EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKA KAR

Programozási Nyelvek és Fordítóprogramok Tanszék

A JoCaml nyelv, a nyelv modellje, mobil kód JoCaml-ben

diplomamunka

Készítette: Mészáros Mónika

(Programtervező matematikus szak, nappali tagozat)

Témavezető: Horváth Zoltán

Budapest, 2004. június 11.

Készült az OTKA T037742 számú "Elosztott funkcionális programok helyessége" című pályázat keretében.

Tartalomjegyzék

1. BEVEZETÉS ... 5

1.1. A PROGRAMOZÁSI NYELVEK OSZTÁLYOZÁSA ... 5
1.2. A FUNKCIONÁLIS PROGRAMOZÁSI NYELVEK ... 7
1.2.1. A funkcionális programok végrehajtása ... 7
1.2.2. A funkcionális programozási nyelvek osztályozása ... 8
1.2.3. A tisztán funkcionális programok jellemzői ... 8
1.3. A JOCAML NYELV ... 9
1.3.1. A JoCaml történeti áttekintése, elődei ... 9
1.3.2. A JoCaml nyelv tulajdonságai ... 10
1.3.3. Egy JoCaml program felépítése ... 11

2. A JOCAML NYELV FUNKCIONÁLIS NYELVI ELEMEI ... 13

2.1. LEXIKÁLIS ELEMEK ... 13
2.2. BEÉPÍTETT ELEMI ADATTÍPUSOK ... 13
2.3. AZONOSÍTÓK ÉS KÖTÉS ... 14
2.3.1. Hatókör és láthatóság ... 15
2.4. FÜGGVÉNYEK ... 15
2.4.1. Egyszerű függvények ... 15
2.4.2. Magasabbrendű függvények ... 16
2.4.3. Többváltozós függvények ... 16
2.4.4. Rekurzív *függvénydefiníció ... 17*
2.5. EGYSZERŰ KIFEJEZÉSEK ... 17
2.5.1. Zárójelezett kifejezés ... 17
2.5.2. Elágazás ... 18
2.6. MINTAILLESZTÉS ... 18
2.7. TÍPUSRENDSZER ... 19
2.7.1. Polimorfizmus ... 19
2.8. TÍPUSKONSTRUKCIÓK (ÖSSZETETT TÍPUS) ... 20
2.8.1. Rendezett n-es ... 20
2.8.2. Lista ... 21
2.8.3. Rekord (direktszorzat) ... 21
2.8.4. Algebrai adattípus ... 22
2.8.5. Típus-szinonimák ... 23
2.9. KIVÉTELKEZELÉS (SZEKVENCIÁLIS PROGRAM ESETE) ... 24
2.10. ABSZTRAKT ADATTÍPUS MEGVALÓSÍTÁSA JOCAML-BEN: FORDÍTÁSI EGYSÉGEK ... 26
2.10.1. Interfész ... 26
2.10.2. Implementáció ... 27
2.10.3. Példa: halmaz megvalósítása ... 27
2.10.4. Az absztrakt adattípus használata más programból ... 28
2.11. KÉSLELTETETT KIÉRTÉKELÉS ... 29

3. A JOCAML NYELV IMPERATÍV NYELVI ELEMEI ... 30

3.1. MÓDOSÍTHATÓ TÍPUSOK (TÍPUSKONSTRUKCIÓK) ... 30
3.1.1. Rekord (direktszorzat) ... 30
3.1.2. Referencia ... 30
3.1.3. Tömb ... 31
3.2. SZEKVENCIA ... 31
3.3. CIKLUS ... 32
3.3.1. Elöltesztelős ciklus ... 32
3.3.2. Léptetős ciklus ... 32
3.4. INPUT ÉS OUTPUT ... 33
3.4.1. Fájlműveletek ... 33
3.4.2. Formázott szöveg kiíratása: a Printf modul ... 35
3.4.3. Egyéb kiíró kifejezések ... 35

4. A JOCAML NYELV OBJEKTUM-ORIENTÁLT NYELVI ELEMEI ... 36

4.1. OSZTÁLY ... 36

4.1.1. Osztály deklarációja ... *36*
4.1.2. Konstruktor ... *37*
4.1.3. Destruktor ... *37*
4.1.4. Információ elrejtés (láthatóság) ... *37*
4.1.5. Példa ... *38*
4.2. OBJEKTUM ... 38
4.2.1. Objektumok létrehozása ... *38*
4.2.2. Metódusok meghívása ... *38*
4.2.3. Az objektum típusa ... *38*
4.2.4. Hivatkozás önmagára („self") ... *39*
4.2.5. Objektumok másolása ... *40*
4.3. ÖRÖKLŐDÉS: ... 40
4.3.1. Hivatkozás az ősosztályra ... *41*
4.3.2. Példa ... *41*
4.3.3. Többszörös öröklődés ... *42*
4.3.4. Példa a többszörös öröklődésre ... *42*
4.3.5. Lezárt osztályok ... *42*
4.4. PARAMÉTEREZETT (POLIMORF) OSZTÁLYOK ... 43
4.4.1. Típus-megszorítás (constraint) ... *43*
4.5. VIRTUÁLIS (ABSZTRAKT) OSZTÁLYOK ... 44
4.6. OBJEKTUMOK KÖZÖTTI ALTÍPUS-RELÁCIÓ ... 45
4.6.1. Altípus definíciója ... *45*
4.6.2. Altípus és leszármazott kapcsolata ... *46*
4.7. KÖLCSÖNÖSEN REKURZÍV OSZTÁLYOK ... 47
4.8. OSZTÁLYOK HASZNÁLATA ÉRTÉKADÁS NÉLKÜL ... 47

5. A JOCAML NYELV PÁRHUZAMOS PROGRAMOZÁST TÁMOGATÓ NYELVI ELEMEI ... 48

5.1. A KIFEJEZÉSEK ÉS FOLYAMATOK ÖSSZEHASONLÍTÁSA ... 48
5.2. CSATORNÁK ... 48
5.2.1. Aszinkron csatorna ... *49*
5.2.2. Szinkron csatorna ... *50*
5.2.3. Csatornák használata ... *51*
5.3. FOLYAMATOK HASZNÁLATA A PROGRAMBAN ... 52
5.3.1. A párhuzamos kompozíció operátor („|") ... *52*
5.4. A JOIN-MINTÁK ... 52
5.4.1. Alternatív minták ... *54*
5.5. A CSATORNÁK TULAJDONSÁGAI ... 54
5.6. KIVÉTELKEZELÉS PÁRHUZAMOS PROGRAM ESETÉBEN ... 55

6. A JOCAML NYELV ELOSZTOTT PROGRAMOZÁST TÁMOGATÓ NYELVI ELEMEI ... 58

6.1. ERŐFORRÁSOK MEGOSZTÁSA NÉV-SZERVER SEGÍTSÉGÉVEL ... 58
6.1.1. A név-szerver használata ... *60*
6.1.2. Példa ... *61*
6.1.3. Példa – több program egyidejű futtatása ... *61*
6.1.4. Távoli erőforrás-hozzáférés másik módja: erőforrás elküldése üzenetben ... *62*
6.2. ABSZTRAKT HELY ÉS MOBILITÁS ... 63
6.2.1. Az absztrakt hely (location) ... *63*
6.2.2. Mobilitás ... *64*
6.2.3. Mobil objektum ... *66*
6.3. MEGHIBÁSODÁS ÉS AZ EZZEL KAPCSOLATOS PROBLÉMÁK MEGOLDÁSA ... 67
6.3.1. Absztrakt hely leállítása ... *67*
6.3.2. Absztrakt hely leállásának felismerése ... *68*
6.3.3. Példák ... *68*

7. JOIN-KALKULUS KIFEJEZŐEREJE ... 70

7.1. FUNKCIONÁLIS STÍLUSÚ PROGRAMOZÁS ... 70
7.2. OBJEKTUM ORIENTÁLT STÍLUSÚ PROGRAMOZÁS ... 70
7.2.1. Példa: számláló ... *70*
7.2.2. Általánosítás: osztály megvalósítása csatornákkal és join-mintákkal ... *71*
7.2.3. Példa: referencia ... *73*

7.3. SZINKRONIZÁCIÓS ESZKÖZÖK MEGVALÓSÍTÁSA 74
7.3.1. Kölcsönös kizárás 74
7.3.2. Zárolás (mutex) 74
7.3.3. Sorompó (barrier) 74
7.4. CIKLUSOK MEGVALÓSÍTÁSA 75
7.4.1. Egyszerű ciklusok 75
7.4.2. Elosztott ciklusok 76
7.5. APPLET 78

8. ELOSZTOTT PROGRAMOZÁS JELLEMZŐI JOCAML-BEN 80

8.1. ÁTLÁTSZÓSÁGI TULAJDONSÁGOK JOCAML-BEN 80
8.1.1. Hozzáférési átlátszóság 80
8.1.2. Elhelyezkedési átlátszóság 81
8.1.3. Vándorlási átlátszóság 82
8.1.4. Áthelyezési átlátszóság 85
8.1.5. Replikációs átlátszóság 85
8.1.6. Konkurencia átlátszóság 86
8.1.7. Meghibásodási átlátszóság 86
8.2. SKÁLÁZHATÓSÁG 87
8.3. HETEROGENITÁS 87
8.4. BIZTONSÁGOS MOBIL-KÓD 87

9. IMPLEMENTÁCIÓ 88

9.1. A SZÁLAK KEZELÉSE 88
9.2. A JOIN-KALKULUS ELMÉLETI HÁTTERE 89
9.2.1. Szintaxis 89
9.2.2. Szemantika 89
9.2.3. A megvalósítás 90
9.3. A JOIN-MINTÁK FORDÍTÁSA 90
9.3.1. Mintaillesztés a join-definícióban 91
9.3.2. Determinisztikus automata 92
9.3.3. Automata és szemantika 93
9.3.4. Futásidejű definíciók 94

10. ÖSSZEHASONLÍTÁS A π-KALKULUSSAL 96

10.1.1. Kétirányú csatornák megvalósítása 96

11. JAVASLAT ÚJ NYELVI ELEMEK BEVEZETÉSÉRE 97

11.1. TÖBB, EGYMÁSSAL KAPCSOLATBAN ÁLLÓ NÉV-SZERVER 97
11.1.1. A JoCaml név-szerverének hiányossága 97
11.1.2. A megvalósítás terve 97
11.2. REPLIKÁCIÓS ÁTLÁTSZÓSÁG MEGVALÓSÍTÁSA 99
11.2.1. A replikációs átlátszóság modellezése 99
11.3. KIFINOMULTABB MEGHIBÁSODÁSI ÁTLÁTSZÓSÁG 101

12. FÜGGELÉK - TESZTPROGRAMOK 103

12.1. PARAMÉTERÁTADÁS 103
12.2. LOKÁLISAN DEKLARÁLT AZONOSÍTÓ ÉLETTARTAMA 103
12.3. SZINKRON CSATORNÁRA TÖRTÉNŐ ÜZENETKÜLDÉSKOR A HÍVÓ FELFÜGGESZTÉSE 104
12.4. MÁSOLAT VAGY REFERENCIA 105
12.5. CSATORNÁK VÁRAKOZÁSI SORÁNAK MÉRETE 107
12.6. STANDARD INPUT ÉS OUTPUT HASZNÁLATA VÁNDORLÁS SORÁN 107
12.7. VÁNDORLÁS FÁJL-BA ÍRÁS KÖZBEN 108
12.8. NÉV-SZERVER MÉRETE 108

A MUNKA ÉRTÉKELÉSE 109

IRODALOMJEGYZÉK 110

1. Bevezetés

A dolgozat célja a JoCaml nyelv bemutatása, elemzése és értékelése. A JoCaml egy párhuzamos, elosztott és objektum-orientált programozást is támogató funkcionális nyelv.

A dolgozat első része a JoCaml különböző nyelvi elemeinek – sorrendben: funkcionális (2.), imperatív (3.), objektum-orientált (4.), párhuzamos (5.) és elosztott (6.) – leírása, amely nemcsak a szintaxis megadását jelenti, hanem a nyelvi elemek vizsgálatát, elemzését, például általános tulajdonságok jelenlétét és a megvalósítás mértékét, vagy bizonyos jellemzők hiányát és a hiányt pótló, hasonló kifejezőerőt elérő megoldások kutatását. Ez egyáltalán nem könnyű feladat, mert a JoCaml nyelvről eddig semmilyen egységes leírás nem készült.

A dolgozat érdekesebb része JoCaml nyelv a párhuzamos és elosztott tulajdonságainak vizsgálata (az 5. Fejezettől a dolgozat végéig), melyet azonban az előtte lévő fejezetek nélkül nem lehetne megérteni. A párhuzamos és elosztott nyelvi elemek vizsgálata magában foglalja a szinkron és aszinkron kommunikáció megvalósítását (két különálló program, illetve programon belüli párhuzamos szálak között) és a mobil funkcionális kód használatát, elemzését is.

A dolgozat hátralévő része is a JoCaml párhuzamos és elosztott tulajdonságaival foglalkozik, de már nem az egyes nyelvi elemekkel, hanem inkább az egész nyelv átfogóbb elemzésével: a kifejezőerő vizsgálatával (7.) és az osztott rendszerek jellemzésére általánosan használt tulajdonságok megvalósításának mértékével (8.). Ezen tulajdonságok vizsgálata után – az értékeléssel összhangban – javaslatot teszek új nyelvi elemek bevezetésére (11.).

Bemutatom a nyelvben a párhuzamos programozást megvalósító join-kalkulus elméleti hátterét, majd kitérek a JoCaml fordítóprogramjának érdekesebb részeire (9.).

A dolgozat több példaprogramot is tartalmaz. Ahol szükségesnek tartom közölni, hogy mi a JoCaml rendszer válasza az adott kifejezésre, ott – a megkülönböztetés érdekében – a JoCaml kódot „`# `” jel vezeti be, a rendszer válasza pedig közvetlenül alatta olvasható (a „`# `” jel nélküli sorok; a válasz mindig tartalmazza a kifejezés típusát és értékét). Ahol inkább egészében maga a program, és nem az egyes kifejezéseinek típusa és értéke a fontos, ott nem szaporítom feleslegesen a sorokat: csak magát a kódot írom le („`# `” és a rendszer válasza nélkül). A program outputját minden esetben „`-> `” jel vezeti be.

1.1. A programozási nyelvek osztályozása

A magasszintű programozási nyelvek két nagy csoportra oszthatók [2]**:**

- **Imperatív nyelvek** (például: Fortran, Ada, C), melyek alapja Neumann-modell
- **Deklaratív nyelvek**, melyeknél a program egyenletekből és állításokból áll, a számítást deklarációk halmazával adjuk meg.
  A deklaratív nyelvek további két csoportra oszthatók:
  - **Funkcionális nyelvek** (például: Lisp, ML)
    A funkcionális nyelvek alapja a λ-kalkulus.
  - **Logikai programnyelvek** (például: Prolog, CLP nyelvcsalád)
    A logikai programnyelvek alapja a predikátumkalkulus. A végrehajtási mechanizmusuk alapja a rezolúció, amelyet logikai tételek bizonyítására fejlesztettek ki. Jellemzői a tények, a szabályok és a következtetőrendszer.

Az objektum orientált programnyelvek nem tekinthetők külön nyelvosztálynak, mivel mindegyik nyelvcsoportban találhatók olyan programnyelvek, amelyek az objektum orientált programozást

támogatják. Például az imperatív nyelveknél a C++, a funkcionális nyelveknél az Objective Caml, a logikai programnyelveknél pedig a Prolog Objects.

Az imperatív és a funkcionális programozási stílus összehasonlítása:

Imperatív nyelvek	Funkcionális nyelvek
A jelmondat a *hogyan*. A program a *megoldás módját* adja meg.	A jelmondat a *mit*. A program a megoldandó *feladat leírását* adja meg.
Az imperatív nyelvek alapja Neumann-modell struktúrája, ehhez a modellhez az alapot a Turing gép működésének matematikai modellje adja.	A funkcionális nyelvek alapja a lambda-kalkulus (például a Lisp-nél), illetve a kibővített lambda-kalkulus (például az ML-nél, vagy a Haskell-nél).
A program lényegében kétfajta információt használ: a gépi kódot reprezentáló *utasítás*t, amelyet a gép központi egysége interpretál, és amely szabályozza a számítógépben lezajló számítási folyamatot, és az *adat*ot, amellyel az utasítások manipulálnak.	A program *típus-* és *függvénydeklarációk*, valamint *kifejezések* összessége. A funkcionális nyelvekben az egész program egy nagy kiértékelendő kifejezésnek tekinthető.
A program végrehajtása a program utasításainak, parancsainak végrehajtását jelenti.	A program végrehajtása a kezdeti kifejezés kiértékelése. A végrehajtás elve a *redukció*, amely a program kifejezéseit egyszerűbb kifejezésekkel helyettesíti. A végrehajtás akkor ér véget, amikor a kifejezés tovább már nem egyszerűsíthető.
A program pillanatnyi állapotát változóinak pillanatnyi tartalma írja le. A program állapotát a leggyakoribb paranccsal, az értékadással – azaz a változók frissítésével – változtathatjuk meg. A változó egy változtatható értékű memóriahely.	A „változó" egy esetleg még ismeretlen, de (előbb –utóbb) rögzített értékű mennyiség, mely nem frissíthető (ezért nem is értékadásról, hanem kötésről beszélünk). A „változó" itt nem memóriahelyet azonosít, hanem egy kifejezés nevét.
Programrészek ismételt végrehajtását jellemzően ciklus használatával lehet elérni.	Az ismétlést rekurzió használatával lehet elérni.
A függvények „függvényeljárások", melyek okozhatnak mellékhatásokat.	„Tiszta" függvény használata (mellékhatás-mentesség). A függvény a matematikai értelemben vett függvényt jelenti: egyértelmű leképzés az argumentuma és az eredménye között. Egy függvény lehet egy másik függvény paramétere és / vagy értéke is.

Míg az imperatív nyelvekben a névvel azonosított változók adatok tárolására szolgálnak, az adatokat pedig utasítások végrehajtásával változtatják meg, addig ez a tulajdonság a funkcionális programnyelvekben *mellékhatás*ként jelentkezik [2]. Az imperatív programnyelvekben egy program működéséhez az utasítássorozatok dinamikus viselkedését kell ismerni, és előfordulhat, hogy egy függvény különböző meghívásokra különböző eredményt ad, éppen a mellékhatások következményeként.

Funkcionális programozási eszközökkel minden olyan feladat megoldható, amelyik megoldható imperatív nyelven, és viszont. A számítógépek teljesítménynövekedése és a funkcionális programozási nyelvek fordítási technikájának fejlődése együttesen azt eredményezte, hogy napjainkban a legtöbb programozási feladat megoldása során a funkcionális nyelven írt program hatékonysági szempontból is lényegében egyenértékű az imperatív nyelven fejlesztettel [1].

Az azonos feladat megoldására funkcionális nyelven írt programkód általában lényegesen rövidebb, kifejezőbb, olvashatóbb és könnyebben módosítható, mint az imperatív nyelven kódolt programszöveg. Ennek az az oka, hogy tisztán funkcionális nyelvekben az imperatív nyelvekből ismert változó nem létezik, így a kódrészletek mellékhatásokat sem okozhatnak. Egy programrészlet megváltoztatásának hatása sokkal inkább lokalizálható, így könnyebben követhető. Funkcionális nyelvek alkalmazásával nagy programok bonyolultsága lényegesen csökkenthető, a fejlesztéshez szükséges idő lerövidül, a program megbízhatóbb, kevesebb hibát tartalmaz [1].

A funkcionális szemléletmód lehetővé teszi olyan programok írását, amelyek helyessége a matematikában alkalmazott módszerekkel (például teljes indukcióval) könnyen bizonyítható, hiszen a funkcionális programozásban használt függvényfogalom a matematikai függvényfogalomnak felel meg.

1.2. A funkcionális programozási nyelvek

1.2.1. A funkcionális programok végrehajtása

Egy funkcionális program *végrehajtás*a nem más, mint a program kezdeti kifejezésének *kiértékelés*e. A kiértékelést úgy képzelhetjük el, mint a kezdeti kifejezés átalakítása a benne szereplő függvények szövegszerű behelyettesítésével (*redukció*): a redukálandó részkifejezésben (*redex-ben*) a függvényhívás helyettesítődik a függvény törzsében megadott kifejezéssel a formális és aktuális paraméterek megfeleltetése mellett. *Normál formá*jú egy kifejezés, ha további redukcióra nincs lehetőség, ez az átírási lépéssorozat végeredménye. Az átírási lépések pontos jelentését a nyelv modellje definiálja. A funkcionális nyelven írt program végrehajtási modellje minden esetben egy *konfluens redukciós rendszer* (*átírórendszer*). Konfluens egy átíró rendszer, ha az egyes részkifejezések átírásának sorrendje a végeredményre nincs hatással, a sorrend legfeljebb azt befolyásolja, hogy az átírási lépéssorozattal eljutunk-e a végeredményig. Konfluens rendszer a λ-kalkulus, illetve léteznek konfluens kifejezésátíró és gráfátíró rendszerek is [1].

A **kiértékelési stratégia** azt határozza meg, hogy milyen sorrendben értékeljük ki a kifejezéseket.

Lusta (lazy) kiértékelési stratégia: a kifejezések kiértékelésekor először a kifejezéssel ekvivalens λ-kifejezésben a legbaloldalibb legkülső redukálható kifejezést (olyan redex, amelyik nincs másik redex belsejében) helyettesíti. Azaz a függvénydefiníciót alkalmazza elsőként és az argumentumokat csak akkor értékeli ki, ha az argumentumra ténylegesen szükség van. Ez a stratégia mindig megtalálja a normálformát, ha létezik [1]. Ilyen kiértékelést használ például a Miranda és a Haskell.

Mohó (szigorú, strict) kiértékelés: a legbaloldalibb, legbelső redukálandó kifejezéssel (olyan redex, amelyik belsejében nincs másik redex), azaz az argumentumok redukálásával kezdi. Ennél a kiértékelésnél a függvény argumentuma a függvény hívásakor mindig kiértékelődik. A mohó kiértékelés gyakran hatékonyabb, de nem mindig terminál akkor sem, ha létezik normálforma [1]. Ilyen kiértékelést használ például az ML, a Lisp és a Scheme.

A két módszer használatában a lényeges különbség az, hogy a nem-szigorú nyelvek programjai akkor is szabályosan befejeződhetnek, ha bennük egy függvény argumentumának kiszámítása végtelen ciklusba kerül, vagy hibajelzést ad [2].

A gyakorlatban ritkán használnak teljesen lusta, vagy teljesen mohó kiértékelést: a mohó nyelvek gyakran tartalmaznak olyan nyelvi elemeket, amelyek egyes kifejezések lusta kiértékelését írják elő, ill. lusta nyelvek gyakran alkalmaznak eljárásokat arra vonatkozóan, hogy szabad-e egy kifejezést hatékonyan, mohó stratégiával kiértékelni.

1.2.2. A funkcionális programozási nyelvek osztályozása

A funkcionális nyelveket többféle szempont szerint lehet osztályozni:

- Egyrészt lehet az előző részben bemutatott ***kiértékelési stratégiák szerint***, eszerint egy nyelv lehet *mohó*, vagy *lusta*.
- Osztályozhatjuk aszerint is, hogy ***tisztán funkcionálisak-e*** (pl.: Clean), vagy sem (pl.: SML).
- A nyelveket csoportosíthatjuk aszerint is, hogy a ***típusok ellenőrzése mikor történik***, így lehetnek *dinamikus típusú* nyelvek (az ellenőrzés futás közben történik, pl. Lisp), és *statikus típusú* nyelvek (fordításkor ellenőrzi a típust, pl.: az ML nyelvcsalád).

1.2.3. A tisztán funkcionális programok jellemzői

Tisztán funkcionális egy programozási nyelv, ha nyelvi elemei felhasználásánál mellékhatások garantáltan nem lépnek fel, az előző értéket megsemmisítő értékadás vagy más imperatív programozásra jellemző nyelvi elem nem áll rendelkezésre. A tisztán funkcionális programozási nyelvek a matematikában megszokott függvényfogalmat valósítják meg: a függvény egyértelmű leképzés a függvény argumentuma és eredménye között, a függvény alkalmazásának nincs semmilyen más hatása [1].

A modern, tisztán funkcionális programozási nyelvek legfontosabb jellemzői [1]**:**

- **Hivatkozási átlátszóság (referential transparency)**
  A kifejezések hivatkozási szempontból átlátszóak, azaz ugyanaz a kifejezés bárhol a program szövegében ugyanazt az értéket jelöli. A függvények kiértékelésének nincs mellékhatása, azaz egy függvény kiértékelése nem változtatja meg egy kifejezés értékét. A tisztán funkcionális program változói valójában konstansok (a változók értéke esetleg még nem ismert, de semmiképpen sem változhat meg a program végrehajtása során).
  Ez a tulajdonság lehetővé teszi, hogy a matematikában használt szimbólum-helyettesítést és a teljes indukciót a programnyelvben is használni lehet, és így egy függvény definíciója a funkcionális nyelven leírva azonos a definíció matematikai leírásával, vagy a két leírás között az eltérés minimális. Ha a matematikai leírás helyessége bizonyítható, akkor, kihasználva a hivatkozási átlátszóság tulajdonságot, a programnyelvi definíció helyessége ugyanazzal a módszerrel bizonyítható [2].

- **Szigorú, statikus típusosság**
  Bár nem kötelező a típusdeklarációk megadása, de megköveteljük, hogy a kifejezések típusa a típuslevezetési szabályok által meghatározott legyen. Biztosítottak a nyelvi eszközök absztrakt és algebrai adattípusok definiálásához is.

- **Magasabbrendű függvények**
  A függvények ugyanolyan értékek, mint az elemi típusértékhalmazok elemei. Magasabbrendű függvénynek nevezzük azokat a függvényeket, amelyeknek valamelyik argumentuma vagy értéke maga is függvény.

- **Curry módszer**
  Többváltozós függvények helyettesítése egyváltozós függvények ismételt alkalmazásával, magasabbrendű függvények felhasználásával.

- **Függvények alkalmazása önmagukra**: rekurzív (sőt, akár kölcsönösen rekurzív) függvénydefiníciók adhatók meg.

- **Zemelo-Frankel halmazkifejezések**
  Iteratív adatszerkezet elemeinek és azok sorrendjének megadására alkalmas, a matematikában halmazok megadásánál alkalmazott jelölésrendszernek megfelelő nyelvi eszköz. Végtelen adatszerkezetek kiértékelése lusta kiértékelési módszerrel történik.
- Argumentumok **mintaillesztés**e
  Függvények definiálásakor megadhatunk mintákat a formális argumentum helyén. Amennyiben az aktuális argumentum illeszkedik a megadott mintára, akkor a függvény értékét a megfelelő függvénytörzs-változat definiálja.
- **Margó szabály**
  Összetartozó kifejezések csoportjának azonosítására és deklarációk hatókörének korlátozására alkalmas a baloldali margó szélességének változtatása. A margó szabály a blokkstruktúra kialakításának egyik nyelvi eszköze.
- A modern funkcionális nyelvek rendelkeznek valamilyen **I/O modell**el is.

1.3. A JoCaml nyelv

A JoCaml rendszer az Objective Caml nyelv kísérleti kiterjesztése az elosztott join-kalkulus programozási modellel. Ez a modell magasszintű kommunikációs és szinkronizációs csatornákat, mobil ágenseket, leállás-detektálást és automatikus memória-kezelést tartalmaz. A JoCaml segítségével nagy elosztott alkalmazások gyorsan fejleszthetők, a programozók kihasználhatják mind az Objective Caml könnyű programozhatóságát és kiterjesztett könyvtárait, mind pedig a join-kalkulus elosztott és párhuzamos jellemzőit [8].

A JoCaml tehát egy **elosztott funkcionális nyelv**, ahol a számítási nyelv az Objective Caml, a vezérlő nyelv pedig a join-kalkulus.

A JoCaml legújabb változata még csak béta-verzió. A nyelv nyílt forrású (*open source*).

1.3.1. A JoCaml történeti áttekintése, elődei

1.3.1.1. ML

Az első típusos funkcionális nyelv az ML (Meta Language), amely eredetileg az Edinburgh-ban tételbizonyításra tervezett LCF (Logic for Computable Functions) rendszer meta-nyelve volt. A nyelvet R. Milner tervezte a 1975-ben. 1983 és 1990 között definiálta Milner, Tofte és Harper az SML-t (Standard ML). Az SML legújabb, átdolgozott szabványa 1997-ben készült el.

Az ML változatok nem tisztán funkcionális nyelvek, az imperatív stílusú programozáshoz szükséges nyelvi elemek is megtalálhatók bennük: frissíthető változók, tömbök, mellékhatással járó függvények stb.

Az SML fontosabb tulajdonságai:

- Mohó kiértékelést használ (először az argumentumokat értékeli ki)
- Erősen típusos (strong typing)
- Statikusan típusos nyelv, azaz minden típusellenőrzés még fordítási időben elvégezhető
- Típus automatikus levezetése.
- Rekurzív típusok definiálhatók, lineárisok (például lista) és nemlineárisok (például fa) is
- Polimorfizmus
- Modularitás (paraméterezhető modulok, szignatúra, struktúra, funktor)
- Absztrakt típus

1.3.1.2. Caml

A Caml egy erősen típusos funkcionális nyelv az ML nyelvcsaládból (INRIA [Institut National de Recherche en Informatique et en Automatique], 1984-1990, a Coq tételbizonyító alapnyelve).

A Caml Light (INRIA) egy kisebb, jobban hordozható (*portable*) implementációja az alap Caml nyelvnek. Az első verzió 1989-ben jelent meg (fejlesztette Xavier Leroy), a legújabb változat (0.75 verziószámmal) pedig 2002. januárban. A nyelv jelenleg még karbantartott, de már nem fejlesztik. Nyílt forrású.

1.3.1.3. Objective-Caml

Az Objective Caml az ML-család objektumelvű programozási nyelve (mint a Caml Special Light továbbfejlesztése, INRIA 1990-).

Az Objective Caml főbb újdonságai a Caml Light-hoz képest:

- Objektumok és osztályok teljes körű támogatása
- Erős modul rendszer a Sandard ML stílusában (de megtartva a különfordítás lehetőségét)
- Jó minőségű natív kód fordító (a Caml Light-stílusú bytecode fordítón kívül)

Fejlesztése jelenleg is folyik, a legújabb, 3.07-es verziója 2003. szeptemberében jelent meg. Ez is nyílt forrású.

1.3.1.4. Join-kalkulus

A join-kalkulus egy kísérleti nyelv, ami a folyamat-algebrán (*homonymous process calculus*) alapul. Egyszerű támogatást biztosít a párhuzamos és elosztott programozáshoz.

A join-kalkulus programozási modell jellemzői a több gépen futó párhuzamos folyamatok, a statikus típusellenőrzés, a globális lexikális láthatóság, az átlátszó távoli kommunikáció, az ágens-alapú mobilitás és bizonyosfajta hiba-detektálás.

1.3.1.5. JoCaml

A JoCaml legújabb változata (2003. április) tartalmazza az 1.07-es verziójú Objective Caml-t és a join-kalkulust megvalósító Join könyvtárat.

1.3.2. A JoCaml nyelv tulajdonságai

- **Funkcionális nyelv:** A függvényeknek kiemelt szerepe van a nyelvben, egy JoCaml program alapvetően függvénydefiníciókból és függvényalkalmazásokból áll. A függvények elsőrendű (*first class*) nyelvi elemek.
- **Nem tisztán funkcionális:** Tartalmaz módosítható adattípusokat is (például tömb, referencia), melyeken értelmezve van az értékadás művelete. Ha a programunkban ilyen típusú változókat használunk, akkor a hivatkozási átlátszóság sérülhet, megjelenhetnek a mellékhatások. A nyelv tartalmaz más imperatív programozásra jellemző nyelvi elemeket is, például ciklusokat.
- **Mohó kiértékelési stratégiát használ.**
- **Erősen típusos:** Minden értéknek, objektumnak, formális paraméternek és kifejezésnek a típusa egyértelműen meghatározható. Ha valamit típus szerint rossz környezetben használunk, akkor fordítási hibát kapunk (nincs implicit típuskényszerítés).
- **Statikusan típusos**: A típusok ellenőrzése (például a formális és aktuális paraméterek kompatibilitásának ellenőrzése) fordítási időben történik. Végrehajtás közben semmilyen ilyen jellegű ellenőrzés nincs, ami növeli a hatékonyságot.

- **Van típuslevezetés:** a programban általában nem kell megadni a típusokat, a fordító a használatból ki tudja következtetni. A levezetés az típus-ellenőrzéssel együtt történik, még fordítási időben.
- Használhatunk **magasabbrendű függvény**eket, és írhatunk – esetleg kölcsönösen **– rekurzív függvény**eket is. A rendszer a többváltozós függvényeket a **Curry módszer** szerint kezeli.
- A nyelvben **Zemelo-Frankel halmazkifejezések nincsenek**, de – mivel a nyelv mohó –, ezen nincs is semmi meglepő, hiszen ezek a halmazkifejezések végtelen sok elemet is tartalmazhatnak, amit mohó módon nem lehet kiértékelni.
- **Van mintaillesztés.**
- **Nincs margó szabály.**
- **A nyelv támogatja a paraméteres polimorfizmust**: írhatunk közös, általános kódú függvényt különböző típusú adatstruktúrákhoz (van típusváltozó).
- **A nyelv tartalmaz kivételkezelést.**
- **Paraméteres modul-rendszer (module system):** lehet absztrakt adattípusokat létrehozni, és funktorokat (modulokon értelmezett függvényeket) használni.
- **Objektum orientált:** Támogatja osztályok definiálását, objektumok létrehozását, az öröklődést és az újrafelhasználást.
- **Támogatja az elosztott és párhuzamos programozást**, a kommunikáció és a szinkronizáció csatornák használatával lehetséges. A párhuzamosan futó és együttműködő komponensek nemcsak szinkronizálnak egymással és adatokat küldenek egymásnak, hanem mobil kódrészleteket is továbbítanak.
- A nyelvben három **végrehajtási mód** érhető el [8]:
    - Interaktív környezet (*top-level*): minden beírt mondatot azonnal kiértékel, majd az eredményt és annak típusát kiírja a képernyőre (*read-eval-print* ciklus); ez egy kényelmes környezet mind a tanuláshoz, mind a programok gyors teszteléséhez és hibakereséséhez.
    - Bytecode fordító: a programot bytekódra fordíthatja le, amit egy C program interpretál.
    - Natív kód fordító.

1.3.3. Egy JoCaml program felépítése

Egy JoCaml program felépítése eltér a „hagyományos” funkcionális programok felépítésétől. A „hagyományos” funkcionális nyelvek ugyanis általában két egymástól jól elkülöníthető részből állnak: egy definíciós részből (amely több definíciót is tartalmazhat) és **egy** kezdeti kifejezésből. Egy ilyen program végrehajtása pedig nem más, mint a kezdeti kifejezés kiértékelése.

Ezzel szemben a JoCaml-ben a definíciók és a kifejezések nem különülnek el egymástól ilyen élesen, a program vegyesen tartalmazhatja őket, kifejezésből pedig akár **több** is lehet. A program végrehajtása itt azt jelenti, hogy a rendszer az összes kifejezést a forráskódbeli leírásuk sorrendjében kiértékeli. (Ez egyfajta kötegelt feldolgozásnak is tekinthető.)

Tehát egy JoCaml program JoCaml **mondat**ok (*phrase*) sorozata. A mondat egy teljes, közvetlenül végrehajtható szintaktikus egység, ami:

- vagy egy **deklaráció**
  négy különböző típusú deklaráció lehetséges, mindegyiket más kulcsszó jelöl:
  - **let**: azonosító deklaráció
  - **exception**: kivétel deklaráció
  - **type**: típusdeklarácó
  - **class**: osztálydeklaráció
- vagy egy **kifejezés** (*expression*). A kifejezés egy számítás leírása. Egy kifejezés kiértékelése mindig visszaad egy értéket – a számítás eredményét – a kiértékelés végén. (Megjegyzés: az elágazások és ciklusok – amik az imperatív nyelvekben a program vezérlőszerkezetei – JoCaml-ben „csak” kifejezések; itt a függvények a végrehajtás vezérlőszerkezetei.)

A mondatok végén dupla pontosvessző (`;;`) áll.

Folyamatok:

Fontos, hogy egy JoCaml program a deklaráción és a kifejezésen kívül **folyamat**okat (*process*) is tartalmazhat. Ugyan a program alapvetően deklarációk és kifejezések sorozata, azonban mind a definíciók, mind a kifejezések – bizonyos jól meghatározott körülmények között – tartalmazhatnak folyamatokat. (Azonban a folyamat nem állhat „csak úgy magában”, tehát például a következő program szintaktikailag nem helyes: *deklaráció*`;;` *kifejezés*$_1$`;;` ***folyamat***`;;` *kifejezés*$_2$`;;`)

A folyamatok – a kifejezésekhez hasonlóan – egy számítást írnak le. Valójában a kifejezések és a folyamatok nagyban hasonlítanak egymásra. Azonban a kiértékelésben alapvető **különbség**ek vannak:

- A rendszer a **kifejezés**eket **szinkron** módon értékeli ki: teljesen végrehajtja a kifejezés kiértékelését, és csak utána lép tovább a következő kifejezésre, folyamatra vagy deklarációra. A kifejezéseknek a kiértékelés végén mindig **van** valamilyen **eredmény**e (értéke).
- A **folyamat**ok **aszinkron** módon hajtódnak végre: a rendszer elkezdi a folyamat kiértékelését (egy külön szálon), de nem vár annak befejezésére, hanem egyből továbblép. A rendszer tehát nem várja meg a kiértékelés végét, így a visszaadott értéknek sem lenne sok értelme (mivel nem tudható előre, hogy a folyamat mikor fejeződik be). Éppen ezért a folyamatok nem adnak (nem is adhatnak) vissza **semmilyen érték**et.

Megjegyzés: Már a kiértékelésből is látszik, hogy a folyamatok leginkább a program párhuzamosításában hasznosak, egy szekvenciális programot minden további nélkül megírhatunk folyamatok nélkül is. (Objective Caml-ben nem is voltak folyamtok, ez csak a join-kalkulussal történő kibővítésénél került bele a nyelvbe.) Mivel a dolgozatban először a nyelv funkcionális, imperatív és objektum-orientált nyelvi elemeit mutatom be, és csak ezek után térek át a párhuzamos programozásra, így a következő három fejezetben folyamatok csak elvétve fognak előfordulni; ezért, ha egy JoCaml mondat nem deklaráció és külön nem írom oda, hogy folyamat, akkor minden további nélkül feltételezhető, hogy egy kifejezésről van szó.

Futtatás:

Az **interaktív rendszer**ben (*top-level*) minden mondat beírása után a rendszer kiértékeli a mondatot, majd kiírja az eredményt a képernyőre. Azonban, ha a programot a **lefordít**juk, majd **végrehajt**juk, akkor program nem írja ki automatikusan sem a típusokat, sem a kifejezések értékekeit. A programban kiírató függvényeket (például a `print_int` *szám*) kell meghívni, ha valami eredményt is szeretnénk látni.

2. A JoCaml nyelv funkcionális nyelvi elemei

2.1. Lexikális elemek

A nyelvben ASCII karaktereket használhatunk.

Elhatároló jelek: szóköz, újsor, tabulátor. JoCaml-ben nincs a funkcionális nyelvekben gyakori *margó szabály*, tehát a program tagolásának a fordító számára semmi jelentése nincs.

Azonosító: kis- és nagybetűkből, számokból, a „'” és a „_” jelből állhat, de nem lehet a „_” jel (önmagában). A nyelv megkülönbözteti a kis- és nagybetűket.
Az azonosítónak kisbetűvel, vagy „_” jellel kell kezdődnie, ha „változónév”, függvénynév, típusnév, típuskonstruktor, rekord mezőjének a neve, osztálynév, metódusnév, vagy adattag.
Nagybetűvel kell kezdődnie az adatkonstruktornak, a kivételnévnek és a modulnévnek.
Az azonosító hosszára nincs megkötés. A foglalt szavak (például if, then, else, for stb.) nem használhatók azonosítóként.

Operátor: operátor_karakter-ekből állhat. Az, hogy az operátor prefix, vagy infix, a kezdő karaktere határozza meg: ha prefix_karakter-rel kezdődik, akkor prefix lesz, ha infix_karakter-rel, akkor infix.

```
operátor_karakter ::= ( ! | $ | % | & | * | + | - | . | / | : | < | = | > | ? | @ | ^ | | | ~ )
prefix_karakter   ::= ( ! | ? | ~ )
infix_karakter    ::= ( = | < | > | @ | ^ | | | & | + | - | * | / | $ | % )
```

Definiáláskor, az operátornevet zárójelbe kell tenni. (Abban az esetben is zárójelbe kell tenni, ha nem számolunk, hanem, mint függvénnyel dolgozunk vele, pl.: let osszead = (+))
Lehet meglévő operátorokat átdefiniálni, így például a következő operátornevek is megengedettek: (+), (-), (*), (/) stb. Azonban, ha létrehozunk ilyen nevű operátorokat, akkor ezek az eredeti jelentést „eltakarják” [6].

Egész számok: alapértelmezett a számrendszer a 10-es, de használhatunk 16-os (0x, vagy 0X prefix-szel), 8-as (0o, vagy 0O prefix-szel) és 2-es (0b, vagy 0B prefix-szel) számrendszert is. A 16-os számrendszerbeli számoknál a betűket írhatjuk kis- és nagybetűkkel is [6].

Lebegőpontos számok: *<egész_rész>*.*<tört_rész>***e***<kitevő>* (a lebegőpontos számnak az egészektől való megkülönböztetés miatt vagy a tizedespontot, vagy az e-t tartalmaznia kell). Például 3.; 2.5; 3e6; 2.7e-2

Karakter: a karaktereket aposztrófok között kell megadni ('*karakter*'), használhatunk ASCII kódot ('*ascii_kód_3_karakteren*') és vezérlőkaraktereket is. Pl.: 'a', '\040', '\112', '\t', '\n'.

Szöveg: a szöveget idézőjelek között kell megadni: "*szöveg*". A szöveg tetszőlegesen hosszú lehet és használhatunk *escape szekvenciát* (pl.: \n) is [6].

Megjegyzés: „(*” és „*)” jelek között, a megjegyzések egymásba ágyazhatók.

2.2. Beépített elemi adattípusok

unit: ebbe a típusba egyetlen elem tarozik: „()”. Ez például a C nyelv void-jának megfelelő elem. Azért van szükség egy ilyen „üres” kifejezésre, mert minden függvénynek vissza kell adnia valamilyen értéket (például az értékadó utasítás mindig ()-t ad vissza) [4].

int: az egész számok típusa. Műveletei (többek között) [4]:

- Alapműveletek: összeadás (+), kivonás (–), szorzás (*), osztás (/), maradékképzés (`mod`) Megjegyzés: ezek a függvények mind `int -> int -> int` típusúak, így az osztás is mindig egész számot ad vissza.
- Bináris léptetések: logikai léptetés balra: $i*2^j$ (*i* **lsl** *j*), logikai léptetés jobbra: $i/2^j$ (*i* **lsr** *j*)
- Bitenkénti operátorok: *i* **land** *j*; *i* **lor** *j*; *i* **lxor** *j*

float: lebegőpontos számok típusa. Műveleti (többek között) [4]:

- Alapműveletek: összeadás (`+.`), kivonás (`-.`), szorzás (`*.`), osztás (`/.`), hatványozás (`**`)
- Egyéb függvények: felfelé kerekítés (`ceil`), lefelé kerekítés (`floor`), gyökvonás (`sqrt`), e^x (`exp`), logaritmus (e-alapú: `log`, 10-es alapú: `log10`)
- Trigonometrikus függvények: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`
- Egész szám lebegőpontossá konvertálása: `float` *egész_szám*

char: karakterek típusa. Műveletei (többek között):

- ASCII kóddal kapcsolatos függvények: ASCII kód meghatározása (`Char.code` vagy `int_of_char`), adott kód alapján a karakter meghatározása (`Char.chr` vagy `char_of_int`)
- Kis- és nagybetűs konverziók: `Char.lowercase`, `Char.uppercase`

string: szöveges típus. Műveletei (többek között):

- Szövegek összefűzése: s_1**^**s_2
- Szöveg hosszának lekérdezése: `String.length`
- Szöveg egy karakterének lekérdezése: `s.[i]` (vagy `String.get s i`). A szöveg karaktereit meg is lehet változtatni: `s.[i] <- c` (vagy `String.set s i c`). A karakterek számozása 0-tól kezdődik.
- Rész-szöveg lekérése: `String.sub` *s kezdő_pozíció hossz*
- Konverziós függvény: például `string_of_int`

bool: logikai értékek, a típusba két érték tartozik: `true`, `false`. Műveletei [4]:

- Összehasonlítás: =, <>, <, <=, >=, >
- Logikai műveletek: tagadás (`not`), és (`&&`), vagy (`||`); az utóbbi kettő lusta módon értékelődik ki.

Megjegyzés: a beépített elemi adattípusok közül a `string` az egyetlen olyan típus, amelybe tartozó elemek értéke változtatható (*mutable* típus).

2.3. Azonosítók és kötés

JoCaml-ben a *változók* tulajdonképpen értékek *nevei* (egy tisztán funkcionális nyelvben nincs különbség, hogy a változó nevét vagy az értékét használjuk). A *kötés* nem értékadás, hanem egy olyan szerkezet, ami bevezet egy új azonosítót (aminek az értékét is egyből megmondjuk) egy új hatókörrel.

Azonosítóhoz értéket hozzárendelni (kötni) a `let` kulcsszóval lehet:

```
let név = kifejezés1 [in kifejezés2]
```

Több kötés is összevonható egy `let`-ben (szimultán deklaráció):

```
let név1 = kifejezés1
and név2 = kifejezés2
…
and névn = kifejezésn
[in kifejezésn+1]
```

vagy:

```
let név1, …, névn =
    kifejezés1, …, kifejezésn
[in kifejezésn+1]
```

2.3.1. Hatókör és láthatóság

A hatókör statikus.

Globális deklaráció: A `let név = kifejezés` egy **deklaráció**, ezért (lásd az 1.3.3. Egy JoCaml program felépítése részben) csak globálisan állhat, nem ágyazható be kifejezésbe vagy folyamatba. Az azonosító hatóköre a deklaráció utáni *mondat*tól a program végéig tart. Ha a programban később készítünk egy újabb deklarációt ugyanezen a néven, akkor nem az eredeti azonosító változik meg, hanem létrejön egy új azonosító, mely – míg a hatóköre tart – eltakarja az eredeti azonosítót. Tehát az azonosítók hatóköre lexikális, ha több definíció is van egy változóra, akkor a legfrissebb érvényes.

Lokális deklaráció: A `let név = kifejezés1 in kifejezés2` Ez így egészében nem deklaráció, hanem **kifejezés** (ami tartalmaz lokális deklarációt) [5]. A rendszer létrehozza a *név* lokális azonosítót (értékét a *kifejezés*$_1$ kiértékelése adja), majd kiértékeli a *kifejezés*$_2$-t. Egyben ennek eredménye lesz az egész `let név = kifejezés1 in kifejezés2` kifejezés értéke is. A *név* azonosító hatóköre és láthatósága egyedül a *kifejezés*$_2$-re terjed ki. Tehát, mivel ez így összességében nem deklarációnak, hanem kifejezések számít, így mindenhol szerepelhet, ahol kifejezés lehet (például függvények, csatornák törzsében, stb).

2.4. Függvények

A funkcionális programozási nyelvekben a függvények „teljes jogú” (*first-class*) értékek. Ez azt jelenti, hogy a függvények ugyanolyan értékek, mint bármi más. Minden olyan helyen, ahol „egyszerűbb” típusok (pl. egész számok, karakterek stb.) állhatnak, ott szerepelhetnek függvények is, például függvényeket átadhatunk paraméterként, egy függvény lehet egy másik függvény visszatérési értéke, készíthetünk függvényekből álló párokat, listákat, tömböket stb.

A tisztán funkcionális programokban a függvények visszatérési értéke determinisztikus, és csak a paraméterek aktuális értékétől függ. A függvényeknek nincsenek külső kapcsolataik, így nem lehetnek mellékhatásaik sem. Azaz indifferens a függvény értékének szempontjából, hogy ki, mikor és milyen környezetben hívta meg.

A függvény típusa α `->` β, ahol α a paraméter (argumentum, vagy értelmezési tartomány), β pedig az eredmény (értékkészlet) típusát jelöli.

2.4.1. Egyszerű függvények

Függvények definiálása:

`fun formális_paraméterek -> függvénytörzs` (pl.: `fun x -> x*x;;`). (Megjegyzés: a paramétereket *paraméter*$_1$ *paraméter*$_2$ … *paraméter*$_n$ alakban kell megadni.) A *függvénytörzs* a függvény értékét és kiszámítási módját is definiáló **kifejezés**. A függvénytörzsben megadhatunk a paraméterekre vonatkozó *feltétel*eket és *mintá*kat is, amelyek az aktuális és a formális paraméterek *illesztés*ét határozzák meg. A függvények definiálásakor a formális paraméterek leírására változóneveket vezetünk be (*paraméter*$_1$ …), a változó deklarációjának *hatóköre* ez esetben a *függvénytörzs*re terjed ki. Megjegyzés: az előbbi

függvény a típusa `int -> int`, ami azt jelenti, hogy egy egész számot kap paraméterként, majd a számítás végén is egy egész számot ad vissza [1].

Függvények használata (alkalmazása):
(függvénydefiníció) aktuális_paraméterek (pl.: `(fun x -> x*x) 5;;`). A függvényhívás prioritása magasabb, mint a legtöbb operátoré.

Látszik, hogy a JoCaml-ben a függvényeknek alapértelmezésben nincs nevük. Mivel azonban a függvények ugyanolyan értékek, mint bármi más, a `let` segítségével elnevezhetjük őket (például: `let incr = fun i -> i+1;;`). A névvel ellátott függvények meghívása: *függvénynév aktuális_paraméterek* (például: `incr 3;;`)

Mivel a névvel ellátott függvények igen gyakoriak, ezért egyszerűbb szintaxissal is létre lehet hozni nevesített függvényt: `let` *név formális_paraméterek* `=` *függvénytörzs* (például: `let incr i = i+1;;`).

Hatókör tekintetében a függvényekre ugyanazok a szabályok érvényesek, mint más azonosítókra.

Paraméterek:
A paraméterátadás az imperatív nyelvek *cím szerinti* paraméterátadásának felel meg. Tehát, ha a függvényben valamely paraméter értékét megváltoztatjuk (a JoCaml tartalmaz frissíthető változókat), akkor ez a változtatás a függvényből kilépve is megmarad.
A paramétereknek nem lehet alapértelmezett értéket adni.

Lokális deklaráció a függvénytörzsön belül:
Mivel a `let` *név* `=` *kifejezés*$_1$ `in` *kifejezés*$_2$ egy kifejezés, ezért természetesen lehet egy függvény törzse is. Így definiálhatunk a függvénytörzsre nézve lokális azonosítókat. Például (x^3):

```
let kob x =
   let negyzet = x*x in
   x*negyzet;;
```

Minden függvényhívásnál létrejön egy lokális azonosító melyre csak a függvény törzséből lehet hivatkozni. Azonban az azonosító élettartama nem feltétlen korlátozódik a csak a függvény törzsére, bizonyos helyzetekben megmarad a függvényből kilépve is. Erre az esetre egy példa 12.2. Lokálisan deklarált azonosító élettartama című fejezetben található (a példa logikailag ide tartozna, de mivel a program tartalmaz módosítható azonosítót és szekvenciát is – melyekről csak később lesz szó –, ezért került a függelékbe).

2.4.2. Magasabbrendű függvények

Magasabbrendű az a függvény, amelynek vagy a paramétere, vagy az értéke függvény. Például nézzük a következő függvényt: `let twice f x = f (f x);;` A `twice` függvény paraméterként kap egy függvényt (`f`) és egy értéket (`x`), majd az értékre kétszer is alkalmazza az `f` függvényt.

2.4.3. Többváltozós függvények

A többváltozós függvény kezelésére elméletileg két lehetőség van:

1. A paramétereket ***összetett adatnak*** – párnak, rekordnak, listának stb. – tekintjük, így lényegében minden függvény egyváltozós lesz.
2. ***Curry módszer***: a többváltozós függvényeket egyváltozós, magasabbrendű függvények ismételt alkalmazásával helyettesítjük. Ez a változat rugalmasabban használható, a JoCaml is – mint a funkcionális nyelvek általában – ezzel a módszerrel kezeli a többváltozós függvényeket.

A Curry módszerrel például az összeadás műveletét (`let sum i j = i+j;;`) is egyváltozósnak tekintjük, oly módon, hogy az összeadás első paraméterét (`i`) tekintjük a `sum` művelet egyetlen paraméterének, a függvény eredménye pedig az a függvény, amely az `i`-nek megfeleltetett aktuális paraméterhez ad hozzá egy numerikus értéket, amelyet az új függvény egyetlen formális paramétere, a `j` biztosít.
A `sum` függvény a típusa: `int -> int -> int`, ami – mivel a függvények típusdefiníciói (`->`) jobbról balra asszociatívak – egyenértékű a következő típussal: `int -> (int -> int)`, tehát a `sum` egy egyváltozós függvény, ami egy másik (szintén egyváltozós) függvényt ad vissza.
Egy ilyen „többváltozós” függvény használata: `sum 3 4 ≡ (sum 3) 4`. A `sum 3` egy olyan függvényt ad vissza, amely az (egyetlen) paraméteréhez mindig `3`-at ad hozzá (`int -> int` típusú függvény), és ezt az eredményül kapott függvényt alkalmazzuk a `4`-re. A többváltozós függvények alkalmazása tehát balról jobbra asszociatív [1].

Egy `n`-változós függvényből könnyen készíthetünk `k`-változós (`1 ≤ k < n`) függvényt, úgy, hogy `n-k` paraméter értékét rögzítjük, így a kapott függvény már csak a maradék `k` paramétertől függ. Például: `let inc = sum 1;;` [2].

JoCaml-ben a paraméterek kiértékelésének sorrendje nem meghatározott. Ugyan jelenleg az összes implementáció balról jobbra értékeli ki őket, azonban ez egy későbbi implementációban megváltozhat [6].

2.4.4. Rekurzív *függvénydefiníció*

Rekurziót alapértelmezésben nem használhatunk, csak a `rec` kulcsszó esetén, például nézzük a hatványozás megvalósítását [4]:

```
let rec power x i =
   if i = 0 then
      1.0
   else
      x *. (power x (i-1));;
```

Kölcsönösen rekurzív függvényeket is készíthetünk. Ezeket kötelező egy `let`-ben elhelyezni, például:

```
let rec paros x =
   if x < 0 then paros ((-1)*x)
   else (if x = 0 then true else paratlan (x-1))
and paratlan x =
   if x < 0 then paratlan ((-1)*x)
   else (if x = 0 then false else paros (x-1));;
```

2.5. Egyszerű kifejezések

2.5.1. Zárójelezett kifejezés

A következő **kifejezés**ek ekvivalensek: `(`*`kifejezés`*`) ≡ begin `*`kifejezés`*` end ≡ `*`kifejezés`*

A zárójelezett kifejezésre akkor van szükség, amikor az alapértelmezett műveleti sorrendet (*prioritás*t) meg szeretnénk változtatni, például:

```
# (3+2)*4
- : int = 20
```

vagy:

```
# begin 3+2 end *4
- : int = 20
```

Megjegyzés: a `begin … end` leginkább az elágazás és a ciklus kifejezésekben használatos, a többi helyen inkább a zárójelek használatával csoportosítsunk.

Folyamatokat is zárójelezhetünk, azonban folyamatoknál sem a `begin … end`, sem a kerek zárójel nem használható; a folyamatokat kapcsos zárójel segítségével lehet csoportosítani: `{ folyamat }`.

2.5.2. Elágazás

Szintaxis: `if` *feltétel*$_1$ `then` *kifejezés*$_1$ `[else` *kifejezés*$_2$`]`. A kifejezések típusa meg kell, hogy egyezzen. Ha az `else` rész hiányzik, akkor az automatikusan egy „`else ()`”-t jelent, emiatt ebben az esetben *kifejezés*$_1$-nek is `unit` típusúnak kell lennie.

Az *if-then-else problémát* úgy oldották meg a nyelvben, hogy az `else` a hozzá legközelebb eső `if`-hez tartozik (természetesen ez zárójelezéssel felülbírálható).

Elágazást kifejezésekből és folyamatokból is létrehozhatunk (azonban „keverni” nem lehet):

- `if` *kifejezés* `then (` *kifejezés*$_1$ `) else (` *kifejezés*$_2$ `)` (ez egy **kifejezés**)
- `if` *kifejezés* `then {` *folyamat*$_1$ `} else {` *folyamat*$_2$ `}` (ez egy **folyamat**)

A zárójelek természetesen nem kötelezőek, egyértelmű esetben elhagyhatók. A folyamatokból épített elágazásnál is elhagyható az `else` rész, ez esetben ez egy automatikus „`else {}`”-t (üres folyamatot) jelent.

2.6. Mintaillesztés

Mintaillesztés alkalmazásával eseteket választhatunk szét. Mintaillesztéskor azt vizsgálja a kiértékelő rendszer, hogy az aktuális paraméter értéke vagy alakja megfelel-e valamelyik mintának [1]. Az első illeszkedő mintához tartozó kifejezés értékelődik ki, ha ilyen nincs, akkor a program hibaüzenettel terminál. (Már fordítási időben is látszik, hogy minden lehetséges esetet lekezeltünk-e, vagy sem; ha valamelyik esetet nem kezeljük le, akkor figyelmeztetést (*warning*) kapunk.)

A minta egy speciális kifejezés, ami konstansokból, változókból, konstruktorokból, és a „_” jelből (joker, vagy mindenesjel) állhat. A változó és a joker bármire illeszkedik. A változó illesztéskor fel is veszi az illesztett kifejezés értékét, és ezt használhatjuk a hozzá tartozó törzsben is (ha a változó már korábban létezett, akkor a mostani a korábbi jelentést eltakarja). A joker is mindenre illeszkedik, de ebben az esetben az illesztett kifejezésre nem hivatkozhatunk (a joker nem használható kifejezésben).
További megkötés, hogy a mintának *lineáris*nak kell lennie: egy változó legfeljebb egyszer szerepelhet egy mintában [5].

Szintaxis:

```
match kifejezés with
     minta1 -> kifejezés1
   | minta2 -> kifejezés2
   …
   | mintan -> kifejezésn
```

Az illesztendő *kifejezés*nek és a *minta*$_i$-knek ugyanolyan típusúnak kell lenniük, továbbá a *kifejezes*$_i$-k típusainak is egymással egyezniük kell.

Például a fibonacci feladat [4]:

```
let rec fib i =
   match i with
       1 -> 1
     | 2 -> 1
     | j -> fib (j-2) + fib (j-1);;
```

Egy mintában több alternatív mintát is összevonhatunk (pl.: `'a' | 'b'`) és intervallumot is megadhatunk (pl.: `'a' .. 'z'`).
A mintához feltétel is adható (őrzött minta: *`minta`* `when` *`feltétel`* `->` *`kifejezés`*): csak akkor értékeli ki a *`kifejezés`*t, ha a *`minta`* illesztődik és a *`feltétel`* is igaz; ha vagy a *`minta`* nem illeszkedik, vagy a *`feltétel`* nem igaz, akkor folytatódik tovább a mintaillesztés.

Nagyon gyakori, hogy a függvény törzse egyetlen `match`-ból áll, ezért az ilyen eseteket egyszerűbb szintaxissal is leírhatjuk, például [4]:

```
let rec fib = function
      1 -> 1
    | 2 -> 1
    | i -> fib (i - 1) + fib (i - 2);;
```

Ebben az esetben a függvénynek azt a paraméterét, amit a mintaillesztésben vizsgálunk nem is kell felsorolni a függvénynevet követő paraméterek között. Ha a függvénynek több paramétere is van, akkor a rendszer a mintaillesztésben résztvevő paramétert tekinti az utolsó paraméternek.

Ha a függvényre csak egy mintánk van, akkor a következő szintaxissal is leírhatjuk:
`let` *`fvnév`* *`minta`* `=` *`érték`*. Például nézzük egy pár (lásd később) első elemét visszaadó függvényt:

```
let fst (x, _) = x;;
```

2.7. Típusrendszer

JoCaml-ben minden értéknek egy jól meghatározott típusa van, azonban általában sehol sem kötelező az explicit típusdeklaráció, a rendszer kikövetkezteti a típust az érték használatából.

Ha egy értéket – típus szerint – rossz környezetben használunk, akkor hibaüzenetet kapunk (még fordítási időben); nincsen automatikus típuskonverzió (se bővítő, se szűkítő).

Típus kényszerítés (*type constraint*): bár a nyelv a legtöbb típust levezeti, néha hasznos lehet megadni a típust: `(`*`kifejezés`* `:` *`típus`*`)` [4]

Típus korlátozás (*type coercion*): egy érték típusát korlátozhatjuk (csak gyengítés) egy *szupertípus*ra: `(`*`kifejezés`* `:>` *`típus`*`)` – a *`kifejezés`*-t *`típus`* típusúra korlátozza; vagy `(`*`kifejezés`* `:` *`típus`*$_1$ `:>` *`típus`*$_2$`)` – a *`kifejezés`*-t *`típus`*$_1$-ről *`típus`*$_2$-re korlátozza. Például korlátozhatunk egész számok listájáról tetszőleges elemet tartalmazó listára; egy osztályról az ősosztályára, vagy a nem definiált osztályra [4] (ezekről majd később bővebben).

2.7.1. Polimorfizmus

Az olyan függvények, amelyek több, egymástól különböző típusú paraméterre is alkalmazhatók az úgynevezett *polimorf* függvények.

A polimorfizmus többféle változatban fordulhat elő a programozásban [1]:

- **Paraméteres polimorfizmus**: Egy *polimorf név* **egyetlen** olyan algoritmust azonosít, amely tetszőleges típusú paraméterre alkalmazható.
  A függvény megvalósítása nem függ a típustól, ezért definíciója is megadható típus-független formában. A függvény típusában egy *típusváltozó* található, ez fejezi ki, hogy a függvény polimorf, bármilyen konkrét típusra alkalmazható. Például (lista első elemét visszaadó függvény):

  ```
  # let head = function
  #     x::xs -> x;;
  val head : 'a list -> 'a = <fun>
  ```

Ez a függvény egy tetszőleges típusú elemekből álló listának (`'a list`) megadja az első elemét (ami természetesen olyan típusú, amilyen típusú elemekből áll a lista). Az „`'a`” egy típusváltozó, a `head` egy polimorf függvény.

- **ad-hoc polimorfizmus** (más néven azonosítók túlterhelése): Egy *többszörösen terhelt név* **több különböző** algoritmust azonosít: ahány típusú paraméterre alkalmazható, annyifélét. Például a „+” műveletet is számos konkrét típus esetén használhatjuk. Az összeadás megvalósítása azonban minden konkrét típus esetén más és más (például más egész számok estében, mint mátrixok esetében). A polimorfizmus ebben az esetben tehát más jellegű, itt valójában több, egymástól különböző függvényre használunk azonos jelölést, az operátor szimbólumát többszörösen használjuk.
- **öröklődéses polimorfizmus**: részletesebben 4. A JoCaml nyelv objektum-orientált nyelvi elemei című részben.

A JoCaml az ad-hoc polimorfizmust **nem** támogatja. Bár a nyelv megenged egy függvénynévre több definíciót, de ezt mégsem nevezhetjük polimorfizmusnak, mert ebben az esetben az új definíció eltakarja a régit. Az azonosítók többszörös használata viszonylag gyakori oka az automatikus típuslevezetés sikertelenségének, a JoCaml éppen ezért korlátozza ezt a lehetőséget [1].

A **paraméteres polimorfizmust** azonban **támogatja** a nyelv. A standard könyvtárak több előre definiált polimorf típust, illetve rajtuk értelmezett polimorf függvényeket tartalmaznak (például tetszőleges elemet tartalmazó lista műveletei); de a programozó maga is készíthet polimorf függvényeket.

Azonban típusváltozó használata esetén nincs lehetőség előírni a típusra fontos tulajdonságok meglétét. Például, ha készítünk egy rendező függvényt, ami egy tömb elemeit rendezi, akkor a típusról csak azt használjuk ki, hogy értelmezve van rajta a rendezés. Azonban – ellentétben például a Clean-nel – nem tudjuk a fordító számára előírni, hogy a függvényt nem lehet tetszőleges típussal meghívni, hanem csak az olyan típusok jók, amelyeken értelmezve van rendezés. Ha a fenti függvényünket egy olyan típussal példányosítjuk, amin nincs rendezés (például `int -> int` típusú függvények), akkor futás közben `Invalid_argument` kivétel keletkezik.

Érték korlátozás (*value restriction*): létezik olyan típusváltozó is (*gyenge típusváltozó*), ami az első hivatkozás előtt még bármilyen típusú lehet, de az első hivatkozáskor a használt típust felveszi, és utána már csak ezzel a típussal használható. A ilyen típusváltozó neve „`'_`”-jel kezdődik (például `'_a`). Erről majd később (3.1.2. Referencia és az 5.5. A csatornák tulajdonságai részben) lesz szó bővebben.

2.8. Típuskonstrukciók (összetett típus)

Összetett típusokat már meglévő típusokból, mint komponensekből hozhatunk létre [1].

2.8.1. Rendezett n-es

A rendezett n-es n db. elemet tartalmaz, az elemek típusa tetszőleges, akár egymástól különböző is lehet. A rendezett n-eseket kerek zárójelben adjuk meg, vesszővel elválasztva. (A kerek zárójel bizonyos egyértelmű esetekben elhagyható, de például mintaillesztéskor mindig kötelező.) A rendezett n-es típusa: $típus_1$ * $típus_2$ * … * $típus_n$

Például nézzük a következő rendezett hármast (`p`):

```
# let p = 1, "hello", 'c';;
val p : int * string * char = 1, "hello", 'c'
```

A rendezett n-eseket komponenseire bonthatjuk:

```
# let x, y, z = p;;
val x : int = 1
val y : string = "hello"
val z : char ='c'
```

A rendezett n-esek komponenseit mintaillesztéssel is elérhetjük, például a következő függvény visszaadja egy rendezett hármas középső elemét:

```
# let mid (_, x, _) = x;;
val mid : 'a * 'b * 'c -> 'b = <fun>
# mid p;;
- : string = "hello"
```

Az n-esek közül a *párok*hoz több előre definiált függvény létezik, például a `fst` vagy a `snd` szelektorfüggvények.

2.8.2. Lista

A lista tetszőleges számú, de azonos típusú elemek sorozata. A listákat az elemek típusa szerint megkülönböztetjük (*paraméteres típus*), így beszélhetünk például `int list`-ről, `string list`-ről stb. A lista egy *rekurzív* lineáris adatszerkezet, amely vagy az üres lista, vagy egy elemből és az elemet követő listából áll.

A lista típusnak két konstruktora van:

- `[]` – üres lista
- e_1`::l` – új listát hoz létre egy elemből és egy (esetleg üres) listából, az új lista első eleme e_1 lesz, ezt követi az `l` lista

A listát szögletes zárójelben is megadhatjuk, az elemeket pontosvesszővel kell elválasztani: $[e_1;\ e_2;\ \dots\ ;\ e_n]$, azonban ez csak egy alternatív jelölésmód az e_1 `::` e_2 `:: … ::` e_n `:: []` szintaxisra.

A lista elemeihez például mintaillesztéssel lehet hozzáférni. A következő függvény összeadja egy lista elemeit:

```
# let rec sum = function
#       []     -> 0
#     | i :: l -> i + sum l;;
val sum : int list -> int = <fun>
```

A listához több előredefiniált függvény is létezik (a `List` modulban):

- Lista hosszának lekérdezése: `List.length`
- Két lista összefűzése: `List.append` *lista*$_1$ *lista*$_2$ (ugyanaz, mint a „`@`” infix operátor)
- Iterátor: `List.map` *`f lista`* (a *`lista`* minden elemére alkalmazza az *`f`* függvényt, majd az eredményekből képzett listát adja vissza; például: `List.map inc [2; 3; 4]` » `[3; 4; 5]`)

2.8.3. Rekord (direktszorzat)

A rendezett n-esek mellett névvel ellátott mezőkkel rendelkező rekordokat is használhatunk. Ha egy rendezett n-es nagyon sok komponensből áll, akkor nagyon körülményes egyes elemeinek mintaillesztéssel történő elérése. Ha a direktszorzat típuskonstrukció értékeinek elemeit névvel ellátott szelektorfüggvénnyel azonosíthatjuk, akkor a programszöveg áttekinthetőbb és könnyebben módosítható lesz [1].

Rekord-típus deklarációja: `type` *név* `= {` *mezőnév*$_1$ `:` *típus*$_1$`; …;` *mezőnév*$_n$ `:` *típus*$_n$ `}`, például.:

```
type szemely =
   { nev:      string;
     magassag: float;
     tel_szam: string
   };;
```

Rekord készítése: {*mezőnév*$_{i1}$ = *kifejezés*$_{i1}$; …; *mezőnév*$_{in}$ = *kifejezés*$_{in}$} (minden mezőt meg kell adni, de a mezők sorrendje tetszőleges), például:

```
let moni =
   { nev      = "Meszaros Monika";
     magassag = 1.7;
     tel_szam = "1-222-333"
   };;
```

Az egyes mezőkhöz közvetlenül is hozzá lehet férni: *rekordnév*.*mezőnév*, például: moni.nev;;

Mintaillesztés rekordoknál: elegendő, ha a mintában csak azokat a rekordmezőket adjuk meg, amelyekre ténylegesen szükségünk van. A minta olyan rekordra illeszkedik, aminek vannak a mintában szereplő mezői, és ezeknek a mezőknek az értéke illeszkedik a mintabeli értékre (azonban a rekord a mintában felsorolt mezőkön kívül más mező(ke)t is tartalmazhat). Például:

```
let nagysag {magassag = x} =
   if x < 1.5 then "alacsony" else (if x < 1.9 then "kozepes" else "magas");;
```

Megjegyzés: már az előző példán is látszik, hogy ha két olyan rekordunk lenne, aminek van magassag nevű mezője, akkor a rendszer nem tudta volna (nem is lehetett volna) meghatározni a nagysag függvény típusát. Ha több rekordnak is ugyanolyan nevű mezője van, akkor az egyik el fogja takarni a másikat (még típuskényszerítés esetén is).

2.8.4. Algebrai adattípus

Típusdeklaráció: type *név*$_1$ = *típusdef*$_1$ [and … [and *név*$_n$ = *típusdef*$_n$]];;

Algebrai adattípus (más néven unió, vagy variáns rekord) egy típus több esetét reprezentálja, például egy bináris fa *node*-okat tartalmaz, egy node pedig ***vagy*** egy levél **vagy** egy belső pont. Az algebrai adattípus deklarációjában egyidejűleg vezetünk be egy új típust (típuskonstrukciót) és az adatkonstruktorait. Kivétel nélkül minden az adott típushoz tartozó érték a típus deklarációjakor megadott valamelyik adatkonstruktor-függvény alkalmazásával áll elő. A típusdefiníció lehet rekurzív is [1].

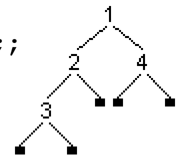
Nézzünk egy példát: tetszőleges típusú (de azonos) elemekből álló bináris fa [4]:

```
type 'a btree=
     Node of 'a * 'a btree * 'a btree
   | Leaf;;
```

A btree egy 1-változós típuskonstroktor, ami egy tetszőleges típusú (ezt jelöli az „'a”) elemekből álló bináris fa típust hoz létre. (A típuskonstruktorok magasabbrendű típusnak tekinthetők, melyek egy típusból (vagy típusokból) egy másik típust hoznak létre.)
A típuskonstrukció adatkonstruktorai a Node és a Leaf. A Node adatkonstruktor-függvény háromváltozós, első paramétere egy 'a alaptípusú érték, amit a fa csúcsában tárol, második és harmadik paramétere egy-egy 'a btree típusú érték; az adatkonstruktor-függvény értéke pedig 'a btree típusú (a Node típusa: 'a * 'a btree * 'a btree -> 'a btree). A Node jelképezi a fa egy belső pontját, ebben a belső pontban van egy elem, és tudjuk, hogy melyek az elemhez tartozó jobb-, illetve baloldali részfák. A Leaf egy konstans adatkonstruktor-függvény, a fa levelét reprezentálja (a fa lezárása, mely értéket nem tartalmaz).

Az adatkonstuktorok használata, példa:

```
let my_tree =
   Node (1, Node(2, Node (3, Leaf, Leaf), Leaf), Node(4, Leaf, Leaf));;
```



Az adatkonstruktorok – bár függvénynek tekinthetők –, mégsem függvények: az adatkonstruktort nem használhatjuk értékként (például nem lehet függvény paramétere), és az adatkonstruktor használatakor mindig meg kell adni az összes paraméterét.

Az, hogy egy `btree`-beli elem hogyan épül fel, rekonstruálható mintaillesztés segítségével. Például a következő függvény megszámolja a fa belső pontjait:

```
let rec pont_szamol = function
     Leaf -> 0
   | Node (_, bal, jobb) -> pont_szamol bal + pont_szamol jobb + 1;;
```

A legegyszerűbb algebrai adattípus a ***felsorolási típus***, melynek adatkonstruktorai mind konstansok. Például:

```
type szinek = Piros | Zold | Sarga | Kek;;
```

A konstruktorok *hatóköre és láthatósága* kiterjed a teljes modulra, amely a deklarációt tartalmazza. Ha egy adatkonstruktorra több definíció is van, akkor a legfrissebb érvényes, ez eltakarja az előző definíció(ka)t.

Megjegyzés: egy típuskonstruktornak több paramétere is lehet. Például készíthetünk egy olyan listát, ami kétféle típusú elemekből állhat (`type ('a, 'b) list2 = …`).

Néhány gyakori beépített algebrai adattípus [4]**:**

- ```
  type bool =
        true
     | false;;
  ```
- Tetszőleges (de azonos) elemekből álló lista, ami vagy az üres lista, vagy egy elem és egy lista összefűzése (a „`::`” operátor segítségével):
  ```
  type 'a list = [] | :: of 'a * 'a list;;
  ```
- NIL „helyett” (JoCaml-ben nincs NIL érték, ami tetszőleges típusú elemnek értékül adható lenne):
  ```
  type 'a option =
        None
     | Some of 'a;;
  ```
  Így a típusnak értékül adható üres elem is, viszont a jó elemek is „becsomagolva” jönnek.

Megjegyzés: a `true`-t, a `false`-t, a `[]`-t és a `::`-t nagybetűsként kezeli a rendszer.

2.8.5. Típus-szinonimák

Típus-szinonima deklarációja nem vezet be új típust, a szinonima ekvivalens az eredeti típuskifejezéssel, szintaktikai rövidítésnek tekinthető. Típus-szinonimára vonatkozó rekurzió, illetve példányosítás nem megengedett [1].

Szintaxis: `type` *`név`* `=` *`típuskifejezés`*

Például:

- ```
  type ujint = int;;
  let (x: ujint) = 3;;
  x+4;;
  ```
- ```
  type 'a olyan_par_aminek_egyik_tagja_int = int * 'a;;
  ```
  ```

2.9. Kivételkezelés (szekvenciális program esete)

A programozó saját kivételeket definiálhat, melyek lehetnek konstans függvények vagy konstruktor-függvények. A deklarált kivételeket ki lehet váltani, és a hívó a kiváltott üzenetekre kivételkezelő függvény kiértékelésével válaszolhat.

Kivételeket az `exception` kulcsszóval lehet **deklarál**ni, és a szintaxisának formája ugyanaz, mint a konstruktor-deklarációnak: `exception` *`Kivételnév`* `[of` *`típus`*`];;`. (Például: `exception Abort of int * string;;`) A kivétel tehát paraméterezhető. Megjegyzés: a kivételek *monomorfak*: típusváltozót nem tartalmazhatnak.

A kivétel **kiváltás**a a `raise` függvénnyel történik. (pl.: `raise (Abort (42, "szöveg"));;`) A hiba kiváltásakor a kivétel a típusának megfelelő értéket kap paraméterként, az értéket a kivétel „becsomagolja”. A kivételkezelőben ezek az értékek felhasználhatók, akár tovább is adhatók.
Amikor egy kivétel keletkezik, akkor az aktuális végrehajtás abbamarad, és az irányítás átadódik a legközelebbi kivételkezelőnek, aki vagy lekezeli a kivételt, vagy továbbadja a következő kivételkezelőnek.

Tetszőleges **kifejezés**hez kapcsolhatunk **kivételkezelő**t, melyet a `try` kulcsszóval lehet létrehozni, a formája ugyanolyan, mint a mintaillesztésnek [4]:

```
try kifejezés with
      minta1 -> kifejezés1
    | minta2 -> kifejezés2
    …
    | mintan -> kifejezésn
```

Először a *`kifejezés`* értékelődik ki. Ha a kiértékelés során kivétel nem keletkezik, akkor a *`kifejezés`* értéke lesz az egész `try` kifejezés eredménye. Azonban, ha a *`kifejezés`* kiértékelése során kiváltódik egy kivétel, akkor a kivétel illesztődik a $minta_1$, …, $minta_n$ mintára. Az első illeszkedő mintához tartozó kifejezés kiértékelődik, ez lesz az egész `try` eredménye. Ha a kivétel egyik mintára sem illeszkedik, akkor a kivétel továbbadódik a következő kivételkezelőnek. Ha a kivételt egyik kivételkezelő sem tudja lekezelni, akkor a program végrehajtása abbamarad, és a rendszer kiírja a kivételt a képernyőre.

Megjegyzés:

- A *`kifejezés`*-nek és a $kifejezés_i$-knek ugyanolyan típusúnak kell lenniük.
- A kivételkezelőben is kiváltható kivétel.

Példa:

A következő példában [4] egy *hash-táblát* hozunk létre, a tábla elemeit listában tároljuk, egy elem pedig egy (kulcs, érték) pár:

```
# let lista = [1, "Hello"; 2, "World"];;
```

A `keres` függvény megkeresi az adott kulcshoz tartozó értéket, ha nem találja, akkor egy `Nem_talalhato` kivétel keletkezik:

```
# exception Nem_talalhato;;
# let rec keres kulcs = function
#       (k, e) :: l -> if k = kulcs then
#                          e
#                      else
#                          keres kulcs l
#     | [] -> raise Nem_talalhato;;
val keres : 'a -> ('a * 'b) list -> 'b = <fun>
```

Használat:
```
# let ertek_kiir i =
#   let j = string_of_int(i) in
#   try "A(z) "^j^" kulcshoz tartozó érték: "^ (keres i lista) with
#      Nem_talalhato -> "A(z) "^j^"kulcshoz nem tartozik érték"
# ;;
val ertek_kiir : int -> string = <fun>
# ertek_kiir 1;;
- : string = "A(z) 1 kulcshoz tartozó érték: Hello"
# ertek_kiir 3;;
- : string = "A(z) 3 kulcshoz nem tartozik érték"
```

Előredefiniált kivételek [6]**:**

- `exception Match_failure of (string * int * int)`
  A használt kifejezés nem illeszkedik egyik mintára sem. A paraméterek a hiba helyét jelzik a programban: *`fájl_név`*, *`match_első_karaktere`*, *`match_utolsó_karaktere`*
- `exception Invalid_argument of string`
  Függvény értelmetlen paraméterrel való meghívása.
- `exception Failure of string`
  Könyvtári függvények használják, hogy jelezzék, hogy az adott paraméterre nem-definiáltak. A `failwith : string -> 'a` egy beépített függvény, amely a megadott paraméterrel kivált egy `Failure` kivélt (`let failwith s = raise (Failure s);;`)
- `exception Not_found`
  Kereső-függvények használják, ha az elem nem található.
- `exception Out_of_memory`
  Elfogyott a memória.
- `exception Stack_overflow`
  A verem elérte a maximális méretét.
- `exception Sys_error of string`
  Input / output függvények váltják ki operációs-rendszer-hiba esetén.
- `exception End_of_file`
  Input függvények jelzik, ha elérték a fájl végét.
- `exception Division_by_zero`
  Nullával való osztás.

Ezeket a kivételeket az előredefiniált (könyvtári) függvények használják, de a programozó maga is kiválthatja őket. A kivételek `exn` típusúak.

Kivételkezelés folyamatok esetén:

Eddig csak a kifejezések kivételkezeléséről volt szó (ugyanis a `try …` csak kifejezést tartalmazhatott), sőt a kivételkezelésnek ez a módja teljesen csak akkor érvényes, ha a program szekvenciális.

Ha a program párhuzamos, akkor a kivételkezelés némileg más: például ha a program egyik párhuzamosan futó részén olyan kivétel keletkezik, ami nem került lekezelésre, akkor ***ezen rész*** kiértékelése nem folytatódhat tovább. Azonban nem feltétlenül kell az egész programnak terminálnia, a többi párhuzamosan futó programrész kiértékelése esetleg még folytatódhat tovább.

A folyamatoknál szintén felmerülhetnek problémák. A folyamatok ugyanis aszinkron módon hajtódnak végre: amikor elindítunk egy folyamatot, akkor az egy új *szálon* indul el, és kiértékelése a program többi részével párhuzamosan történik, a kiértékelés eredményére pedig nem vár senki.

Mi történik, amikor egy ilyen folyamat kivételt vált ki? A legfőbb kérdés: ki kapja a kivételt? JoCaml-ben ezt a problémát nagyon egyszerűen lerendezték: mivel a folyamat eredményére úgysem vár senki, így a folyamat (és csak a folyamat) kiértékelése abbamarad, és a folyamatban keletkezett kivételt nem lehet elkapni.

A párhuzamos program kivételkezeléséről bővebben a 5.6. Kivételkezelés párhuzamos program esetében című részben lesz szó.

2.10. Absztrakt adattípus megvalósítása JoCaml-ben: fordítási egységek

Az **absztrakt adattípus** (Abstract Data Type) egy olyan program-egység, ami egy adattípust definiál a hozzá tartozó függvényekkel (metódusokkal) együtt.

Az absztrakt adattípus két részből áll (mindkettő külön fordítási egység):

- **Interfész**: ez tartalmazza az elérhető adatstruktúrák és metódusok deklarációját (specifikációját), az importálásra szánt azonosítókat és típusukat. Az interfész fájlnak „`mli`” kiterjesztésűnek kell lennie.
- **Implementáció**: ez tartalmazza az interfészben deklarált struktúrák és egyéb azonosítók konkrét implementációit és a metódusok megvalósításait. Az implementáció rejtett. Az implementációs fájl nevének meg kell egyeznie az interfész nevével, a kiterjesztésnek „`ml`”-nek kell lennie.

Az absztrakt adattípus használatának tulajdonságai, előnyei [3]**:**

- **Modularitás**: az egyes típusokat önálló fordítási egységekben lehet megvalósítani. Ez biztosítja a típusok újrafelhasználhatóságát, valamint a hatékony programfejlesztést, hiszen az egyes modulok könnyedén átvihetők más programokba, és a különböző egységeket más-más programozó is fejlesztheti, nem zavarva egymás munkáját. Ha a programot több (értelmes) modulra bontjuk, akkor áttekinthetőbb, könnyebben érthető és jobban módosítható lesz.
- **Adatelrejtés és enkapszuláció (egységbezárás)**: A nyelv támogatja a reprezentáció elrejtését. Ezen eszköz segítségével a nyelv maga biztosítja, hogy az adott típus használója csak a specifikációban megadott tulajdonságokat használhassa ki. Ez a megszorítás lehetővé teszi a reprezentáció, illetve az implementáció magváltoztatását anélkül, hogy a változások a programban felfelé gyűrűznének (ha a metódusok viselkedését megőrizzük).
- **Specifikáció és implementáció szétválasztása külön fordítási egységbe**: az adott típust használó más modulok a típus-specifikáció birtokában elkészíthetők, függetlenül a tényleges implementációtól.
- **Konzisztens használhatóság**: a felhasználói és a beépített típusok nem különböznek a használat szempontjából.
- **Generikus programsémák támogatása**: az absztrakt adattípus tetszőleges típussal (illetve típusokkal) paraméterezhető.

Megjegyzés: JoCaml-ben minden programfájl úgy viselkedik, mint egy absztrakt modul; ha nincs hozzá tartozó interfész fájl, akkor az alapértelmezett interfész tartalmaz minden – a fájlban definiált – típust és függvényt.

2.10.1. Interfész

Az interfész három részből áll:

1. Adattípusok deklarációja;
2. Metódus-deklarációk;

3. A modulban használt kivételek.

Csak az interfészben deklarált típusok és függvények lesznek láthatóak más modulokban.

Típusdeklaráció:

A típusdeklaráció kétféle lehet: absztrakt, vagy átlátszó [4].

- Az **absztrakt típusdeklaráció** anélkül deklarálja a típust, hogy megadná a definícióját. Például a `type 'a t` egy absztrakt (és polimorf) típusdeklaráció, mert a definíció specifikálatlanul maradt. Ebben az esetben a konkrét típusdefiníció – ami az implementációs részben van – nem látszik más programokban, ezért más programokból csak az interfészben deklarált metódussokkal lehet használni a típust.
- **Átlátszó definíció**nál a struktúra típusát teljesen megadjuk. Például a fa adattípus egy átlátszó definíciója:

  ```
  type 'a t =
         Node of 'a * 'a t * 'a t
       | Leaf
  ```

  A struktúra más programokban is látszik, így más programok – a metódusok kihagyásával – közvetlenül is használhatják a típust (példánkban a fa elemei kívülről is közvetlenül elérhetők).

  Az átlátszó definíció több szempontból is rossz lehet: egyrészt, ha valamikor megváltoztatjuk a reprezentációt (például kiegyensúlyozott fára), akkor minden olyan programot is módosítani kell, ami a fa elemeit közvetlenül használja; másrészt, az adatstruktúránkról bizonyos invariánsokat tételezhetünk fel (például, hogy a fában az elemek rendezettek), és ha a definíció átlátszó, akkor más programok megsérthetik az invariánsokat, ami nehezen felderíthető hibákhoz vezet.

Metódusdeklaráció:

Az interfésznek tartalmaznia kell az összes, más programokból is látható függvény és érték *szignatúrá*ját (típusdeklarációját).

Szignatúra: `val` *`név`* `:` *`típus`*, például az összeadás szignatúrája: `val (+) : int -> int-> int`.

2.10.2. Implementáció

Az implementációs rész a következő részekből épül fel:

1. A modul által használt adattípusok;
2. Metódus-definíciók;
3. A modul által használt kivételek.

Típusdefiníció:

Meg kell adni az interfészben lévő összes típus definícióját. Az implementáció az interfészben felsorolt típusokon kívül más típusokat is definiálhat, ezen típusok azonban – mivel csak az implementációban szerepelnek – „kívülről” nem látszanak (*privát*).

Metódusdefiníció

Minden, az interfész részben deklarált metódust implementálni kell, de új függvényeket is megadhatunk (ezeket azonban más programokból nem lehet elérni).

2.10.3. Példa: halmaz megvalósítása

Készítünk egy azonos típusú elemekből álló, véges halmazt megvalósító absztrakt algebrai adattípust.

Az interfész fájlban (`fset.mli`) deklaráljuk a halmaz absztrakt adattípust:

```
type 'a set
```

Megjegyzés: ez az absztrakt típusdefiníció polimorf, mert paraméterezve van egy `'a` típusváltozóval.

A metódusok deklarációja: üres halmaz létrehozása (`empty`), halmazhoz tartozás lekérdezése (`mem`), elem berakása a halmazba (`insert`):

```
val empty : 'a set
val mem : 'a -> 'a set -> bool
val insert : 'a -> 'a set -> 'a set
```

Az implementációs fájl (`fset.ml`): a halmazt rendezett bináris fával valósítjuk meg. A típus tényeleges megadása a következő:

```
type 'a set =
     Node of 'a * 'a set * 'a set
   | Leaf
```

A metódusok megadása:

```
let empty = Leaf

let rec mem x = function
     Leaf -> false
   | Node (e, bal, jobb) ->
        if      x = e  then  true
        else if x < e  then  mem x bal
        else                 mem x jobb

let rec insert x = function
     Leaf -> Node (x, Leaf, Leaf)
   | Node (e, bal, jobb) ->
        if      x = e  then  Node (e, bal, jobb)
        else if x < e  then  Node (e, insert x bal, jobb)
        else                 Node (e, bal, insert x jobb)
```

2.10.4. Az absztrakt adattípus használata más programból

Ha más programból akarunk használni egy modult, akkor a modul interfész részében deklarált típusokra és metódusokra *`Modulnév.név`* formában hivatkozhatunk (a *`Modulnév`* a modul-fájl neve, melynek az első betűje nagy, a többi kicsi), például `Fset.empty`.
Ha nem akarjuk mindig kiírni a modulnevet, akkor az `open` *`Modulnév`* segítségével „megnyithatjuk” a modult (egész konkrétan az interfész részét): ezután már „*`Modulnév.`*” nélkül is hivatkozhatunk a modul elemeire. Ha névütközés van (például megnyitunk két modult: egy halmazt és egy listát; és mindkettő tartalmaz `insert` függvényt), akkor a legutoljára megnyitott modulban lévő elemek eltakarják a többi, ugyanilyen nevű elemet. Ebben az esetben az eltakart elemekre csak *`Modulnév.név`* formában hivatkozhatunk [4] (ez a forma tehát használható megnyitott modulok esetén is).

Példa a használatra (`teszt.ml`):

```
# print_string "Most keszitunk egy halmazt: {alma, korte}\n";;
# let halmaz = Fset.insert "korte" (Fset.insert "alma" Fset.empty);;
#
# print_string "Az alma benne van?";;
# if Fset.mem "alma" halmaz
#   then print_string " igen\n"
#   else print_string " nem\n";;
val halmaz : string Fset.set
-> Most keszitunk egy halmazt: {alma, korte}
-> Az alma benne van? igen
```

Megjegyzés:
A lefordított modulokat össze kell szerkeszteni egyetlen futtatható fájllá. JoCaml-ben – ellentétben például a C-vel – nincs „`main`” függvény (C-ben a program végrehajtása a `main` függvény végrehajtását jelenti). Egy JoCaml program végrehajtásánál a rendszer a programban lévő összes mondatot szépen sorrendben kiértékeli. Ha több programot szerkesztünk össze, akkor a fájlok az összeszerkesztés sorrendjében értékelődnek ki [4].

2.11. Késleltetett kiértékelés

JoCaml-ben a kifejezések kiértékelése mohó módon történik. Azonban a `Lazy` modul használatával lehetőségünk van késleltetett számítás megvalósítására is.

A `lazy` *`kifejezés`* kifejezés egy `Lazy.t` típusú *`v`* értéket ad vissza, ami egységbe zárja (*encapsulate*) a *`kifejezés`* számítását. A *`kifejezés`* a programnak ezen a pontján még nem értékelődik ki. A kiértékelés az első `Lazy.force` *`v`* alkalmazásakor történik meg: a `Lazy.force` *`v`* visszaadja a *`kifejezés`* aktuális értékét. A `Lazy.force v`-re történő későbbi hívások nem értékelik ki újra a *`v`*-t [6].

3. A JoCaml nyelv imperatív nyelvi elemei

3.1. Módosítható típusok (típuskonstrukciók)

3.1.1. Rekord (direktszorzat)

A rekord mezőit alapértelmezésben nem lehet módosítani. Ahhoz, hogy egy mező módosítható legyen a következő módon kell deklarálni: **`mutable`** *`mezőnév : típus`*. Az ilyen típusú mezők módosítása a *`rekordnév.mezőnév <- új_érték`* szerkezettel lehetséges [4] (ez az értékadás mindig egy `()`-t ad vissza).

3.1.2. Referencia

A referencia egy frissíthető változóra hivatkozó mutató.

A referenciákat típusuk szerint megkülönböztetjük; tetszőleges típusból létrehozhatunk referenciát: `'a ref` (például lehet: `int ref`, `string ref`, `int list ref`). Típus nélküli referencia nincs.

Referenciát a `ref` függvénnyel lehet **létrehozni**: `ref` *`kezdeti_érték`*, például:

```
# let i = ref 1;;
val i : int ref = {contents=1}
```

Beépített **operátorok**: „`!`”: dereferencia operátor (prefix), „`:=`”: értékadó operátor (infix). Az értékadás mindig `()`-t ad vissza. Példa:

```
# !i;;
- : int = 1
# i := 17;;
- : unit = ()
# !i;;
- : int = 17
```

Megjegyzés: az `'a ref` típus tulajdonképpen egy rekord, ami egyetlen módosítható mezőt tartalmaz [5]:

```
type 'a ref = {mutable contents : 'a};;
let (!) r = r.contents;;
let (:=) r x = r.contents <- x;;
```

Az előbbi ok miatt az összes rekordra értelmezett operátor és függvény használható a referenciákra is (például: `i.contents`).

Ahogy korábban ígértem mutatok egy példát az **érték korlátozás**ra (*value restriction*), tehát **a gyenge típusváltozó** használatának szükségességére:

A referencia típus paraméterezett, tehát bármilyen típusból készíthetünk referenciát. Azonban mi lesz, ha szintén paraméterezett típusból készítjük a referenciát, például `'a list`-ből? Nézzük a következő deklarációt: `let x = ref [];;`
Az `x` típusának elvileg `'a list ref`-nek kéne lennie (hiszen ez egy üres lista, még nem tudni, hogy milyen típusú elemeket fog tartalmazni), de ha ez így lenne, akkor mind a két következő utasítás helyes lenne: `x := 1 :: !x;; x := true :: !x;;`
Tehát ugyanannak a változónak a típusa egyszer `int list`, másszor `bool list`. Ez pedig ellentmond a JoCaml szigorú statikus típusosságának [4].

Ennek az ellentmondásnak az elkerülésére az `x` típusában egy gyenge típusváltozó fog szerepelni:

```
# let x = ref [];;
val x : '_a list ref = {contents = []}
```

Az `'_a` nem igazi típusváltozó, hanem egy „sima” típus, melynek értéke – egyelőre – ismeretlen. A típus értéke az első (deklaráció utáni) használatkor kiderül, ebben a pillanatban az `'_a` gyenge típusváltozó értéket kap, ez az érték (tehát a típus) később már nem változhat [4].

```
# x := 1 :: !x;;
- : unit = ()
# x;;
val x : int list ref = {contents = [1]}
# x := true :: !x;;
Characters 13-15:
This expression has type int list but is here used with type bool list
```

3.1.3. Tömb

A tömb azonos típusú elemekből álló fix méretű vektor. Listák használata során a sorozat elemeinek elérése csak úgy lehetséges, ha a listát az adott elemig bejárjuk. Tömbök esetén a kiértékelő rendszer címaritmetikát alkalmaz, így az elemeket konstans időben elérjük [1].

A tömböket a típusuk szerint különböztetjük meg. Tetszőleges típusból készíthetünk tömböt, például `int array`. A tömb mérete nem változtatható és már fordításkor eldől. Többdimenziós tömb nincs. A tömbben lévő elemek értéke változtatható (frissíthető).

A tömböt az `[|`*elem*$_1$`;` *elem*$_2$`;` …`;` *elem*$_n$`|]` szintaxissal **hozhatjuk létre**.
A tömb elemeihez a *tömbnév*`.(`*i*`)` szerkezettel lehet **hozzáfér**ni. A tömb-elemek indexelése 0-tól kezdődik, az utsó elem a „hossz-1”. Ha rossz indexre hivatkozunk `Invalid_argument` kivétel keletkezik.
A tömb elemeinek **új értéket lehet adni** a *tömbnév*`.(`*i*`) <-` *új_érték* értékadással (mindig `()`-t ad vissza).

Az `Array` könyvtár sok függvényt szolgáltat a tömbökhöz, például:

- Tömb készítése: `Array.create` *méret inicializáló_érték* (létrehozza az adott *méret*ű tömböt és a tömb minden elemének az *inicializáló_érték* lesz az értéke).
- Tömb méretének lekérdezése: `Array.length` *tömbnév*
- Tömbök összefűzése: `Array.append` *tömb*$_1$ *tömb*$_2$ (visszaad egy új tömböt, ami a *tömb*$_1$ és a *tömb*$_2$ konkatenációját tartalmazza).
- Tömb adott elemének lekérdezése: `Array.get` *tömb i* (≡ *tömb*`.(`*i*`)`).
- Tömb adott elemének módosítása: `Array.set` *tömb i x* (≡ *tömb*`.(`*i*`) <-` *x*).
- Iterátor: `Array.map` *f tömb* (a *tömb* minden elemére alkalmazza az *f* függvényt, majd az eredményekből képzett tömböt adja vissza).

A `Sys.argv` egy beépített `string`-eket tartalmazó tömb (`string array`), ami a parancs-sori paramétereket tartalmazza, az első paraméter a `Sys.argv.(1)`

3.2. Szekvencia

A szekvenciális végrehajtásnak nincs értelme a tisztán funkcionális nyelvekben: miért számítanánk ki az értéket, ha utána nem használjuk fel? Az imperatív nyelvekben a szekvenciális végrehajtás azonban igen gyakran használt a mellékhatásokkal való számításra [4].

Szekvenciát a „`;`”-vel hozhatunk létre (infix operátor). A *kifejezés*$_1$`;`*kifejezés*$_2$ **kifejezés** a következőt jelenti: a rendszer először kiértékeli a *kifejezés*$_1$-et, majd megsemmisíti az értéket (azonban a *kifejezés*$_1$ kiértékelése során mellékhatások előfordulhatnak), majd kiértékeli a

`kifejezés`$_2$-t. Az egész `kifejezés`$_1$`;kifejezés`$_2$ kifejezés értéke a `kifejezés`$_2$ kiértékelése során kapott érték.

Szekvenciát **folyamat**ok esetén is használhatunk: a `kifejezés;folyamat` egy **folyamat**ot ad meg [8]. A rendszer a `kifejezés` eredményét nem veszi figyelembe (a `kifejezés` természetesen szintén lehet szekvencia). A folyamatok szekvencia-operátor bal oldalán nem fordulhatnak elő, így egy szekvenciális láncnál mindig csak az ***utolsó*** lehet folyamat: `kif`$_1$`; …; kif`$_n$`; folyamat` (Megjegyzés: a `folyamat;kifejezés` nem „véletlenül” tilos: a folyamatok aszinkron módon hajtódnak végre és a kiértékelés végén értéket nem produkálnak, ezért nem tudható előre – az ütemezőtől függ –, hogy egy folyamat kiértékelése mikor ér véget; ezért értelmetlen olyat leírni, hogy a `folyamat` kiértékelése *után* értékeljen ki egy `kifejezés`-t.)

Megjegyzés: A `;` operátor nem lezáró (mint pl. a C-ben), hanem elválasztó (mint pl. a Pascalban). Sőt, a „`kifejezés`” és a „`kifejezés;`” ***teljesen*** mást jelentenek: míg az első esetben egy kifejezésről van szó, addig a második esetben már egy folyamatról. A rendszer ugyanis a „`kifejezés;`”-t úgy értelmezi, mint a `kifejezés` kifejezésből és az üres folyamatból (amit üres karakterlánccal („”) jelölünk) álló szekvenciát. Azonban mivel a szekvencia második tagja egy folyamat (az üres folyamat), így az egész „`kifejezés;`” egy folyamat lesz.

A szekvencia prioritása alacsonyabb, mint a legtöbb operátoré.

3.3. Ciklus

A ciklusok JoCaml-ben nem vezérlési szerkezetek, hanem kifejezések (a függvények és függvényalkalmazások a vezérlési szerkezetek). Mivel a nyelvben minden kifejezésnek egy jól meghatározott értéke van, ezért a ciklusnak is kell, hogy legyen visszatérési értéke, még abban az esetben is, ha a ciklusmag egyszer sem „fut le” (JoCaml-ben csak elöltesztelős ciklusok vannak, melyeknél ez az eset előfordulhat). Ebből már látszik, hogy a ciklus-kifejezés értéke csak a `()` lehet.

3.3.1. Elöltesztelős ciklus

Szintaxis: `while` `kifejezés`$_1$ `do` `kifejezés`$_2$ `done`. A `kifejezés`$_1$-nek logikai típusúnak kell lennie.

Szemantika: Legelőször a `kifejezés`$_1$ értékelődik ki; ha az érték hamis, akkor a ciklus befejeződik. Ha igaz, akkor kiértékelődik a `kifejezés`$_2$, majd a rendszer újra megvizsgálja a `kifejezés`$_1$-et, … A ciklus akkor fejeződik be, amikor a `kifejezés`$_1$ hamissá válik.

Példa: készítünk egy függvényt [4], ami összeadja egy tömb elemeit:

```
let osszead t =
   let i = ref 0 in
   let s = ref 0 in
   while !i < (Array.length t) do
      s := !s + t.(!i); i := !i + 1
   done;
   !s;;
```

3.3.2. Léptetős ciklus

Szintaxis: `for` `ciklusvált = kezd_érték` `to` `végérték` `do` `kifejezés` `done`
vagy `for` `ciklusvált = végérték` `downto` `kezd_érték` `do` `kifejezés` `done`

A `kezd_érték`-nek és a `végérték`-nek int típusúnak kell lennie.

Szemantika: Legelőször a *kezd_érték* és a *végérték* kifejezéseket értékeli ki a rendszer, majd a [*kezd_érték* .. *végérték*] intervallum által meghatározott szám-szor kiértékeli a *kifejezés*-t (a két határra is kiértékeli). Ha a `kezd_érték > végérték`, akkor a `kifejezés` egyszer sem értékelődik ki. A határok csak egyszer – a ciklus legelején – számítódnak ki.
A *ciklusvált* a ciklus lokális konstansa, itt deklarálódik, ezért nem kell korábban létrehozni (ha korábban már létezett ilyen nevű érték, akkor azt a ciklus idejére a *ciklusvált* eltakarja). A *ciklusvált* hatóköre és láthatósága csak a *kifejezés*-re terjed ki, a ciklus végrehajtása után megszűnik. A *ciklusvált* a ciklus minden lépésében felveszi a soron következő értéket a `kezd_érték`-től a `végérték`-ig (`downto` használata esetén fordítva). A ciklus lépésköze tehát csak `1`, vagy `-1` lehet.

Példa: a feladat ugyanaz, mint az előbb [4]: a tömb elemeinek összeadása:

```
let osszead t =
   let s = ref 0 in
   for i = 0 to (Array.length t)-1 do
      s := !s + t.(i)
   done;
   !s;;
```

3.4. Input és output

3.4.1. Fájlműveletek [4]

Az input / output függvények JoCaml-ben *csatorná*kon értelmezettek. Kétféle csatorna létezik: `in_channel` – amiről olvasni lehet, `out_channel` – amire írni lehet (Megjegyzés: ez a csatorna nem egyenlő azzal a csatornával, amiről majd a párhuzamos programozásról szóló részben lesz szó.)
Előredefiniált (eleve nyitott) csatornák:

`val stdin  :  in_channel` – standard input
`val stdout : out_channel` – standard output
`val stderr : out_channel` – standard error

A fájlokhoz úgy lehet hozzáférni, ha megnyitunk hozzájuk egy kommunikációs csatornát. Ha nem sikerül megnyitni a csatornát (például azért, mert a fájl nem létezik), akkor egy `Sys_error` kivétel keletkezik.

Fájl megnyitása olvasásra (input fájl):

- `open_in : string -> in_channel` (a függvény paramétere a fájl neve, visszatérési értéke pedig a fájlhoz kapcsolódó input csatorna)
- `open_in_gen : open_flag list -> int -> string -> in_channel` (kifinomultabb megnyitás; paraméterek: (1) kapcsolók listája – hogyan nyissa meg a rendszer a fájlt, (2) a Unix rendszerben a fájl a létrehozáskor milyen jogosultságokat kapjon, (3) a fájl neve; visszatérési érték: fájlhoz kapcsolódó csatorna)

Fájl megnyitása írásra (output fájl) – analóg módon:

- `open_out : string -> out_channel` (ha már létezett a fájl, akkor törli a tartalmát)
- `open_out_gen : open_flag list -> int -> string -> out_channel`

Az `open_in_gen` és az `open_out_gen` során használt kapcsolók (első paraméter) például a következők lehetnek (egyszerre több kapcsolót is megadhatunk):

- `Open_rdonly`: megnyitás olvasásra;
- `Open_wronly`: megnyitás írásra;
- `Open_append`: megnyitás hozzáfűzésre;

- `Open_creat`: létrehozza a fájlt, ha eddig nem létezett;
- `Open_trunc`: törli a fájl tartalmát (ha létezik)

Ha `open_in_gen` és az `open_out_gen` során egy fájlt hozunk létre (az `Open_creat` kapcsoló használatával), akkor a létrehozott fálj jogosultságát a függvény második paramétere (ami egész szám típusú) határozza meg. Ha a számot kettes számrendszerben nézzük, akkor az egyes bitek jelentése a következő:

tulajdonos jogai			tulajdonos csoportjának jogai			többiek jogai		
olvasás	írás	futtatás	olvasás	írás	futtatás	olvasás	írás	futtatás

Például, ha a jogosultságot jelző szám `489` (kettes számrendszerben `111101001`), akkor a tulajdonos tudja olvasni, írni és futtatni a fájlt; a tulajdonos csoportja csak olvasni és futtatni tudja, a többiek pedig csak futatni.

Fájlok lezárása:

- `close_in  :  in_channel -> unit`
- `close_out : out_channel -> unit`

Ha nem zárjuk le a fájlt, akkor az automatikus szemétgyűjtő (*garbage collector*) lezárja.

Írás a csatornára:

- `output_char : out_channel -> char -> unit` (kiírja a megadott csatornára a karaktert)
- `output_string : out_channel -> string -> unit` (szöveget ír ki)

Olvasás a csatornáról:

- `input_char : in_channel -> char` (karakter olvasása)
- `input_line : in_channel -> string` (egy sor beolvasása: az aktuális pozíciótól az első sorvégejelig olvas; a sorvégejelet azonban nem adja vissza)

Az olvasó függvnyek `End_of_file` kivételt váltanak ki, ha azért nem tudnak olvasni a fájlból, mert elérték a végét.

Egyéb függvények:

- Ha a csatorna egy normál fájl, akkor a fájlhoz tartozik egy fájlmutató (a következő írás és olvasás a fájlmutató által megadott pozíción fog történni). Az író és olvasó műveletek a beolvasott, illetve kiírt szövegnek megfelelően mozgatják a fájlmutatót (a beolvasott, illetve kiírt szöveg mögé). Azonban a **fájlmutató mozgatása** a következő függvények használatával is megtehető (a paraméter által adott sorszámú pozícióra állítja a fájlmutatót; ha a fájlt hozzáfűzésre nyitottuk meg, akkor a függvény hatástalan):
  ```
  seek_in  :  in_channel -> int -> unit
  seek_out : out_channel -> int -> unit
  ```
- Aktuális pozíció lekérdezése:
  ```
  pos_in  :  in_channel -> int
  pos_out : out_channel -> int
  ```
- A fájlban lévő karakterek számának lekérdezése:
  ```
  in_channel_length  :  in_channel -> int
  out_channel_length : out_channel -> int
  ```
- A csatornák puffereltek. Ezért (alapértelmezésben) csak a fájl lezárása után lehetünk biztosak abban, hogy a kiíró utasítások ténylegesen kiírták az adatokat. Az írás kikényszerítése:
  ```
  val flush : out_channel -> unit
  ```

3.4.2. Formázott szöveg kiíratása: a Printf modul

Formázott szöveg kiíratása a standard outputra: `printf` (ez a függvény használatában nagyban hasonlít a C nyelvbeli `printf` függvényre).

Használat: `printf` *`szöveg`* `[`*`paraméterek`*`]`. A *`szöveg`* tartalmazhat *escape-szekvenciá*t (pl.: `\n`) is. A *`szöveg`*-ben ezen kívül elhelyezhetünk *konverziós karakter*eket (pl.: `%s`), ami azt jelenti, hogy ezen a helyen egy – jelen esetben string-típusú – érték áll. Ez(eke)t az érték(ek)et a paraméterek között kell megadni (olyan sorrendben, amilyen sorrendben a hozzájuk tartozó konverziós karakterek voltak a *`szöveg`*-ben).

Lehetséges konverziós (vezérlő) karakterek:

konverziós karakter	milyen típust vár	Hogyan írja ki
`d`	int	Előjeles egész
`u`	int	Előjel nélküli egész (negatív szám esetén túlcsordulás történik)
`x` vagy **`X`**	int	Az egész számot hexadecimálissá konvertálja („`x`”-nél a hexa számban lévő betűk kicsik lesznek, „`X`”-nél pedig nagyok)
`f`	float	Lebegőpontos szám *`egész_rész.tört_rész`* alakban
`e`	float	Lebegőpontos szám normál-alakban
`b`	bool	`true`-t vagy `false`-t ír ki
`c`	char	Kiírja a karaktert
`s`	string	Kiírja a szöveget

Például: `Printf.printf "Szám: %d\nSzöveg: %s" 42 "bla-bla";;`

A Printf modul által szolgáltatott egyéb kiírató függvények:

- `fprintf`: fájlba ír, első paramétere a fájl, a többi paramétere ugyanaz, mint a `printf`-nek.
- `eprintf`: standard error-ra ír, paraméterei ugyanazok, mint a `printf`-nek.
- `sprintf`: nem ír ki semmit, hanem egy szöveges típusú értéket ad vissza, amely a formázott szöveget tartalmazza; paraméterei ugyanazok, mint a `printf`-nek.

3.4.3. Egyéb kiíró kifejezések

JoCaml-ben a következő beépített típusokhoz léteznek „saját” kiírató függvények: `print_int`, `print_float`, `print_char`, `print_string` (ezek mind a standard outputra írnak ki).

4. A JoCaml nyelv objektum-orientált nyelvi elemei

Az objektum-orientált programozás adatközpontú programtervezést valósít meg: az adatokat és a rajtuk értelmezett műveleteket egy egységbe zárja.

Az objektum-orientált programozás előnyei [3]**:**

- Enkapszuláció (egységbezárás);
- Adatelrejtés;
- Kód újrafelhasználás.

Az objektum-orientált programozás fő elvei [3]**:**

- Objektum-orientált moduláris struktúra;
- Adatabsztrakció;
- Osztályok;
- Öröklődés;
- Polimorfizmus és dinamikus összekapcsolás;
- Többszörös és ismételt öröklődés.

4.1. Osztály

JoCaml-ben az osztályok adatokat (**adattag**okat) és függvényeket (**metódus**okat) tartalmazhatnak egy egységben (pont úgy, ahogy az más objektum-orientált programozási nyelveknél is „szokás"). A JoCaml azonban egy funkcionális nyelv, amely bár tartalmaz imperatív elemeket, alapvetően mégiscsak egy funkcionális nyelv. Így jogosan vetődik fel a kérdés, hogy vajon az adattagok frissíthetőek-e? A válasz: alapértelmezésben nem, de – a rekordokhoz hasonlóan – `mutable` kulcsszó használatával már módosíthatók lesznek. A módosítás is a rekordoknál megszokott szintaxissal történik: *`adattag <- új_érték`*.

A JoCaml objektum-rendszerében **osztályszintű adattag és osztályszintű metódus nincs**, szükség esetén globális azonosítókkal és globális függvényekkel lehet „pótolni" a hiányukat. (Megjegyzés: globális azonosítók és függvények használata kevésbé biztonságos és jó eszköz, mert így az osztályon kívülről is „bárki" használhatja őket.)

Modularitás:

Az objektum-orientált programozásban a modularitás azt jelenti, hogy az osztályokat elhelyezhetjük (bizonyos nyelveknél ez kötelező is) egy önálló modulban (fordítási egységben). Az ezt megvalósító nyelveket további két csoportra oszthatjuk, aszerint, hogy az osztály deklarációja és megvalósítása külön fordítási egységbe tehető-e (esetleg ez egyenesen kötelező is), vagy sem.

JoCaml-ben ilyen értelemben **korlátozott modularitás** van. A nyelv modul-rendszere megengedi az önálló fordítási egységek létrehozását, egy ilyen modul pedig osztály(oka)t is tartalmazhat. Tehát van lehetőség az osztályok külön fordítási egységben történő elhelyezésére. Azonban a nyelv egyáltalán nem támogatja az osztály deklarációjának és megvalósításának szétválasztását. [Megjegyzés: ez a JoCaml alapjául szolgáló 1.07-es verziószámú Objective Caml-ben még így volt, azonban az Objective Caml 3.0-ás verziójától már szétválasztható az osztály deklarációja (*osztály típus*) és megvalósítása (*osztály kifejezés*).]

4.1.1. Osztály deklarációja

```
class osztálynév paraméterek = adattagok_és_metódusok end
```

Megjegyzés: a paraméterek (*osztály-paraméterek*) használata kötelező, ha nem akarjuk paraméterezni az osztályt, akkor is legalább egy `()`-t ki kell rakni a definícióban. Az osztály-paraméterek aktuális értékét az osztályhoz tartozó objektumok létrehozásakor kell megadni.

Adattag definíciója: `val [mutable]` *`adattag_neve`* **`=`** *`kezdeti_érték`*

Metódus deklaráció: `method` *`metódus_neve`* `[`*`paraméterek`*`] =` *`metódustörzs`*

Az elnevezés tekintetében az osztályokra, az osztály-paraméterekre, az adattagokra és a metódusokra ugyanazok a szabályok érvényesek, mint az egyéb azonosítókra. Az osztály-paraméterek, az adattagok és a metódusok nevei között lehetnek átfedések, tehát lehet ugyanolyan nevű adattagunk és metódusunk, stb. Ebből nem származik „keveredés", ugyanis: (1) az osztályparaméterek csak az adattagok kezdeti értékeiben szerepelhetnek, azonban a kezdeti értékekben metódus, vagy adattag nem lehet; (2) a metódusok törzsében szerepelhetnek adattagok, de nem szerepelhetnek más metódusok és osztályparaméterek; (3) „kívülről" csak a metódusok látszanak.

4.1.2. Konstruktor

JoCaml-ben **nincs konstruktor-függvény** (ellentétben például a C++-al); tehát nincs egy olyan metódus, ami egy új objektum létrehozásakor automatikusan meghívódna és inicializálná az objektumot.
Azonban az adattagok JoCaml-ben sem maradnak inicializálatlanul: egy új objektum létrehozásakor mindig kell valamilyen értéket kapniuk. Ezt a **kezdeti érték**et már az adattag definíciójakor meg kell adni, azonban az érték kiértékelése csak egy új objektum létrehozásakor történik meg (minden új objektum létrehozásakor kiértékelődik). A kezdeti érték számításához felhasználhatjuk az osztály-paraméterek aktuális értékét is, így egy új objektum létrehozásakor „kívülről" is befolyásolhatjuk az adattagok kezdeti értékét.
Összegzésül tehát: konstruktor-függvény nincs, de osztály-paraméterek és adattagok kezdeti értékét megadó kifejezés használatával ugyanazt a kifejezőerőt érhetjük el.

4.1.3. Destruktor

A JoCaml-ben **nincs destruktor**, tehát nincs lehetőségünk, hogy egy objektum megszűnésekor bármiféle tevékenységet végezzünk.

4.1.4. Információ elrejtés (láthatóság)

Az **adattag**ok JoCaml-ben mindig **rejtett**ek: őket az osztályon kívülről nem lehet elérni. Az adatok elrejtésének fő oka a biztonság, ugyanis az adatok felhasználók általi közvetlen manipulálása inkonzisztenciát okozhat, mivel a felhasználó nem feltétlenül ismeri a belső adatstruktúra összefüggéseit teljes részletességében [3]. Szükség esetén az adattagokhoz lekérdező és beállító metódusokat lehet írni.

JoCaml-ben a **metódus**okat láthatóság szempontjából két csoportra bonthatjuk:

- **Publikus** (ez az alapértelmezés): ezek a metódusok láthatóak az osztályon kívülről is;
- **Védett** (a `method protected` kulcsszóval): a metódus kívülről nem látható; csak ugyanannak az objektumnak más metódusaiból hívható [7].
  A védett metódusok az osztály leszármazottjainak metódusaiból is hívhatók, azonban ebben az esetben egy igen érdekes (más programozási nyelvben nem tapasztalt) jelenség lép fel: a metódus a leszármazottban publikussá változik. Ha nem használjuk a leszármazott osztály metódusaiból, akkor védett marad.

A láthatósági szabályok alól semelyik osztálynál nem tehetünk kivételt (nincs olyan mechanizmus, mint pl. C++-ban a *friend*).

Megjegyzés: JoCaml-ben **nincs privát metódus** (mint pl. a C++-ban), tehát nincs olyan láthatósági szabály, amely a metódust még az osztály leszármazottjai elöl is elrejtené.

4.1.5. Példa

Létrehozunk egy osztályt (`pont` néven), ami egy kétdimenziós pontot valósít meg. Az osztálynak két adattagja van: a pont `x` és `y` koordinátája; az adattagok értéke változhat. A `pont` osztályhoz két metódus tartozik: az egyikkel (`get`) a pont aktuális értékét lehet lekérdezni, a másikkal (`eltol`) pedig el lehet tolni a pontot. Az osztálynak két paramétere van: a pont koordinátáinak kezdeti értéke.

```
class pont x0 y0 =
    val mutable x = x0
    val mutable y = y0
    method get = x, y
    method eltol x1 y1 = x <- x+x1; y <- y+y1
end;;
```

4.2. Objektum

4.2.1. Objektumok létrehozása

Szintaxis: `new` *`osztálynév aktuális_paraméterek`* Például:

```
let my_pont = new pont 7 3;;
```

Az objektum létrehozásakor kötelező az összes osztály-paraméter aktuális értékének megadása. Ha valamelyik paramétert nem adjuk meg, akkor nem egy objektumot, hanem egy függvényt kapunk ami a nem megadott paraméterekről az adott osztályba tartozó objektumra képez:

```
# let pont2 = new pont 7;;
val pont2 : int -> pont = <fun>
```

Megjegyzés: Ha az osztálynak nincs paramétere, akkor egy `()`-t kell megadni az osztályba tartozó objektum létrehozásakor, különben egy `unit ->` *`osztály`* típusú függvényt kapunk.

Az osztály kifejezései nem az osztály definiálásakor, hanem egy új objektum létrehozásakor értékelődnek ki.

4.2.2. Metódusok meghívása

Szintaxis: *`objektum`* `#` *`metódus aktuális_paraméterek`*, például: `my_pont#eltol 2 3;;`

4.2.3. Az objektum típusa

Az objektum típusa a metódusok nevének és típusának felsorolása:

<*metódus_név*$_1$: *típus*$_1$; …; *metódus_név*$_n$: *típus*$_n$>,

például a `my_pont` típusa: `<get : int * int; eltol : int -> int -> unit>`

Tehát az objektum típusánál csak a metódusok neve és típusa számít: ha van két különböző osztályunk, amelyek csak az adattagokban és az osztály-paraméterekben különböznek, de a metódusok neve és típusa ugyanaz, akkor a két osztályba tartozó objektumok ugyanolyan **típusúak** (még akkor is, ha a metódusok megvalósítása különbözik) [7].

Hiányos (általános) objektumtípus:

Nézzük például a következő függvényt:

```
let tavolsag_az_origotol p =
      let x, y = p#get in
      sqrt( float(x*x + y*y));;
```

Vajon mi lesz a `p` típusa? A használatából (`p#get`) az derül ki, hogy a `p` egy olyan objektum, aminek van egy `get` nevű metódusa, ami egy `int * int` párt ad vissza. Ennek megfelelően a `p` típusa a következő lesz: `<get : int * int; ..>` (a függvény típusa pedig ennek megfelelően: `<get : int * int; ..> -> float`; a függvény tehát tetszőleges olyan objektumra használható, aminek van egy `get : int * int` metódusa). A `<get : int * int; ..>` egy **hiányos objektumtípus**, mert tartalmaz „. . ”-t, ami helyén tetszőleges metódus(ok) állhat(nak) [7].

Teljesen hiányos (általános) objektumtípus: `< .. >`. Ez a típus egy tetszőleges objektumot jelöl.

Megjegyzés: Ezek a hiányos objektumtípusok pontosan úgy viselkednek, mint a **típusváltozó**k [7].

Osztályba tartozó objektum általános típusa: `#`*`osztály_név`*. Ez a rövidítés egyesíti az adott osztályba tartozó objektumok típusát az összes olyan objektum típusával, ami rendelkezik az *`osztály_név`* összes metódusával, (tehát ami az *`osztály_név`*-nek *altípusa* – lásd később). Az `#`*`osztály_név`* tehát egy hiányos objektumtípus:

`#`*`osztály_név`* ≡ `<` *`osztály_név_metódusai`*`; .. >` [7]

Ezt a típust tetszőleges helyen használhatjuk, például az előző függvényt így is megírhattuk volna: `let tavolsag_az_origotol (p : #pont) = …` (a függvény típusa ebben az esetben a következő lesz: `#pont -> float`).

Típus korlátozás (*coercion*): egy objektum típusát korlátozhatjuk egy tetszőleges (akár hiányos) objektumtípusra, amely nem tartalmaz olyan metódust, ami az adott objektumban nincs. Tehát az objektum típusát **csak gyengíteni** lehet. (Pontosabban: akkor korlátozhatunk egy objektumot egy típusra, ha az objektum típusa az adott típusnak altípusa – ld. később. Jelölés: (*objektum* [: $típus_1$] :> $típus_2$), ahol *objektum* típusa $típus_1$, és a $típus_1$ altípusa a $típus_2$-nek) Például:

```
(my_pont :> <get : int * int; ..>);;
(my_pont :> < .. >);;
(my_pont :> <get : int * int>);;
```

4.2.4. Hivatkozás önmagára („self”)

Egy metódus is küldhet üzenetet annak az objektumnak, amelyik meghívta. JoCaml-ben nincs *„self”* vagy *„this”* tehát nincs egy olyan mutató, ami az aktuális objektumra mutat. Azonban ez könnyen megvalósítható, ha az osztályhoz kötünk egy nevet:

`class` *`osztálynév paraméterek`* **`as`** ***`név`*** `= …`

Ezután az aktuális objektumra „*név*” néven hivatkozhatunk. A név kötése mindig a metódus-hívásakor fog megtörténni [7].

Megjegyzés: a *`név`* nem egy azonosító (ugyanis nem használható önmagában); csak a metódusok elérésére (*`név#metódus`*) lehet használni.

Ez a *módszer* csak a metódusoknál szükséges, az adattagokhoz közvetlenül is hozzá lehet férni.

Hivatkozás az adott osztályba tartozó objektumok típusára (a saját típusára): `class` *`osztálynév paraméterek`* `:` **`'a`** `= …` Ekkor az *`'a`* típusváltozó az osztály törzsében az osztályba tartozó objektumok típusát fogja jelölni. Például a pont osztályban bevezetünk egy új metódust [7], ami a ponthoz hozzáad egy másikat:

```
class pont x0 y0 : 'a =
    …
    method hozzaad (p : 'a) = let x1, y1 = p#get in x <- x +x1; y <- y+y1
end
```

Megjegyzés: a kétféle hivatkozás együttesen is használható:

```
class osztálynév paraméterek as név : 'típus
```

4.2.5. Objektumok másolása

Az objektumokat mutatóként kezeli a rendszer. Ezért, ha egy másik azonosítóhoz hozzárendelünk egy objektumot, akkor az új azonosító ugyanarra az objektumra fog mutatni [7]:

```
# let my_pont = new pont 7 3;;
val my_pont : pont = <obj>
# let my_pont_2 = my_pont;;
val my_pont_2 : pont = <obj>
# my_pont#eltol 2 2;;
- : unit = ()
# my_pont_2#get;;
- : int * int = 9, 5
```

Az `Oo.copy` „rendes” másolatot készít az objektumról:

```
# let my_pont' = new pont 7 3;;
val my_pont' : pont = <obj>
# let my_pont_2' = Oo.copy my_pont';;
val my_pont_2' : pont = <obj>
# my_pont'#eltol 2 2;;
- : unit = ()
# my_pont_2'#get;;
- : int * int = 7, 3
```

Megjegyzés: az `Oo.copy` típusa: `(< .. > as 'a) -> 'a`. Tehát egy tetszőleges objektumról egy ugyanolyan típusú objektumra képez. (Ez nem ugyanaz a típus, mint a `< .. > -> < .. >`, mert ez utóbbi tetszőleges objektumról tetszőleges objektumra képez, de itt a paraméter és az eredmény típusa egymástól különböző is lehet) [7].

4.3. Öröklődés:

A leszármazott osztály tartalmazza az ősosztály összes adattagját és metódusát. Ezen kívül definiálhatunk új adattagokat és metódusokat is.

Szintaxis:

```
class osztálynév paraméterek =
        inherit ős_osztály ős_osztály_aktuális_paraméterei
        új_adattagok_és_új_metódusok
    end
```

Alapértelmezett ősosztály nincs.

Az örökölt metódusokat és az adattagok kezdeti értékét **felül lehet definiálni**, azonban a metódus és az adattag típusa nem változhat. Ha az adattag az ősben nem volt változtatható, akkor a leszármazottban definiálhat*juk* változtathatóként (`mutable`), azonban ezt fordítva nem lehet (változtathatóból nem lehet változtathatatlan). A felüldefiniálás szintaxisa egyszerű: a metódust és az adattagot újra meg kell adni (most az új törzzsel / kezdeti kifejezéssel) az új adattagok és metódusok között.

Polimorfizmus és dinamikus összekapcsolás:

Polimorfizmus (többalakúság): az objektum-orientált nyelveknél a polimorfizmus az a jelenség, hogy egy változó nem csak egyfajta típusú objektumra hivatkozhat [3].
Dinamikus összekapcsolás: a változó éppen aktuális típusának megfelelő metódus-implementáció hajtódik végre [3].

A JoCaml szigorú típusossága egyáltalán **nem engedi meg a többalakúságot**. Ugyan az objektumot korlátozhatjuk egy nálánál gyengébb (csak a metódusai egy részét tartalmazó) típusra, de ettől az objektum típusa nem változik meg: az `(`*`objektum`*` :> `*`gyengébb_típus`*`)` kifejezés típusa *`gyengébb_típus`*, de ettől az *`objektum`* típusa nem lesz *`gyengébb_típus`*. Mivel többalakúság nincs, így a dinamikus összekapcsolás szóba sem kerül.

Azonban a dinamikus összekapcsoláshoz hasonló kérdés **típus korlátozás** esetén is felmerül: ha egy adott *`metódus`* az *`objektum`* osztályában és a *`gyengébb_típus`* osztályban másképpen van definiálva, akkor az `(`*`objektum`*` :> `*`gyengébb_típus`*`)#`*`metódus`* melyik metódus meghívását jelenti? Válasz: mindig az *`objektum`* osztályának megfelelőt. A típuskorlátozás tehát semmi mást nem jelent, mint bizonyos metódusok meghívásának **tiltás**át: az `(`*`objektum`*` :> `*`gyengébb_típus`*`)` objektumban csak azokat a metódusokat hívhatjuk meg, amik `gyengébb_típus` osztálynak is metódusai, de **nem** a *`gyengébb_típus`* metódusa fog meghívódni, hanem mindig az eredeti (a korlátozás előtt lévő) osztály metódusa.

Miért van akkor egyáltalán szükség a típuskorlátozásra? Azért, mert a JoCaml-ben nincsen automatikus típuskonverzió (az altípusképzés sosem implicit, mindig ki kell írni). Ha egy függvényt egy bizonyos osztályon értelmeztünk, akkor azon az osztályon kívül semmilyen más osztályra nem hívhatjuk meg; ha az adott osztály altípusára szeretnénk használni, akkor típuskorlátozást kell alkalmaznunk.

Késleltetett kötés (*delayed binding*):

Tegyük fel, hogy a leszármazott osztályban egy metódust (nevezzük $\texttt{metódus}_1$-nek) felüldefiniálunk, azonban egy másik metódus (legyen $\texttt{metódus}_2$), – amelyik a törzsében meghívja a $\texttt{metódus}_1$-et –, változatlan marad. Ha egy leszármazott osztálybeli objektumnál meghívjuk a $\texttt{metódus}_2$ metódust, akkor a törzsében a $\texttt{metódus}_1$-nek melyik változata fog meghívódni? Ha késleltetett kötést alkalmazunk, akkor a felüldefiniált $\texttt{metódus}_1$, ha statikus kötést (*static binding*) használunk, akkor az eredeti.
A JoCaml **késleltetett kötés**t alkalmaz [5], tehát bár a $\texttt{metódus}_2$ változatlan maradt, mégis máshogy fog viselkedni a leszármazottban (a $\texttt{metódus}_1$ felüldefiniálása miatt).

4.3.1. Hivatkozás az ősosztályra

Nincs beépített „*super*” mutató. Ha valamelyik szülő osztályra szeretnénk hivatkozni, azt a „*self*”-hez hasonlóan lehet megoldani: egy nevet kell kötni az ősosztályhoz [7]:

```
inherit ős_osztály ős_osztály_aktuális_paraméterei as név.
```

Ezután az ősosztályra „*`név`*” néven hivatkozhatunk. Ezzel a *módszerrel* elérhetjük az ősosztály azon metódusait is, amelyeket a leszármazottban felüldefiniáltunk.

Megjegyzés: az ősosztály adattagjaira közvetlenül is hivatkozhatunk.

4.3.2. Példa

Készítünk egy kétdimenziós pontot, melynek színe is van. Ez az osztály a `pont` osztály leszármazottja lesz. Az örökölt metódusokon kívül tartalmaz egy úgy adattagot (amely a színt tárolja), és egy hozzá tartozó lekérdező-függvényt [7]:

```
class szines_pont x y (c : string) =
    inherit pont x y
    val szin = c
    method get_szin = szin
end;;
```

4.3.3. Többszörös öröklődés

A nyelv megengedi a többszörös öröklődést is. Többszörös öröklődés használatával az új osztály tartalmazni fogja az összes ősosztályának összes adattagját és metódusát.

Szintaxis: `class` *`osztálynév paraméterlista`* **=**

```
        inherit ős_osztály1 ős_osztály1_aktuális_paraméterei
        inherit ős_osztály2 ős_osztály2_aktuális_paraméterei
        …
        új_adattagok_és_új_metódusok
    end
```

Többszörös öröklődéskor mindig felmerül az a kérdés, hogy mi van akkor, ha ugyanaz a név több ősosztályban is előfordul? Ilyen esetben a fordító csak akkor fogadja el az osztályt, ha az azonos nevű metódusok, vagy adattagok **típusa** megegyezik. Azonban ekkor van még egy kérdés: az azonos nevű metódusok, vagy adattagok közül melyik lesz az érvényes? Ez esetben – ahogy az JoCaml-ben névütközéskor megszokott – az öröklődési listában szereplő utolsó osztály metódusa (vagy adattagja) lesz az érvényes, ez eltakarja a többi ugyanilyen nevű metódust vagy adattagot. Az eltakart adattagra, vagy metódusra az ősosztályhoz rendelt azonosítóval hivatkozhatunk (*`„super”`*`#`*`metódus`*).

4.3.4. Példa a többszörös öröklődésre

Készítünk egy változtatható színű pontot. Ehhez létrehozunk egy `szin` osztályt, ami a szín lekérdezésére és megváltoztatására használható metódusokat tartalmaz. A `valzoztathato_szinu_pont` örököl mind a `pont`, mind a `szin` osztályból.

```
type szin_tipus = Fekete | Feher | Kek | Zold | Piros | Sarga;;
class szin () =
    val mutable szin = Fekete
    method get_szin = szin
    method set_szin szin1 = szin <- szin1
end;;

class valtoztathato_szinu_pont () =
    inherit pont 0 0
    inherit szin ()
    method reset = x <- 0; y <- 0; szin <- Fekete
end;;
```

4.3.5. Lezárt osztályok

Szinatxis: `class closed osztálynév …`

Egy lezárt osztály leszármazottaiban nem lehet új metódust bevezetni. Új adattagokat azonban be lehet vezetni és a régi adattagokat és metódusokat is felül lehet definiálni. A leszármazott osztálynál tehát az a kikötés, hogy az objektumainak a típusa ne változzon az őséhez képest. A leszármazott osztály is lezárt lesz, és kötelező így deklarálni.

4.4. Paraméterezett (polimorf) osztályok

Az osztályt paraméterezhetjük egy vagy több típusváltozóval. Polimorf osztály használatával elérhető, hogy az osztály többféle típusra is működjön. A típusra kikötéseket is tehetünk.

A paraméteres osztálynak egy vagy több polimorf adattagja vagy metódusa van. A polimorf adattagban vagy metódusban lévő típusváltozót expliciten fel kell tüntetni az osztály deklarálásakor az osztálynév előtt (pont úgy, mint az unió típuskonstrukciónál). A feltüntetett típusváltozót kötelező használni valahol az osztály törzsében (legalább annyi helyen, hogy az automatikus típuslevezetés ne tételezze fel, hogy valamelyik metódusban vagy adattagban a feltüntetettől eltérő típusváltozó szerepel).

Szintaxis: `class` *`típusváltozók osztálynév paraméterlista`* `= …` Ha több típusváltozót használunk, akkor a típusváltozókat kerek zárójelek között vesszővel elválasztva kell felsorolni, egy típusváltozó esetén nem kell zárójel.

Példa: Megvalósítok egy verem osztályt [7]. Az osztály a verem elemeit egy listában tárolja. A vermen a szokásos műveletek értelmezettek: elem berakása a verembe (`push`), elem kivétele a veremből (`pop`), legfelső elem megnézése (`top`), lekérdezés, hogy a verem üres-e (`is_empty`). A verem tetszőleges (de azonos) típusú elemekből állhat (=> `'a`).

```
class 'a verem () =
    val mutable v = ([] : 'a list)
    method is_empty = (v = [])
    method push e = v <- (e :: v)
    method pop = match v with
        e :: l -> v <- l; ()
       | []     -> raise (Invalid_argument "Üres lista")
    method top = match v with
        e :: l -> e
       | []     -> raise (Invalid_argument "Üres lista")
end;;
```

4.4.1. Típus-megszorítás (constraint)

A típusparaméter típusára vonatkozóan megkötéseket tehetünk. Például ha bevezetünk egy `'a` típusváltozót és az `m` névhez kötjük (például `val m = (`*`érték`*` : 'a)`), majd valahol az osztálytörzsben előfordul egy `m#metódus` kifejezés, akkor ez egy – jelen esetben automatikus – **megszorítás**t jelent az `'a` típusváltozóra: most már nem tetszőleges típusú lehet, hanem csak egy olyan objektum jó a helyére, aminek van `metódus` névre hallgató (esetleg ismert típusú) metódusa. Ezt a megszorítást az automatikus típuslevezetés „kitalálja”. Azonban mi is elhelyezhetünk az osztályban megszorításokat. A megszorítás **szintaxis**a: `constraint` $típus_1$ `=` $típus_2$ [7]

Példa: megvalósítunk egy kör osztályt, aminek középpontja és sugara van, a kört el lehet tolni, az eltolás a középpont eltolását jelenti. Az osztályban expliciten nincs megszorítás, de mint látható, a fordító levezeti [7]:

```
# class 'a kor (kozeppont : 'a) (sugar : int) =
    val mutable kozeppont = kozeppont
    val mutable sugar = sugar
    method get_kozeppont = kozeppont
    method eltol = (kozeppont#eltol : int -> int -> unit)
  end;;
class 'a kor ('a) (int) =
    constraint 'a = <eltol : int -> int -> unit; ..>
    val mutable kozeppont : 'a
    val mutable sugar : int
    method get_kozeppont : 'a
    method eltol : int -> int -> unit
end
```

A megszorítást expliciten is meg lehet adni, például kiköthetjük, hogy a kör középpontjának típusa csak `pont` típusú lehet. Ez szigorúbb megkötést jelent, mint amit az előző példában használtunk, mert ott csak azt tételeztük fel, hogy a középpontnak rendelkeznie kell `eltol : int -> int -> unit` metódussal.
Ha a megkötésben egy **osztályra** akarunk **hivatkozni**, akkor ez az *#osztály_név* szerkezettel tehetjük meg (ez a rövidítés az osztályon belül csak megszorításban használható, de az osztályon kívül használható teljes-értékű típusként) [7].

```
class 'a kor (kozeppont : 'a) (sugar : int) =
    constraint 'a = #pont
    val mutable kozeppont = kozeppont
    val mutable sugar = sugar
    method get_kozeppont = kozeppont
    method eltol = kozeppont#eltol
end;;
```

4.5. Virtuális (absztrakt) osztályok

Virtuális osztálynak nevezzük azt az osztályt, amelynek legalább az egyik metódusa virtuális. A virtuális metódusnak csak a szignatúráját adjuk meg, míg a metódus törzse definiálatlan marad. A virtuális osztályból nem lehet objektumot létrehozni (nem lehet példányosítani). Az ilyen osztályokat sablonnak is tekinthetjük, mert előírják bizonyos típusú metódusok meglétét, de a definíciót nem adják meg. Ha az osztály leszármazottjában az összes virtuális metódusnak megadjuk a konkrét definícióit, akkor már példányosítható osztályt kapunk.

A virtuális osztályok használata akkor célszerű, ha az osztály egyes metódusait meg tudjuk adni általános formában is, de a metódusok többi részét, csak bizonyos speciális esetben tudnánk megadni. Ezek a speciális esetek lesznek a leszármazottak, azokat a metódusokat pedig, amiket az ősosztályban nem lehetett általánosan megadni, minden leszármazottnál külön definiáljuk.

Szintaxis: **Virtuális metódus**oknál a `method` helyett a `virtual` kulcsszót kell használni. A **virtuális osztály**okat a `class virtual` kulcsszóval vezeti be.

Példa: Készítek egy általános pont osztályt, ahol a pont egy euklideszi tér egy pontja. A pont lehet 1, 2, 3 stb. dimenziós, de az is lehet, hogy az euklideszi tér csak bizonyos feltételeknek eleget tevő pontokat tartalmaz. Minden euklideszi téren értelmezett a skaláris szorzás, ami a tér két pontjához egy valós számot rendel. Azonban akármi is a skaláris szorzás, az biztos, hogy a normát úgy kell kiszámolni, hogy a pontnak kell venni a skaláris szorzatát saját magával, majd az eredményből négyzetgyököt kell vonni.

```
class virtual altalanos_pont () as p : 'a =
    virtual skalaris_szorzas : 'a -> float
    method  norma = sqrt (p#skalaris_szorzas p)
end;;
```

Most nézzünk meg egy kétdimenziós pontot. A kétdimenziós esetben a skaláris szorzást úgy számolják, hogy a két pont megfelelő koordinátáit összeszorozzák, majd az eredményeket összeadják.

```
class ket_d_pont x y : 'a =
    inherit altalanos_pont ()
    val x = x
    val y = y
    method get = x, y
    method skalaris_szorzas (p : 'a) = let x1, y1 = p#get in x*.x1+.y*.y1
end;;
```

Megjegyzés: JoCaml-ben a virtual nem ugyanazt jelenti, mint C++-ban. C++-ban ugyanis a virtual azt jelenti, hogy a leszármazottban a metódus felüldefiniálható és a dinamikus összekapcsolás biztosított. JoCaml-ben a virtual-nak az előbbihez semmi köze, itt egyszerűen azt jelenti, hogy a definíció nincs megadva [5].

Rekurzív típus: a példákon látszik, hogy az osztályok rekurzívak is lehetnek, például az altalanos_pont osztályba tartozó objektumok típusa:

```
<skalaris_szorzas : 'a -> float; norma : float> as 'a
```

(Megjegyzés: a rekurzív típusnak a virtuális osztályhoz semmi köze.)

4.6. Objektumok közötti altípus-reláció

4.6.1. Altípus definíciója

Legyen $t = \langle m_1 : t_1;\ m_2 : t_2;\ \ldots;\ m_n : t_n \rangle$,
$t' = \langle m_1 : s_1;\ m_2 : s_2;\ \ldots;\ m_n : s_n;\ m_{n+1} : s_{n+1};\ \ldots;\ m_k : s_k \rangle$

A t' **osztály** akkor **altípus**a a t osztálynak ($t' \leq t$), ha $s_i \leq t_i$ ($\forall\ i \in \{1 \ldots n\}$-re), tehát [5]:

1. a t-nek nincsen olyan **nevű** metódusa, ami az t'-ben nincs (azonban a t' tartalmazhat új metódusokat is);
2. a közös nevű metódusoknál a t' metódusainak típusa **altípusa** a t metódusai típusának (az implementációnak azonban nem kell egyeznie).

Legyen f' : t' -> s' és f : t -> s függvény. Az f' **függvény** akkor **altípus**a az f függvénynek ($f' \leq f$), ha $s' \leq s$ és $t \leq t'$, tehát az f paraméterének (ha a függvényeknek több paramétere van akkor mindegyiknek külön-külön) típusa altípusa az f' paramétere típusának és f' eredményének típusa altípusa az f eredménye típusának.

A függvény-altípusdefiníció helyességének (értelmességének) igazolása [5]:
Nézzük a következő két osztályt: c' = <m : t' -> s'> és c = <m : t -> s>, és tegyük fel, hogy $c' \leq c$ (tehát $t' \to s' \leq t \to s$). Nézzünk egy példát a lehetséges használatra: vegyünk két objektumot: o' : c' és o : c és a két függvényt: g : s -> a és h (o : c) (x : t) = g(o#m(x)).

1. h az első paraméterében c típusú objektumot vár, így meghívhatjuk (o' :> c)-vel is, de ebben az esetben a c'-beli m értékelődik ki, ami egy s' típusú értéket ad vissza. Mivel a g-t (ami s-t vár) ezzel hívjuk meg, így szükséges, hogy **s' ≤ s**.

2. `h` a második paraméterében egy `t` típusú érték van. Azonban, ha a `h` első paraméterében `(o' :> c)` van, akkor – mivel a `c'`-beli `m` értékelődik ki – az `o'#m` egy `t'`-t vár, ezért szükséges, hogy **t ≤ t'**.

4.6.2. Altípus és leszármazott kapcsolata

Már az altípus definíciójából is látszik, hogy **az altípus és a leszármazott egymástól különböző fogalmak** [5]. (A legtöbb objektum-orientált nyelvben – például C++, Java – az öröklődés altípusképzést jelent.)

1. Az altípus, nem feltétlenül leszármazott, ugyanis egymástól teljesen függetlenül is létrehozhatunk olyan osztályokat, ahol az egyik osztály szignatúrája a másik osztályénak részhalmaza, például:

   ```
   class egy_d_pont x =
       val mutable x = x
       method get_x = x
       method eltol x1 = x <- x+x1
   end;;
   ```

   ```
   class szines_egy_d_pont (c : string) =
       val mutable x = 0
       val c = c
       method get_x = x
       method get_color = c
       method eltol x1 = x <- x1
   end;;
   ```

   Az `egy_d_pont` metódusai: `get_x` és `eltol`. Ilyen nevű és típusban egyező metódusai `szines_egy_d_pont` osztálynak is vannak (látszik azonban, hogy az `eltol` metódus törzse mindkét osztályban más). A `szines_egy_d_pont` osztály az `egy_d_pont` osztály **altípusa, de nem leszármazottja**.

2. Az öröklődés során a leszármazott osztály örökli az ősosztály összes metódusát. Bevezethet új metódusokat és felüldefiniálhat régieket, de a metódus típusa nem változhat. Ezek után azt hihetnénk, hogy a leszármazott osztály az ősosztályának mindenképpen altípusa lesz. Ez azonban nem mindig van így, mert bizonyos speciális körülmények között az újradefiniált metódus típusa mást jelenthet a leszármazottban, mint az ősében (például, ha a metódus típusváltozót tartalmaz).
   Példaként tekintsük az előző részben mutatott példát, `ket_d_pont` osztály **leszármazottja** az `altalanos_pont` osztálynak, **de nem altípusa**. Ugyanis: a `ket_d_pont` `skalaris_szorzas` metódusának típusa nem altípusa `altalanos_pont` osztály `skalaris_szorzas` metódusának. Ugyan a metódus mind a két osztályban két osztálybeli elemről egy valós számra képez, de az osztálybeli elem az egyik osztályban egy `altalanos_pont` osztályba tartozó objektum, míg a másik osztályban egy `ket_d_pont` típusú objektum. Az `altalanos_pont` (az `altalanos_pont`-beli `skalaris_szorzas` paramétere) pedig nem altípusa a `ket_d_pont` (a `ket_d_pont`-beli `skalaris_szorzas` paramétere) osztálynak, mert csak az utóbbinak van `get` metódusa.

   Megjegyzés: Miért tiltja meg a fordító, hogy a `ket_d_pont` az `altalanos_pont` altípusa legyen? Nézzük a következő helyzetet: legyen `t1 : altalanos_pont`, `t2 : ket_d_pont` (a példa kedvéért tegyük fel, hogy `altalanos_pont` NEM virtuális – erre csak azért van szükség, mert virtuális osztálynak nem lehet objektuma, de az altípus szempontjából ez most indifferens). Ha a `ket_d_pont` az `altalanos_pont` altípusa lenne, akkor a következő

kifejezés szintaktikailag helyes lenne: `(t2 :> altalanos_pont)#skalaris_szorzas t1`. Azonban ebben az esetben a `ket_d_pont` skaláris szorzása hívódna meg, ami megkísérelné használni a `t1` – nem létező – `get` metódusát, ami hibához vezetne.

4.7. Kölcsönösen rekurzív osztályok

Ha több olyan osztályunk van, amik rekurzívan hivatkoznak egymásra, akkor ezt – a kölcsönösen rekurzív függvények definiálásához hasonlóan – egy osztály-definícióban kell elhelyezni [7]:

```
class osztánydefiníció1
 and  osztálydefiníció2
 …
end
```

Például:

```
class window () =
    val mutable top_widget = (None : widget option)
    method top_widget = top_widget
  and  widget (w : window) =
    val window = w
    method window = window
  end;;
```

4.8. Osztályok használata értékadás nélkül

Megírhatjuk a pont osztály egy olyan változatát is, amiben nincs változó és nincs értékadás. A `{< … >}` szerkezet visszaadja az aktuális objektum másolatát; melyben bizonyos adattagok értéke más lehet [7].

```
# class funkcionalis_pont x0 y0 =
    val x = x0
    val y = y0
    method get = x, y
    method eltol x1 y1 = {< x = x + x1; y = y + y1 >}
  end;;
class funkcionalis_pont (int) (int) : 'a =
    val x : int
    val y : int
    method get_x : int * int
    method move : int -> int -> 'a
end
```

Használat:

```
# let p = new funkcionalis_pont 7 3;;
val p : funkcionalis_pont = <obj>
# p#get;;
- : int * int = 7, 3
# (p#eltol 3 1)#get;;
- : int * int = 10, 4
# p#get;;
- : int * int = 7, 3
```

Megjegyzés: A funkcionális pont típusa más lett, mint korábban, ugyanis itt rekurzív.

5. A JoCaml nyelv párhuzamos programozást támogató nyelvi elemei

Már az 1.3.3. Egy JoCaml program felépítése részben is kiderült, hogy egy JoCaml program első közelítésben JoCaml mondatok (*phrase*) sorozata. Egy mondat pedig vagy egy deklaráció, vagy egy kifejezés. Azonban ezen a két elemen kívül van a nyelvnek egy harmadik, nagyon fontos eleme is: a **folyamat**. A folyamatok sosem állhatnak önállóan, azonban léteznek olyan deklaráció- és kifejezésfajták, amelyek folyamatokat tartalmaznak.

A folyamatokkal eddig nem nagyon foglalkoztam, csak bizonyos speciális esetben kerültek említésre (például lehet folyamatokból is elágazást vagy szekvenciát létrehozni). Azonban ebben a fejezetben a folyamatok igen nagy szerepet kapnak.

5.1. A kifejezések és folyamatok összehasonlítása

	Kifejezés	**Folyamat**
Mit ír le?	A kifejezés egy számítás leírása.	A folyamat is egy számítás leírása.
Van-e vissza-adott érték? Ha igen, akkor mi?	Egy kifejezés minden esetben **visszaad egy értéket**, mégpedig a számítás eredményét.	A folyamatok sosem adnak vissza **semmilyen érték**et, ha a számításnak mégis lenne valamilyen eredménye, azt meg kell „semmisíteni”.
Milyen a kiér-tékelés?	**Szinkron**: a rendszer teljesen végrehajtja a kifejezés kiértéke-lését, és csak azután lép tovább.	**Aszinkron**: a rendszer egy új szálon elindítja a folyamat kiértékelését, majd – mivel visszatérési érték nincs – nem vár a befejezésére, hanem egyből megy tovább (így a folyamat kiértékelése és a program további része egymással párhuzamosan fut*hat*).

A folyamatok tehát a program párhuzamosításában játszanak nagy szerepet.

Párhuzamosan futó programoknál az egyik nagy kérdés mindig az, hogy a párhuzamos részek hogyan **kommunikál**nak egymással.

- Egyrészt a folyamatok kommunikálhatnak egymással **közös változók** használatával, ugyanis az azonosítók láthatási szabályai a folyamatok számára ugyanazok, mint a kifejezések számára.
- A folyamatok ezen kívül ***csatorná***kon (más szóval *port-nevek*en) keresztüli üzenetküldéssel is tarthatják egymással a kapcsolatot. (Megjegyzés: a dolgozatban a *csatorna* és a *port-név* teljesen szinonim fogalmak. Ez a csatorna nem ugyanaz, mint a 3.4.1. Fájlműveletek fejezetben használt csatorna)

5.2. Csatornák

A csatorna-deklaráció egyetlen önálló nyelvi szerkezetben deklarál egy portot (amely fogadja a csatornára küldött üzeneteket) és egy ezeket figyelő folyamatot (*őrzött folyamat*) [8]. A folyamat azért „őrzött”, mert a végrehajtása csak akkor kezdődik meg, amikor a csatornára üzenet érkezik – a folyamat minden egyes üzenet érkezésekor végrehajtódik (kiértékelődik). Az üzenet több, vagy akár nulla értékből állhat, azonban minden csatornánál már a deklaráláskor meg kell mondani, hogy a csatorna hány és milyen típusú részekből álló üzenetet vár. A várt üzenetekhez formális

paramétereket rendelünk, melyeket felhasználhatunk a csatorna őrzött folyamatában is, így a csatorna különböző üzenetekre különbözőképpen reagálhat. Amikor a csatornára üzenet érkezik, akkor a csatorna formális paraméterei felveszik az üzenetben kapott értéket, és az őrzött folyamat végrehajtása megkezdődik (az őrzött folyamat *tüzel*). Mivel folyamatról van szó, így a végrehajtás természetesen aszinkron lesz.

A háttérben minden csatornához tartozik egy *várakozási sor*, ahol a csatornára küldött, de még fel nem dolgozott üzenetek várakoznak. A várakozási sornak csak a memória mérete szab határt.

Megjegyzés a paraméterekhez: (1) A csatornának mindig kell, hogy legyen paramétere, még abban az esetben is, ha az őrzött folyamat nem függ semmitől; ekkor is legalább egy `()`-t ki kell tenni a csatorna deklarációjában. (2) Ha a csatornának egynél több paramétere van, akkor a paramétereket kerek zárójelben, egymástól vesszővel elválasztva kell megadni (lásd 2.8.1. Rendezett n-es).

A csatornák – a függvényekhez hasonlóan – elsőrendű (*first class*) értékek [8]. Így egy csatorna lehet egy függvény, vagy egy másik csatorna paramétere és / vagy értéke, készíthetünk csatornákból álló listákat, rendezett n-eseket, tömböket, rekordokat stb.

A csatornáknak két nagy csoportja létezik: szinkron csatorna és aszinkron csatorna (a név a csatornával történő kommunikáció típusára utal). Nézzük részletesen:

5.2.1. Aszinkron csatorna

Deklarálásának **szintaxis**a: `let def` *`csatornanév`*`!` *`paraméter = őrzött_folyamat`*

Megjegyzés: azt, hogy egy csatornát deklarálunk a `def` kulcsszó jelzi, a csatorna aszinkron voltát pedig a *`csatornanév`* után lévő „**!**” (ez a jelölés csak a csatorna-definícióban szerepel, a csatornára történő hivatkozáskor nem kell kiírni) [8].

Az aszinkron csatornák tulajdonságai:

1. Mivel a folyamatok kiértékelése semmilyen értéket nem produkál, így nem meglepő, hogy egy aszinkron csatornára jövő üzenetküldés sem eredményez **semmilyen érték**et.
2. Az aszinkron csatornára történő **üzenetküldés típusa aszinkron**: a küldő elküldi az üzenetet, amely a fogadónál egy üzenetsorba kerül. A küldő *szál* végrehajtása csak az üzenet elküldésének idejére függesztődik fel, a küldő nem vár az őrzött folyamat kiértékelésének befejezésére. A fogadó az üzenet feldolgozásakor kiveszi azt az üzenetsorából és kiértékeli az őrzött folyamatot. Azonban – a hívó oldaláról – nem tudható előre, hogy a kiértékelés pontosan mikor fog megtörténni.
3. Tehát, ha üzenetet küldünk egy aszinkron csatornára, akkor a küldő csak elküldi az üzenetet, de végrehajtása nem függesztődik fel az üzenet kiértékelésének idejére, hanem a kiértékelés folyik tovább (esetleg az őrzött folyamat kiértékelésével párhuzamosan). Az őrzött folyamat kiértékelése után a küldő semmilyen értéket nem kap vissza. Az előbbiekből már látszik, hogy az **aszinkron csatornára történő üzenetküldés** maga is egy **folyamat**.
4. Egy aszinkron csatornák **típus**a: <<*paraméter$_1$_típusa * … * paraméter$_n$_típusa*>>

Példa: készítek egy olyan aszinkron csatornát [8], mely `int` típusú értékeket szállít, melyek értékét tüzeléskor kiírja a konzolra:

```
# let def echo! x = print_int x;
# ;;
val echo : <<int>>
```

Megjegyzés: mivel a `print_int i` egy kifejezés, amely visszaadna egy értéket: `()`, ezért volt szükséges hozzáfűzni egy „`;`”-t, ami ezt elnyomja. A „`print_int x`” ugyanis egy kifejezés, míg a „`print_int x;`” egy folyamat (lásd a 3.2. Szekvencia részben).

5.2.2. Szinkron csatorna

Deklarálásának **szintaxis**a: `let def` *`csatornanév paraméter = őrzött_folyamat`*

A szinkron csatornák tulajdonságai:

1. Az aszinkron csatornákkal ellentétben a szinkron csatornának mindig **kell**, hogy legyen **visszatérési érték**e. Mivel a szinkron csatorna egy folyamatot tartalmaz – amely nem produkál semmilyen értéket – így jogos a kérdés, hogy vajon mi lesz a csatorna visszatérési értéke? A visszatérési értéket az explicit `reply [érték]` folyamattal kell megadni. A `reply [érték]` folyamatot tehát kötelező használni az *`őrzött_folyamat`*-ban (ha az *`érték`*-t nem adjuk meg, akkor az „üres" `reply` egy `()`-t ad vissza).
   Megjegyzés: a visszaadott értéket úgy kell megadni, hogy minden végrehajtáskor pontosan egy `reply` kerüljön kiértékelésre; ha az őrzött folyamat több `reply`-t tartalmaz (például: `if` *kifejezés* `then reply` $érték_1$ `else reply` $érték_2$), akkor a visszaadott értékek típusának meg kell egyeznie.
2. A szinkron csatornára történő **üzenetküldés típusa szinkron** (randevú): a küldő elküldi az üzenetet, majd a kiértékelése felfüggesztődik addig, amíg a szólított őrzött folyamat kiértékelése el nem jut a `reply [érték]` folyamatig. Ez a folyamat adja meg ugyanis a csatorna visszatérési értékét, a hívó rész további kiértékelése csak ezután folytatódhat. A szinkron kommunikáció előnye, hogy itt az információ áramlása kétirányú: a hívó elküldi a csatornának az üzenetet, majd a csatorna is küld egy értéket a hívónak.
3. Tehát, ha üzenetet küldünk egy szinkron csatornára, akkor a küldő végrehajtása az eredmény (ami szinkron csatornáknál kötelező) visszakapásáig felfüggesztődik, ezért az őrzött folyamat és a program további részének kiértékelése nem történik egymással párhuzamosan. Ezen okok miatt a **szinkron csatornára történő üzenetküldés** egy **kifejezés**.
4. Egy szinkron csatorna **típus**a: $param_1$*`_típusa`* `* … *` $param_n$*`_típusa`* `->` *`eredmény_típusa`*

Példa: a feladat ugyanaz, mint az előbb [8] (egy `int` típusú értékeket szállító csatorna, amely tüzeléskor kiírja az üzenet értékét a konzolra); azonban ez most szinkron csatorna lesz:

```
# let def echo_szinkron x = print_int x; reply
# ;;
val print : int -> unit
```

Megjegyzés: látható, hogy a csatorna – miután kiírta az üzenet aktuális értékét a konzolra – a `reply` segítségével visszaad egy `()` értéket.

5.2.2.1. Szinkron csatornák és függvények

A szinkron csatorna leírásából – de legfőképpen a típusából – kiderül, hogy a szinkron csatornák elég közeli rokonságban vannak a függvényekkel. Ha a szinkron csatornák és a függvények típusát megnézzük, akkor rá kell jönnünk, hogy a két típus egy és ugyanaz ($típus_1$ `->` $típus_2$). Ebből következően mindenütt, ahol szinkron csatornák szerepelhetnek, ott szerepelhetnek függvények is, és fordítva, ahol a fordító függvényt vár, ott lehet szinkron csatorna is (természetesen csak akkor, ha a paraméter és az eredmény típusa is egyezik).

Bár a szinkron csatornák és a függvények típusa ugyanaz, azonban a két fogalom több dologban is különbözik egymástól:

	Függvény	Szinkron csatorna
Több paraméter kezelése:	A függvények a több paramétert a ***Curry-módszer*** szerint kezelik: a függvénynek igazából mindig csak egy paramétere van (az első), eredményül pedig egy (magasabbrendű) függvényt kapunk, ami majd kezeli a többi paramétert. A függvény meghívásakor a paramétere-ket szóközzel elválasztva adjuk meg. Ha kevesebb paramétert adunk meg, akkor nem egy értéket (a függvény visszatérési értékét) kapunk, hanem egy magasabbrendű függvényt, amelynek értéke a meg nem adott paraméterektől függ.	A szinkron csatornáknak is igazából mindig csak egy paramétere van. Ha több paramétert szeretnénk, akkor azt egy **összetett adatszerkezet**ben (konkrétan rendezett n-esben) kell megadni (ez jól látszik abból, hogy ha több paraméter van, akkor azokat kerek zárójelben vesszővel elválasztva kell megadni – pont úgy, ahogyan a rendezett n-eseket). A csatornára történő üzenetküldéskor a paramétereket (mivel az egy rendezett n-es) szintén kerek zárójelben, vesszővel elválasztva kell megadni. Ha a csatorna használatakor kevesebb paramétert adunk meg, akkor szintaktikus hibát kapunk. Megjegyzés: ez a paraméterkezelés nem csak a szinkron csatorna sajátossága: az aszinkron csatornák is pontosan így kezelik az egynél több paramétert.
Érték visszaadá-sa	Az érték visszaadása a függvényeknél implicit módon történik: a függvény visszatérési értéke automatikusan a függvénytörzs (ami egy kifejezés) értéke lesz.	A szinkron csatornáknál az érték visszaadása explicit: a törzsben kötelező a `reply` *`érték`* szerkezet használata.

Megjegyzés: mivel a szinkron csatornák és a függvények más módszerrel kezelik az egynél több paraméter problémáját, így egy több paraméteres függvény típusa nem fog egyezni egy ugyanolyan paramétereket és visszatérési értéket tartalmazó szinkron csatorna típusával. Ha típus-egyezőséget szeretnénk, akkor a függvény paramétereit is rendezett n-esként kell kezelnünk:

```
# let def osszead_csatorna (x, y) = reply (x+y);;
val osszead_csatorna : int * int -> int
# let osszead_fuggveny x y = x+y;;
val osszead_fuggveny : int -> int -> int
# let osszead_fuggveny_2 (x, y) = x+y;;
val osszead_fuggveny_2 : int * int -> int
```

Látható, hogy az `osszead_csatorna` csatorna típusa és az `osszead_fuggveny_2` függvény típusa ugyanaz, de az `osszead_fuggveny` függvény típusa más.

Ha a program egyetlen gépen fut (csak egyetlen program van), akkor a függvények és a csatornák a használat szempontjából a fent felsoroltakon kívül semmi másban nem térnek el egymástól. Ha azonban az elosztott rendszerünk több programból áll, és esetleg több gépen fut (lásd 6. A JoCaml nyelv elosztott programozást támogató nyelvi elemei című résznél), és a függvényt, vagy a csatornát el szeretnénk juttatni egyik programtól a másikig, akkor mint majd látni fogjuk, a két primitív egészen máshogy viselkedik (részletesen később).

5.2.3. Csatornák használata

Üzenetküldés csatornára: *`csatornanév aktuális_paraméter`*

A paraméter megadása kötelező, ami mindig egy – a csatorna definíciójában meghatározott számú és típusú elemekből álló – rendezett n-es.

Mint arról korábban szó volt: az aszinkron csatornára történő üzenetküldés egy folyamat, míg a szinkron csatornára történő üzenetküldés egy kifejezés. (Tehát például az „`echo 42`” egy folyamat, míg az „`echo_szinkron 42`” egy kifejezés.) [8]

5.3. Folyamatok használata a programban

Már több szó esett a folyamatokról (például a csatorna deklarálásakor egy folyamatot kell megadni, az aszinkron csatorna meghívása maga is egy folyamat, létre tudunk hozni folyamatokból álló szekvenciát és elágazást, stb.), azonban arról még nem írtam, hogy ezeket hogyan lehet a programban használni. A program ugyanis alapvetően deklarációk és kifejezések listája, így ha „csak úgy” beleírnánk a programba, hogy például „`echo 42`”, akkor szintaktikus hibát kapnánk (hiszen ez se nem deklaráció, se nem kifejezés; ez egy folyamat). A folyamatok – a kifejezésekkel ellentétben – valójában nagyon korlátozott formában fordulhatnak csak elő a programban: például **nem** szerepelhetnek kötés jobb oldalán, nem adhatók át paraméterként (sem függvénynek, sem csatornának) stb. [8]

A folyamatot ezért valamilyen módon „át kell alakítani” („be kell csomagolni”) egy kifejezésbe, ha ki szeretnénk értékeltetni.

A `spawn` folyamatpéldányból kifejezést állít elő, melynek kiértékelésekor egy konkurens folyamat indul el [1]. A `spawn {` *`folyamat`* `}` tehát már egy kifejezés, amely elindítja a *`folyamat`* kiértékelését, majd azonnal visszatér egy `()` értékkel. A *`folyamat`* kiértékelése a program további részének kiértékelésével párhuzamosan történik.

Nézzük a következő programot:

```
spawn { echo 1 } ;;
spawn { echo 2 } ;;
```

A programban mind a két `spawn` csak elindította a folyamatok kiértékelését, így az `echo 1` és az `echo 2` folyamatok kiértékelése egymással párhuzamosan történik. Ezért előre nem tudható, hogy a fenti program `12`-t, vagy `21`-t fog kiírni [8].

5.3.1. A párhuzamos kompozíció operátor („|”)

Párhuzamos végrehajtást a párhuzamos kompozíció operátorral („|”) is el lehet érni. Ez egy bináris infix operátor, amely a megadott két folyamatot egymással párhuzamosan értékeli ki. **Szintaxis**: $folyamat_1$ `|` $folyamat_2$ (eredményül egy folyamatot kapunk) [8].

A következő példa [8] szemantikailag ekvivalens az előzővel:

```
spawn { echo 1 | echo 2 } ;;
```

A párhuzamos kompozíció operátor tulajdonságai:

- Kommutatív: `P | Q ≡ Q | P`
- Asszociatív: `{P | Q} | R ≡ P | {Q | R} ≡ P | Q | R` (emlékezzünk rá, hogy a folyamatokat – a kifejezésekkel ellentétben – nem kerek, hanem kapcsos zárójel használatával lehet csoportosítani)

5.4. A Join-minták

Eddig szó esett arról, hogy hogyan tudunk csatornák és folyamatok segítségével párhuzamosan futó programrészeket készíteni, azonban arról még nem volt szó, hogy hogyan lehet (egyáltalán lehet-e)

valamiféle **szinkronizációt** létrehozni a párhuzamosan futó részek között. A szinkonizációt a **join-mintá**k segítségével valósíthatjuk meg.

Szinkronizációs mintákkal írhatjuk le a konkurens programozás összetett vezérlési és adatkezelési struktúráit. Csatornahalmazokhoz tartozó kommunikációs események között fennálló konfliktust, illetve szinkronizációt fejezhetünk ki velük. Szinkronizációs minták mindig egyetlen `let def` deklarációban egyidejűleg definiált csatornákra vonatkoznak [1]. A join-minták jelentősen kibővítik a csatorna definíciót.

Szintaxis:

`let def` *csatornanév*$_1$ *paraméter*$_1$ `| … |` *csatornanév*$_n$ *paraméter*$_n$ `=` *őrzött_folyamat*

Az előbbi join-minta egyszerre több (`n` darab) csatornát **definiál** és meghatároz közöttük egy szinkronizációs mintát. A mintában megadott csatornák mindegyikének ugyanaz lesz az őrzött folyamata. A join-mintákban a szinkron és az aszinkron csatornák vegyesen is előfordulhatnak. Azonban, ha egy mintában több szinkron csatorna is van, akkor a „sima” `reply [`*érték*`]` nem elegendő: pontosan meg kell határozni, hogy melyik csatorna visszatérési értékét adjuk meg: `reply [`*érték*`]` **`to`** ***csatornanév***. Minden végrehajtás esetén minden szinkron csatornára pontosan egy `reply` –nek kell jutnia [8].
A mintának *lineáris*nak kell lennie, ami azt jelenti, hogy egy csatorna legfeljebb egyszer szerepelhet a mintában.

A **szinkronizáció** abban nyilvánul meg, hogy a Join-mintában megadott őrzött folyamat (ami a mintában lévő csatornáknak közös törzse) csak akkor fog kiértékelődni (tüzelni), amikor már minden egyes – a mintához tartozó – csatornáján várakozik üzenet. Ekkor minden csatornájáról pontosan egy üzenetet felhasznál, a paraméterek pedig felveszik ezen üzenetek értékekeit, és az őrzött folyamat kiértékelése megkezdődik.

Ha a Join-mintában **aszinkron port-név** szerepel: amikor egy ilyen csatornára üzenetet küldünk, akkor a hívó „szokás szerint” csak az elküldés idejére függesztődik fel. Az üzenet bekerül a megfelelő csatorna várakozási sorába, és addig ott is marad, amíg a Join-mintában megadott összes csatorna várakozási sorában nem lesz legalább egy érték. Ebből azonban a hívó semmit sem vesz észre, az ő kiértékelése folytatódik tovább.

Ha a Join-mintában **szinkron port-név** szerepel: amikor egy ilyen csatornára üzenetet küldünk, akkor a hívó felfüggesztődik egészen addig, amíg az őrzött folyamat kiértékelése el nem jut az ehhez a csatornához tartozó `reply`-ig (azonban ahhoz, hogy az őrzött folyamat kiértékelése egyáltalán elkezdődjön szükséges, hogy a Join-mintában megadott összes csatorna várakozási sorában legyen legalább egy érték).

Megállapíthatjuk tehát, hogy függetlenül attól, hogy a csatornát „simán”, vagy Join-mintában definiáltuk-e, ha egy csatornára üzenetet küldünk, akkor a használat szempontjából a hívó oldalon nem látunk különbséget.

Valójában – amint az a 7. Join-kalkulus kifejezőereje című fejezetben látható lesz – a Join-minta nagy kifejezőerejű eszköz a szinkronizációra.

Példa:
A következő kódrész [8] két szinkronizált (amúgy aszinkron) csatornát definiál:

```
# let def fruit! f | cake! c = print_string (f^" "^c); print_newline ();
# ;;
val cake : <<string>>
val fruit : <<string>>
```

A példában az őrzött folyamat (`print_string (f^" "^c); print_newline ();`) csak akkor tüzelhet, amikor már mind a `fruit`-ra, mind a `cake`-re küldtünk üzenet [8].

```
# spawn { fruit "apple" | cake "pie" }
# ;;
-> apple pie
```

```
# spawn { fruit "apple" | fruit "raspberry" | cake "pie" | cake "crumble" }
# ;;
-> raspberry pie
-> apple crumble
```

A program két süteményt ír ki a konzolra, azonban a nem-determinisztikus végrehajtás miatt például az „`apple pie`, `raspberry crumble`” is lehetett volna az eredmény.

5.4.1. Alternatív minták

Csatorna-részhalmazokra vonatkozó, több alternatív mintát kapcsolhatunk össze az `or` művelettel [1]. Szintaxis: `let def` *join_minta*$_1$ `=` *folyamat*$_1$ `or … or` *join_minta*$_n$ `=` *folyamat*$_n$
Az `or`-ral összekapcsolt join-mintákban lévő csatornanevek között lehetnek átfedések, tehát egy csatorna egyszerre több join-mintában is részt vehet (azonban a csatorna természetesen csak egyszer jön létre). Megjegyzés: igazából, ha az `or`-ral összekapcsolt join-minták között nincsen olyan csatorna, ami egyszerre több mintában is részt vesz, akkor az összetett minta szemantikailag ekvivalens azzal a megoldással, mintha a mintákat külön-külön definiáltuk volna.

Az összetett minta akkor tüzelhet, ha valamelyik mintája tüzelhet; ha több minta is tüzelőképes, akkor a rendszer véletlenszerűen választ: a kiválasztott minta minden csatornájáról felhasznál egy üzenetet és a mintához tartozó folyamat kiértékelődik.

Példa [8]**:**

```
# let def apple!     () | pie! () = print_string "apple pie";
#   or    raspberry! () | pie! () = print_string "raspberry pie";
# ;;
val pie : <<unit>>
val apple : <<unit>>
val raspberry : <<unit>>
```

Látható, hogy a `pie` csatorna csak egyszer definiálódik, de két szinkronizációban is részt vesz.

5.5. A csatornák tulajdonságai

1. A szinkron és az aszinkron csatornák típusa különböző; ha rossz környezetben használjuk őket, akkor hibaüzenetet kapunk [8]:

```
# spawn { echo_szinkron 1 } ;;
Characters 7-14:
Expecting an asynchronous channel, but receive int -> unit
```

2. Mivel a csatornák elsődleges (*first class*) elemek, így küldhetők üzenetként egy másik csatornára (szinkronra és aszinkronra is), és a szinkron csatornáknál visszatérési értékként is használhatóak. Az olyan csatornát, amelynek (valamelyik) paramétere és / vagy értéke is csatorna **magasabb-rendű csatorná**nak hívjuk.

A következő példa [8] egy magasabb-rendű (szinkron) csatornát (`twice`) definiál, melynek paramétere és visszatérési értéke is csatorna: a paramétere egy egy-változós egy-értékű szinkron csatorna (vagy egy egy-változós egy-értékű függvény), visszatérési értéke pedig egy ugyanilyen típusú csatorna, amely azonban az üzeneteire kétszer is végre fogja hajtani a műveletet. (Fontos, hogy a bemeneti paraméter olyan csatorna vagy függvény legyen, amelynél a paraméter és eredmény ugyanolyan típusú.)

```
# let def twice f =
#   let def r x = reply f (f x) in
#   reply r
# ;;
val twice : ('a -> 'a) -> 'a -> 'a
```

3. Az előző példán látszik, hogy a csatornák is – mint szinte minden más típus – **polimorf**ak is lehetnek (tehát tartalmazhatnak típusváltozót – ugyanis JoCaml-ben csak paraméteres polimorfizmus van). A következő példán [8] látható, hogy `'a` típusváltozó egyszer `int`, máskor `string`:

```
# let def succ x = reply x+1 ;;
# let def double s = reply s^s ;;
#
# let f = twice succ in
# let g = twice double in
#    print_int (f 0) ; print_string (g "X") ;;
val succ : int -> int
val double : string -> string
-> 2XXXX
```

4. Érték korlátozás (*value restriction*): gyenge típusváltozó (`'_a`) használata. Az ilyen típusváltozó csak egyszer testesítődik meg (első használatkor eldől a tényleges típusa, utána már más típussal nem lehet használni). Erről már többször volt szó korábban (lásd 2.7.1. Polimorfizmus, 3.1.2. Referencia fejezetekben), de nézzük, hogy csatornák esetében ez mikor fordul elő.

Például nézzük a következő példát [8]: definiálunk egy olyan csatornát, ami egy adott aszinkron port-nevet hív meg egy adott paraméterrel; a csatorna folyamata csak akkor értékelhető ki, ha mind a port-név, mind a paraméter adott.

```
# let def port! p | arg! x = p x ;;
val arg : <<'_a>>
val port : <<<<'_a>>>>
```

Az őrzött folyamatból látszik, hogy a `p` egy tetszőleges paraméterű aszinkron csatorna és az `x` is tetszőleges típusú. Azonban a `p` paraméterének típusa az `x` típusával egyező kell, hogy legyen (ugyanis az `x` üzenetet küldjük el a `p` csatornára). Ezért a `p` és az `x` nem lehetnek tetszőleges típusúak, mert ez esetben a `spawn { port echo | arg "szoveg"}` is helyes lenne, ami nyilvánvalóan típushibás. (Ha mindkét típusban `'a` típusváltozó szerepelne, akkor ez minden használatkor más típust vehetne fel, például az `port echo` folyamatban `int` lenne, míg az `arg "szoveg"` folyamatban `string`.)
Éppen ezért, ha vagy a `p`, vagy az `x` típusa eldől, akkor ezt a típust le kell rögzíteni (az `'_a` gyenge típusváltozó minden előfordulásában felveszi az adott típust), így a másik résztvevő típusa is biztosítottan helyes típus lesz [8].

Csatornák esetében gyenge típusváltozót akkor kapunk, amikor az egy definícióban szereplő port-nevek osztoznak a típusváltozón.

5.6. Kivételkezelés párhuzamos program esetében

A szekvenciális program kivételkezelésről a 2.9. Kivételkezelés (szekvenciális program esete) című részben már szó volt. Most nézzük mi a helyzet a párhuzamos programok esetében:

A kifejezések kivételkezelése tehát `try` … kifejezés írásával lehetséges. (Mint arról már korábban szó volt a folyamatokhoz nem kapcsolhatunk `try`-t.) Ha a kivételt nem kezeljük le (vagy azért mert

nem is tartozott hozzá `try`, vagy azért, mert ugyan volt `try`, de a kivételt nem tudta lekezelni), akkor a viselkedés attól függ, hogy a hívó vár-e a kiértékelés eredményére.
Megjegyzés: a folyamatok kiértékelésére a hívó sosem vár, de elképzelhető, hogy egy kifejezés kiértékelésére sem vár, például abban az esetben, ha a kifejezés egy folyamat része (pl. a *`kifejezés`*`;`*`folyamat`* szekvencia egy folyamat, mert a szekvencia utolsó tagja folyamat – ebben az esetben az összetett folyamat kiértékelésére a hívó nem vár, de a kiértékelés magában foglalja a *`kifejezés`* kiértékelését is).

1. Aszinkron szálak kivételkezelése [8] (a hívó nem vár a kiértékelés eredményére): a kivételt **nem lehet elkapni**, a rendszer kiírja a kivételt a *standard output*-ra, majd az **aszinkron szál terminál**. A kivétel **semmilyen más szálat nem érint**.

Példa [8]**:**

```
# spawn {
#     { failwith "Kivetel"; print_string "Bye";}
#   | { for i = 1 to 10 do print_string "-" done; }
# } ;;
-> Uncaught exception: Failure("Kivetel")
-> ----------
```

A példában a „`failwith "Kivetel"; print_string "Bye";`” és a „`for i = 1 to 10 do print_string "-" done;`” folyamatok egymással párhuzamosan futnak. Az első folyamat váltja ki a kivételt, melynek hatására ennek a résznek a kiértékelése abbamarad (látható, hogy a `Bye` már nem íródik ki), azonban ez a kivétel a többi párhuzamosan futó részt „nem zavarja” (a másik szál így gond nélkül ki tudja írni a kötőjeleit).

2. Szinkron szálak kivételkezelése [8] (a hívó vár a kiértékelés eredményére – ez csak kifejezés esetén lehetséges; de mint láttuk kifejezés esetén sincs mindig így): ha a kivételt nem kezeljük le, akkor a kivételt **a hívó fogja megkapni** (az, hogy utána mit kezd a kivétellel, az már csak rajta múlik).

Példa [8]**:**

```
# let def die () = failwith "die"; reply ;;
#
# try
#   die ()
# with _ -> print_string "dead\n" ;;
val die : unit -> 'a
-> dead
```

A `failwith "die"; reply` folyamat kiértékelésére a `die` szinkron csatornára üzenetet küldő kifejezés vár, így ő kapja a kivételt, amelyet le is kezel (kiírja, hogy `dead`).

Ha megnézzük a Join-mintákat, akkor látható, hogy egy rész **kiértékelésére többen is várhatnak** (a Join-minta több szinkron csatornát is tartalmazhat, melyek mind az őrzött folyamat kiértékelésekor kapják meg a visszatérési értéküket; azonban ez az őrzött folyamat is kiválthat kivételt – még az előtt is, hogy bármelyik csatorna megkapná a visszatérési értékét). Mi történik ebben az esetben, ki kapja a kivételt? A válasz: mindenki. Tehát, ha a számítás eredményére többen is várakoznak, akkor a **kivétel többszöröződik**, és az összes – a kivételt kiváltó rész kiértékelésére váró – szálra dobódik [8].

Példa [8]**:**

```
# let def a () | b () = failwith "die"; reply to a | reply to b ;;
#
# spawn {
#     { (try a () with _ -> print_string "hello a\n"); }
#   | { (try b () with _ -> print_string "hello b\n"); }
# } ;;
val b : unit -> unit
val a : unit -> unit
-> hello a
-> hello b
```

Látszik, hogy a kivételt mind az `a`, mind a `b` csatornára üzenetet küldő kifejezés megkapta (és mindkettő le is kezelte).

6. A JoCaml nyelv elosztott programozást támogató nyelvi elemei

Az elosztott rendszer definíciója:

Több definíció is létezik a szakirodalomban, de egyik sem kielégítő és egyik sincs összhangban a másikkal. Íme az egyik: az elosztott rendszer független számítógépek gyűjteménye, melyek a felhasználó számára egyetlen összefüggő gépként látszanak [9]. (A definíció első része a hardverről szól: *az elosztott rendszerben lévő gépek egymástól függetlenek*, a második rész pedig a szoftverről: *a szoftvernek olyannak kell lennie, hogy a felhasználó úgy érezze, mintha csak egyetlen egy géppel lenne kapcsolatban*.)
Az elosztott rendszerben résztvevő gépek architektúrája és a rajtuk futó operációs rendszer is gépenként változó lehet.

Az elosztott program JoCaml-ben:

Az előző fejezet arról szólt, hogy hogyan lehet JoCaml-ben párhuzamos programot írni. Eddig azonban minden programunk egyetlen futtatható részből állt (a 2.10. Absztrakt adattípus megvalósítása JoCaml-ben: fordítási egységek című részben ugyan már volt szó arról, hogy hogyan készíthetünk több fordítási egységből álló programot, de eddig minden esetben a külön lefordított részeket a végén egyetlen futtatható programmá szerkesztettünk össze). Ebben a fejezetben arról lesz szó, hogy hogyan készíthetünk elosztott programokat, ahol a külön lefordított részeket nem szerkesztjük össze, hanem a programok külön futnak (esetleg különböző gépeken), de egymással együttműködve egyazon feladaton dolgoznak.

A programok különböző architektúrájú gépeken futva is tudnak együttműködni egymással, ami nemcsak egymás erőforrásainak használatát, aszinkron üzenetküldést és szinkronizációt jelent, hanem maguk a végrehajtás alatt lévő programok is átvándorolhatnak egyik gépről a másikra (mobil kódrészletek továbbítása). Az implementáció számos rendszerszintű folyamatot tartalmaz, melyek TCP/IP-n keresztül kommunikálnak a hálózaton [8].

6.1. Erőforrások megosztása név-szerver segítségével

Az első nagy kérdés az elosztott rendszereknél mindig az, hogy hogyan oldjuk meg a programok közötti kommunikációt. Mint az előző fejezetben kiderült, JoCaml-ben a párhuzamosan futó szálak csatornákon keresztüli üzenetküldéssel tudnak kommunikálni egymással. Mint látni fogjuk ez elosztott rendszerben lévő programok esetében is működik.

Ha a programok csatornán keresztüli üzenetküldéssel akarnak kommunikálni egymással, akkor további kérdések merülnek fel: (1) Egy program csatornái közül melyekhez lehet hozzáférni? (2) Hogyan tud egy program egy másik program csatornájára hivatkozni (3) Szükséges-e, hogy a programok ismerjék egymás tényleges fizikai elhelyezkedését?

A fenti kérdésekre a választ a **név-szerver** bevezetése adja (a név-szerver segítségével lehet a **távoli erőforrásokhoz** – például egy másik program csatornájához – **hozzáférni**):

1. Egy program futás közben felkínálhatja az erőforrásai (ami nem feltétlenül csatorna, lehet egyéb érték, például függvény, objektum, egész szám, rekord, lista stb.) egy halmazát a többi program számára, a többi program csak ezekhez az értékekhez férhet hozzá. Ez a „felkínálás” (megosztás) úgy történik, hogy a program regisztrálja a megosztásra szánt értékeket a név-szerverben (egy logikai néven, ami nem feltétlenül egyezik meg az érték programbeli azonosítójával).

2. Ha egy program egy másik program erőforrására akar hivatkozni, akkor azt a név-szervertől kell lekérnie (ehhez ismernie kell a regisztrációnál használt nevet). Megjegyzés: a lekérdező program lehet ugyanaz, mint a regisztráló program, bár a saját programban deklarált azonosítókhoz egyszerűbben is hozzá lehet férni.
3. Név-szerver használatával a programoknak elegendő, ha csak a név-szolgáltató fizikai címét ismerik. Mivel mind a felkínáló, mind a lekérő programok „jelentkeznek” a név-szervernél, így a rendszer nyomon tudja követni, hogy melyik program ténylegesen hol van, a programoknak ezzel egyáltalán nem kell törődniük.

Tehát JoCaml-ben a különálló programok között kezdetben nincs kapcsolat, a partnerek név-szolgáltató segítségével találnak egymásra [1]. Egy program alapértelmezésben csak a saját azonosítóihoz fér hozzá, egy másik programéhoz csak akkor, ha azt a másik program engedélyezte (megosztotta).

Egy fontos kérdés, hogy amikor a név-szervertől „lekérünk” egy értéket, akkor tulajdonképpen **mit** is **kapunk**: magát az értéket (ami így többszöröződik a rendszerben), vagy csak a referenciáját? Ez attól függ, hogy mit kérünk le: **egyszerű értékeknél és függvényeknél egy másolatot, míg csatornáknál és objektumoknál csak referenciát** kapunk [8]. (Az az egyszerű érték, amely nem tartalmaz olyan kifejezést vagy folyamatot, melynek kiértékelését csak meghívással lehet elérni; tehát például az `int`, a `string`, a lista, a rekord, a referencia egyszerű értékek, míg a függvények, a csatornák és az objektumok nem).

Elosztott rendszerben a használat szempontjából ez az egyetlen lényeges **különbség a függvények és a szinkron csatornák között**. Tehát amikor egy függvényt küldünk egy távoli gépre, akkor a függvény kódja és lokális változóinak értékei is többszöröződnek és a távoli gépen is lesz egy példányuk. Ezután a távoli gépről érkező minden hívás lokálisan, a távoli gépen fog végrehajtódni. Ezzel ellentétben amikor egy szinkron port-nevet küldünk egy távoli gépre, akkor csak a név (a referencia) megy át, és a névre érkező hívások továbbítódnak arra a gépre, ahol a csatorna ténylegesen van (ez alapértelmezésben az a hely, ahol definiálták, de mint nemsokára látni fogjuk, ez a program futása során változhat), úgy, mint egy távoli eljáráshívásnál [8].

Mivel objektumok esetében is referenciát kapunk, ezért ha a másik gépről – az objektum valamelyik metódusának használatával – megváltoztatjuk valamelyik adattag értékét, akkor ez a változás az eredeti gépen is érezhető lesz; és fordítva: ha az eredeti gépen változik egy adattag értéke, akkor a változást a másik gépről is látjuk. Tehát a két program ugyanazt az objektumot fogja kezelni.

Megjegyzés: ez az átadásbeli különbség az egyszerű értékek és függvények, valamint a csatornák és objektumok között nem a név-szerver sajátossága, hanem az azonosítók implementálásából ered. Az egyszerű értékeknél és a függvényeknél ugyanis az azonosító magát az értéket jelöli, míg a csatornáknál és objektumoknál csak egy értékre mutató pointert. Mivel azonban JoCaml-ben explicit pointer-kezelés nincs, így ezt a tárolási módot a rendszer – az átadás kivételével – elrejti előlünk.

Az előbbiekből már jól látszik, hogy a csatorna a programok közötti kommunikációra egy megfelelő eszköz. A csatornát regisztrálhatjuk a név-szerverben, amelyet ha egy másik program lekér, akkor az eredeti csatornához szerez egy hozzáférést. Így a program tud egy másik program csatornájának üzeneteket küldeni.
A programok közötti kommunikációra a függvények nem alkalmasak, mert név-szervertől való lekérésüktől egy másolat-példányt kapunk, így sose tudunk egy másik gépen lévő függvényre hivatkozni.

A névszervertől lekért értékeket teljesen ugyanúgy kell kezelni, mint a helyileg deklaráltakat, tehát a használat módjának szempontjából semmi különbség nincs.

6.1.1. A név-szerver használata

Maga a név-szerver a felhasználó számára nem más, mint egy táblázat, amely a megosztásra szánt értékeket, és típusaikat tárolja. A típus tárolására azért van szükség, mert a JoCaml erősen (és statikusan) típusos, ezért a fordítónak kötelessége ellenőrizni, hogy a név-szervertől lekért azonosítókat típus-helyesen használják-e; ehhez viszont elengedhetetlen a név-szerveren tárolt azonosítók típusának ismerete.
A név-szerverben minden értékhez tartozik egy név, amivel hivatkozni lehet rá (lásd hash-tábla).

A név-szervert az `Ns` modul (egy beépített könyvtár) valósítja meg. Az `Ns` modul [8] két legfontosabb függvénye a következő:

- `register` *`név azonosító metatípus`*: ennek a függvénynek a segítségével lehet egy adott azonosító értékét egy megadott néven regisztrálni a név-szerverben. A *`metatípus`*-nak az *`azonosító`* típusával egyeznie kell.
  A `register` típusa: `string -> 'a -> 'a metatype -> unit`
- `lookup` *`név metatípus`*: ezzel lehet egy adott néven regisztrált értéket lekérni a táblázatból. A szerveren tárolt típusnak a *`metatípus`*-sal egyeznie kell; ha a típusok nem egyeznek, akkor az `Ns.lookup` egy `TypeMismatch` kivételt vált ki. A regisztrált nevekhez bárki hozzáférhet. A `lookup` típusa: `string -> 'a metatype -> 'a`

A függvények egyik paraméterének típusa egy eddig ismeretlen típus: `'a metatype`. Ez jelöli a regisztrálandó, illetve lekérendő érték típusát. A típust a `vartype` kulcsszóval lehet beállítani, szintaxisa: `(vartype :` *`típus`* `metatype)`, vagy csak egyszerűen `vartype` [8] (ez utóbbi esetben a rendszer az azonosítónak a környezetéből (használatából) kikövetkeztetett típusát adja át; mivel a JoCaml a legtöbb azonosító típusát le tudja vezetni, ez általában elegendő is; bár látni fogunk erre ellenpéldát is).

A névszerver **csak monomorf** (azaz nem polimorf) típusokkal dolgozik.

Az név-ütközések elkerülése érdekében a rendszer az `Ns.user` (típusa `string ref`) lokális azonosítót is hozzáfűzi a regisztrálandó névhez, ezért ennek mindkét oldalon egyeznie kell. Az `Ns.user` alapértelmezésben a lokális user-nevet tartalmazza (ez a belépéskor használt felhasználói név), de – mivel ez egy referencia – a program futása közben is meg lehet változtatni [8]. Az `Ns.user` használatával több értéket is lehet ugyanazzal a névvel (de más-más user-névvel) regisztrálni, a lekérdező program a user-név változtatásával dönthet arról, hogy melyiket is szeretné használni (sőt, egy program futása során különbözőképpen is dönthet).
Ha ***teljes*** névütközés van (mind a regisztrált név, mind a user-név egyezik), akkor – mint JoCaml-ben névütközéskor megszokott – a később regisztrált név felülírja a korábban regisztráltat.
Megjegyzés (egészen pontosan milyen néven regisztrálódnak az értékek): tegyük fel, hogy az `Ns.user` értéke **`user`**, a regisztrálandó azonosító neve pedig **`név`**. Ebben az esetben a rendszer a **`user`***`elválasztó`***`név`** néven fogja regisztrálni az értéket, ahol az *`elválasztó`* az adott operációs rendszerbeli könyvtár-elválasztó jel (Unix-nál /, WinNt-nél \\, Mac-nál :). Az elválasztójel megválasztásának nincs különösebb oka, bármi más olyan jel is lehetett volna, ami nem szerepelhet azonosítóban.

6.1.2. Példa

A következő példa [8] két folyamatot tartalmaz, melyek párhuzamosan futnak. Az egyik folyamat bejegyzi az `f` (lokálisan definiált csatorna) közös erőforrást `square` néven, a másik (aki nincs benne az `f` hatókörében) pedig megkeresi és használja:

```
# spawn { let def f x = reply x*x in Ns.register "square" f vartype; } ;;
# spawn { let sqr = Ns.lookup "square" vartype in print_int (sqr 2); exit 0; } ;;
File "ex1.ml", line 3, characters 36-43:
Warning: VARTYPE replaced by type
( int -> int) metatype
File "ex1.ml", line 1, characters 57-64:
Warning: VARTYPE replaced by type
( int -> int) metatype
-> 4
```

Természetesen a név-szerver használatának igazából csak akkor van értelme, amikor a két folyamat két különálló program részeként két különböző gépen fut.

6.1.3. Példa – több program egyidejű futtatása

A következő példa bemutatja a név-szerver használatát két külön gépen. Mint látni fogjuk a névszervert teljesen ugyanúgy kell kezelni abban az esetben is, ha a két programrész két különálló program (esetleg két különböző gépen).

Tegyük fel, hogy több szám négyzetét kell kiszámolnunk, és az **egyik gép**ünk főleg az egész számok négyzetének kiszámításában jó, ezért jó lenne, ha a négyzetre-emelést ez a gép végezné. Ezért ezen a gépen definiálunk egy szinkron csatornát, ami híváskor a négyzet-számítás mellett ki is ír valamit – hogy nyomon követhessük, hogy mikor mi történik –, majd regisztráljuk `square` kulcsszó alatt. Megjegyzés: ebben az esetben nem mindegy, hogy szinkron csatornát, vagy függvényt definiálunk, ugyanis a szinkron csatorna akkor is az eredeti helyén hajtódik végre, ha a hívás egy másik gépről jött, míg a függvények mindig lokálisan hajtódnak végre. Mivel jelen esetben az a cél, hogy a négyzet-számítás mindig a szerveren hajtódjon végre, míg a hívás mindig a klienstől jön, így itt csak csatornát használhatunk.

A szerver program [8]**:**

```
let def f x =
   print_string ("["^string_of_int(x)^"] "); flush stdout; reply x*x in
   Ns.register "square" f vartype
;;
Join.server ();;
```

A `Join.server` hatására a program az összes lokális utasítása végrehajtása után is futni fog. Erre azért van szükség, mert így fogadni tudja a később jövő távoli hívásokat (négyzet-számítási kéréseket). A névszerver ugyanis – mint az a nevéből is látszik – csak az azonosítók neveit tárolja az értékét (jelen esetben a csatorna törzsét) nem.

Nézzük, hogy tudja egy **másik gép** (a kliens) távolról használni a szerveren lévő csatornát. Ehhez először le kell kérnie a név-szervertől a `square` néven regisztrált csatornát, majd ezután ugyanúgy használhatja, mintha a csatornát lokálisan definiálták volna.

Kliens program [8]**:**

```
let sqr = Ns.lookup "square" vartype ;;
let def log (s,x) =
  print_string ("q: "^s^"= "^string_of_int(x)^"\n"); flush stdout; reply
;;
let def sum (s,n) = reply (if n = 0 then s else sum (s+sqr(n),n-1)) ;;

log ("sqr 3",sqr 3);
log ("sum 5",sum (0,5));
exit 0
```

Lássuk, mi lesz a számítás eredménye! Amikor egy folyamat definiál egy új port-nevet, akkor ezt lokálisan teszi, azaz, a hozzá tartozó őrzött folyamat ugyanazon a helyen fog végrehajtódni, ahol definiálták. Minden `square`-re érkező hívás tehát egy távoli hívás a szerver gépre. A képernyőre került kiírások elárulják a folyamatok aktuális elhelyezkedését: az `f` (ami a kliens gépen `sqr` névre hallgat) mindig a szerverre, míg a `log` mindig a kliensre ír.

A szerver kimenete:

```
-> [3] [5] [4] [3] [2] [1]
```

A kliens kimenete:

```
-> q: sqr 3= 9
-> q: sum 5= 55
```

Megjegyzés az automatikus típuslevezetéshez:

Ha az előző programot elkezdjük beírni a *top-level*-be, akkor egy érdekes hibaüzenetet kapunk a következő utasításnál [8]:

```
# let sqr = Ns.lookup "square" vartype ;;
-> Failure: Failure("vartype: no polymorphism is allowed")
```

Mivel lekérdezéskor még nem tudni az `sqr` típusát, így hibát a kapunk, mert a név-szerverben az `sqr` mellett tárolt típus (`int->int`) nem egyezik az `sqr` típusával (amiről – mivel semmit sem tudni róla – a fordító feltételezi, hogy `'a`). A megoldás az, hogy expliciten megadjuk az `sqr` típusát. Erre két lehetőség van:

1. `let sqr = Ns.lookup "square" (vartype : (int->int) metatype);;` vagy
2. `let (sqr : int -> int) = Ns.lookup "square" vartype;;`

A két megoldás – bár ugyanazt eredményezik –, mégis különbözőek. Az első esetben közvetlenül megadtuk, hogy egy `int->int` típusú `square` nevű értéket szeretnénk lekérni (a fordító ellenőrzi, hogy a `square`-t tényleg ezzel a típussal regisztrálták-e, majd miután ez egyezik, már tudni fogja, hogy az `sqr` típusa is `int -> int`). A második esetben az automatikus típuslevezetést „segítettük” egy kicsit: közvetlenül megmondtuk, hogy az `sqr` típusa is `int -> int`, így a fordító tudja, hogy a `square`-nek is ilyen típusúnak kell lennie („sima” `vartype`).

Ha a kódot nem a *top-level*-be írjuk, hanem az egész programot a fordítóval fordítjuk le, akkor ez a hiba nem jelentkezik, mert a kódban később (az `sqr` használatakor) a típusa egyértelművé válik.

6.1.4. Távoli erőforrás-hozzáférés másik módja: erőforrás elküldése üzenetben

Az egyes gépek nemcsak a névszerver használatával férhetnek hozzá egy másik program erőforrásaihoz, hanem azt üzenetben is megkaphatják. Ennek a módszernek a lényege, hogy az a program, amelyik használni szeretné az erőforrást, létrehoz egy csatornát, melynek paramétere a használni kívánt erőforrás lesz, a törzse (őrzött folyamata) pedig az erőforrás használata. A

program regisztrálja ezt a csatornát a név-szerverben, a másik program (akinél az erőforrás van) pedig lekéri ezt a csatornát, és meghívja az erőforrással. (Itt kötelező a csatorna használata, mert függvény esetén az erőforrás használata az erőforrást birtokló gépen történne).

Mindkét módszer (név-szervertől való lekérés, illetve üzenetben történő megkapás) esetében a program tudja használni egy másik program / gép erőforrását, a különbség csak az, hogy a felhasználást melyik program kezdeményezi. Látszik, hogy név-szerverre mindkét esetben szükség volt, de míg az egyik esetben közvetlenül az erőforrást regisztráltuk, addig az utóbbi módszernél csak egy közvetítő csatornát. A felhasználó gép egyik esetben sem tudta, hogy az erőforrás fizikailag hol van, és egyáltalán lokális-e, vagy távoli.

Azonosítók üzenetben való elküldésére is ugyanazok a szabályok érvényesek, mint név-szerver használatakor: egyszerű azonosítók és függvények esetében a fogadó egy másolatot kap (és a függvény végrehajtása a fogadó gépre nézve lokálisan fog történni), míg csatornáknál és objektumoknál csak referenciát (így a használatuk egy távoli hívás lesz).

6.2. Absztrakt hely és mobilitás

Motiváció:

Név-szerver és csatornák segítségével már tudunk elosztott alkalmazásokat fejleszteni JoCaml-ben, azonban idáig a folyamatok és a kifejezések elhelyezkedése teljesen statikus volt. Bizonyos esetekben ennél finomabb irányítás kell. Például, amikor az előző példában négyzetek összegét számoltuk, akkor a ciklusban minden `sqr`-hívás két hálózati üzenetküldést vont maga után (egy üzenet a kérés, egy üzenet a válasz). Jobb lenne, ha az egész ciklus azon a gépen futna, amelyik a négyzetet számolja. Megoldásként módosíthatnánk a szerver programot is, azonban elképzelhető, hogy legközelebb valamilyen más szempont szerint kell sok négyzet-számolást végezni. Ezért jobb lenne egy olyan megoldás, ahol nem a szerveren futó programot kell módosítani, hanem csak a klienst, de a számolás mégis a szerveren történjen [8]. Egy kézenfekvő megoldás, hogy maga a kliens program a négyzet-számítás előtt vándoroljon át a szerverre, és ott végezze el a számításait (majd – ha szükséges – jöjjön vissza). JoCaml-ben ezt pontosan így meg is valósíthatjuk.

JoCaml-ben a folyamatok tehát vándorolhatnak egyik gépről a másikra. A kód mozgása során a JoCaml nyelv absztrakt eszközöket biztosít a kompozíció, a kommunikáció és a belső állapot helyfüggetlen kezelésére és távoli szolgáltatások igénybevételére [1].

6.2.1. Az absztrakt hely (location)

Az absztrakt hely csatorna-definíció(ka)t és egy folyamatot fog össze, névvel rendelkezik és a nyelv teljes jogú (*first class*) értéke (tehát lehet függvény paramétere, eredménye, el lehet küldeni egy csatornára üzenetként, lehet egy szinkron csatorna visszatérési értéke, meg lehet osztani a név-szerver segítségével, stb). Megjegyzés: egy absztrakt hely név-szerverben történő regisztrálásakor és üzenetben történő elküldésekor – hasonlóan a csatornákhoz és az objektumokhoz – a fogadó egy referenciát kap.

Az absztrakt hely definíciójának **szintaxis**a:

```
let loc név [def csat_definíció] [do { folyamat }]
```
[8]

A definíció több csatornát is definiálhat (`def` $csat_1$ `=` $törzs_1$ `and` $csat_2$ `=` $törzs_2$ …), tartalmazhat join-mintát és alternatív mintákat is (`def` $minta_1$ `=` $törzs_1$ `or` $minta_2$ `=` $törzs_2$ …), pont mintha a csatornát az absztrakt helytől függetlenül definiáltuk volna (csak a definíciót nem `let def`, hanem simán `def` vezeti be). A *`folyamat`* tetszőleges lehet, akár az üres folyamat is.

A `folyamat`-ot egy inicializáló résznek lehet tekinteni, amely kiértékelése az absztrakt hely definíciójának kiértékelésénél egyből megkezdődik. Mivel ez egy folyamat, így nem meglepő, hogy kiértékelése a program további részének kiértékelésével párhuzamosan folyik [8].

Az absztrakt helyen belül lévő `csat_definíció` részben deklarált csatornák hatóköre nem csak az absztrakt helyre, hanem a hely deklarációja utáni egész programrészre kiterjed [8] (pont úgy, mintha a csatornákat az absztrakt hely *előtt* definiáltuk volna).

Az absztrakt hely **típus**a mindig `Join.location` (függetlenül attól, hogy a hely milyen csatornákat és folyamatokat tartalmaz).

Példa [8]**:**

```
# let loc this_location
#   def  square x = reply x*x
#    and cubic  x = reply (square x)*x
#   do {
#       print_int (square 2);
#   };;
# print_int (cubic 2) ;;
val this_location : Join.location
val cubic : int -> int
val square : int -> int
-> 48
```

A példában látható, hogy a `this_location` nevű absztrakt helyen belül deklarált `cubic` függvényt a helyen kívülről is meg lehet hívni. Mivel a hely inicializáló folyamatának (`print_int (square 2);`) kiértékelése az absztrakt hely deklarációja utáni programrész kiértékelésével párhuzamosan folyik, ezért eredményül vagy `84`-et, vagy `48`-at kaphatunk (előre nem tudható).

Absztrakt helyeket egymásba is ágyazhatunk, mert a folyamat is tartalmazhat lokális hely-deklarációt (`let loc … in …`). Az absztrakt hely felépítése statikus: a definíciók és a folyamatok véglegesen (statikusan) ahhoz az absztrakt helyhez tartoznak, ahol a forráskódban szerepelnek; amint egy absztrakt helyet létrehoztunk, már nincs mód arra, hogy ebbe kívülről új deklarációkat és folyamatokat helyezzünk el.

Ez idáig az absztrakt hely csak egy deklaráció és egy folyamat összefogása, mely utána egy egységként kezelhető. Azonban az absztrakt hely ennél sokkal több: **a mobil kód alapegysége**.

6.2.2. Mobilitás

A JoCaml nyelvben a mobil kód megfelelője az ágens. Míg a folyamatok és a definíciók statikusan az absztrakt helyükhöz tartoznak, addig egy absztrakt hely képes elvándorolni (*migrate*). A vándorlást a vándorló helyen belüli folyamat indítja, és az absztrakt hely egy egységként (a csatornáival és a vándorlást kezdeményező folyamatával együtt) vándorol el. A vándorlás eredményeképpen a mozgó hely a cél-helynek egy al-helye lesz.

Az absztrakt helyek több célra is használhatók [8]**:**

1. **Mobil ágensként**: az absztrakt hely a folyamatán belül kiadott `Join.go` utasítás hatására képes átvándorolni egy másik gépre.
2. **Célcímként**: egy ágens vándorlásakor sohasem egy megadott IP címre vándorol (hiszen így nem kapnánk könnyen hordozható programot), hanem címként egy másik programban lévő absztrakt helyet kell megadni (ehhez természetesen hozzá kell férni („látni kell”) a másik programban lévő absztrakt helyet – ezt például a név-szerver segítségével érhetjük el). A program majd az absztrakt hely (éppen) aktuális fizikai helyére vándorol (tehát nem a

definíciójának helyére). Megjegyzés: a `Join.go` paramétere tehát egy absztrakt hely (`Join.go : Join.location -> unit`).

3. **Az előző kettő kombinációjaként.**

A következő példa a fenti négyzet-számolós feladat egy ágens-alapú változata [8]. A számoló oldalon (ahol a négyzet-számítás történik) készítünk egy `here` nevű üres helyet, majd regisztráljuk a név-szerverben (ezt a helyet csak *célcímként* fogjuk használni, hogy a kliens ide tudjon találni).

```
let def f x =
   print_string ("["^string_of_int(x)^"] "); flush stdout; reply x*x in
   Ns.register "square" f vartype
;;
let loc here do {};;
Ns.register "here" here vartype;

Join.server ();;
```

A kliens oldalon készítünk egy `mobile` nevű absztrakt helyet, ami több `square` hívást is tartalmaz. A `mobile` ágens lekérdezi a névszolgáltatótól a `here` ágens adatait, majd átvándorol a `here` aktuális fizikai helyére (a szerverre), és ennek mindenkori helyén, alágensként végzi el a számításokat (így az összes `square` hívás lokális lesz) [8]:

```
let loc mobile
   do {
      let here = Ns.lookup "here" vartype in
      go here;
      let sqr = Ns.lookup "square" vartype in
      let def sum (s,n) = reply (if n = 0 then s else sum (s+sqr n, n-1)) in
      let result = sum (0,5) in
      print_string ("q: sum 5= "^string_of_int result^"\n"); flush stdout;
   }
```

A kliens a `go here` kifejezés hatására vándorol át a szerver gépre, ahol a `here` al-helye lesz. Ezután a teljes számítás (a név-szervertől a `square` lekérdezése, az `sqr` és a `sum` hívása is) lokális a szerver gépre. A szerver és a kliens gépek között csak három üzenetváltás történik: a `here` lekérése, a hozzá tartozó válasz és a vándorlás.

6.2.2.1. Hatókör:

Az absztrakt hely a folyamaton kívül csatornadefiníciókat is tartalmaz, amely a hellyel együtt vándorol (így ha a hely folyamatában egy ilyen csatornára történő hívás van, az minden esetben lokális hívás marad).

A vándorlás a hatóköri szabályokat nem befolyásolja.

1. A helyen belül definiált csatornák hatóköre nem csak a helyre, hanem az utána következő programrészre is kiterjed. Azonban mi történik a csatornák hatókörével, ha a hely elvándorol? Válasz: semmi, a hatókör nem változik. A helyen belüli csatornák tehát a hely elvándorlása után is ugyanúgy hívhatók, csak akkor esetleg ez már nem egy lokális, hanem egy távoli hívás lesz.
2. Fordítva: az absztrakt helyen belüli folyamat tartalmazhat lokális hívásokat a hely definíciója előtt (de ugyanabban a programban) lévő csatornákra is (hiszen a deklaráláskor az absztrakt hely benne volt az előtte definiált csatornák hatókörében). A hely a vándorlása után a másik gépen folytatja a folyamatának kiértékelést, azonban mi lesz így ezekkel a hívásokkal? A válasz most is: semmi, a hatókör most sem változik. A helyen belülről tehát a vándorlás után is ugyanúgy látszanak a deklarációkor látszó azonosítók, azonban ezek hívása esetleg már nem lokális, hanem távoli lesz.

Például, nézzük a következő kliens programot [8]:

```
let sqr  = Ns.lookup "square" vartype
let here = Ns.lookup "here" vartype
let def done1! () | done2! () = exit 0; ;;
let def log (s,x) =
   print_string ("agent: "^s^" is "^string_of_int x^"\n");
   flush stdout; reply;;

let loc mobile
   def quadratic x = reply sqr(sqr x)
   and sum (s,n,f) = reply (if n = 0 then s else sum(s+f n, n-1, f))
   do {go here;
       log ("sum ( i^2 , i= 1..10 )",sum (0,10,sqr)); done1 ()
   };;

spawn { log ("sum ( i^4 , i= 1..10 )", sum (0,10,quadratic)); done2 () };;
```

A `mobile` hely a folyamatának kiértékelése (ami az utána lévő folyamattal párhuzamosan történik) közben elvándorol a szerverre. Ez azonban `quadratic` és a `sum` hatókörét nem befolyásolja, azok ugyanúgy használhatók a `mobile`-t követő – és a kliens gépen maradó – kifejezésekben, mintha a `mobile` nem ment volna sehova. Amint az ágens megérkezik a szerverre, a `sum` és a `quadratic` távoli hívás lesz, mintha mindkét függvényt a szerveren definiálták és regisztrálták volna.
Fordítva, a mobil ágens törzséből jövő `log` hívások a kliensre érkeznek, így az eredmény itt jelenik meg (a `log` nem a `mobile` helyen belül volt, így nem is vándorolt el). A `log` az ágens elvándorlása után is teljesen ugyanúgy használható, mint előtte.

A kliens program eredménye:

```
-> agent: sum ( i^4 , i= 1..10 ) is 25333
-> agent: sum ( i^2 , i= 1..10 ) is 385
```

Megjegyzés: a `done1` és a `done2` üzenetek csak azt jelzik, hogy melyik rész számítása fejeződött már be; így biztosítjuk a program terminálását.

JoCaml-ben tehát az elhelyezkedés és a hatókör egymástól független: egy ágens pontosan ugyanazokat a műveleteket hajthatja végre, függetlenül attól, hogy aktuálisan hol helyezkedik el. A kommunikációs csatorna tehát fennmarad helyüket időközben változtató ágensek között is, és az ágensek abból az állapotból folytatják működésüket, ahol a helyváltoztatás előtt voltak [1].

6.2.3. Mobil objektum

JoCaml-ben az objektum egyben absztrakt hely is [8]**.**

Az objektumok rendelkeznek az absztrakt helyek összes képességeivel, tehát el tudnak vándorolni egy másik helyre és lehetnek egy vándorlás célpontjai is. Az absztrakt helyhez hasonlóan, az objektum vándorlásakor az adattagok és a metódusok is átvándorolnak, ezért egy metódus-hívás mindig azon a gépen fog lefutni, ahol az objektum aktuálisan van, ami nem szükségszerűen egyezik meg azzal a géppel, ahol a hívás történt, vagy azzal, ahol az objektumot definiálták.

Objektum vándorlása: az objektum a metódusában kiadott `Join.go` *`absztrakt_hely`* hatására tud elvándorolni az *`absztrakt_hely`* aktuális fizikai címére [8].

Objektum, mint a vándorlás célpontja: egy objektum lehet vándorlás célpontja is, olyat azonban nem írhatunk le, hogy `Join.go` *`objektum`*, ugyanis a `Join.go` típusa `Join.location -> unit`, tehát paraméterként csak absztrakt helyet fogad el. Az objektum típusát pedig nem lehet absztrakt hely típusra korlátozni, hiszen érezhető, hogy nincsen közöttük altípus-reláció. Ezért létezik egy másik primitív is, ami viszont csak objektumhoz tud vándorolni: `Join.goo` [8] (típusa:

`< > -> unit`). Megjegyzés: az automatikus típuskonverzió hiánya miatt a `Join.goo` használatakor az objektumot expliciten korlátozni kell az üres objektum-típusra – kivéve persze magát az üres objektumot).
Ahhoz, hogy ezt az objektumot címként használni lehessen, le kell kérni a név-szervertől, vagy üzenetben kell megkapni (mindkét esetben csak az objektum referenciáját kapjuk meg, nem magát az objektumot).

Példa: az előbbi négyzet-számolós feladat; a szerver legyen az előbbi program, a kliens pedig a következő:

```
class szamolo l as self =
   val elemek = l
   val sqr = Ns.lookup "square" (vartype : (int -> int) metatype)
   method szerverre =
      let here = Ns.lookup "here" vartype in Join.go here
   method debug = print_string "itt vagyok\n"
   method negyzet_osszeg =
      let rec meg s = function
           []     -> s
         | e :: l -> meg (s + sqr e) l
      in self#szerverre; self#debug; meg 0 elemek
end;;

let p = new szamolo [1; 2; 3; 4; 5];;
print_string ("negyzetosszeg 1..5 = "^string_of_int p#negyzet_osszeg^"\n");
  flush stdout;;
```

Szerver kimenete:

```
-> itt vagyok
-> [1] [2] [3] [4] [5]
```

Kliens kimenetele:

```
-> negyzetosszeg 1..5 = 55
```

Megjegyzés: a `here` absztrakt hely is lehetett volna objektum. Ekkor a szerveren a „`let loc here do {};;`” helyett „`class oszt () = end;; let here = new oszt ();;`”, a kliensen pedig „`Join.go here`” helyett „`Join.goo here`” kellett volna.

6.3. Meghibásodás és az ezzel kapcsolatos problémák megoldása

Előfordulhat, hogy az elosztott számítás néhány része nem jár sikerrel, például azért mert a gépet, amin futott kikapcsolták. Hogyan lehet kezelni az ilyen helyzeteket? A legegyszerűbb megoldás az lenne, hogy amikor ilyet észlelünk, akkor az egész számítást megszakítjuk, de ez nem reális, ha sok gépen dolgozunk. Jobb lenne, ha a programunk magától felismerné az ilyen hibákat, és megfelelő intézkedéseket tenne (például jelentené a hibát és lezárná a számítás érintett részeit, vagy tenne még egy kísérletet egy másik gépen). Ez a fejezet arról szól, hogy milyen eszközöket nyújt a JoCaml az ilyen esetek kezelésére.

6.3.1. Absztrakt hely leállítása

Egy absztrakt hely folyamatán vagy valamelyik csatornájának törzsén belül elhelyezett `halt ()` kifejezés kiértékeléskor a rendszer leállítja az absztrakt helyet és rekurzívan minden al-helyét is [8]. A *leállítás* egész pontosan azt jelenti, hogy a helyen belüli összes folyamat kiértékelése abbamarad, és a csatornák többé már nem tüzelnek.
Itt kap nagy jelentőséget, hogy vándorláskor a mozgó hely a cél-helynek al-helye lesz, mert emiatt, ha a cél-helyet leállítjuk, akkor az odavándorolt (és a leállítás pillanatában is még ott tartózkodó) absztrakt helyek (vagy objektumok) is leállnak.

Megjegyzés: a `halt ()` primitívet nem csak egy absztrakt helyen belül használhatjuk, hanem a program minden olyan helyén, ahol kifejezés előfordulhat. Ha a `halt ()`-ot nem egy absztrakt helyen belüli folyamatban használjuk, hanem valami egyéb helyen, akkor kiértékelésekor csak az adott végrehajtási szál kiértékelése szakad meg. Az absztrakt helyen belüli használatnak az a különlegessége, hogy itt a `halt ()` az egész helyet (sőt még az al-helyeit is) leállítja.

Mivel az objektumok is absztrakt helyek, ezért nem meglepő, hogy az objektum valamelyik metódusában alkalmazott `halt ()` az egész objektumot leállítja. A leállított objektum nem válaszol többé az üzenetekre.

6.3.2. Absztrakt hely leállásának felismerése

A `fail` primitív használatával fel lehet ismerni, ha egy absztrakt hely leállt. Szintaxis: `fail` *`there`*`; P` ahol a *`there`* a megfigyelt absztrakt hely. Amikor `P` folyamat elindul, akkor a *`there`* már garantáltan leállt [8]. A `P` folyamatot egy hibakezelő eljárásnak lehet tekinteni.

Az aktuális implementációban a leállást csak akkor lehet felismerni, ha a leállítás **ugyanabban a futó programban** a `halt ()` primitív használatával történt (ahol vagy magát a megfigyelt absztrakt helyet állították le, vagy egy olyan helyet, aminek ő az al-helye – de a lényeg, hogy ugyanabban a programban). A `halt ()` használata tehát nem fog elindítani egy hozzá tartozó `fail`-t egy másik programban [8]. A másik programban lévő (vagy oda vándorolt) absztrakt helyek leállását csak akkor lehet a `fail` segítségével felismerni, amikor a megfigyelt absztrakt helyet tartalmazó **futó program befejeződik** (például `exit 0` primitív) vagy elérhetetlenné válik (például megszakad vele a hálózati kapcsolat). A `fail` utáni kifejezés tehát nem a külső absztrakt hely leállása, hanem csak a tartalmazó program leállása után fog lefutni.

A jelenlegi implementációban a hibafelismerés csak részben implementált, ezért nincs garancia, hogy ha egy program abnormálisan terminál, akkor ennek a helynek a hibáját detektálják is [8].

6.3.3. Példák

A példa [8] tartalmaz egy bizonytalan absztrakt helyet (`agent`), ami bármikor leállhat. Miután a leállás bekövetkezett kiírunk egy tájékoztató üzenetet. A példában megkíséreljük használni ennek a helynek egy portját (`say`). Megjegyzés: mivel a hívás nem biztos, hogy visszatér (például ha a hely leállt), ezért a kiértékelést csak elindítjuk, de nem várunk a hívás befejezésére (*say "vmi";;* helyett *spawn { say "vmi"; } ;;*).

```
# let loc agent
#   def say s = print_string s; reply
#   do { halt (); }
# ;;
#
# spawn { say "it may worked before.\n"; } ;;
# spawn {
#   fail agent; print_string "the location stopped\n";
#   say "it never works after\n";
# };;
val agent : Join.location
val say : string -> unit
-> the location stopped
```

A következő példa [8] még trükkösebb. Egyrészt nem a hely állítja le magát, hanem elvándorol egy olyan helyre, ami leáll; másodszor, a `halt ()` csak kívülről indítható egy `kill ()` üzenettel. Mivel az `agent` helyen belül nincs `halt`, így csak a `fallible` hely leállása miatt állhat le, ezért a `fail agent;` magával vonja, hogy a `fallible` is leállt.

```
# let loc fallible
#   def kill! () = halt ();
# ;;
#
# let loc agent
#   def say s = print_string s; reply
#   do { go fallible; }
# ;;
#
# spawn { say "it always works.\n"; kill () };;
# spawn { say "it may worked before.\n"; };;
# spawn {
#   fail agent; print_string "both locations stopped.\n";
#   say "it never works after.\n";
# };;
val fallible : Join.location
val kill : <<unit>>
val agent : Join.location
val say : string -> unit
-> it always works.
-> it may worked before.
-> both locations stopped.
```

7. Join-kalkulus kifejezőereje

A Join-kalkulus csatornái erős kommunikációs eszközt adnak a kezünkbe, a join-mintákkal pedig szinkornizálhatjuk a párhuzamosan futó programrészeket. A kommunikáción és a szinkronizáción kívül a join-kalkulus segítségével megvalósíthatunk többféle programozási stílust, adatstruktúrát és vezérlési szerkezetet is.

7.1. Funkcionális stílusú programozás

Mivel a szinkron csatornák típusa ugyanaz, mint a függvényeké és úgy is viselkednek, mint a függvények (kivéve a vándorlást), ezért szinkron csatornák használatával lehetőség van funkcionális stílusú programozásra is. Egy csatorna lehet rekurzív is, tehát az őrzött folyamatából saját magának is küldhet üzenetet (a függvények csak a `rec` kulcsszó használatával lehetnek rekurzívak). A *Fibonacci* példa:

```
# let def fib n =
#   if n <= 1 then reply 1
#   else reply fib (n-1) + fib (n-2)
# ;;
#
# print_int (fib 10);;
val fib : int -> int
-> 89
```

7.2. Objektum orientált stílusú programozás

7.2.1. Példa: számláló

Konkurens környezetben használt számlálóra példa a `count` aszinkron csatorna [1], amelyre küldött üzeneteket vagy az `inc`, vagy a `get` szinkron csatornákra érkező üzenetekkel szinkronizálunk. Egy kezdeti `count 0` üzenet után az így kapott konkurens rendszer az `inc` üzenet hatására növeli a számlálót, és önmagának küld újabb `count` üzenetet, abban eltárolva a számláló aktuális értékét. A `get` üzenet hatására visszaadja a számláló értékét, egyben önmagának egy `count` üzenetet küld, ezzel biztosítva a további helyes működést.

```
let def count! n | inc () = count(n + 1) | reply to inc
  or    count! n | get () = count n      | reply n to get
;;
spawn {count 0};;
```

A `count` (`<<int>>`) egy tipikus módja a kölcsönös kizárás megvalósításának. Látható ugyanis, hogy ha a kezdeti `count 0` üzenet után „kívülről” csak az `inc` (`unit -> unit`) és a `get` (`unit -> int`) csatornákra érkezik üzenet, akkor a `count` csatornán minden pillanatban legfeljebb egy üzenet várakozik: a számláló értéke. Mind az `inc`, mind a `get` csatorna őrzött folyamata leveszi ezt az értéket, és küld egy új üzenetet a `count`-ra a számláló új értékével. Az `inc` és `get` törzse emiatt sosem futhat egymással párhuzamosan.

Azonban előbbiekből már látszik a „módszer” gyengesége is: a számláló csak akkor működik helyesen, ha pontosan egy hívás érkezik a `count`-ra. Ha több hívás érkezik, akkor a kölcsönös kizárás nem biztosított, ha pedig nem érkezik kezdeti hívás, akkor a számláló egyáltalán nem működik.

A helytelen használat elkerülhető: készítünk egy `create_counter` definíciót [8], melyben a `count`, az `inc` és a `get` nevek lokálisak, és csak az `inc`-et és a `get`-et tesszük kívülről elérhetővé (a `count` tehát rejtve marad, így nem lehet kívülről meghívni):

```
let def create_counter () =
    let def count! n | inc0 ()     = count (n+1) | reply
      or    count! n | get0 ()     = count n     | reply n in
    count 0 | reply inc0, get0
;;
let inc, get = create_counter () ;;
```

A `create_counter` (`unit -> (unit -> unit) * (unit -> int)`) minden hívásakor szinkronizációs mintával létrehoz három csatornát, melyek közül kettő (a lekérdező és a módosító) lesz a `create_counter` visszatérési értéke. Azonban mivel az `inc` és a `get` törzséből a `create_counter` őrzött folyamatának kiértékelése után is érkezhet üzenet a `count` csatornára, ezért a `count` csatorna nem fog megszűnni miután kilépünk a `create_counter`-ből. A `create_counter` különböző hívásai alkalmával az őrzött folyamatában létrehozott csatornák és join-minták mind különbözőek és egymástól függetlenek lesznek, így a `create_counter`-rel több számlálót is létrehozhatunk és egymástól függetlenül használhatunk.

Ez a programozási stílus az *objektum-orientált* programozásra emlékeztet: a számláló egy „objektum", melynek van belső állapota (`count` és „tartalma"); bizonyos metódusokat exportál (`inc`, `get`), míg másokat elrejt (`count`); van konstruktora (`create_counter`), ami létrehozza az objektumot, beállítja a belső állapotát és visszaadja az exportált metódusokat.

7.2.2. Általánosítás: osztály megvalósítása csatornákkal és join-mintákkal

Legyen az **általános osztály** a következő:

```
class osztálynév param_1 … param_n =
   val adattag_1 = érték_1
   …
   val adattag_m = érték_m
   val mutable mut_adattag_1 = mut_érték_1
   …
   val mutable mut_adattag_k = mut_érték_k
   method metódus_1 param_1,1 … param_1,i1 = törzs_1
   …
   method metódus_j param_j,1 … param_j,ij = törzs_j
   method protected prot_metódus_1 prot_param_1,1 … prot_param_1,g1 = prot_törzs_1
   method protected prot_metódus_f prot_param_f,1 … prot_param_f,gf = prot_törzs_f
end
```

Az osztály **join-kalkulussal** történő megvalósítása:

```
let def create_osztálynév (param1, …, paramn) =
   let adattag1 = érték1
   …
   and adattagm = értékm in
   let def mut_adattag1! adattag_érték1 | … | mut_adattagk! adattag_értékk |
            metódus1 (param1,1, …, param1,i1) = csat_törzs1
   …
     or    mut_adattag1! adattag_érték1 | … | mut_adattagk! adattag_értékk |
            metódusj (paramj,1, …, paramj,ij) = csat_törzsj
     or    mut_adattag1! adattag_érték1 | … | mut_adattagk! adattag_értékk |
            prot_metódus1 (prot_param1,1, …, prot_param1,g1) = csat_prot_törzs1
   …
     or    mut_adattag1! adattag_érték1 | … | mut_adattagk! adattag_értékk |
            prot_metódusf (prot_paramf,1, …, prot_paramf,gf) = csat_prot_törzsf
   in
   mut_adattag1 mut_érték1 |
   …
   mut_adattagk mut_értékk |
   reply metódus1, …, metódusj
;;
```

- **Nem változtatható adattagok megvalósítása:** egyszerű azonosítóval valósítjuk meg a `create_osztálynév` szinkron csatorna törzsében. Az `in` kulcsszó hatására használhatjuk ezeket az azonosítókat a „metódusok” törzsében is.
- **Változtatható adattagok megvalósítása:** aszinkron csatorna segítségével valósítjuk meg. Ezeket a csatornákat a `create_osztálynév` nem adja vissza, ezért ezek csak a „metódusok” törzséből változtathatók; a kezdeti értéküket a `create_osztálynév` törzsében kapják meg. Mint látni fogjuk ezeken a csatornákon minden pillanatban legfeljebb egy üzenet lehet: az „adattag” aktuális értéke.
- **Metódusok megvalósítása:** szinkron csatorna segítségével valósítjuk meg. Minden metódust egy külön szinkronizációs mintában adunk meg, összekapcsolva az összes változtatható „adattag”-gal. Ezért egy tetszőleges „metódus” a törzsének kiértékelésekor elfogyasztja az „adattag” csatornákon található egyetlen értéket, így – míg minden csatornára vissza nem kerül egy érték – más „metódus” nem hívható meg. Ezzel a kölcsönös kizárás biztosított: a „metódusok” nem futhatnak egymással párhuzamosan így az „objektum” mindig konzisztens tud maradni.
  A „metódusok” törzsét át kell alakítani:
  - A változtatható „adattagokra” az eddigi `mut_adattag`$_i$ helyett `adattag_érték`$_i$ néven lehet hivatkozni.
  - A törzsben kötelező minden változtatható „adattagnak” meghatározni az új értékét (ami akár a régivel is egyezhet) a `mut_adattag`$_i$ `új_érték`$_i$ szintaxissal.
  - A visszatérési értéket expliciten a `reply visszatérési_érték` szerkezettel kell megadni.

  Látható, hogy a `create_osztálynév` csak a publikus „metódusokat” adja vissza.

Megjegyzés: A join-mintákban nem feltétlen szükséges a „metódusokhoz” az összes változtatható „adattagot” hozzákapcsolni. Azokat az „adattagok”-at kötelező csak hozzákapcsolni, amelyeket a „metódus” törzse megváltoztat. Ennél több adattag használata csak akkor szükséges, ha ki szeretnénk zárni, hogy egy másik „metódus” – ami csak az itt nem használt „adattag”-okat használja – vele párhuzamosan fusson.

Ezzel a módszerrel minden – az általános osztálynak megfelelő – osztály megvalósítható. Azonban az így kapott adatszerkezetet mégsem nevezhetjük osztálynak, mert bár az információelrejtést

megvalósítja, de nem támogatja az objektum-orientált programozás egy másik fontos jellemzőjét: az egységbezárást. A `create_osztálynév` használatával ugyanis – a `new osztálynév`-vel ellentétben – nem egy objektumot, hanem a publikus metódusoknak megfelelő csatornákból álló rendezett n-est kapunk. És bár valamilyen szinten ezek is összetartoznak, azért érezhető, hogy ez nem az igazi. Ezzel a módszerrel nem valósítható meg az öröklődés sem, ami pedig az objektum-orientált programozás egyik sarokköve.
Ezen okok miatt a csatornákkal megvalósított „osztályok” helyett inkább igazi osztályok használata javasolt.

Azonban léteznek olyan kivételes esetek, amikor a csatornákkal történő megvalósítás jobb az igazi osztályoknál:

- Az egységbezárás hiánya néha előny is lehet. Például, ha az adatszerkezetet egy elosztott rendszerben egyszerre több gépről szeretnénk használni, mégpedig úgy, hogy míg az adatot birtokló gép az összes metódust használhatja, addig a többi gép csak egy részét. Ennél a módszernél megtehetjük, hogy nem az egész adatszerkezetet, csak néhány „metódus”-csatornát regisztrálunk. (Megjegyzés: bár ezt rendes osztály esetében is elérhetjük: az engedélyezett metódusokhoz létre kell hozni egy csatornát, ami továbbítja az üzenetet az objektumhoz és csak ezeket a csatornákat osztjuk meg.)
- Van azonban egy olyan helyzet is, amit – legalábbis a jelenlegi implementációban (béta verzió) – nem lehet megvalósítani rendes objektumok esetében, de csatornákkal és join-mintákkal igen. Ez pedig nem más, mint az objektum-létrehozás képességének vándoroltatása. Ha ugyanis nem magát az objektumot regisztráljuk / küldjük üzenetben / vándoroltatjuk, hanem például egy olyan függvényt / csatornát / ilyen metódust tartalmazó osztályt, ami objektumot tud „készíteni” (például `let keszit param = new osztaly param`), akkor az új helyen a függvény / csatorna / metódus által visszaadott objektum nem biztos, hogy helyes lesz. Ha ilyesmire van szükség, akkor csak csatornákkal és join-mintákkal megvalósított osztályszerű szerkezetet használhatunk.

7.2.3. Példa: referencia

„Konvertáljunk” át egy osztályt, például a tetszőleges típusból álló (polimorf) referenciát (változtatható adatszerkezet):

```
class 'a referencia (x : 'a) =
   val mutable state = x
   method get = state
   method set new_state = state <- new_state
end
```

Csatornákkal és join-mintákkal történő megvalósítás:

```
let def create_referencia x =
   let def state! y | get ()        = state y         | reply y
     or    state! y | put new_state = state new_state | reply
   in
   state x | reply get, put
;;
```

Az adatszerkezetet a `create_referencia` *`kezdeti_érték`* hívással lehet létrehozni, ennek hatására létrejönnek a csatornák, melyek közül a `get`-et és a `put`-ot vissza is kapjuk. Egyidőben több ilyen adatszerkezetet is definiálhatunk (a lexikális láthatóság miatt minden ilyen változóhoz saját `state` tartozik, így a változók értékei nem keverednek össze):

```
# let get_i, put_i = create_referencia 0
# and get_s, put_s = create_referencia ""
# ;;
val get_i : unit -> int
val put_i : int -> unit
val get_s : unit -> string
val put_s : string -> unit
```

7.3. Szinkronizációs eszközök megvalósítása

7.3.1. Kölcsönös kizárás

Azt szeretnénk elérni, hogy a csatornáinkhoz ($\texttt{csatorna}_1$ .. $\texttt{csatona}_n$) tartozó őrzött folyamatok közül mindig legfeljebb csak egy fusson:

```
let def kolcsonos_kizaras () =
   let def free! () | csatorna_1 paraméter_1 = törzs_1
   …
     or    free! () | csatorna_n paraméter_n = törzs_n
   in free () | reply csatorna_1, …, csatorna_n
;;
```

Minden $\texttt{törzs}_\texttt{i}$-nek (`i = 1 .. n`) pontosan egy üzenetküldést kell tartalmaznia a `free` csatornára. A `free`-re történő üzenetküldéssel lehetőséget adunk a többi csatornának is a tüzelésre, ezért a törzsben az üzenetküldés utáni részben a kölcsönös kizárás már nem biztosított. A $\texttt{törzs}_\texttt{i}$-ket tehát úgy kell szervezni, hogy az egész kritikus szakasz a `free` csatornára történő üzenetküldés előtt legyen.

7.3.2. Zárolás (mutex)

Join-mintákkal emulálhatjunk a zárolást:

```
let def new_lock () =
     let def free! ()     |  lock () = not_free() | reply
       and   not_free! () |unlock () = free ()    | reply in
     free () | reply lock, unlock
;;
```

Ha egy szál be akar lépni a kritikus szakaszába, akkor zárolnia kell (azért, hogy amíg el nem hagyja a kritikus szakaszt, addig más ne tudjon belépni). Zároláshoz a `lock` szinkron csatornát kell használnia. Ekkor – a `lock` definíciója szerint – a rendszer elfogyasztja a `free`-n várakozó üzenetet is. Ha ekkor egy másik szál is megpróbál zárolni, akkor blokkolódik (mert a `free`-n nem várakozik üzenet), egészen addig, amíg a zároló szál fel nem oldja a zárolást az `unlock` szinkron csatorna segítségével. Az `unlock` hatására egy `free`-hívás történik, így újra lehet zárolni. A `not_free` csatorna segítségével lehet elérni, hogy a zárolást csak akkor lehessen feloldani, ha tényleg zárolva van.

7.3.3. Sorompó (barrier)

A sorompó egy másik gyakori szinkronizációs mechanizmus. A sorompó szinkronizációs pontokat definiál a párhuzamos végrehajtásban (itt tudják egymást bevárni a szálak, majd innen párhuzamosan folyik tovább a végrehajtás). A következő példa [8] egy egyszerű sorompó, ami két szálat szinkronizál:

```
let def join1 () | join2 () = reply to join1 | reply to join2 ;;
```

A barrier használatára példa a következő program, ami vagy `(ab)`-t, vagy `(ba)`-t ír ki (a zárójelek a szinkronizáció miatt biztosan kívül lesznek):

```
spawn {
    {print_string "("; join1 (); print_string "a"; join1(); print_string ")" ;}
    |
    {join2 (); print_string "b"; join2 () ;}
}
;;
```

7.4. Ciklusok megvalósítása

7.4.1. Egyszerű ciklusok

Aszinkron ciklus – ha a feldolgozás sorrendje nem számít és az iterációk párhuzamosan is számíthatók (az `a` egy `0`-aritású aszinkron csatorna, ezt kell `x`-szer meghívni) [8]:

```
# let def loop! (a, x) = if x > 0 then { a () | loop (a, x-1) };;
#
# let def echo_star! () = print_string "*" ; ;;
#
# spawn { loop (echo_star, 5) } ;;
val loop : <<(<<unit>> * int)>>
val echo_star : <<unit>>
-> * * * * *
```

Szekvenciális ciklus – ha a sorrend számít és az iterációk számítása nem párhuzamosítható (az `a` egy `1`-aritású szinkron csatorna, ezt kell `x`-szer meghívni, `a`-nak mindig átadjuk a „ciklusváltozó”-t is) [8]:

```
# let def loop (a, x) =
#     if x > 0 then {a x ; loop (a, x-1) ; reply} else reply ;;
#
# let def print x = print_int x ; reply ;;
#
# loop (print, 5) ;;
val loop : (int -> 'a) * int -> unit
val print : int -> unit
-> 5 4 3 2 1
```

Ha a ciklus során **számolunk** valamit (például a következő program [8] `i`-szer végrehajtja az `f` függvényt és az eredményeket összeadja):

```
# let def sum (i, f) =
#     if i > 0 then reply (f i) + sum (i-1, f) else reply 0;;
#
# let def square x = reply x*x;;
#
# print_int (sum (32,square));;
val sum : int * (int -> int) -> int
val square : int -> int
-> 11440
```

Megjegyzés: az `(f i)` számítások nem párhuzamosan hajtódnak végre, hanem az iteráció miatt szépen egymás után.

Azonban – mivel az összeadás asszociatív és kommutatív művelet – az `(f i)` számításokat párhuzamosan kéne végezni (legalábbis aszinkron iterációval több lehetőséget adni a párhuzamos végrehajtásra), és a sorrend sem számít [8]:

```
# let def sum (i0, f) =
#   let def add! dr | total (r, i) =
#      let r' = r+dr     in
#      if i > 1 then reply total (r',i-1)
#      else reply r'     in
#   let def loop! i = if i > 0 then {add (f i) | loop (i-1)} in
#   loop i0 | reply total (0,i0)
# ;;
#
# print_int (sum (32,square));;
val sum : int * (int -> int) -> int
-> 11440
```

A `loop` aszinkron csatorna végzi az `(f i)` számításokat párhuzamosan, az eredményeket az `add` aszinkron csatornának adja át.
Megfigyelhető, hogy a ciklus visszatérési értéke akkumulálódik a szinkron `total`-t használva (a `total` első paramétere az eddigi részösszeg, a második paramétere a hátralévő iterációk száma). A `sum` visszatérési értéke egy kezdeti `total` hívás (eddigi részösszeg: `0`, hátralévő iterációk száma: `i0`). A `sum` akkor tér vissza, amikor az aszinkron `loop` befejeződik, és a `total` minden részösszeget összeadott.

7.4.2. Elosztott ciklusok

Egy ciklus párhuzamosítását úgy is el lehet érni, ha *ágens*ek között megosztjuk a ciklusbeli feladatokat. Mivel most a hangsúly a ciklus megvalósításán van az ágenst szinkron csatornával reprezentáljuk.

Példa az elosztott ciklusra [8]: két ágensünk van: `square1`, `square2`. A `square1` ágens modellezi a gyors gépet, míg a `square2` a lassút, úgy, hogy nem hatékony módon számítja az eredményt (a négyzetet összeadással számolja ki). Hogy nyomon követhessük a működést, a `square1` kiír egy „+”-t a válasz előtt, a `square2` pedig egy „*”-ot kezdéskor és egy „–”-t a válasz előtt.

```
#let def square1 i = print_string "+" ; reply i*i;;
#
# let def square2 i =
#   print_string "*" ;
#   let def total! r | wait () = reply r in
#   let def mult! (r, j) =
#      if j <= 0 then total r else mult (r+i, j-1) in
#   mult (0, i) |
#   let r =  wait () in
#   print_string "-" ; reply r
# ;;
val square1 : int -> int
val square2 : int -> int
```

Ahhoz, hogy a ciklust el tudjuk osztani az ágensek között, valamilyen módon ki kell osztani közöttük az iterációkat. A `make_sum` két csatornát ad vissza: a `register`-t és a `wait`-et. Az ágensek regisztrálják magukat úgy, hogy elküldik a számolást végző csatornájukat a `register`-re, a `make_sum` pedig kiosztja a feladatokat a regisztrált ágensek között. A `wait` adja vissza a számolás eredményét.

```
# let def make_sum i0 =
#   let def add! dr | total i =
#      if i > 1 then reply dr+total(i-1)
#      else reply dr
#   and wait () = reply total i0 in
#   let def loop! i | register! f =
#      if i > 0 then {add (f i) | register f | loop (i-1) } in
#   loop i0 | reply register, wait
# ;;
val make_sum : int -> <<(int -> int)>> * (unit -> int)
```

Ebben a ciklusban egy fontos különbség az előző megoldáshoz képest: a „`let def loop! i = …`" helyett „`let def loop! i | register! f = …`" van.

A `square1` és `square2` ágensek most már teljessé tehetik az iterációt két hívással: `register square1` és `register square2`.

```
# let register, wait = make_sum 32;;
#
# spawn {register square1 | register square2};;
#
# print_int (wait ());;
val register : <<(int -> int)>>
val wait : unit -> int
-> *+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+---------------11440
```

A fenti elosztott ciklus azonban nem kielégítő, mert egyenletesen osztja el az iterációkat, és nem veszi figyelembe a `square1` és a `square2` futásideje közti különbséget.
Az `add (f i)` aszinkron hívás, ezért az ágensek az iterációkat párhuzamosan hajtják végre. Az `add (f i)` hívással egyidejűleg egy újabb `register` hívás is történik, így az ágens újra kaphat feladatot. Ennek következtében az iterációk egyenletesen oszlanak el `square1` és `square2` között, ami a kiírt szövegen is jól látszik. Ez viszont rossz egyensúlyhoz vezet, mert így a gyors `sqaure1` a ciklus végén tétlenül vár, amíg a lassú `square2` be nem fejezi a számolást.

Jobb megoldás, ha egy ágensen belül szekvenciális végrehajtás van [8]. A szekvenciális végrehajtás a `loop` és a `register` definícióján történő kis módosítással elérhető:

```
# let def make_sum i0 =
#   let def add! dr | total i =
#      if i > 1 then reply dr+total (i-1)
#      else reply dr in
#   let def wait () = reply total i0 in
#   let def loop! i | register! f =
#      if i > 0 then
#         { loop(i-1) |
#           let r = f i in
#              add r | register f } in
#   loop i0 | reply register, wait
# ;;
val make_sum : int -> <<(int -> int)>> * (unit -> int)
```

Az új definícióban `register f` csak `(f i)` kiszámítása után hívódik meg újra. Ellenben `loop (i-1)` azonnal meghívódik, így egy másik ágens amint kész van, azonnal hozzáfoghat az iteráció számításához. A kiírt jeleken jól látszik, hogy a gyorsabb ágens sokkal több munkát végez, mint a lassúbb [8]:

```
# let register, wait = make_sum 32;;
#
# spawn {register square1 | register square2};;
#
# print_int (wait ());;
val register : <<(int -> int)>>
val wait : unit -> int
-> * + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + - 11440
```

7.5. Applet

A következő példa [8] megmutatja, hogy hogyan készíthetünk *„applet”*-eket. Az applet egy olyan program, ami letöltődik a távoli szerverről, majd lokálisan lehet használni. Az ágensnek paraméterként adjuk meg, hogy híváskor hova vándoroljon.

A következő applet definiál egy *reference cell*-t, ami egy értéket tartalmazhat és destruktív olvasással rendelkezik (az olvasási művelettel párhuzamosan a buffer kiürül):

```
# let def cell there =
#   let def log s =
#      print_string ("cell "^s^"\n"); flush stdout; reply in
#
#   let loc applet
#      def get () | some! x  = log ("is empty"); none () | reply x
#      and put x  | none! () = log ("contains "^x); some x | reply
#      do { go there; none () }    in
#
#   reply get, put
# ;;
#
# Ns.register "cell" cell vartype;
# Join.server ();;
```

Az applet-nek két állapota – `none ()` és `some s` (`s` egy `string`) – és két metódusa – `get` és `put` – van. Minden `cell`-híváskor egy új `applet` készül, ami egyből a paraméterében megadott absztrakt helyre (`there`) vándorol. Az áthelyezést a `go there; none ()` folyamat irányítja, ami először elvégzi a vándorlást, majd küld egy belső üzenetet az aktív `cell`-nek. Ezenkívül, a `cell` az `applet`-en kívül definiál egy `log` csatornát is, ami – mivel az `applet`-en kívül van – a szerveren marad. A kliens gépen lévő `applet`-en belülről rendszeresen hívás érkezik rá, így nyomon lehet követni a `cell` használatát a szerver gépről. Ez ellentétes a *Java applet*-tel, mert a hely ugyan JoCaml-ben is elvándorol a kódjával együtt, de itt a kommunikációs képességek változatlanok maradnak.
Több, egymástól független `cell`-t is készíthetünk.

Kiegészítjük a példánkat egy egyszerű klienssel, ami lefoglal és használ egy lokális `cell`-t:

```
# let cell = Ns.lookup "cell" vartype
#
# let loc user
#   do {
#      let get, (put : string -> unit) = cell user in
#      put "world";
#      put ("hello, "^get ());
#      print_string (get ());
#      exit 0;
#   }
-> hello, world
```

A szerveren megkapjuk a nyomkövetés eredményét:

```
-> cell contains world
-> cell is empty
-> cell contains hello, world
-> cell is empty
```

A kliens oldalon eredményül kapott `applet` tartalma tekinthető a hoszt-hely egy részének, mintha a kezdetben is lokálisan definiálták volna.

8. Elosztott programozás jellemzői JoCaml-ben

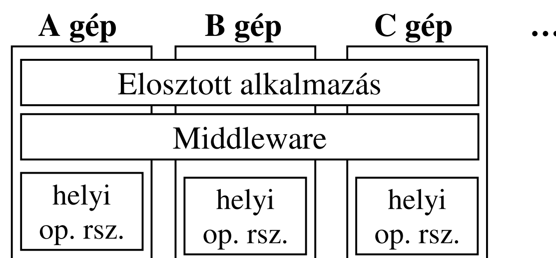
8.1. Átlátszósági tulajdonságok JoCaml-ben

Az elosztott rendszerek egyik fontos célja, hogy elrejtse azt a tényt, hogy a folyamatok és az erőforrások fizikailag több különálló gépen helyezkednek el. Az olyan elosztott rendszert, ami úgy képes megjelenni a felhasználó számára, mintha egyetlen számítógép lenne, **átlátszónak** nevezzük [9].

Egy elosztott rendszernek viszonylag könnyen bővíthetőnek, skálázhatónak kell lennie: a számítás közben az elosztott rendszerhez újabb számítógépek csatlakoztathatók, míg más gépek átmenetileg működésképtelenné válhatnak. Ezeket a változásokat is amennyire lehet célszerű elrejteni a felhasználók elől [9].

Elosztott alkalmazások írásához tehát az ágenseket támogató nyelvi elemek (primitívek) önmagukban általában nem elégségesek: szükséges bizonyos típusú átlátszóságok és valamilyen szintű skálázhatóság megvalósítása. Ezen funkciókat általában a *middleware* szolgáltatja [10] (ő támogatja a heterogén gépeket, míg segít egyetlen gép látszatát kelteni).

Felépítés [9]**:**



Ebben a részben bemutatom, hogy milyen átlátszósági és skálázhatósági fajták léteznek egy elosztott rendszer esetében, és megvizsgálom, hogy ezek milyen szinten teljesülnek JoCaml-ben.

8.1.1. Hozzáférési átlátszóság

A hozzáférési átlátszóság az adatábrázolás különbségeinek és az erőforrások hozzáférésének a felhasználó elől való elrejtésével foglalkozik [9].

Ez az átlátszóság például a következő problémával foglalkozik: ahhoz, hogy egy integer értéket átküldjünk egy Intel alapú munkaállomásról egy Sun SPARC gépre, figyelembe kell vennünk, hogy az Intel processzorok a bájtoknál a little-endian (a magasabb értékű bit kerül először átadásra), míg a Sun processzorok a big-endian formátumot használják. Az adatábrázolás természetesen egyéb módokon is eltérhet, például az elosztott rendszer tartalmazhat különböző operációs rendszereket futtató gépeket, melyek mindegyike saját fájl elnevezési konvenciót alkalmazhat. A felhasználók és az alkalmazások elől ezeket ugyanúgy el kell rejteni, mint a fájlmanipuláció esetleges eltéréseit. Tehát a hozzáférési átlátszóság biztosítja, hogy a heterogén gépeket egységesen lehessen kezelni [9].

A JoCaml több UNIX platformot és a Windows NT-t is támogatja. (Pontosabban, támogatja például a következő platformokat: Alpha processzorok [Linux alatt], Sparc processzorok [Sun OS 4.1-től, Solaris 2-től vagy BSD-n], Intel processzorok (PC) [Linux, Nextstep, BSD, Solaris 2, Windows NT], PowerPC [Linux].) A Windows NT-t csak a Cygwin segítségével támogatja, ami lényegében egy Linux-emulátor.

Összességében tehát elmondhatjuk, hogy a JoCaml több UNIX platformon is működik. Azonban a fent felsorolt operációs rendszereket sem támogatja minden esetben, ugyanis installáláskor erős megkötéseket tesz a rendszerre, például: csak 32 vagy 64 bites architektúrán fut, az integer-t csak 4 byte-on lehet ábrázolni, a double-nak double-word-határra kell igazodnia. Ezeken kívül a C fordítóra is megkötéseket tesz (például legyen ANSI kompatibilis, a PC-s Linux-on nem használható a 2.7.2.1-es verziószámú GCC), és több könyvtári függvény létezését is megköveteli (például.: sys modul / strerror()), továbbá az operációs rendszernek tudnia kell a szálakat kezelni (például legyen Threads könyvtár). A byte-ok lehetnek big vagy little endian szerintiek is.

A JoCaml fordítóprogramja alapján a fenti megszorításokkal a **hozzáférési átlátszóság teljesül**.

A JoCaml installálása előtt lefuttatandó `configure` program térképezi fel a rendszer tulajdonságait – úgy, mint platform, elemi típusok tárolása, big/little endian, C fordító, könyvtári függvények létezése stb. –, és dönti el, hogy ezek alapján egyáltalán fel lehet-e installálni a JoCaml-t. Az átlátszóság biztosítása érdekében a JoCaml fordító egy program fordításakor mindig a rendszer feltérképezett és feljegyzett tulajdonságai szerint jár el.

A hozzáférési átlátszóság egy része program-könyvtárak segítségével valósul meg: a könyvtár általános függvényeket nyújt, melyekkel tetszőleges platform felett egységesen kezelhetjük a rendszer-specifikus elemeket. Ezek a könyvtári függvények mindig az aktuális gép tulajdonságaitól függően végzik el a feladatukat. Ezen könyvtári függvények használatával biztosak lehetünk abban, hogy a fordító tetszőleges (támogatott) platformon le tudja fordítani a programot, és az eredmény helyesen – az adott rendszernek megfelelően – fog működni.

Például az egyes operációs rendszerekhez tartozó fájl-elnevezési konvenciókat a `Filename` modul valósítja meg, amely a fájlok eléréséhez és azonosításához biztosít általános függvényeket: például `Filename.current_dir_name` (az aktuális könyvtár jelölése, mely Unix-nál és WinNT-nél „.”, Mac-nál „:”), `Filename.concat` (könyvtár és fájlnév összefűzése, amelynél Unix esetén a „/”, WinNt-nél a „\\”, Mac-nél a „:” elválasztót használja). Ezen függvények használatával a fájlrendszert anélkül kezelhetjük, hogy tudnánk, hogy a gépen milyen operációs rendszer van.

A `Sys` modulban több olyan azonosító is található, mellyel a gép aktuális konfigurációja futás közben is lekérdezhető. Például: `Sys.os_type` (`string`) mindig az aktuális operációs rendszer nevét tartalmazza (ami `"Unix"`, `"Win32"` vagy `"MacOS"` lehet); `Sys.word_size` (`int`) - visszaadja, hogy a rendszer hány bit-en tárolja a `word`-öt (`32` vagy `64`) stb. Ezek az azonosítók segítik a programozókat, hogy rendszer-specifikus függvényeket készíthessenek figyelembe véve az adott rendszer konfigurációját.

8.1.2. Elhelyezkedési átlátszóság

Az elhelyezkedési átlátszóság arra a tényre utal, hogy a felhasználók vagy ágensek nem ismerik az egyes erőforrások tényleges fizikai elhelyezkedését a rendszeren belül [9].

A névadásnak fontos szerepe van az elhelyezkedési átlátszóság elérésében. Valójában az elhelyezkedési átlátszóság úgy érhető el, hogy az erőforrásokhoz csak logikai neveket rendelünk, vagyis olyan neveket használunk, amibe az erőforrás elhelyezkedését nem kódoljuk bele [9].

JoCaml-ben ezt a fajta átlátszóságot úgynevezett név-szerver bevezetésével érhetjük el. [10] Ha egy program szeretné, hogy az erőforrásait (ami lehet függvény, csatorna, absztrakt hely, objektum stb.) más programok is használhassák, akkor azt az erőforrást meg kell osztania: regisztrálnia kell név-szerveren egy logikai névvel. Ezután bárki, aki ehhez a név-szerverhez csatlakozik – a logikai név használatával – lekérdezheti (használatba veheti). Mivel a programok a név-szervertől kérik le az erőforrásokat, így csak a névszerver fizikai helyét kell tudniuk, a többi gépét nem (nem kell tudniuk, és nem is tudhatják).

A név-szervertől lekért erőforrásokat teljesen ugyanúgy lehet használni, mint a helyileg deklaráltakat, a felhasználó program tulajdonképpen nem is tudja, hogy egy helyi, vagy egy távoli erőforrással van dolga. Tehát a definíciók hatóköre és értéke nem függ az elhelyezkedéstől, így amikor például egy port-név látszik egy folyamatban (vagy azért, mert előtte deklarálták és benne van a hatókörében, vagy azért, mert a névszervertől kérte le, vagy azért, mert üzenetben kapta meg), akkor használható üzenetküldésre (akár címként, akár üzenettartalomként használjuk a nevet), anélkül, hogy tudnánk, hogy ez a port-név fizikailag hol van, és hogy egyáltalán lokálisan van-e definiálva, vagy pedig egy távoli gépen.

Kimondhatjuk tehát, JoCaml-ben az **elhelyezkedési átlátszóság fennáll**, és a láthatóság nem függ az elhelyezkedéstől.

Természetesen, az elhelyezkedés bizonyos körülmények között számít. Például, ha valamit ki akarunk íratni a terminálra, vagy billentyűzetről akarunk adatot bekérni, akkor ez függ az aktuális géptől: a szöveg mindig azon a terminálon fog megjelenni, és az adatot is mindig azon a gépen kell megadni, ahol a kiírást, illetve a bekérést kezdeményező kifejezés (például `print_string`, `input_line`) ténylegesen van. (Megjegyzés: ezt a tulajdonságot kihasználva kiírató kifejezések segítségével nyomon követhetjük a kifejezések / folyamatok tényleges fizikai elhelyezkedését). Ezeken kívül, az elhelyezkedés a teljesítményre is hatással lehet, mert egy hálózaton keresztüli üzenetküldés tovább tart, mint egy lokális hívás. Végül az elhelyezkedés a terminálásra is hatással lehet, mert például egy absztrakt helyen belül kiadott `halt` (vagy maga a tartalmazó program terminálása) hatással lesz az összes lokális folyamatára (az itt tartózkodó al-helyek és folyamataik is leállnak) [8].

Az elosztott rendszerben lévő programoknak tehát ismerniük kell a névszerver fizikai címét. A cím alapértelmezett értékét két környezeti változó tárolja: `JNSNAME` az IP címet, `JNSPORT` a port-számot (alapértelmezésben `localhost.localdomain:20001`, de ez természetesen változtatható). A számítás megkezdése előtt az egyik gépen el kell indítani a név-szervert, a többi gépen pedig be kell állítani a címét. A név-szerver címe a program futása során is változhat, ugyanis a címet az `Ns.name` és az `Ns.port` azonosítók tárolják (mindkettő `string ref` típusú). Tehát lehetőségünk van **több név-szerver elindításá**ra is – természetesen mindegyiket más IP címen, vagy porton –, majd a program futása közben az `Ns.name` és az `Ns.port` azonosítók értékének változtatásával (`:=` operátor) váltogathatunk a név-szerverek között.
Azonban mindig tudnunk kell, hogy melyik azonosítót melyik név-szerverben regisztráltuk, ugyanis a különböző név-szerverek között semmilyen kapcsolat nincs.

Jogosultságok kezelése: a JoCaml névszerverében nincs jogosultság-kezelés. Így tehát a regisztrált azonosítók láthatóságát nem lehet csak egy bizonyos csoportra korlátozni, a regisztrált nevekhez bárki hozzáférhet.

Hiányosságok: a JoCaml-nek jelenleg hiányolt jellemzője az objektum-létrehozás képességének vándoroltatása.
Tegyük fel, hogy nem magát az objektumot, hanem egy olyan függvényt regisztrálunk a név-szerverben, ami egy objektumot ad vissza. Ha egy másik program a név-szervertől lekéri ezt a függvényt (vagy más módon kapja meg, például üzenetben), akkor a függvény használatakor a visszaadott objektum nem biztos, hogy működni fog [8]. Ez a jelenlegi implementáció egyik hiányossága.

8.1.3. Vándorlási átlátszóság

Az olyan elosztott rendszerekre, melyekben az erőforrásokat anélkül mozgathatjuk, hogy az befolyásolná az erőforrások elérését, azt mondjuk, hogy biztosítja a vándorlási átlátszóság elvét [9].

A vándorlási átlátszóság azt eredményezi, hogy amikor egy erőforrás, vagy egy ágens vándorol, erről a felhasználók nem szereznek tudomást. Ezután is ugyanúgy lesznek képesek használni ezeket, mint a vándorlás előtt. Ez úgy érhető el, hogy a név-szervernek nyomon kell követnie a regisztrált erőforrásokat. Normális esetben egy ágens / erőforrás jelenteni tudja a sikeres vándorlást és az új helyét az eredeti helyének. Tanácsos jelenteni a vándorlás előtt is, mert ha a vándorlás nem sikerül, akkor a név-szerver újraküldi ágenst. Ebben az esetben az ágens egy másolatát meg kell tartanunk az eredeti helyen is a vándorlás befejezéséig, bár javasolt ezen másolat futtatásának felfüggesztése a vándorlás ideje alatt [10].

Az ágensek vándoroltatásánál az első kérdés az, hogy egészen pontosan mi vándorol el [10]. Ez a kérdés nem olyan magától érthetődő, mint amilyennek látszik. Például, ha egy ágens egy külső azonosítóra történő hívásokat tartalmaz, akkor ennek a külső azonosítónak is el kell vándorolni az ágenssel együtt? Ha nem vándorol el vele, akkor az ágens képes lesz az új helyén futni? Már ezekből a kérdésekből is látszik, hogy egy nyelvnél ennek a problémának a kezelésére alapvetően két lehetőség van: (1) Az ágens vándoroltatásakor a rendszernek minden olyan erőforrást, amire az ágens hivatkozik szintén át kell küldenie (természetesen ez esetben amire a hivatkozott erőforrás hivatkozik, azt is át kell küldeni, és így tovább, tehát ezen függvényekre ki kell számítaniuk a hivatkozási láncok tranzitív lezártját [10]). Így az ágens fogadása után a hivatkozott erőforrásokat mint helyit használhatjuk. (2) A rendszer csak magát az ágenst küldi át. Ebben az esetben az ágens törzséből érkező – eddig helyi – hívások a vándorlás után távoliak lesznek. A vándorláskor a `go` utasításnak (ez végzi az ágens vándoroltatását) fel kell jegyeznie, hogy az ágens honnan vándorolt el, hogy tudja, hogy a vándorlás után a törzsből érkező hívásokat hol kell keresni. A vándorlási átlátszóság biztosítása érdekében azonban ezeket a visszahívásokat használati szempontból el kell rejteni a felhasználó elől, tehát a vándorlás után távolivá vált erőforrásokra szintaktikailag ugyanúgy lehessen hivatkozni, mint a vándorlás előtt, amikor az erőforrás még helyi volt (a vándorlás után se kelljen ezen erőforrásokhoz távoli primitíveket használni). A rendszernek ezeket a hívásokat automatikusan (és csendesen) le kell tudnia kezelni, ebből a felhasználónak semmit sem szabad észrevennie.

Mindkét módszernek vannak előnyei és hátrányai: Az eredeti helyre történő visszahívások hátránya lehet például az, hogy amennyiben egy hely sokszor intéz visszahívásokat, a futtatási idő jelentősen megnövekedhet, különösen, ha a hálózat lassú. A másik hátrány abból fakad, hogy az ágens eredeti helyének mindig üzemelnie kell. Ha a hálózati kapcsolat megszakad, az is ágens leáll [9].
Ezek a hátrányok az (1)-es módszer használatával kiküszöbölhetők, azonban ennek a módszernek is vannak hátrányai: a hivatkozási láncok tranzitív lezártjának kiszámítása költséges feladat, továbbá ez esetben lehet, hogy az eredeti ágensnél sokkal nagyobb „csomagot” kell átküldeni, ami lassíthatja a munkát. Ez esetben az új helyen a hivatkozott erőforrásoknak csak másolatai lesznek, melyek nincsenek kapcsolatban az eredeti példányukkal. Ez további problémákat okozhat, ha az erőforrást nem csak az ágens, hanem egy másik – az eredeti helyen maradó – kódrész is használta. Képzeljük el például a következő helyzetet: az erőforrás egy több join-mintával összekapcsolt csatorna (például legyen a következő: `let def a () | b () = f;;`), a vándorló ágens csak az `a` csatornára, míg az ottmaradó csak a `b` csatornára intéz hívásokat. Az `f` folyamat csak akkor fog lefutni, amikor mindkét csatornájára érkezik hívás. Mivel a vándorló ágens hivatkozik az `a` csatornára, így azt is át kell küldeni. Ha a `b` csatorna az eredeti helyen marad, akkor megszűnik közöttük a kapcsolat (ennél a módszernél nincsenek visszahívások), így az `f` többé már nem fog lefutni.

JoCaml-ben a (2)-es módszert választották, tehát vándorláskor **csak maga az ágens vándorol**. Az absztrakt hely képes az ő eredeti helyére visszahívást intézni, és ezért képes minden, a definiálásakor még helyi erőforrást is használni.

Vegyük a már ismert *applet* példát [10]. A példában az ágens visszahív az ő származási helyére:

```
let def new_cell there =
    let def log s = print_string ("cell "^s^"\n"); reply in
    let loc applet
        def get () | some! x  = log ("is empty");   none();  reply x
        and put x  | none! () = log ("contains"^x); some x | reply
    do {go there; none ()} in
    reply get,put
;;
```

A példa definiál egy `new_cell` nevű buffer-t. Az absztrakt helyen belül definiáltuk a buffer műveleteit (`get`, `put`). Ez az absztrakt hely átvándorol egy másik hely alá (`there`), amit a `new_cell` paraméterében adunk meg. A `get` és a `put` definíciója jelenleg irreleváns, az egyetlen dolog, amit figyelembe kell venni az az, hogy mindkét definíció tartalmaz hívásokat a `log` csatornára, ami információkat ír ki a képernyőre. A `log` az ágensen kívül található, így nem vándorol el. A `log` csatornát azonban az ágens a vándorlás után is el tudja érni, így nyomon tudjuk követni a *cell* használatát az eredeti helyéről.

Összefoglalva elmondhatjuk, hogy az absztrakt helyek kommunikációs képessége vándorlásuk után is változatlan marad. Egy ágens kódja ebből következően csak az eredeti helyén definiált kódból áll. Amikor egy absztrakt helyet vándoroltatunk, nincs szükség egyéb kód csatolásához, a futtató környezet gondoskodni fog minden létező kommunikációs kapcsolatról. Ez az jelenti, hogy minden, amit az ágens az eredeti helyén képes volt használni, az új helyén is működőképes marad. Miután egy absztrakt hely egy másik alá vándorol át, úgy viselkedik, mint egy helyi függvény, tehát a futtatásához nincs szükség távoli primitívekre [10].

Hivatkozási probléma nem csak „erről az oldalról” keletkezhet: egy absztrakt hely tartalmazhat csatornadefiníciókat, melyek hatóköre nem csak magára a helyre, hanem a hely utáni kód-részre is kiterjed. Ha a vándorlási átlátszóságot fenn akarjuk tartani, akkor ezen csatornák láthatósága nem szűnhet meg abban az esetben sem, ha az absztrakt hely elvándorol. A JoCaml ezért vándorláskor nemcsak azt „jegyzi fel”, hogy mi volt az eredeti hely (hogy az ágens vissza tudjon oda hívni), hanem azt is, hogy mi lett az új hely, hogy az ágens csatornáira kívülről érkező hívásokat továbbítani tudja az új helyre. Ebből azonban kifelé semmi sem látszik, az ágens csatornái a vándorlás után is ugyanúgy használhatók az eredeti helyről.

Önálló futtatás: az ágens akkor tud az új helyén önállóan futni, ha nem tartalmaz visszahívásokat az eredeti helyére. JoCaml-ben az ágensek nem rendelkeznek az implicit visszahívás képességével, tehát amikor a visszahívás szükséges, akkor a programozónak azt explicit módon bele kell írni a program kódjába. A JoCaml-ben a programozó úgy definiálja explicit módon a visszahívásokat [10], hogy vagy függvényeket definiál az absztrakt helyeken kívül, de ugyan azon a hatókörön belül, vagy lekérdezést (`lookup`) használ.

Mindezen vizsgálatok után elmondhatjuk, hogy a JoCaml teljesen **megvalósítja a vándorlási átlátszóságot**. A vándorlás semmilyen szinten nem befolyásolja a láthatóságot, tehát az, hogy egy ágens ténylegesen hol van, meg az, hogy miket lát, az két teljesen különböző dolog.

Az elhelyezkedési átlátszóság biztosítása miatt vándorláskor célcímként sosem egy konkrét IP-címet adunk meg, hanem egy másik absztrakt helyet, és a vándorlás célpontja ezen absztrakt helynek az adott pillanatbeli tényleges fizikai címe lesz.

Problémák / hiányosságok:

- Képzeljük el a következő helyzetet: az absztrakt hely folyamata legyen két folyamat párhuzamos kompozíciója (`P | Q`). Mi történik abban az esetben, ha például csak a `P`

tartalmaz `go` utasítást, vagy ha ugyan mindketten tartalmaznak `go`-t, de mindketten máshova. Elméletileg az absztrakt hely egy egészként vándorol a folyamatával és a definiált csatornáival együtt. Azonban a párhuzamos kompozíció operátor két külön szálon indítja el a paramétereinek kiértékelését, és a rendszer nem tartja nyilván, hogy melyik szál kiértékelése hol tart, esetleg indítottak-e el újabb párhuzamos kiértékeléseket. Tehát ennek a problémának a kezelése nem olyan magától érthetődő. Egy elképzelhető megoldás lenne, hogy a két párhuzamosan futó folyamat közül az egyik elvándorol, a másik a helyén marad (vagy mind a ketten elvándorolnak két különböző helyre). A nagyobb kérdés azonban az, hogy hol lesznek a csatornák (mennek / maradnak, ha több helyre kéne menniük, akkor melyiket választják, ha mindkettőt választják, akkor az egyes másolatok között milyen kapcsolat lesz, stb)?
Ez a probléma tehát több kérdést is felvet, azonban a tesztek sajnos azt mutatták ki, hogy ebben az esetben a mindkét folyamat leáll (az ottmaradt folyamat leállt, a vándorló pedig nem érkezett meg). A JoCaml jelenlegi verziójában nem működik a vándorlás, ha a folyamat több párhuzamos szálból áll.
- Amint arról már szó volt, a JoCaml nem tudja az objektum-létrehozás képességét vándoroltatni. Így ha egy olyan hely, vagy objektum vándorol el valahová, melynek egyik csatornája vagy metódusa objektumokat készít, akkor az az objektum, amit visszatérési értékeként kapunk esetleg nem fog működni.

8.1.4. Áthelyezési átlátszóság

A vándorlási átlátszóságnál erősebb megkötés az, ha az éppen használt erőforrások helyezhetők át úgy, hogy az erőforrásokat felhasználó gépek ezt a műveletet nem veszik észre, azokra semmilyen hatást nem gyakorol. Az ilyen tulajdonsággal rendelkező rendszerekről mondjuk, hogy megfelelnek az áthelyezési átlátszóság elvének [9].

Erre az átlátszóságra egy példa a következő: a folyamatosan helyet változtató felhasználók zavartalanul használhatják a vezeték nélküli kapcsolódással ellátott laptopjaikat, anélkül, hogy bármikor is – akár csak átmenetileg – le kellene csatlakozniuk a hálózatról [9].

A JoCaml teljesíti ezt a fajta átlátszóságot is.

8.1.5. Replikációs átlátszóság

Az erőforrásokról másolatokat hozhatunk létre az elérhetőség vagy a teljesítmény növelése érdekében (a hozzáférési hely közelében elhelyezünk egy újabb másolatot az erőforrásról). A replikációs átlátszóság az erőforrások esetleges többszörös példányainak létezését rejti el a felhasználók elől [9].

Ennek az átlátszóságnak a biztosításához szükséges, hogy az erőforrás-másolatoknak ugyanaz legyen a neve. Ebből következően egy a replikációs átlátszóságot biztosító rendszernek az elhelyezési átlátszóságot is biztosítania kell, mert máskülönben lehetetlen lenne a különböző helyeken lévő replikák elérése [9].

A JoCaml **nem teljesíti a replikációs átlátszóságot**, semmilyen nyelvi elem nem létezik a másolatok készítésére.

Másolatok létrehozására egy nagyon korlátozott lehetőség van: az erőforrásokat (csak csatornákat és az ezeket összekapcsoló join-mintákat) elhelyezhetjük egy absztrakt helyben. Ezt a helyet pedig egy szinkron csatorna törzsében hozzuk létre, ahol maga az absztrakt hely lesz a csatorna visszatérési értéke. A csatorna paraméterként kaphat egy helyet, ahová létrehozás után az absztrakt hely elvándorol:

```
let def make there =
   let loc location =
      def csatorna_deklarációk
      do {
         go there;
         folyamat
      }
   in location
;;
```

Ezt a `make` csatornát regisztrálhatjuk a név-szerverben. Ezután ha lekérik a csatornát és üzenetet küldenek rá, akkor a csatorna létrehozza az erőforrást, ami a paraméterben megadott helyre vándorol.
Így elérhetjük, hogy egy erőforrást többszörösen is létrehozzuk, ráadásul az is megmondhatjuk, hogy az erőforrás hol legyen, így mindig az adott hely közelébe is rakhatjuk.

Ezzel a módszerrel másolhatjuk az erőforrásokat, azonban a másolatok teljesen különállóak lesznek, közöttük semmiféle kapcsolat nincs, akár különböző névre is hallgathatnak. A módszer hiánya még, hogy a másolatot csak létrehozáskor készíthetünk, tehát nincs lehetőség egy már működő ágensről futás közben másolatot készíteni. Így ez még nagyon messze van a replikációs átlátszóság megvalósításától.

8.1.6. Konkurencia átlátszóság

A konkurencia átlátszóság az erőforrások több versengő felhasználó közötti szétosztási lehetőségének elrejtését jelenti [9].

Az elosztott rendszerek egyik fontos célja az erőforrások megosztásának lehetősége. Ez a megosztás lehet versengő is, amikor többen egyszerre szeretnék megszerezni ugyanazt az erőforrást. Például: két, egymástól független felhasználó el szeretné tárolni a fájljait ugyanazon a fájl-szerveren, vagy mindketten ugyanazt a táblát használják egy elosztott adatbázisban. Ilyen esetekben fontos, hogy egyik felhasználó se vegye észre, hogy a másik ugyanazt az erőforrást használja. Ezt a jelenséget hívjuk konkurencia átlátszóságnak [9].

Fontos kérdés ezzel kapcsolatban, hogy a konkurensen elért erőforrás konzisztens állapotban maradjon. A konzisztens állapotot lock-oló mechanizmusokkal érhetjük el, amivel egyúttal biztosítjuk az egyes felhasználóknak az erőforrások kizárólagos használatát. Egy ennél kifinomultabb módszer a tranzakció-kezelő mechanizmus alkalmazása.

A JoCaml **nem teljesíti a konkurencia átlátszóságot**. A szinkronizálásra az egyetlen nyelvi elem a join-minta. A programozónak saját magának kell lekezelnie a versengő megosztással járó problémákat, a JoCaml-ben erre semmilyen automatizmus nincsen.

8.1.7. Meghibásodási átlátszóság

Az elosztott rendszerek egy népszerű alternatív definícióját Leslie Lamportnak köszönhetjük: *„Tudjuk, hogy elosztott rendszerrel van dolgunk, ha egy eddig teljesen ismeretlen gép leállása megakadályozza bárminemű munka elvégzését”*. Ez a leírás az elosztott rendszerek tervezésének egy újabb problémáját veti fel: a hibakezelést. Előfordulhat ugyanis, hogy az elosztott rendszerben bizonyos gépek meghibásodnak, vagy megszűnik velük a kapcsolat. A meghibásodási átlátszóság az erőforrások hibáinak, elérhetetlenségeinek és a helyreállító folyamatoknak az elrejtését jelenti, tehát a felhasználó nem észleli az erőforrás meghibásodásának következményeit, és a rendszer a hiba után automatikusan helyreállítja magát [9].

A hibák elrejtése az egyik legbonyolultabb feladat az elosztott rendszerekben, sőt bizonyos feltételek esetén lehetetlen megoldani. A fő probléma abban rejlik, hogy szinte lehetetlen

megkülönböztetni a „halott” erőforrást a valójában csak kínosan lassú erőforrástól. Például amikor egy nagyon leterhelt web-szerverhez szeretnénk kapcsolódni és a böngésző időtúllépés után jelenti, hogy a weblap nem elérhető. Ebben helyzetben a felhasználó nincs olyan információ birtokában, ami alapján eldönthetné, hogy a szerver ténylegesen leállt-e [9].

JoCaml-ben csak korlátozott lehetőségünk van az ilyen meghibásodások kezelésre: egy absztrakt helyet és az összes al-helyeit leállíthatjuk a `halt` primitívvel, és a `fail` primitív használatával lehetőség van az ilyen leállások detektálására és egy helyreállítási folyamatot indítására. Ezek a primitívek javítják a programok megbízhatóságát, azonban még **nem elégségesek a meghibásodási átlátszósághoz** [10].
A probléma az, hogy a JoCaml nem szolgáltat csendes felépülés mechanizmust; a programozónak ki kell találnia, hogy mit kezdjen ha problémák adódnak. Amire nekünk szükségünk lenne, az az automatikus helyreállás.

Hiányosságok: a JoCaml jelenlegi változata még béta-verzió, ezért tartalmaz hiányosságokat. Valójában hibakezelésnél még a fent felsorolt tulajdonságokat sem teljesíti maradéktalanul. A `fail` ugyanis csak abban az esetben ismeri fel, hogy egy absztrakt hely leállt, ha a leállítás ugyanabban a programban a `halt` használatával történt. Egy másik programbeli absztrakt hely leállását a `fail` csak akkor érzékeli, ha maga a program is leállt. Így ha azt akarjuk, hogy egy hely leállását a másik program is észrevegye, akkor le kell állítanunk magát a tartalmazó programot is, ami nem túl kifinomult módszer.

8.2. Skálázhatóság

Amikor az elosztott rendszerhez csatlakozó csomópontok száma növekszik, szűk keresztmetszet lehet, ha csak egy névszerver áll szolgálatban [10].

A JoCaml-ben a név-szerver az adatait egy hash-táblában tárolja (ami kapacitásának csak a gép memóriája szab határt). Azonban egy érték regisztrálásakor a regisztrálást végző kód operációs-rendszer specifikus hívásokat is végez, és ezen hívások szabnak gátat a név-szerver nagyságának. A név-szerver mérete tehát függ az operációs rendszertől, a tesztek alapján 500-2000 bejegyzést tud kezelni.

8.3. Heterogenitás

A heterogenitás azt jelenti, hogy az elosztott rendszerben lévő gépek akkor is együtt tudnak működni egymással, ha a gépek architektúrája és a rajtuk futó operációs rendszer gépenként más. A JoCaml-ben ez elméletuleg teljesül. (Azt biztosan elmondhatom, hogy ha az elosztott rendszer RedHat Linux, Debian és WindowsNT-s (CigWin-en keresztül) gépekből áll (mind Intel processzoron), akkor a heterogenitás teljesül, az ágensek képesek egyik platformról a másikra vándorolni. Egyéb operációs rendszereken még nem sikerült tesztelni.)

8.4. Biztonságos mobil-kód

Biztonságos mobil kód: nem csak típushelyesség, hanem a kód garantáltan azt csinálja, amit elvárunk tőle. Sajnos a JoCaml-nek semmilyen ilyen biztonsági jellemzője nincs.

9. Implementáció

9.1. A szálak kezelése [8]

A szálak nem a JoCaml nyelv részei, hanem az implementációjáé. Implementációs szinten a szál egy végrehajtási egység.

Egy szál életciklusa:

- létrejön,
- végrehajtás során a szál futása felfüggesztődhet, majd folytatódhat
- és végül megszűnik.

A szálakat közvetlenül nem lehet létrehozni (nincs „create_szál” utasítás), azonban jól lefektetett szabályok vannak arra vonatkozóan, hogy a JoCaml mikor készít új szálat. A szekvenciában lévő kifejezések például egyetlen szálon hajtódnak végre, míg a párhuzamos kompozíció operátor a folyamatait két külön szálon hajtja végre (egymástól függetlenül).

Például a következő program három szálat készít:

```
# spawn {
#   { print_int 1 ; print_int 2 ; print_int 3 ; } |
#   { { print_int 4 ; } |
#     { print_int 5 ; print_int 6 ; } }
# };;
-> 123456
```

A program outputja a három szál outputjának (`123`, `4` és `56`) keverékéből áll össze. Az eredmény például `124356` is lehetne, a lényeg az, hogy egy szálon belül szekvencia van.

Egy csatornára történő üzenetküldés egy új folyamatot indít el, és elméletileg a folyamatot egy új szálon kéne végrehajtania. Azonban hatékonysági okokból nem mindig készül új szál.
Például nézzük a következő példát: az egyik definíció végtelen sok szálat készít (`rabbit`), a másik pedig egyet, ami sose szűnik meg (`forever`):

```
# let def rabbit! () = print_string "+" ; rabbit  () | rabbit  ();;
# let def forever () = print_string "-" ; forever () ; forever () ; reply;;
val rabbit : <<unit>>
val forever : unit -> 'a
```

Az aszinkron `rabbit` csatornára történő üzenetküldés egy kis szálat hoz létre, ami kiírja a „+”-t, majd elindítja a `rabbit` két új példányát, két új szálat létrehozva, majd a kis szál megszűnik. Ellenben a szinkron `forever` csatornára történő üzenetküldéskor a küldő szál blokkolódik, amíg a csatorna be nem fejezi tevékenységét. Ezt kihasználva az implementáció nem készít új szálat, hanem végrehajtja az őrzött folyamatot a küldő szálon. Itt az üzenetküldés egy közönséges függvényhívásnak felel meg.

Az implementáció viselkedése megfigyelhető a következő programon:

```
# spawn {
#   rabbit () |
#   { print_newline () ; exit 0; } |
#   { forever () ; }
# };;
->
-> + - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

A program három szálat indít el párhuzamosan. Ha csak a folyamatokat vesszük figyelembe, akkor a program leállhat előre nem ismert számú „–” és „+” kiírása után. Azonban a szálakat tekintve

felfedezhető, hogy több „–”-nak kellene megjelennie a konzolon, mint „+”-nak, mert a „–”-osak egy szálhoz tartoznak, míg a „+”-osok különbözőkhöz. (Az összes szálnak versenyeznie kell az ütemezésért, ha a „–”-os szál kapja meg, akkor rögtön elkezdi kiírni a „–”-okat, ellenben a „+”-ossal, ami csak egy „+”-t ír ki, majd újból megindul a versengés az ütemezésért.)

Összegzésül – az implementációt figyelembe véve – a szálak informális magyarázata segít megjósolni a program kimenetelét. Azonban amit megjósolunk az csak egy valószínű kimenet, hiszen az ütemező nem-determinisztikus.

9.2. A join-kalkulus elméleti háttere

A join-kalkulus egy alapnyelv a párhuzamos és elosztott programozásra. A nyelv alapját a csatornák (port-nevek) és a join-minták alkotják. A join-mintákkal deklarálhatjuk a csatornákat és szinkronizálhatjuk a párhuzamos végrehajtást.

A join-kalkulus nagyjából úgy tekinthető, mint egy funkcionális kalkulus + join-minták. A számítást aszinkron folyamatok végzik, és a join-minták szolgáltatják a szinkronizációt.

A join-kalkulusban az alapvető érték a port-név. Bár a nyelv tartalmaz konstansokat és egyéb primitíveket, de ezekkel most nem foglalkozom, ugyanis a port-nevek elegendő kifejezőerőt szolgáltatnak. (Bár a példákban – az egyszerűség kedvéért – szerepelni fognak konstansok és egész számok is, de a lényeg most nem ezeken van.)

9.2.1. Szintaxis [11]

A modellben egy program folyamatokból és definíciókból áll.

Folyamat: $P ::= x\ (x_i^{i \in 1..p})$
$\quad\quad | \ \textbf{let}\ D\ \textbf{in}\ P$
$\quad\quad | \ P \mid P$

Definíció: $D ::= J \triangleright P$
$\quad\quad | \ D\ \textbf{and}\ D$

Join-minta: $J ::= x\ (x_i^{i \in 1..p})$
$\quad\quad | \ J \mid J$

9.2.2. Szemantika

- **x**: egy port-nevet jelöl [11], az x_i-k ($i \in 1..p$) jelölik a csatorna paramétereit (mind a formális mind az aktuális paraméternél ezt a jelölést használjuk).
- **P**: folyamat, lehet [11]
  - egy **üzenet** egy csatorna számára (az x_i-k ($i \in 1..p$) aktuális paraméterek – az üzenet tartalma); látszik, hogy egy üzenet több értékből is állhat (a csatorna *polyadic*). A név és az üzenet aritásának mindig egyeznie kell.
  - vagy egy lokális definícióval ellátott **folyamat**
  - vagy folyamatok párhuzamos kompozíciója (ez esetben az egyes folyamatok kiértékelése párhuzamosan történik)
- **D**: egy definíció, ami egy vagy több klózból áll. Minden klóz egy pár: egy join-mintát (J) és egy őrzött folyamatot (P) tartalmaz [11].
- **J**: join-minta, egy vagy több párhuzamosan kapcsolt csatornából áll (az x_i-k ($i \in 1..p$) formális paraméterek). A join-minta definiálja a port-neveket [11] (az **|** operátor segítségével többet is definiálhat egyszerre). A join-minta lineáris: egy port-név legfeljebb egyszer szerepelhet egy adott mintában (de egy port-név több `and`-el összekapcsolt definícióban lévő join-mintában is előfordulhat).

A join-minta fejezi ki a szinkronizációt. Amikor az adott join-mintában előforduló összes csatornán várakozik üzenet (vagy várakoznak üzenetek), akkor a klóz aktívvá válik, és az őrzött folyamat tüzel*het*. Amikor a tüzelés megtörtént azt mondjuk, hogy a *mintaillesztés* bekövetkezett. Az őrzött folyamat tüzelésekor felhasználja a join-mintájában lévő port-neveken várakozó üzeneteit (csatornánként egyet-egyet): a csatorna formális paraméterei felveszik az üzenet értékét és a rendszer kiértékeli az őrzött folyamatot.

Megjegyzések:

- A csatornák elsődleges elemek, így egy csatorna is lehet üzenet egy másik csatorna számára.
- Egy definíció, aminek a join-mintái neveket osztanak meg egymással kifinomult szinkronizációt fejezhet ki.

9.2.3. A megvalósítás

A join nyelv támogatja a szinkron port-neveket is, míg az alap join kalkulus nem (legalábbis közvetlenül nem). A szinkron csatornák visszaküldik az eredményt, egy picit olyanok, mint a függvények. Ezek a szinkron csatornák is alkalmazhatók tetszőleges join-szinkronizációban, ugyanúgy, mint az aszinkron port-nevek. Azonban az olyan program, amelyben szinkron csatornák is szerepelnek átalakítható alap join nyelvre [11].

A Join nyelv volt az első prototípus. A Join nyelvhez írt fordító és az interpreter is egy Objective Caml-ben írt program, ezért könnyű Objective Caml adattípusokat és függvényeket beépíteni a Join nyelv alap adattípusai és függvényei közé. A Join nyelv ezért tekinthető úgy, mint egy különálló nyelv (rengeteg primitívvel), de úgy is, mint egy Objective Caml-re épülő réteg [11].

A JoCaml rendszer a második implementáció. JoCaml-ben az összes join-kalkulusos konstrukció a párhuzamosságra, a kommunikációra, a szinkronizációra és a folyamat mobilitásra közvetlenül elérhető, az Objective Caml szintaktikus kiterjesztéseként. A futtató környezet oldaláról az eredeti Objective Caml futtató környezetet (amiben már van `Thread` könyvtár) ki kell egészíteni egy speciális „join” könyvtárral és elosztott garbage collector-ral. A fordító oldaláról az Objective Caml fordítót ki kell egészíteni egy fordítással a join-kalkulus forráskódról a „join” könytárbeli függvényhívásokra. Bár magát az Objective Caml bytecode fordítót is át kellett alakítani egy kicsit, hogy a mobilitást kezelni tudja [11].

9.3. A join-minták fordítása

A join-mintákat nagyon sokszor használják a programozók, mivel ez az egyetlen szinkronizációs primitív. Ezért az implementáció oldaláról nézve a join-minták és az ezzel összefüggésben lévő üzenetfelhasználás és szinkronizáció fordítása nagy figyelmet érdemel (a join-kalkulus implementálásának egy lényeges része).

A programok tartalmazhatnak szinkron és aszinkron csatornákat is. Mi az implementációban a mintaillesztésre koncentrálunk, ezért a szinkron nevek úgy viselkednek, mintha aszinkronok lennének, csak szállítanak egy hozzáadott folytatás-argumentumot. Az összes implementáció ennek az extra argumentumnak a kezelésére koncentrál, ennek azonban a mintaillesztésre nincs hatása.

A join-mintákat determinisztikus véges-állapotú automatára fogjuk lefordítani. A fordítást több példán keresztül fogom bemutatni.

9.3.1. Mintaillesztés a join-definícióban [11]

Nézzük a következő join-definíciót:

```
let A(n) | B() = P(n)
 or A(n) | C() = Q(n)
;;
```

Ez a definíció három port-nevet definiál: `A`, `B` és `C`. (`A` `1`-argumentumú, `B` és `C` `0`-argumentumú.) A definícióban két őrzött folyamat van: `P(n)`, `Q(n)`. A csatornák lehetnek szinkron vagy aszinkronok is (ez attól függ, hogy van-e „`reply … to …`” a hozzájuk tartozó őrzött folyamatban), a folyamatokról semmit sem tételezünk fel.

`P(n)` őrzött folyamat akkor tüzelhet, ha `A`-n és `B`-n is várakoznak üzenetek (`Q(n)`-nél hasonlóan: `A`-n és `C`-n). Tüzeléskor a folyamat formális paramétere (`n`) lecserélődik (más szóval kötődik) az egyik `A`-n várakozó üzenetre.

Ha véges-állapotú automatát akarunk használni, akkor ki kell alakítani a lehetséges állapotok véges halmazát. Mivel most a mintaillesztés a lényeg, és a minta lineáris, ezért csak a várakozó üzenetek megléte, illetve hiánya számít.

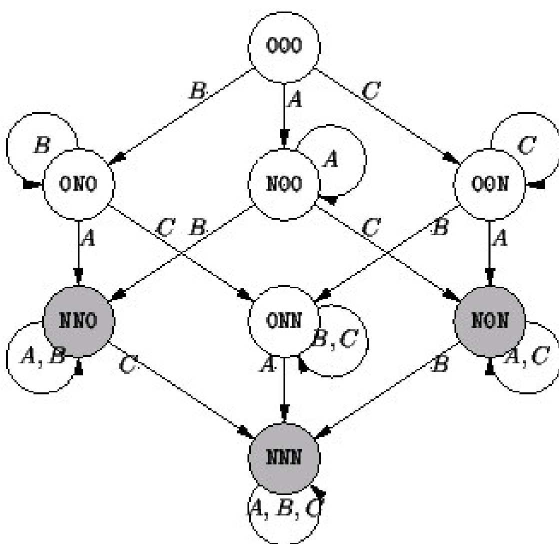
Rendeljünk: `0`-t az olyan csatornákhoz, amelyen nem várakozik semmilyen üzenet; és
`N`-t az olyanokhoz, amelyen legalább egy üzenet várakozik.

Az egész definíció állapota egy rendezett n-es (n különböző port-név szerepel a definícióban), amely valamilyen jól meghatározott sorrendben tartalmazza a port-nevek állapotait. Például, ha `A`-n nincs várakozó üzenet, `B`-n és `C`-n pedig van, akkor a definíció állapota: `0NN`.

Ezek a rendezett n-esek lesznek a gráf csúcsai. Az állapotok közül emeljünk ki egy részhalmazt: nevezzük ***illesztő állapot***-nak *(matching status)* az olyan állapotot, ahol van elég „`N`” ahhoz, hogy legalább egy őrzött folyamat tüzelni tudjon.

A gráf átmeneti legyenek a következők: ***növekvő átmenet***: ha egy új üzenet érkezik egy névre, akkor a név állapota `0`-ról `N`-re változhat, vagy maradhat `N`. A definíció állapota is eszerint változik, a növekvő átmenetek jelöljék ezt a változást.

A csúcsok és a növekvő átmenetek a következő gráfot adják:

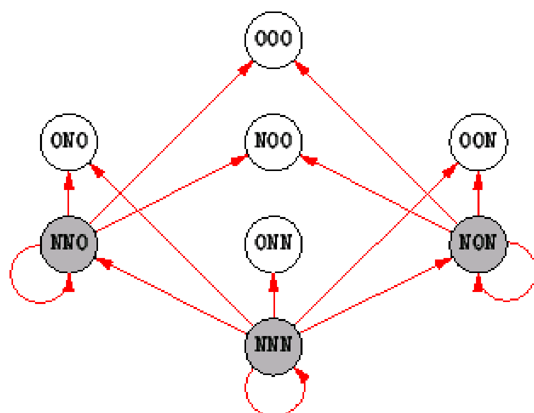


A csúcsok az állapotok, ezek közül a szürkék az illesztő állapotok.
Az átmenet címkéje azt jelöli, hogy melyik port-névre jött új üzenet.

Nem csak azt jelölhetjük egy gráfon, hogy mi történik egy új üzenet érkezésekor, hanem azt is, hogy mi lesz az üzenet felhasználásakor.

A gráf átmenetei most legyenek a következők: ***csökkenő átmenet*** *(illesztés)*: milyen állapotba jutunk, amikor a minta-illesztés bekövetkezik. A join-kalkulus szemantikája szerint az illesztés bármelyik pillanatban bekövetkezhet (persze csak akkor, ha az illesztés egyáltalán lehetséges). A csökkenő átmenet illesztő állapotból indul és nincs címkéje.

Csökkenő átmenetek használatával a következő automatához jutunk:



Megfigyelhető, hogy egy adott állapotból több átmenet is kiindul, ez azonban nem mindig következménye a join-kalkulus nem-determinisztikus szemantikájának.

Gyakran a kétértelműség csak látszólagos. Például `NN0`-ból négy él is megy (`NN0`-ba, `N00`-ba, `0N0`-ba és `000`-ba). Amikor az illesztés bekövetkezik, akkor két üzenet (1 db. `A`-n és 1 db. `B`-n várakozó) fogy el. Az, hogy ezután melyik állapotba jutunk, attól függ, hogy hány várakozó üzenet maradt `A`-n, illetve `B`-n. Az implementáció oldaláról nézve ez egy futási-idejű tesztet igényel illesztéskor – itt fizetjük meg a véges-állapotú automata árát.

De néhány igazi nem-determinizmus is jelen van. Például nézzük az `NNN` állapotot – ebben az állapotban mindkét őrzött folyamat tud tüzelni. Ha `P(n)` tüzel, akkor 1 db. `A`-n és 1 db. `B`-n várakozó üzenet, ha `Q(n)` tüzel, akkor pedig 1 db. `A`-n és 1 db. `C`-n várakozó üzenet fogy el. Tehát, hogy melyik állapotba jutunk, az attól (is) függ, hogy melyik őrzött folyamat tüzel.

Megjegyzés: Nem lineáris mintákra is lehet ilyen automatákat készíteni. Például, ha egy név legfeljebb kétszer szerepel egy mintában, akkor az adott név állapota lehet `0`, `1`, vagy `N`. Mi azonban ezzel az esettel nem foglalkozunk.

9.3.2. Determinisztikus automata

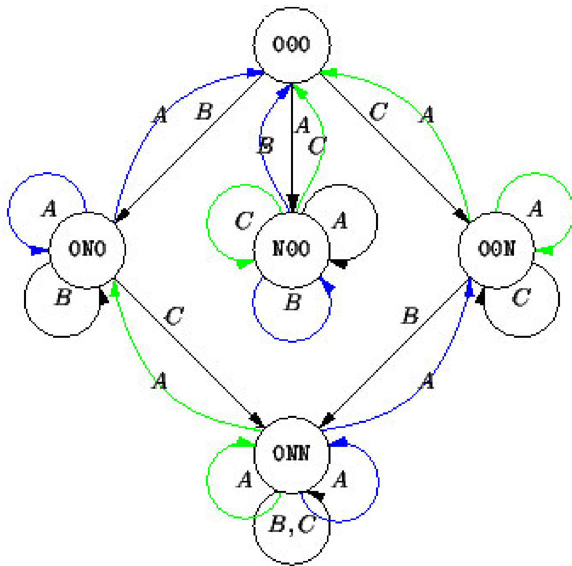
Ahhoz, hogy kifejezzük a join-minták szemantikáját mindkét automata szükséges, ezért most össze fogjuk vonni őket. Mivel a csökkenő átmenetet használó automata nem-determinisztikus, ezért az összevont automata sem lenne az.

A hatékonyság és az egyszerűség miatt azonban a minta-illesztés és az ezzel kapcsolatban lévő üzenet-felhasználás implementálására a determinisztikus automatát választották.

Ahhoz, hogy a determinisztikus automatát elkészítsük szükség lesz két plusz kikötésre.

1. Az egyik, hogy mindig végrehajtjuk az illesztést, amikor lehetséges. Tehát, ha egy üzenet érkezik, és illesztő állapotba érünk, akkor az illesztés azonnal megtörténik, és a definíció állapotát azonnal beállítjuk az üzenet-felhasználásra reagálva. Emiatt kevesebb állapot is lesz, ugyanis az illesztő állapotok eltűnnek (mert amint egy üzenet hatására ilyen állapotba lépünk, rögtön fel is használjuk az üzenetet és elkerülünk innen).
2. A nem-determinisztikusságot megszüntetjük.

Az előbbiek figyelembevételével és a két automata összevonásával a következő automatához jutunk (ez egyelőre még nem-determinisztikus):



átmenet címkéje: melyik port-névre jött új üzenet

kék élek: `P(n)` tüzeléséhez tartozó átmenetek

zöld élek: `Q(n)` tüzeléséhez tartozó átmenetek

fekete élek: üzenet érkezett, de semelyik őrzött folyamat nem tudott tüzelni

Itt már jól látszik a különbség az igazi és a nem igazi nem-determinizmus között: igazi nem-determinizmus akkor van jelen, amikor kék és zöld élek is kiindulnak egy adott állapotból ugyanazzal a címkével. Például: `N00`-ból két `B`-vel címkézett kék él is indul, itt a nem-determinisztikusság csak látszólagos, mert mindkét esetben `P(n)` tüzel, és az, hogy hova jut attól függ, hogy hány üzenet várakozik `A`-n `P(n)` tüzelése után. Ugyanakkor nézzük a `0NN` állapotot. Ebben az állapotban, ha `A`-ra érkezik üzenet, akkor `P(n)` és `Q(n)` is tud tüzelni. Ez két különböző viselkedést ad. Jól látszik az ábrán, hogy két `A`-val címkézett él is indul `0NN`-ből: egy kék és egy zöld.

Egy egyszerű mód az igazi nem-determinisztikus választás elkerülésére futásidőben, hogy végrehajtjuk a választást még fordítási időben. Ezt azt jelenti, hogy ahol ugyanazzal a címkével kék és zöld élek is vannak, ott az egyiket egyszerűen töröljük.

9.3.3. Automata és szemantika

Mind az „illeszt, amint lehet” viselkedésnek, mind az élek törlésének ára van a szemantikában (bizonyos viselkedések eltűnnek).

Nézzük a következő példát az „illeszt, amint lehet” viselkedés árára:

```
let A()        = print(1);
 or B()        = print(2);
 or A() | B() = print(3);
;;
```

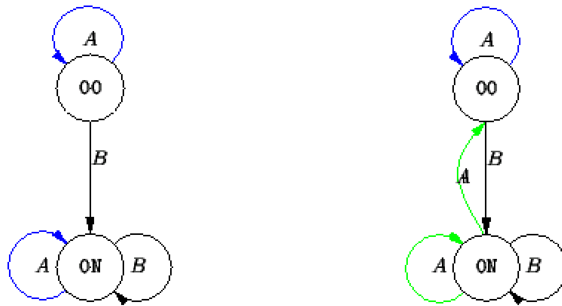
A következő automatát kapjuk:

Ez a program sose ír ki 3-at, mert `NN` és az ezt megelőző `0N` és `N0` állapotok nem elérhetők (illesztő állapotok).

Most nézzük a következő példát az élek törlésének árára:

```
let A()        = print(1);
 or A() | B() = print(2);
;;
```

Ha végrehajtjuk a törlést, akkor a következő determinisztikus automaták valamelyikéhez jutunk (attól függően, hogy az ellentmondásos élek közül melyiket töröljük):



Az első automata csak `1`-et ír ki. A második automata `2`-t is ki tud írni, ha `B`-n várakoznak üzenetek. Mindkét automata helyes implementáció, bár a második jobbnak látszik, mert többféle viselkedést ad.

9.3.4. Futásidejű definíciók

A csatornák a join-kalkulus alap elemei, ezért a join-kalkulus bármely implementációjának szolgáltatnia kell a futásidejű reprezentációjukat. Azok a csatornák amik együtt definiálódnak egyazon join definícióban a futás közben is szinkronizálnak egymással (a mintaillesztés végrehajtásakor tesztelni kell őket és fogyasztani róluk). Ezért ezeket a csatornákat futás közben is nyilván kell tartanunk: a fordítás során a *forrásidejű definíciót* le kell fordítani *futásidejű „definíció”*-ra, ami lehet például egy vektor-struktúra, ami csoportosítja az egyes definíciókban szereplő neveket. A csatornáknak létezniük kell egyedien is.

A JoCaml rendszer implementálja az előző részben bemutatott automatát. A rendszer minden üzenet érkezésekor egy szisztematikus futásidejű tesztet hajt végre az adott állapotban.

A JoCaml a csatornákat egy mutatóval (ami a definícióra mutat) és egy index-el reprezentálja. A definíció (a futásidejű „definíció”) egy vektor-struktúra, ami összefogja az összes csatornát, ami az adott join-definícióban szerepel. A definícióban minden csatornának egy bejegyzése van. A csatorna bejegyzése egy üzenetsort tartalmaz az adott névre küldött üzenetek számára. Ezeken kívül a definícióhoz tartozik még egy őrzött klóz (ebben szerepel az őrzött folyamat kódja és a folyamat szabad változóinak értékei), egy állapot-mező és egy illesztő adatstruktúra.

Az **állapot-mező** tartalmazza a definíció aktuális állapotát. Az állapot-mező biteket tartalmaz; minden port-névhez tartozik egy bit (az, hogy melyik port-névhez melyik bit tartozik, a név index-éből derül ki). Az 1-es bit jelöli azt, hogy a név állapota `N`, a 0-ás bit, hogy az állapot `0` (lásd az előző részben a definíció állapotát).
Az **illesztő adatstruktúra** egy tömb, ami egy őrzött folyamat index-éből és egy mintából álló párokat tartalmaz. A minta bitekből áll, ugyanúgy, mint a definíció állapota, egy bit akkor `N`, ha az indexnek megfelelő port-név szerepel a join mintában. Az illesztő adatstruktúra tehát azt írja le, hogy melyik őrzött folyamathoz melyik csatornákon kell, hogy várakozzanak üzenetek.

Az üzenetküldés egy általános C függvény (a „join” könyvtárból) meghívásával valósul meg. A függvény paraméterei: az üzenet értéke, a definíció mutatója és a csatorna indexe. Amikor üzenet érkezik egy csatornára, akkor a csatornához tartozó bit ellenőrzésre kerül. Ha a bit már be volt állítva, az azt jelenti, hogy már vannak üzenetek a csatornához tartozó üzenetsorban. Ebben az esetben a definíció állapota nem változik. Mivel az üzenet érkezése előtti állapot garantáltan nem illesztő állapot (mert a folyamatok azonnal tüzelnek, amint tudnak), így nincs más teendő, mint az üzenetet berakni az üzenetsorba.

Ellenkező esetben (ha ellenőrzéskor a bit nincs beállítva) be kell állítani a definíció aktuális állapotát. Azt, hogy illesztő állapotban vagyunk-e, az illesztő adatstruktúra segítségével határozza meg a rendszer (szekvenciális kereséssel az illesztő adatstruktúrában, és minden mintára bitenkénti *és*-sel). Így meg lehet állapítani, hogy illesztő állapotban vagyunk-e (a definíció állapota megfelel-e az illesztő adatstruktúra valamelyik mintájának), és ha igen, akkor melyik őrzött folyamatnak kell tüzelnie. A kérdés eldöntése legfeljebb N_x darab tesztet igényel, ahol N_x a mintában lévő `x`-et tartalmazó klózok száma (amelyeknek a mintája tartalmazza ezt a csatornát).

Ha nem vagyunk illesztő állapotban, akkor beállítjuk az automata állapotát, és az üzenet bekerül a név üzenetsorába.
Ellenben, ha illesztő állapotban vagyunk, akkor az illesztő struktúrából kapott indexű őrzött folyamat fog majd tüzelni. A folyamat a paramétereit az üzenetsorokból kapja (az adott folyamathoz tartozó illesztő állapot határozza meg, hogy melyik csatornákról kell egy értéket levenni). Az aktuális állapotot ebben a pillanatban frissíteni kell (ha a sor kiürül, akkor a megfelelő bitet 0-ra kell állítani). Végül az őrzött folyamat tüzel.

10. Összehasonlítás a π-kalkulussal

A π-kalkulus egy párhuzamos folyamatok leírására szolgáló nyelv, melyben a párhuzamos szálak csatornákon keresztüli üzenetküldéssel tartják egymással a kapcsolatot, és lehetőség van a párhuzamos szálak szinkronizálására is. A JoCaml csatornái és join-mintái lényegében lefedik a π-kalkulus képességeit, egy primitív kivételével: ez pedig nem más, mint a kétirányú csatorna. Azonban ez a nyelvi elem könnyen megvalósítható JoCaml-ben is.

A π-kalkulus szintaktikája (részlet):

- Érték küldése a csatornára: `csatorna![érték]` – az `érték` bekerül a `csatorna`-hoz tartozó várakozási sorba.
- Érték olvasása a csatornáról (a várakozási sorából): `csatorna?formális_paraméter.kifejezés` – a `formalis_parameter` felveszi a `csatorna`-ról leolvasott értéket, majd a `kifejezés` kiértékelődik.
- Párhuzamos végrehajtást itt is a „`|`” operátor segítségével érhetünk el.
- Csatornákat a `new csatorna in kifejezés` segítségével hozhatunk létre – a `csatorna` hatóköre csak a `kifjezés`-re korlátozódik.

10.1.1. Kétirányú csatornák megvalósítása

Kétirányúnak nevezzük azt a csatornát, amelyre írni, és onnan olvasni is lehet. Kétirányú csatornák a legtöbb folyamatalgebrai nyelvben megtalálhatók. JoCaml-ben a csatornákra „hivatalosan” csak írni lehet, azonban két csatornával lehet emulálni egy kétirányút [8]:

```
# let def new_pi_channel () =
#   let def send! x | receive () = reply x in
#   reply send, receive
# ;;
val new_pi_channel : unit -> <<'a>> * (unit -> 'a)
```

Példa a kétirányú csatorna használatára: [a következő pi-kalkulusbeli folyamat „fordítása”:

```
new c,d in c![1] | c![2] | c?x.d![x+x] | d?y.print(y)]
```

```
# spawn {
#   let sc, rc = new_pi_channel ()
#   and sd, rd = new_pi_channel () in
#   sc 1 | sc 2 | {let x = rc () in sd (x+x)} | {let y = rd () in print_int y ;}
# };;
-> 2
```

Kétirányú szinkron csatorna emulálása:

```
# let def new_pi_sync_channel () =
#   let def send x | receive () = reply x to receive | reply to send in
#   reply send, receive
# ;;
val new_pi_sync_channel : unit -> ('a -> unit) * (unit -> 'a)
```

A π-kalkulus bővebb tárgyalása túlmutat a dolgozat keretein. Az érdeklődő olvasó például Joachim Parrow: An Introduction to the π-Calculus című leírásban nézhet utána. Ez a leírás megtalálható a hálózaton is a következő címen: http://www.cs.auc.dk/~hans/Dat5/Artikler/intro.ps (2004. június 8.).

11. Javaslat új nyelvi elemek bevezetésére

11.1. Több, egymással kapcsolatban álló név-szerver

11.1.1. A JoCaml név-szerverének hiányossága

Amikor az elosztott rendszerhez csatlakozó csomópontok száma növekszik, szűk keresztmetszet lehet, ha csak egy névszerver áll szolgálatban. JoCaml-ben ezért lehetőségünk van egyszerre több név-szerver használatára is. Azonban ezek a név-szerverek semmilyen kapcsolatban nincsenek egymással: ha a program bejegyez egy erőforrást az egyik név-szerverbe, azt utána csak ettől a név-szervertől lehet lekérni; a használat során minden programnak pontosan kell tudnia, hogy melyik erőforrást melyik név-szerverben kell keresnie. Ráadásul a név-szerverek kapacitása véges (500 – 2000, operációs-rendszer függő), így a név-szerverek telítődésének figyelése is a programozó gondja.

Jobb megoldás lenne az elosztott névszerver használata. Ennél a megoldásnál amikor több név-szervert indítunk el, akkor az egyes szerverek nem lennének teljesen önállóak, hanem valamilyen módon kapcsolatban állnának egymással: így egy erőforrást nem csak attól a név-szervertől lehetne lekérni, amelyiken azt regisztrálták, hanem akármelyik másiktól is, és az eredmény nem függne attól, hogy melyik név-szervert használtuk. A név-szerverek telítődésének figyelése is automatikus lenne: ha az egyik név-szerver megtelik, akkor a rendszer a regisztrációt automatikusan egy másikban végzi el, és ez a művelet a felhasználó elől teljesen rejtett lenne.

Az elosztott név-szervernek egyetlen nagy név-szerverként kell látszania. Bár célszerű megtartani azt a lehetőséget is, hogy a felhasználók az egyes névszerverekhez külön-külön is hozzáférjenek.

11.1.2. A megvalósítás terve

JoCaml-ben a név-szervert Objective Caml-ben írt könyvtári függvények és C-ben írt programok valósítják meg. Szerencsére a név-szerver megvalósítása olyan, hogy az elosztottság megvalósítása – bár nem triviális feladat – kényelmesen beleilleszthető a programba.

11.1.2.1. Név-szerverek összekapcsolása

A cél megvalósítása érdekében az első feladat, hogy az egyes név-szervereknek valamilyen módon meg kell találniuk egymást. Ez nem egy magától érthetődő probléma, mert azok a név-szerverek amelyek egymással kapcsolatban állnának a világ különböző pontjain is lehetnek. Ráadásul valahogyan szabályozni kell azt is, hogy csak az általunk kívánt név-szerverek álljanak egymással kapcsolatban (egymástól teljesen különböző feladatokon dolgozó elosztott programoknál nem szükséges, hogy a név-szerverjeink összekapcsolódjanak). Ki kell dolgozni egy egységes módszert, hogy az összetartozónak gondolt név-szerverek megtalálják egymást.

Egy lehetőség, hogy bevezetünk egy felügyelő névszervert. Minden „normális” név-szervernek indításkor ismernie kell a felügyelő név-szerver fizikai címét. A név-szerver megvalósításának terminológiája szerint ezt egy környezeti változó (például `JNSSUPER`) tárolná, majd a program futása során az `Ns` modulban egy `string ref` típussú érték (például `Ns.name_super`). A rendes név-szerverekhez hasonlóan esetleg a felügyelő port-száma is állítható lehetne, de lehet konstans is. A név-szerver kódját (`Ns_server` modul) úgy kell módosítani, hogy induláskor küldjön egy előre meghatározott szerkezetű üzenetet a felügyelő név-szervernek, jelezve a rendszerbe történő bejelentkezését. A felügyelő név-szervernek gyűjtenie kell a bejelentkezett név-szervereket, ezt megteheti például egy hash-tábla használatával.

A felügyelő név-szerver így összefogja a kapcsolatban álló név-szervereket: azok a név-szerverek tartoznak egy rendszerbe, amelyek a felügyelőnél be vannak jegyezve.

11.1.2.2. Egy érték regisztrálása az elosztott név-szerverben

A regisztrálási kérelem irányulhat egy tetszőleges valódi név-szerverhez. A regisztrálást – eddig is – az `Ns.register` függvény hívásával lehetett kezdeményezni, azonban a regisztrálás tényleges megvalósítását az `Ns.request` függvény végezte (ez az `Ns` modulban egy rejtett függvény).

Az `Ns.request` kódján minimális változtatás szükséges: az `Ns.request`-nek ugyanúgy kell elvégeznie a regisztrálást, mint eddig. Esetleg annyit lehetne még beleírni, hogy minden regisztráláskor jelentse a felügyelő név-szervernek, hogy hány elemet tartalmaz (így a felügyelő majd tudja figyelni az egyes név-szerverek telítődését). Az `Ns.request` kódján ténylegesen csak akkor kell változtatni, ha a név-szerver már nem tud több kérést fogadni (a méret operációs-rendszer függő). Ekkor a név-szerver továbbítja a kérést a felügyelő név-szerver felé, aki majd egy másik név-szerverben fogja regisztrálni az értéket.

A programok maguk is küldhetnek regisztrálási kérelmet a felügyelő név-szervernek, ha ismerik a címét (és esetleg a port-számát). Fontos, hogy a felügyelő név-szervert kívülről teljesen ugyanúgy lehessen kezelni, mint bármelyik más név-szervert, a programozónak – ha nem akar – nem is kell törődnie azzal, hogy a felügyelővel, vagy egy rendes név-szerverrel kommunikál.

Ha a felügyelő név-szerver kap egy regisztrálási kérelmet, akkor azt nem maga tárolja el, hanem továbbítja egy valódi név-szervernek. Ekkor vesszük hasznát, hogy a felügyelő név-szerver azt is gyűjti, hogy melyik valódi név-szerver mennyi elemet tartalmaz, mert ebben az esetben a regisztrálást a legkevésbé leterheltnek továbbíthatná.

11.1.2.3. Egy érték lekérése az elosztott név-szerverből

A lekérés úgy történik, mint eddig: ha lekérési kérelem érkezik egy név-szerverhez, akkor a név-szerver átküldi az értéket a kérőhöz. A lekérést megvalósító függvény (ezt is az `Ns.request` valósítja meg) kódján akkor kell módosítani, ha a kérdezett értéket a név-szerver nem találja. Eddig ebben az esetben a név-szerver egy kivételt váltott ki, most azonban ehelyett továbbítania kell a lekérdezési kérelmet a felügyelő név-szervernek. Az értékeket közvetlenül a felügyelő névszervertől is lehet lekérni.

Ha a felügyelő név-szerverhez egy lekérdezési kérelem érkezik, akkor az elkezdi keresni értéket a bejegyzett név-szervereiben (ez történhet párhuzamosan is). Ha valahol megtalálta, akkor a megtalált név-szerver szolgálja ki a kérőt.

Ez a megoldás azonban még nem tökéletes. Képzeljük el azt a helyzetet, amikor két különböző név-szerverben regisztrálunk ugyanazon a logikai néven. Ha később ezt a nevet nem a konkrét név-szerverektől, hanem a felügyelőtől kérjük le, akkor a visszaadott érték attól függ, hogy melyik név-szerver találja meg hamarabb. Ez a nem-determinisztikusság úgy kerülhető el, ha a név-szerver minden regisztrálás előtt elküldi a regisztrálandó azonosító nevét a felügyelőnek, aki megnézi, hogy van-e ilyen nevű elem a rendszerben. Ha ilyen érték már van, akkor – a JoCaml név-ütközési koncepciója szerint – a régebbit törölnie kell.

11.1.2.4. Az elosztott rendszer használata

A JoCam programokban semmiféle változtatást nem kell tenni, a programok nyugodtan használhatnak egy név-szervert úgy, mint eddig. Elindíthatunk több név-szervert és egy felügyelőt is, és használhatjuk egyedül a felügyelő név-szervert, aki majd egyenletesen szétosztja a regisztrálandó értékeket a valódi név-szerverek között.

Azonban a programozók készíthetnek olyan programokat is, ahol az értékeket különböző név-szerverekbe regisztrálhatják (amint ezt eddig is lehetett), most azonban lekérdezéskor akármelyik név-szervertől lekérhetik az értéket, függetlenül attól, hogy hova regisztrálták. További előny, hogy az egyes név-szerverek kapacitáskorlátját a rendszer elfedi előlünk: ha megtelt a név-szerver, akkor automatikusan egy másikba regisztrálja az értéket, a felhasználó ebből semmit sem vesz észre.

11.1.2.5. Továbbfejlesztés: cache-elés

Ha a programozó egy bizonyos név-szervertől kér le egy értéket és nem a felügyelőtől, annak az is lehet az oka, hogy birtokában van olyan információknak, hogy melyik név-szerver van hozzá fizikailag a legközelebb. Ezért célszerű a név-szervereket úgy megvalósítani, hogy ha lekérdezéskor a név-szerver nem tartalmazza az értéket, akkor a bejegyzés másolódjon ide, így a következő ideirányuló lekérdezéskor már itt is megtalálható lesz. Így a bejegyzések ugyan többszöröződnek, de a hatékonyság növekedhet, főleg ha a bejegyzést lekérő és a bejegyzést eredetileg tartalmazó név-szerver egymástól távol van vagy közöttük a hálózat lassú: ekkor ugyanis a programozó nyithat egy név-szervert a közelben és a kéréseit egyenesen hozzá intézheti. Ekkor minden értéknél csak az első lekérés tart sokáig.

A valódi név-szerverek nyilvántarthatnák, hogy melyik értéket mikor kérdezték le tőlük utoljára. Ha a név-szerver betelt, akkor nem az újonnan jövő kérelmet irányítja át egy másik szerverhez, hanem a legrégebben használtat.

Ha a bejegyzések több példányban találhatók a rendszerben, akkor figyelni kell arra, hogy az egyes szervereken tárolt adatok mindig konzisztensek legyenek. Ha például egy felüldefiniálás jön, akkor célszerű a bejegyzést minden helyen átdefiniálni.

Ezek a változtatások már egy kicsit a replikációs átlátszóság felé viszik a rendszert.

11.2. Replikációs átlátszóság megvalósítása

A replikációs átlátszóság lényege, hogy az egyes erőforrásokról másolatokat készíthetünk (például a hozzáférési hely közelében, így az erőforrás elérése gyors(abb) lesz), a rendszer azonban a többszörözött erőforrások létezését elrejti a felhasználó elől: a másolatok mindig konzisztensek maradnak, tehát akármelyiket is változtatjuk, a rendszer a módosítást minden másolat-példányban könyvvelni fogja (esetleg nem egyből, csak szükség esetén, de ez már hatékonysági implementációs kérdés).

A replikációs átlátszóság megvalósítása során a két legnagyobb megoldandó feladat a másolatok létrehozása és a konzisztens állapot karbantartása.

A JoCaml egyáltalán nem teljesíti a replikációs átlátszóságot, még a másolatok készítésére sem áll rendelkezésre semmilyen nyelvi elem. Az átlátszóság JoCaml-beli tényleges programozói megvalósítása eléggé nehéz és szerteágazó feladat.

11.2.1. A replikációs átlátszóság modellezése

Azonban írhatunk olyan programot, ami a replikációs átlátszóságot valamilyen szinten modellezi: az erőforrás egy absztrakt hely, a másolatokat egy speciális csatorna segítségével készítjük el, ami az erőforrást egy megadott helyre vándoroltatja, a konzisztens állapotot pedig visszahívásokkal tartjuk fenn.

Vegyünk például egy „osztály”-t, ami egy egész szám referenciát valósít meg. Az „osztály”-t csatornákkal valósítjuk meg (az osztálylétrehozás vándoroltatásának hiánya miatt) egy absztrakt helyen (`szam_kiir`) belül. Az „osztály” értékét, a számot ki lehet íratni (`kiir: unit -> unit`)

és meg lehet változtatni (`modosit: int -> unit`), a szám értékért a `szam` (`<<int>>`) csatorna tárolja (minden pillanatban legfeljebb egy érték tartózkodhat a csatornán: a szám értéke). Az adatszerkezetet a `keszit` szinkron csatorna hozza létre, majd a paraméterként kapott helyre (`there`) vándoroltatja (ezért szükséges a csatornákat egy absztrakt helyen belül létrehozni). A `keszit` csatorna csak a `kiir` és a `modosit` csatornákat exportálja, így az adatszerkezet belső felépítéséhez kívülről nem lehet hozzáférni.

Ezzel a módszerrel el lehet érni, hogy egy adatszerkezetről tetszőleges számú másolatot készítsünk tetszőleges helyekre. Azonban az eddig leírtak alapján ezek a másolatok (melyek fizikailag különböző helyen is lehetnek) teljesen függetlenek egymástól. Tehát ezek csak létrehozáskor másolatok, utána már külön életet élnek, így ezt még nagyképűség lenne a replikációs átlátszóság modellezésének hívni. Meg kell oldani, hogy ezek a különböző helyen lévő adatszerkezetek végig konzisztensek maradjanak: ha valamelyik másolatnak megváltozik az értéke, akkor a többi másolat értéke is ennek megfelelően változzon.

A feladat tehát az, hogy a `modosit` csatornát kiegészítsük egy visszahívással (`modositas (y, n)`) az eredeti helyre, mely visszahívás paraméterében szállítaná, hogy hányadik másolat (`n`) értéke mire változott (`y`), a `modositas` csatorna ennek megfelelően módosítaná a többi másolat értékét. Ahhoz, hogy a többi másolat értékét meg tudja változtatni szükséges, hogy minden másolatnál nyilvántartsa a módosításhoz szükséges csatornát (`modosit_safe: int -> unit`). A szerver program a különböző másolatokhoz tartozó módosító csatornákat egy módosítható listában tartja nyilván (`modositok`). A többi másolat módosítását a `modositas0` rekurzív csatorna (`(int * (int->unit) list * int * int) -> unit`) végzi: a listában szereplő minden módosító-függvényt – az utolsó paraméterként kapott indexűt kivéve – meghív az első paraméterben kapott új értékkel (a harmadik paraméterében számolja, hogy hányadik indexű másolatnál tart). A `modositas0` nem az eredeti módosító függvényt (`modosit`) hívja meg, hanem egy másikat (`modosit_safe`), erre azért van szükség, hogy az érték módosítása nem indítson el egy új másolat-módosítási kört (a `modosit_safe` nem tartalmaz visszahívást a `modositas`-ra).

Ezzel a módszerrel elérhetjük, hogy ha akármelyik másolat módosul, akkor a többi is ennek megfelelően módosuljon. Egy dolog hiányzik még: a kezdeti érték dinamikus beállítása. Ha létrehozunk egy új másolatot miközben már léteznek példányok a rendszerben, akkor ennek az új másolatnak az értékét a többi másolattal konzisztens állapotúra kell állítani. Ezért a szerveren mindig nyilván kell tartani a másolatok aktuális értékét: `aktualis_ertek` (`int ref`).

A program:

```
let (modositok : (int -> unit) list ref) = ref [];;

let def modositas0 (x, lista, i, n) =
   let vt = match lista with
        []   ->  ()
      | e::l ->  if i = n then modositas0 (x, l, (i+1), n)
                 else    (e x; modositas0 (x, l, (i+1), n))
   in reply vt
;;

let aktualis_ertek = ref 0;;

let def modositas (x, n) =
   aktualis_ertek := x;
   modositas0 (x, !modositok, 1, n);
   reply
;;

let generator = ref 1;;
```

```
let def index_ad () = generator := !generator + 1; reply (!generator-1);;

let def keszit (there) =
   let n = (index_ad ()) in
   let loc szam_kiir
      def szam! x | kiir () =
         print_string (string_of_int n^". erteke: "^string_of_int x^"\n");
         flush stdout; {szam x | reply}
       or szam! x | modosit y =
         print_string (string_of_int n^". modosit: "^string_of_int y^"-ra/re\n");
         flush stdout; (modositas (y, n)); {szam y | reply}
       or szam! x | modosit_safe y =
         print_string (string_of_int n^". modosul: "^string_of_int y^"-ra/re\n");
         flush stdout; {szam y | reply}
      do { go there;
           print_string (string_of_int n^". masolat elkeszult\n");
           flush stdout;
           spawn {modosit_safe !aktualis_ertek;}; szam 0
         }
   in modositok := (!modositok @ (modosit_safe :: []));
      reply kiir, modosit
;;

Ns.register "keszit" keszit vartype;;
Join.server ();;
```

Ez nem csak egy példa, hanem egy módszer leírása is egyben. A 7.2.2. Általánosítás: osztály megvalósítása csatornákkal és join-mintákkal című fejezetben leírtak szerint bármilyen adatszerkezetet megvalósíthatunk csatornákkal. Ha az „osztály” több adattaggal rendelkezik, akkor mindegyikhez nyilván kell tartani az aktuális értéküket és a módosító csatornájukat és a szerver-programot eszerint kell elkészíteni.

11.3. Kifinomultabb meghibásodási átlátszóság

A JoCaml fejlesztése során a leállás-detektálás terén legelőször azt a problémát kell megoldani, hogy a `fail` primitív tudja érzékelni egy másik programbeli ágens leállását is. Csak ezután lehet elgondolkozni egyéb (fejlettebb) hibakezelő mechanizmusok kidolgozásán, az automatikus helyreállítás megoldásán.

Sok olyan szituáció létezik, amikor egy ágens leállhat, meghibásodhat – ha a rendszer szét tudja választani az eseteket, akkor lehetősége van valamilyen szintű automatikus helyreállításra [10]:

- A gép összeomlik, mielőtt az ágens elindulhatott volna. Ebben az esetben egyszerűen el kell indítanunk az ágenst, amint a gép működőképes lesz.
- Az ágenst futtató gép az ágens futtatása közben omlik össze. Ebben az esetben újra kell indítani az ágenst. Az előző futás során (talán) tárolt részeredményeket felhasználhatjuk.
- Az ágens befejezte futását, de további teendőkre van szükség. Ha az ágensnek vissza kell küldenie a jelentést, vagy egy másik helyre vándorolna, elküldhetjük a jelentést tartalmazó üzenetet, vagy vándoroltatjuk az absztrakt helyet, amint a megfelelő gép működése helyreállt.
- Az ágens befejezte futását és további műveletekre nincs szükség. Ebben az esetben az ágenst egyszerűen elvethetjük.

Ezen helyzetek kezeléséhez az ágenseinknek speciális állapotokkal kell rendelkezniük. Az ágensünknek van egy vezérlő állapota, ami az ágens futásának állapotáról, a további teendőkről, a küldőről és a tulajdonosról tartalmaz információkat. Az ágens futásának állapota tájékozatja a middleware réteget, hogy az ágens elindult-e, befejezte-e a működését, és a további feladatok

(például: jelentés küldése, vagy vándorlás) készen vannak-e, vagy sem. Ezt az információt felhasználhatjuk arra, hogy eldöntsük, melyik fent leírt esemény következett be az ágenst futtató gép összeomlásakor. Ha a rendszer leállása az ágens futtatása közben történt, a middleware odaadhatja a kiértékelési állapotban eltárolt már kiszámolt adatokat az ágensnek (így ezeket már nem kell még egyszer számítani).

12. Függelék - tesztprogramok

Ebben a részben az érdekesebb tulajdonságokra, problémákra mutatok egy-egy szemléltető példát.

12.1. Paraméterátadás

A függvények esetében a paraméterátadás az imperatív nyelvek *cím szerinti* paraméterátadásának felel meg. Tehát, ha a függvényben valamely paraméter értékét megváltoztatjuk, akkor ez a változtatás a függvényből kilépve is megmarad.

Példaprogram: a függvény a paraméter értékét megnöveli eggyel, majd visszaadja ennek kétszeresét.

```
# let proba r =
    r := !r + 1; !r*2;;
val proba : int ref -> int
# let i = ref 2;;
val i : int ref = {contenst=2}
# proba i;;
- : int = 6
# !i;;
- : int = 3
# proba i ;;
- : int = 8
```

Látható, hogy az `i` referencia értéke minden függvényhíváskor eggyel növelődött.

12.2. Lokálisan deklarált azonosító élettartama

A függvényen belül létrehozhatunk lokális azonosítót:

`let` *`függvénynév paraméterek`* `= let` *`név`* `=` *`kifejezés`*$_1$ `in` *`kifejezés`*$_2$

A *`név`* azonosító lokális a függvénytörzsre nézve. Minden függvény-hívásnál létrejön egy *`kifejezés`*$_1$ értékét tartalmazó, *`név`* nevű lokális azonosító, melyre csak a függvény törzséből (*`kifejezés`*$_2$) lehet hivatkozni. Azonban a lokális azonosító élettartama nem feltétlen korlátozódik a csak a függvény törzsére.

Példaprogram: definiálunk egy függvényt (`szamlalo_keszit`), mely két lokális azonosítót hoz létre: egy `int ref` típusú értéket (`aktualis_ertek`), amely a számláló aktuális értékét tartalmazza, és egy `unit -> unit` típusú függvényt (`novel`), amely az aktuális értéket növeli eggyel, majd kiírja az értékét. A `szamlalo_keszit` függvény visszatérési értéke a `novel` függvény. A `novel` függvényértéket a `szamlalo_keszit` függvény visszaadja, így ez a függvény nem szűnik meg a programból kilépve sem. Azonban ez a függvény hivatkozik az `aktualis_ertek` lokális azonosítóra, így ez az azonosító sem fog megszűnni a kilépés után: ugyan kívülről nem lehet elérni az azonosítót, de a `novel` függvénytörzse tud rá hivatkozni (mert az benne van a lokális azonosító hatókörében).
A `szamlalo_keszit` függvény minden meghívásakor létrejön két lokális azonosító (`aktualis_ertek`, `novel`), melyek nem szűnnek meg a függvényből kilépve sem (csak kívülről nem lehet elérni őket). Ezek a létrejött azonosítók mind különbözőek.

```
# let szamlalo_keszit kezdeti_ertek =
#   let aktualis_ertek = ref kezdeti_ertek in
#   let novel () =
#      aktualis_ertek := !aktualis_ertek + 1;
#      print_string
#         ("A szamlalo erteke: "^string_of_int !aktualis_ertek^".\n") in
#   print_string
#      ("Szamlalo letrehozva, erteke: "^string_of_int kezdeti_ertek^".\n ");
#   novel;;
val szamlalokeszit : int -> unit -> unit
```

Használat:

```
# let my_szamlalo_1 = szamlalo_keszit 0;;
val my_szamlalo_1 : unit -> unit
-> Szamlalo letrehozva, erteke: 0
# my_szamlalo_1 ();;
- : unit
-> A szamlalo erteke: 1
# my_szamlalo_1 ();;
- : unit
-> A szamlalo erteke: 2
# let my_szamlalo_2 = szamlalo_keszit 1;;
val my_szamlalo_2 : unit -> unit
-> Szamlalo letrehozva, erteke: 1
# my_szamlalo_1 ();;
- : unit
-> A szamlalo erteke: 3
```

Látható, hogy a különböző számlálókhoz tartozó értékek nem „keveredtek” össze, mindegyik létrejött azonosító egyedi.

12.3. Szinkron csatornára történő üzenetküldéskor a hívó felfüggesztése

Ha egy szinkron csatornára küldünk üzenetet, akkor a hívó az eredmény visszakapásáig (tehát a rá vonatkozó `reply` folyamatig, és nem a teljes tárolt folyamat kiértékelésének a végéig) függesztődik fel.

Példaprogram: A programban található egy csatorna, melynek törzse két párhuzamosan futó folyamatból áll. Az első folyamat egyből visszaad egy értéket a hívónak, míg a másik szál dolgozik (10 db. `"a"`-t ír ki a képernyőre). A csatornára érkezik egy hívás, majd közvetlen utána a hívó kiír egy üzenetet a képernyőre.

```
# let def proba () =
#     reply
#   | { for i = 1 to 10 do print_string "a"; flush stdout done; };;
# proba ();;
# print_string "\nAz hivo felfuggesztese befejezodott\n"; flush stdout;;
val proba : unit -> unit
->a
->Az hivo felfuggesztese befejezodott
->aaaaaaaaa
```

A program outputjából jól látható, hogy a hívó szál kiértékelése folytatódott tovább, pedig a hívást fogadó csatorna törzsének kiértékelése még nem fejeződött be. A hívó szál felfüggesztése tehát csak az eredmény visszakapásáig tart.

12.4. Másolat vagy referencia

Kérdés: Ha a névszervertől lekérünk egy értéket, akkor mit kapunk: az azonosító értékének másolatát, vagy csak a referenciáját?

1. **Függvények** esetében másolatot kapunk, így meghíváskor nem az eredeti függvény fog végrehajtódni, hanem a másolat, ezért a végrehajtás a lekérdezést végző gépen fog történni.

   Példaprogram: az **A** program regisztrál egy `f` függvényt (a függvény egy egész szám négyzetét számolja ki, azonban számolás előtt kiír egy tájékoztató szöveget a konzolra, hogy nyomon lehessen követni, hogy a végrehajtás hol zajlik), a **B** program lekéri a függvényt és használja.

   A program:
   ```
   let f x =
       print_string ("Feldolgozas : "^string_of_int(x)^"\n"); x*x;;
   Ns.register "f" f vartype;;
   Join.server ();;
   ```
   B program:
   ```
   let f = Ns.lookup "f" vartype;;
   print_string ("Az 5 negyzete: "^string_of_int (f 5));;
   ```
   Eredmény: azon a gépen, ahol az **A** program futott semmilyen eredmény nem keletkezett, a **B** program eredménye viszont a következő:
   ```
   Feldolgozas : 5
   Az 5 negyzete: 25
   ```
   Látható, hogy a függvény a **B** gépen futott le, tehát a **B** program lekérdezéskor megkapta a függvény másolatát.

2. **Csatornák** esetében azonban csak a csatorna referenciáját kapjuk meg, így meghíváskor az eredeti csatorna fog végrehajtódni, ezért a végrehajtás a regisztrálást végző gépen fog történni:

   Példaprogram: a feladat ugyanaz, mint az előbb, csak az `f` most nem függvény, hanem szinkron csatorna lesz.
   Mivel a függvények és a szinkron csatornák típusa megegyezik (azonos típusú paraméter és eredmény esetén), ezért a **B** programot nem kell változtatni. Az **A** program pedig a következő lesz:
   ```
   let def f x =
       print_string ("Feldolgozas : "^string_of_int(x)^"\n"); reply x*x ;;
   Ns.register "f" f vartype;;
   Join.server ();;
   ```
   Eredmény:
   A program eredménye: `Feldolgozas : 5`, **B** program eredménye: `Az 5 negyzete: 25`.

   Látható, hogy az eredeti gépen lévő csatorna futott le, tehát a **B** gép lekérdezéskor csak a csatorna referenciáját kapta meg.

3. **Objektumok** esetében is csak referenciát kapunk, így ha egy objektumot regisztrálunk, majd egy másik gépről lekérünk, akkor a két gépről ugyanazt az objektumot fogjuk kezelni.

   Példaprogram: az **A** program definiál és regisztrál egy objektumot, majd vár egy üzenetre. Ha ez megérkezett, akkor kiírja az objektum adattagjának értékét. A **B** program lekéri az objektumot, megváltoztatja az adattag értékét, majd jelez, hogy a másik program jön.

A program:

```
class osztaly (x0 : int) =
   val mutable x = x0
   method get = x
   method set x1 = x <- x1
end;;
let obj = new osztaly 5;;

let def kiir! () =
   print_string ("obj.x = "^(string_of_int obj#get)^"\n"); flush stdout; ;;

spawn {kiir ()};;
Ns.register "obj" obj vartype;;
Ns.register "kiir" kiir vartype;;
Join.server ();;
```

B program:

```
let (obj : <get : int; set : int -> unit;>) = Ns.lookup"obj" vartype;;
let kiir = Ns.lookup "kiir" vartype;;
obj#set 10;;
spawn {kiir ()};;
```

Eredmény: kimenet csak az **A** programot futtató gépen jelentkezik:

```
obj.x = 5
obj.x = 10
```

Látható, hogy a B program az A-beli objektumot kezelte.

4. **Egyszerű azonosítók** esetében másolatot kapunk. (Az az egyszerű azonosító, melynek értéke nem tartalmaz olyan kifejezést, vagy folyamatot, melynek kiértékelését csak meghívással lehet elérni; tehát egy egyszerű azonosító például `int`, `string`, lista, rekord, referencia stb. típusú értéket jelöl). Ezek az egyszerű típusok lehetnek változtathatók (*mutable*), vagy állandók. Állandó adattípusoknál a használat szemszögéből nincs különbség a között, hogy magával az azonosítóval, vagy csak a referenciájával dolgozunk. Ezért vizsgáljunk meg egy változtatható típust, például az `int ref`-et. Ugyanazt fogjuk tapasztalni, mint a függvények esetében: az azonosító többszöröződik, mind a két gépen lesz belőle egy példány, melyek nincsenek egymással kapcsolatban (ezért, ha az egyik gépen megváltozik az értéke, az a másik gépen lévő azonosító értékére nem lesz hatással).

   Példaprogram: az objektumos példához hasonlóan.

   A program:

```
let i = ref 5;;
let def kiir! () =
   print_string ("i = "^(string_of_int !i)^"\n"); flush stdout; ;;

spawn {kiir ()};;
Ns.register "i" i vartype;;
Ns.register "kiir" kiir vartype;;
Join.server ();;
```

   B program:

```
let i = Ns.lookup "i" vartype;;
let kiir = Ns.lookup "kiir" vartype;;
i := 10;;
spawn {kiir ()};;
```

Eredmény: kimenet csak az **A** programot futtató gépen jelentkezik:

```
i = 5
i = 5
```

Látható, hogy a **B** programbeli értékadás nem változtatta meg az **A**-beli azonosító értékét.

12.5. Csatornák várakozási sorának mérete

Kérdés: mekkora a csatornákhoz tartozó várakozási sor mérete? Van valami felső határ, vagy csak a memória mérete?

Példaprogram: A program tartalmaz két join-mintával összekapcsolt aszinkron csatornát (`a`, `b`). A csatorna őrzött folyamata csak akkor tud kiértékelődni, amikor mindkét csatornáján várakozik üzenet. A program azonban csak az `a` csatornának küld üzeneteket (parancssori paraméterben megadott darab-számút), ezért az összes üzenet az `a` csatorna várakozási sorába kerül és ott is marad.

```
let def a! i | b! j = print_int (i+j); ;;

let rec kuld i =
   if i>0 then
      (print_string (string_of_int i^". uzenet megy\n"); flush stdout;
       spawn {a i}; kuld (i-1));;

kuld (int_of_string Sys.argv.(1));;
```

Eredmény: 100.000.000-val simán meg lehetett hívni.

12.6. Standard input és output használata vándorlás során

Kérdés: Mi történik a standard input és a standard outputnak megfelelő értékekkel (`stdin`, `stdout`) a vándorlás során? **Válasz:** a rendszer mindig az aktuális gép standard inputját és outputját használja.

Példaprogram: Az ágens bekér a billentyűzetről két szöveget, majd összefűzve kiírja. Azonban az egyik szöveg bekérése után átvándorol egy másik gépre.

A program: ennek csak az a feladata, hogy várja a másk programot:

```
let loc here do {};;
Ns.register "here" here vartype;;
Join.server ();;
```

B program:

```
let here = Ns.lookup "here" vartype;;
let loc location do {
   output_string stdout "Elso resz: ";
   let s1 = input_line stdin in
   go here;
   output_string stdout "Masodik resz: ";
   let s2 = input_line stdin in
   output_string ("Az osszefuzott szoveg: "^s1^s2);
   flush stdout;
   exit 0
};;
```

Eredmény: amint azt vártuk a vándorlás előtt a **B** programot futtató gép be- és kimenetét használta, míg a vándorlás után az **A** programot futtató gépét.

12.7. Vándorlás fájl-ba írás közben

Kérdés: Mi történik, ha az ágens fájlba írást hajt végre, de közben átmegy egy másik gépre? Le tudja ezt a JoCaml valamilyen szinten kezelni (például ugyanúgy az eredeti fájlba ír bele, vagy létrehoz az új helyen is egy ilyen fájlt, vagy hibaüzenettel leáll)?

A program: legyen ugyanaz, mint az előbb.

B program:

```
let loc location
   do {
      let here = Ns.lookup "here" vartype
      and f = open_out_gen [Open_creat; Open_wronly] 0b110000000 "teszt.txt" in
      output_string f "Elso sor"; flush f;
      print_string "itt vagyok\n"; flush stdout;
      Join.go here;
      print_string "atjottem\n"; flush stdout;
      output_string f "Masodik sor"; flush f;
      close_out f;
   };;
```

Eredmény: a fájlba csak az `"Elso sor"` került bele. Sőt, az ágens a vándorlás után teljesen le is állt, még azt sem írta ki `"atjottem"`. Tehát egy megnyitott fájlhoz hozzáférést tartalmazó absztrakt hely vándoroltatása teljesen megzavarta a rendszert.

12.8. Név-szerver mérete

Kérdés: Hány bejegyzést bír el a név-szerver?

Program: ugyanazt a függvényt (négyzetre-emelés) regiszrálja be a név-szerverben, de mindig más néven (`1_nev`, `2_nev`, …). A regisztrációt a parancs-sori argumentumában megadott szám-szor (ha nincs paraméter, akkor egyszer) végzi el.

```
let regisztral nev =
   let f x = x*x in
   Ns.register nev f vartype;
   Printf.printf "A %s nev regisztralva lett." nev
;;

let rec hozzaad i =
   if i > 0 then
      (regisztral ((string_of_int i)^"_nev");
       hozzaad (i-1))
;;

let n = if (Array.length Sys.argv) > 1
         then int_of_string(Sys.argv.(1))
         else 1;;

hozzaad n;;
```

Eredmény: Különböző platformokon különböző eredményeket kaptam. Átlagosan azt tapasztaltam, hogy egy név-szerver 500 - 2000 bejegyzést tud tárolni.

A munka értékelése

Sikerült megalkotni a nyelv teljes és egységes leírását. Ez egyáltalán nem volt egyszerű feladat ugyanis a nyelvről – mostanáig – ilyen leírás nem létezett. A nyelv funkcionális alapját képező Objective Caml több verziójáról is létezik leírás, de ezek közül kevés foglalkozik érdemben a JoCaml által tartalmazott 1.07-es verzióval. Ráadásul a különféle leírások sem adtak választ minden kérdésre, így több kérdést is csak a fordító kódjának vizsgálatával vagy az adott problémát megvilágító teszt-program készítésével lehetett megválaszolni.

A nyelv általános elemzése és leírása után a JoCaml-t az osztott rendszerek jellemzésére általában is alkalmas szempontrendszer szerint értékeltem. Ezen értékelés alapján világossá váltak a nyelv előnyei és hátrányai is. Az értékelés alapján és azzal összhangban javaslatokat tettem a hiányolt tulajdonságokat megvalósító új nyelvi elemek bevezetésére és ezek egy lehetséges megvalósítására.

Bemutattam a nyelvi elemek – különösen a join kalkulus – elméleti hátterét.

A dolgozatban – ahol szükséges – több példaprogram is található, melyek érthetőbbé teszik a nyelv leírását. A függelékben is található számos, a nyelv érdekesebb tulajdonságait szemléltető program.

A dolgozat minden részénél – az érthetőség kedvéért – röviden kitértem az adott rész elméleti hátterére is.

Összegzésül elmondhatom, hogy minden, a diplomamunka-téma bejentőben megfogalmazott feladatot megoldottam.

Irodalomjegyzék

[1] Horváth Zoltán: 16. fejezet, ***A funkcionális programozási nyelvek elemei***, [47], megjelent: Nyékyné Gaizler Judit (szerk.): Programozási nyelvek, Kiskapu, 2003, ISBN: 963-9301-47-7; oldalak: 589-636.

[2] Csörnyei Zoltán, Nagy Sára: ***Funkcionális programnyelvek implementációja***, Informatika a felsőoktatásban '96 – Networkshop '96, Debrecen 1996, [7], megjelent a konferencia CD-s kiadványában (ISBN 693-0470-28-4).
Elérhető: http://www.iif.hu/rendezvenyek/networkshop/96/eloadas/04e07.pdf (2004. június 8.)

[3] Nyékyné Gaizler Judit (szerk.): ***Programozási nyelvek***, Kiskapu Könyvkiadó, 2003, [784], ISBN: 963-9301-46-9

[4] Jason Hickey: ***Introduction to the Objective Caml Programming Language***, 2002, [107], elérhető: http://files.metaprl.org/doc/ocaml-book.pdf (2004. június 8.)

[5] Emmanuel Chailloux, Pascal Manoury, Bruno Pagano: ***Developing Applications With Objective Caml***, O'Reilly, 2000, [744], ISBN 2-84177-121-0

[6] Xavier Leroy: ***The Objective Caml system release 3.07 documentation and user's manual***, INRIA, 2003, [308], elérhető: http://pauillac.inria.fr/ocaml/htmlman (2004. június 8.)

[7] Xavier Leroy: ***The Objective Caml system release 1.07 documentation and user's manual***, INRIA, 1997, [280], elérhető: http://pauillac.inria.fr/jocaml/ocaml-1.07-refman/htmlman (2004. június 8.)

[8] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, Alan Schmitt: ***The JoCaml language beta release documentation and user's manual***, INRIA, 2001, [45], elérhető: http://pauillac.inria.fr/jocaml (2004. június 8.)

[9] Andrew S. Tanenbaum, Maarten Van Steen: ***Distributed Systems: Principles and Paradigms***, Prentice Hall, 2001, [803], ISBN 0-13-088893-1

[10] Hegedűs Hajnalka, Horváth Zoltán: ***Distributed Computing Based on Clean Dynamics***, 6th International Conference on Applied Informatics, Eger 2004, [8].
Elérhető: http://aszt.inf.elte.hu/~fun_ver/2004/papers/icai2004_paper_heha.pdf (2004. június 8.)

[11] Fabrice Le Fessant, Luc Maranget: ***Compiling join-patterns***, INRIA, 1998, [20], elérhető: http://para.inria.fr/~maranget/papers/pat (2004. június 8.)