajthatók végre, ha a K-nak kettő, S-nek pedig három argumentuma van.

A definíció azt mondja ki, hogy KEF és $SEFG$ *gyenge redukálható kifejezések*. A „gyenge” jelző arra utal, hogy ha egy *CL*-kifejezés nem redukálható, akkor előfordulhat, hogy a neki megfeleltetett λ-kifejezésnek még van redexe.

A kombinátor logikában gyakran használjuk az I identitás kombinátort, de ez az I kombinátor az SKK, vagy akár az SKE kifejezéssel is megadható.

Két *CL*-kifejezés között is definiáljuk az *egyenlőséget*, az egyenlőséget az = jellel jelöljük.

Az E és F CL-kifejezésekre $E = F$, ha

- $E \equiv F$, *vagy*
- $E \overset{*}{\leftrightarrow}_w F$.

Miután megadtuk a kombinátor logika szintaktikáját és operációs szemantikáját, megadjuk a kombinátor logika pontos definícióját.

Az egyszerű kombinátor logika az egyszerű *CL*-kifejezések közötti olyan

$E =_w F \quad (E, F \in \mathcal{K})$

egyenlőségeket tartalmaz, amelyek a következő axiómák felhasználásával bizonyíthatók:

I.i. $KEF =_w E$
I.ii. $SEFG =_w EG(FG)$
II.i. $E =_w E$
II.ii. *ha* $E =_w F$, *akkor* $F =_w E$
II.iii. *ha* $E =_w F$ *és* $F =_w G$, *akkor* $E =_w G$
II.iv. *ha* $E =_w F$, *akkor* $EG =_w FG$
II.v. *ha* $E =_w F$, *akkor* $GE =_w GF$

A λ-kalkulusban is érvényesek lesznek, természetesen az itt szereplő kifejezésekre és redukciókra az *I. Church–Rosser-tétel* és annak következményei is, itt is beszélhetünk a λ-kalkulusnál szereplő redukálási stratégiákról, és érvényes lesz a *II. Church–Rosser-tétel* is.

A kombinátor logikat bővíthetjük konstansokkal, függvények kifejezéseivel, δ-függvényekkel, δ-redukcióval, és így kapjuk a *konstansos kombinátor logikát*.

A kombinátor logikában nincs olyan függvény jellegű absztrakció, mint a λ-kalkulusban. A kombinátor logikában absztrakción a *CL*-kifejezésben levő változók absztrahálását értjük, de ezzel absztrakcióval a *CL*-kifejezést csak egy másik, az eredetivel megegyező tulajdonságokkal rendelkező alakjában írjuk fel. Az E kifejezés x szerinti absztrakciójára a $\lambda^* x\,.\,E$ jelölést használjuk.

Az Előszóban utaltunk rá, hogy a funkcionális programok leírhatók a kombinátor logika kifejezéseivel. A zárójeles absztrakciók „eltüntetik” ezekből a kifejezésekből a változókat, így az absztrakciók elvégzésével egy olyan kifejezést kapunk, amelyben csak kombinátorok és konstansok vannak. A program végrehajtása, azaz a kifejezés egyszerűsítése ezek után csak a kombinátorok gyenge redukcióinak és a δ-függvényeknek a végrehajtásait jelenti, és ezáltal az implementáció rendkívül leegyszerűsödik.

Többfajta zárójeles absztrakció ismert, ezek a felhasznált kombinátorokban különböznek egymástól.

A legegyszerűbb zárójeles absztrakció a *CL*-kifejezést az I, K és S kombinátorokból álló kifejezésre alakítja, de ezzel a módszerrel az eredményül kapott kifejezés hossza rendkívül nagy. Ha a *CL*-kifejezés hossza m és benne n változó van, akkor a kifejezés hossza $O(mn^3)$. Rövidebb kifejezést kapunk, ha új kombinátorokat vezetünk be, és ezeket is használjuk a zárójeles absztrakcióban. Az új B és C kombinátorok redukciós szabályai:

$$BEFG \rightarrow_w E(FG)$$
$$CEFG \rightarrow_w EGF$$

Ezekkel a kombinátorokkal az eredmény hossza $O(mn^2)$. Még rövidebb kifejezést kapunk, ha a zárójeles absztrakcióban az S', B' és C' kombinátorokat is alkalmazzuk:

$$S'CEFG \rightarrow_w C(EG)(FG) \qquad C \in \mathcal{K}^0$$
$$B'CEFG \rightarrow_w CE(FG) \qquad C \in \mathcal{K}^0$$
$$C'CEFG \rightarrow_w C(EG)F \qquad C \in \mathcal{K}^0$$

ahol $\mathcal{K}^0$ az olyan kifejezések halmaza, amelyekben nincs változó, azaz C egy kombinátor. Ezeknek a kombinátoroknak a használatával az eredményül kapott kifejezés hossza $\Theta(mn)$ lesz.

Egy ilyen zárójeles absztrakció [2]-ben a λ_4^* jellel jelölt transzformáció, amit most csak egyszerűen λ^*-gal jelölünk.

A zárójeles absztrakció a következő:

$$
\begin{array}{lll}
\lambda^*x.x & \equiv I & \\
\lambda^*x.E & \equiv KE, & \textit{ha } x \notin FV(E), \\
\lambda^*x.Ex & \equiv E, & \textit{ha } x \notin FV(E), \\
\lambda^*x.EFG & \equiv B'EF(\lambda^*x.G), & \textit{ha } x \notin \{FV(E) \cup FV(F)\}, \\
\lambda^*x.EFG & \equiv C'E(\lambda^*x.F)G, & \textit{ha } x \notin \{FV(E) \cup FV(G)\}, \\
\lambda^*x.EFG & \equiv S'E(\lambda^*x.F)(\lambda^*x.G), & \textit{ha } x \notin FV(E), \\
\lambda^*x.EF & \equiv BE(\lambda^*x.F), & \textit{ha } x \notin FV(E), \\
\lambda^*x.EF & \equiv C(\lambda^*x.E)F, & \textit{ha } x \notin FV(F), \\
\lambda^*x.EF & \equiv S(\lambda^*x.E)(\lambda^*x.F) & \textit{egyébként}.
\end{array}
$$

Összefoglalva, a kombinátor logikában a következő kombinátorokat és gyenge redukcióikat fogjuk használni:

$$
\begin{array}{lll}
IE & \rightarrow_w E & \\
KEF & \rightarrow_w E & \\
SEFG & \rightarrow_w EG(FG) & \\
BEFG & \rightarrow_w E(FG) & \\
CEFG & \rightarrow_w EGF & \\
S'CEFG & \rightarrow_w C(EG)(FG) & C \in \mathcal{K}^0 \\
B'CEFG & \rightarrow_w CE(FG) & C \in \mathcal{K}^0 \\
C'CEFG & \rightarrow_w C(EG)F & C \in \mathcal{K}^0
\end{array}
$$

2.3. A kibővített λ-kalkulus

A *konstansos λ-kalkulus* kifejezéseit úgy adtuk meg úgy, hogy az *egyszerű λ-kalkulust* konstansokkal és δ-függvényekkel bővítettük. Módszerünk az volt, hogy az egyszerű λ-kalkulus egyes kifejezéseit elneveztük konstansoknak, adatszerkezeteknek és rajtuk értelmezett műveleteknek, függvényeknek, és ezeket a λ-kifejezéseket a szokásos matematikai jeleikkel jelöltük. A műveletek voltak a δ-függvények, és a műveletek eredményét kiszámító redukciók a δ-redukciók. Ennek a bővítésnek a célja a kifejezések egyszerűbb leírása és a redukciók egyszerűbb végrehajtása volt.

A konstansos λ-kalkulust a funkcionális programok implementációjának egyszerűbb megvalósítása érdekében tovább bővítjük, és majd csak később mutatjuk

meg, hogy a bővítések hogyan alakíthatók át az eredeti, azaz a konstansos λ-kalkulus kifejezéseire.

A konstansos λ-kalkulus *változó*, *konstans* és *applikáció* kifejezései változatlanok maradnak. A kibővített konstansos λ-kalkulusban bevezetjük azonban a *minta* fogalmát. A minta lehet változó, konstans, vagy egy összetett minta. Az összetett minta egy *konstruktorból* és a konstruktor aritásának megfelelő számú mintából áll, ezek a minták a konstruktor paraméterei lesznek.

A függvények megadásánál megkülönböztetünk *összegtípusú* és *szorzattípusú konstruktorokat*. Mint majd látni fogjuk, ha egy függvényt egy mintával adunk meg, és ez a minta konstruktort tartalmaz, akkor ezt a konstruktort szorzattípusú mintának nevezzük, míg ha a függvény definíciójában több olyan minta is van, amelyik konstruktort tartalmaz, akkor ezek a konstruktorok lesznek az összegtípusú konstruktorok.

A kibővített λ-kalkulusban a λ-absztrakciók lényegesen megváltoznak, egy absztrakció változójának helyére *mintát* is írhatunk.

Új kifejezéseket is bevezetünk, a *let-*, *letrec-* és *case-kifejezéseket*, ezeket és tulajdonságaikat majd az első előfordulásukkor részletesen kifejtjük.

A mintaillesztések kezelésére a λ-kalkulust egy új operátorral, a *vastagvonal* operátorral is bővítjük, ennek szerepe majd a sikertelen mintaillesztésnél lesz. Ez az operátor fog majd arra szolgálni, hogy egy sikertelen mintaillesztés után a program a következő minta illesztését kísérelje meg.

2.3.1. Definíció. **A kibővített λ-kalkulus λ-kifejezései:**

$$
\begin{aligned}
\langle \lambda\text{-kifejezés} \rangle \;::=\; & \langle \text{változó} \rangle \\
\mid\; & \langle \text{konstans} \rangle \\
\mid\; & \langle \lambda\text{-kifejezés} \rangle \langle \lambda\text{-kifejezés} \rangle \\
\mid\; & \lambda \langle \text{minta} \rangle \,.\, \langle \lambda\text{-kifejezés} \rangle \\
\mid\; & \textit{let}\ \langle \text{minta} \rangle_1 = \langle \lambda\text{-kifejezés} \rangle_1 \\
& \quad \ldots \\
& \quad \langle \text{minta} \rangle_{n_1} = \langle \lambda\text{-kifejezés} \rangle_{n_1} \\
& \textit{in}\ \langle \lambda\text{-kifejezés} \rangle \\
\mid\; & \textit{letrec}\ \langle \text{minta} \rangle_1 = \langle \lambda\text{-kifejezés} \rangle_1 \\
& \quad \ldots \\
& \quad \langle \text{minta} \rangle_{n_2} = \langle \lambda\text{-kifejezés} \rangle_{n_2} \\
& \textit{in}\ \langle \lambda\text{-kifejezés} \rangle
\end{aligned}
$$

	|	*case* ⟨*változó*⟩ *of*
		⟨*minta*⟩$_1$: ⟨*λ-kifejezés*⟩$_1$
		...
		⟨*minta*⟩$_{n_3}$: ⟨*λ-kifejezés*⟩$_{n_3}$
	|	⟨*λ-kifejezés*⟩ [] ⟨*λ-kifejezés*⟩
⟨*minta*⟩	::=	⟨*változó*⟩
	|	⟨*konstans*⟩
	|	⟨*konstruktor*⟩ ⟨*minta*⟩$_1$ ... ⟨*minta*⟩$_{n_4}$ $(l \geq 0)$

ahol $n_1, n_2, n_3 \geq 1$ *és* $n_4 \geq 0$.

A definícióban a kifejezések külső zárójeleit nem tettük ki, mivel a kifejezések szerkezetét meghatározó precedenciákat már korábban megadtuk.

2.4. A funkcionális program

Ebben a szakaszban megadjuk azt a programnyelvet, aminek a fordítását a jegyzetben elemezni fogjuk. Nem egy konkrét funkcionális programnyelvről lesz szó, hanem kiemeljük a programnyelvek közös tulajdonságait, és ezekkel foglalkozunk. Az egyes programnyelvek speciális elemeinek fordítását, ezek vizsgálatát az olvasónak tűzzük ki feladatul.

A következő definícióban a funkcionális programok szerkezetének leírása nem teljes, csupán a jegyzetben vizsgált, a fordítás szempontjából lényeges részeket adjuk meg.

2.4.1. Definíció. **A funkcionális programnyelv:**

⟨*program*⟩	::=	⟨*definíció*⟩$_1$ ... ⟨*definíció*⟩$_{n_1}$ ⟨*kifejezés*⟩
⟨*definíció*⟩	::=	⟨*változó-def*⟩
	|	⟨*függvény-def*⟩
⟨*változó-def*⟩	::=	⟨*változó*⟩ = ⟨*kifejezés*⟩
⟨*függvény-def*⟩	::=	⟨*egysoros-fdef*⟩
	|	⟨*többsoros-fdef*⟩
	|	⟨*lokális-fdef*⟩
⟨*egysoros-fdef*⟩	::=	⟨ *függvény*⟩⟨*minta*⟩$_1$ ... ⟨*minta*⟩$_{n_2}$ = ⟨*kifejezés*⟩ ⟨*őrfeltétel*⟩
⟨*többsoros-fdef*⟩	::=	⟨*egysoros-fdef*⟩$_1$ ... ⟨*egysoros-fdef*⟩$_{n_3}$

$\langle$*minta*$\rangle$ ::= $\langle$*változó*$\rangle$
| $\langle$*konstans*$\rangle$
| $\langle$*konstruktor*$\rangle$ $\langle$*minta*$\rangle_1 \ldots \langle$*minta*$\rangle_{n_4}$

$\langle$*őrfeltétel*$\rangle$ ::= '|' $\langle$*kifejezés*$\rangle$
| ε

$\langle$*lokális-fdef*$\rangle$ ::= $\langle$*függvény-def*$\rangle$ where $\langle$*definíció*$\rangle_1 \ldots \langle$*definíció*$\rangle_{n_6}$

$\langle$*kifejezés*$\rangle$ ::= $\langle$*változó*$\rangle$
| $\langle$*konstans*$\rangle$
| $\langle$*applikáció*$\rangle$
| $\langle$*lista-kifejezés*$\rangle$

$\langle$*konstans*$\rangle$ ::= $\langle$*számkonstans*$\rangle$
| $\langle$*logikai konstans*$\rangle$
| $\langle$*művelet jele vagy neve*$\rangle$

$\langle$*applikáció*$\rangle$ ::= $\langle$*kifejezés*$\rangle$ $\langle$*kifejezés*$\rangle$
| $\langle$*kifejezés*$\rangle$ ρ $\langle$*kifejezés*$\rangle$

ρ ::= $\langle$*infix művelet jele vagy neve*$\rangle$

...

$\langle$*lista-kifejezés*$\rangle$ ::= [$\langle$*kifejezés*$\rangle$ | $\langle$*minősítő*$\rangle_1$; ... ; $\langle$*minősítő*$\rangle_{n_5}$]

$\langle$*minősítő*$\rangle$ ::= $\langle$*generátor*$\rangle$
| $\langle$*szűrő*$\rangle$

$\langle$*generátor*$\rangle$ ::= $\langle$*változó*$\rangle$ <– $\langle$*lista*$\rangle$

$\langle$*szűrő*$\rangle$:= $\langle$*Bool értékű kifejezés*$\rangle$

...

ahol $n_1, n_2, n_4, n_5 \geq 0$, $n_6 \geq 1$ *és* $n_3 \geq 2$.

3. FEJEZET

Programok átalakítása a kibővített λ-kalkulusba

Egy funkcionális program két fő részből áll. Az első a *definíciós rész*, a második rész a kiértékelendő *kezdeti kifejezés*. A program a következő alakban írható fel:

$\langle$*definíció*$\rangle_1$
...
$\langle$*definíció*$\rangle_n$
$\langle$*kezdeti kifejezés*$\rangle$ $\quad (n \geq 0)$

A programról feltesszük, hogy típusosan helyes, azaz a típusellenőrzés vagy a típusok meghatározása már hibajelzés nélkül megtörtént.

A definíciós rész definícióinak a kibővített λ-kalkulusba történő átalakítását végezze egy TD függvény, a kezdeti kifejezés átalakítását pedig egy TE függvény. A TD és TE függvények argumentumait a speciális $(\!|$ és $|\!)$ zárójelek közé tesszük. Jelöljük a definíciókat D_i-vel ($1 \leq i \leq n$), a kezdeti kifejezést E-vel. A definíciókban a rekurzió-mentességet nem követeljük meg, ezért a fenti program kibővített λ-kalkulusba történő átalakítása egy *letrec-kifejezés* lesz:

$$
\begin{array}{lcl}
D_1 & & \\
\ldots & & \\
D_n & & \\
E & \Longrightarrow & \mathit{letrec}\ \mathrm{TD}(\!| D_1 |\!) \\
 & & \quad \ldots \\
 & & \quad \mathrm{TD}(\!| D_n |\!) \\
 & & \mathit{in}\ \mathrm{TE}(\!| E |\!)
\end{array}
$$

3.1. A kifejezések átalakítása

A TE függvény a funkcionális program egy kifejezését alakítja át λ-kifejezésre. Használható a funkcionális program kezdeti kifejezésének átalakítására, valamint a definíciós rész átalakításával kapott egyes kifejezésekre.

A függvényt a kifejezés szerkezete szerint adjuk meg. Először a változókkal és konstansokkal, valamint az applikációval foglalkozunk. Az átalakítással kapott let- és letrec-kifejezéseket a 3.4. és 3.3. szakaszokban tárgyaljuk.

3.1.1. Konstansok, konstansokon értelmezett műveletek

Az a kalkulus, amelynek kifejezéseire a funkcionális programot végül is lefordítjuk, a konstansos típusnélküli λ-kalkulus. Ezért a programban a szokásos matematikai jelölésekkel hivatkozhatunk a *konstansokra* és az értelmezett műveleteket is a megszokott matematikai jelükkel jelölhetjük. A műveletek közé bevehetjük a konstruktorokat is, ezeket a műveletek nevéhez hasonlóan az eredeti nevükkel vagy jelükkel jelölhetjük. Így az átalakítás rendkívül egyszerű.

$\mathrm{TE}(\!\lvert k \rvert\!)$	$\Longrightarrow$	k,	*ahol k egy konstans*
$\mathrm{TE}(\!\lvert \rho \rvert\!)$	$\Longrightarrow$	ρ,	*ahol ρ a művelet jele vagy neve*

3.1.1. Példa. *(Konstansok átalakítása)*

$\mathrm{TE}(\!\lvert 3 \rvert\!) \Longrightarrow 3$

$\mathrm{TE}(\!\lvert + \rvert\!) \Longrightarrow +$

$\mathrm{TE}(\!\lvert \mathit{add} \rvert\!) \Longrightarrow \mathit{add}$

$\mathrm{TE}(\!\lvert \mathit{cons} \rvert\!) \Longrightarrow \mathit{cons}$

$\mathrm{TE}(\!\lvert \mathit{pair} \rvert\!) \Longrightarrow \mathit{pair}$ □

3.1.2. Változók

A funkcionális programban szereplő *változókat* is jelölhetjük a λ-kalkulusban az eredeti, a programban levő nevükkel. Nem kell a λ-kalkulus elméletében használt egybetűs változókhoz ragaszkodnunk, mert az eredeti nevekkel a λ-kalkulus kifejezései olvashatóbbak lesznek, és így az egyes változók λ-kifejezésbeli megfelelőjét könnyen felismerhetjük.

$\mathrm{TE}(\!\lvert x \rvert\!)$	$\Longrightarrow$	x,	*ahol x egy változó*

A változó fogalomba beleérthetjük a funkcionális programban megadott *függvények* nevét is, ezeket is változtatás nélkül használhatjuk a λ-kifejezésekben.

3.1.2. Példa. *(Változók átalakítása)*

$\mathrm{TE}(\!| \mathit{abc12} |\!) \implies \mathit{abc12}$
$\mathrm{TE}(\!| \mathit{alma} |\!) \implies \mathit{alma}$
$\mathrm{TE}(\!| \mathit{hatványoz} |\!) \implies \mathit{hatványoz}$
$\mathrm{TE}(\!| \mathit{zero\_pair} |\!) \implies \mathit{zero\_pair}$ □

3.1.3. Az applikáció

A funkcionális programnyelvekben két kifejezés *applikációját* a kifejezések egymásmellé írásával jelöljük, és ugyanez a jelölés használatos a λ-kalkulusban is. Az applikáció fordítása a

$$\mathrm{TE}(\!| E\ F |\!) \implies \mathrm{TE}(\!| E |\!)\ \mathrm{TE}(\!| F |\!)$$

szabállyal adható meg.

A funkcionális programokban két kifejezésre alkalmazott művelet esetén gyakran használunk *infix* jelölésmódot. A λ-kalkulusban az első tagnak a műveletnek kell lennie, és erre a műveletre kell applikálni a kifejezéseket, mint a művelet argumentumait. Ha ρ-val jelöljük az infix műveletet, ezt az átalakítást a következő szabállyal írhatjuk le:

$$\mathrm{TE}(\!| E\ \rho\ F |\!) \implies \mathrm{TE}(\!| \rho |\!)\ \mathrm{TE}(\!| E |\!)\ \mathrm{TE}(\!| F |\!), \quad \textit{ahol } \rho \textit{ egy infix művelet}$$

3.1.3. Példa. *(Infix művelet)*

$\mathrm{TE}(\!| \mathit{alma} * 3 |\!) \implies$
$\mathrm{TE}(\!| * |\!)\ \mathrm{TE}(\!| \mathit{alma} |\!)\ \mathrm{TE}(\!| 3 |\!) \implies^{+}$
$* \mathit{alma}\ 3$ □

Kettőnél több kifejezés applikációjánál a balasszociativitás elve érvényesül.

3.1.4. Példa. *(Három kifejezés applikációja)*
Ha F nem egy infix műveletet jelöl, akkor

$\mathrm{TE}(\!| E\ F\ G |\!) \equiv$
$\mathrm{TE}(\!| (E\ F)G |\!) \implies$

$\mathrm{TE}(\!| E\ F |\!)\ \mathrm{TE}(\!| G |\!) \Longrightarrow$
$(\mathrm{TE}(\!| E |\!)\ \mathrm{TE}(\!| F |\!))\ \mathrm{TE}(\!| G |\!) \equiv$
$\mathrm{TE}(\!| E |\!)\ \mathrm{TE}(\!| F |\!)\ \mathrm{TE}(\!| G |\!)$ □

3.2. A definíciós rész átalakítása

A funkcionális programok *definíciós részében* adjuk meg a változók definícióját, és többféle módon is megadhatjuk függvények leírását: egy függvényt definiálhatunk egy kifejezéssel, vagy a definíciót megadhatjuk minták felhasználásával. Például az x változó definíciója

$x = y\ +\ 2,$

az f egyváltozós függvény definíciója

$f\ x = *\ x\ x,$

vagy egy másik függvény mintákkal:

$f\ 0 = 0$
$f\ x = +\ 1\ x$

Először az első két esetet vizsgáljuk, megnézzük, hogy ha a definíciós rész csak változóknak vagy kifejezéssel megadott függvényeknek a definícióit tartalmazza, akkor hogyan alakítható át a program a kibővített λ-kalkulus kifejezésére. Mintákkal megadott függvényekkel később, a 3.5. szakaszban foglalkozunk.

3.2.1. Változódefiníciók

A definíciós részben szerepelhetnek egyszerű *változódefiníciók*, ezeket a programnyelv megadásakor a

⟨*változó*⟩ = ⟨*kifejezés*⟩

leírással adtuk meg. Az

$x = E$

alakú változódefiníció átalakítása a következő:

$\mathrm{TD}(\!| x = E |\!) \quad \Longrightarrow \quad x = \mathrm{TE}(\!| E |\!)$

3.2.1. Példa. *(Az $x = 3 + 4$ definíció)*
Legyen a funkcionális program definíciós részének egy sora

$x = 3 + 4,$

ekkor

$\mathrm{TD}(\!| x = 3 + 4 |\!) \Longrightarrow$
$x = \mathrm{TE}(\!| 3 + 4 |\!) \Longrightarrow$
$x = \mathrm{TE}(\!| + |\!)\mathrm{TE}(\!| 3 |\!)\ \mathrm{TE}(\!| 4 |\!) \Longrightarrow^{+}$
$x = + 3\ 4$ □

3.2.2. Egy- és többváltozós függvények definíciói

Egy *egyváltozós függvény* a funkcionális program definíciós részében a

$$\langle \textit{függvény} \rangle \langle \textit{változó} \rangle = \langle \textit{kifejezés} \rangle$$

formában írható le, azaz például egy f függvény definíciója

$f\,x = E$

alakú. Ez pedig nyilvánvalóan ekvivalens a λ-kalkulus $\lambda x\,.\,\mathrm{TE}(\!| E |\!)$ absztrakciójával, így a

$$\mathrm{TD}(\!| f\ x = E |\!) \quad \Longrightarrow \quad f = \lambda x\,.\,\mathrm{TE}(\!| E |\!)$$

átalakítást kapjuk.

Ezt megismerve, a *többváltozós függvények* átalakítása már nem lesz probléma. A definíciós részben az n-változós függvények megadása

$$\langle \textit{függvény} \rangle \langle \textit{változó}_1 \rangle \langle \textit{változó}_2 \rangle \ldots \langle \textit{változó}_n \rangle = \langle \textit{kifejezés} \rangle \qquad (n > 1)$$

volt, azaz egy f függvény

$f\,x_1\,x_2 \ldots x_n = E$

alakú, ahol $x_1, x_2, \ldots, x_n$ a függvény változói. A függvénydefiníció átalakítási szabálya a következő:

$$\mathrm{TD}(\!| f\ x_1 x_2 \ldots x_n = E |\!) \quad \Longrightarrow \quad f = \lambda x_1 x_2 \ldots x_n\,.\,\mathrm{TE}(\!| E |\!)$$

Megjegyezzük, hogy a 3.2.1. pontban szereplő változódefinícióval megadott változó egy nulla aritású függvénynek is tekinthető.

3.2.2. Példa. *(Az $f\ x = 2 * x$ függvény)*

Legyen a függvénydefiníció

$f\ x = 2 * x,$

ekkor

$\mathrm{TD}(\!|\ f\ x = 2 * x\ |\!) \Longrightarrow$
$f = \lambda x\,.\,\mathrm{TE}(\!|\ 2 * x\ |\!) \Longrightarrow^{+}$
$f = \lambda x\,.\ *\ 2\ x$ □

3.2.3. Példa. *(Egy háromváltozós függvény)*

Legyen adott az

$f(a,b,c) = \sqrt{(b^2 - 4ac)/2}$

függvény. Ha a négyzetgyökvonást az *sqrt* függvény végzi és a hatványozás jele $\wedge$, a függvény alakja a funkcionális nyelvünkben a következő:

$f\ a\ b\ c = sqrt\ (((b\ \wedge\ 2) - (4 * a * c))\ /\ 2)$

ekkor

$\mathrm{TD}(\!|\ f\ a\ b\ c) = sqrt\ (((b\ \wedge\ 2) - (4 * a * c))\ /\ 2)\ |\!) \Longrightarrow^{+}$
$f = \lambda abc\,.\,sqrt\ (/\ \mathrm{TE}(\!|\ (b\ \wedge\ 2) - (4 * a * c)\ |\!)\ \mathrm{TE}(\!|\ 2\ |\!)) \Longrightarrow^{+}$
$f = \lambda abc\,.\,sqrt\ (/\ (-\ (\wedge\ b\ 2)\ (*\ (*\ 4\ a)\ c))\ 2)$ □

3.2.4. Példa. *(A TE és a TD alkalmazása)*

Tekintsük a következő funkcionális programot:

átlag a b = $(a + b)/2$
átlag $2(3 + 5)$

A program λ-kifejezése

letrec $\mathrm{TD}(\!|$ *átlag a b* = $(a + b)/2\ |\!)$
in $\mathrm{TE}(\!|$ *átlag* $2(3 + 5)\ |\!)$

A definíciós rész átalakítása:

$\mathrm{TD}(\!|\, \textit{átlag}\ a\ b = (a+b)/2 \,|\!) \Longrightarrow$
$\textit{átlag} = \lambda ab\,.\,\mathrm{TE}(\!|\, (a+b)/2 \,|\!) \Longrightarrow$
$\textit{átlag} = \lambda ab\,.\,\mathrm{TE}(\!|\, / \,|\!)\ \mathrm{TE}(\!|\, a+b \,|\!)\ \mathrm{TE}(\!|\, 2 \,|\!) \Longrightarrow$
$\textit{átlag} = \lambda ab\,.\,/\ (\mathrm{TE}(\!|\, + \,|\!)\ \mathrm{TE}(\!|\, a \,|\!)\ \mathrm{TE}(\!|\, b \,|\!))\ \mathrm{TE}(\!|\, 2 \,|\!) \Longrightarrow^{+}$
$\textit{átlag} = \lambda ab\,.\,/\ (+\ a\ b)\ 2$

A kezdeti kifejezés pedig:

$\mathrm{TE}(\!|\, \textit{átlag}\ 2(3+5) \,|\!) \Longrightarrow$
$\mathrm{TE}(\!|\, \textit{átlag} \,|\!)\ \mathrm{TE}(\!|\, 2 \,|\!)\ \mathrm{TE}(\!|\, 3+5 \,|\!) \Longrightarrow^{+}$
$\textit{átlag}\ 2\ (\mathrm{TE}(\!|\, + \,|\!)\ \mathrm{TE}(\!|\, 3 \,|\!)\ \mathrm{TE}(\!|\, 5 \,|\!)) \Longrightarrow^{+}$
$\textit{átlag}\ 2\ (+\ 3\ 5)$

Így a program λ-kifejezése:

letrec $\textit{átlag} = \lambda ab\,.\,/\ (+\ a\ b)\ 2$
in $\textit{átlag}\ 2\ (+\ 3\ 5)$

□

3.3. Az egyszerű letrec-kifejezés átalakítása let-kifejezéssé

Ha a kibővített λ-kalkulus

letrec $\langle \textit{minta} \rangle_1 = \langle \lambda\textit{-kifejezés} \rangle_1$
$\langle \textit{minta} \rangle_2 = \langle \lambda\textit{-kifejezés} \rangle_2$
$\ldots$
$\langle \textit{minta} \rangle_n = \langle \lambda\textit{-kifejezés} \rangle_n$
in $\langle \lambda\textit{-kifejezés} \rangle$

kifejezésében a $\langle \textit{minta} \rangle_i$ helyett $\langle \textit{változó} \rangle_i$ szerepel ($1 \leq i \leq n$), akkor a kifejezést *egyszerű letrec-kifejezésnek* nevezzük. A

letrec $x_1 = E_1$
$x_2 = E_2$
$\ldots$
$x_n = E_n$
in F

kifejezésben x_i-k az egyszerű letrec-kifejezés *változói*, F a kifejezés *törzse*, és E_i-t a x_i *definíciójának* nevezzük. A x_i változók *hatáskörei* az E_i kifejezések és az F kifejezés.

Ebben a szakaszban azt vizsgáljuk meg, hogy hogyan lehet az egyszerű letrec-kifejezést egyszerű let-kifejezésre átalakítani.

3.3.1. Egydefiníciós egyszerű letrec-kifejezés átalakítása λ-kifejezéssé

Először nézzük meg azt az esetet, amikor az egyszerű letrec-kifejezésben csak *egy definíció* van, azaz a kifejezés

letrec $x = E$
in F

alakú. A célunk az, hogy a letrec-kifejezést let-kifejezéssé alakítsuk át.

Ha a letrec-kifejezésben az x változó csak az F törzsben szerepel, akkor a kifejezésben nincs *rekurzió*, és a letrec-kifejezés helyett let-kifejezés is használható.

letrec $x = E$
in F $\Longrightarrow$ *let* $x = E$
in F, *ha* $x = E$*-ben nincs rekurzió*

3.3.1. Példa. *(Letrec-kifejezés helyett let-kifejezés)*
A 3.2.4. példában eredményül a

letrec *átlag* $= \lambda ab\,.\,/\ (+\ a\ b)\ 2$
in *átlag* $2\ (+\ 3\ 5)$

kifejezést kaptuk. Látható, hogy a kifejezésben nincs rekurzió, ezért a kifejezés let-kifejezés alakjában is felírható:

let *átlag* $= \lambda ab\,.\,/\ (+\ a\ b)\ 2$
in *átlag* $2\ (+\ 3\ 5)$ □

A *letrec* $x = E$ *in* F kifejezés $x = E$ részében levő rekurziót a λ-kalkulusból már jól ismert módszerrel meg tudjuk szüntetni. Az E kifejezést $\cdots x \cdots$ formában írva, az

$$x = \cdots x \cdots$$

rekurzív egyenlet jobb oldalára hajtsunk végre egy x-szerinti absztrakciót, és erre az absztrakcióra applikáljuk az x változót:

$$x = (\lambda x\,.\,\cdots x \cdots)\ x$$

Ebből az egyenletből az látható, hogy az x a λ-absztrakció fixpontja, azaz x ebből az absztrakcióból egy fixpont-kombinátorral, például az Y-nal kiszámítható:

$$x = \mathsf{Y}\,(\lambda x\,.\,\cdots x\cdots)$$
$$= \mathsf{Y}\,(\lambda x\,.\,E)$$

Ennek felhasználásával az eredeti rekurzív letrec-kifejezés definíciójában a rekurziót megszüntettük, azaz a rekurzió nélküli

$$\textit{letrec}\ x = \mathsf{Y}\,(\lambda x\,.\,E)$$
$$\textit{in}\ F$$

kifejezést kapjuk, amelyre alkalmazhatjuk a már ismert átalakítást:

$$\textit{letrec}\ x = E \quad \textit{in}\ F \quad \Longrightarrow \quad \textit{let}\ x = \mathsf{Y}\,(\lambda x\,.\,E)\ \textit{in}\ F$$

3.3.2. Példa. *(A faktoriális)*
A $fac = \lambda n\,.\,\textit{if}\,(=\ n\ 0)\ 1\ (fac\ (-\ n\ 1))$ faktoriális függvény egy alkalmazása legyen a következő:

$$\textit{letrec}\ fac = \lambda n\,.\,\textit{if}\,(=\ n\ 0)\ 1\ (fac\ (-\ n\ 1))$$
$$\textit{in}\ fac\ 2$$

Látható, hogy a kifejezésben van rekurzió, tehát a *letrec* helyett nem írható *let*. De mivel csak egy definíció van benne, erre az egyszerű letrec-kifejezésre alkalmazható a fenti átalakítás:

$$\textit{let}\ fac = \mathsf{Y}\,(\lambda fac\,.\,\lambda n\,.\,\textit{if}\,(=\ n\ 0)\ 1\ (fac\ (-\ n\ 1)))$$
$$\textit{in}\ fac\ 2$$

□

3.4. Egyszerű let-kifejezés átalakítása λ-kifejezéssé

Ha a kibővített λ-kalkulus

$$\textit{let}\ \langle \textit{minta} \rangle = \langle \lambda\textit{-kifejezés} \rangle$$
$$\textit{in}\ \langle \lambda\textit{-kifejezés} \rangle$$

kifejezésében a $\langle \textit{minta} \rangle$ helyett $\langle \textit{változó} \rangle$ szerepel, akkor a kifejezést *egyszerű let-kifejezésnek* nevezzük. A

let $x = E$
in F

kifejezésben x a let-kifejezés *változója*, F a *törzse*, és E-t az x *definíciójának* nevezzük. Az x változó *hatásköre* az F kifejezés.

A *let-kifejezés* azt jelenti, hogy a kifejezés törzsében a változót a változó definíciójával kell helyettesíteni, azaz

let $x = E$ *in* F $\quad\rightarrow\quad F[x := E]$

3.4.1. Példa. *(A 3.3.2. példa folytatása)*
A 3.3.2. példában láttuk, hogy

letrec $fac = \lambda n\,.\,$ *if* $(= n\ 0)\ 1\ (fac\ (-\ n\ 1))$
in $fac\ 2$
$\Longrightarrow$
let $fac = \mathsf{Y}\ (\lambda fac\,.\,\lambda n\,.\,$ *if* $(= n\ 0)\ 1\ (fac\ (-\ n\ 1)))$
in $fac\ 2$

végrehajtva az előírt helyettesítést,

$(\mathsf{Y}\ (\lambda fac\,.\,\lambda n\,.\,$ *if* $(= n\ 0)\ 1\ (fac\ (-\ n\ 1))))\ 2$

aminek a megoldása a λ-kalkulusból már jól ismert ([2] 92. oldal). □

3.4.2. Példa. *(A 3.3.1. példa folytatása)*
A fenti átalakítással is meg tudjuk határozni a

let $átlag = \lambda ab\,.\,/\ (+\ a\ b)\ 2$
in $átlag\ 2\ (+\ 3\ 5)$

kifejezés értékét is.

let $átlag = \lambda ab\,.\,/\ (+\ a\ b)\ 2$
in $átlag\ 2\ (+\ 3\ 5)$
$\rightarrow$
$(\lambda ab\,.\,/\ (+\ a\ b)\ 2)\ 2\ (+\ 3\ 5)\ \rightarrow^{+}$
$/\ (+\ 2\ (+\ 3\ 5))\ 2\ \rightarrow^{+}$
5 □

Eddig olyan funkcionális programokat néztünk, amelyekből a megismert átalakításokkal egydefiníciós egyszerű let-kifejezéseket kaptunk. Most nézzük meg ezeknek a let-kifejezéseknek két általánosítását.

3.4.1. Skatulyázott egydefiníciós egyszerű let-kifejezések

Az egydefiníciós egyszerű let-kifejezések *skatulyázhatók*, azaz a *let* $v = E$ *in* F kifejezésben az F maga is lehet egy egyszerű let-kifejezés.

Mindegyik let-kifejezés azt jelenti, hogy a kifejezés saját törzsében kell a saját változóját a változó definíciójával helyettesíteni, és mivel a különböző let-kifejezések változói között semmilyen kapcsolat nincs (nincs rekurzió sem), a helyettesítés sorrendje közömbös. A skatulyázott let-kifejezés értékének meghatározásakor ezért rendkívül fontos a let-kifejezések változóinak hatáskörét szem előtt tartani, hiszen előfordulhat, hogy több let-kifejezés változójának neve azonos.

let $x_1 = E_1$
in (*let* $x_2 = E_2$
in (...
(*let* $x_n = E_n$
in F) ...)) $\rightarrow$ $F[x_1 := E_1][x_2 := E_2] \ldots [x_n := E_n]$

3.4.3. Példa. *(Skatulyázott egyszerű let-kifejezés)*
Határozzuk meg a

let $x = 2$
in (*let* $y = 3$
in $(+ (* x\ y)\ (* y\ x))$)

kifejezés értékét. Végezzük el külön-külön a helyettesítéseket. Először az y, majd utána az x helyettesítéseit:

let $x = 2$
in (*let* $y = 3$
in $(+ (* x\ y)\ (* y\ x))$)
$\rightarrow$
let $x = 2$
in $(+ (* x\ 3)\ (* 3\ x))$
$\rightarrow$

$+\ (*\ 2\ 3)\ (*\ 3\ 2)\ \rightarrow^{+}$
$+\ 6\ 6\ \rightarrow$
12

Ha a változók helyettesítését a fordított sorrendben hajtjuk végre:

let $x = 2$
in (*let* $y = 3$
in $(+\ (*\ x\ y)\ (*\ y\ x))$)
$\rightarrow$
let $y = 3$
in $(+\ (*\ 2\ y)\ (*\ y\ 2))$
$\rightarrow$
$+\ (*\ 2\ 3)\ (*\ 3\ 2)\ \rightarrow^{+}$
$+\ 6\ 6\ \rightarrow$
12

A fenti átalakítási képletet alkalmazva:

let $x = 2$
in (*let* $y = 3$
in $(+\ (*\ x\ y)\ (*\ y\ x))$)
$\rightarrow$
$(+\ (*\ x\ y)\ (*\ y\ x))[x := 2][y := 3]\ \rightarrow$
$+\ (*\ 2\ 3)\ (*\ 3\ 2)\ \rightarrow^{+}$
12 □

3.4.2. Többdefiníciós egyszerű let-kifejezések

Az egyszerű let-kifejezések skatulyázhatósága és a skatulyázott egyszerű let-kifejezések szemantikája lehetőséget ad arra, hogy a többdefiníciós egyszerű let-kifejezéseket skatulyázott egydefiníciós egyszerű let-kifejezésre alakítsuk át.

A többdefiníciós egyszerű let-kifejezések alakja a kibővített λ-kalkulusban:

let $\langle$*változó*$\rangle_1 = \langle\lambda$*-kifejezés*$\rangle_1$
$\langle$*változó*$\rangle_2 = \langle\lambda$*-kifejezés*$\rangle_2$
...
$\langle$*változó*$\rangle_n = \langle\lambda$*-kifejezés*$\rangle_n$
in $\langle\lambda$*-kifejezés*$\rangle$

Legyen a többdefiníciós egyszerű let-kifejezés

$$\begin{aligned}&\textit{let } x_1 = E_1\\ &\quad x_2 = E_2\\ &\quad \ldots\\ &\quad x_n = E_n\\ &\textit{in } E,\end{aligned}$$

ennek átalakítása

$$\begin{aligned}&\textit{let } x_1 = E_1\\ &\quad x_2 = E_2\\ &\quad \ldots\\ &\quad x_n = E_n\\ &\textit{in } E \quad \Longrightarrow \quad \textit{let } x_1 = E_1\\ &\qquad\qquad\qquad\quad \textit{in } (\ \textit{let } x_2 = E_2\\ &\qquad\qquad\qquad\qquad \textit{in } (\ \ldots\\ &\qquad\qquad\qquad\qquad\quad (\ \textit{let } x_n = E_n\\ &\qquad\qquad\qquad\qquad\qquad \textit{in } E\ )\ \ldots))\end{aligned}$$

Ezek alapján a továbbiakban nem is kell foglalkoznunk a többdefiníciós egyszerű let-kifejezésekkel, hiszen a fenti átalakítás minden esetben elvégezhető, és az eredményül kapott átalakított kifejezésben csak egydefiníciós egyszerű let-kifejezések szerepelnek.

3.4.4. Példa. *(Többdefiníciós egyszerű let-kifejezés)*

Alakítsuk át a

$$\begin{aligned}&\textit{let } x = 2\\ &\quad y = 3\\ &\textit{in } (+\ (*\ x\ y)\ (*\ y\ x)\end{aligned}$$

kifejezést skatulyázott let-kifejezésre. Látható, hogy eredményül az előző példában szereplő skatulyázott let-kifejezést kapjuk:

$$\begin{aligned}&\textit{let } x = 2\\ &\textit{in } (\ \textit{let } y = 3\\ &\qquad \textit{in } (+\ (*\ x\ y)\ (*\ y\ x))\ )\end{aligned}$$

□

3.4.3. Az egyszerű let-kifejezés λ-kifejezése

Láttuk, a *let-kifejezés* azt jelenti, hogy a kifejezés törzsében a változót a változó definíciójával kell helyettesíteni, azaz

$$\begin{array}{ll} \textit{let } x = E & \\ \textit{in } F & \rightarrow F[x := E]. \end{array}$$

Ezt az eredményt egy β-redukcióval a $(\lambda x\,.\,F)E$ kifejezésből is megkapjuk, ezért ebből azonnal adódik, hogy az egyszerű let-kifejezés a λ-kalkulus egy ilyen függvényapplikációjára alakítható át, azaz

$$\begin{array}{ll} \textit{let } x = E & \\ \textit{in } F & \rightsquigarrow \quad (\lambda x\,.\,F)E \end{array}$$

Látható, hogy a let-kifejezésnek a λ-kalkulus függvényapplikációjára való átalakítása majd egy β-redukció végrehajtása egy lépéssel hosszabb, mint ha a let-kifejezést a benne megadott helyettesítéssel egyszerűsítenénk. A probléma azonban nem a végrehajtás lépésszámának a növekedése, erre a kérdésre az 5.3. szakaszban még visszatérünk.

3.4.5. Példa. *(A 3.3.2. példa folytatása)*
A 3.3.2. példában láttuk, hogy

$$\begin{array}{l} \textit{letrec } fac = \lambda n\,.\,\textit{if } (=\ n\ 0)\ 1\ (fac\ (-\ n\ 1)) \\ \quad \textit{in } fac\ 2 \\ \Longrightarrow \\ \quad \textit{let } fac = \mathsf{Y}\ (\lambda fac\,.\,\lambda n\,.\,\textit{if } (=\ n\ 0)\ 1\ (fac\ (-\ n\ 1))) \\ \quad \textit{in } fac\ 2 \end{array}$$

Ezt a kifejezést a λ-kalkulus kifejezésére alakítva, majd egy β-redukciót végrehajtva:

$$\begin{array}{l} (\lambda fac\,.\,(fac\ 2))(\mathsf{Y}\ (\lambda fac\,.\,\lambda n\,.\,\textit{if } (=\ n\ 0)\ 1\ (fac\ (-\ n\ 1)))) \rightarrow \\ (\mathsf{Y}\ (\lambda fac\,.\,\lambda n\,.\,\textit{if } (=\ n\ 0)\ 1\ (fac\ (-\ n\ 1))))\ 2 \end{array}$$

aminek a megoldását már megadtuk. □

3.4.6. Példa. *(A 3.3.1. példa folytatása)*
A fenti átalakítással is meg tudjuk határozni a

$$\begin{array}{ll} \textit{let } átlag = \lambda ab\,.\,/\ (+\ a\ b)\ 2 & \\ \textit{in } átlag\ 2\ (+\ 3\ 5) & \end{array}$$

kifejezés értékét. A λ-absztrakció változója az „átlag” szó lesz.

let *átlag* $= \lambda ab\,.\,/\ (+\ a\ b)\ 2$
in *átlag* $2\ (+\ 3\ 5)$
$\rightsquigarrow$
$(\lambda átlag\,.\,(átlag\ 2\ (+\ 3\ 5)))(\lambda ab\,.\,/\ (+\ a\ b)\ 2) \rightarrow$
$(\lambda ab\,.\,/\ (+\ a\ b)\ 2)\ 2\ (+\ 3\ 5) \rightarrow^{+}$
$/\ (+\ 2\ (+\ 3\ 5))\ 2 \rightarrow^{+}$
5 □

A módszer természetesen a többdefiníciós egyszerű let-kifejezésekre is alkalmazható, ha először a skatulyázást végezzük el.

3.4.7. Példa. *(Többdefiníciós egyszerű let-kifejezés)*

Alakítsuk át a

let $x = 2$
$\quad y = 3$
in $(+\ (*\ x\ y)\ (*\ y\ x)$

kifejezést skatulyázott let-kifejezésre, majd határozzuk meg a kifejezés értékét.

let $x = 2$
$\quad y = 3$
in $(+\ (*\ x\ y)\ (*\ y\ x)$
$\Longrightarrow$
let $x = 2$
in (*let* $y = 3$
$\qquad$ *in* $(+\ (*\ x\ y)\ (*\ y\ x))$)
$\rightsquigarrow$
let $x = 2$
in $((\lambda y\,.\,(+\ (*\ x\ y)\ (*\ y\ x)))\ 3)$
$\rightsquigarrow$
$(\lambda x\,.\,((\lambda y\,.\,(+\ (*\ x\ y)\ (*\ y\ x)))\ 3))\ 2 \rightarrow$
$(\lambda y\,.\,(+\ (*\ 2\ y)\ (*\ y\ 2)))\ 3 \rightarrow$
$+\ (*\ 2\ 3)\ (*\ 3\ 2) \rightarrow^{+}$
12 □

3.5. Mintákkal megadott függvénydefiníciók

A funkcionális programok definíciós részében egy függvényt mintákkal is megadhatunk:

$$\langle \textit{függvény} \rangle \langle \textit{minta} \rangle_{1,1} \ldots \langle \textit{minta} \rangle_{1,n} = \langle \textit{kifejezés} \rangle_1$$
$$\ldots$$
$$\langle \textit{függvény} \rangle \langle \textit{minta} \rangle_{m,1} \ldots \langle \textit{minta} \rangle_{m,n} = \langle \textit{kifejezés} \rangle_m$$

ahol $n, m \geq 1$ *és*

$$\begin{aligned} \langle \textit{minta} \rangle ::= {} & \langle \textit{változó} \rangle \\ | \quad & \langle \textit{konstans} \rangle \\ | \quad & \langle \textit{konstruktor} \rangle \langle \textit{minta} \rangle_1 \ldots \langle \textit{minta} \rangle_k \end{aligned}$$

$k \geq 0$-*ra*.

Megjegyezzük, hogy nem minden funkcionális programnyelvben kell a definíció mindegyik sorában a függvény nevét megismételni, ilyenkor a függvény nevét csak a definíció első sora tartalmazza.

A továbbiakban a minták rövid jelölésére a *p* vagy a *q* betűt fogjuk használni és az azonos betűkkel jelölt mintákat indexekkel különböztetjük meg.

A függvény megadásának leírásából látható, hogy a mintával való megadás lehet egy- vagy többsoros, és egy konstruktorral megadott minta konstruktorának argumentumai újabb, akár konstruktoros minták is lehetnek.

Ha a függvényt *egy mintával* adjuk meg és a minta konstruktort tartalmaz, akkor ezt a konstruktort *szorzattípusú konstruktornak* vagy röviden *szorzatkonstruktornak* nevezzük.

Ha a függvény megadása *többsoros* és a mintákban konstruktorok vannak, akkor azt mondjuk, hogy a függvény megadása *összegtípusú konstruktorokkal* vagy röviden *összegkonstruktorokkal* történik.

3.5.1. Példa. *(Szorzat- és összegtípusú konstruktorok)*
Szorzattípusú konstruktor a *pair* konstruktor. Egy függvény definíciója például ezzel a konstruktorral:

first (*pair* $x\ y$) $= x$

Összegtípusúak például a bináris fát leíró *leaf* és *branch*, vagy a listát megadó *nil* és *cons* konstruktorpárok. Két függvénydefiníció:

reflect (*leaf* n) $=$ *leaf* n
reflect (*branch* $t_1\ t_2$) $=$ *branch* (*reflect* t_2) (*reflect* t_1)

és

header nil = *hiba*
header (*cons x xs*) = *x* □

3.5.1. Függvénydefiníciók egy mintával

Egy függvény *egysoros* és *egy mintával* való megadása

⟨*függvény*⟩⟨*minta*⟩ = ⟨*kifejezés*⟩

alakú. A minta természetesen itt is egy változó vagy egy konstans lehet, és ha a minta konstruktorral van megadva, akkor ez egy szorzattípusú konstruktor a megfelelő argumentumokkal.

Az egysoros és a minta helyén egy változót tartalmazó függvénydefiníció pontosan a 3.2.2. pontban leírt egyváltozós függvénydefiníciónak felel meg, és ennek az átalakítása

$$\mathrm{TD}(\!| f\ x = E |\!) \implies f = \lambda x\,.\,\mathrm{TE}(\!| E |\!)$$

volt. Ezt általánosítva, a mintát tartalmazó függvénydefinícióra a

$$\mathrm{TD}(\!| f\ p = E |\!) \implies f = \lambda \mathrm{TE}(\!| p |\!)\,.\,\mathrm{TE}(\!| E |\!)$$

átalakítást kapjuk. Látható, hogy az átalakítás a kiterjesztett λ-kalkulus egy olyan λ-absztrakcióját adja meg, amelyben az absztrakció változójának helyén egy átalakított minta van. Az ilyen absztrakciót *mintaabsztrakciónak* nevezzük.

3.5.2. Példa. *(Függvények egy mintával, első kísérlet)*

$$\mathrm{TD}(\!| f\ 0 = 0 |\!) \implies f = \lambda 0\,.\,0$$
$$\mathrm{TD}(\!| g\ 1 = 100 |\!) \implies g = \lambda 1\,.\,100$$

és a 3.5.1. példában szereplő *first* függvényre:

$$\mathrm{TD}(\!| \mathit{first}\ (\mathit{pair}\ x\ y) = x |\!) \implies \mathit{first} = \lambda(\mathit{pair}\ x\ y)\,.\,x$$ □

Mit jelent az, hogy az absztrakció változójának helyén minta van, és hogyan működik a minta illesztése? Egy mintaillesztés sikertelen is lehet, mi lesz ebben az esetben az átalakítás eredménye? Ezeket a problémákat általánosan a többsoros definícióval megadott függvényekre nézzük meg, azaz azokra, amelyeknél több mintaillesztés is lehetséges, és utána visszatérünk az $f\ p = E$ definíció pontos átalakítására.

A többsoros és minden sorban egy mintával megadott függvények alakja

$$\begin{aligned}
&\langle \textit{függvény} \rangle \langle \textit{minta} \rangle_1 \;\; = \langle \textit{kifejezés} \rangle_1 \\
&\ldots \\
&\langle \textit{függvény} \rangle \langle \textit{minta} \rangle_m = \langle \textit{kifejezés} \rangle_m \qquad (m > 1)
\end{aligned}$$

Megjegyezzük, hogy ha a sorokban levő minták konstruktorokat tartalmaznak, akkor $m = 1$ esetén ez egy szorzattípusú konstruktor, $m > 1$ esetén pedig ezeknek összegtípusú konstruktoroknak kell lenniük.

Ez a függvénymegadás azt jelenti, hogy először az első mintával kell a mintaillesztést elvégezni, ha ez sikeres, akkor a függvény értéke az első kifejezés lesz, ha nem, akkor a második mintát kell illeszteni. Ha ez sikeres, akkor a függvény értéke a második kifejezés, ha nem, akkor a harmadik mintát kell illeszteni, és így tovább. A mintaillesztések a függvényt leíró sorok sorrendjében történnek, ezt a műveletet *funkcionális kiértékelésnek* nevezzük.

Mint majd látni fogjuk (3.7. szakasz), a mintaillesztés egy olyan EF applikációval írható le, ahol E egy a változójában mintát tartalmazó absztrakció, F pedig erre a mintára illesztett kifejezés. Ha a mintaillesztés nem sikeres, akkor az EF kifejezés, azaz a mintaillesztés eredménye legyen egy *fail*-nek, „sikertelennek” nevezett konstans, és azt mondjuk, hogy ekkor az EF kifejezést a *fail* konstansra redukáljuk.

Mivel itt redukciókról van szó, a kibővített λ-kalkulus konstansai közé vezessük be a $\perp$ jellel jelölt *bottom* konstanst is, ezt a konstanst annak a jelölésére fogjuk majd használni, hogy egy kifejezésnek nincs normál formája. Az $E = \perp$ egyenlőség tehát azt jelenti, hogy az E *normál sorrendű* redukciós sorozata nem terminál.

A funkcionális kiértékelés műveletének leírásához vezettük be a 2.3. szakaszban a $[\!]$ jellel jelölt operátort, amit „vastagvonal”-nak nevezünk. Természetesen ennek a jelnek a kibővített λ-kalkulus ábécéjében is benne kell lennie. Láttuk, hogy a kibővített λ-kalkulus λ-kifejezései egy új szabállyal is bővültek:

$\langle \lambda\textit{-kifejezés} \rangle ::= \langle \lambda\textit{-kifejezés} \rangle \; [\!] \; \langle \lambda\textit{-kifejezés} \rangle$

A *vastagvonal* tehát egy infix műveletet jelöl, a művelet jelentését a következő leírással adjuk meg:

$$\begin{aligned}
&E \;\; [\!] \; F \;\; \rightarrow E, \;\; \textit{ha } E \neq \perp \textit{ és } E \neq \textit{fail}, \\
&\textit{fail} \; [\!] \; F \;\; \rightarrow F, \\
&\perp \;\; [\!] \; F \;\; \rightarrow \perp.
\end{aligned}$$

A *vastagvonal* operátor tehát először meghatározza a baloldali argumentumát, ha ez a kifejezés terminál és nem *fail*, akkor befejezi a működését és eredményül adja

ezt a kifejezést. Ha a bal oldal kiértékelése *fail*, akkor eredményül a jobboldali kifejezést kapjuk. A harmadik sorban levő szabály azt mondja ki, hogy ha a kifejezés nem terminál, akkor függetlenül az operátor jobboldalán levő kifejezéstől, a teljes kifejezés *bottom* lesz.

A *vastagvonal* művelet *jobbasszociatív*, azaz

$$E \,[\!]\, (F \,[\!]\, G) \equiv E \,[\!]\, F \,[\!]\, G,$$

így ha az E kiértékelése *fail* lesz, akkor az $F \,[\!]\, G$ kiszámítása következik.

A zárójelek nélküli

$$E_1 \,[\!]\, E_2 \,[\!]\, \ldots \,[\!]\, E_{m-1} \,[\!]\, E_m$$

kifejezésben tehát balról-jobbra értékelődnek ki a *vastagvonal* művelettel elválasztott kifejezések, egészen addig, amíg egy E_i $(1 \leq i \leq m-1)$ kifejezés nem lesz *fail*. Ha az E_{m-1} is *fail*, akkor a kifejezés értéke E_m lesz. Ezt felhasználva, vezessünk be egy új *ERROR* konstanst annak a jelzésére, hogy egyik mintaillesztés sem volt sikeres. Bővítsük az m darab mintaillesztést egy $m+1$-edik kifejezéssel, amelyik ezt az *ERROR*-t tartalmazza:

$$E_1 \,[\!]\, E_2 \,[\!]\, \ldots \,[\!]\, E_{m-1} \,[\!]\, E_m \,[\!]\, \mathit{ERROR}$$

és ez jelentse azt, hogy ha az $E_1, E_2, \ldots, E_m$ minták egyike sem illeszthető, akkor az *ERROR* eredményt, azaz hibajelzést kapunk. Az *ERROR* konstansnak természetesen nincs szerepe akkor, ha a definícióban a típus minden konstruktora szerepel, hiszen ekkor a futási időben legalább egy mintának illeszkednie kell.

Megjegyezzük, hogy λ-kalkulusban megszokottakkal ellentétben, a *vastagvonal* operátor infix, azaz az operandusai közé írandó, de ez csak a könnyebb olvashatóság miatt van, és később majd megadjuk a prefix *vastagvonal* operátor λ-kifejezés-t is.

Ezek után most már megadhatjuk az *egysoros* és *egymintás* függvénydefiníció pontos átalakítási szabályát:

$$\mathrm{TD}(\!| f\ p = E |\!) \implies f = \lambda x\,.\ (\ ((\lambda \mathrm{TE}(\!| p |\!)\,.\,\mathrm{TE}(\!| E |\!))\ x) \ [\!]\ \mathit{ERROR}\ )$$

ahol x egy új változó.

Megjegyezzük, hogy ha a fordítóprogram olyan típusellenőrzést végez, amivel ellenőrizni tudja a függvénydefiníció argumentumának és a függvényre applikált kifejezésnek a típushelyességét, akkor a fenti átalakításban szereplő vastagvonal opcióra, azaz az *ERROR*-ra nincs is szükség.

3.5.3. Példa. *(Függvények egy mintával)*

A 3.5.2. példában szereplő függvénydefiníciók pontos átalakítása a következő:

$$\mathrm{TD}(\!| f\ 0 = 0 |\!) \Longrightarrow^{+}$$
$$f = \lambda x\,.\ (\ ((\lambda 0\,.\,0)\ x)$$
$$\qquad [\!]\ \mathit{ERROR}$$
$$\qquad)$$
$$\mathrm{TD}(\!| g\ 1 = 100 |\!) \Longrightarrow^{+}$$
$$g = \lambda x\,.\ (\ ((\lambda 1\,.\,100)\ x)$$
$$\qquad [\!]\ \mathit{ERROR}$$
$$\qquad)$$

és a *first* függvényre:

$$\mathrm{TD}(\!| \mathit{first}\ (\mathit{pair}\ x\ y) = x |\!) \Longrightarrow^{+}$$
$$\mathit{first} = \lambda z\,.\ (\ ((\lambda(\mathit{pair}\ x\ y)\,.\,x)\ z)$$
$$\qquad [\!]\ \mathit{ERROR}$$
$$\qquad)$$

Megfelelő típusellenőrzést végző fordítóprogram esetén, azaz ha nincs szükség az *ERROR*-ra, a kifejezések alakja:

$$f = \lambda x\,.\,(\lambda 0\,.\,0)\ x$$
$$g = \lambda x\,.\,(\lambda 1\,.\,100)\ x$$
$$\mathit{first} = \lambda z\,.\,(\lambda(\mathit{pair}\ x\ y)\,.\,x)\ z$$ □

A *többsoros* és minden sorban *egy mintával* megadott függvények alakja

$$f\ p_1 = E_1$$
$$f\ p_2 = E_2$$
$$\ldots$$
$$f\ p_m = E_m$$

A $[\!]$ operátort használva meg tudjuk adni ennek a függvénydefiníciónak is az átalakítását:

$$\mathrm{TD}(\!| \; f\ p_1 = E_1 \quad f\ p_2 = E_2 \quad \ldots \quad f\ p_m = E_m \;|\!) \implies f = \lambda x\,.\ (\ ((\lambda \mathrm{TE}(\!| p_1 |\!)\,.\,\mathrm{TE}(\!| E_1 |\!))\ x) \ [\!]\ ((\lambda \mathrm{TE}(\!| p_2 |\!)\,.\,\mathrm{TE}(\!| E_2 |\!))\ x) \ [\!]\ \ldots \ [\!]\ ((\lambda \mathrm{TE}(\!| p_m |\!)\,.\,\mathrm{TE}(\!| E_m |\!))\ x) \ [\!]\ \mathit{ERROR}\ )$$

ahol x egy új változó.

3.5.4. Példa. *(Többsoros egymintás függvénydefiníció)*
A *flip* függvény mintákkal történő megadása legyen a következő:

$$\mathit{flip}\ 0 = 1$$
$$\mathit{flip}\ 1 = 0$$

Ez egy többsoros és egymintás függvénydefiníció, a λ-kifejezése:

$$\mathit{flip} =$$
$$\lambda x\,.\ (\ ((\lambda \mathrm{TE}(\!| 0 |\!)\,.\,\mathrm{TE}(\!| 1 |\!))x) \ [\!]\ ((\lambda \mathrm{TE}(\!| 1 |\!)\,.\,\mathrm{TE}(\!| 0 |\!))x) \ [\!]\ \mathit{ERROR}\ )$$
$$\Longrightarrow^{+}$$
$$\lambda x\,.\ (\ ((\lambda 0\,.\,1)x) \ [\!]\ ((\lambda 1\,.\,0)x) \ [\!]\ \mathit{ERROR}\ )$$

□

3.5.5. Példa. *(Többsoros egymintás függvénydefiníció)*
A 3.5.1. példában szereplő

$$\mathit{reflect}\ (\mathsf{leaf}\ n) = \mathsf{leaf}\ n$$
$$\mathit{reflect}\ (\mathsf{branch}\ t_1\ t_2) = \mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1)$$

függvény alakja a kibővített λ-kalkulusban a következő:

$$
\begin{array}{l}
\mathit{reflect} = \\
\lambda x.\ (\ ((\lambda \mathrm{TE}(\!|\ \mathsf{leaf}\ n\ |\!).\ \mathrm{TE}(\!|\ \mathsf{leaf}\ n\ |\!))\ x) \\
\quad [\!]\ ((\lambda \mathrm{TE}(\!|\ \mathsf{branch}\ t_1\ t_2\ |\!).\ \mathrm{TE}(\!|\ \mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1)\ |\!))\ x) \\
\quad [\!]\ \mathsf{ERROR} \\
\quad) \\
\Longrightarrow^{+} \\
\lambda x.\ (\ ((\lambda(\mathsf{leaf}\ n).\ \mathsf{leaf}\ n)\ x) \\
\quad [\!]\ ((\lambda(\mathsf{branch}\ t_1\ t_2).\ \mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\
\quad [\!]\ \mathsf{ERROR} \\
\quad)
\end{array}
$$

□

3.5.2. Függvénydefiníciók több mintával

Az előző ponthoz hasonlóan, most is az *egysoros* definícióval kezdjük, melynek alakja

$$\langle \mathit{függvény} \rangle \langle \mathit{minta} \rangle_1 \ldots \langle \mathit{minta} \rangle_n = \langle \mathit{kifejezés} \rangle \qquad (n > 1)$$

Ha a minták helyén változók vannak, akkor a 3.2.2. pontban leírt esetet kapjuk, amelynek az átalakítása a

$$\mathrm{TD}(\!|\ f\ x_1\ x_2\ \ldots\ x_n = E\ |\!) \Longrightarrow f = \lambda x_1 x_2 \ldots x_n.\ \mathrm{TE}(\!|\ E\ |\!)$$

volt. Ezt általánosítjuk a mintákra, az

$$f\ p_1\ p_2 \ldots p_n = E$$

alakú kifejezést kell átalakítanunk.

Mivel most is mintaillesztést kell végezni, a definíció átalakításához szükség van a vastagvonal műveletre és az *ERROR* konstansra:

$$
\begin{array}{l}
\mathrm{TD}(\!|\ f\ p_1\ p_2\ \ldots\ p_n = E\ |\!) \\
\Longrightarrow \\
f = \lambda x_1 x_2 \ldots x_n.\ (\ ((\lambda \mathrm{TE}(\!|\ p_1\ |\!).\lambda \mathrm{TE}(\!|\ p_2\ |\!).\ \cdots\ .\lambda \mathrm{TE}(\!|\ p_n\ |\!).\ \mathrm{TE}(\!|\ E\ |\!))\ x_1 x_2 \ldots x_n) \\
\qquad\qquad\qquad [\!]\ \mathsf{ERROR} \\
\qquad\qquad\qquad) \\
\textit{ahol } x_1, x_2, \ldots, x_n \textit{ új változók}
\end{array}
$$

Látható, hogy az absztrakciók változói helyén minták állnak, tehát mindegyik λp_i-re ($1 \leq i \leq n$) mintaillesztést kell végezni. Előfordulhat, hogy valamelyik

mintaillesztésre *fail* eredményt kapunk, például egy j $(1 \leq j \leq n)$ értékre

$$
\begin{array}{l}
f\ E_1 E_2 \ldots E_n \\
\rightarrow^+ \\
(\ ((\lambda \mathrm{TE}(\!| p_1 |\!) . \lambda \mathrm{TE}(\!| p_2 |\!) . \cdots . \lambda \mathrm{TE}(\!| p_n |\!) . \mathrm{TE}(\!| E |\!))\ E_1 E_2 \ldots E_n) \\
[\!]\ \mathit{ERROR} \\
) \\
\rightarrow^+ \\
(\ ((\lambda \mathrm{TE}(\!| p_j |\!) . (\lambda \mathrm{TE}(\!| p_{j+1} |\!) . \cdots . \lambda \mathrm{TE}(\!| p_n |\!) . \mathrm{TE}(\!| E |\!))\ E_j\ E_{j+1} \ldots E_n) \\
[\!]\ \mathit{ERROR} \\
) \\
\rightarrow^+ \\
(\ (\mathit{fail}\ E_{j+1} \ldots E_n) \\
[\!]\ \mathit{ERROR} \\
)
\end{array}
$$

Ha a j-edik mintaillesztés sikertelen volt, akkor a többi mintaillesztésnek már nincs értelme, tehát a *fail* $E_{j+1} \ldots E_n$ kifejezést a *fail* konstansra kell redukálni. Ehhez vezessünk be egy új redukálási szabályt, tetszőleges E kifejezésre legyen

$\mathit{fail}\ E \ \rightarrow\ \mathit{fail}$

Így a *fail* $E_{j+1} \ldots E_n \ \rightarrow^+$ *fail* valóban teljesülni fog, és az $f\ E_1 E_2 \ldots E_n$ kifejezésre ekkor az *ERROR* hibajelzést kapjuk.

3.5.6. Példa. *(Egysoros definíció több mintával)*

A függvénydefiníció legyen

$$f\ 0\ 1 = 0$$

itt az első minta a 0, a második az 1 konstans. Ekkor a definíció átalakítása

$$
\begin{array}{l}
f = \\
\lambda xy .\ (\ ((\lambda \mathrm{TE}(\!| 0 |\!) . \lambda \mathrm{TE}(\!| 1 |\!) . \mathrm{TE}(\!| 0 |\!))\ xy) \\
\qquad [\!]\ \mathit{ERROR} \\
\qquad) \\
\Longrightarrow^+
\end{array}
$$

$\lambda xy\,.\;(\;((\lambda 0\,.\,\lambda 1\,.\,0)\;xy)$
$\quad [\!]\;\mathit{ERROR}$
$\quad)$ □

A többsoros egymintás és az egysoros többmintás függvénydefiníció átalakítását megismerve, ezek kombinációjával most már meg tudjuk adni a *többsoros* és *többmintás* függvénydefiníció átalakítását is. Az ilyen függvény definíciója

$$\begin{aligned}&\langle \textit{függvény}\rangle\langle \textit{minta}\rangle_{1,1}\ldots\langle \textit{minta}\rangle_{1,n} \;=\; \langle \textit{kifejezés}\rangle_1\\ &\ldots\\ &\langle \textit{függvény}\rangle\langle \textit{minta}\rangle_{m,1}\ldots\langle \textit{minta}\rangle_{m,n} \;=\; \langle \textit{kifejezés}\rangle_m \qquad (n,m>1)\end{aligned}$$

Először tegyük fel, mint a definícióból is látszik, hogy a sorokban levő minták darabszáma azonos. A program alakja ekkor

$$\begin{aligned}&f\;p_{1,1}\;\;p_{1,2}\;\;\ldots p_{1,n}\;=E_1\\ &f\;p_{2,1}\;\;p_{2,2}\;\;\ldots p_{2,n}\;=E_2\\ &\ldots\\ &f\;p_{m,1}\;\;p_{m,2}\;\ldots p_{m,n}\;=E_m\end{aligned}$$

alakú. Ennek a programnak az átalakítása a következő:

$$\begin{aligned}&\mathrm{TD}(\!|\; f\;p_{1,1}\;\;p_{1,2}\;\;\ldots p_{1,n}\;=E_1\\ &\qquad f\;p_{2,1}\;\;p_{2,2}\;\;\ldots p_{2,n}\;=E_2\\ &\qquad\ldots\\ &\qquad f\;p_{m,1}\;\;p_{m,2}\;\ldots p_{m,n}\;=E_m\;|\!)\\ &\Longrightarrow\\ &f=\lambda x_1x_2\ldots x_n\,.\\ &\qquad(\;((\lambda\mathrm{TE}(\!|\,p_{1,1}\,|\!)\,.\,\lambda\mathrm{TE}(\!|\,p_{1,2}\,|\!)\,.\,\cdots\,.\,\lambda\mathrm{TE}(\!|\,p_{1,n}\,|\!)\,.\,\mathrm{TE}(\!|\,E_1\,|\!))\;x_1x_2\ldots x_n)\\ &\qquad[\!]\;((\lambda\mathrm{TE}(\!|\,p_{2,1}\,|\!)\,.\,\lambda\mathrm{TE}(\!|\,p_{2,2}\,|\!)\,.\,\cdots\,.\,\lambda\mathrm{TE}(\!|\,p_{2,n}\,|\!)\,.\,\mathrm{TE}(\!|\,E_2\,|\!))\;x_1x_2\ldots x_n)\\ &\qquad\ldots\\ &\qquad[\!]\;((\lambda\mathrm{TE}(\!|\,p_{m,1}\,|\!)\,.\,\lambda\mathrm{TE}(\!|\,p_{m,2}\,|\!)\,.\,\cdots\,.\,\lambda\mathrm{TE}(\!|\,p_{m,n}\,|\!)\,.\,\mathrm{TE}(\!|\,E_m\,|\!))\;x_1x_2\ldots x_n)\\ &\qquad[\!]\;\mathit{ERROR}\\ &\qquad)\end{aligned}$$

ahol $x_1, x_2, \ldots, x_n$ *új változók.*

3.5.7. Példa. *(Többsoros, többmintás függvénydefiníció)*
Legyen az *xor* függvény definíciója a következő:

$$\begin{aligned}
&\mathit{xor}\ \mathit{false}\ y \quad = y\\
&\mathit{xor}\ \mathit{true}\ \ \mathit{false} = \mathit{true}\\
&\mathit{xor}\ \mathit{true}\ \ \mathit{true}\ \ = \mathit{false}
\end{aligned}$$

Az *xor* függvény alakja a kibővített λ-kalkulusban:

$$\begin{aligned}
&\mathit{xor} =\\
&\lambda uv\,.\ (\ ((\lambda \mathrm{TE}(\!|\,\mathit{false}\,|\!)\,.\,\lambda \mathrm{TE}(\!|\,y\,|\!)\,.\,\mathrm{TE}(\!|\,y\,|\!))\ uv)\\
&\qquad [\!]\ ((\lambda \mathrm{TE}(\!|\,\mathit{true}\,|\!)\,.\,\lambda \mathrm{TE}(\!|\,\mathit{false}\,|\!)\,.\,\mathrm{TE}(\!|\,\mathit{true}\,|\!))\ uv)\\
&\qquad [\!]\ ((\lambda \mathrm{TE}(\!|\,\mathit{true}\,|\!)\,.\,\lambda \mathrm{TE}(\!|\,\mathit{true}\,|\!)\,.\,\mathrm{TE}(\!|\,\mathit{false}\,|\!))\ uv)\\
&\qquad [\!]\ \mathit{ERROR}\\
&\qquad)\\
&\Longrightarrow^{+}\\
&\lambda uv\,.\ (\ ((\lambda \mathit{false}\,.\,\lambda y\,.\,y)\ uv)\\
&\qquad [\!]\ ((\lambda \mathit{true}\,.\,\lambda \mathit{false}\,.\,\mathit{true})\ uv)\\
&\qquad [\!]\ ((\lambda \mathit{true}\,.\,\lambda \mathit{true}\,.\,\mathit{false})\ uv)\\
&\qquad [\!]\ \mathit{ERROR}\\
&\qquad)
\end{aligned}$$

□

Előfordulhat, hogy egy többsoros n-mintás függvénydefiníció i-edik sorában k minta van megadva ($k < n$). A fenti eljárás szerint akkor ehhez a definícióhoz a

$$\begin{aligned}
&\lambda x_1 x_2 \ldots x_n\,.\ (\ \ldots\\
&\qquad .\ \ldots\\
&\qquad [\!]\ ((\lambda \mathrm{TE}(\!|\,p_{i,1}\,|\!)\,.\,\lambda \mathrm{TE}(\!|\,p_{i,2}\,|\!)\,.\,\cdots\,.\,\lambda \mathrm{TE}(\!|\,p_{i,k}\,|\!)\,.\,\mathrm{TE}(\!|\,E_i\,|\!))\ x_1 x_2 \ldots x_n)\\
&\qquad .\ \ldots\\
&\qquad)
\end{aligned}$$

programot rendeljük. A probléma az, hogy a függvény egy n-változós absztrakció. Ha k argumentumot applikálunk, akkor biztosan egy

$$\lambda x_{k+1} \ldots x_n\,.\,E$$

absztrakciót kapunk eredményül, ha pedig n argumentumot adunk meg, akkor az i-edik sor mintaillesztése az $F_1, F_2, \ldots, F_n$ argumentumokkal a

$$\mathrm{TE}(\!|\,E_i\,|\!)\ F_{k+1} \ldots F_n$$

applikációt adja, még akkor is, ha a mintaillesztés a k-ik mintáig sikeres.

Vezessünk be egy új *konstanst*, a „bármi” jelentéssel. Jelöljük ezt a konstanst a _ jellel, nevezzük *joker*-nek, és legyen $\mathrm{TE}(\!| \_ |\!) = \_$. Egészítsük ki az i-edik sor mintáit $n - k$ darab _ konstanssal, így a függvénydefiníció kifejezésébe $n - k$ darab $\lambda\mathrm{TE}(\!| \_ |\!)$ absztrakció kerül (a kitevőben zárójelben a minta sorszáma látható):

$$
\begin{array}{l}
\lambda x_1 x_2 \ldots x_n \,.\, (\ldots \\
\qquad\qquad . \quad \ldots \\
\qquad\qquad [] \quad ((\lambda\mathrm{TE}(\!| p_{i,1} |\!) \,.\, \cdots \,.\, \lambda\mathrm{TE}(\!| p_{i,k} |\!) \,.\, \lambda\mathrm{TE}(\!| \_ |\!)^{(k+1)} \,.\, \cdots \,.\, \lambda\mathrm{TE}(\!| \_ |\!)^{(n)} \,.\, \mathrm{TE}(\!| E_i |\!)) \\
\qquad\qquad\quad x_1 x_2 \ldots x_n) \\
\qquad\qquad . \quad \ldots \\
\qquad\qquad)
\end{array}
$$

azaz a kifejezés

$$
\begin{array}{l}
\lambda x_1 x_2 \ldots x_n \,.\, (\ \ldots \\
\qquad\qquad . \ \ldots \\
\qquad\qquad [] \ ((\lambda\mathrm{TE}(\!| p_{i,1} |\!) \,.\, \cdots \,.\, \lambda\mathrm{TE}(\!| p_{i,k} |\!) \,.\, \lambda\_^{(k+1)} \,.\, \cdots \,.\, \lambda\_^{(n)} \,.\, \mathrm{TE}(\!| E_i |\!))\ x_1 x_2 \ldots x_n) \\
\qquad\qquad . \ \ldots \\
\qquad\qquad)
\end{array}
$$

alakú.

Megjegyezzük, hogy a *joker* jelek a függvénymegadásban tetszőleges minta pozícióján előfordulhatnak, nem kell feltétlenül az utolsó mintáknak lenniük.

3.5.8. Példa. *(Nem egyforma darabszámú minták)*
Legyen az f függvény definíciója

$$
\begin{array}{l}
f\ 0 \quad = 2 \\
f\ x\ 1 = 4 \\
f\ x\ 2 = 6
\end{array}
$$

Ekkor

$$
\begin{array}{l}
f = \\
\lambda xy \,.\, (\ ((\lambda\mathrm{TE}(\!| 0 |\!) \,.\, \lambda\mathrm{TE}(\!| \_ |\!) \,.\, \mathrm{TE}(\!| 2 |\!))xy) \\
\qquad [] \ ((\lambda\mathrm{TE}(\!| x |\!) \,.\, \lambda\mathrm{TE}(\!| 1 |\!) \,.\, \mathrm{TE}(\!| 4 |\!))xy) \\
\qquad [] \ ((\lambda\mathrm{TE}(\!| x |\!) \,.\, \lambda\mathrm{TE}(\!| 1 |\!) \,.\, \mathrm{TE}(\!| 6 |\!))xy) \\
\qquad [] \ \mathit{ERROR} \\
\qquad)
\end{array}
$$

$\Longrightarrow^{+}$

$$\begin{aligned}
&\lambda xy.\ (\ ((\lambda 0\,.\,\lambda_\,.\,2)xy)\\
&\qquad [\!]\ ((\lambda x\,.\,\lambda 1\,.\,4)xy)\\
&\qquad [\!]\ ((\lambda x\,.\,\lambda 1\,.\,6)xy)\\
&\qquad [\!]\ \mathit{ERROR}\\
&\qquad)
\end{aligned}$$

□

A joker konstanst nem csak abban az esetben használhatjuk, amikor a többsoros definíciók egyes soraiban a paraméterek darabszáma különböző. Használhatjuk a konstanst a „joker” valódi értelmében is, azaz akkor, amikor a minta megadásánál közömbös, hogy az adott helyen milyen kifejezés áll.

3.5.9. Példa. *(A joker konstans használata)*
Legyen az f függvény definíciója a következő:

$$\begin{aligned}
f\ _\ 0\ _ &= 1\\
f\ _\ _\ 0 &= 2\\
f\ 0\ _\ _ &= 3
\end{aligned}$$

A függvénydefiníció λ-kifejezése:

$$\begin{aligned}
&f =\\
&\lambda xyz.\ (\ ((\lambda \mathrm{TE}(\!|_|\!)\,.\,\lambda \mathrm{TE}(\!|0|\!)\,.\,\lambda \mathrm{TE}(\!|_|\!)\,.\,\mathrm{TE}(\!|1|\!))xyz)\\
&\qquad [\!]\ ((\lambda \mathrm{TE}(\!|_|\!)\,.\,\lambda \mathrm{TE}(\!|_|\!)\,.\,\lambda \mathrm{TE}(\!|0|\!)\,.\,\mathrm{TE}(\!|2|\!))xyz)\\
&\qquad [\!]\ ((\lambda \mathrm{TE}(\!|0|\!)\,.\,\lambda \mathrm{TE}(\!|_|\!)\,.\,\lambda \mathrm{TE}(\!|_|\!)\,.\,\mathrm{TE}(\!|3|\!))xyz)\\
&\qquad [\!]\ \mathit{ERROR}\\
&\qquad)\\
&\Longrightarrow^{+}\\
&\lambda xyz.\ (\ ((\lambda_\,.\,\lambda 0\,.\,\lambda_\,.\,1)xyz)\\
&\qquad [\!]\ ((\lambda_\,.\,\lambda_\,.\,\lambda 0\,.\,2)xyz)\\
&\qquad [\!]\ ((\lambda 0\,.\,\lambda_\,.\,\lambda_\,.\,3)xyz)\\
&\qquad [\!]\ \mathit{ERROR}\\
&\qquad)
\end{aligned}$$

□

3.6. Speciális függvénydefiníciók

3.6.1. Az őrfeltételek fordítása

Az egysoros n-mintás definíció bal oldalához egy „|” jel után tehetünk egy *true* vagy *false* értékű kifejezést, úgynevezett *őrfeltételt*:

$\langle \textit{függvény} \rangle \langle \textit{minta} \rangle_1 \ldots \langle \textit{minta} \rangle_n = \langle \textit{kifejezés} \rangle_1 \mid \quad \langle \textit{kifejezés} \rangle_2 \quad (n > 1)$

Az őrfeltételes, egydefiníciós n-mintás program alakja tehát

$f\ p_1\ p_2 \ldots p_n = E \quad | \quad F.$

Az őrfeltétel azt jelenti, hogy a mintaillesztést csak abban az esetben kell elvégezni, ha $F = \textit{true}$. Így az őrfeltétellel megadott minta egyszerűen egy olyan *if* kifejezésre alakítható át, amelyiknek a feltétele $\mathrm{TE}(\!| F |\!)$, a *then* ágán a $\mathrm{TE}(\!| E |\!)$ kifejezés, az *else* ágán pedig a *fail* konstans áll. Ugyanis $F = \textit{false}$ esetén a mintaillesztés sikertelennek tekinthető, ezért az eredménynek ebben az esetben *fail*-nek kell lennie.

Így az f definíciójából származó n-változós absztrakció törzsében a mintaillesztést végző kifejezésbe az E átalakításra ezt a feltételvizsgálatot is elő kell írni egy *if* kifejezéssel:

$\mathrm{TD}(\!| f\ p_1\ p_2\ \ldots\ p_n = E \quad | \quad F |\!)$
$\Longrightarrow$
$f = \lambda x_1 x_2 \ldots x_n .\ (\ ((\lambda \mathrm{TE}(\!| p_1 |\!) .\ \lambda \mathrm{TE}(\!| p_2 |\!) .\ \cdots .\ \lambda \mathrm{TE}(\!| p_n |\!) .$
$\qquad \textit{if}\ \mathrm{TE}(\!| F |\!)\ \ \mathrm{TE}(\!| E |\!)\ \ \textit{fail}\ )\ x_1 x_2 \ldots x_n)$
$\qquad [\!]\ \textit{ERROR}$
$\qquad)$
ahol $x_1, x_2, \ldots, x_n$ *új változók*

Többsoros, mintákat tartalmazó őrfeltételes függvénydefiníció alakja a következő:

$\langle \textit{függvény} \rangle \langle \textit{minta} \rangle_{1,1} \ldots \langle \textit{minta} \rangle_{1,n} = \langle \textit{kifejezés} \rangle_{11} \mid \langle \textit{kifejezés} \rangle_{12}$
$\langle \textit{függvény} \rangle \langle \textit{minta} \rangle_{2,1} \ldots \langle \textit{minta} \rangle_{2,n} = \langle \textit{kifejezés} \rangle_{21} \mid \langle \textit{kifejezés} \rangle_{22}$
$\ldots$
$\langle \textit{függvény} \rangle \langle \textit{minta} \rangle_{m,1} \ldots \langle \textit{minta} \rangle_{m,n} = \langle \textit{kifejezés} \rangle_{m1} \mid \langle \textit{kifejezés} \rangle_{m2} \quad (n, m > 1)$

azaz a program röviden az

$$\begin{aligned} f\ p_{1,1}\ p_{1,2}\ \ldots\ p_{1,n} &= E_1 \quad | \quad F_1 \\ f\ p_{2,1}\ p_{2,2}\ \ldots\ p_{2,n} &= E_2 \quad | \quad F_2 \\ &\ldots \\ f\ p_{m,1}\ p_{m,2}\ \ldots\ p_{m,n} &= E_m \quad | \quad F_m \end{aligned}$$

alakban írható fel.

Látható, hogy minden sorban a mintához őrfeltétel van megadva, ezeket az egysoros mintánál megadott módszerrel külön-külön be kell építeni a mintához tartozó kifejezésbe. Természetesen, ha egy mintához nincs őrfeltétel megadva, akkor a minta átalakításánál ilyen módosításra nincs szükség. Az átalakítás tehát a következő:

$$
\begin{aligned}
&\mathrm{TD}(\!| \; f\ p_{1,1}\ p_{1,2}\ \ldots\ p_{1,n} = \ E_1 \ \mid \ F_1 \\
&\qquad f\ p_{2,1}\ p_{2,2}\ \ldots\ p_{2,n} = \ E_2 \ \mid \ F_2 \\
&\qquad\qquad\qquad \ldots \\
&\qquad f\ p_{m,1}\ p_{m,2}\ \ldots\ p_{m,n} = \ E_m \ \mid \ F_m \; |\!) \\
&\Longrightarrow \\
&f = \lambda x_1 x_2 \ldots x_n \,.\, (\ ((\lambda \mathrm{TE}(\!| \, p_{1,1} \,|\!) \,.\, \lambda \mathrm{TE}(\!| \, p_{1,2} \,|\!) \,.\, \cdots \,.\, \lambda \mathrm{TE}(\!| \, p_{1,n} \,|\!) \,. \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{if}\ \mathrm{TE}(\!| \, F_1 \,|\!)\ \ \mathrm{TE}(\!| \, E_1 \,|\!)\ \ \mathit{fail}\ )\ x_1 x_2 \ldots x_n) \\
&\qquad\qquad [\!] \ ((\lambda \mathrm{TE}(\!| \, p_{2,1} \,|\!) \,.\, \lambda \mathrm{TE}(\!| \, p_{2,2} \,|\!) \,.\, \cdots \,.\, \lambda \mathrm{TE}(\!| \, p_{2,n} \,|\!) \,. \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{if}\ \mathrm{TE}(\!| \, F_2 \,|\!)\ \ \mathrm{TE}(\!| \, E_2 \,|\!)\ \ \mathit{fail}\ )\ x_1 x_2 \ldots x_n) \\
&\qquad\qquad \ldots \\
&\qquad\qquad [\!] \ ((\lambda \mathrm{TE}(\!| \, p_{m,1} \,|\!) \,.\, \lambda \mathrm{TE}(\!| \, p_{m,2} \,|\!) \,.\, \cdots \,.\, \lambda \mathrm{TE}(\!| \, p_{m,n} \,|\!) \,. \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{if}\ \mathrm{TE}(\!| \, F_m \,|\!)\ \ \mathrm{TE}(\!| \, E_m \,|\!)\ \ \mathit{fail}\ )\ x_1 x_2 \ldots x_n) \\
&\qquad\qquad [\!] \ \mathit{ERROR} \\
&\qquad\qquad)
\end{aligned}
$$

ahol $x_1, x_2, \ldots, x_n$ *új változók*

3.6.1. Példa. *(Őrfeltételek)*

Az *sgl* függvény definíciója legyen a következő:

$$
\begin{aligned}
&\mathit{sgl}\ \mathit{nil} &&= 0 \\
&\mathit{sgl}\ (\mathit{cons}\ x\ xs) &&= 1 \quad \mid \quad x < 0 \\
&\mathit{sgl}\ (\mathit{cons}\ x\ xs) &&= 2
\end{aligned}
$$

A definícióból a következő kifejezést kapjuk:

$$\mathit{sgl} =$$

$$
\begin{array}{l}
\lambda y\,.\ (\ ((\lambda nil\,.\,0)\ y) \\
\quad [\!]\ ((\lambda(cons\ x\ xs)\,.\,if\ (<\ x\ 0)\ 1\ fail\ )\ y) \\
\quad [\!]\ ((\lambda(cons\ x\ xs)\,.\,2)\ y) \\
\quad [\!]\ ERROR \\
\quad)
\end{array}
$$

□

Speciális esetekben, amikor egy függvénydefinícióban nincsenek minták, azaz a definícióban csak változók szerepelnek, a függvénydefiníció szokásos alakja a következő:

$$
\begin{array}{lcl}
f\ x_1\ x_2\ \ldots\ x_n & = & E_1 \ \mid\ F_1 \\
& = & E_2 \ \mid\ F_2 \\
& \ldots & \\
& = & E_m \ \mid\ F_m
\end{array}
$$

A *funkcionális kiértékelés* szerint ez a megadás azt jelenti, hogy először az F_1 értékét kell vizsgálni, és ha ez *true*, akkor a függvény az E_1, ha *false*, akkor F_2-vel kell ezt a műveletet végrehajtani, és így tovább. Ha az F_m értéke is *false*, csak ebben az esetben kell a *fail* konstanst adni eredményül. Ezt a műveletsorozatot egymásba ágyazott *if* utasításokkal lehet megvalósítani, tehát

$$
\begin{array}{l}
\mathrm{TD}(\!|\ f\ x_1\ x_2 \ldots x_n = E_1 \ \mid\ F_1 \\
\qquad\qquad\qquad = E_2 \ \mid\ F_2 \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad = E_m \ \mid\ F_m\ |\!) \\
\Longrightarrow \\
f = \lambda x_1\ x_2 \ldots x_n\,.\ (\ (if\ \mathrm{TE}(\!|\ F_1|\!)\ \ \mathrm{TE}(\!|\ E_1|\!) \\
\qquad\qquad\qquad\qquad (if\ \mathrm{TE}(\!|\ F_2|\!)\ \ \mathrm{TE}(\!|\ E_2|\!) \\
\qquad\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\qquad\qquad (if\ \mathrm{TE}(\!|\ F_m|\!)\ \ \mathrm{TE}(\!|\ E_m|\!)\ \ fail) \ldots)) \\
\qquad\qquad\qquad [\!]\ ERROR \\
\qquad\qquad\qquad)
\end{array}
$$

Megjegyezzük, hogy ha az $F_1, F_2, \ldots, F_m$ feltételrendszer *teljes*, azaz a feltételek közül legalább egy biztosan teljesül, akkor az *ERROR* ágra nincs is szükség. Mivel ekkor $\mathrm{TE}(\!|\ F_m|\!)$-nek biztosan *true*-nak kell lennie, az utolsó

$$if\ \mathrm{TE}(\!|\ F_m|\!)\ \ \mathrm{TE}(\!|\ E_m|\!)\ \ fail$$

kifejezés törölhető, a $\mathrm{TE}(\!|\ E_m|\!)$ pedig a közvetlenül megelőző *if* utasítás *else* ágába

építhető be.

3.6.2. Példa. *(Őrfeltételek változókkal megadott függvénynél)*
Legyen a függvénydefiníció a következő:

$$\begin{aligned} gcd\ x\ y &= gcd\ (x-y)\ y && \mid x > y \\ &= gcd\ x\ (y-x) && \mid x < y \\ &= x && \mid x = y \end{aligned}$$

A fentiek alapján a *gcd* függvény kifejezése a következő lesz:

$$\begin{aligned} &gcd = \\ &\lambda xy\,.\ (\mathit{if}\ \mathrm{TE}(\!|\, x > y \,|\!)\ \ \mathrm{TE}(\!|\, gcd\ (x-y)\ y \,|\!) \\ &\qquad (\mathit{if}\ \mathrm{TE}(\!|\, x < y \,|\!)\ \ \mathrm{TE}(\!|\, gcd\ x\ (y-x) \,|\!)\ \ \mathrm{TE}(\!|\, x \,|\!)\ )) \\ &\Longrightarrow^{+} \\ &\lambda xy\,.\ (\mathit{if}\ \ (>\ \ x\ y)\ \ (gcd\ (-\ x\ y)\ y) \\ &\qquad (\mathit{if}\ \ (<\ \ x\ y)\ \ (gcd\ x\ (-\ y\ x))\ \ x\ )) \end{aligned}$$

□

3.6.2. A lokális definíciók fordítása

A funkcionális programnyelvek lehetőséget adnak arra is, hogy egy függvénydefinícióra az őrfeltételeken kívül további megkötéseket, *lokális definícióknak* nevezett korlátozásokat is adjunk. Egyes programnyelvekben ezeket a lokális definíciókat a *where* kulcsszó vezeti be, ezt a jelölést használjuk mi is.

A definíció szerkezete:

⟨*függvénydefiníció*⟩
    where ⟨*definíció*⟩$_1$
        ...
        ⟨*definíció*⟩$_m$ $(m \geq 1)$

A *where* kulcsszó utáni m definíció az egész függvénymegadásra vonatkozik, azaz hatáskörük a függvénydefiníció és az összes *where* utáni definíció.

Jelöljük ezt a definíciót röviden a következőképpen:

D
  where D_1
        ...
        D_m $(m \geq 1)$

D az akár többsoros, őrfeltételes, mintákat is tartalmazó lokális definíciós függvénydefiníció lokális definíciók nélküli része, a definiált függvény legyen az f.

Határozzuk meg TD(⦇ D ⦈)-t. Jelöljük H(TD(⦇ D ⦈))-vel jelöljük a D-ből kapott λ-absztrakció λ-jelét és változóit, továbbá jelöljük C(TD(⦇ D ⦈))-vel a D λ-absztrakciójának törzsét, és ha az f definíciójában n minta szerepel,

$$\text{TD}(\!| D |\!) \equiv f = H(\text{TD}(\!| D |\!)) \,.\, C(\text{TD}(\!| D |\!)),$$

ahol tehát

$$H(\text{TD}(\!| D |\!) = \lambda x_1 x_2 \ldots x_n.$$

Mivel a definíciókban rekurziót is megengedünk, a *where* kulcsszót tartalmazó kifejezést a *letrec* kifejezés felhasználásával fogjuk átalakítani, ahol a *letrec* kifejezést a függvénydefinícióból származó λ-absztrakció törzsébe fogjuk beépíteni.

Így egy lokális definíciókat tartalmazó függvénydefinícióból következő kifejezést állíthatjuk elő:

$$\begin{array}{l} D \\ \quad \textit{where}\ D_1 \\ \qquad \ldots \\ \qquad D_m \quad \Longrightarrow \quad f = H(\text{TD}(\!| D |\!)) \,.\, (\ \textit{letrec}\ \text{TD}(\!| D_1 |\!) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{TD}(\!| D_m |\!) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textit{in}\ C(\text{TD}(\!| D |\!)\ ) \qquad (m \geq 1) \end{array}$$

3.6.3. Példa. *(Függvénydefiníció őrfeltétellel és lokális definícióval)*
Alakítsuk át a következő definíciót:

$$\begin{array}{l} f\ x\ y = a + b + c \\ \quad \textit{where}\ a = x * x \\ \qquad\quad\ \ b = y * y \\ \qquad\quad\ \ c = x * y \end{array}$$

Az $f\ x\ y = a + b + c$ definíció átalakítása $f = \lambda xy \,.\, +\ (+\ a\ b)\ c$, így

$$H(\text{TD}(\!|\ f\ x\ y = a + b + c\ |\!)) = \lambda xy,$$

és az absztrakció törzse, azaz

$$C(\text{TD}(\!|\ f\ x\ y = a + b + c\ |\!)) = +\ (+\ a\ b)\ c.$$

Ezt felhasználva

$$
\begin{array}{l}
f = \\
\lambda xy\,.\ \mathit{letrec}\ \mathrm{TD}(\!|\, a = x * x \,|\!) \\
\qquad\qquad\quad \mathrm{TD}(\!|\, b = y * y \,|\!) \\
\qquad\qquad\quad \mathrm{TD}(\!|\, c = x * y \,|\!) \\
\qquad\qquad \mathit{in}\ +\ (+\ a\ b)\ c \\
\Longrightarrow \\
\lambda xy\,.\ \mathit{letrec}\ a = *\ x\ x \\
\qquad\qquad\quad b = *\ y\ y \\
\qquad\qquad\quad c = x * y \\
\qquad\qquad \mathit{in}\ +(+\ a\ b)\ c
\end{array}
$$

Itt most nincs rekurzió, ezért a *letrec* helyett *let* kulcsszót írva és a let-kifejezéseket skatulyázva

$$
\begin{array}{l}
f = \\
\lambda xy\,.\ \mathit{let}\ a = *\ x\ x \\
\qquad \mathit{in}\ (\ \mathit{let}\ b = *\ y\ y \\
\qquad\qquad \mathit{in}\ (\ \mathit{let}\ c = *\ x\ y \\
\qquad\qquad\qquad \mathit{in}\ +\ (+\ a\ b)\ c)\ )
\end{array}
$$

A let-kifejezéseket λ-kifejezésre alakíthatjuk és a lehetséges redukciókat végrehajtva

$$
\begin{array}{l}
f = \\
\lambda xy\,.\ \mathit{let}\ a = *\ x\ x \\
\qquad \mathit{in}\ (\ \mathit{let}\ b = *\ y\ y \\
\qquad\qquad \mathit{in}\ (\ \mathit{let}\ c = *\ x\ y \\
\qquad\qquad\qquad \mathit{in}\ +\ (+\ a\ b)\ c)\ ) \\
\rightsquigarrow \\
\lambda xy\,.\ \mathit{let}\ a = *\ x\ x \\
\qquad \mathit{in}\ (\ \mathit{let}\ b = *\ y\ y \\
\qquad\qquad \mathit{in}\ \underline{(\lambda c\,.\ +\ \ (+\ a\ b)\ c)(*\ x\ y)}\ ) \\
\rightarrow \\
\lambda xy\,.\ \mathit{let}\ a = *\ x\ x \\
\qquad \mathit{in}\ (\ \mathit{let}\ b = *\ y\ y \\
\qquad\qquad \mathit{in}\ +\ (+\ a\ b)\ (*\ x\ y)\ ) \\
\rightsquigarrow
\end{array}
$$

$\lambda xy\,.\,$ *let* $a = * \; x \; x$
in $\underline{(\lambda b\,.\,+\;(+\;a\;b)\;(*\;x\;y))(*\;y\;y)}$
$\rightarrow$
$\lambda xy\,.\,$ *let* $a = * \; x \; x$
in $+\;(+\;a\;(*\;y\;y))\;(*\;x\;y)$
$\rightsquigarrow$
$\lambda xy\,.\,\underline{(\lambda a\,.\,(+\;(+\;a\;(*\;y\;y))\;(*\;x\;y))(*\;x\;x)}$
$\rightarrow$
$\lambda xy\,.\,+\;(+\;(*\;x\;x)\;(*\;y\;y))\;(*\;x\;y)$ □

3.7. Mintaabsztrakciók

A mintákkal megadott függvénydefiníciók átalakításánál láttuk, hogy eredményül olyan λ-absztrakciókat kaptunk, amelyekben az absztrakció változója helyén minták szerepeltek. Ebben a szakaszban ezeknek az értelmezését adjuk meg.

Az olyan applikációt, ahol az applikáció első tagja egy mintaabsztrakció, *mintaapplikációnak* nevezzük. A mintaabsztrakciók jelentését ilyen mintaapplikációkkal írjuk le úgy, hogy megadjuk a mintaapplikációk redukciós szabályait.

3.7.1. Konstansos absztrakciók

A konstansos absztrakció, azaz a

$\lambda\langle konstans\rangle\,.\,\langle\lambda\text{-}kifejezés\rangle$

kifejezés szemantikáját a következőképpen adjuk meg (a leírásokban a konstansok rövid jelölésére a *k* betűt használjuk).

$(\lambda k\,.\,E)F \;\rightarrow E, \qquad ha\; F \;\rightarrow^{*}\; k$
$(\lambda k\,.\,E)F \;\rightarrow$ *fail*, $\qquad ha\; F \;\not\rightarrow^{*}\; k\; és\; F \neq \bot$
$(\lambda k\,.\,E)F \;\rightarrow \bot, \qquad ha\; F = \bot$

Az értelmezés szerint tehát, ha egy konstansos absztrakcióra egy olyan kifejezést applikálunk, aminek

- van értéke és ez az érték megegyezik az absztrakció konstansával, akkor a mintaillesztés eredményéül az absztrakció törzsét kapjuk,
- ha az érték nem azonos az absztrakció konstansával, akkor a mintaillesztés a *fail* eredményt adja,

- ha az applikált kifejezésnek nincs értéke, azaz a kifejezés redukciós sorozata nem terminál, akkor a mintaillesztés is ezt a „bottom” értéket adja.

Megjegyezzük, hogy a konstansos absztrakció kiértékelése *mohó*, hiszen mindegyik esetben először a függvényapplikáció második tagjának, azaz az argumentumnak kell az értékét meghatározni.

A 3.5.2. pontban vezettük be a _ konstanst a „bármi” kifejezés, a *joker* konstans jelölésére, és itt vezettük be a $\lambda\_\,.\,E$ absztrakciót is.

Most adjuk meg ennek a kifejezésnek a jelentését is. Mivel _ egy konstans, a fenti átalakítás szerint $(\lambda\_\,.\,E)\_ \;\rightarrow\; E$. Mivel a _ jelentése „bármi”, a $\lambda\_\,.\,E$ absztrakcióra tetszőleges F kifejezést applikálva az eredmény legyen az absztrakció törzsének kifejezése.

$$(\lambda\_\,.\,E)F \;\rightarrow E$$

Így ha a

$$(\lambda p_1\,.\,\lambda p_2\,.\,\cdots\,.\,\lambda p_k\,.\,\lambda\_^{(k+1)}\,.\,\cdots\,.\,\lambda\_^{(n)}\,.\,\mathrm{TE}(\!|\,E_i\,|\!))\;F_1F_2\ldots F_kF_{k+1}\ldots F_n$$

és az $F_1, F_2, \ldots, F_n$ argumentumokkal a mintaillesztés a k-ik mintáig sikeres, akkor a

$$(\lambda\_^{(k+1)}\,.\,\cdots\,.\,\lambda\_^{(n)}\,.\,\mathrm{TE}(\!|\,E_i\,|\!))F_{k+1}\ldots F_n$$

kifejezést kapjuk. A $(\lambda\_\,.\,E)F$ fenti értelmezése alapján

$$(\lambda\_^{(k+1)}\,.\,\cdots\,.\,\lambda\_^{(n)}\,.\,\mathrm{TE}(\!|\,E_i\,|\!))F_{k+1}\ldots F_n \;\rightarrow^{+}$$
$$\mathrm{TE}(\!|\,E_i\,|\!),$$

és így valóban a kívánt $\mathrm{TE}(\!|\,E_i\,|\!)$ eredményt kapjuk meg.

3.7.1. Példa. *(Mintaillesztés konstansos absztrakciókkal)*

$(\lambda 1\,.\,2)1 \;\rightarrow 2$
$(\lambda 1\,.\,2)2 \;\rightarrow$ *fail*
$(\lambda 1\,.\,2)(-\;4\;3) \;\rightarrow 2$
$(\lambda 1\,.\,EF)1 \;\rightarrow EF$ □

3.7.2. Példa. *(Az xor értéke)*

A 3.5.7. példában adtuk meg az *xor* függvény definíciójának λ-kifejezését:

xor =

$\lambda uv\,.\;(\;((\lambda \mathit{false}\,.\,\lambda y\,.\,y)\;uv)$
$\quad [\!]\;((\lambda \mathit{true}\,.\,\lambda \mathit{false}\,.\,\mathit{true})\;uv)$
$\quad [\!]\;((\lambda \mathit{true}\,.\,\lambda \mathit{true}\,.\,\mathit{false})\;uv)$
$\quad [\!]\;\mathit{ERROR}$
$\quad)$

Határozzuk meg az *xor false true* kifejezés értékét.

$\mathit{xor}\;\mathit{false}\;\mathit{true} \equiv$
$(\lambda uv\,.\;(\;((\lambda \mathit{false}\,.\,\lambda y\,.\,y)\;uv)$
$\quad [\!]\;((\lambda \mathit{true}\,.\,\lambda \mathit{false}\,.\,\mathit{true})\;uv)$
$\quad [\!]\;((\lambda \mathit{true}\,.\,\lambda \mathit{true}\,.\,\mathit{false})\;uv)$
$\quad [\!]\;\mathit{ERROR}$
$\quad)$
$)\;\mathit{false}\;\mathit{true}$
$\rightarrow^{+}$
$(\;((\lambda \underline{\mathit{false}}\,.\,\lambda y\,.\,y)\;\underline{\mathit{false}}\;\mathit{true})$
$[\!]\;((\lambda \mathit{true}\,.\,\lambda \mathit{false}\,.\,\mathit{true})\;\mathit{false}\;\mathit{true})$
$[\!]\;((\lambda \mathit{true}\,.\,\lambda \mathit{true}\,.\,\mathit{false})\;\mathit{false}\;\mathit{true})$
$[\!]\;\mathit{ERROR}$
$)$
$\rightarrow$
$(\lambda y\,.\,y)\;\mathit{true}\;\rightarrow$
true □

3.7.3. Példa. *(Nem egyforma darabszámú minták)*

A 3.5.8. példában szereplő

$f\;0\;\;\;=2$
$f\;x\;1=4$
$f\;x\;2=6$

függvénydefiníció λ-kifejezése

$f=$

$\lambda xy.$ ($((\lambda 0\,.\,\lambda\_\,.\,2)xy)$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,4)xy)$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,6)xy)$
$[\!]$ *ERROR*
)

volt. Határozzuk meg az $f\ 0\ 7$ kifejezés értékét.

$f\ 0\ 7 \equiv$
$(\lambda xy.$ ($((\lambda 0\,.\,\lambda\_\,.\,2)xy)$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,4)xy)$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,6)xy)$
$[\!]$ *ERROR*
)
$)\ 0\ 7$
$\rightarrow^{+}$
($(\underline{(\lambda 0\,.\,\lambda\_\,.\,2)0}\ 7)$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,4)0\ 7)$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,6)0\ 7)$
$[\!]$ *ERROR*
)
$\rightarrow^{+}$
($(\underline{(\lambda\_\,.\,2)7})$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,4)0\ 7)$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,6)0\ 7)$
$[\!]$ *ERROR*
)
$\rightarrow$
(2
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,4)0\ 7)$
$[\!]$ $((\lambda x\,.\,\lambda 1\,.\,6)0\ 7)$
$[\!]$ *ERROR*
)
$\rightarrow^{+}$
2 □

3.7.4. Példa. *(A joker konstans használata)*

A 3.5.9. példában az

$f\ \_\ 0\ \_ = 1$
$f\ \_\ \_\ 0 = 2$
$f\ 0\ \_\ \_ = 3$

függvénydefinícióból a következő λ-kifejezést határoztuk meg:

$$
\begin{array}{l}
f = \\
\lambda xyz.\ (\ ((\lambda\_\ .\ \lambda 0\ .\ \lambda\_\ .\ 1)xyz) \\
\qquad [\!]\ ((\lambda\_\ .\ \lambda\_\ .\ \lambda 0\ .\ 2)xyz) \\
\qquad [\!]\ ((\lambda 0\ .\ \lambda\_\ .\ \lambda\_\ .\ 3)xyz) \\
\qquad [\!]\ \mathit{ERROR} \\
\qquad)
\end{array}
$$

Határozzuk meg az $f\ 2\ 1\ 0$ kifejezés értékét.

$$
\begin{array}{l}
f\ 2\ 1\ 0 \equiv \\
(\lambda xyz.\ (\ ((\lambda\_\ .\ \lambda 0\ .\ \lambda\_\ .\ 1)xyz) \\
\qquad [\!]\ ((\lambda\_\ .\ \lambda\_\ .\ \lambda 0\ .\ 2)xyz) \\
\qquad [\!]\ ((\lambda 0\ .\ \lambda\_\ .\ \lambda\_\ .\ 3)xyz) \\
\qquad [\!]\ \mathit{ERROR} \\
\qquad) \\
)2\ 1\ 0\ \rightarrow^{+} \\
(\ (\underline{(\lambda\_\ .\ \lambda 0\ .\ \lambda\_\ .\ 1)2}\ 1\ 0) \\
[\!]\ ((\lambda\_\ .\ \lambda\_\ .\ \lambda 0\ .\ 2)2\ 1\ 0) \\
[\!]\ ((\lambda 0\ .\ \lambda\_\ .\ \lambda\_\ .\ 3)2\ 1\ 0) \\
[\!]\ \mathit{ERROR} \\
)\ \rightarrow \\
(\ (\underline{(\lambda 0\ .\ \lambda\_\ .\ 1)1}\ 0) \\
[\!]\ ((\lambda\_\ .\ \lambda\_\ .\ \lambda 0\ .\ 2)2\ 1\ 0) \\
[\!]\ ((\lambda 0\ .\ \lambda\_\ .\ \lambda\_\ .\ 3)2\ 1\ 0) \\
[\!]\ \mathit{ERROR} \\
)\ \rightarrow^{+}
\end{array}
$$

(*fail* 0
‖ ((λ_ . λ_ . λ0 . 2)2 1 0)
‖ ((λ0 . λ_ . λ_ . 3)2 1 0)
‖ *ERROR*
) →
(*fail*
‖ ((λ_ . λ_ . λ0 . 2)2 1 0)
‖ ((λ0 . λ_ . λ_ . 3)2 1 0)
‖ *ERROR*
) →
(((λ_ . λ_ . λ0 . 2)2 1 0)
‖ ((λ0 . λ_ . λ_ . 3)2 1 0)
‖ *ERROR*
) $\rightarrow^+$
2 □

3.7.2. Összegkonstruktoros absztrakciók

Az összegkonstruktoros absztrakcióknak az a jellemzőjük, hogy olyan mintákat tartalmazó függvénydefinícióból származnak, ahol a függvényt többsoros mintákkal adtuk meg. Mint láttuk, egy minta első tagja egy konstruktor, és ha a konstruktornak argumentumai vannak, akkor ezek újabb minták lehetnek:

$\lambda(\langle \textit{konstruktor} \rangle\ \langle \textit{minta}_1 \rangle \cdots \langle \textit{minta}_n \rangle)\ .\langle \lambda\textit{-kifejezés} \rangle$

A rövidebb leíráshoz a továbbiakban jelöljük az összegkonstruktort az *s* betűvel, a mintákat a *p* betű jelöli.

Az összegkonstruktoros absztrakció szemantikáját mintaapplikációkkal a következőképpen adjuk meg:

$$
\begin{array}{lll}
(\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E)\ (s\ F_1 F_2 \ldots F_n) & \rightarrow (\lambda p_1\,.\,\lambda p_2\,.\,\cdots\,.\,\lambda p_n\,.\,E)\ F_1 F_2 \ldots F_n & \\
(\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E)\ (s'\ F_1 F_2 \ldots F_m) & \rightarrow \textit{fail}, & \textit{ha } s \neq s' \\
(\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E)\ (s'\ F_1 F_2 \ldots F_m) & \rightarrow \bot, & \textit{ha } s'\ F_1 F_2 \ldots F_m = \bot
\end{array}
$$

A definíció alapján először az applikáció argumentumát kell meghatározni, de látható, hogy elegendő csak az argumentum konstruktoráig eljutni. Ez azonban mindenképpen egy *mohó* kiértékelést jelent, ami nem kerülhető el, nem hagyható ki, hiszen a redukció a konstruktorok összehasonlításával kezdődik. Ha ez a reduk-

ció terminál, és a két konstruktor megegyezik, eredményül egy újabb applikációt kapunk, és csak ezután, ennek az applikációnak a végrehajtásakor lesz szükség az argumentum $F_1, F_2, \ldots, F_n$ értékeire. Ez azt jelenti, hogy az összegkonstruktoros mintaillesztés kiértékelése a konstruktor meghatározása után már *lusta* kiértékelés is lehet.

A definíció első sorából az is látható, hogy mindkét mintában az argumentumok darabszáma megegyezik. Ezt csak azért kell megkövetelni, hogy az eredményül kapott kifejezésben az absztrakciók és az argumentumok száma megegyezzen, azaz ezek redukcióiból ne maradjon absztrakció vagy felesleges argumentum. Megjegyezzük, hogy a mintaabsztrakció két tagjában az argumentumok számának egyezését a típushelyesség biztosítja.

A fenti átalakítások nulla aritású konstruktorokra pontosan a konstansos absztrakcióra megadott átalakítási szabályoknak felelnek meg.

3.7.5. Példa. *(Mintaillesztés összegkonstruktoros absztrakciókkal)*
A 3.5.5. példában szereplő

$$\begin{aligned} &\mathit{reflect}\ (\mathsf{leaf}\ n) &&= \mathsf{leaf}\ n \\ &\mathit{reflect}\ (\mathsf{branch}\ t_1\ t_2) &&= \mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1) \end{aligned}$$

függvény alakja a kibővített λ-kalkulusban a következő volt:

$$\begin{aligned} &\mathit{reflect} = \\ &\lambda x\,.\ (\ ((\lambda(\mathsf{leaf}\ n)\,.\,\mathsf{leaf}\ n)\ x) \\ &\quad [\!]\ ((\lambda(\mathsf{branch}\ t_1\ t_2)\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\ &\quad [\!]\ \mathsf{ERROR} \\ &\quad) \end{aligned}$$

Határozzuk meg a $\mathit{reflect}\ (\mathsf{leaf}\ E)$ kifejezés értékét.

$$\begin{aligned} &\mathit{reflect}\ (\mathsf{leaf}\ E) \equiv \\ &(\lambda x\,.\ (\ ((\lambda(\mathsf{leaf}\ n)\,.\,\mathsf{leaf}\ n)\ x) \\ &\quad [\!]\ ((\lambda(\mathsf{branch}\ t_1\ t_2)\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\ &\quad [\!]\ \mathsf{ERROR} \\ &\quad)\ )\ (\mathsf{leaf}\ E) \\ &\rightarrow \\ &(\ ((\lambda(\underline{\mathsf{leaf}}\ n)\,.\,\mathsf{leaf}\ n)\ (\underline{\mathsf{leaf}}\ E)) \\ &[\!]\ ((\lambda(\mathsf{branch}\ t_1\ t_2)\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathsf{leaf}\ E)) \\ &[\!]\ \mathsf{ERROR} \end{aligned}$$

)
→
$(\lambda n\,.\,\mathit{leaf}\ n)\ \ E \quad \rightarrow$
$\mathit{leaf}\ E$ □

3.7.6. Példa. *(Mintaillesztés összegkonstruktoros absztrakciókkal)*

Az előző példa függvénydefiníciójával határozzuk meg a

$\mathit{reflect}\ (\mathit{branch}(\mathit{leaf}\ E)(\mathit{leaf}\ F))$

kifejezés értékét is.

$\mathit{reflect}\ (\mathit{branch}(\mathit{leaf}\ E)(\mathit{leaf}\ F)) \equiv$
$(\lambda x\,.\ (\ ((\lambda(\mathit{leaf}\ n)\,.\,\mathit{leaf}\ n)\ x)$
$\quad [\!]\ ((\lambda(\mathit{branch}\ t_1\ t_2)\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x)$
$\quad [\!]\ \mathit{ERROR}$
$\quad)$
$)\ (\mathit{branch}(\mathit{leaf}\ E)(\mathit{leaf}\ F))$
→
$(\ \ ((\lambda(\underline{\mathit{leaf}}\ n)\,.\,\mathit{leaf}\ n)(\underline{\mathit{branch}}(\mathit{leaf}\ E)(\mathit{leaf}\ F))\ )$
$[\!]\ \ ((\lambda(\mathit{branch}\ t_1\ t_2)\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathit{branch}(\mathit{leaf}\ E)(\mathit{leaf}\ F)))$
$[\!]\ \ \mathit{ERROR}$
)
→
$(\ \ \mathit{fail}$
$[\!]\ \ ((\lambda(\mathit{branch}\ t_1\ t_2)\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathit{branch}(\mathit{leaf}\ E)(\mathit{leaf}\ F)))$
$[\!]\ \ \mathit{ERROR}$
)
→
$(\ \ ((\lambda(\underline{\mathit{branch}}\ t_1\ t_2)\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\underline{\mathit{branch}}\ (\mathit{leaf}\ E)(\mathit{leaf}\ F)))$
$[\!]\ \ \mathit{ERROR}$
)
→
$(\lambda\ t_1\,.\,\lambda\ t_2\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathit{leaf}\ E)(\mathit{leaf}\ F) \quad \rightarrow^{+}$
$\mathit{branch}\ (\mathit{reflect}\ (\mathit{leaf}\ F))\ (\mathit{reflect}\ (\mathit{leaf}\ E))$

és most az előző példa eredményét felhasználva a *reflect* kifejezésekre

branch (*reflect* (*leaf* F)) (*reflect* (*leaf* E)) $\rightarrow^{+}$
branch (*leaf* F) (*leaf* E) □

3.7.3. Szorzatkonstruktoros absztrakciók

Az előző pontban láttuk, hogy az összegtípusú absztrakcióval képzett mintaapplikáció, legalábbis az első lépésben, mohó kiértékelő algoritmust igényel. A szorzatkonstruktoros absztrakció esetén más a helyzet, ugyanis megoldható, hogy a konstruktorok összehasonlítását az absztrakció argumentumában levő konstruktor paramétereinek, azaz mintáinak feldolgozásáig elhalasszuk. Megjegyezzük, hogy az elhalasztás meggátlására egyes programnyelvek már nyelvi szinten lehetőséget adnak.

A szorzatkonstruktoros absztrakció szemantikáját a következőképpen adjuk meg, t a szorzatkonstruktort jelöli:

$(\lambda(t\ p_1 p_2 \ldots p_n)\,.\,E)\ F\ \rightarrow$
$(\lambda p_1\,.\,\lambda p_2\,.\,\cdots\,.\,\lambda p_n\,.\,E)\ (SEL\_t\_1\ F)\ (SEL\_t\_2\ F)\ldots(SEL\_t\_n\ F)$
ahol
$SEL\_t\_i\ (t\ F_1 F_2 \ldots F_n) \rightarrow F_i \qquad (1 \le i \le n)$
$SEL\_t\_i\ (t'\ F_1 F_2 \ldots F_n) \rightarrow fail, \qquad ha\ t \neq t'$
$SEL\_t\_i\ \bot \rightarrow \bot$

Látható, hogy a $(\lambda(t\ p_1 p_2 \ldots p_n)\,.\,E)\ F$ applikáció redukciós lépésében még nem történik meg a t és az F-ben levő konstruktorok vizsgálata, a konstruktorok összehasonlítását egy későbbi lépésben a *SEL*_..._... operátorok végzik.

Így lehetővé válik az is, hogy egy nemterminálló argumentumra a szorzatkonstruktoros absztrakcióval képzett applikáció terminál̈jon. Ezt a következő példában mutatjuk meg.

3.7.7. Példa. *(Szorzatkonstruktoros absztrakció nemterminálló argumentummal)* Nézzük a

zero_pair (*pair* $x\ y$) = 0

függvénydefiníciót. Ennek λ-kifejezése

zero_pair =
λz . ($((\lambda(pair\ x\ y)\,.\,0)\ z)$
| *ERROR*
)

Lusta kiértékeléssel, függetlenül az argumentumban levő konstruktortól, minden argumentumra, még az argumentum nemterminálása esetén is a 0 értéket kapjuk. Legyen

zero_pair $\perp$ =
$\lambda z\,.\,(\ ((\lambda(\mathit{pair}\ x\ y)\,.\,0)\ z)$
(| *ERROR*
)) $\perp$
$\rightarrow$
($((\lambda(\mathit{pair}\ x\ y)\,.\,0)\ \perp)$
| *ERROR*
)

Ekkor

$(\lambda(\mathit{pair}\ x\ y)\,.\,0)\ \perp\ \rightarrow$
$(\lambda xy\,.\,0)\ (\mathit{SEL\_pair\_1}\ \perp)\ (\mathit{SEL\_pair\_2}\ \perp)\ \rightarrow^{+}$
0 □

3.7.8. Példa. *(Mintaillesztés szorzatkonstruktoros absztrakcióval)*

A függvénydefiníció legyen az

add_pair (*pair* $x\ y$) = + $x\ y$

definíció, ezt átalakítva λ-kifejezésre az

add_pair = $\lambda z\,.\,(\ ((\lambda(\mathit{pair}\ x\ y)\,.\,+\ x\ y)\ z)$
[] *ERROR*
)

kifejezést kapjuk. Vizsgáljuk ezt a kifejezést a (*pair* 3 4) argumentummal.

add_pair (*pair* 3 4) $\equiv$
$(\lambda z\,.\,(\ ((\lambda(\mathit{pair}\ x\ y)\,.\,+\ x\ y)\ z)$
[] *ERROR*
)
) (*pair* 3 4)
$\rightarrow$
($((\lambda(\mathit{pair}\ x\ y)\,.\,+\ x\ y)\ (\mathit{pair}\ 3\ 4))$
[] *ERROR*
)

$\rightarrow$
($((\lambda x\,.\,\lambda y\,.\,+\;x\,y)\;(SEL\_pair\_1\;(pair\;3\;4))\;(SEL\_pair\_2\;(pair\;3\;4))$
$[\!]\;ERROR$
)
$\rightarrow$
($((\lambda y\,.\,+\;(SEL\_pair\_1\;(pair\;3\;4))\;y)\;(SEL\_pair\_2\;(pair\;3\;4))$
$[\!]\;ERROR$
) $\rightarrow$
($+\;(SEL\_pair\_1\;(pair\;3\;4))\;(SEL\_pair\_2\;(pair\;3\;4))\;[\!]\;ERROR$) $\rightarrow$ $(*)$
($+\;3\;(SEL\_pair\_2\;(pair\;3\;4))\;[\!]\;ERROR$) $\rightarrow$ $(*)$
($+\;3\;4\;[\!]\;ERROR$) $\rightarrow$
($7\;[\!]\;ERROR$) $\rightarrow$
7

Látható, hogy a két $*$ jellel megjelölt lépésben történik meg a konstruktorok vizsgálata. □

3.7.4. Azonos minták

A programozási gyakorlatban előfordulhatnak olyan többmintás függvénydefiníciók is, amelyekben azonos minták szerepelnek. Például az

$f\;u\;u = 0$

definícióban kétszer szerepel a *u* változó mintaként. Ennek a definíciónak a fordítása a már ismert módszerrel

$f = \lambda x\,.\,\lambda y\,.$ ($((\lambda u\,.\,\lambda u\,.\,0)x\,y)$
$[\!]\;ERROR$
)

Ha erre a kifejezésre például az 1 és 2 konstansokat applikáljuk, akkor

$f\;1\;2 \equiv$
$(\lambda x\,.\,\lambda y\,.$ ($((\lambda u\,.\,\lambda u\,.\,0)x\,y)$
$[\!]\;ERROR$
)) $1\;2\;\rightarrow^{+}$

($((\lambda u\,.\,\lambda u\,.\,0)1\ 2)$
[] *ERROR*
) $\rightarrow^{+}$
0

eredményt kapjuk, ami nyilvánvalóan hibás. A problémát az okozza, hogy a függvénydefinícióban szereplő minta nem csak egyszerűen két u változó, hanem ebben az is benne van, hogy a két minta azonos, és ez az információ nem került át a definícióból származó λ-kifejezésbe.

Erre a problémára egyszerűnek tűnik az a megoldás, hogy akkor a függvénydefiníció legyen például

$f'\ u\ v = 0 \quad | \quad u = v,$

azaz a minták azonosságát egy őrfeltétellel írjuk elő.

Ebből a mintából a következő kifejezést kapjuk:

$f' = \lambda x\,.\,\lambda y\,.$ ($((\lambda u\,.\,\lambda v\,.\,$ *if* $(=\ u\ v)\ 0$ *fail* $)x\ y)$
[] *ERROR*
)

és az $f'\ 1\ 2$ kifejezésre

$f'\ 1\ 2 \equiv$
$(\lambda x\,.\,\lambda y\,.$ ($((\lambda u\,.\,\lambda v\,.$ *if* $(=\ u\ v)\ 0$ *fail* $)x\ y)$
[] *ERROR*
)) 1 2 $\rightarrow^{+}$
($((\lambda u\,.\,\lambda v\,.$ *if* $(=\ u\ v)\ 0$ *fail* $)1\ 2)$
[] *ERROR*
) $\rightarrow^{+}$
(*if* $(=\ 1\ 2)\ 0$ *fail*
[] *ERROR*
) $\rightarrow$
(*fail*
[] *ERROR*
) $\rightarrow$
ERROR,

ami valóban a helyes eredmény.

Az őrfeltétel bevezetésével azonban a problémát nem sikerült teljesen megolda-

nunk, az átalakítás nem lesz azonos átalakítás. Vannak olyan argumentumok, amelyre az eredeti függvénydefiníció és az őrfeltételes definíció különbözőképpen viselkedik. Ilyen argumentumsorozat például a $\bot, 1, 2$ sorozat az

$f\ u\ v\ v = u$

definícióra.

$$f = \lambda x\,.\,\lambda y\,.\,\lambda z\,.\ (\ ((\lambda u\,.\,\lambda v\,.\,\lambda v\,.\,u)x\ y\ z)$$
$$[\!]\ \mathit{ERROR}$$
$$)$$

és erre a kifejezésre a $\bot, 1, 2$ kifejezéseket applikálva eredményül a $\bot$-t kapjuk:

$f\ \bot\ 1\ 2 \equiv$
$(\lambda x\,.\,\lambda y\,.\,\lambda z\,.\ (\ ((\lambda u\,.\,\lambda v\,.\,\lambda v\,.\,u)x\ y\ z)$
$[\!]\ \mathit{ERROR}$
$)\ )\bot\ 1\ 2\ \rightarrow^{+}$
$(\ ((\lambda u\,.\,\lambda v\,.\,\lambda v\,.\,u)\bot\ 1\ 2)$
$[\!]\ \mathit{ERROR}$
$)\ \rightarrow^{+}$
$(\ \bot$
$[\!]\ \mathit{ERROR}$
$)\ \rightarrow$
$\bot$

Ugyanakkor, az őrfeltételes átalakítással az

$f'\ u\ v\ w = u \quad | \quad v = w$

definícióra

$$f' = \lambda x\,.\,\lambda y\,.\,\lambda z\,.\ (\ ((\lambda u\,.\,\lambda v\,.\,\lambda w\,.\,\mathit{if}\ (=\ v\ w)\ u\ \mathit{fail})x\ y\ z)$$
$$[\!]\ \mathit{ERROR}$$
$$)$$

és erre a kifejezésre a $\bot, 1, 2$ kifejezéseket applikálva

$f'\ \bot\ 1\ 2 \equiv$
$(\lambda x\,.\,\lambda y\,.\,\lambda z\,.\ (\ ((\lambda u\,.\,\lambda v\,.\,\lambda w\,.\,\mathit{if}\ (=\ v\ w)\ u\ \mathit{fail})x\ y\ z)$
$[\!]\ \mathit{ERROR}$
$)\ )\bot\ 1\ 2\ \rightarrow^{+}$

($((\lambda u\,.\,\lambda v\,.\,\lambda w\,.\,\mathit{if}\,(=\,v\,w)\,u\,\mathit{fail})\bot\ 1\ 2)$
[] *ERROR*
) $\rightarrow^{+}$
($\mathit{if}\,(=\,1\ 2)\ \bot\ \mathit{fail}$)
[] *ERROR*
) $\rightarrow^{+}$
(*fail*
[] *ERROR*
) $\rightarrow$
ERROR

Az eredeti definícióval tehát a $\bot$, az átalakított kifejezéssel az *ERROR* eredményt kaptuk.

Látható tehát, hogy az őrfeltételes átalakítás nem oldotta meg az azonos minták problémáját. Jelenleg nincs általános módszer az azonos mintákat tartalmazó függvénydefiníciók fordítására, és ez az oka annak, hogy az ilyen függvénydefiníciók írását a programozási nyelvekben nem teszik lehetővé.

4. FEJEZET

A mintaillesztés programja

Az előző fejezetben láttuk, hogy az

$$
\begin{aligned}
&f\ p_{1,1}\ \ p_{1,2}\ \ \ldots p_{1,n}\ = E_1\\
&f\ p_{2,1}\ \ p_{2,2}\ \ \ldots p_{2,n}\ = E_2\\
&\ldots\\
&f\ p_{m,1}\ \ p_{m,2}\ \ldots p_{m,n}\ = E_m
\end{aligned}
$$

definíciókkal megadott f függvényre az

$$
\begin{aligned}
&f =\\
&\lambda x_1 x_2 \ldots x_n \,.\, (\ ((\lambda \mathrm{TE}(\!|\, p_{1,1} \,|\!) \,.\, \lambda \mathrm{TE}(\!|\, p_{1,2} \,|\!) \,.\, \cdots \,.\, \lambda \mathrm{TE}(\!|\, p_{1,n} \,|\!) \,.\, \mathrm{TE}(\!|\, E_1 \,|\!))\ x_1 x_2 \ldots x_n)\\
&\qquad\qquad [\!]\ ((\lambda \mathrm{TE}(\!|\, p_{2,1} \,|\!) \,.\, \lambda \mathrm{TE}(\!|\, p_{2,2} \,|\!) \,.\, \cdots \,.\, \lambda \mathrm{TE}(\!|\, p_{2,n} \,|\!) \,.\, \mathrm{TE}(\!|\, E_2 \,|\!))\ x_1 x_2 \ldots x_n)\\
&\qquad\qquad \ldots\\
&\qquad\qquad [\!]\ ((\lambda \mathrm{TE}(\!|\, p_{m,1} \,|\!) \,.\, \lambda \mathrm{TE}(\!|\, p_{m,2} \,|\!) \,.\, \cdots \,.\, \lambda \mathrm{TE}(\!|\, p_{m,n} \,|\!) \,.\, \mathrm{TE}(\!|\, E_m \,|\!))\ x_1 x_2 \ldots x_n)\\
&\qquad\qquad [\!]\ \mathit{ERROR}\\
&\qquad\qquad)
\end{aligned}
$$

absztrakciót kaptuk. Már ismerve a mintaabsztrakciók jelentését is, megállapíthatjuk, hogy ennek a λ-kifejezésnek a redukálási sorozatában a mintaillesztéshez rendkívül sok és gyakran ismétlődő műveletet kell elvégezni. Egy minta felismerése rendkívül nehézkes és lassú. A $[\!]$ vastagvonal műveletek miatt a sorokban leírt minták illesztése a kifejezésre szekvenciálisan történik, ha a minta konstruktora megegyezik a kifejezés konstruktorával, akkor az argumentumok illesztése következik, ez is szekvenciális, és előfordulhat, hogy csak a konstruktor utolsó argumentumánál kapunk *fail* eredményt. Előfordulhat az is, hogy a definíció különböző soraiban azonos minták is vannak, ez azt jelenti, hogy ugyanazt a mintaillesztést többször is el kell végezni. Pesszimális esetben még az is előfordulhat, hogy mindegyik sor konstruktorát, és mindegyik sorban a konstruktor mindegyik argumentumát ellenőrizni kell.

Sokkal hatékonyabb mintaillesztő programot kapnánk, ha a számítógép case-

utasítását tudnánk használni. Természetesen a case-utasítás is végez összehasonlításokat, de sokkal alacsonyabb szinten, a „gépi kód” szintjén. A felhasználó számára a case-utasítás egy többirányú elágaztató utasítás, amelyik „egy lépésben” ki tudja választani a megfelelő ugráscímet, azaz számunkra a megadott m sor közül az illeszthető $p_{i,1}$ $(1 \leq i \leq m)$ konstruktort. A case-utasítást megvalósító λ-kifejezés a case-kifejezés. Ennek felhasználásával a mintaillesztés sokkal egyszerűbb és gyorsabb lesz annál, mint ha a felhasználó saját maga írja meg valamilyen magasszintű programnyelven a mintaillesztés műveleteit.

4.1. A *match* függvény

Az egyszerűbb leírás érdekében jelöljük a $\lambda\mathrm{TE}(\!| p_{i,j} |\!)$ kifejezést $\lambda q_{i,j}$-vel, a $\mathrm{TE}(\!| E_i |\!)$ kifejezést F_i-vel, így a fenti f függvény

$$
\begin{aligned}
&f = \\
&\lambda x_1 x_2 \ldots x_n \,.\, (\ ((\lambda q_{1,1} \,.\, \lambda q_{1,2} \,.\, \cdots \,.\, \lambda q_{1,n} \,.\, F_1)\ x_1 x_2 \ldots x_n) \\
&\qquad\qquad [\!] \ ((\lambda q_{2,1} \,.\, \lambda q_{2,2} \,.\, \cdots \,.\, \lambda q_{2,n} \,.\, F_2)\ x_1 x_2 \ldots x_n) \\
&\qquad\qquad \ldots \\
&\qquad\qquad [\!] \ ((\lambda q_{m,1} \,.\, \lambda q_{m,2} \,.\, \cdots \,.\, \lambda q_{m,n} \,.\, F_m)\ x_1 x_2 \ldots x_n) \\
&\qquad\qquad [\!] \ \mathit{ERROR} \\
&\qquad\qquad)
\end{aligned}
$$

alakú λ-kifejezés lesz.

Egyszerűsítsük a λ-absztrakció törzsének leírását egy *match* háromváltozós függvény bevezetésével. A *match* a kibővített λ-kalkulusnak ebben a fejezetben ideiglenesen használt segédkifejezése. *Operációs szemantikáját*, azaz redukciós szabályait a következő pontokban adjuk meg, és majd látni fogjuk, hogy a *match* kifejezés applikációit olyan kifejezésekre alakítjuk át, amelyekben már csak a kibővített λ-kalkulus case-kifejezései szerepelnek.

A *match*

- első argumentuma legyen az f függvény $x_1 x_2 \ldots x_n$ változóinak listája,
- a második a $([q_{i,1} q_{i,2} \ldots q_{i,n}], F_i)$ párokból $i = 1, 2, \ldots, m$-re készített lista,
- a harmadik argumentum pedig legyen az a kifejezés, amelyiket arra az esetre adunk meg, amikor az összes mintaillesztés eredménye *fail*, ez az f kifejezésében az *ERROR* konstans.

Így az f függvénydefiníciót a következő kifejezés írja le:

$$
\begin{aligned}
f = \lambda x_1 x_2 \ldots x_n \,.\ & \mathit{match}\ [x_1 x_2 \ldots x_n] \\
& [\ ([q_{1,1}\ q_{1,2} \ldots q_{1,n}], \quad F_1), \\
& \ \ ([q_{2,1}\ q_{2,2} \ldots q_{2,n}], \quad F_2), \\
& \ \ \ldots \\
& \ \ ([q_{m,1}\ q_{m,2} \ldots q_{m,n}],\ F_m)\ ] \\
& \mathit{ERROR}
\end{aligned}
$$

Az átalakítás szabálya tehát a következő:

$$
\begin{aligned}
& \lambda x_1 x_2 \ldots x_n \,.\ (\ ((\lambda q_{1,1} \,.\, \lambda q_{1,2} \,.\, \cdots \,.\, \lambda q_{1,n} \,.\, F_1)\ x_1 x_2 \ldots x_n) \\
& \qquad [\!] \ ((\lambda q_{2,1} \,.\, \lambda q_{2,2} \,.\, \cdots \,.\, \lambda q_{2,n} \,.\, F_2)\ x_1 x_2 \ldots x_n) \\
& \qquad \ldots \\
& \qquad [\!] \ ((\lambda q_{m,1} \,.\, \lambda q_{m,2} \,.\, \cdots \,.\, \lambda q_{m,n} \,.\, F_m)\ x_1 x_2 \ldots x_n) \\
& \qquad [\!] \ \mathit{ERROR} \\
& \qquad)
\end{aligned}
$$

$$\Longrightarrow$$

$$
\begin{aligned}
\lambda x_1 x_2 \ldots x_n \,.\ & \mathit{match}\ [x_1 x_2 \ldots x_n] \\
& [\ ([q_{1,1}\ q_{1,2} \ldots q_{1,n}], \quad F_1), \\
& \ \ ([q_{2,1}\ q_{2,2} \ldots q_{2,n}], \quad F_2), \\
& \ \ \ldots \\
& \ \ ([q_{m,1}\ q_{m,2} \ldots q_{m,n}],\ F_m)\ ] \\
& \mathit{ERROR}
\end{aligned}
$$

4.1.1. Példa. *(Az add_pair kifejezés)*

A 3.7.8. példában láttuk, hogy az

$\mathit{add\_pair}\ (\mathit{pair}\ x\ y) = +\ x\ y$

definíciót a kibővített λ-kalkulus kifejezésére alakítva az

$$
\begin{aligned}
\mathit{add\_pair} = \lambda z \,.\ & (\ ((\lambda(\mathit{pair}\ x\ y) \,.\ +\ x\ y)\ z) \\
& \qquad [\!] \ \mathit{ERROR} \\
& \qquad)
\end{aligned}
$$

kifejezést kaptuk. A definíció a *match* kifejezéssel a következő:

$\mathit{add\_pair} =$

λz . *match* [z]
[([*pair* x y], + x y)]
ERROR □

4.1.2. Példa. *(A reflect kifejezés)*
A 3.7.5. példában levő

reflect (*leaf* n) = *leaf* n
reflect (*branch* t_1 t_2) = *branch* (*reflect* t_2) (*reflect* t_1)

függvény alakja a kibővített λ-kalkulusban a következő volt:

reflect =
λx . ((((λ(*leaf* n) . *leaf* n) x)
▯ ((λ(*branch* t_1 t_2) . *branch* (*reflect* t_2) (*reflect* t_1)) x)
▯ *ERROR*
)

és ez a *match* kifejezéssel:

reflect =
λx . *match* [x]
[([*leaf* n)], *leaf* n)
([*branch* t_1 t_2], *branch* (*reflect* t_2) (*reflect* t_1)]
ERROR □

4.1.3. Példa. *(A maplist kifejezés)*
A *maplists* függvény definíciója legyen a következő:

maplists f [ ] y:ys = [ ]
maplists f x:xs [ ] = [ ]
maplists f x:xs y:ys = (f x y):(*maplists* f xs ys)

A *maplists* függvény λ-kifejezése a már jól ismert átalakítással:

maplists =
$\lambda x_1 x_2 x_3$. ((((λf . λ*Nil* . λ(*cons* y ys) . *Nil*)$x_1 x_2 x_3$)
▯ ((λf . λ(*cons* x xs) . λ*Nil* . *Nil*)$x_1 x_2 x_3$)
▯ ((λf . λ(*cons* x xs) . λ(*cons* y ys) .
cons (f x y) (*maplists* f xs ys))$x_1 x_2 x_3$)
▯ *ERROR*
)

és ezt a *match* függvénnyel kifejezve a

$$
\begin{array}{l}
\mathit{maplists} = \\
\lambda x_1 x_2 x_3 .\ \mathit{match}\ [x_1 x_2 x_3] \\
\qquad\qquad [\ ([f\ \mathit{Nil}\ \ (\mathit{cons}\ y\ ys)],\ \mathit{Nil}) \\
\qquad\qquad\ \ ([f\ (\mathit{cons}\ x\ xs)\ \mathit{Nil}],\ \ \mathit{Nil}) \\
\qquad\qquad\ \ ([f\ (\mathit{cons}\ x\ xs)\ (\mathit{cons}\ y\ ys)],\ \mathit{cons}\ (f\ x\ y)\ (\mathit{maplists}\ f\ xs\ ys))\ ] \\
\qquad\qquad \mathit{ERROR}
\end{array}
$$

kifejezést kapjuk. □

4.1.1. A változó szabálya

Ha a *match* kifejezés első argumentuma $[x_1 x_2 \cdots]$, második argumentumának egy sora egy változóval kezdődik, például

$$([x\ q \ldots], F)$$

alakú, akkor ez egy

$$(\lambda x .\ \lambda q .\ \cdots .\ F) x_1 x_2 \ldots$$

alakú mintaillesztésből származik. Mivel x egy változó, ez a kifejezés egy egyszerű β-redukcióval redukálható, azaz egyszerűbb alakra hozható:

$$
\begin{array}{l}
(\lambda x .\ (\lambda q .\ \cdots .\ F)) x_1 x_2 \ldots \ \rightarrow \\
((\lambda q .\ \cdots .\ F)[x := x_1]) x_2 \ldots
\end{array}
$$

Mivel a mintákban változó-duplikáció nem lehetséges, a redukció eredménye

$$(\lambda q .\ \cdots .\ F[x := x_1]) x_2 \ldots$$

lesz. Ezt a műveletet a *match* argumentumaként írva a

$$
\begin{array}{l}
([x\ q \ldots], F) \Longrightarrow \\
([q \ldots], F[x := x_1])
\end{array}
$$

átalakítást kapjuk. Az átalakításból az is látszik, hogy a sor egy elemmel rövidebb lesz és az $[x_1 x_2 \ldots]$ argumentum az $[x_2 \ldots]$ sorozatra csökken.

Ahhoz, hogy az ilyen típusú redukció ne csak egy sorra, hanem a *match* második argumentumának mindegyik sorára elvégezhető legyen, meg kell követelnünk, hogy a második argumentum mindegyik sorában az első komponens egy változó legyen. Természetesen ezeknek a változóknak nem kell azonosaknak lenniük, mindegyik sorban az ott levő első változóval kell a redukciót végrehajtani. Az

átalakítást *változó-szabálynak* nevezzük.

$$
\begin{array}{l}
\mathit{match}\ [x_1 x_2 \ldots x_n] \\
\qquad [\ ([y_1\ q_{1,2} \ldots q_{1,n}],\ \ F_1), \\
\qquad\ \ ([y_2\ q_{2,2} \ldots q_{2,n}],\ \ F_2), \\
\qquad\ \ \ldots \\
\qquad\ \ ([y_m\ q_{m,2} \ldots q_{m,n}],\ F_m)\ ] \\
\qquad \mathit{ERROR} \\
\Longrightarrow \\
\mathit{match}\ [x_2 \ldots x_n] \\
\qquad [\ ([q_{1,2} \ldots q_{1,n}],\ \ F_1[y_1 := x_1]), \\
\qquad\ \ ([q_{2,2} \ldots q_{2,n}],\ \ F_2[y_2 := x_1]), \\
\qquad\ \ \ldots \\
\qquad\ \ ([q_{m,2} \ldots q_{m,n}],\ F_m[y_m := x_1])\ ] \\
\qquad \mathit{ERROR}
\end{array}
$$

4.1.4. Példa. *(A változó szabálya)*
A 4.1.3. példában adtuk meg a *maplists* függvény *match* alkalmazásával leírt alakját:

$$
\begin{array}{l}
\mathit{maplists} = \\
\lambda x_1 x_2 x_3\ .\ \mathit{match}\ [x_1 x_2 x_3] \\
\qquad\qquad [\ ([f\ \mathit{Nil}\ \ \ (\mathit{cons}\ y\ ys)],\ \mathit{Nil}) \\
\qquad\qquad\ \ ([f\ (\mathit{cons}\ x\ xs)\ \mathit{Nil}],\ \ \ \mathit{Nil}) \\
\qquad\qquad\ \ ([f\ (\mathit{cons}\ x\ xs)\ (\mathit{cons}\ y\ ys)],\ \mathit{cons}\ (f\ x\ y)\ (\mathit{maplists}\ f\ xs\ ys))\ ] \\
\qquad\qquad \mathit{ERROR}
\end{array}
$$

Látható, hogy a második argumentum mindegyik sorának első eleme az f változó, így alkalmazható a változókra vonatkozó átalakítási szabály:

$$
\begin{array}{l}
\mathit{maplists} = \\
\lambda x_1 x_2 x_3\ .\ \mathit{match}\ [x_2 x_3] \\
\qquad\qquad [\ ([\mathit{Nil}\ \ \ (\mathit{cons}\ y\ ys)],\ \mathit{Nil}) \\
\qquad\qquad\ \ ([(\mathit{cons}\ x\ xs)\ \mathit{Nil}],\ \ \ \mathit{Nil}) \\
\qquad\qquad\ \ ([(\mathit{cons}\ x\ xs)\ (\mathit{cons}\ y\ ys)],\ \mathit{cons}\ (\underline{x_1}\ x\ y)\ (\mathit{maplists}\ \underline{x_1}\ xs\ ys))\ ] \\
\qquad\qquad \mathit{ERROR}
\end{array}
$$

□

4.1.2. A konstruktor szabálya

Ha a *match* kifejezés első argumentuma $[x_1 x_2 x_3 \dots]$, és a kifejezés második argumentumának egy sora a c konstruktorral kezdődik, például

$$([(c\ p_1 p_2 \dots)\ q_2 q_3 \dots], F)$$

alakú, akkor ez a sor egy

$$(\lambda(c\ p_1 p_2 \dots)\,.\,\lambda q_2\,.\,\lambda q_3\,.\,\cdots\,.\,F) x_1 x_2 x_3 \dots$$

alakú mintaillesztésből származik. Ismerve a konstruktoros absztrakciókra vonatkozó mintaillesztések szabályait, a sikeres mintaillesztéshez x_1-nek, az első argumentumnak $c\ F_1 F_2 F_3 \dots$ alakúnak kell lennie.

Ez azt jelenti, hogy ennek a sornak a mintaillesztését biztosan a c konstruktorral kell elvégezni, és a mintaillesztés biztosan csak akkor lesz sikeres, ha az x_1 argumentumban is a c konstruktor szerepel. Tehát ennek a sornak a vizsgálatát hozzárendelhetjük a c konstruktorhoz, a sor kiválasztását egy olyan *case* kifejezéssel végezhetjük el, amelyben az elágazások az x_1 konstruktora szerint történnek. A fenti

$$([(c\ p_1 p_2 \dots)\ q_2 q_3 \dots], F)$$

sorhoz a

case x_1 *of*
  $c\ p_1 p_2 \dots$: ...

kifejezést rendelhetjük hozzá. Az is nyilvánvaló, hogy ennek a sornak a további feldolgozása független lesz a többi sorban leírtaktól, tehát erre a sorra egy önálló *match* kifejezést adhatunk meg. A *match* változói nem csak a hátralévő $q_2 q_3 \dots$ feldolgozásához szükséges $x_2 x_3 \dots$ változók lesznek, hanem a c mintaillesztéséből adódó $p_1 p_2 \dots$ paraméterek feldolgozásához kapcsolódó új $u_1 u_2 \dots$ változóknak is itt kell szerepelniük. Befejezhetjük tehát a

$$([(c\ p_1 p_2 \dots)\ q_2 q_3 \dots], F)$$

sor kifejezését:

case x_1 *of*
  $c\ p_1 p_2 \dots$: *match* $[u_1 u_2 \dots x_2 x_3 \dots]$
    $[([p_1 p_2 \dots q_2 q_3 \dots], F)]$
    ERROR

Ha a *match* második argumentumának minden sora konstruktorral kezdődik,

akkor a *match* kifejezés átalakítása a következő lesz. Az átalakítást *konstruktor-szabálynak* nevezzük.

match $[x_1 x_2 \ldots x_n]$
$\quad$ [$([(c_1\ p_{1,1} \ldots p_{1,r_1})q_{1,2} \ldots q_{1,n}],\quad E_1)$,
$\quad\quad ([(c_2\ p_{2,1} \ldots p_{2,r_2})q_{2,2} \ldots q_{2,n}],\quad E_2)$,
$\quad\quad \ldots$
$\quad\quad ([(c_m\ p_{m,1} \ldots p_{m,r_m})q_{m,2} \ldots q_{m,n}],\ E_m)$]
$\quad$ *ERROR*
$\Longrightarrow$
case x_1 *of*
$c_1\ u_{1,1} \ldots u_{1,r_1}$: *match* $[u_{1,1} \ldots u_{1,r_1}, x_2 \ldots x_n]$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ [$([p_{1,1} \ldots p_{1,r_1} q_{1,2} \ldots q_{1,n}],\quad E_1)$]
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ *ERROR*
$c_2\ u_{2,1} \ldots u_{2,r_2}$: *match* $[u_{2,1} \ldots u_{2,r_2}, x_2 \ldots x_n]$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ [$([p_{2,1} \ldots p_{2,r_2} q_{2,2} \ldots q_{2,n}],\quad E_2)$]
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ *ERROR*
$\ldots$
$c_m\ u_{m,1} \ldots u_{m,r_m}$: *match* $[u_{m,1} \ldots u_{m,r_m}, x_2 \ldots x_n]$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ [$([p_{m,1} \ldots p_{m,r_m} q_{m,2} \ldots q_{m,n}],\ E_m)$]
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ *ERROR*

ahol $u_{1,1}, \ldots u_{m,r_m}$ *új változók.*

Megállapíthatjuk, hogy a konstruktorok szabálya szerint az eredeti *match* kifejezés [] műveletei eltűnnek, a mintaillesztést a konstruktorok szintjén a *case* kifejezés végzi el, és a bonyolult többsoros, sok mintaillesztést leíró *match* kifejezés soronként, azaz elágazásokként önálló, lényegesen egyszerűbb *match* kifejezésekre bomlik fel.

Mivel a konstansok nulla aritású konstruktorok, ez a módszer a *konstansok* mintaillesztésére is alkalmazható.

Előfordulhat, hogy ugyanaz a mintakonstruktor több mintaillesztésben is szerepel, azaz a *match* második argumentumának több sorában is első helyen van ugyanaz a konstruktor. Ha egy ilyen kifejezésre végrehajtanánk a konstruktor-szabályban leírt átalakítást, egy olyan kifejezést kapnánk, amelyben a *case* elágazások nem lennének egyértelműek, mivel ugyanarra a konstruktorra több elágazás is meg lenne adva.

Ezért először rendezzük át a mintaillesztések sorait úgy, hogy az azonos kon-

struktorok egymás utáni sorokban szerepeljenek, de a *funkcionális kiértékelés* elvének betartása érdekében ezen sorok sorrendje ne változzon meg. Az azonos konstruktorú mintákat ezután egyetlen *case-ágba* foghatjuk össze úgy, hogy a megkülönböztetést a konstruktor paramétereinek feldolgozására halasztjuk el. Tehát a

$$
\begin{array}{l}
\mathit{match}\ [x_1 x_2 \ldots x_n] \\
\quad [\ \ldots \\
\quad\ ([(c\ p_{i,1} \ldots p_{i,r_i}) q_{i,2} \ldots q_{i,n}], E_i), \\
\quad\ ([(c\ p_{i+1,1} \ldots p_{i+1,r_i}) q_{i+1,2} \ldots q_{i+1,n}], E_{i+1}), \\
\quad\ \ldots\ ] \\
\quad \mathit{ERROR}
\end{array}
$$

kifejezésből a

$$
\begin{array}{l}
\mathit{case}\ x_1\ \mathit{of} \\
\quad \ldots \\
\quad c\ u_{i,1} \ldots u_{i,r_i}\ :\ \mathit{match}\ [u_{i,1} \ldots u_{i,r_1}, x_2 \ldots x_n] \\
\qquad\qquad\qquad\qquad [\ ([p_{i,1} \ldots p_{i,r_i}\ q_{i,2} \ldots q_{i,n}], E_i) \\
\qquad\qquad\qquad\qquad\ \ ([p_{i+1,1} \ldots p_{i+1,r_i}\ q_{i+1,2} \ldots q_{i+1,n}], E_{i+1})\ ] \\
\qquad\qquad\qquad\qquad \mathit{ERROR} \\
\quad \ldots
\end{array}
$$

kifejezést állítjuk elő.

4.1.5. Példa. *(A konstruktor szabálya)*

A 4.1.4. példában eredményül kapott kifejezésre:

$$
\begin{array}{l}
\mathit{maplists} = \\
\lambda x_1 x_2 x_3\ .\ \mathit{match}\ [x_2 x_3] \\
\qquad\qquad [\ ([\mathit{Nil} \quad\quad (\mathit{cons}\ y\ ys)], \mathit{Nil}) \\
\qquad\qquad\ \ ([(\mathit{cons}\ x\ xs)\ \mathit{Nil}], \quad\quad \mathit{Nil}) \\
\qquad\qquad\ \ ([(\mathit{cons}\ x\ xs)\ (\mathit{cons}\ y\ ys)], \mathit{cons}\ (x_1\ x\ y)\ (\mathit{maplists}\ x_1\ xs\ ys))\ ] \\
\qquad\qquad \mathit{ERROR} \\
\Longrightarrow \\
\lambda x_1 x_2 x_3\ .
\end{array}
$$

```
case x2 of
  Nil       : match [x3]
                    [ ([cons y ys],          Nil) ]
                    ERROR
  cons u1 u2 : match [u1 u2 x3]
                    [  ([x xs Nil],          Nil)
                      ([x xs (cons y ys) ], cons (x1 x y) (maplists x1 xs ys)) ]
                    ERROR
```

A *Nil*-ágban egy egysoros mintaillesztés van, ennek átalakítása:

```
match [x3]
      [ ([cons y ys], Nil) ]
      ERROR
⟹
case x3 of
  cons u3 u4 : match [u3 u4]
                    [ ([y ys], Nil) ]
                    ERROR
```

A változó-szabályt az *y*-ra, majd az *ys*-re alkalmazva a

```
case x3 of
  cons u3 u4 : match [ ]
                    [ ([ ], Nil) ]
                    ERROR
```

kifejezést kapjuk.

A *maplists* kifejezés case-kifejezésének *cons*-ágán is kétszer alkalmazhatjuk a változó-szabályt, először az *x*, majd az *xs* változóra:

```
match [u1 u2 x3]
      [  ([x xs Nil], Nil)
        ([x xs (cons y ys)], cons (x1 x y) (maplists x1 xs ys)) ]
      ERROR
⟹+
```

match $[x_3]$
  [([*Nil*], *Nil*)
    ([(*cons* $y\ ys$)], *cons* $(x_1\ u_1\ y)$ (*maplists* $x_1\ u_2\ ys$))]
  ERROR

Ebből a konstruktor-szabály alkalmazásával:

case x_3 *of*
  Nil : *match* [ ]
    [([ ], *Nil*)]
    ERROR
  cons $u_5\ u_6$: *match* $[u_5\ u_6]$
    [([$y\ ys$], *cons* $(x_1\ u_1\ y)$ (*maplists* $x_1\ u_2\ ys$))]
    ERROR

Ez utóbbira ismét a változó-szabály kétszeri alkalmazásával:

match $[u_5\ u_6]$
  [([$y\ ys$], *cons* $(x_1\ u_1\ y)$ (*maplists* $x_1\ u_2\ ys$))]
  ERROR
$\Longrightarrow$
match [ ]
  [([ ], *cons* $(x_1\ u_1\ u_5)$ (*maplists* $x_1\ u_2\ u_6$))]
  ERROR

Összefoglalva az eddigi lépéseket, a *maplists* függvényre a következő átalakítást kapjuk:

maplists f [ ] $y{:}ys$ = [ ]
maplists f $x{:}xs$ [ ] = [ ]
maplists f $x{:}xs$ $y{:}ys$ = $(f\ x\ y){:}$(*maplists* f $xs\ ys$)

$\Longrightarrow^{+}$

maplists =
$\lambda x_1 x_2 x_3$.

case x_2 *of*
  Nil : *case* x_3 *of*
    cons u_3 u_4 : *match* [ ]
      [([ ], *Nil*)]
      ERROR
  cons u_1 u_2 : *case* x_3 *of*
    Nil : *match* [ ]
      [([ ], *Nil*)]
      ERROR
    cons u_5 u_6 : *match* [ ]
      [([ ], *cons* (x_1 u_1 u_5) (*maplists* x_1 u_2 u_6))]
      ERROR

Látható, hogy a *maplists* eredményül kapott kifejezésében a case-kifejezések mindegyikére olyan *match*-kifejezést kaptunk, melyben a változók és a minták listája üres. □

4.1.6. Példa. *(Konstansok, mint konstruktorok)*
A 3.5.9. példában az

$f\ \_\ 0\ \_ = 1$
$f\ \_\ \_\ 0 = 2$
$f\ 0\ \_\ \_ = 3$

függvénydefinícióból a következő λ-kifejezést határoztuk meg:

$f =$
$f = \lambda xyz.\ (\ ((\lambda\_\ .\ \lambda 0\ .\ \lambda\_\ .\ 1)xyz)$
$\quad [\!]\ ((\lambda\_\ .\ \lambda\_\ .\ \lambda 0\ .\ 2)xyz)$
$\quad [\!]\ ((\lambda 0\ .\ \lambda\_\ .\ \lambda\_\ .\ 3)xyz)$
$\quad [\!]$ *ERROR*
$\quad)$

Ezt a kifejezést átalakítva

match $[xyz]$
  [([_ 0 _], 1)
    ([_ _ 0], 2)
    ([0 _ _], 3)]
  ERROR

Mivel a konstansok nulla aritású konstruktoroknak tekinthetők, alkalmazhatjuk a konstruktorok szabályát. Mivel az első és a második sorban az első konstansok azonosak, ezeket a sorokat egy *case-ágba* fogjuk össze.

```
case x of
  _ : match [y z]
            [ ([ 0 _ ], 1)
              ([ _ 0 ], 2) ]
            ERROR
  0 : match [y z]
            [ ([ _ _ ], 3) ]
            ERROR
```

A *match* kifejezéseket tovább alakítva, az első ág kifejezése:

```
case y of
  0 : match [z]
            [ ([ _ ], 1) ]
            ERROR
  _ : match [z]
            [ ([ 0 ], 2) ]
            ERROR
⟹
case y of
  0 : case z of
        _ : match  [ ]
                   [ ([ ], 1) ]
                   ERROR
  _ : case z of
        0 : match [ ]
                  [ ([ ], 2) ]
                  ERROR
```

és a második ág kifejezése:

```
case y of
```

```
  _: match [z]
           [ ([_], 3) ]
           ERROR
⟹
case y of
  _ : case z of
        _ :match [ ]
                 [ ([ ], 3) ]
                 ERROR
```

Tehát végeredményül a

```
case x of
  _ : case y of
        0 : case z of
              _ :match  [ ]
                        [ ([ ], 1) ]
                        ERROR
        _ : case z of
              0:  match [ ]
                        [ ([ ], 2) ]
                        ERROR
  0:  case y of
        _ : case z of
              _ :match [ ]
                       [ ([ ], 3) ]
                       ERROR
```

kifejezést kaptuk. A kifejezés helyes működéséhez a *case* végrehajtását módosítani kell, ezzel a problémával majd a 4.1.10. példában foglalkozunk. □

4.1.3. Az üres minta szabálya

A következő kifejezésben nincs mivel mintaillesztést végezni:

match [ ]
[([ ], F)]
ERROR

ezért ez a kifejezés az F kifejezésre egyszerűsíthető, hiszen mintaillesztés hiányában *fail* eredményt sem kaphatunk, és így az *ERROR* konstansra sincs szükség.

match [ ]
[([ ], F)]
ERROR $\Longrightarrow$ F

Elképzelhető, hogy egy *match* kifejezés átalakításakor egy

match [ ]
[([ ], F_1)
([ ], F_2)
...
([ ], F_m)]
ERROR

szerkezetű kifejezést kapunk. Ez az eset például akkor áll elő, ha a függvénydefinícióban azonos mintákat adunk meg. Ennek a kifejezésnek a redukciója

$F_1 \,[\!]\, F_2 \,[\!]\, \ldots \,[\!]\, F_m \,[\!]\, \mathit{ERROR}$,

de ha biztosítjuk, hogy az $F_1, F_2, \ldots, F_m$ egyike sem lehet *fail*, akkor $m \geq 1$ esetén a *funkcionális kiértékelés* miatt a redukció eredménye F_1, $m = 0$ esetén pedig az *ERROR*.

4.1.7. Példa. *(Az add_pair kifejezés* case *művelettel)*

A 4.1.1. példában láttuk, hogy

add_pair =
λz . *match* [z]
[([*pair* x y], + x y)]
ERROR

Ezt a kifejezést tovább alakítva:

add_pair =

λz . *case* z *of*
  pair $u\ v$: *match* $[u, v]$
    $[\ ([x\ y], +\ x\ y)\ ]$
    ERROR
$\Longrightarrow$
λz . *case* z *of*
  pair $u\ v$: *match* $[v]$
    $[\ ([y], +\ u\ y)\ ]$
    ERROR
$\Longrightarrow$
λz . *case* z *of*
  pair $u\ v$: *match* $[\ ]$
    $[\ ([\ ], +\ u\ v)\ ]$
    ERROR
$\Longrightarrow$
λz . *case* z *of*
  pair $u\ v$: $+\ u\ v$ □

4.1.8. Példa. *(A reflect kifejezés* case *művelettel)*
A 4.1.2. példában szerepelt a *reflect* definíciójának a következő alakja:

reflect =
λx . ($((\lambda(\mathit{leaf}\ n)\ .\ \mathit{leaf}\ n)\ x)$
  ▯ $((\lambda(\mathit{branch}\ t_1\ t_2)\ .\ \mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x)$
  ▯ *ERROR*
)

Ezt a kifejezést most tovább alakítva:

reflect =
λx . *case* x *of*
  leaf u: *match* $[u]$
    $[\ ([n], \mathit{leaf}\ n)\ ]$
    ERROR
  branch $v\ w$: *match* $[v, w]$
    $[\ ([t_1, t_2], \mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1)\ )\ ]$
    ERROR

$\Longrightarrow^{+}$

$\lambda x\,.$ *case* x *of*
  leaf u: *match* [ ]
    [([ ], *leaf* u)]
    ERROR
  branch $v\ w$: *match* [ ]
    [([ ], *branch* (*reflect* w) (*reflect* v))]
    ERROR

$\Longrightarrow^{+}$

$\lambda x\,.$ *case* x *of*
  leaf u: *leaf* u
  branch $v\ w$: *branch* (*reflect* w) (*reflect* v) □

4.1.9. Példa. *(Az üres-szabály alkalmazása)*

A 4.1.5. példa eredményként kapott kifejezésekre alkalmazzuk az üres-szabályban leírt átalakításokat:

maplists f [ ] $y{:}ys$ = [ ]
maplists $f\ x{:}xs$ [ ] = [ ]
maplists $f\ x{:}xs\ y{:}ys = (f\ x\ y){:}(maplists\ f\ xs\ ys)$

$\Longrightarrow^{+}$

maplists =
$\lambda x_1 x_2 x_3\,.$
case x_2 *of*
  Nil : *case* x_3 *of*
    cons $u_3\ u_4$: *match* [ ]
      [([ ], *Nil*)]
      ERROR
  cons $u_1\ u_2$: *case* x_3 *of*
    Nil : *match* [ ]
      [([ ], *Nil*)]
      ERROR
    cons $u_5\ u_6$: *match* [ ]
      [([ ], *cons* $(x_1\ u_1\ u_5)$ (*maplists* $x_1\ u_2\ u_6$))]
      ERROR

$\Longrightarrow^{+}$

$\lambda x_1 x_2 x_3$.
  case x_2 *of*
    Nil : *case* x_3 *of*
              cons u_3 u_4 : *Nil*
    cons u_1 u_2 : *case* x_3 *of*
              Nil : *Nil*
              cons u_5 u_6 : *cons* $(x_1\ u_1\ u_5)$ (*maplists* $x_1\ u_2\ u_6$)

Ezzel a 4.1.3. példában megadott *maplists* függvény case-kifejezésekre történő átalakítását be is fejeztük. □

4.1.10. Példa. *(Konstansok, mint konstruktorok)*
Ha az üres minta szabályát alkalmazzuk a 4.1.6. példában kapott eredményre, akkor az

$f\ \_\ 0\ \_ = 1$
$f\ \_\ \_\ 0 = 2$
$f\ 0\ \_\ \_ = 3$

függvénydefinícióra a következő kifejezést kapjuk:

case x *of*
  _ : *case* y *of*
      0 : *case* z *of*
          _ : *match* []
                [([ ], 1)
                ERROR
          _ : *case* z *of*
              0: *match* [ ]
                 [([ ], 2)]
                 ERROR
  0: *case* y *of*
      _ : *case* z *of*
          _ : *match* []
              [([ ], 3)]
              ERROR

$\Longrightarrow^{+}$
case x *of*

```
_ : case y of
      0 : case z of
            _ : 1
      _ : case z of
            0: 2
0: case y of
      _ : case z of
            _ : 3
```

A kifejezés helyes működéséhez a *case* végrehajtását módosítani kell, ugyanis ha

```
case x of
  _ : E
  ...
```

esetén az E végrehajtásakor *fail* eredményt kapunk, akkor ennek „tovább-ugrást” kell jelentenie, azaz vissza kell térni az x-szerinti következő case-ág vizsgálathoz. Ez egy verem használatával könnyen megvalósítható. □

4.1.4. Vegyes minták

Eddig a függvénydefinícióknak olyan megadásaival foglalkoztunk, ahol a minták vagy csak változókkal, vagy csak konstruktorokkal kezdődtek. Természetesen vegyes megadás is lehetséges, de erre az esetre az eddig megismert szabályok nem alkalmazhatók.

A *match* függvény megadásánál láttuk, hogy a harmadik argumentum kifejezése arra szolgál, hogy a második argumentumban leírt mintaillesztések sikertelensége esetén eredményül ezt a kifejezést kapjuk. Ezt, valamint a *match* függvények skatulyázhatóságát használjuk fel a vegyes minták kezelésére.

A *match* függvény harmadik argumentuma ne hibajelzés legyen, hanem ide írjuk be a következő típusú mintaillesztés kifejezését, azaz a megfelelő argumentumokkal együtt a következő típusú mintaillesztések *match* függvényét. Így ha az első típusú minta illesztése sikertelen volt, akkor nem hibajelzést kapunk, hanem a második típusú mintaillesztés végrehajtása fog megtörténni. A minta típusának újabb váltása esetén ennek a *match*-nek a harmadik argumentumába kerüljön be a harmadik típusú mintaillesztés kifejezése, és csak az utolsó típusú mintaillesztésben hagyjuk meg az *ERROR* konstanst.

4.1.11. Példa. *(A vegyes minták szabálya)*
Legyen a függvénydefiníció megadása a következő:

$mix\ x \quad = 1$
$mix\ (y{:}ys) = 2$

A *mix* λ-kifejezése:

$$
\begin{array}{l}
mix = \\
\lambda u\,.\ (\ ((\lambda x\,.\,1)u) \\
\qquad [\!] \ ((\lambda(y{:}ys)\,.\,2)u) \\
\qquad [\!] \ \mathit{ERROR} \\
\qquad)
\end{array}
$$

A függvény leírása a *match* függvénnyel:

$$
\begin{array}{l}
mix = \\
\lambda u\,.\,\mathit{match}\ [u] \\
\qquad\qquad [\ ([x], 1) \\
\qquad\qquad\ \ ([\mathit{cons}\ y\ ys], 2)\ ] \\
\qquad\qquad \mathit{ERROR}
\end{array}
$$

Látható, hogy a *cons y ys* vizsgálatot a *match* mintaillesztésének *fail* hiba-ágába építettük be.

Ebből a

$$
\begin{array}{l}
mix = \lambda u\,.\,\mathit{match}\ [u] \\
\qquad\qquad [\ ([x], 1)\ ] \\
\qquad\qquad (\ \mathit{match}\ [u] \\
\qquad\qquad\qquad\qquad [\ ([\mathit{cons}\ y\ ys], 2)\ ] \\
\qquad\qquad\qquad\qquad \mathit{ERROR}\ )
\end{array}
$$

skatulyázott *match* kifejezést kapjuk meg. □

5. FEJEZET

Kibővített λ-kalkulusból λ-kalkulusba

A 3. fejezetben megadtuk, hogy a funkcionális programot hogyan alakítjuk át a kibővített λ-kalkulus kifejezésére. Most azzal foglalkozunk, hogy a kibővített λ-kalkulus kifejezéseit hogyan tudjuk átalakítjuk a konstansos λ-kalkulus kifejezéseire. Ennek az átalakításnak az a szerepe, hogy ekkor a funkcionális program végrehajtása egyszerűen a λ-kalkulus redukciós lépéseivel valósítható meg. Ezt az átalakítást a $\rightsquigarrow$ jellel jelöljük.

5.1. Mintaabsztrakciók

A 3.5. szakaszban a mintákkal megadott függvénydefiníciókat mintaabsztrakciókra alakítottuk át, és a 3.7. szakaszban leírtuk a mintaabsztrakciók jelentését is. Most először azzal foglalkozunk, hogy a mintaabsztrakciók hogyan alakíthatók át a konstansos λ-kalkulus kifejezéseire.

5.1.1. Konstansos absztrakciók

Megismételjük a konstansos absztrakció jelentését, amit a 3.7.1. pontban az argumentum applikációjával adtunk meg:

$(\lambda k\,.\,E)F \;\rightarrow E, \qquad ha\ F \;\rightarrow^{*}\; k$

$(\lambda k\,.\,E)F \;\rightarrow \mathsf{fail}, \qquad ha\ F \;\not\rightarrow^{*}\; k\ és\ F \neq \bot$

$(\lambda k\,.\,E)F \;\rightarrow \bot, \qquad ha\ F = \bot$

Látható, hogy az applikáció kiértékelésekor egy összehasonlítást kell elvégezni, ilyen „összehasonlítás” művelet azonban nincs a λ-kalkulusban. Itt most konstansok összehasonlításáról van szó, amelyre a [2]-ben, a konstansos λ-kalkulusban megadtuk az equal λ-kifejezést, és amelyet az „=” jellel jelöltünk.

A konstansos λ-kalkulus *if* kifejezésével és az = felhasználásával a konstansos absztrakció könnyen átalakítható a λ-kalkulus kifejezésévé:

$\lambda k\,.\,E \quad \leadsto \quad \lambda x\,.\,\mathit{if}\,(=\ k\ x)\ E\ \mathit{fail}$
ahol x egy új változó

5.1.1. Példa. *(Konstansos absztrakció)*

A 3.5.4. példában láttuk, hogy a

$\mathit{flip}\ 0 = 1$
$\mathit{flip}\ 1 = 0$

egymintás, többsoros *flip* függvény λ-kifejezése kibővített λ-kalkulusban:

$$\begin{aligned}\mathit{flip} = \lambda x\,.\ (\ & ((\lambda 0\,.\,1)\ x)\\ \| \ & ((\lambda 1\,.\,0)\ x)\\ \| \ & \mathit{ERROR}\\ )\end{aligned}$$

A konstansos absztrakciók fenti átalakítási szabályával a

$$\begin{aligned}\mathit{flip} = \lambda x\,.\ (\ & ((\lambda y\,.\,\mathit{if}\,(=\ 0\ y)\ 1\ \mathit{fail})\ x)\\ \| \ & ((\lambda y\,.\,\mathit{if}\,(=\ 1\ y)\ 0\ \mathit{fail})\ x)\\ \| \ & \mathit{ERROR}\\ )\end{aligned}$$

kifejezést kapjuk. A kifejezésben még benne maradtak a $\|$ vastagvonal operátorok, de majd ennek a fejezetnek a végén látjuk, hogy ezek is átírhatók a konstansos λ-kalkulus kifejezésére.

Határozzuk meg a *flip* 0 kifejezés értékét. A levezetésből látható, hogy a $\|$ műveleten kívül csak β- és δ-redukciókat kell alkalmaznunk:

$$\begin{aligned}&\mathit{flip}\ \ 0 \equiv\\ &\lambda x\,.\ (\ ((\lambda y\,.\,\mathit{if}\,(=\ 0\ y)\ 1\ \mathit{fail})\ x)\\ &\quad \| \ ((\lambda y\,.\,\mathit{if}\,(=\ 1\ y)\ 0\ \mathit{fail})\ x)\\ &\quad \| \ \mathit{ERROR}\\ &\quad)\ 0\\ &\rightarrow\end{aligned}$$

(((λ*y* . *if* (= 0 *y*) 1 *fail*) 0)
[] ((λ*y* . *if* (= 1 *y*) 0 *fail*) 0)
[] *ERROR*
)
→
((*if* (= 0 0) 1 *fail*)
[] (*if* (= 1 0) 0 *fail*)
[] *ERROR*
)
→
(1
[] (*if* (= 1 0) 0 *fail*)
[] *ERROR*
)
→ 1 □

5.1.2. Összegkonstruktoros absztrakciók

A $\lambda(s\ p_1\ p_2 \ldots p_n)\,.\,E$ absztrakciót, ahol s egy összegkonstruktor és $p_1, p_2, \ldots, p_n$ a konstruktor argumentumai, összegkonstruktoros absztrakciónak neveztük.

Az ilyen mintaabsztrakció jelentését a 3.7.2. pontban a következőképpen adtuk meg:

$$(\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E)\ (s\ F_1 F_2 \cdots F_n) \quad \to (\lambda p_1\,.\,\lambda p_2\,.\,\cdots\,.\,\lambda p_n\,.\,E)\ F_1 F_2 \ldots F_n$$
$$(\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E)\ (s'\ F_1 F_2 \ldots F_m) \to \mathit{fail}, \qquad \mathit{ha}\ s \neq s'$$
$$(\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E)\ (s'\ F_1 F_2 \ldots F_m) \to \bot, \qquad \mathit{ha}\ s'\ F_1 F_2 \ldots F_m = \bot$$

A konstansos absztrakcióhoz hasonlóan, itt is az absztrakciókban szereplő és az applikált kifejezésben levő összegkonstruktorok összehasonlítása okoz problémát. Az s és s' konstruktorok összehasonlítását építsük be egy *UPS_s* függvénybe (a függvény neve az „UnPack-Sum” kifejezésből származik). Az s-től függő *UPS_s* függvény a konstansos λ-kalkulus δ-függvényének tekinthető. A csak s-ben különböző *UPS_s* függvények darabszámának csökkentésével később foglalkozunk.

Az összegkonstruktoros mintaabsztrakciót alakítsuk át az *UPS_s* függvénynek a felhasználásával:

$$\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E \quad \rightsquigarrow \quad \mathit{UPS\_s}\ (\lambda p_1\,.\,\lambda p_2\,.\,\cdots\,.\,\lambda p_n\,.\,E)$$

Látható, hogy a konstruktoros absztrakció egyszerűbb lett, hiszen a jobboldalon az

absztrakcióban már nincs s konstruktor.

Ha az *UPS_s* függvény $\lambda p_1 . \lambda p_2 . \cdots . \lambda p_n . E$ argumentumát f-fel jelöljük, akkor összegkonstruktoros absztrakció jelentése a következőképpen adható meg:

$$
\begin{array}{lcll}
\mathit{UPS\_s}\ f\ (s\ F_1 F_2 \ldots F_n) & \to & f\ F_1 F_2 \ldots F_n & \\
\mathit{UPS\_s}\ f\ (s'\ F_1 F_2 \ldots F_m) & \to & \mathit{fail}, & \text{ha } s \neq s' \\
\mathit{UPS\_s}\ f\ (s'\ F_1 F_2 \ldots F_m) & \to & \bot, & \text{ha } s'\ F_1 F_2 \ldots F_m = \bot
\end{array}
$$

5.1.2. Példa. *(Mintaillesztés összegkonstruktoros absztrakciókkal)*
A 3.5.5. példában láttuk, hogy az összegkonstruktorokkal megadott

$$
\begin{aligned}
&\mathit{reflect}\ (\mathit{leaf}\ n) &&= \mathit{leaf}\ n \\
&\mathit{reflect}\ (\mathit{branch}\ t_1\ t_2) &&= \mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1)
\end{aligned}
$$

függvény alakja a kibővített λ-kalkulusban a következő volt:

$$
\begin{aligned}
\mathit{reflect} = \lambda x . \ (\ & ((\lambda(\mathit{leaf}\ n) . \mathit{leaf}\ n)\ x) \\
& [\!] \ ((\lambda(\mathit{branch}\ t_1\ t_2) . \mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\
& [\!] \ \mathit{ERROR} \\
&)
\end{aligned}
$$

Az absztrakciókból az *UPS_leaf* és *UPS_branch* alkalmazásával kiküszöbölve az összegkonstruktorokat, a

$$
\begin{aligned}
\mathit{reflect} = \lambda x . \ (\ & (\mathit{UPS\_leaf}\ (\lambda n . \mathit{leaf}\ n)\ x) \\
& [\!] \ (\mathit{UPS\_branch}\ (\lambda t_1 . \lambda t_2 . \mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\
& [\!] \ \mathit{ERROR} \\
&)
\end{aligned}
$$

kifejezést kapjuk.

Határozzuk meg a *reflect* (*branch* $E\ F$) kifejezést, most is látható, hogy a $[\!]$ műveleten kívül csak β- és δ-redukciókat kell alkalmaznunk:

$$
\begin{aligned}
&\mathit{reflect}\ (\mathit{branch}\ E\ F) \equiv \\
&(\lambda x . \ (\ \ (\mathit{UPS\_leaf}\ (\lambda n . \mathit{leaf}\ n)\ x) \\
&\quad [\!] \ (\mathit{UPS\_branch}\ (\lambda t_1 . \lambda t_2 . \mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\
&\quad [\!] \ \mathit{ERROR} \\
&\quad)\ )\ \ (\mathit{branch}\ E\ F) \\
&\to
\end{aligned}
$$

$$
\begin{array}{l}
(\ \ (\mathit{UPS\_leaf}\ (\lambda n\,.\,\mathit{leaf}\ n)\ (\mathit{branch}\ E\ F)) \\
[\!]\ (\mathit{UPS\_branch}\ (\lambda t_1\,.\,\lambda t_2\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathit{branch}\ E\ F)) \\
[\!]\ \mathit{ERROR} \\
) \\
\rightarrow \\
(\ \ \mathit{fail} \\
[\!]\ (\mathit{UPS\_branch}\ (\lambda t_1\,.\,\lambda t_2\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathit{branch}\ E\ F)) \\
[\!]\ \mathit{ERROR} \\
) \\
\rightarrow \\
(\ \ (\mathit{UPS\_branch}\ (\lambda t_1\,.\,\lambda t_2\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathit{branch}\ E\ F)) \\
[\!]\ \mathit{ERROR} \\
) \\
\rightarrow \\
(\lambda t_1\,.\,\lambda t_2\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ E\ F \quad \rightarrow^{+} \\
\mathit{branch}\ (\mathit{reflect}\ F)\ (\mathit{reflect}\ E)
\end{array}
$$

□

5.1.3. Szorzatkonstruktoros absztrakciók

A 3.7.3. pontban a szorzatkonstruktor jelentését is egy applikációval adtuk meg:

$$
\begin{array}{l}
(\lambda(t\ p_1 p_2 \ldots p_n)\,.\,E)\ F\ \rightarrow \\
(\lambda p_1\,.\,\lambda p_2\,.\,\cdots\,.\,\lambda p_n\,.\,E)\ (\mathit{SEL\_t\_}1\ F)\ (\mathit{SEL\_t\_}2\ F)\ldots(\mathit{SEL\_t\_}n\ F)
\end{array}
$$

ahol

$$
\begin{array}{lll}
\mathit{SEL\_t\_}i\ (t\ F_1 F_2 \ldots F_n) & \rightarrow F_i & (1 \le i \le n) \\
\mathit{SEL\_t\_}i\ (t'\ F_1 F_2 \ldots F_n) & \rightarrow \mathit{fail}, & \mathit{ha}\ t \neq t' \\
\mathit{SEL\_t\_}i\ \bot & \rightarrow \bot &
\end{array}
$$

Azért, hogy az absztrakcióban itt se szerepeljen a t konstruktor, a konstruktort építsük be egy *UPP_t* függvénybe (a függvény neve az „UnPack-Product” kifejezésből származik), és ezzel a függvénnyel fejezzük ki az absztrakciót. A t-től függő *UPP_t* függvény is, a *SEL_t_i*-hez hasonlóan, a konstansos λ-kalkulus δ-függvényének tekinthető.

$$\lambda(t\ p_1 p_2 \ldots p_n)\,.\,E \quad \rightsquigarrow \quad \mathit{UPP\_t}\ (\lambda p_1\,.\,\lambda p_2\,.\,\cdots\,.\,\lambda p_n\,.\,E)$$

Látható, hogy a konstruktoros absztrakció – az összegkonstruktoros átalakításhoz hasonlóan – most is egyszerűbb lett, hiszen a jobboldalon már itt sincs az absztrakcióban t konstruktor.

Ha $f = \lambda p_1 . \lambda p_2 . \cdots . \lambda p_n . E$, akkor azonnal látható, hogy a szorzatkonstruktoros absztrakció fent leírt jelentése az *UPP_t* függvény alkalmazásával a következőképpen adható meg:

$$UPP\_t\ f\ F \quad \rightarrow \quad f\ (SEL\_t\_1\ F)\ (SEL\_t\_2\ F) \ldots (SEL\_t\_n\ F)$$

A fentiekből az is látható, hogy az absztrakcióban levő t és az F kifejezésben levő konstruktorok összehasonlítása most is átkerült a $SEL\_t\_i\ F$ applikációkba.

5.1.3. Példa. *(Mintaillesztés szorzatkonstruktoros absztrakcióval)*
A 3.7.8. példában láttuk, hogy az

$addpair\ (pair\ x\ y) = +\ x\ y$

függvénydefiníció λ-kifejezése

$$add\_pair = \lambda z . \ (\ ((\lambda(pair\ x\ y) . \ +\ x\ y)\ z)$$
$$[\!]\ ERROR$$
$$)$$

A szorzatkonstruktort az absztrakcióból kiküszöbölve az

$$add\_pair = \lambda z . \ (\ (UPP\_pair\ (\lambda x . \lambda y . \ +\ x\ y)\ z)$$
$$[\!]\ ERROR$$
$$)$$

kifejezést kapjuk.

Határozzuk meg az $add\_pair\ (pair\ 5\ 6)$ kifejezés értékét.

$$add\_pair\ (pair\ 5\ 6) \equiv$$
$$(\lambda z . \ (\ (UPP\_pair\ (\lambda x . \lambda y . \ +\ x\ y)\ z)$$
$$[\!]\ ERROR$$
$$)\ (pair\ 5\ 6)$$
$$\rightarrow$$
$$(\ \ (UPP\_pair\ (\lambda x . \lambda y . \ +\ x\ y)\ (pair\ 5\ 6))$$
$$[\!]\ ERROR$$
$$)$$
$$\rightarrow$$
$$(\lambda x . \lambda y . \ +\ x\ y)\ (SEL\_pair\_1\ (pair\ 5\ 6))\ (SEL\_pair\_2\ (pair\ 5\ 6)) \ \rightarrow^{+}$$

$+\ (\mathit{SEL\_pair\_}1\ (\mathit{pair}\ 5\ 6))\ (\mathit{SEL\_pair\_}2\ (\mathit{pair}\ 5\ 6)) \rightarrow^{+}$
$+\ 5\ 6 \rightarrow$
11 □

5.2. A függvények számának csökkentése

Az előző szakaszban láttuk, hogy az összeg- és szorzatkonstruktoros applikációk átírásához minden egyes konstruktorra egy *UPS* vagy *UPP* függvényt kellett bevezetni. Ha feltesszük azt, hogy a funkcionális program típusellenőrzése már megtörtént és a program típusosan helyes, akkor ezeknek a függvényeknek a számát csökkenteni tudjuk.

5.2.1. Az összegtípusú mintaillesztés függvényei

Először foglalkozzunk az összegtípusú konstruktorokkal. Ha egy típus konstruktorait megsorszámozzuk, akkor típusosan helyes programok esetén elegendő azt nyilvántartani, hogy az adott típus hányadik konstruktoráról van szó, és teljesen közömbös az, hogy ennek a konstruktornak mi a neve. Ugyanis a típushelyesség miatt az összegkonstruktoros mintaillesztés esetén, azaz a mintaabsztrakció és argumentuma vizsgálatakor, ha nem is az első mintaillesztésre, de legalább az egyikre biztosan *nem-fail* eredményt kapunk. Egy mintaillesztés eredménye akkor lesz *fail*, ha az absztrakcióban és az argumentumában a konstruktorok sorszáma nem egyezik meg, és nyilván nem lesz *fail*, ha a konstruktorok sorszáma azonos.

Az összegkonstruktorok azonosítására vezessük be az *FPS* kulcsszót a *Function* és „Pack Sum” szavak kezdőbetűiből. Ha s a típus i-edik konstruktora és a konstruktor aritása d_i, akkor jelöljük a konstruktort *FPS*_i_d_i-vel. Az *UPS*_s függvényt pedig, ahol s az i-edik konstruktor a d_i aritással, jelöljük *FUPS*_i_d_i-vel.

Ezeknek a felhasználásával az összegkonstruktoros absztrakciókra az 5.1.2. pontban megadott

$$\begin{aligned}
&\mathit{UPS_s}\ f\ (s\ F_1F_2\ldots F_n) &&\rightarrow f\ F_1F_2\ldots F_n \\
&\mathit{UPS_s}\ f\ (s'\ F_1F_2\ldots F_m) &&\rightarrow \mathit{fail}, \qquad \mathit{ha}\ s \neq s' \\
&\mathit{UPS_s}\ f\ (s'\ F_1F_2\ldots F_m) &&\rightarrow \bot, \qquad \mathit{ha}\ s'\ F_1F_2\ldots F_m = \bot
\end{aligned}$$

redukciós szabályok – figyelembe véve, hogy a kifejezések típusosan helyesek, – így írhatók át:

$$\begin{aligned}
&\mathit{FUPS_i_d_i}\ f\ (\mathit{FPS_i_d_i}\ F_1F_2\ldots F_{d_i}) &&\rightarrow f\ F_1F_2\ldots F_n \\
&\mathit{FUPS_i_d_i}\ f\ (\mathit{FPS_j_d_j}\ F_1F_2\ldots F_{d_j}) &&\rightarrow \mathit{fail}, \quad \mathit{ha}\ i \neq j
\end{aligned}$$

Ez azt is jelenti, hogy a mintaillesztésre elegendő egy *FUPS* kétváltozós függvényt megvalósító programot írni, az i és d_i számok ennek a függvénynek a paraméterei lehetnek. A mintaillesztés programja is nagyon egyszerű lett, hiszen elegendő csak az *FUPS* és az *FPS* argumentumainak egyezését vizsgálni. Megjegyezzük, hogy a második argumentumra, az aritásra a mintaillesztés eredményének pontos meghatározásához van szükség. Az is látható, hogy az f kifejezésben szereplő konstruktorok nem szerepelnek a redukcióban, ezért ezeket a konstruktorokat nem is kell a kódjaikra átírni.

5.2.1. Példa. *(Összegkonstruktoros függvények)*
A *List* típus két konstruktora a *nil* és a *cons*. E két konstruktor azonosítója *FPS*_1_0 és *FPS*_2_2. Az *UPS_nil* és *UPS_cons* függvényeket az *FUPS*_1_0 és *FUPS*_2_2 függvények reprezentálják.

A *leaf* és *branch* konstruktorú *Tree* típushoz a *FPS*_1_1 és *FPS*_2_2 azonosítók tartoznak, és az *UPS_leaf* és *UPS_branch* függvényeket az *FUPS*_1_1 és *FUPS*_2_2 függvények jelölik.

Látható, hogy az *UPS_cons* és *UPS_branch* függvényekhez ugyanaz a név tartozik, de ez nyilván nem lesz probléma, mert a konstruktorok különböző típushoz tartoznak. □

5.2.2. Példa. *(Mintaillesztés összegkonstruktoros absztrakciókkal)*
A 5.1.2. példában láttuk, hogy a

$$\begin{aligned} \mathit{reflect}\ (\mathsf{leaf}\ n) &= \mathsf{leaf}\ n \\ \mathit{reflect}\ (\mathsf{branch}\ t_1\ t_2) &= \mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1) \end{aligned}$$

függvény alakja a

$$\begin{aligned} \mathit{reflect} = \lambda x .\ (\ & (\mathit{UPS\_leaf}\ (\lambda n .\ \mathsf{leaf}\ n)\ x) \\ [\!] \ & (\mathit{UPS\_branch}\ (\lambda t_1 .\ \lambda t_2 .\ \mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\ [\!] \ & \mathit{ERROR} \\ &) \end{aligned}$$

kifejezést volt. Ezt átírva a most bevezetett függvényekkel, ismét csak a kifejezések típusos helyessége miatt, a

$$\begin{aligned} \mathit{reflect} = \lambda x .\ (\ & (\mathit{FUPS\_1\_1}\ (\lambda n .\ \mathsf{leaf}\ n)\ x) \\ [\!] \ & (\mathit{FUPS\_2\_2}\ (\lambda t_1 .\ \lambda t_2 .\ \mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\ &) \end{aligned}$$

kifejezést kapjuk.

Határozzuk meg a *reflect* (*branch E F*) kifejezést. Megállapíthatjuk, hogy a kifejezés típusosan helyes, tehát alkalmazhatjuk a fenti módszert.

$$
\begin{aligned}
&\mathit{reflect}\ (\mathsf{branch}\ E\ F) \equiv \\
&\mathit{reflect}\ (\mathsf{FPS\_2\_2}\ E\ F) \equiv \\
&(\lambda x\,.\ (\quad (\mathsf{FUPS\_1\_1}\ (\lambda n\,.\,\mathsf{leaf}\ n)\ x) \\
&\qquad [\!]\ (\mathsf{FUPS\_2\_2}\ (\lambda t_1\,.\,\lambda t_2\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x) \\
&\qquad)\ )\quad (\mathsf{FPS\_2\_2}\ E\ F) \\
&\to \\
&(\ \ (\mathsf{FUPS\_1\_1}\ (\lambda n\,.\,\mathsf{leaf}\ n)\ (\mathsf{FPS\_2\_2}\ E\ F)) \\
&[\!]\ (\mathsf{FUPS\_2\_2}\ (\lambda t_1\,.\,\lambda t_2\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))(\mathsf{FPS\_2\_2}\ E\ F)) \\
&) \\
&\to \\
&(\ \ \mathsf{fail} \\
&[\!]\ (\mathsf{FUPS\_2\_2}\ (\lambda t_1\,.\,\lambda t_2\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathsf{FPS\_2\_2}\ E\ F)) \\
&) \\
&\to \\
&(\ (\mathsf{FUPS\_2\_2}\ (\lambda t_1\,.\,\lambda t_2\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ (\mathsf{FPS\_2\_2}\ E\ F)) \\
&) \\
&\to \\
&(\lambda t_1\,.\,\lambda t_2\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ E\ F\ \ \to^{+} \\
&\mathsf{branch}\ (\mathit{reflect}\ F)\ (\mathit{reflect}\ E) \qquad \square
\end{aligned}
$$

5.2.2. A szorzattípusú mintaillesztés függvényei

Természetesen most is fel kell tennünk, hogy a típusellenőrzés már megtörtént, és a programunk típusosan helyes.

A szorzatkonstruktorok azonosítására vezessük be az *FPP* kulcsszót a *function* és „Pack Product” szavak kezdőbetűiből. Ha t a szorzattípus konstruktora és a konstruktor aritása d, akkor jelöljük a konstruktort *FPP_d*-vel. Az *UPP_t* függvényt pedig, ahol a t konstruktor aritása d, jelöljük *FUPP_d*-vel.

A típushelyesség miatt a *SEL_t_i* szelektorok esetén sincs szükség a t konstruktor tárolására, helyettük vezessük be az *FSEL_d_i* függvényt, ahol d most is a t konstruktor aritása.

Ezeknek a felhasználásával a d aritású t szorzatkonstruktoros absztrakciókra az 5.1.3. pontban megadott

$$\mathsf{UPP\_t}\ f\ F \quad \to \quad f\ (\mathsf{SEL\_t\_1}\ F)\ (\mathsf{SEL\_t\_2}\ F)\ldots(\mathsf{SEL\_t\_}d\ F)$$

redukciós szabályok így írhatók át:

$$\mathit{FUPP\_d}\ f\ F \quad \to \quad f\ (\mathit{FSEL\_d\_1}\ F)\ (\mathit{FSEL\_d\_2}\ F)\ldots(\mathit{FSEL\_d\_d}\ F)$$

Megjegyezzük, hogy a *FSEL_d_i F* applikációban szerepel az *FPP_d*, hiszen a típusellenőrzés helyes eredménye szerint F csak *FPP_d* $F_1 \ldots F_d$ alakú lehet. A *FSEL_d_i* függvényt az új jelölésekkel a következőképpen adhatjuk meg:

$$\begin{array}{ll} \mathit{FSEL\_d\_i}\ (\mathit{FPP\_d}\ F_1F_2\ldots F_n) \to F_i & (1 \le i \le n) \\ \mathit{FSEL\_d\_i}\ (\mathit{FPP\_d'}\ F_1F_2\ldots F_n) \to \mathit{fail}, & \mathit{ha}\ d \neq d' \end{array}$$

Látható, hogy az *FSEL* bevezetésével a szelektor program is nagyon leegyszerűsödött.

5.2.3. Példa. *(Mintaillesztés szorzatkonstruktoros absztrakcióval)*
Az 5.1.3. példában láttuk, hogy az

$\mathit{addpair}\ (\mathit{pair}\ x\ y) = +x\ y$

függvénydefinícióból az

$$\begin{array}{l} \mathit{add\_pair} = \lambda z\,.\ (\ (\mathit{UPP\_pair}\ (\lambda x\,.\,\lambda y\,.\ +\ x\ y)\ z) \\ \qquad\qquad [\!]\ \mathit{ERROR} \\ \qquad\qquad) \end{array}$$

kifejezést kaptuk. Ezt átírva a most bevezetett függvényekkel,

$\mathit{add\_pair} = \lambda z\,.\,\mathit{FUPP\_2}\ (\lambda x\,.\,\lambda y\,.\ +\ x\ y)\ z$

Határozzuk meg az *add_pair* (*pair* 5 6) kifejezés értékét.

$\mathit{add\_pair}\ (\mathit{pair}\ 5\ 6) \equiv$
$\mathit{add\_pair}\ (\mathit{FPP\_2}\ \ 5\ 6) \equiv$
$(\lambda z\,.\,\mathit{FUPP\_2}\ (\lambda x\,.\,\lambda y\,.\ +\ x\ y)\ z)\ (\mathit{FPP\_2}\ \ 5\ 6) \quad \to$
$\mathit{FUPP\_2}\ (\lambda x\,.\,\lambda y\,.\ +\ x\ y)\ (\mathit{FPP\_2}\ \ 5\ 6) \quad \to$
$(\lambda x\,.\,\lambda y\,.\ +\ x\ y)\ (\mathit{FSEL\_2\_1}\ (\mathit{FPP\_2}\ \ 5\ 6))\ (\mathit{FSEL\_2\_2}\ (\mathit{FPP\_2}\ \ 5\ 6)) \ \to^+$
$(\lambda x\,.\,\lambda y\,.\ +\ x\ y)\ 5\ 6 \ \to^+$
11 □

5.3. A letrec- és let-kifejezések átalakítása

A 3.3. szakaszban láttuk az egydefiníciós egyszerű letrec-kifejezés átalakítását:

$\mathit{letrec}\ x = E\ \mathit{in}\ F \Longrightarrow \mathit{let}\ x = \mathsf{Y}\ (\lambda x\,.\,E)\ \mathit{in}\ F,$

és a 3.4. szakaszban már foglalkoztunk az egy- és többdefiníciós egyszerű let-kifejezésekkel. Megadtuk, hogy hogyan kell az egyszerű let-kifejezést λ-kifejezéssé alakítani:

$\mathit{let}\ x = E\ \mathit{in}\ F \rightsquigarrow (\lambda x\,.\,F)E$

Mivel egy β-redukciót alkalmazva $(\lambda x\,.\,F)E = F[x := E]$, a kifejezés értéke nem változik meg, ha a $(\lambda x\,.\,F)E$ helyett $F[x := E]$-t írunk.

Azonban ezekkel az átalakításokkal kapcsolatban több probléma is felmerül.

1. A letrec-kifejezés átalakításával egy olyan kifejezést kapunk, amiben egy fixpont-kombinátor van. Az ilyen kifejezés redukciós sorozatának előállítása legtöbbször nem egyszerű feladat.
2. Ennél sokkal komolyabb probléma van a második átalakítással. A let-kifejezésből egy olyan applikációt kapunk, amiben az E és F kifejezések rögzítettek, azaz típusaikat egyértelműen meg lehet határozni. Ez azt jelenti, hogy az átalakítással kapott kifejezésre már nem alkalmazható semmiféle polimorfizmus, míg az eredeti let-kifejezésre a Hindley–Milner vagy a Milner–Mycroft típuskikövetkeztető módszerek a let-kifejezés törzsében lehetővé teszik az x polimorfikus típusozottságát (lásd [3]).

Ezek miatt a problémák miatt a továbbiakban csak arra fogunk törekedni, hogy a bonyolultabb let- és letrec-kifejezéseket egyszerű let-kifejezésekre alakítsuk át, azaz a célnyelvünk az egyszerű let-kifejezéssel bővített λ-kalkulus lesz.

A 3.4. szakaszban láttuk, hogy a többdefiníciós egyszerű let-kifejezés skatulyázott egydefiníciós egyszerű let-kifejezésekre alakítható át. Ezt megtehetjük nem csak az egyszerű kifejezésekre, hanem azokra is, amelyekben minta van.

$$
\begin{array}{l}
\mathit{let}\ x_1 = E_1 \\
\quad\ x_2 = E_2 \\
\quad\ \ldots \\
\quad\ x_n = E_n \\
\mathit{in}\ E \qquad\quad \Longrightarrow \quad \mathit{let}\ x_1 = E_1 \\
\qquad\qquad\qquad\quad\ \mathit{in}\ (\ \mathit{let}\ x_2 = E_2 \\
\qquad\qquad\qquad\qquad\ \mathit{in}\ (\ \ldots \\
\qquad\qquad\qquad\qquad\quad (\ \mathit{let}\ x_n = E_n \\
\qquad\qquad\qquad\qquad\qquad\ \mathit{in}\ E\ )\ \ldots))
\end{array}
$$

Ez azt jelenti, hogy a továbbiakban csak azokkal a let-kifejezésekkel kell

foglalkoznunk, amelyekben egy definíció van.

5.3.1. A biztonságos let-kifejezés

Most tehát olyan kifejezéseket vizsgálunk, ahol a let-kifejezés definíciójának bal oldalán minta is állhat, mint például a

let (*cons x xs*) = *E in F*

kifejezésben. Ez a lehetőség azt jelenti, hogy az egyenlőség bal oldalán álló minta és az *E* között mintaillesztést is kell végezni, és meg kell gondolni, hogy mit lehet csinálni akkor, ha a mintaillesztés *fail* eredményt ad. Az ilyen vizsgálatok az implementáció költségét, például a futási időt is jelentősen megnövelhetik.

Egy minta lehet változó, konstans, szorzatkonstruktoros vagy összegkonstruktoros nulla-, egy- vagy többparaméteres minta. Az összegkonstruktoros minta esetében a minta konstruktorai különbözőek lehetnek, és nem csak ekkor, hanem a konstans minta esetén is előfordulhat, hogy a mintaillesztés *fail* eredményt ad.

Ha a mintaillesztés egyáltalán nem adhat *fail* eredményt, akkor azt mondjuk, hogy a *minta biztonságos*.

- Ha a minta egy változó, akkor nem kell mintaillesztést végezni, azaz mondhatjuk, hogy a változó biztonságos.
- A szorzatkonstruktoros mintáról pedig tegyük fel, hogy a fordítóprogram statikus típusellenőrzése már elvégezte az ellenőrzését, és a kifejezés típusosan helyes, azaz a mintaillesztés biztosan nem ad *fail* eredményt. Mivel a konstruktor argumentumai újabb minták lehetnek, azért, hogy a mintaillesztés teljes egészében biztonságos legyen, az argumentumokról is fel kell tennünk, hogy biztonságosak. Egy biztonságos szorzatkonstruktoros minta paraméterei tehát vagy változók, vagy szorzatkonstruktoros minták.

A konstansos és összegtípusú konstruktorokat tartalmazó mintaapplikációk *nem biztonságosak*, mert az argumentumtól függően *fail* eredményt is adhatnak.

Azokat a let- és letrec-kifejezéseket, amelyekben biztonságos minta van, *biztonságos let-* és *biztonságos letrec-kifejezéseknek* nevezzük.

A biztonságos let-kifejezés alakja tehát

let p = E in F

ahol *p* egy biztonságos minta. Ha *p* egy *x* változó, akkor az átalakítása, mint láttuk,

$(\lambda x\,.\,F)E$.

Ezek analógiájára a tetszőleges $p \equiv t\ p_1\ p_2\ \ldots\ p_n$ szorzatkonstruktoros mintára az átalakítás legyen

$(\lambda(t\ p_1\ p_2 \ldots p_n)\,.\,F)E.$

Mint a 3.7.3. pontban láttuk,

$(\lambda(t\ p_1 p_2 \ldots p_n)\,.\,F)\ E\ \rightarrow$
$(\lambda p_1\,.\,\lambda p_2\,.\,\cdots\,.\,\lambda p_n\,.\,F)\ (SEL\_t\_1\ E)\ (SEL\_t\_2\ E)\ldots(SEL\_t\_n\ E)$

és végrehajtva a β-redukciókat, eredményül az

$F[p_1 := (SEL\_t\_1\ E)][p_2 := (SEL\_t\_2\ E)]\ldots[p_n := (SEL\_t\_n\ E)]$

kifejezést kapjuk. Ezt azonban úgy kaptuk, hogy a let-kifejezést függvényapplikációra írtuk át, amiről az előzőekben láttuk, hogy nem az optimális megoldás.

De ugyanezt az eredményt megkaphatjuk függvényapplikáció nélkül is, a következő, csak let-kifejezéseket tartalmazó átalakítással is:

$$
\begin{array}{l}
\mathit{let}\ (t\ p_1 p_2 \ldots p_n) = E \\
\ \mathit{in}\ F \qquad\qquad \Longrightarrow\ \mathit{let}\ x = E \\
\qquad\qquad\qquad\qquad\quad \mathit{in}\ (\mathit{let}\ p_1 = SEL\_t\_1\ x \\
\qquad\qquad\qquad\qquad\qquad\quad p_2 = SEL\_t\_2\ x \\
\qquad\qquad\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\qquad\qquad\quad p_n = SEL\_t\_n\ x \\
\qquad\qquad\qquad\qquad\qquad\ \mathit{in}\ F)
\end{array}
$$

ahol x egy új változó.

Látható, hogy a szorzatkonstruktoros, tehát biztonságos mintájú let-kifejezés egy olyan egyszerű, egydefiníciós let-kifejezéssé alakítható át, amelynek törzsében egy többdefiníciós, de biztonságos mintájú let-kifejezés van.

Már láttuk, hogy a többdefiníciós let-kifejezés skatulyázott egydefiníciós let-kifejezésekké alakítható át. Alkalmazva ezt az átírást, a skatulyázott let-kifejezések továbbra is biztosan biztonságosak lesznek. Ezekre a let-kifejezésekre külön-külön alkalmazható a most megadott átalakítás, és ennek az átalakításnak az ismételt alkalmazásaival és a skatulyázásokkal elérhető az, hogy a kifejezésekben minden let-kifejezés egydefiníciós és egyszerű let-kifejezés legyen.

5.3.1. Példa. *(A biztonságos let-kifejezés átalakítása)*
A biztonságos minta legyen a szorzatkonstruktoros *pair x y* minta, a kifejezés pedig legyen

$$
\begin{array}{l}
\mathit{let}\ (\mathit{pair}\ x\ y) = \mathit{pair}\ 5\ 6 \\
\ \mathit{in}\ +x\ y
\end{array}
$$

Alkalmazva a fenti átalakítást, a

let z = *pair* 5 6
 in (*let* x = *SEL_pair_*1 z
 y = *SEL_pair_*2 z
 in $+x\,y$)

kifejezést kapjuk. A megfelelő redukciókat végrehajtva:

let x = *SEL_pair_*1 (*pair* 5 6)
 y = *SEL_pair_*2 (*pair* 5 6)
 in $+\,x\,y$
$\rightarrow^+$
let $x = 5$
 $y = 6$
 in $+\,x\,y$
$\rightarrow^+$
11 □

5.3.2. A biztonságos letrec-kifejezés

A 3.3. szakaszban már láttuk, hogy az egydefiníciós egyszerű letrec-kifejezés hogyan alakítható át:

letrec $x = E$ *in* $F \quad \Longrightarrow \quad$ *let* $x = \mathsf{Y}\,(\lambda x\,.\,E)$ *in* F

A *letrec* helyett azért használható *let*, mert a definíciós részben megszünt a rekurzió, hiszen az E szabad x változói az absztrakcióval kötötté váltak.

Először nézzük azt az esetet, amikor a letrec-kifejezés definíciós részében több egyenlet van, és nem változók, hanem minták szerepelnek benne.

letrec $p_1 = E_1$
 $p_2 = E_2$
 $\ldots$
 $p_n = E_n$
 in F

Mivel most a biztonságos letrec-kifejezésekről van szó, a $p_1, p_2, \ldots, p_n$ minták is biztonságosak.

A $p_1, p_2, \ldots, p_n$ mintákból készítsünk egy n-est egy t_n konstruktorral, a

$t_n\ p_1\ p_2\ \ldots\ p_n$

kifejezésben t_n nyilvánvalóan egy n-paraméteres szorzatkonstruktor (a *pair* kétparaméteres szorzatkonstruktor általánosítása n paraméterre), és mivel a $p_1, p_2, \ldots, p_n$ minták biztonságosak, a $t_n\ p_1\ p_2\ \ldots\ p_n$ minta is biztonságos. Ezt felhasználva, alakítsuk át a többdefiníciós biztonságos letrec-kifejezést egy $t_n\ p_1\ p_2\ \ldots\ p_n$ mintájú, szintén biztonságos letrec-kifejezésre:

$$\begin{array}{lll} \textit{letrec}\ p_1 = E_1 & & \\ \qquad p_2 = E_2 & & \\ \qquad \ldots & & \\ \qquad p_n = E_n & & \\ \textit{in}\ F & \Longrightarrow & \textit{letrec}\ (t_n\ p_1\ p_2\ \ldots\ p_n) = (t_n\ E_1\ E_2\ \ldots\ E_n) \\ & & \qquad \textit{in}\ F \end{array}$$

Látható, hogy az eredményül kapott letrec-kifejezésben csak egy definíció van. Ezt pedig a fenti, egyszerű letrec-kifejezésre megadott módszerhez hasonlóan, az Y fixpont-kombinátor felhasználásával átalakíthatjuk egy let-kifejezésre.

$$\textit{letrec}\ p = E\ \textit{in}\ F \quad \Longrightarrow \quad \textit{let}\ p = \mathsf{Y}\ (\lambda p\,.\,E)\ \textit{in}\ F$$

Az absztrakció változójának helyére egy szorzatkonstruktoros minta került, de ezzel korábban, a 3.7.3. pontban már foglalkoztunk.

Tehát eredményül a következőket kapjuk:

$$\begin{array}{lll} \textit{letrec}\ p_1 = E_1 & & \\ \qquad p_2 = E_2 & & \\ \qquad \ldots & & \\ \qquad p_n = E_n & & \\ \textit{in}\ F & \Longrightarrow & \\ & & \textit{let}\ (t_n\ p_1\ p_2\ \ldots\ p_n) = \mathsf{Y}\ (\lambda(t_n\ p_1\ p_2\ \ldots\ p_n)\,.\,t_n\ E_1\ E_2\ \ldots\ E_n) \\ & & \quad \textit{in}\ F \end{array}$$

5.3.2. Példa. *(Biztonságos letrec-kifejezés)*

Alakítsuk át a következő kifejezést:

$$\begin{array}{l} \mathit{letrec}\ x = \mathit{cons}\ 1\ y \\ \qquad\ \ y = \mathit{cons}\ 2\ x \\ \quad \mathit{in}\ x \end{array}$$

A kifejezés definíciós részében a minták biztonságosak, mivel a definíciók baloldalán változók vannak. Így alkalmazható a fenti átalakítás, a t_2 kétparaméteres szorzatkonstruktor legyen a *pair*.

$$\begin{array}{l} \mathit{letrec}\ x = \mathit{cons}\ 1\ y \\ \qquad\ \ y = \mathit{cons}\ 2\ x \\ \quad \mathit{in}\ x \end{array}$$

$$\Longrightarrow$$

$$\begin{array}{l} \mathit{letrec}\ (\mathit{pair}\ x\ y) = \mathit{pair}\ (\mathit{cons}\ 1\ y)\ (\mathit{cons}\ 2\ x) \\ \quad \mathit{in}\ x \end{array}$$

majd a kifejezést az Y fixpont-kombinátor felhasználásával let-kifejezésre alakítva:

$$\begin{array}{l} \mathit{let}\ (\mathit{pair}\ x\ y) = \mathsf{Y}(\lambda(\mathit{pair}\ x\ y)\,.\,\mathit{pair}\ (\mathit{cons}\ 1\ y)\ (\mathit{cons}\ 2\ x)) \\ \ \mathit{in}\ x \end{array}$$

A szorzatkonstruktoros mintára vonatkozó átalakítás szerint a kifejezés a következő alakra hozható:

$$\begin{array}{l} \mathit{let}\ z = \mathsf{Y}(\lambda(\mathit{pair}\ x\ y)\,.\,\mathit{pair}\ (\mathit{cons}\ 1\ y)\ (\mathit{cons}\ 2\ x)) \\ \ \mathit{in}\ (\mathit{let}\ x = \mathit{SEL\_pair\_}1\ z \\ \qquad\ \ y = \mathit{SEL\_pair\_}2\ z \\ \quad \mathit{in}\ x) \end{array}$$

Ebből többszöri átalakítással

$$\begin{array}{l} (\lambda z\,.\,((\lambda x\,.\,\lambda y\,.\,x)(\mathit{SEL\_pair\_}1\ z)\ (\mathit{SEL\_pair\_}2\ z))) \\ \quad (\mathsf{Y}(\lambda(\mathit{pair}\ x\ y)\,.\,\mathit{pair}\ (\mathit{cons}\ 1\ y)\ (\mathit{cons}\ 2\ x))) \end{array}$$ □

5.3.3. Az általános let- és letrec-kifejezés

A nem-biztonságos let- és letrec-kifejezéseknél a minta összegkonstruktoros minta vagy akár konstans is lehet, ezért itt az a probléma, hogy mi történjen akkor, ha a mintaillesztés *fail* eredményt ad.

A mintaillesztés helyességét egy *illeszkedési vizsgálattal* kell eldönteni. A kérdés az, hogy ezt mikor kell elvégezni? A *mohó* algoritmus szerint a let- vagy letrec-kifejezés kiértékelésének elején, míg a *lusta* kiértékelés szerint akkor, amikor ténylegesen szükség van rá.

5.3.3. Példa. *(Mohó vagy lusta illeszkedési vizsgálat)*
A következő kifejezésnek túl sok értelme nincs, de a vizsgálatok különbözőségét jól mutatja:

$$\mathit{let}\ (\mathit{cons}\ y\ ys) = \mathit{nil}\ \mathit{in}\ 5$$

A mohó kiértékelés erre a *fail* jelzést adja, azaz a kifejezés feldolgozásakor egy *ERROR* hibajelzést kapunk, míg a lusta illeszkedési vizsgálat eredményül az 5 konstanst adja. □

A let- és letrec-kifejezés definíciós részének bal és jobb oldala közötti lusta illeszkedési vizsgálatot a függvénydefiníciók átalakításánál már megismert módszer felhasználásával adhatjuk meg. A bal oldalból készítsünk egy absztrakciót, a jobb oldal pedig legyen ennek az absztrakciónak a paramétere. A 3.7.2. pontban az összegkonstruktoros absztrakció szemantikájának megadásakor láttuk, hogy ez az applikáció elvégzi az illeszkedési vizsgálatot. Tehát a $p = E$ definícióból készítsünk egy

$$(\lambda p. \ldots)\ E \ [\!]\ \mathit{ERROR}$$

kifejezést, de mi legyen a pontok helyén ahhoz, hogy a p mintában levő paramétereket használni tudjuk? Mivel a biztonságos mintájú kifejezések kezelése és átalakítása egyszerű, készítsünk a $p \equiv s\ x_1\ \ldots\ x_n$ mintához egy n-est az $x_1\ \ldots\ x_n$ változókkal, és ez legyen az absztrakció törzse, és ez szerepeljen az átalakítással kapott let-kifejezés definíciójának bal oldalán is.

5.3.4. Példa. *(Az illeszkedési vizsgálat megoldása)*
Folytassuk tovább az előző példában szereplő kifejezés átalakítását. A

$$\mathit{let}\ (\mathit{cons}\ y\ ys) = \mathit{nil}\ \mathit{in}\ 5$$

kifejezés

$$\mathit{cons}\ y\ ys = \mathit{nil}$$

definíciós részéből készítsük el a

$$\mathit{pair}\ y\ ys = ((\lambda(\mathit{cons}\ y\ ys)\ .\ \mathit{pair}\ y\ ys)\ \mathit{nil}) \ [\!]\ \mathit{ERROR}$$

kifejezést, és ez lesz az új let-kifejezés definíciója. □

A példákban a minta paraméterei változók voltak, természetesen az átalakítást arra az esetre is meg kell adni, amikor a mintában levő konstruktornak minta paraméterei vannak. Ezt az esetet azonban vissza tudjuk vezetni a változókat tartalmazó esetre úgy, hogy meghatározzuk a minta összes változóját, és ha m változó

van, akkor az *m*-eseket ezekkel alkotjuk meg.

Egy p minta változóinak halmazát jelölje $Var(p)$, ezt a következő algoritmussal tudjuk meghatározni:

- ha $p = x$, akkor $Var(p) = \{x\}$,
- ha $p = k$, akkor $Var(p) = \emptyset$,
- ha $p = c\ p_1\ \ldots\ p_r$, akkor $Var(p) = Var(p_1) \cup \cdots \cup Var(p_r)$

Ezt a halmazt felhasználva megadhatjuk a $p = E$ definíciós rész illeszkedési vizsgálatát.

$$p = E \quad \Longrightarrow \quad t\ x_1\ \ldots\ x_n = ((\lambda p\,.\,t\ x_1\ \ldots\ x_n)\ E) \ [\!]\ \mathit{ERROR}$$

ahol $Var(p) = \{x_1\ \ldots\ x_n\}$,
és t az n-es konstruktor.

A függvényapplikációt írjuk át let-kifejezésre, hogy a polimorfikus típuskikövetkeztetést is lehetővé tegyük:

$$p = E \quad \Longrightarrow \quad t\ x_1\ \ldots\ x_n = \mathsf{let}\ x = E\ \mathsf{in}\ ((\lambda p\,.\,t\ x_1\ \ldots\ x_n)\ x) \ [\!]\ \mathit{ERROR}$$

ahol $Var(p) = \{x_1\ \ldots\ x_n\}$,
t az n-es konstruktor,
és x egy új változó.

Természetesen, ha $|Var(p)| = 1$, akkor nincs szükség a t konstruktorra, a λ-absztrakcióban csak ez az egy változó szerepel.

5.3.5. Példa. *(Let-kifejezés összegkonstruktoros mintákkal)*
Határozzuk meg a

$$\begin{array}{l}\mathsf{let}\ \mathit{cons}\ x\ xs = \mathit{cons}\ 1\ xs\\ \quad \mathit{cons}\ 2\ ys = \mathit{cons}\ 2\ (\mathit{cons}\ 3\ \mathit{nil})\\ \mathsf{in}\ \mathit{cons}\ x\ ys\end{array}$$

kifejezés értékét. Először alakítsuk át a definíciós rész két sorát biztonságos

mintákra. Az első sorban két változó van, ezért

$$\begin{aligned} &\mathit{pair}\ x\ xs\ =\ \mathit{let}\ u = \mathit{cons}\ 1\ xs \\ &\qquad\qquad \mathit{in}\ ((\lambda(\mathit{cons}\ x\ xs)\,.\,\mathit{pair}\ x\ xs)\ u) \\ &\qquad\qquad\quad [\!]\,\mathit{ERROR} \end{aligned}$$

és a második sor, ahol csak egy változó van:

$$\begin{aligned} &ys\ =\ \mathit{let}\ v = \mathit{cons}\ 2\ (\mathit{cons}\ 3\ \mathit{nil}) \\ &\qquad \mathit{in}\ ((\lambda(\mathit{cons}\ 2\ ys)\,.\,ys)\ v) \\ &\qquad\quad [\!]\,\mathit{ERROR} \end{aligned}$$

tehát az eredeti kifejezés

$$\begin{aligned} &\mathit{let}\ \mathit{pair}\ x\ xs = \mathit{let}\ u = \mathit{cons}\ 1\ xs \\ &\qquad\qquad\qquad\quad \mathit{in}\ ((\lambda(\mathit{cons}\ x\ xs)\,.\,\mathit{pair}\ x\ xs)\ u) \\ &\qquad\qquad\qquad\qquad [\!]\,\mathit{ERROR} \\ &\quad ys\qquad\quad = \mathit{let}\ v = \mathit{cons}\ 2\ (\mathit{cons}\ 3\ \mathit{nil}) \\ &\qquad\qquad\qquad\quad \mathit{in}\ ((\lambda(\mathit{cons}\ 2\ ys)\,.\,ys)\ v) \\ &\qquad\qquad\qquad\qquad [\!]\,\mathit{ERROR} \\ &\mathit{in}\ \mathit{cons}\ x\ ys \end{aligned}$$

alakú lett. A let-kifejezés skatulyázható,

$$\begin{aligned} &\mathit{let}\ \mathit{pair}\ x\ xs = \mathit{let}\ u = \mathit{cons}\ 1\ xs \\ &\qquad\qquad\qquad\quad \mathit{in}\ ((\lambda(\mathit{cons}\ x\ xs)\,.\,\mathit{pair}\ x\ xs)\ u) \\ &\qquad\qquad\qquad\qquad [\!]\,\mathit{ERROR} \\ &\mathit{in}\ (\mathit{let}\ ys = \mathit{let}\ v = \mathit{cons}\ 2\ (\mathit{cons}\ 3\ \mathit{nil}) \\ &\qquad\qquad\quad \mathit{in}\ \ ((\lambda(\mathit{cons}\ 2\ ys)\,.\,ys)\ v) \\ &\qquad\qquad\qquad [\!]\,\mathit{ERROR} \\ &\quad \mathit{in}\,\mathit{cons}\ x\ ys \qquad\qquad\qquad\qquad) \end{aligned}$$

Először alakítsuk át a belső kifejezést.

$$\begin{aligned} &\mathit{let}\ ys = \mathit{let}\ v = \mathit{cons}\ 2\ (\mathit{cons}\ 3\ \mathit{nil}) \\ &\qquad\quad \mathit{in}\ ((\lambda(\mathit{cons}\ 2\ ys)\,.\,ys)\ v) \\ &\qquad\qquad [\!]\,\mathit{ERROR} \\ &\mathit{in}\ \mathit{cons}\ x\ ys \end{aligned}$$

A definíció jobb oldala is egy let-kifejezés, erre

$$
\begin{array}{l}
\mathit{let}\ v = \mathit{cons}\ 2\ (\mathit{cons}\ 3\ \mathit{nil}) \\
\ \mathit{in}\ ((\lambda(\mathit{cons}\ 2\ ys)\,.\,ys)\ v) \\
\quad [\!]\,\mathit{ERROR} \\
\rightarrow \\
(\lambda(\underline{\mathit{cons}}\ 2\ ys)\,.\,ys)\ (\underline{\mathit{cons}}\ 2\ (\mathit{cons}\ 3\ \mathit{nil})) \\
[\!]\,\mathit{ERROR} \\
\rightarrow \\
(\lambda\underline{2}\,.\,\lambda ys\,.\,ys)\ \underline{2}\ (\mathit{cons}\ 3\ \mathit{nil}) \\
[\!]\,\mathit{ERROR} \\
\rightarrow \\
(\lambda ys\,.\,ys)\ (\mathit{cons}\ 3\ \mathit{nil}) \\
[\!]\,\mathit{ERROR} \\
\rightarrow \\
\mathit{cons}\ 3\ \mathit{nil}
\end{array}
$$

Tehát a belső kifejezés:

$$
\begin{array}{l}
\mathit{let}\ ys = \mathit{cons}\ 3\ \mathit{nil} \\
\ \mathit{in}\ \mathit{cons}\ x\ ys \\
\rightarrow \\
\mathit{cons}\ x\ (\mathit{cons}\ 3\ \mathit{nil})
\end{array}
$$

Most térjünk vissza a skatulyázott let-kifejezés külső kifejezésére,

$$
\begin{array}{l}
\mathit{let}\ \mathit{pair}\ x\ xs = \mathit{let}\ u = \mathit{cons}\ 1\ xs \\
\qquad\qquad\qquad\qquad \mathit{in}\ ((\lambda(\mathit{cons}\ x\ xs)\,.\,\mathit{pair}\ x\ xs)\ u) \\
\qquad\qquad\qquad\qquad\quad [\!]\,\mathit{ERROR} \\
\ \mathit{in}\ \mathit{cons}\ x\ (\mathit{cons}\ 3\ \mathit{nil})
\end{array}
$$

A definíció jobb oldalát átalakítva:

$$
\begin{array}{l}
\mathit{let}\ u = \mathit{cons}\ 1\ xs \\
\ \mathit{in}\ ((\lambda(\mathit{cons}\ x\ xs)\,.\,\mathit{pair}\ x\ xs)\ u) \\
\quad [\!]\,\mathit{ERROR} \\
\rightarrow \\
(\lambda(\underline{\mathit{cons}}\ x\ xs)\,.\,\mathit{pair}\ x\ xs)\ (\underline{\mathit{cons}}\ 1\ xs) \\
(\lambda x\ xs\,.\,\mathit{pair}\ x\ xs)\ 1\ xs \\
\rightarrow^{+}
\end{array}
$$

pair 1 *xs*

Tehát a külső let-kifejezés

let pair x xs = pair 1 *xs*
in cons x (*cons* 3 *nil*)
→
let w = pair 1 *xs*
in (*let x = SEL_pair_*1 *w*
*xs = SEL_pair_*2 *w*
in cons x (*cons* 3 *nil*))
$\rightarrow^+$
let w = pair 1 *xs*
in cons (*SEL_pair_*1 *w*) (*cons* 3 *nil*)
→
cons (*SEL_pair_*1 (*pair* 1 *xs*)) (*cons* 3 *nil*) →
cons 1 (*cons* 3 *nil*) □

5.4. A case-kifejezés

A 4. fejezetben a mintákat tartalmazó definíciókat case-kifejezésekre alakitottuk át, és az 5.1. szakaszban láttuk, hogy a mintaabsztrakciók hogyan alakíthatók át a konstansos λ-kalkulus kifejezéseivé. Most megadjuk, hogy a mintadefiníciók szorzat- és összegtípusú kostruktorai esetén a belőlük levezetett case-kifejezések hogyan alakíthatók át közvetlenül a λ-kalkulus kifejezéseire. Az egyszerűség kedvéért feltesszük, hogy a minták argumentumai változók. Ha az argumentumok konstruktorokat tartalmazó minták, akkor az alábbi átalakításokat ezekre is alkalmazni kell.

5.4.1. A case-kifejezés szorzattípusú konstruktorral

A 3.5.2. pontban láttuk, hogy az

$f\ (t\ x_1\ x_2 \ldots x_n) = E$

definícióból, ahol t szorzattípusú konstruktor, x_i változó és így $\mathrm{TE}(\!|\, x_i \,|\!) = x_i$ $(1 \leq i \leq n)$, továbbá feltéve, hogy $\mathrm{TE}(\!|\, E \,|\!) = E$, a következő kifejezést kapjuk:

$f = \lambda z\,.$ ($((\lambda(t\ x_1\ x_2 \ldots x_n)\,.\,E)\ z)$
$(\!|$ *ERROR*
)

A 4. fejezetben leírtak szerint, meghatározva a minta *case* műveletet tartalmazó alakját, ebből az

$$f = \lambda z\,.\,\mathit{case}\ z\ \mathit{of}$$
$$t\ x_1\ x_2\ \ldots\ x_n{:}E,$$

kifejezéshez jutunk. Itt viszont látható, hogy ebben a kifejezésben tulajdonképpen a *case* használatának nincs túl sok szerepe, mivel a z kiértékelésére és az „egyirányú” elágazásra nincs is szükség.

Az 5.1.3. pontban leírtak szerint

$$\lambda(t\ x_1 x_2 \ldots x_n)\,.\,z \rightsquigarrow \mathit{UPP\_t}\ (\lambda x_1\,.\,\lambda x_2\,.\,\cdots\,.\,\lambda x_n\,.\,z),$$

így ezekből azonnal kapjuk a következő átalakítást:

$$\lambda z\,.\,\mathit{case}\ z\ \mathit{of}$$
$$t\ x_1\ x_2\ \ldots\ x_n\ :\ E \rightsquigarrow \lambda z\,.\,\mathit{UPP\_t}\ (\lambda x_1\,.\,\lambda x_2\,.\,\cdots\,.\,\lambda x_n\,.\,E)\ z$$

A jobboldalra egy η-konverziót alkalmazva:

$$\lambda z\,.\,\mathit{case}\ z\ \mathit{of}$$
$$t\ x_1\ x_2\ \ldots\ x_n\ :\ E \quad \rightsquigarrow \quad \mathit{UPP\_t}\ (\lambda x_1\,.\,\lambda x_2\,.\,\cdots\,.\,\lambda x_n\,.\,E)$$

5.4.1. Példa. *(Az add_pair kifejezés)*
A 4.1.7. példa szerint az

$$\mathit{add\_pair}(\mathit{pair}\ x\ y) = +\ x\ y$$

definíció *case*-t tartalmazó alakja:

$$\mathit{add\_pair} =$$
$$\lambda z\,.\ \mathit{case}\ z\ \mathit{of}$$
$$\mathit{pair}\ u\ v\ :\ +\ u\ v$$

és így a fenti szabály szerint

$$\mathit{add\_pair} =$$
$$\mathit{UPP\_pair}\ (\lambda xy\,.\ +\ x\ y)$$

A *pair* 2 3 argumentumra, felhasználva az *UPP* definícióját,

$$\mathit{add\_pair}(\mathit{pair}\ 2\ 3) =$$
$$\mathit{UPP\_pair}\ (\lambda xy\,.\ +\ x\ y)\ (\mathit{pair}\ 2\ 3)\ \rightarrow$$

$(\lambda xy\,.\,+\;x\,y)(SEL\_pair\_1\;(pair\;2\;3))(SEL\_pair\_2(pair\;2\;3)) \;\rightarrow^{+}$
$(\lambda xy\,.\,+\;x\,y)\;2\;3 \;\rightarrow^{+}$
$+\;2\;3 \;\rightarrow$
5 □

5.4.2. A case-kifejezés összegtípusú konstruktorral

A 3.5.2. pontban láttuk, hogy az

$$f\;(s_1\;\;x_{1,2}\;\;\ldots x_{1,n}) \;=\; E_1$$
$$f\;(s_2\;\;x_{2,2}\;\;\ldots x_{2,n}) \;=\; E_2$$
$$\ldots$$
$$f\;(s_m\;\;x_{m,2}\;\;\ldots x_{m,n}) \;=\; E_m$$

definícióból, ahol s_i $(1 \le i \le m)$ összegtípusú konstruktor és $x_{i,j}$ $(1 \le i \le m, 1 \le j \le n)$ változó, továbbá feltéve, hogy $\mathrm{TE}(\!|\,E_i\,|\!) = E_i$ $(1 \le i \le m)$, a következő kifejezést kaptuk:

$$\begin{aligned} f = \lambda z\,.\;(\;\; & ((\lambda(s_1\;x_{1,1}\;x_{1,2}\ldots x_{1,n})\,.\,E_1)\;z) \\ & [\!]\;((\lambda(s_2\;x_{2,1}\;x_{2,2}\ldots x_{2,n})\,.\,E_2)\;z) \\ & \ldots \\ & [\!]\;((\lambda(s_m\;x_{m,1}\;x_{m,2}\ldots x_{m,n})\,.\,E_m)\;z) \\ & [\!]\;ERROR \\ &) \end{aligned}$$

A 4. fejezetben megadott módszert alkalmazva, a *match* kifejezéseket *case* kifejezésekre átírva, a kifejezés alakja:

$$\begin{aligned} f = \lambda z\,.\;\; & case\;z\;of \\ & s_1\;x_{1,1}\;x_{1,2}\ldots x_{1,n} \;\;:\; E_1 \\ & s_2\;x_{2,1}\;x_{2,2}\ldots x_{2,n} \;\;:\; E_2 \\ & \ldots \\ & s_m\;x_{m,1}\;x_{m,2}\ldots x_{m,n} \;:\; E_m \end{aligned}$$

A case-kifejezés λ-kalkulusba történő átalakításához az elágazási lehetőségek felsorolását kell megszüntetni, hiszen a λ-kalkulusban ilyen nincs. Ezt egy új, *Case_T* nevű kifejezéssel fogjuk megvalósítani, amelyet úgy definiálunk, hogy az első argumentuma legyen a *case*-ben felsorolt minták egyike, és a további argumentumok legyenek $E_1, E_2, \ldots, E_m$. Ha az első argumentum az s_i konstruktort tartalmazó minta, akkor a *Case_T* eredményül az E_i kifejezést adja. Látható, hogy a *Case_T*

redukciója az $s_1, s_2, \ldots, s_m$ konstruktorokkal és argumentumaikkal meghatározott típusra működik, a nevében szereplő T erre a megadott típusra utal.

A $\mathsf{Case\_}T$ δ-függvény szabálya a következő:

$$\mathsf{Case\_}T\ (s_i\ x_{i,1}\ x_{i,2} \ldots x_{i,n})\ E_1\ E_2 \ldots E_n\ \to E_i \qquad (1 \le i \le m)$$
$$\mathsf{Case\_}T\ \bot \qquad\qquad E_1\ E_2 \ldots E_n\ \to \bot$$

ahol s_i az $s_1, s_2, \ldots, s_m$ összegkonstruktorokkal meghatározott T típus egyik konstruktora.

Az s_i konstruktorhoz tartozó műveletet a 5.1.2. pontból már ismert $\mathsf{UPS\_}s_i$ művelettel végezhetjük el, emlékeztetőül:

$$\lambda(s\ x_1 x_2 \ldots x_n)\,.\,E \ \rightsquigarrow\ \mathsf{UPS\_}s\ (\lambda x_1\,.\,\lambda x_2\,.\,\cdots\,.\,\lambda x_n\,.\,E)$$

Ennek felhasználásával a fenti case-t tartalmazó f kifejezés így írható át a λ-kalkulus kifejezésévé:

$$\begin{aligned}
&\lambda z\,.\,\mathsf{case}\ z\ \mathit{of}\\
&\qquad s_1\ x_{1,1}\ x_{1,2} \ldots x_{1,n} \ : E_1\\
&\qquad s_2\ x_{2,1}\ x_{2,2} \ldots x_{2,n} \ : E_2\\
&\qquad \ldots\\
&\qquad s_m\ x_{m,1}\ x_{m,2} \ldots x_{m,n} \ : E_m\\
&\rightsquigarrow\\
&\lambda z\,.\,\mathsf{Case_}T\ z\ \ (\mathsf{UPS_}s_1\ (\lambda x_{1,1}\,.\,\lambda x_{1,2}\,.\,\cdots\,.\,\lambda x_{1,n}\,.\,E_1)\ z)\\
&\qquad\qquad\ (\mathsf{UPS_}s_2\ (\lambda x_{2,1}\,.\,\lambda x_{2,2}\,.\,\cdots\,.\,\lambda x_{2,n}\,.\,E_2)\ z)\\
&\qquad\qquad\ \ldots\\
&\qquad\qquad\ (\mathsf{UPS_}s_m\ (\lambda x_{m,1}\,.\,\lambda x_{m,2}\,.\,\cdots\,.\,\lambda x_{m,n}\,.\,E_m)\ z)
\end{aligned}$$

5.4.2. Példa. *(A reflect kifejezés)*
A 4.1.8. példában szerepelt a *reflect* kifejezés case-t tartalmazó alakja:

```
reflect =
λx . case x of
        leaf u:        leaf u
        branch v w:    branch (reflect w) (reflect v)
```

Ez a kifejezés a következőképpen alakítható át a konstansos λ-kalkulus kifejezésére:

$$
\begin{aligned}
&\mathit{reflect} = \\
&\lambda x\,.\,\mathsf{Case\_Tree}\ \ x\ \ (\mathsf{UPS\_leaf}\,(\lambda u\,.\,\lambda x_{1,2}\,.\,\mathsf{leaf}\,u)\ x) \\
&\qquad\qquad\qquad (\mathsf{UPS\_branch}\,(\lambda v\,.\,\lambda w\,.\,\mathsf{branch}\,(\mathit{reflect}\ w)\,(\mathit{reflect}\ v))\ x)
\end{aligned}
$$

Határozzuk meg a $\mathit{reflect}\ (\mathsf{branch}\ E\ F)$ kifejezést, a branch a *Tree* típus második konstruktora volt.

$$
\begin{aligned}
&\mathit{reflect}\ (\mathsf{branch}\ E\ F) \equiv \\
&(\lambda x\,.\,\mathsf{Case\_Tree}\ \ x\ \ (\mathsf{UPS\_leaf}\,(\lambda u\,.\,\lambda x_{1,2}\,.\,\mathsf{leaf}\,u)\ x) \\
&\qquad\qquad\qquad (\mathsf{UPS\_branch}\,(\lambda v\,.\,\lambda w\,.\,\mathsf{branch}\,(\mathit{reflect}\ w)\,(\mathit{reflect}\ v))\ x)) \\
&\qquad\qquad\qquad\quad (\mathsf{branch}\ E\ F) \\
&\to \\
&\mathsf{Case\_Tree}\ \ (\mathsf{branch}\ E\ F)\ \ (\mathsf{UPS\_leaf}\,(\lambda u\,.\,\lambda x_{1,2}\,.\,\mathsf{leaf}\,u)\,(\mathsf{branch}\ E\ F)) \\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{UPS\_branch}\,(\lambda v\,.\,\lambda w\,.\,\mathsf{branch}\,(\mathit{reflect}\ w)\,(\mathit{reflect}\ v)) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{branch}\ E\ F)) \\
&\to \\
&\mathsf{UPS\_branch}\,(\lambda v\,.\,\lambda w\,.\,\mathsf{branch}\,(\mathit{reflect}\ w)\,(\mathit{reflect}\ v))\,(\mathsf{branch}\ E\ F)\ \to \\
&(\lambda v\,.\,\lambda w\,.\,\mathsf{branch}\,(\mathit{reflect}\ w)\,(\mathit{reflect}\ v))\ E\ F\ \to^{+} \\
&\mathsf{branch}\,(\mathit{reflect}\ F)\,(\mathit{reflect}\ E)
\end{aligned}
$$

□

5.5. A vastagvonal operátor

Alakítsuk át az infix [] vastagvonal operátort a λ-kalkulus egy δ-függvényére. A [] definíciója a 3.5.1. pontban a következő volt:

$$
\begin{aligned}
&E \ [] \ F \ \to E, \ \ \mathit{ha}\ E \neq \bot\ \mathit{és}\ E \neq \mathsf{fail}, \\
&\mathsf{fail} \ [] \ F \ \to F, \\
&\bot \ [] \ F \ \to \bot.
\end{aligned}
$$

A [] műveletet könnyen megvalósíthatjuk egy fatbar δ-függvénnyel, melyre

$$
\begin{aligned}
&\mathsf{fatbar}\ E \ \ F \ \to E, \ \ \mathit{ha}\ E \neq \bot\ \mathit{és}\ E \neq \mathsf{fail}, \\
&\mathsf{fatbar}\ \mathsf{fail}\ F \ \to F, \\
&\mathsf{fatbar}\ \bot \ \ F \ \to \bot.
\end{aligned}
$$

Látható, hogy a [] csak egy *szintaktikus cukor* az egyszerűbb infix jelölésre, hiszen az

$$E \parallel F \rightsquigarrow \mathit{fatbar}\ E\ F$$

összefüggés nyilvánvalóan teljesül.

6. FEJEZET

Listakifejezések átalakítása a kibővített λ-kalkulusba

A *listakifejezések* a funkcionális programozás tipikus szintaktikus egységei, a listákat kezelő programok írását (és olvasását) lényegesen megkönnyítik. Egy listát megadhatunk a *Nil* és *cons* konstruktorokkal, a *cons* x xs kifejezést röviden $x{:}xs$-sel is jelölhetjük. A listát megadhatjuk elemeinek felsorolásával is, ekkor a lista elemeit vesszővel választjuk el, és a listát alkotó elemeket a [ ] zárójelpár határolja. Egy listát röviden az *xs* vagy *ys* jellel is jelölhetünk. A felsorolást a szokásos módon rövidíthetjük a $..$ jel alkalmazásával.

A listakifejezéseket, hasonlóan a listák rövid leírásához, a [ ] szögletes zárójelpár közé tesszük. A listakifejezések a következő formájúak:

$$\langle \textit{listakifejezés} \rangle ::= [\langle \textit{kifejezés} \rangle \mid \langle \textit{minősítő}_1 \rangle ; \cdots ; \langle \textit{minősítő}_n \rangle] \qquad (n \geq 0)$$

ahol

$$\begin{array}{lll} \langle \textit{minősítő} \rangle & := & \langle \textit{generátor} \rangle \\ & \mid & \langle \textit{szűrő} \rangle \\ \langle \textit{generátor} \rangle & := & \langle \textit{minta} \rangle \mathrel{<\!\!-} \langle \textit{listakifejezés} \rangle \\ \langle \textit{szűrő} \rangle & := & \langle \textit{Bool értékű kifejezés} \rangle \end{array}$$

A generátorban levő változót *generátorváltozónak*, a listát *generátorlistának* nevezzük. Ha a generátorok csak változóikban különböznek, akkor azok összevonhatók, például az $x <\!\!- \langle lista \rangle$ és az $y <\!\!- \langle lista \rangle$ generátorok $x, y <\!\!- \langle lista \rangle$ alakban is írhatók.

Ha egy listakifejezés minősítői között több generátor van, akkor mindig az utolsó változik leggyorsabban.

6.0.1. Példa. *(Két listakifejezés)*
Az *xs* és az *ys* listák elemeiből az elemek sorrendje szerint képzett párokat a

$$[\textit{pair}\ x\ y \mid x <\!\!- xs; y <\!\!- ys]$$

listakifejezéssel lehet leírni. Például, ha $xs = [1, 2, 3]$ és $ys = [a, b]$, akkor ebből a két listából ez a listakifejezés a

[*pair* 1 *a*, *pair* 1 *b*, *pair* 2 *a*, *pair* 2 *b*, *pair* 3 *a*, *pair* 3 *b*]

listát készíti, mivel a második generátor gyorsabban változik, mint az első.

Azoknak a Pitagoraszi számhármasoknak a listáját, melyben a számhármas számainak összege nem nagyobb n-nél, a következő listakifejezés írja le:

$$[\mathit{triple}\ x\ y\ z \mid x, y, z <- [1\,..\,n];\ x + y + z \leq n;\ \mathit{square}\ x + \mathit{square}\ y = \mathit{square}\ z] \qquad \square$$

6.1. Listakifejezések redukciós szabályai

Jelölje a továbbiakban két lista összefűzését az *append* művelet, amit röviden az infix ++ jellel is leírhatunk.

Most megadjuk a listakifejezések *redukciós szabályait*:

(1)	$[E \mid x <- [\,];\ Q]$	$\rightarrow [\,]$
(2)	$[E \mid x <- y{:}ys;\ Q]$	$\rightarrow [E \mid Q][x := y] \mathbin{+\!\!+} [E \mid x <- ys;\ Q]$
(3)	$[E \mid \mathit{true};\ Q]$	$\rightarrow [E \mid Q]$
(4)	$[E \mid \mathit{false};\ Q]$	$\rightarrow [\,]$
(5)	$[E \mid\ ]$	$\rightarrow [E]$

Az első két szabály a generátorokra vonatkozik. A (2) szabály szerint egy generátor két listát generál, és ezeket a listákat az *append* művelettel kapcsolja össze. Az első lista a generátor listája első elemének felhasználásával készül, és látható, hogy ezzel az elemmel egy helyettesítést kell elvégezni. A második lista generátorának listájából az első elem kikerül, vagyis ez eggyel rövidebb lesz. Az (1) szabály biztosítja azt, hogy a (2) szabály ismételt alkalmazásai terminálnak, hiszen ha egy listakifejezés generátorlistája kiürül, akkor ez a kifejezés az üres listát jelenti.

A harmadik és negyedik szabály a szűrőkre vonatkozik, a szabályokból látható, hogy ha több szűrő van, akkor ezek az „és” logikai művelettel vannak összekapcsolva. $[E \mid Q]$ esetén, ha mindegyik szűrő értéke *true*, akkor az E kifejezést, egyébként az üres listát kapjuk eredményül.

6.1.1. Példa. *(Listakifejezés redukciói)*
Határozzuk meg az

$$[* x\ 2 \mid x <- [1, 2, 3, 4];\ \mathit{even}\ x]$$

listakifejezés által generált listát.

Ha a redukciókat mohó algoritmussal végezzük, akkor az (2) szabály alkalmazásaival, végül az (1) szabállyal a következő listát kapjuk:

$[*\ x\ 2 \mid x <- [1,2,3,4];\ \mathit{even}\ x] \rightarrow$

$[*\ x\ 2 \mid \mathit{even}\ x][x := 1] +\!+ [*\ x\ 2 \mid x <- [2,3,4];\ \mathit{even}\ x] \equiv$

$[*\ 1\ 2 \mid \mathit{even}\ 1] +\!+ [*\ x\ 2 \mid x <- [2,3,4];\ \mathit{even}\ x] \rightarrow$

$[*\ 1\ 2 \mid \mathit{even}\ 1] +\!+ [*\ x\ 2 \mid \mathit{even}\ x][x := 2] +\!+ [*\ x\ 2 \mid x <- [3,4];\ \mathit{even}\ x] \equiv$

$[*\ 1\ 2 \mid \mathit{even}\ 1] +\!+ [*\ 2\ 2 \mid \mathit{even}\ 2] +\!+ [*\ x\ 2 \mid x <- [3,4];\ \mathit{even}\ x] \rightarrow$

$[*\ 1\ 2 \mid \mathit{even}\ 1] +\!+ [*\ 2\ 2 \mid \mathit{even}\ 2] +\!+ [*\ x\ 2 \mid \mathit{even}\ x][x := 3] +\!+$
$[*\ x\ 2 \mid x <- [4];\ \mathit{even}\ x] \equiv$

$[*\ 1\ 2 \mid \mathit{even}\ 1] +\!+ [*\ 2\ 2 \mid \mathit{even}\ 2] +\!+ [*\ 3\ 2 \mid \mathit{even}\ 3] +\!+$
$[*\ x\ 2 \mid x <- [4];\ \mathit{even}\ x] \rightarrow$

$[*\ 1\ 2 \mid \mathit{even}\ 1] +\!+ [*\ 2\ 2 \mid \mathit{even}\ 2] +\!+ [*\ 3\ 2 \mid \mathit{even}\ 3] +\!+$
$[*\ x\ 2 \mid \mathit{even}\ x][x := 4] +\!+ [*\ x\ 2 \mid x <- [\,];\ \mathit{even}\ x] \rightarrow^{+}$

$[*\ 1\ 2 \mid \mathit{even}\ 1] +\!+ [*\ 2\ 2 \mid \mathit{even}\ 2] +\!+ [*\ 3\ 2 \mid \mathit{even}\ 3] +\!+$
$[*\ 4\ 2 \mid \mathit{even}\ 4]$

Kiértékelve a szűrőket, eredményül a $[4] +\!+ [8] \equiv [4,8]$ listát kapjuk. □

A példából is látható, hogy a listakifejezés *mohó kiértékelése* előállítja a generátorlistából származó összes elemet, és csak ezután vizsgálja a szűrők értékeit. A *lusta kiértékelés* először pontosan meghatározza az eredménylistába kerülő, a generátorlista első eleméből származó első elemet, és nem foglalkozik a generátorlista többi elemével. Ha ez megtörtént, veszi a generátorlista következő elemét, ebből pontosan meghatározza a második elemet, és így tovább. Látható, hogy a lusta kiértékelés alkalmas végtelen generátorlisták és végtelen listák kezelésére is.

6.1.2. Példa. *(Redukciók lusta kiértékeléssel)*
Határozzuk meg az előző példában szereplő

$[*\ x\ 2 \mid x <- [1,2,3,4];\ \mathit{even}\ x]$

listakifejezés által generált listát, most lusta kiértékeléssel.

$[*\ x\ 2 \mid x <- [1,2,3,4];\ \mathit{even}\ x] \rightarrow$

$[*\ x\ 2 \mid \mathit{even}\ x][x := 1] +\!+ [*\ x\ 2 \mid x <- [2,3,4];\ \mathit{even}\ x] \equiv$

$[*\ 1\ 2 \mid \mathit{even}\ 1] +\!+ [*\ x\ 2 \mid x <- [2,3,4];\ \mathit{even}\ x] \rightarrow$

$[\,] +\!+ [*\ x\ 2 \mid x <- [2,3,4];\ \mathit{even}\ x] \rightarrow^{+}$

$[*\ x\ 2 \mid \mathit{even}\ x][x := 2] +\!+ [*\ x\ 2 \mid x <- [3,4];\ \mathit{even}\ x] \equiv$

$[*\ 2\ 2 \mid \mathit{even}\ 2] +\!+ [*\ x\ 2 \mid x <- [3,4];\ \mathit{even}\ x] \rightarrow^{+}$

$[4] \mathbin{++} [* \; x \; 2 \mid x <\!- [3,4]; \mathit{even}\; x] \;\rightarrow$
$[4] \mathbin{++} [* \; x \; 2 \mid \mathit{even}\; x][x := 3] \mathbin{++} [* \; x \; 2 \mid x <\!- [4]; \mathit{even}\; x] \equiv$
$[4] \mathbin{++} [* \; 3 \; 2 \mid \mathit{even}\; 3] \mathbin{++} [* \; x \; 2 \mid x <\!- [4]; \mathit{even}\; x] \;\rightarrow$
$[4] \mathbin{++} [\,] \mathbin{++} [* \; x \; 2 \mid x <\!- [4]; \mathit{even}\; x] \;\rightarrow^{+}$
$[4] \mathbin{++} [* \; x \; 2 \mid \mathit{even}\; x][x := 4] \mathbin{++} [* \; x \; 2 \mid x <\!- [\,]; \mathit{even}\; x] \equiv$
$[4] \mathbin{++} [* \; 4 \; 2 \mid \mathit{even}\; 4] \mathbin{++} [* \; x \; 2 \mid x <\!- [\,]; \mathit{even}\; x] \;\rightarrow^{+}$
$[4] \mathbin{++} [8] \mathbin{++} [* \; x \; 2 \mid x <\!- [\,]; \mathit{even}\; x] \;\rightarrow$
$[4] \mathbin{++} [8] \mathbin{++} [\,] \equiv$
$[4, 8]$ □

6.1.3. Példa. *(Végtelen lista)*
Határozzuk meg az $[x \mid x <\!- [1, 2, \ldots]; \mathit{odd}\; x]$ listakifejezés által generált listát.

$[x \mid x <\!- [1, 2, \ldots]; \mathit{odd}\; x] \;\rightarrow$
$[x \mid \mathit{odd}\; x][x := 1] \mathbin{++} [x \mid x <\!- [2, 3, \ldots]; \mathit{odd}\; x] \equiv$
$[1 \mid \mathit{odd}\; 1] \mathbin{++} [x \mid x <\!- [2, 3, \ldots]; \mathit{odd}\; x] \;\rightarrow$
$[1] \mathbin{++} [x \mid x <\!- [2, 3, \ldots]; \mathit{odd}\; x] \;\rightarrow$
$[1] \mathbin{++} [x \mid \mathit{odd}\; x][x := 2] \mathbin{++} [x \mid x <\!- [3, 4, \ldots]; \mathit{odd}\; x] \equiv$
$[1] \mathbin{++} [2 \mid \mathit{odd}\; 2] \mathbin{++} [x \mid x <\!- [3, 4, \ldots]; \mathit{odd}\; x] \;\rightarrow$
$[1] \mathbin{++} [x \mid x <\!- [3, 4, \ldots]; \mathit{odd}\; x] \;\rightarrow$
$[1] \mathbin{++} [x \mid \mathit{odd}\; x][x := 3] \mathbin{++} [x \mid x <\!- [4, 5, \ldots]; \mathit{odd}\; x] \equiv$
$[1] \mathbin{++} [3 \mid \mathit{odd}\; 3] \mathbin{++} [x \mid x <\!- [4, 5, \ldots]; \mathit{odd}\; x] \;\rightarrow$
$[1] \mathbin{++} [3] \mathbin{++} [x \mid x <\!- [4, 5, \ldots]; \mathit{odd}\; x] \;\rightarrow$
$[1, 3] \mathbin{++} [x \mid x <\!- [4, 5, \ldots]; \mathit{odd}\; x] \;\rightarrow$
$\ldots$ □

6.2. Listakifejezések átalakítása

A programozási nyelvekben az üres lista jele [ ], a λ-kalkulusban *nil*, az $x{:}xs$ lista *cons x xs*, ezért ezekre megadhatjuk a következő triviális átalakítási szabályokat:

$\mathrm{TE}(\!|\,[\,]\,|\!) \;\Longrightarrow\; \mathit{nil}$
$\mathrm{TE}(\!|\,x{:}xs\,|\!) \;\Longrightarrow\; \mathit{cons}\; x\; xs$

Az egyszerűbb leírás miatt azonban a λ-kifejezésekben is gyakran a listák záró-

jeles alakját használjuk.

A listakifejezéseknek a kibővített λ-kalkulusba történő átalakítása a listakifejezések redukciós szabályain alapul.

Az (1) és (2) redukciós szabály egy *flatmap* függvénnyel valósítható meg, a (3) és (4) redukciós szabály egyszerűen egy *if* utasítással, az (5) redukciós szabály pedig a *cons* listakonstruktorral írható le.

A *flatmap* függvény a „szokásos” map-függvény, az első argumentumában megadott kifejezést applikálja a második argumentumában megadott lista mindegyik elemére.

$$\mathit{flatmap}\ f\ \mathit{nil} = \mathit{nil}$$
$$\mathit{flatmap}\ f\ (\mathit{cons}\ x\ xs) = (f\ x) +\!+ (\mathit{flatmap}\ f\ xs)$$

Listakifejezések átalakítása a kibővített λ-kalkulusba:

$$\begin{array}{lcl}
\mathrm{TE}(\!|\,[E \mid x <\!- L; Q]\,|\!) & \Longrightarrow & \mathit{flatmap}\ (\lambda x\,.\,\mathrm{TE}(\!|\,[E \mid Q]\,|\!))\ \mathrm{TE}(\!|\,L\,|\!) \\
\mathrm{TE}(\!|\,[E \mid Q_1; Q]\,|\!) & \Longrightarrow & \mathit{if}\ \mathrm{TE}(\!|\,Q_1\,|\!)\ \mathrm{TE}(\!|\,[E \mid Q]\,|\!)\ \mathit{nil} \\
\mathrm{TE}(\!|\,[E \mid\ ]\,|\!) & \Longrightarrow & \mathit{cons}\ \mathrm{TE}(\!|\,E\,|\!)\ \mathit{nil}
\end{array}$$

Az átalakítás első szabálya szerint az $L \equiv [\ ]$ üres listára a *flatmap* függvény a *nil* kifejezést adja, ez az (1) redukciós szabálynak felel meg. Látható, hogy a listakifejezés generátorának változója lesz annak az absztrakciónak a változója, amelyik a *flatmap* első argumentuma, a generátorlista pedig a *flatmap* második argumentumát adja. Az applikációk eredménye pontosan a (2) redukciós szabály.

6.2.1. Példa. *(Listakifejezés átalakítása)*
Először alakítsuk át a 6.1.1. és 6.1.2. példákban szereplő listakifejezést, de a szűrővel most ne foglalkozzunk.

$$\begin{array}{l}
\mathrm{TE}(\!|\,[*\,x\,2 \mid x <\!- xs]\,|\!) \Longrightarrow \\
\mathit{flatmap}\ (\lambda x\,.\,\mathrm{TE}(\!|\,[*\,x\,2 \mid\ ]\,|\!))\mathrm{TE}(\!|\,xs\,|\!) \Longrightarrow \\
\mathit{flatmap}\ (\lambda x\,.\,\mathrm{TE}(\!|\,[*\,x\,2 \mid\ ]\,|\!))\ xs \Longrightarrow \\
\mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (\mathrm{TE}(\!|\,[*\,x\,2]\,|\!)\ \mathit{nil}))\ xs \Longrightarrow^{+} \\
\mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\,2)\ \mathit{nil}))\ xs
\end{array}$$

Ha a listagenerátorban az $xs \equiv [1,2,3,4]$ lista szerepel, azaz a

$$[*\,x\,2 \mid x <\!- [1,2,3,4]]$$

listakifejezésre:

$\mathrm{TE}(\!|\,[\,*\,x\,2 \mid x <\!- [1,2,3,4]]\,|\!) \Longrightarrow^{+}$
$\mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ [1,2,3,4] \rightarrow$
$\underline{(\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ 1}\ ++\ \mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ [2,3,4] \rightarrow^{+}$
$\underline{[\,*\ 1\ 2]}\ ++\ \mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ [2,3,4] \rightarrow$
$[2]\ ++\ \mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ [2,3,4] \rightarrow$
$[2]\ ++\ \underline{(\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ 2}\ ++\ \mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ [3,4] \rightarrow^{+}$
$[2]\ ++\ \underline{[\,*\ 2\ 2]}\ ++\ \mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ [3,4] \rightarrow$
$[2]\ ++\ [4]\ ++\ \mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ [2,3,4] \equiv$
$[2,4]\ ++\ \mathit{flatmap}\ (\lambda x\,.\,(\mathit{cons}\ (*x\ 2)\ \mathit{nil}))\ [2,3,4] \rightarrow^{+}$
$\ldots \rightarrow^{+}$
$[2,4,6,8]$ □

6.2.2. Példa. *(Listakifejezés szűrővel)*
Most alakítsuk át a 6.1.1. és 6.1.2. példákban szereplő listakifejezést úgy, hogy a szűrőt is figyelembe vesszük.

$\mathrm{TE}(\!|\,[\,*\,x\,2 \mid x <\!- xs;\ \mathit{even}\ x]\,|\!) \Longrightarrow$
$\mathit{flatmap}\ (\lambda x\,.\,\mathrm{TE}(\!|\,[\,*\,x\,2 \mid \mathit{even}\ x]\,|\!)\ xs \Longrightarrow$
$\mathit{flatmap}\ (\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ \mathrm{TE}(\!|\,[\,*\,x\,2 \mid\ ]\,|\!)\ \mathit{nil})\ xs \Longrightarrow$
$\mathit{flatmap}\ (\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ (\mathit{cons}\ \mathrm{TE}(\!|\,*x\,2\,|\!)\ \mathit{nil})\ \mathit{nil})\ xs \Longrightarrow$
$\mathit{flatmap}\ (\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ (\mathit{cons}\ (*x\ 2)\ \mathit{nil})\ \mathit{nil})\ xs$

Erre a kifejezésre, ha $xs \equiv [1,2,3,4]$

$\mathit{flatmap}\ (\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ (\mathit{cons}\ (*x\ 2)\ \mathit{nil})\ \mathit{nil})\ [1,2,3,4] \rightarrow$
$(\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ (\mathit{cons}\ (*x\ 2)\ \mathit{nil})\ \mathit{nil})\ 1\ ++$
$(\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ (\mathit{cons}\ (*x\ 2)\ \mathit{nil})\ \mathit{nil})\ [2,3,4] \rightarrow^{+}$
$\mathit{if}\ (\mathit{even}\ 1)\ (\mathit{cons}\ (*1\ 2)\ \mathit{nil})\ \mathit{nil}\ ++$
$(\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ (\mathit{cons}\ (*x\ 2)\ \mathit{nil})\ \mathit{nil})\ [2,3,4] \rightarrow^{+}$
$[\,]\ ++\ (\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ (\mathit{cons}\ (*x\ 2)\ \mathit{nil})\ \mathit{nil})\ [2,3,4] \rightarrow^{+}$
$\ldots \rightarrow^{+}$
$[4,8]$ □

6.2.1. Átalakítás case-utasítással

Az előző szakaszban a listakifejezések átalakítását a *flatmap* függvénnyel adtuk meg. Hatékonyabb eredményt kapunk, ha a *flatmap* függvényt case-utasítással adjuk meg. A listakifejezések átalakítási szabályában a *flatmap* két kifejezés applikációjával szerepelt, ezért elegendő a

$\mathit{flatmap}\ (\lambda x\,.\,E)\ F$

alakkal foglalkoznunk, ahol F egy lista.

$$
\begin{array}{l}
\mathit{flatmap}\ (\lambda x\,.\,E)F \equiv \\
\mathit{letrec}\ f = \lambda y\,.\ \mathit{case}\ y\ \mathit{of} \\
\qquad\qquad\qquad \mathit{nil} \quad\ : \mathit{nil} \\
\qquad\qquad\qquad \mathit{cons}\ x\ xs\ : \mathit{append}\ E\ (f\ xs) \\
\quad \mathit{in}\ (f\ F)
\end{array}
$$

ahol f, y és xs új változók. Könnyen belátható, hogy ez a kifejezés valóban megfelel az eredeti kifejezésnek.

Ha ezt beépítjük a 6.2. szakaszban szereplő

$$\mathrm{TE}(\!|\,[E \mid x <\!- L; Q]\,|\!) \implies \mathit{flatmap}\ (\lambda x\,.\,\mathrm{TE}(\!|\,[E \mid Q]\,|\!))\ \mathrm{TE}(\!|\,L\,|\!)$$

szabályba, akkor a következő átalakítási szabályt kapjuk:

$$
\begin{array}{l}
\mathrm{TE}(\!|\,[E \mid x <\!- L; Q]\,|\!) \implies \\
\mathit{letrec}\ f = \lambda y\,.\ \mathit{case}\ y\ \mathit{of} \\
\qquad\qquad\qquad \mathit{nil} \quad\ : \mathit{nil} \\
\qquad\qquad\qquad \mathit{cons}\ x\ xs\ : \mathit{append}\ \mathrm{TE}(\!|\,[E \mid Q]\,|\!)\ (f\ xs) \\
\quad \mathit{in}\ (f\ \mathrm{TE}(\!|\,L\,|\!))
\end{array}
$$

ahol f, y és xs új változók.

Ez bonyolultabbnak tűnik az előző alaknál, de végrehajtása a *case* miatt sokkal gyorsabb lesz. A 6.2. szakaszban szereplő további két átalakítási szabály nem változik.

6.2.3. Példa. *(Átalakítás case-utasítással)*

Határozzuk meg a 6.1.1. példában szereplő

$$[\ast\ x\ 2 \mid x <\!- [1,2,3,4]; \mathit{even}\ x]$$

listakifejezés λ-kifejezését. A 6.2.2. példában láttuk, hogy

$$
\begin{array}{l}
\mathrm{TE}(\!|\,[\ast\ x\ 2 \mid x <\!- [1,2,3,4]; \mathit{even}\ x]\,|\!) \implies \\
\mathit{flatmap}\ (\lambda x\,.\,\mathit{if}\ (\mathit{even}\ x)\ (\mathit{cons}\ (\ast x\ 2)\ \mathit{nil})\ \mathit{nil})\ [1,2,3,4]
\end{array}
$$

Erre alkalmazva a fenti átalakítási szabályt:

$$\mathrm{TE}(\!|\,[\ast\ x\ 2 \mid x <\!- [1,2,3,4]; \mathit{even}\ x]\,|\!) \implies$$

```
letrec f = λy . case y of
                  nil       : nil
                  cons x xs : append
                                (if (even x) (cons (∗x 2) nil) nil)
                                (f xs)
  in (f TE⦇[1, 2, 3, 4]⦈)
```

kifejezést kapjuk.

Ez egy egydefiníciós egyszerű letrec-kifejezés, ami a 3.3.1.-ben megadott átalakítással, az Y fixpont-kombinátor felhasználásával egy egydefiníciós egyszerű let-kifejezéssé alakítható át:

```
let f = Y (λf . (λy . case y of
                        nil       : nil
                        cons x xs : append
                                      (if (even x) (cons (∗x 2) nil) nil)
                                      (f xs) ) )
  in (f TE⦇[1, 2, 3, 4]⦈)
```

Ez pedig a 3.4.3. pontban megadott átalakítással, majd egy β-redukcióval

```
(λf . f TE⦇[1, 2, 3, 4]⦈)
    (Y (λf . λy . case y of
                    nil       : nil
                    cons x xs : append
                                  (if (even x) (cons (∗x 2) nil) nil)
                                  (f xs) ) )
→
Y (λf . λy . case y of
               nil       : nil
               cons x xs : append
                             (if (even x) (cons (∗x 2) nil) nil)
                             (f xs) )
   TE⦇[1, 2, 3, 4]⦈
```

Az eredményül kapott kifejezés alakja Y $(\lambda xy\,.\,E)\;F$, amire alkalmazható a λ-kalkulusból jól ismert

$$\mathsf{Y}\;(\lambda xy\,.\,E)\;F \;=\; E\;[x := \mathsf{Y}\;(\lambda xy\,.\,E)\;F]\;[y := F]$$

egyenlőség. Ezt alkalmazhatjuk a fenti kifejezésre, így ha a rövidebb leírás érdekében a *case* részkifejezést E-vel jelöljük, azaz a kifejezésünk

$$\mathsf{Y}\;(\lambda f\,.\,\lambda y\,.\,E)\;\mathrm{TE}⦇[1, 2, 3, 4]⦈ \qquad (*)$$

alakú, akkor

(*case* y *of*
  nil : *nil*
  cons x xs : *append*
    (*if* (*even* x) (*cons* ($*x$ 2) *nil*) *nil*)
    (f xs))
[f := Y($\lambda f\,.\,\lambda y\,.\,E$)] [$y$:= TE(⦇[1, 2, 3, 4]⦈)]

Az [1,2,3,4] listát most átírva

TE(⦇[1, 2, 3, 4]⦈ $\Longrightarrow$ *cons* 1 (TE(⦇[2, 3, 4]⦈),

végrehajthatjuk a *case* utasítást, és az

append (*if* (*even* 1) (*cons* ($*1 2) *nil*) *nil*))
((Y ($\lambda f\,.\,\lambda y\,.\,E$) TE(⦇[2, 3, 4]⦈)

kifejezést kapjuk, melyből az *if* végrehajtásával kifejezésünk az

append nil ((Y $\lambda f\,.\,\lambda y\,.\,E$) TE(⦇[2, 3, 4]⦈)

azaz az

Y ($\lambda f\,.\,\lambda y\,.\,E$) TE(⦇[2, 3, 4]⦈)

kifejezésre egyszerűsödik. Látható, hogy visszakaptuk a fenti *-gal megjelölt kifejezést, csak eggyel rövidebb listára.

Tovább folytatva a kifejezés kiértékelését, a fentiekhez teljesen hasonló lépéseket alkalmazva, eljutunk a [4, 8] eredményhez. □

A listakifejezések átalakításának algoritmusát tovább lehet javítani. A lehetséges módosítások közül csak egyet emelünk ki:

Az előző példában is többször előfordult ez a két sor:

nil : *nil*
cons x xs : *append*
  (*if* (*even* x) (*cons* ($*x$ 2) *nil*) *nil*)
  (f xs))

A *cons* x xs esetén, az *append* művelet szerint, először meg kell határozni az első, majd a második listát, és utána ezeket össze kell fűzni.

Az első lista meghatározásakor, ha a logikai feltétel *true*, egy *nil* végű egyelemes listát kell készíteni, *false* esetén pedig a *nil* üres listát kapjuk eredményül. Az így kapott, a végén *nil*-t tartalmazó listához kell majd a második listát hozzáfűzni. Az időigényes *append* műveletet elkerülhetjük, ha a *nil* helyett azonnal a második listát

határozzuk meg, azaz ha a

nil : *nil*
cons x xs : *if* (*even x*) (*cons* (∗*x* 2) (*f xs*)) (*f xs*)

kifejezést állítjuk elő.

6.3. Listakifejezések mintával

Az előző szakaszokban olyan listakifejezéseket vizsgáltunk, ahol a generátorban, a nyíl baloldalán egy változó szerepelt. Azonban ezen a helyen tetszőleges minta is előfordulhat.

⟨*generátor*⟩ := ⟨*minta*⟩ <− ⟨*lista*⟩

Változó használata esetén a generátort tartalmazó listakifejezés redukciójára a következő két szabályt adtuk meg:

(1) $[E \mid x <- [\,];\ Q] \quad \rightarrow [\,]$
(2) $[E \mid x <- y{:}ys;\ Q] \rightarrow [E \mid Q][x := y] \mathbin{++} [E \mid x <- ys;\ Q]$

Minta használatakor az első szabály természetesen változatlan marad. A második szabályban az $[E \mid Q][x := y]$ helyettesítés a $(\lambda x\,.\,[E \mid Q])y$ függvényapplikációra vezethető vissza, hiszen

$(\lambda x\,.\,[E \mid Q])y \ \rightarrow\ [E \mid Q][x := y]$

Ezt az ötletet felhasználva, és már ismerve a minta-absztrakciókat, tudjuk képezni a $(\lambda p\,.\,[E \mid Q])\ y$ absztrakciót. Az applikáció azonban *false* eredményt is adhat, ha a mintaillesztés nem volt sikeres. A $[\!]$ vastagvonal operátort éppen erre a célra vezettük be, így megadhatjuk, hogy sikertelen mintaillesztés esetén a generátor által generált lista legyen üres.

A mintát tartalmazó listakifejezések *redukciós szabályai* tehát a következők:

(6) $[E \mid p <- [\,];\ Q] \quad \rightarrow [\,]$
(7) $[E \mid p <- y{:}ys;\ Q] \rightarrow (\ \ ((\lambda p\,.\,[E \mid Q])\ y)\ [\!]\ [\,]\ )\ \mathbin{++}\ [E \mid x <- ys;\ Q]$

6.3.1. Példa. *(Listakifejezés mintával)*

A listakifejezés legyen

$[\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y \ \mathrel{<\!\!-}\ [1, \mathit{pair}\ 2\ 3, \mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]]$

A kifejezés redukálása a következő:

$[\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y \ \mathrel{<\!\!-}\ [1, \mathit{pair}\ 2\ 3, \mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]\ ] \rightarrow$
$(\ ((\lambda(\mathit{pair}\ x\ y)\ .\ \mathit{pair}\ x\ y)\ 1)\ [\!]\ [\ ])$
$\quad ++\ [\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y \ \mathrel{<\!\!-}\ [\mathit{pair}\ 2\ 3, \mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]] \rightarrow^{+}$
$(\mathit{fail}\ [\!]\ [\ ])\ ++\ [\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y \ \mathrel{<\!\!-}\ [\mathit{pair}\ 2\ 3, \mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]] \rightarrow$
$[\ ]\ ++\ [\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y \ \mathrel{<\!\!-}\ [\mathit{pair}\ 2\ 3, \mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]] \rightarrow$
$[\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y \ \mathrel{<\!\!-}\ [\mathit{pair}\ 2\ 3, \mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]] \rightarrow$

a generátorlista második elemével folytatva,

$(\ ((\lambda(\mathit{pair}\ x\ y)\ .\ \mathit{pair}\ x\ y)\ \mathit{pair}\ 2\ 3)\ [\!]\ [\ ])$
$\quad ++\ [\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y \ \mathrel{<\!\!-}\ [\mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]] \rightarrow^{+}$
$[\mathit{pair}\ 2\ 3]\ ++\ [\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y \ \mathrel{<\!\!-}\ [\mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]] \rightarrow$
$\ldots$

Látható, hogy a listagenerátor további elemeire a mintaillesztés *fail* lesz, így a listakifejezés a

$[\mathit{pair}\ 2\ 3]$

listát generálja. □

A változós listakifejezésekhez hasonlóan, a mintát tartalmazó listakifejezéseknek a kibővített λ-kalkulusba történő átalakítása is a listakifejezések redukciós szabályain alapul. A (6) és (7) redukciós szabályoknak megfelelő átalakításokat most is a *flatmap* függvénnyel írjuk le.

$$\mathrm{TE}(\!|\,[E \mid p \mathrel{<\!\!-} L; Q]\,|\!) \implies \mathit{flatmap}\ (\lambda x\ .\ (\ ((\lambda \mathrm{TE}(\!|\,p\,|\!)\ .\ \mathrm{TE}(\!|\,[E \mid Q]\,|\!))\ x) \ [\!]\ [\ ]\ )\ )\ \mathrm{TE}(\!|\,L\,|\!)$$

A mintát tartalmazó listakifejezés is leírható a *case* utasítás felhasználásával.

$\mathrm{TE}(\!|\,[E \mid p <\!- L; Q]\,|\!) \Longrightarrow$
letrec $f = \lambda y\,.$ *case* y *of*
$\quad$ *nil* $\quad$: *nil*
$\quad$ *cons* $x\ xs$: *append*
$\quad\quad$ ($((\lambda \mathrm{TE}(\!|\,p\,|\!)\,.\,\mathrm{TE}(\!|\,[E \mid Q]\,|\!))\ x)$
$\quad\quad$ [] [ ]
$\quad\quad$)
$\quad\quad$ $(f\ xs)$
in $(f\ \mathrm{TE}(\!|\,L\,|\!))$

ahol f, y *és* xs *új változók.*

6.3.2. Példa. *Listakifejezés mintával)*

Alakítsuk át az előző, 6.3.1. példában szereplő listakifejezést:

[*pair* $x\ y$ | *pair* $x\ y$ <− [1, *pair* 2 3, *cons* 4 [5], *leaf* 6]]

Először határozzuk meg a listakifejezés λ-kifejezését a *flatmap* felhasználásával.

[*pair* $x\ y$ | *pair* $x\ y$ <− [1, *pair* 2 3, *cons* 4 [5], *leaf* 6]] $\Longrightarrow$
flatmap $(\lambda x\,.$ ($((\lambda(\mathit{pair}\ u\ v)\,.\,\mathit{pair}\ u\ v)x)$
$\quad$ [] [ ]
$\quad$)
)
$\mathrm{TE}(\!|\,[1, \mathit{pair}\ 2\ 3, \mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]\,|\!)$

Írjuk fel a λ-kifejezést a *case* utasítás használatával, a fenti átalakítási szabály szerint

$\mathrm{TE}(\!|\,[\mathit{pair}\ x\ y \mid \mathit{pair}\ x\ y$ <− [1, *pair* 2 3, *cons* 4 [5], *leaf* 6]]$\,|\!) \Longrightarrow$
letrec $f = \lambda y\,.$ *case* y *of*
$\quad$ *nil* $\quad$: *nil*
$\quad$ *cons* $x\ xs$: *append*
$\quad\quad$ ($((\lambda(\mathit{pair}\ u\ v)\,.\,\mathit{pair}\ u\ v)\ x)$
$\quad\quad$ [] [ ]
$\quad\quad$)
$\quad\quad$ $(f\ xs)$
in $(f\ \mathrm{TE}(\!|\,[1, \mathit{pair}\ 2\ 3, \mathit{cons}\ 4\ [5], \mathit{leaf}\ 6]\,|\!)\ )$

Ez egy egydefiníciós egyszerű letrec-kifejezés, és erre alkalmazhatjuk a 6.2.3. példában már használt átalakítási lépéseket.

A kifejezés a 3.3.1.-ben megadott átalakítással, az Y fixpont-kombinátor felhasználásával egy egydefiníciós egyszerű let-kifejezéssé alakítható át:

```
let f = Y (λf . (λy . case y of
                        nil          : nil
                        cons x xs    : append
                                         (  ((λ(pair u v) . pair u v) x)
                                         ⟦ [ ]
                                         )
                                         (f xs)  ) )
   in (f TE⟪[1, pair 2 3, cons 4 [5], leaf 6] ⟫ )
```

Ez pedig a 3.4.3. pontban megadott átalakítással, majd egy β-redukcióval

```
(λf . f TE⟪[1, pair 2 3, cons 4 [5], leaf 6] ⟫ )
     Y (λf . (λy . case y of
                        nil          : nil
                        cons x xs    : append
                                         (  ((λ(pair u v) . pair u v) x)
                                         ⟦ [ ]
                                         )
                                         (f xs)  ) )
→
Y (λf . (λy . case y of
                    nil          : nil
                    cons x xs    : append
                                     (  ((λ(pair u v) . pair u v) x)
                                     ⟦ [ ]
                                     )
                                     (f xs)  ) )
   TE⟪[1, pair 2 3, cons 4 [5], leaf 6] ⟫
```

Az eredményül kapott kifejezés alakja most is Y $(\lambda xy\,.\,E)\;F$, amire alkalmazható az előzőpontban is megadott

$$\mathsf{Y}\;(\lambda xy\,.\,E)\;F \;=\; E\;[x := \mathsf{Y}\;(\lambda xy\,.\,E)\;F]\;[y := F]$$

egyenlőség. Így, ha a rövidebb leírás érdekében a case részkifejezést most is E-vel jelöljük, azaz a kifejezésünk

$$\mathsf{Y}\;(\lambda f\,.\,\lambda y\,.\,E)\;\mathrm{TE}\langle\!|[1, \mathit{pair}\;2\;3, \mathit{cons}\;4\;[5], \mathit{leaf}\;6]|\!\rangle \qquad (**)$$

alakú, akkor

(*case* y *of*
  nil : *nil*
  cons x xs : *append*
    (((λ(*pair* u v) . *pair* u v) x)
    ⟦ [ ]
)
    (f xs))
[f := Y(λf . λy . E)] [y := TE⟪[1, *pair* 2 3, *cons* 4 [5], *leaf* 6]⟫]

Az TE⟪[1, *pair* 2 3, *cons* 4 [5], *leaf* 6]⟫ listát most átírva

TE⟪[1, *pair* 2 3, *cons* 4 [5], *leaf* 6]⟫ ⟹ *cons* 1 (TE⟪[*pair* 2 3, *cons* 4 [5], *leaf* 6]⟫

végrehajthatjuk a *case* utasítást, ekkor az

append
  (((λ(*pair* u v) . *pair* u v) 1)
  ⟦ [ ]
)
  ((Y(λf . λy . E) TE⟪[*pair* 2 3, *cons* 4 [5], *leaf* 6]⟫)

kifejezést kapjuk, melyből látható, hogy a

(λ(*pair* u v) . *pair* u v) 1

mintaillesztés nem lesz sikeres, azaz az *append* első listája üres lesz. Így a kifejezésünk az

(Y(λf . λy . E) TE⟪[*pair* 2 3, *cons* 4 [5], *leaf* 6]⟫)

kifejezésre egyszerűsödik. Látható, hogy visszakaptuk a fenti **-gal megjelölt kifejezést, csak eggyel rövidebb listára.

Tovább folytatva a kifejezés kiértékelését, a fentiekhez teljesen hasonló lépéseket alkalmazva, csak egy mintaillesztés lesz sikeres, így végül a [*pair* 2 3] eredményt kapjuk. □

7. FEJEZET

Kibővített λ-kalkulusból a kombinátor logikába

Ebben a fejezetben először megmutatjuk a *konstansos λ-kalkulus* λ-kifejezéseinek kombinátor logikai kifejezésekre történő átalakítását, majd egy módszert adunk a *kibővített λ-kalkulus* néhány egyszerű kifejezésének, a mintaabsztrakcióknak az átalakítására.

7.1. λ-kifejezés átalakítása kombinátor logika kifejezésére

Jelöljük egy λ-kifejezésnek *CL*-kifejezésbe történő átalakítását a $\rightsquigarrow_{\mathcal{K}}$ nyíllal, és az E λ-kifejezésnek a *CL*-kifejezésbe átalakított formáját $(E)_{\mathcal{K}}$-val, azaz $E \rightsquigarrow_{\mathcal{K}} (E)_{\mathcal{K}}$. A 2. fejezetben bevezetett jelölések szerint Λ a λ-kifejezések, $\mathcal{K}$ a *CL*-kifejezések halmaza, tehát most a $\Lambda \rightsquigarrow_{\mathcal{K}} \mathcal{K}$ átalakításról lesz szó.

Az E λ-kifejezésből származó $(E)_{\mathcal{K}}$ kifejezést az E szerkezete szerint adjuk meg.

$$
\begin{array}{llll}
x & \rightsquigarrow_{\mathcal{K}} (x)_{\mathcal{K}} & \equiv x, & \\
k & \rightsquigarrow_{\mathcal{K}} (k)_{\mathcal{K}} & \equiv k, & \textit{ahol } k \textit{ konstans}, \\
EF & \rightsquigarrow_{\mathcal{K}} (EF)_{\mathcal{K}} & \equiv (E)_{\mathcal{K}}\,(F)_{\mathcal{K}}, & \\
\lambda x\,.\,E & \rightsquigarrow_{\mathcal{K}} (\lambda x\,.\,E)_{\mathcal{K}} & \equiv \lambda^* x\,.\,(E)_{\mathcal{K}} &
\end{array}
$$

A λ^* jel a 2.2. szakaszban leírt *zárójeles absztrakciót* jelöli.

7.1.1. Példa. *(Az $f(x) = x^2 + 2x + 1$ kifejezés és az $f(3)$ értéke)*
Tekintsünk egy egyszerű programot, legyen $f(x) = x^2 + 2x + 1$ és határozzuk meg az $f(3)$ értéket. A funkcionális program a 3. fejezetben leírtak alapján a

letrec $f\ x = +(+(*\ x\ x)(*\ 2\ x))\ 1$
    in $f\ 3$

kifejezés lesz. A letrec-kifejezés definíciós részéből az

$f = \lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1$

összefüggést kapjuk, a let-kifejezés törzsének λ-kifejezése pedig $f\ 3$. Ezeket felhasználva, szintén a 3. fejezetben megadott átalakítások szerint meghatározhatjuk a program λ-kifejezését:

$(\lambda f\,.\,f\ 3)(\lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1)$

Alakítsuk át a λ-kifejezést *CL*-kifejezéssé, alkalmazzuk a fenti átalakítási szabályokat.

$(\lambda f\,.\,f\ 3)(\lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1) \rightsquigarrow_{\mathcal{K}}$
$(\lambda f\,.\,f\ 3)(\lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1)_{\mathcal{K}} \equiv$
$(\lambda f\,.\,f\ 3)_{\mathcal{K}}(\lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1)_{\mathcal{K}} \equiv$
$(\lambda^* f\,.\,(f\ 3)_{\mathcal{K}})(\lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1))_{\mathcal{K}} \equiv^+$
$(\lambda^* f\,.\,f\ 3)(\lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1))_{\mathcal{K}} \equiv^+$
$(C\ I\ 3)(\lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1))_{\mathcal{K}}$

Az applikáció második tagját külön levezetve:

$(\lambda x\,.\ +\,(+(*\ x\ x)(*\ 2\ x))\ 1))_{\mathcal{K}} \equiv^+$
$\lambda^* x\,.\ +\ (+\ (*\ x\ x)(*\ 2\ x))\ 1 \equiv$
$C'+\ (\lambda^* x\,.\ +\ (*\ x\ x)(*\ 2\ x))\ 1 \equiv$
$C'+\ (S'+\ (\lambda^* x\,.\ *\ x\ x)(\lambda^* x\,.\ *\ 2\ x))\ 1 \equiv$
$C'+\ (S'+\ (S'*\ (\lambda^* x\,.\,x)(\lambda^* x\,.\ x))(\lambda^* x\,.\ *\ 2\ x))\ 1 \equiv^+$
$C'+\ (S'+\ (S'*\ I\ I)(\lambda^* x\,.\ *\ 2\ x))\ 1 \equiv^+$
$C'+\ (S'+\ (S'*\ I\ I)(B'*\ 2\ I))\ 1$

Így tehát a program *CL*-kifejezése:

$(C\ I\ 3)(C'+\ (S'+\ (S'*\ I\ I)(B'*\ 2\ I))\ 1)$

A program eredményének a meghatározásához ezek után már csak a gyenge redukciókat kell végrehajtanunk,

$C\ I\ 3\ \underline{(C'+\ (S'+\ (S'*\ I\ I)(B'*\ 2\ I))\ 1)} \rightarrow_w$
$I\ \underline{(C'+\ (S'+\ (S'*\ I\ I)(B'*\ 2\ I))\ 1)}\ 3 \rightarrow_w$
$C'\underline{+}\ \underline{(S'+\ (S'*\ I\ I)(B'*\ 2\ I))}\ \underline{1}\ \underline{3} \rightarrow_w$
$+\ (S'\underline{+}\ \underline{(S'*\ I\ I)}\ \underline{(B'*\ 2\ I)}\ \underline{3})\ 1 \rightarrow_w$

$+\ (+\ (S'*\ I\ I\ 3)(B'*\ 2\ I\ 3))\ 1 \rightarrow_w$
$+\ (+\ (*\ (I\ 3)(I\ 3))(B'*\ 2\ I\ 3))\ 1 \rightarrow_w^+$
$+\ (+\ (*\ 3\ 3)(B'*\ 2\ I\ 3))\ 1 \rightarrow_w^+$
$+\ (+\ (*\ 3\ 3)(*\ 2\ 3))\ 1 \overset{+}{\rightarrow}_\delta$
$+\ (+\ 9\ 6)\ 1 \overset{+}{\rightarrow}_\delta$
16 □

7.2. A mintaabsztrakció átalakítása a kombinátor logika kifejezésére

Először a konstansos absztrakciók átalakítását vizsgáljuk, és csak ezek után foglalkozunk az összeg- és szorzatkonstruktoros absztrakciók *CL*-kifejezésekre történő transzformációjával. Az egyszerűség kedvéért most nem foglalkozunk a $\perp$ konstans problémájával.

7.2.1. Konstansos absztrakciók

A $\lambda k\,.\,E$ λ-kifejezés szemantikáját a 3.7.1. pontban adtuk meg. Az értelmezés szerint ha egy konstansos absztrakcióra egy olyan kifejezést applikálunk, aminek az értéke megegyezik az absztrakció konstansával, akkor a mintaillesztés eredményéül az absztrakció törzsét kapjuk, ha az érték nem azonos az absztrakció konstansával, akkor a mintaillesztés a *fail* eredményt adja.

A $(\lambda x\,.\,E)_{\mathcal{K}} \equiv \lambda^* x\,.\,(E)_{\mathcal{K}}$ azonossághoz hasonlóan a konstansos absztrakció átalakítását is a λ^* zárójeles absztrakcióval adjuk meg.

$$\lambda k\,.\,E \rightsquigarrow_{\mathcal{K}} (\lambda k\,.\,E)_{\mathcal{K}} \equiv \lambda^* k\,.\,(E)_{\mathcal{K}}$$

Bevezetünk egy új speciális kombinátort, a kombinátor neve legyen *Match*, és a konstansos absztrakció jelentését ezzel a kombinátorral írjuk le:

$$\lambda^* k\,.\,E \equiv \mathit{Match}\ k\ E$$

A *Match* kombinátor gyenge redukciós szabályai pedig legyenek a következők:

$$\mathit{Match}\ E\ F\ G \rightarrow_w F, \qquad \text{ha } E = G,$$
$$\mathit{Match}\ E\ F\ G \rightarrow_w \mathit{fail} \qquad \text{egyébként}$$

7.2.1. Példa. *(Konstansos absztrakció)*
A kibővített λ-kalkulusban az $f\ 1 = 2$ definícióból az f-re a $\lambda 1\,.\,2$ kifejezést kapjuk, és például

$f\ 1 = (\lambda 1\,.\,2)1 \ \rightarrow 2$
$f\ 3 = (\lambda 1\,.\,2)3 \ \rightarrow \mathit{fail}$

Az $f \equiv \lambda 1\,.\,2$ λ-kifejezést *CL*-kifejezésre alakítva

$(\lambda 1\,.\,2)_{\mathcal{K}} \equiv \lambda^* 1\,.\,2 \equiv \mathit{Match}\ 1\ 2$

és

$f\ 1 \equiv \mathit{Match}\ 1\ 2\ 1 \rightarrow_w 2$
$f\ 3 \equiv \mathit{Match}\ 1\ 2\ 3 \rightarrow_w \mathit{fail}$ □

7.2.2. Összegkonstruktoros absztrakciók

Az összegkonstruktoros absztrakciók értelmezésével a 3.7.2. pontban foglalkoztunk, és láttuk, hogy egy mintaapplikációból nem *fail* eredményt akkor kapunk, ha az absztrakcióban és az argumentumában szereplő konstruktor megegyezik. Hasonlóan értelmezzük az összegkonstruktoros mintákat a kombinátor logikában is.

Ha s az absztrakció konstruktora, akkor legyen

$$\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E) \ \rightsquigarrow_{\mathcal{K}} \ (\lambda(s\ p_1 p_2 \ldots p_n)\,.\,E)_{\mathcal{K}} \ \equiv \ \lambda^*(s\ p_1 p_2 \ldots p_n)\,.\,(E)_{\mathcal{K}}$$

Az összegkonstruktoros kifejezésre vonatkozó zárójeles absztrakció jelentését applikációval adjuk meg. A konstruktorok vizsgálatát egy új, *U_s* nevű kombinátorral hajtjuk végre, az *U* név az „uncurrying” szóból származik, *s* pedig az absztrakció kombinátora.

$$\lambda^*(s\ p_1 p_2 \ldots p_n)\,.\,E \equiv \mathit{U\_s}\ (\lambda^* p_1\,.\,\lambda^* p_2\,.\,\cdots\,.\,\lambda^* p_n\,.\,E)$$

Az *U_s* kombinátor gyenge redukciós szabályai legyenek a következők:

$$\mathit{U\_s}\ f\ (s\ F_1 F_2 \ldots F_n) \ \rightarrow_w \ f\ F_1 F_2 \ldots F_n$$
$$\mathit{U\_s}\ f\ (s'\ F_1 F_2 \ldots F_n) \rightarrow_w \mathit{fail}, \qquad \text{ha } s \neq s'$$

7.2.2. Példa. *(Összegkonstruktoros absztrakció)*

Tegyük fel, hogy egy definíció átalakításának egyik sora a kibővített λ-kalkulusban

$\lambda(\mathit{cons}\ x\ xs)\,.\,E$

Határozzuk meg ennek a *CL*-kifejezését. Ha $E \rightsquigarrow_{\mathcal{K}} E'$, akkor

$(\mathit{cons}\ x\ xs)\,.\,E \rightsquigarrow^{+}_{\mathcal{K}}$
$\lambda^{*}(\mathit{cons}\ x\ xs)\,.\,E' \equiv$
$\mathit{U\_cons}\ (\lambda^{*}x\,.\,\lambda^{*}xs\,.\,E')$

Applikáljuk a kifejezésre a $\mathit{cons}\ 1\ \mathit{nil}$, majd a $\mathit{branch}\ F\ G$ kifejezéseket. Legyen $F \rightsquigarrow_{\mathcal{K}} F'$ és $G \rightsquigarrow_{\mathcal{K}} G'$. Mivel

$\mathit{cons}\ 1\ \mathit{nil} \rightsquigarrow^{+}_{\mathcal{K}} \mathit{cons}\ 1\ \mathit{nil}$

és

$\mathit{branch}\ F\ G \rightsquigarrow^{+}_{\mathcal{K}} \mathit{branch}\ F'\ G'$,

a következő eredményt kapjuk:

$(\lambda^{*}(\mathit{cons}\ x\ xs)\,.\,E')\ (\mathit{cons}\ 1\ \mathit{nil}) \equiv$
$\mathit{U\_cons}\ (\lambda^{*}x\,.\,\lambda^{*}xs\,.\,E')\ (\mathit{cons}\ 1\ \mathit{nil}) \rightarrow_{w}$
$(\lambda^{*}x\,.\,\lambda^{*}xs\,.\,E')\ 1\ \mathit{nil}$

és

$(\lambda^{*}(\mathit{cons}\ x\ xs)\,.\,E')\ (\mathit{branch}\ F'\ G') \equiv$
$\mathit{U\_cons}\ (\lambda^{*}x\,.\,\lambda^{*}xs\,.\,E')\ (\mathit{branch}\ F'\ G') \rightarrow_{w}$
fail □

A kiterjesztett λ-kalkulusban az összegkonstruktoros minták mintaillesztésének λ-kifejezésében a mintaillesztést a $[\!]$ vastagvonal operátorral oldottuk meg. Vezessünk be a mintaillesztésre is egy új kombinátort, amelyet nevezzünk *Try*-nak, és a gyenge redukciós szabályok legyenek a következők:

$\mathit{Try}\ \mathit{fail}\ F$	$\rightarrow_{w}\ F$	
$\mathit{Try}\ E\ \ F$	$\rightarrow_{w}\ \mathit{Try}\ E'\ F,$	*ha* $E \rightarrow E'$
$\mathit{Try}\ E\ \ F$	$\rightarrow_{w}\ E,$	*egyébként*

Az $E\ [\!]\ F$ λ-kifejezésnek a kombinátor logikában a $\mathit{Try}(E)_{\mathcal{K}}(F)_{\mathcal{K}}$ kifejezést feleltessük meg.

$$E \,[\!]\, F \;\leadsto_{\mathcal{K}}\; (E \,[\!]\, F)_{\mathcal{K}} \;\equiv\; \mathit{Try}\,(E)_{\mathcal{K}}\,(F)_{\mathcal{K}}$$

Ezzel a kibővített λ-kalkulusban levő vastagvonal operátorokat tartalmazó kifejezéseket a kombinátor logikában applikációkra alakítottuk át.

Most nézzük meg, hogy a bevezetett *Try* kombinátor felhasználásával hogyan lehet az összegkonstruktoros mintákat tartalmazó λ-kifejezéseket *CL*-kifejezésekké átalakítani.

Először nézzünk meg egy egymintás és kétsoros függvénydefiníciót. Tegyük fel, hogy a definícióból a kibővített λ-kalkulusban a következő kifejezést kaptuk:

$$\begin{aligned} f = \lambda x.\ (\ & ((\lambda p_1 . E_1)\ x) \\ [\!]\ & ((\lambda p_2 . E_2)\ x) \\ [\!]\ & \mathit{ERROR} \\ &) \end{aligned}$$

Ha $E_1 \leadsto_{\mathcal{K}} E_1', E_2 \leadsto_{\mathcal{K}} E_2'$ és $p_1 \leadsto_{\mathcal{K}} p_1', p_2 \leadsto_{\mathcal{K}} p_2'$, ebből a kombinátor logikában a következő kifejezést kapjuk:

$$f \leadsto^{+}_{\mathcal{K}}\ \lambda^* x.\ \mathit{Try}\ ((\lambda^* p_1' . E_1')x)\ \ (\mathit{Try}\,((\lambda^* p_1' . E_2')x)\ \mathit{ERROR})$$

Az egyszerűség kedvéért jelöljük a $((\lambda^* p_1' . E_1')x)$ kifejezést E_1''-vel, a $((\lambda^* p_1' . E_2')x)$ kifejezést E_2''-vel, így az f átalakítását a

$$\lambda^* x.\ \mathit{Try}\ E_1''\ \ (\mathit{Try}\,E_2''\ \mathit{ERROR})$$

kifejezéssel írhatjuk le. Mivel a *Try* konstans, alkalmazhatjuk a λ^* zárójeles absztrakció

$$\lambda^* x.\, E\,F\,G \equiv \mathit{S'}E\,(\lambda^* x.\, F)(\lambda^* x.\, G)$$

átalakítási szabályát, így az $f_{\mathcal{K}}$-ra az

$$\mathit{S'}\ \mathit{Try}\ (\lambda^* x.\, E_1'')\ (\lambda^* x.\, (\mathit{Try}\,E_2''\ \mathit{ERROR}))$$

kifejezést kapjuk. A jobb oldali kifejezésre ismét alkalmazva a zárójeles absztrakció szabályát, az $f_{\mathcal{K}}$ kifejezés alakja

$$\mathit{S'}\ \mathit{Try}\ (\lambda^* x.\, E_1'')\ (\mathit{S'}\ \mathit{Try}\ (\lambda^* x.\, E_2'')\ \mathit{ERROR}))$$

Ezt a kifejezést tovább tudjuk egyszerűsíteni, írjuk ki részletesen az E_1'' kifejezést:

$$\lambda^* x.\, E_1'' \equiv\ \lambda^* x.\, (\lambda^* p_1' . E_1')x,$$

amire ha E_1' nem tartalmaz szabad x változót, alkalmazható a zárójeles absztrakció

$\lambda^* x\,.\,E\,x \equiv E$

szabálya, így ebből a $\lambda^* p_1\,.\,E_1' \equiv \lambda^* p_1\,.\,(E_1)_{\mathcal{K}}$ kifejezést kapjuk. Hasonló módszerrel a $\lambda^* x\,.\,E_2''$ kifejezés a $\lambda^* p_2\,.\,(E_2)_{\mathcal{K}}$-ra egyszerűsíthető. Tehát az f kifejezés alakja a kombinátor logikában

$$
\begin{array}{l}
\lambda x\,.\,(\ ((\lambda p_1\,.\,E_1)\ x) \\
\qquad [\!] \ ((\lambda p_2\,.\,E_2)\ x) \\
\qquad [\!] \ \mathit{ERROR} \\
\qquad) \\
\rightsquigarrow_{\mathcal{K}} \\
\mathit{S'\ Try}\ (\lambda^* p_1\,.\,(E_1)_{\mathcal{K}}) \\
\qquad (\mathit{S'\ Try}\ (\lambda^* p_2\,.\,(E_2)_{\mathcal{K}}) \\
\qquad\qquad \mathit{ERROR}\ )
\end{array}
$$

Ebből már következtethetünk a többmintás, több soros, összegtípusú konstruktorokat tartalmazó definíció átalakítási szabályára is.

$$
\begin{array}{l}
\lambda x_1 \ldots x_n\,.\,(\quad ((\lambda p_{1,1}\,.\,\cdots\,.\,\lambda p_{1,n}\,.\,E_1) x_1 \ldots x_n) \\
\qquad\qquad [\!] \quad ((\lambda p_{2,1}\,.\,\cdots\,.\,\lambda p_{2,n}\,.\,E_2) x_1 \ldots x_n) \\
\qquad\qquad \ldots \\
\qquad\qquad [\!] \quad ((\lambda p_{m,1}\,.\,\cdots\,.\,\lambda p_{m,n}\,.\,E_m) x_1 \ldots x_n) \\
\qquad\qquad [\!] \quad \mathit{ERROR} \\
\qquad\qquad) \\
\rightsquigarrow_{\mathcal{K}} \\
\mathit{S'\ Try}\ (\lambda^* p_{1,1}\,.\,\cdots\,.\,\lambda^* p_{1,n}\,.\,(E_1)_{\mathcal{K}}) \\
\qquad (\mathit{S'\ Try}\ (\lambda^* p_{2,1}\,.\,\cdots\,.\,\lambda^* p_{2,n}\,.\,(E_2)_{\mathcal{K}}) \\
\qquad\qquad \ldots \\
\qquad\qquad\qquad (\mathit{S'\ Try}\ (\lambda^* p_{m,1}\,.\,\cdots\,.\,\lambda^* p_{m,n}\,.\,(E_m)_{\mathcal{K}}) \\
\qquad\qquad\qquad\qquad \mathit{ERROR}\ )\ldots)
\end{array}
$$

Megjegyezzük, hogy ha a zárójeles absztrakciók végrehajtásánál egy *S'*-s szabálytól különböző szabály alkalmazásának feltétele is teljesül, akkor a fenti átalakításban az *S'*-s szabály helyett ez a zárójeles absztrakció is alkalmazható.

7.2.3. Példa. *(Egymintás, kétsoros definíció)*

A 3.5.4. példában adtuk meg a *flip* függvény

$$\mathit{flip}\ 0 = 1$$
$$\mathit{flip}\ 1 = 0$$

definícióját, és itt szerepelt a függvény

$$\lambda x\,.\ (\ ((\lambda 0\,.\,1)\ x)$$
$$\quad [\!]\ ((\lambda 1\,.\,0)\ x)$$
$$\quad [\!]\ \mathit{ERROR}$$
$$\quad)$$

λ-kifejezése is. A *flip* függvénydefiníció *CL*-kifejezése:

$$\mathit{S'\ Try}\ (\lambda^* 0\,.\,1)\ (\mathit{S'\ Try}\ (\lambda^* 1\,.\,0)\ (\mathit{ERROR})\ )$$

A konstansos absztrakciókat is beírva, az

$$\mathit{S'\ Try}\ (\mathit{Match}\ 0\ 1)$$
$$\quad (\mathit{S'\ Try}\ (\mathit{Match}\ 1\ 0)$$
$$\quad\quad (\mathit{ERROR})\ )$$

kifejezést kapjuk.

Most határozzuk meg a *flip* 1 kifejezés értékét.

$$(\mathit{S'\ Try}\ (\mathit{Match}\ 0\ 1)\ (\mathit{S'\ Try}\ (\mathit{Match}\ 1\ 0)\ (\mathit{ERROR})\ )\ )\ 1 \rightarrow_w$$
$$\mathit{Try}\ (\underline{(\mathit{Match}\ 0\ 1)\ 1})\ ((\mathit{S'\ Try}\ (\mathit{Match}\ 1\ 0)\ (\mathit{ERROR})\ )\ 1) \rightarrow_w$$
$$\mathit{Try\ fail}\ ((\mathit{S'\ Try}\ (\mathit{Match}\ 1\ 0)\ (\mathit{ERROR})\ )\ 1) \rightarrow_w$$
$$\mathit{S'\ Try}\ (\mathit{Match}\ 1\ 0)\ (\mathit{ERROR})\ )\ 1 \rightarrow_w$$
$$\mathit{Try}\ ((\mathit{Match}\ 1\ 0)\ 1)\ ((\mathit{ERROR})\ 1) \rightarrow_w$$
$$0 \qquad \square$$

7.2.4. Példa. *(Mintaillesztés összegkonstruktoros absztrakciókkal)*
A 3.5.5. példában szereplő

$$\mathit{reflect}\ (\mathit{leaf}\ n) \quad = \mathit{leaf}\ n$$
$$\mathit{reflect}\ (\mathit{branch}\ t_1\ t_2) = \mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1)$$

függvény λ-kifejezését a 3.7.5. példában adtuk meg:

$$\mathit{reflect} = \lambda x\,.\ (\ ((\lambda(\mathit{leaf}\ n)\,.\,\mathit{leaf}\ n)\ x)$$
$$\quad [\!]\ ((\lambda(\mathit{branch}\ t_1\ t_2)\,.\,\mathit{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))\ x)$$
$$\quad [\!]\ \mathit{ERROR}$$
$$\quad)$$

Határozzuk meg a *reflect* függvény *CL*-kifejezését. Először alakítsuk át az első mintához tartozó λ-kifejezést, és használjuk a mintákra megadott átalakításokat is.

$\lambda^*(\mathsf{leaf}\ n)\,.\,(\mathsf{leaf}\ n)_{\mathcal{K}} \equiv$
$\lambda^*(\mathsf{leaf}\ n)\,.\,\mathsf{leaf}\ n \equiv$
$\mathsf{U\_leaf}\ (\lambda^* n\,.\,\mathsf{leaf}\ n) \equiv$
$\mathsf{U\_leaf}\ \mathsf{leaf}$

A második minta λ-kifejezésének az átalakítása:

$(\lambda^*(\mathsf{branch}\ t_1\ t_2)\,.\,(\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1))_{\mathcal{K}} \equiv$
$\lambda^*(\mathsf{branch}\ t_1\ t_2)\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1) \equiv$
$\mathsf{U\_branch}\ (\lambda^* t_1\ t_2\,.\,\mathsf{branch}\ (\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1)) \equiv$
$\mathsf{U\_branch}\ (\lambda^* t_1\,.\,(\mathsf{C'\ branch}\ (\lambda^* t_2\,.\,\mathit{reflect}\ t_2)\ (\mathit{reflect}\ t_1)) \equiv$
$\mathsf{U\_branch}\ (\lambda^* t_1\,.\,(\mathsf{C'\ branch}\ \mathit{reflect})\ (\mathit{reflect}\ t_1)) \equiv$
$\mathsf{U\_branch}\ (\mathsf{B}\ (\mathsf{C'\ branch}\ \mathit{reflect})\ (\lambda^* t_1\,.\,\mathit{reflect}\ t_1)) \equiv$
$\mathsf{U\_branch}\ (\mathsf{B}\ (\mathsf{C'\ branch}\ \mathit{reflect})\ \mathit{reflect})$

így a mintákat összefogva az $\mathsf{S'}$ és Try kombinátorokkal:

$\mathit{reflect} =$
$\mathsf{S'\ Try}\ (\mathsf{U\_leaf}\ \mathsf{leaf})$
$\quad(\mathsf{S'\ Try}\ (\mathsf{U\_branch}\ (\mathsf{B}\ (\mathsf{C'\ branch}\ \mathit{reflect})\ \mathit{reflect}))$
$\quad\quad(\mathsf{ERROR})\ )$

és ezzel a *reflect* függvény *CL*-kifejezését meghatároztuk. □

7.2.5. Példa. *(A reflect* (leaf E) *kifejezés)*

Az előző példában meghatároztuk a *reflect* kifejezését, most határozzuk meg a *reflect* (leaf E) kifejezés értékét.

$\mathit{reflect}\ (\mathsf{leaf}\ E) \equiv$
$(\mathsf{S'\ Try}\ (\mathsf{U\_leaf}\ \mathsf{leaf})$
$\quad(\mathsf{S'\ Try}\ (\mathsf{U\_branch}\ (\mathsf{B}\ (\mathsf{C'\ branch}\ \mathit{reflect})\ \mathit{reflect}))(\mathsf{ERROR})\ )\ )\ (\mathsf{leaf}\ E) \rightarrow_w$
$\mathsf{Try}\ (\underline{\mathsf{U\_leaf}}\ \underline{\mathsf{leaf}}\ (\underline{\mathsf{leaf}\ E}))$
$\quad(\mathsf{S'\ Try}\ (\mathsf{U\_branch}\ (\mathsf{B}\ (\mathsf{C'\ branch}\ \mathit{reflect})\ \mathit{reflect}))(\mathsf{ERROR})\ (\mathsf{leaf}\ E)) \rightarrow_w$
$\mathsf{Try}\ (\mathsf{leaf}\ E)$
$\quad(\mathsf{Try}\ (\mathsf{U\_branch}\ (\mathsf{B}\ (\mathsf{C'\ branch}\ \mathit{reflect})\ \mathit{reflect}))(\mathsf{ERROR})\ )\ (\mathsf{leaf}\ E)) \rightarrow_w$
$\mathsf{leaf}\ E$ □

7.2.6. Példa. *(A reflect* (*branch*(*leaf E*)(*leaf F*)) *kifejezés)*
Határozzuk meg a *reflect* (*branch*(*leaf E*)(*leaf F*)) kifejezés értékét is.

reflect (*branch*(*leaf E*)(*leaf F*))) ≡
S' Try (*U_leaf leaf*)
(*S' Try* (*U_branch* (*B* (*C' branch reflect*) *reflect*))(*ERROR*))
(*branch*(*leaf E*)(*leaf F*)) $\rightarrow_w$
Try (*U_leaf leaf* (*branch*(*leaf E*)(*leaf F*)))
(*S' Try* (*U_branch* (*B* (*C' branch reflect*) *reflect*))(*ERROR*)
(*branch*(*leaf E*)(*leaf F*))) $\rightarrow_w$
Try fail
(*S' Try* (*U_branch* (*B* (*C' branch reflect*) *reflect*))(*ERROR*))
(*branch*(*leaf E*)(*leaf F*))) $\rightarrow_w$
S' Try (*U_branch* (*B* (*C' branch reflect*) *reflect*))(*ERROR*)
(*branch*(*leaf E*)(*leaf F*)) $\rightarrow_w$
Try (*U_branch* (*B* (*C' branch reflect*) *reflect*)) (*branch*(*leaf E*)(*leaf F*)))
((*ERROR*) (*branch*(*leaf E*)(*leaf F*)) $\rightarrow_w$
Try ((*B* (*C' branch reflect*) *reflect*) (*leaf E*) (*leaf F*))
((*ERROR*) (*branch*(*leaf E*)(*leaf F*)) $\rightarrow_w$
Try ((*C' branch reflect*) (*reflect* (*leaf E*)) (*leaf F*))
((*ERROR*) (*branch*(*leaf E*)(*leaf F*)) $\rightarrow_w$
Try (*branch* (*reflect leaf F*)) (*reflect* (*leaf E*)))
((*ERROR*) (*branch*(*leaf E*)(*leaf F*)) $\rightarrow_w$
branch (*reflect leaf F*)) (*reflect* (*leaf E*)) □

7.2.3. Szorzatkonstruktoros absztrakciók

A szorzatkonstruktoros applikációk értelmezését a 3.7.3. szakaszban vizsgáltuk. Az összegkonstruktoros absztrakcióhoz hasonlóan legyen

$$\lambda(t\ p_1\ p_2 \ldots p_n)\,.\,E \quad \rightsquigarrow_{\mathcal{K}} \quad (\lambda(t\ p_1\ p_2 \ldots p_n)\,.\,E)_{\mathcal{K}} \quad \equiv \lambda^*(t\ p_1\ p_2 \ldots p_n)\,.\,(E)_{\mathcal{K}}$$

ahol *t* a szorzatkonstruktort jelöli. Most is egy olyan megoldást keresünk, hogy a konstruktorok összehasonlítását az absztrakció argumentumában levő konstruktor paramétereinek, azaz mintáinak feldolgozásáig elhalasszuk.

A szorzatkonstruktort tartalmazó kifejezés zárójeles absztrakcióját egy új, *V_t*

kombinátorral adjuk meg:

$$\lambda^*(t\ p_1\ p_2 \ldots p_n)\,.\,E \quad \equiv \quad V\_t\ (\lambda^* p_1\,.\,\lambda^* p_2\,.\,\cdots\,.\,\lambda^* p_n\,.\,E)$$

ahol a V_t kombinátor gyenge redukciós szabálya a következő:

$$V\_t\ f\ F \quad \rightarrow_w \quad f\ (VSEL\_t\_1\ F)\ (VSEL\_t\_2\ F) \ldots (VSEL\_t\_n\ F)$$

A konstruktorok összehasonlítását a $VSEL\_...\_...$ operátor végzi, ennek az operátornak a redukciós szabálya megfelel a kibővített λ-kalkulusban megadott $SEL\_...\_...$ operátor szabályának.

$$\begin{array}{ll} VSEL\_t\_i\ (t\ F_1 F_2 \ldots F_n) \ \rightarrow_w\ F_i & (1 \leq i \leq n) \\ VSEL\_t\_i\ (t'\ F_1 F_2 \ldots F_n) \ \rightarrow_w\ fail, & ha\ t \neq t' \end{array}$$

Az átalakítások megadott szabályai szerint tehát a szorzatkonstruktorok összehasonlítására csak akkor kerül sor, ha az F paraméter adataira szükség van.

Mint a 3. fejezetben láttuk, egy egymintás szorzatkonstruktorral megadott definíciónak a kibővített λ-kalkulusban az

$$\begin{array}{l} f = \lambda x\,.\,(\ ((\lambda p\,.\,E)\ x) \\ \qquad [\!]\ ERROR \\ \qquad) \end{array}$$

kifejezést feleltettük meg. Ez a kifejezés az előző pontban bevezetett Try kombinátorral könnyen átalakítható CL-kifejezésre:

$$f \rightsquigarrow^+_{\mathcal{K}}\ \lambda^* x\,.\,Try\ ((\lambda^* p\,.\,E)x)\ \ ERROR$$

Mivel a Try és az $ERROR$ konstans, alkalmazhatjuk a zárójeles absztrakció

$$\lambda^* x\,.\,E\,F\,G \equiv C'E\,(\lambda^* x\,.\,F)\,G$$

szabályát, így a következő kifejezést kapjuk:

$$C'\ Try\ (\lambda^* x\,.\,((\lambda^* p\,.\,E)x))\ ERROR$$

A kifejezésben lévő $\lambda^* x$ zárójeles absztrakcióra alkalmazva a

$$\lambda^* x\,.\,E\,x \equiv E$$

szabályt, a kifejezés még egyszerűbb lesz. Tehát a kifejezés alakja a kombinátor

logikában

$$
\begin{array}{l}
f = \lambda x\,.\ (\ ((\lambda p\,.\,E)\ x) \\
\qquad\quad [\!]\ \mathit{ERROR} \\
\qquad\quad) \\
\rightsquigarrow_{\mathcal{K}} \\
\mathit{C'\ Try}\ (\lambda^{*} p\,.\,E)\ \mathit{ERROR}
\end{array}
$$

Ennek alapján már megadhatjuk a többmintás, szorzatkonstruktorokat tartalmazó definíció átalakítási szabályát is:

$$
\begin{array}{l}
\lambda x_1 \ldots x_n\,.\ (\ ((\lambda p_1\,.\,\cdots\,.\,\lambda p_n\,.\,E) x_1 \ldots x_n) \\
\qquad\quad [\!]\ \mathit{ERROR} \\
\qquad\quad) \\
\rightsquigarrow_{\mathcal{K}} \\
\mathit{C'\ Try}\ (\lambda^{*} p_1\,.\,\cdots\,.\,\lambda^{*} p_n\,.\,(E)_{\mathcal{K}})\ \mathit{ERROR}
\end{array}
$$

7.2.7. Példa. *(Szorzatkonstruktoros absztrakció)*

A 3.7.8. példában láttuk, hogy az

$$\mathit{add\_pair}\ (\mathit{pair}\ x\ y) = +\ x\ y$$

definíció λ-kifejezése

$$
\begin{array}{l}
\mathit{add\_pair} = \lambda z\,.\ (\ ((\lambda(\mathit{pair}\ x\ y)\,.\ +\ x\ y)\ z) \\
\qquad\qquad [\!]\ \mathit{ERROR} \\
\qquad\qquad)
\end{array}
$$

Alakítsuk át ezt a kifejezést *CL*-kifejezésre.

$$
\begin{array}{l}
\mathit{add\_pair} \rightsquigarrow_{\mathcal{K}} \\
(\lambda z\,.\ (\ ((\lambda(\mathit{pair}\ x\ y)\,.\ +\ x\ y)\ z) \\
\qquad [\!]\ \mathit{ERROR} \\
\qquad)\ )_{\mathcal{K}} \\
\equiv \\
\mathit{C'\ Try}\ (\lambda^{*}(\mathit{pair}\ x\ y)\,.\,(+\ x\ y)_{\mathcal{K}})\ \mathit{ERROR}
\end{array}
$$

Először a kifejezésben levő zárójeles absztrakció törzsét alakítva, majd az absztrakciókat elvégezve:

$\lambda^*(\mathit{pair}\ x\ y)\,.\,(+\ x\ y)_{\mathcal{K}} \equiv$
$\lambda^*(\mathit{pair}\ x\ y)\,.\,+\ x\ y \equiv$
$\mathit{V\_pair}\ (\lambda^* x\,.\,\lambda^* y\,.\,+\ x\ y)) \equiv$
$\mathit{V\_pair}\ (\lambda^* x\,.\,+\ x) \equiv$
$\mathit{V\_pair}\ +$

Így az *add_pair* kifejezésre a

$\mathit{C'\ Try}\ (\mathit{V\_pair}\ +)\ \mathit{ERROR}$

kifejezést kaptuk. □

7.2.8. Példa. *(Az add_pair* (*pair* 3 4) *kifejezés)*
Az előző példában határozzuk meg az *add_pair* definíciójának *CL*-kifejezését, most számoljuk ki az *add_pair* (*pair* 3 4) kifejezés értékét.

add_pair (*pair* 3 4) ≡
$\mathit{C'\ Try}\ (\mathit{V\_pair}\ +)\ \mathit{ERROR}\ (\mathit{pair}\ 3\ 4) \rightarrow_w$
$\mathit{Try}\ (\underline{\mathit{V\_pair}}\ \underline{+}\ (\underline{\mathit{pair}\ 3\ 4}))\ \mathit{ERROR} \rightarrow_w$
$\mathit{Try}\ (+\ (\underline{\mathit{VSEL\_pair\_1}}\ \underline{(\mathit{pair}\ 3\ 4)})\ (\underline{\mathit{VSEL\_pair\_2}}\ \underline{(\mathit{pair}\ 3\ 4)})\ \mathit{ERROR} \rightarrow_w^+$
$\mathit{Try}\ (+\ 3\ 4)\ \mathit{ERROR} \rightarrow_\delta$
$\mathit{Try}\ 7\ \ \mathit{ERROR} \rightarrow_w$
7 □

7.2.9. Példa. *(A zero_pair kifejezés)*
A 3.7.7. példában láttuk a

zero_pair (*pair* *x y*) = 0

függvénydefiníciót, és itt mutattuk meg, hogy a *zero_pair* függvénnyel képzett függvényapplikációban a függvény argumentumának kiértékelésére nem kerül sor. Most megmutatjuk, hogy ez a tulajdonság teljesül akkor is, ha a függvényapplikációt a kombinátor logika kifejezéseivel végezzük el.

Először határozzuk meg a *zero_pair CL*-kifejezését.

$\mathit{zero\_pair} \rightsquigarrow_{\mathcal{K}}$
$(\lambda z\,.\,(\ ((\lambda(\mathit{pair}\ x\ y)\,.\,0)\ z)$
$\quad\ |\ \mathit{ERROR}$
$\quad\ )\ )_{\mathcal{K}}$
≡
$\mathit{C'\ Try}\ (\lambda^*(\mathit{pair}\ x\ y)\,.\,0)\ \ \mathit{ERROR}$

Hajtsuk végre a zárójeles absztrakciót:

$\lambda^*(\mathit{pair}\ x\ y)\,.\,0 \equiv$
$\mathit{V\_pair}\ (\lambda^* x\,.\,(\underline{\lambda^* y\,.\,0})) \equiv$
$\mathit{V\_pair}\ (\underline{\lambda^* x\,.\,(\mathit{K}\ 0)}) \equiv$
$\mathit{V\_pair}\ (\mathit{K}\ (\mathit{K}\ 0))$

Tehát a *zero_pair* *CL*-kifejezése:

$\mathit{C'\ Try}\ (\mathit{V\_pair}\ (\mathit{K}\ (\mathit{K}\ 0)))\ \mathit{ERROR}$

Határozzuk meg a *zero_pair* (*pair* 2 3) kifejezés értékét.

zero_pair (*pair* 2 3) ≡
$\mathit{C'\ Try}\ (\mathit{V\_pair}\ (\mathit{K}\ (\mathit{K}\ 0)))\ \mathit{ERROR}\ (\mathit{pair}\ 2\ 3) \rightarrow_w$
$\mathit{Try}\ (\underline{\mathit{V\_pair}}\ \underline{(\mathit{K}\ (\mathit{K}\ 0))}\ \underline{(\mathit{pair}\ 2\ 3)})\ \mathit{ERROR} \rightarrow_w$
$\mathit{Try}\ (\underline{\mathsf{K}}\ \underline{(\mathit{K}\ 0)}\ \underline{(\mathit{VSEL\_pair\_}1\ (\mathit{pair}\ 3\ 4))}\ (\mathit{VSEL\_pair\_}2\ (\mathit{pair}\ 3\ 4)))\ \mathit{ERROR} \rightarrow_w$
$\mathit{Try}\ (\underline{\mathsf{K}}\ \underline{0}\ \underline{(\mathit{VSEL\_pair\_}2\ (\mathit{pair}\ 3\ 4))})\ \mathit{ERROR} \rightarrow_w$
$\mathit{Try}\ \ 0\ \ \mathit{ERROR} \rightarrow_w$
0

A levezetésből látható, hogy a *zero_pair* függvény *pair* 2 3 argumentumának kiértékelésére valóban nem kerül sor. □

Irodalomjegyzék

[1] Appel, Andrew W.: *Modern Compiler Implementation in C*. Cambridge University Press, 2004.

[2] Csörnyei Zoltán: *Lambda-kalkulus, a funkcionális programozás alapjai*. Typotex, Budapest, 2007.

[3] Csörnyei Zoltán: *Programozási nyelvek típusrendszerei*. kézirat, 2010.
URL "`http://people.inf.elte.hu/csz`".

[4] Diller, Antoni: *Compiling Functional Languages*. John Wiley & Sons, 1988.

[5] Peyton Jones, Simon L.: *The implementation of functional languages*. Prentice Hall, 1987.

[6] Peyton Jones, Simon L. – Lester, David R.: *Implementing Functional Languages: a tutorial*. Prentice Hall, 2000.

[7] Plasmeijer, Marinus J. – van Eekelen, Marko C. J. D.: *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[8] Terese: *Term Rewriting Systems*. Cambridge University Press, 2003.

Tárgy- és névmutató