PÁRHUZAMOS FUNKCIONÁLIS PROGRAMOZÁS *

PARALLEL FUNCTIONAL PROGRAMMING

Zsók Viktória, zsv@inf.elte.hu
Horváth Zoltán, hz@inf.elte.hu
Tejfel Máté, matej@inf.elte.hu
Eötvös Loránd Tudományegyetem
Általános Számítástudományi Tanszék

Abstract

One of the most interesting research topics in the domain of functional programming is the problem of parallel, distributed programming and mobile code exchange. There are several trends for solving this problem: the introduction of new language constructs, parallel and distributed function evaluation, defining of parallel strategies, the use of dynamics and the use of TCP-IP communication protocol. Concurrent Clean, Distributed Haskell with Ports and JoCaml are functional languages, which contains several parallel language elements. Our aim is to present the parallel language primitives of these three languages using small examples. The language primitives are analysed from the point of view of the efficiency, applicability and expressiveness.

These languages offer various possibilities to express parallelism in functional programming, for this reason it is very important to study the parallel functional opportunities of the different platforms. The survey and the synthesis of these language constructs enable us to choose the right tools in case of a concrete problem in order to solve it easily and efficiently.

Összefoglaló

A funkcionális programozásban egyik legérdekesebb kutatási terület a párhuzamos és osztott programok írása. Többféle irányvonal határolható el: új absztrakt nyelvi elemek bevezetése, párhuzamos és elosztott függvénykiértékelés létrehozása, kiértékelési stratégiák módosítása, dinamikusan összeszerkesztett mobil kód, valamint TCP-IP kommunikációs protokoll használata. Célunk a Concurrent Clean és a Distributed Haskell with ports lusta kiértékelésű funkcionális nyelvek, valamint a JoCaml imperatív és objektum orientált elemeket is tartalmazó funkcionális nyelv párhuzamos nyelvi elemeinek példák segítségével történő bemutatása. A nyelvi elemeket a hatékonyság, az alkalmazhatóság, a kifejezőerő szempontjából vizsgáljuk.

A különböző fejlesztési környezetek más-más lehetőséget nyújtanak a párhuzamos funkcionális programozás számára, ezért számunkra fontos a különböző platformok párhuzamos funkcionális programozási lehetőségeinek elemzése. Ezek vizsgálata és szintézise lehetővé teszi, hogy bármilyen nagyobb feladat esetén a számunkra legmegfelelőbb eszközt válasszuk ahhoz, hogy ezt könnyebben és hatékonyabban tudjuk megoldani.

* Készült az OTKA T037742 sz. és a Grid 01548 sz. pályázat támogatásával.

PÁRHUZAMOS FUNKCIONÁLIS PROGRAMOZÁS [*]

PARALLEL FUNCTIONAL PROGRAMMING

Zsók Viktória, zsv@.inf.elte.hu
Horváth Zoltán, hz@.inf.elte.hu
Tejfel Máté, matej@.inf.elte.hu
Eötvös Loránd Tudományegyetem
Általános Számítástudományi Tanszék

1. Bevezetés

A függvények kompozíciója asszociatív, így a funkcionális nyelvű programok kiértékelése jól párhuzamosítható. A legfontosabb kérdés annak eldöntése, hogy mely részkifejezések kiértékelését célszerű párhuzamos vagy elosztott módon elvégezni. Számos eredmény született, több funkcionális programozási nyelv (Concurrent Clean, Erlang, Concurrent ML, JoCaml, Haskell) új párhuzamos nyelvi elemet épített be. Ezek közül a Concurrent Clean nyelvi elemeit, a JoCaml párhuzamos és osztott nyelvi elemeit, valamint a Distributed Haskell with ports párhuzamossági lehetőségeit vizsgáljuk. Célunk, hogy kisebb példák segítségével szemléltessük a nyelvi elemek kifejező erejét, hatékonyságát és alkalmazhatóságát.

2. Concurrent Clean

A Concurrent Clean a nijmegeni egyetemen fejlesztett párhuzamos funkcionális programozási nyelv. Elsőként a nyelvben található legrégebbi párhuzamos nyelvi elemet szeretnénk bemutatni, az *annotáció*kat. Az annotációk segítségével meghatározhatjuk, hogy a fordítóprogram egy függvény vagy egy kifejezés mely részeit értékelje ki párhuzamosan. Három típusú annotációt különböztetünk meg [1]:

- `I`: összefésüléses annotáció, mely lehetővé teszi két különböző kifejezés kiértékelését egyetlen egy processzoron.
- `P`: párhuzamos annotáció, mely egy kifejezést egy vagy több processzoron értékel ki.
- `P at procid`: párhuzamos kiértékelést végez megadott processzoron.

A következő függvény annotációk segítségével számítja ki a Fibonacci szám értékét:

```
Nfib :: Int -> Int
Nfib n     | n < 2  =       1
                    =       {| P |}Nfib (n - 1) +  Nfib (n - 2) + 1
```

[*] Készült az OTKA T037742 sz. és a Grid 01548 sz. pályázat támogatásával.

A fenti példában a `P` annotáció hatására egy új kiértékelési szál indul el, azaz `Nfib (n - 1)` kiértékelése az `Nfib (n - 2)` kiértékelésével párhuzamosan történik.

A Clean lehetővé teszi, hogy irányítsuk a kifejezések kiértékelésének módját és mélységét, így a párhuzamos kiértékelését is. Ezt az annotáción alapuló stratégiákkal tehetjük meg. A két elemi stratégia (`'seq'` – a szekvenciális kiértékelési stratégia, `'par'` – a párhuzamos kiértékelési stratégia) segítségével összetett stratégiákat építhetünk fel [2]. Például az összetett `parlist` stratégia segítségével párhuzamos `map` függvényt írhatunk:

```
parmap :: (Strategy b) (a -> b) [a] -> [b]
parmap s f x = map f x 'using' parlist s
```

Egy másik párhuzamos programozási lehetőség a Concurrent Cleanben a csatornák használata. Az üzenetküldés kétféle típusát különböztetünk meg: két különálló program közötti kommunikációt, illetve egy programon belüli párhuzamos szálak üzenetváltását [3].

Az első esetben két absztrakt csatorna adattípust használunk az `:: SChannel a` és az `:: RChannel a` típust. Ezek segítségével küldhetünk, illetve fogadhatunk üzeneteket. Az üzeneteket a fogadó csatorna listában tárolja, érkezési sorrendben. Egy csatornát a `createRChannel`-el hozunk létre (amennyiben csak helyi használatra szeretnénk egy csatornát létrehozni, akkor a `newChannel`-el), `findSChannel`-el, illetve `lookupSChannel`-el keressük meg. A folyamatok egymással a `send`, a `receive` és az `available` utasításokkal kommunikálnak. Az alábbi program a termelő-fogyasztó feladat megvalósítása csatornák segítségével:

```
// 1. a termelő program
Start :: *World -> *World
Start w
   # (sc, w) = findSChannel „Consumer” w
   = produce sc 10 1 sc w

produce :: (SChannel Int) Int Int *World -> *World
produce sc i n w
   | i == 10   = w
   | otherwise = produce sc (i - 1) (n + 1) (send sc n w)

// 2. a fogyaszó program
Start :: *World -> (Int, *World)
Start w
   # (maybe_rc, w) = createRChannel „Consumer” w
   = consumer maybe_rc w
where
   consumer Nothing w = abort „already exists”
   consumer (Just rc) w = consume rc 10 0 w

consume :: (RChannel Int) Int Int *World -> (Int, *World)
consume rc i r w
   # (n, rc, w) = receive rc w
   | i == 0    = (r + n, w)
   | otherwise = consume rc (i - 1) (r + n) w
```

A második esetben lusta kiértékelésű szálakat hozhatunk létre a `newThread` függvénnyel ugyanazon a processzoron. A szálak közötti kommunikálást ugyancsak `send`, `receive` és az `available` típusú utasításokkal valósíthatjuk meg.

Példaprogramként az előbbi termelő-fogyasztó feladat implementálását mutatjuk be, ez alkalommal a termelő és a fogyasztó két különálló szálnak felel meg. A két szál törzse ugyanazon programhoz tartozik, de egy szálat elindíthatunk egy másik processzoron is. Ezt a `newThreadAt pid` segítségével tehetjük meg:

```
module ProdCons
import StdEnv, StdParallel, StdThread, StdChannel

Start w = startProcessorsW' myStart w

// létrehoz egy csatornát a pid azonosítóval rendelkező processzoron
// a csatorna a két külön szálon futó produce, illetve a consume
// függvényeket köti össze
myStart :: (Set ProcId) *World -> (Int, *World)
myStart pids w
   # (sc, rc, w) = newChannel w
     w           = newThreadAt pid (produce sc 10 1) w
   = newThread' (consume rc 10 0) w
where
   pid = pickFromSet pids 0

// a termelő, produce függvény
produce :: (SChannel Int) Int Int *env -> *env | ThreadEnv env
produce sc i n env
   | i == 0    = env
   | otherwise = produce sc (i - 1) (n + 2) (send' n sc env)

// a fogyasztó, consume függvény
consume :: (RChannel Int) Int Int *env -> (Int, *env) | ThreadEnv env
consume rc i r env
   # (n, rc, env) = receive' rc env
   | i == 1    = (r + n, env)
   | otherwise = consume rc (i - 1) (r + n) env
```

Csatornákat használ a TCP-IP protokoll alapú párhuzamos funkcionális programozás is [4]. A hálózat bármely gépét az ip címe szerint azonosíthatjuk, majd a géphez rendelt csatornákon üzeneteket küldhetünk a vele összekötött gépeknek. Például, miután létrehoztunk egy kliens-szerver kapcsolatot két gép között, akkor ezek a csatornákon keresztül a következőképpen küldhetnek egymásnak üzeneteket:

```
// a kliens elküldi a „hello server" üzenetet a TCP_SChannel-en keresztül
clientSend :: TCP_SChannel *World -> (TCP_SChannel, *World)
clientSend sChannel world=send (toByteSeq „hello server") sChannel world

// a szerver TCP_SChannel-en fogadja az üzenetet, amennyiben ez megfelel
// az elvárt üzenetnek
serverReceive :: String TCP_RChannel *World -> (TCP_RChannel, *World)
serverReceive expectedMessage rChannel world
   # (message, rChannel, world)        = receive rChannel world
   | toString message <> expectedMessage = abort „wrong message"
   | otherwise                           = (rChannel, world)
```

A Clean nyelv szigorúan statikus típusrendszerét egészíti ki az új `Dynamic` típus [5], amelynek segítségével tetszőleges típusú kifejezést futási időben becsomagolhatunk és kicsomagolhatunk. A `sendDynamic` és `receiveDynamic` függvényekkel lehetőségünk van arra is, hogy a Clean programok konstansokat és mobil kódrészleteket küldjenek és

fogadjanak. A `writeDynamic` és `readDynamic` műveletek alkalmasak arra, hogy fájlba írjunk ki, fájlból olvassunk be dinamikus kifejezéseket. Az alábbiakban egy példán azt mutatjuk be, hogy egy Clean program hogyan épít fel egy fát, majd elment egy fájlba; egy másik program elkészít egy olyan függvényt, amely megszámolja egy fa leveleit és a függvényt kiírja egy fájlba, majd végül egy harmadik program beolvassa a fát a fájlból és alkalmazza rá az ugyancsak fájlból beolvasott függvényt, majd kiírja az eredményt.

Az első program létrehozza a `tree2` fát, becsomagolja a `dynamic`-al, majd kiírja egy fájlba a `writeDynamic`-al.

```
module v
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
   #! (ok, world) = writeDynamic (p +++ "value")
                        DynamicDefaultOptions dt world
   | not ok = abort "could not write dynamic"
   = (dt, world)
   where    dt = dynamic (Node 99 tree2 tree2)
         tree2 = (Node 2 (Node 1 Leaf Leaf) Leaf)
             p = "C:\\Clean 2.0\\Examples\\Dynamic\\"

// count_leafs függvény megszámolja egy fa leveleit és a program kiírja
// a függvényt egy fájlba
module f
import StdDynamic, StdEnv
:: Tree b = Node b (Tree b) (Tree b) | Leaf
Start world
   #! (ok,world) = writeDynamic (p +++ "function")
                         DynamicDefaultOptions dt world
   | not ok = abort "could not write dynamic"
   = (dt,world)
   where    dt = dynamic count_leafs
             p = "C:\\Clean 2.0\\Examples\\Dynamic\\"
```

Az `apply` alkalmazza a fájlból beolvasott függvényt az ugyancsak fájlból beolvasott fára, majd kiírja az eredményt mintaillesztéssel ellenőrzi, hogy a beolvasott függvény és a beolvasott érték típusa megegyezik-e az elvárt típusokkal.

```
module apply
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
   # (ok,f,world)  = readDynamic (p +++ "function") world
   | not ok            = abort " could not read"
   # (ok,v,world) = readDynamic (p +++ "value") world
   | not ok            = abort " could not read"
   # result = apply f v;
   = (result, world);
where
     apply (f :: (Tree Int) -> Real) (v :: (Tree Int))= f v
     apply _ _                                        = abort "unmatched"
     p = "C:\\Clean 2.0\\Examples\\Dynamic\\"
```

3. **Distributed Haskell with ports**

A Distributed Haskell könyvtár a Haskell funkcionális programozási nyelv egy kiterjesztése, melynek használatával osztott Haskell programokat írhatunk. A kiterjesztés lehetőséget ad arra, hogy a programok között egyirányú, pont-pont kapcsolatot létesítsünk portok segítségével. Nézzük meg, hogy milyen alapvető nyelvi elemek állnak ehhez rendelkezésünkre. A könnyebb megértés céljából tekintsünk egy egyszerű példaalkalmazást, melyben egy szerver és két (azonos kódú) kliens programunk van. A kliensek egy paraméterként megadott számot küldenek el a szervernek, a szerver visszaküldi a klienseknek az elküldött két szám összegét, azok pedig kiírják az összeget a képernyőre.

Ahhoz, hogy a kliensek üzenetet tudjanak küldeni a szervernek, először a szerveroldalon létre kell hoznunk egy portot. Ezt a `newport` függvénnyel tehetjük meg, amellyel bármilyen olyan típushoz, amely a Haskell Readable és Showable osztályába is beletartozik létrehozhatunk portot, amelyen az adott típusú adat küldhető. A portot csak az a program (és ha esetleg a programon belül több szál van, akkor csak az a szál) olvashatja, amely létrehozta azt. A kliensekkel valamilyen módon tudatnunk kell, hogy melyik az a port, amelyre a szervernek üzenetet küldhetnek. Ehhez a szerveroldalon a `registerPort` függvénnyel regisztrálnunk kell a szerverportot valamilyen (a kliensek által ismert) néven.

```
-- port létrehozás és regisztrálás
serverPort <- newPort
registerPort serverPort "ServerPort"
```

A regisztrálás után a kliensoldalon a `lookupPort` függvénnyel lekérdezhetjük az adott névhez tartozó portot. Ehhez a néven kívül szükségünk van a szervert futtató gép IP címére is (a példánkban a szerveroldali IP címet a kliensek parancssori argumentumként kapják meg). Ezek után a klienseknek már elég információ áll a rendelkezésükre ahhoz, hogy elküldjék az argumentumként kapott számot a szervernek. Mivel az üzenetküldés csak egyirányú, a klienseknél is létre kell hoznunk egy-egy portot, amelyre a szerver elküldheti az eredményül kapott értéket. Ugyanakkor tudatnunk kell a szerverrel, hogy melyek ezek a portok. Erre a legegyszerűbb módszer, ha a kliensek a paraméterként kapott számokkal együtt az általuk létrehozott portok leírását is elküldik a szervernek. A portokra való adatküldésre a `writePort` függvény szolgál. A függvénynek a küldeni kívánt adatot illetve azt a portot kell megadnunk paraméterül, amelyre az adatot küldeni szeretnénk.

```
-- adatok elküldése a szerver számára
[host,numstr] <- getArgs
serverPort <- lookupPort host "ServerPort"
inPort <- newPort
writePort serverPort (num,inPort)
```

A szerver az előzőekben leírt portregisztrálás után a `readPort` függvény meghívásával olvasni próbál egy számból és egy portleírásból álló adatpárt a szerverportról. A `readPort` függvény akkor ad vissza adatot, ha valaki küld valamit (a `writePort` függvénnyel) az adott portra, amíg ez meg nem történik, a program blokkolódik a függvényhívásnál. Miután a szerver sikeresen olvassa az egyik kliens által küldött értékeket, még egyszer meghívja ugyanazt a függvényt, hogy a másik kliens adatait is megkapja.

```
-- a kliensoldali adatok fogadása
```

```
(num1,outPort1) <- readPort serverPort
(num2,outPort2) <- readPort serverPort
```

Ha ez is sikeresen megtörténik, akkor a már ismert `writePort` függvénnyel elküldi a kapott számok összegét a számokkal egyidejűleg kapott portokra.

```
-- az összeg visszaküldése
writePort outPort1 (num1 + num2)
writePort outPort2 (num1 + num2)
```

A kliensek a fentiekben leírt adatküldés után megpróbálnak egy `Int` értéket olvasni a `readPort` függvénnyel az általuk létrehozott portról, majd miután ez sikerül kiírják a kapott értéket a képernyőre.

```
-- az eredmény fogadása és kiírása
sum <- readPort inPort
putStrLn (show sum)
```

Ezek után nézzük meg a teljes programokat:

A szerver program:

```
import IO
import Port
import System

main = do
     serverPort <- newPort
     registerPort serverPort "ServerPort"
     (num1,outPort1) <- readPort serverPort
     (num2,outPort2) <- readPort serverPort
     writePort outPort1 (num1 + num2)
     writePort outPort2 (num1 + num2)
```

A kliens program:

```
import IO
import Port
import System

main = do
     [host,numstr] <- getArgs
     serverPort <- lookupPort host "ServerPort"
     inPort <- newPort
     client inPort serverPort (read numstr)

client :: Port Int -> Port (Int,(Port Int)) -> Int -> IO ()
client inPort serverPort num = do
     writePort serverPort (num,inPort)
     sum <- readPort inPort
     putStrLn (show sum)
```

4. **A JoCaml párhuzamos és osztott nyelvi elemei**

A JoCaml nem tisztán funkcionális nyelv, imperatív és objektum orientált elemeket is tartalmaz. A JoCaml nyelvben a mobil kód megfelelője az ágens. A nyelv tervezésének három fő szempontja van: eszközök biztosítása ágensek egyszerű létrehozására, összetett elosztott számítások kifejezhetősége, az absztrakt nyelvi eszközök pontos szemantikai leírhatósága. A legfontosabb két absztrakt nyelvi elem: a csatorna és az absztrakt hely fogalma. [7]. Szinkron és aszinkron csatornát a `let def` segítségével hozhatunk létre. Aszinkron csatornán folyamatok segítségével küldhetünk adatokat.

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

A `spawn` konkurens folyamatokat indít. Az üzenetküldést az alábbi példán szemléltetjük, mely párhuzamos folyamatokhoz csatolt csatornák értékét írja ki, azaz az 1, 2, 3 egész számokat tetszőleges sorrendben:

```
# spawn{
#   let x = 1 in
#   {let y = x + 1 in echo y | echo (y + 1)} | echo x}
# ;;
```

Szinkronizációs mintákkal írhatjuk le a konkurens programozás összetett vezérlési és adatkezelési struktúráit. Csatornahalmazokra tartozó kommunikációs események között fennálló konfliktust, illetve szinkronizációt fejezhetünk ki velük. Szinkronizációs minták mindig egyetlen `let def` kifejezésben egyidejűleg definiált csatornákra vonatkoznak. Mintákat a „|” párhuzamos kompozíció operátorral írhatunk. Csatornarészhalmazokra vonatkozó, több alternatív mintát kapcsolhatunk össze az `or` művelettel. Konkurens környezetben használt számlálóra példa a `count` művelet, mely a kölcsönös kizárást biztosít az `n` változó elérésére:

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

JoCaml-ban elosztott programokat is írhatunk. A folyamatok vándorolhatnak egyik gépről a másikra. Kezdetben a különböző gépeken futó folyamatok között nincs kapcsolat, a partnerek névszolgáltató segítségével találnak egymásra.

Az absztrakt hely fogalma magában foglalja az ágens kódjának és aktuális fizikai helyének együttesét. Ily módon a kód mozgása során a JoCaml nyelv absztrakt eszközöket biztosít a kompozíció, a kommunikáció és a belső állapot helyfüggetlen kezelésére és távoli szolgáltatások igénybevételére. Ágensek komponensei együtt mozognak az őket tartalmazó szülő ágenssel, a kommunikációs csatorna fennmarad helyüket időközben változtató ágensek között is, és az ágensek abból az állapotból folytatják működésüket, ahol a helyváltoztatás előtt voltak.

A következő példában az egyik folyamat bejegyzi az `f` közös erőforrást `square` néven, a másik pedig megkeresi és felhasználja:

```
# 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;}
# ;;
```

Absztrakt helyeket a `let loc <azonosító> do {}` konstrukcióval hozhatunk létre. Egy fizikai helyet, mint egy absztrakt hely aktuális megfelelőjét, azaz mint az ágens aktuális fizikai helyét azonosíthatjuk. Erre használjuk majd a `here` üres ágenst:

```
# let loc here do {}
# ;;
#
# Ns.register „here” here vartype;
# Join.server ()
```

A `mobile` ágens lekérdezi a névszolgáltatótól a `here` ágens adatait, majd átvándorol a `here` aktuális fizikai helyére és ennek mindenkori helyén, alágenseként végzi el a számításokat.

```
# let loc mobile
#  do {
#    let here = Ns.loopkup „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;
#  }
```

Letölthető ágenseket is definiálhatunk oly módon, hogy az ágensnek paraméterként adjuk meg, hogy híváskor hova vándoroljon. Adatvezérelt migrációt is megvalósíthatunk.

5. **Összefoglalás**

Olyan nyelvi elemeket mutattunk be Clean, Haskell, illetve JoCaml példákon keresztül, amelyek segítségével osztott, párhuzamos programokat készíthetünk funkcionális stílusban. A nyelvek kifejező ereje különböző, az egyes elemek absztrakciós szintje eltérő. Bemutattuk a legalacsonyabb absztrakciós szintet képviselő annotációkat, az azokra épülő stratégiákat, az explicit üzenetküldés eszközeit, a csatornákat, a mobil funkcionális kód megfelelőit, a dynamic-ot és az ágenseket. Ezek hatékonyságban, alkalmazhatóságban, implementációjukban is nagyon különböznek egymástól, a megoldandó feladatnak megfelelően választhatunk a bő kínálatból és hozhatunk létre párhuzamos és elosztott alkalmazásokat.

6. Irodalomjegyzék

1. Kesseler, M.H.G.: *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*, PhD Thesis, Catholic University of Nijmegen, 1996.

2. Horváth Z., Zsók V., Serrarens P., Plasmeijer R.: Parallel Elementwise Processable Functions in Concurrent Clean, to appear in *Proceedings of the 5th International Conference on Applied Informatics*, Eger, Hungary, January 2001 and selected for *Computers & Mathematics with Applications*, Elsevier.

3. Serrarens, P.R.: Explicit Message Passing for Concurrent Clean, In: Hammond, K. et al., eds., *Implementation of Functional Languages, 10th International Workshop, IFL'98*, London, UK, September 1998, LNCS, Vol. 1595, pp. 229-245, Springer-Verlag, 1999.

4. Achten, P., Wierich, M.: *A Tutorial to the Clean Object I/O Library*, University of Nijmegen, 2000.

5. Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Language Report*, University of Nijmegen, 2001.

6. http://www-i2.informatik.rwth-aachen.de/hutch/distributedHaskell

7. Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: *The JoCaml language beta release*, Documentation and user's manual, INRIA, 2001.