Az Alice nyelvről

Jövők, csomagok és kóddisztribúció

Szoboszlay Dániel

Hanák Péter kiegészítéseivel

2004. október 18.

1 Jövők

A konkurens és lusta kiértékelést az Alice a jövők segítségével valósítja meg.

Jövő (*Future*): egy még ki nem számított érték helyett szereplő helyfoglaló.

A jövő típusai: a jövőknek négy típusa van.

- Lusta jövő (*Lazy future*): egy lusta kiértékelésű számítás eredménye.
- Konkurens jövő (*Concurrent future*): egy konkurens számítás eredménye.
- Ígért jövő, ígéret (*Promised future*): olyan érték, amelyet később fog meghatározni a program.
- Meghiúsult jövő (*Failed future*): olyan jövő, amelynek eredménye végül egy kivételcsomag lett.

A lusta és a konkurens jövő létrehozásának pillanatától kezdve ismert az a függvény, amely elő fogja állítani az értéküket – csak éppen ennek a függvénynek még nem fejeződött be a kiértékelése.

Ezzel szemben az ígért jövő létrehozásakor még nem ismert a leendő érték meghatározásának módja. Ezt a függvényt a program később adja meg.

1.1 Lusta kiértékelés

Egy függvény alkalmazásának lusta kiértékelése: a `lazy` kulcsszóval írjuk elő.

```
- val x = lazy 3 + 2;
val x : int = _lazy
- x + 1;
val it : int = 6
```

Az Alice a függvényt egészen addig nem értékeli ki, amíg az értékére szükség nem lesz. (Ez egy informális meghatározás, később pontosabban is definiáljuk.) Mivel a „lustaság” jelen formájában csak egyetlen alkalomra szól, ha a függvény rekurzív, akkor a többi rekurziós szint már nem lesz lusta!

Lusta kiértékelésű függvény létrehozása: két egyenértékű szintaxis létezik:

```
fun f x = lazy x + 1
fun lazy f x = x + 1
```

Az így definiált függvényt az Alice mindig lustán értékeli ki.

```
- val l = lazy map (fn x => x + 1) [1, 2, 3];
val l : int list = _lazy
- hd l;
val it : int = 2
- l;
val it : int list = [2, 3, 4]
- fun lazy mapz f [] = nil
        | mapz f (x::xs) = f x :: mapz f xs;
val mapz : ('a -> 'b) -> 'a list -> 'b list = _fn
- val l = mapz (fn x => x + 1) [1, 2, 3];
val l: int list = _lazy
- hd l;
val it : int = 2
- l;
val it : int list = 2 :: _lazy
```

A `map` függvényt a `lazy` kulcsszóval meghívva lustává tettük a kiértékelését. De a lista fejének lekérdezésénél szükség lett a számítás eredményére, ezért az végrehajtódott, méghozzá *az egész listára*.

Ezzel szemben az alapértelmezés szerint lustának definiált `mapz` függvény – amely a lustaságán kívül mindenben megegyezik a jól ismert `map`-pel – csak a listának azon részét dolgozta fel, amelyre ténylegesen szükség is volt – azaz csupán az első elemét.

Mindez egyben azt is jelenti, hogy lusta függvényekkel egyszerűen generálhatók végtelen sorozatok nyílt végű listákba. Ha a `zip` függvény lusta verzióját is elkészítjük, a Fibonacci számok listája a következő módon állítható elő:

```
fun lazy zipz (x::xs, y::ys) = (x, y) :: zipz (xy, ys)
  |       zipz _              = nil
val rec fibs =
      1 :: 1 :: (lazy mapz op+ (zipz (fibs, tl fibs)))
```

(Ebben az esetben a `lazy` kulcsszó explicit használatára a rekurzív értékdefiníció miatt volt szükség, hiszen egyébként mind `mapz`, mind `zipz` lusta kiértékelésű.)

1.2 Konkurens jövő

Egy függvény alkalmazásának konkurens kiértékelésére való a `spawn` kulcsszó:

```
- val x = spawn 3 + 2;
val x : int = _future
- x;
val it : int = 5
```

Konkurens kiértékelésű függvény létrehozására két egyenértékű szintaxis létezik:

```
fun f x = spawn x + 1
fun spawn f x = x + 1
```

Egy konkurens számítás eredményét használni kívánó függvény kiértékelése mindaddig blokkolódik, amíg az érték rendelkezésre nem áll.

A konkurencia megvalósítása könnyű súlyú: a rendszer képes szálak százezreit is kezelni.

1.3 Ígéretek

Az ígéret egy olyan adattípus, amely egy jövőt tartalmaz. A jövő értékét használó függvények blokkolódnak, amíg az rendelkezésre nem áll. Az ígéret legfeljebb egyszer explicit módon vagy *beváltható* egy adott értékkel, vagy *meghiúsulhat* valamilyen kivételcsomag révén.

Ígéretek létrehozására a `Promise` struktúra ad módot, szignatúrája a következő dián látható.

A `Promise` kivételt egy már beváltott vagy meghiúsult ígéret újbóli beváltása, ill. meghiúsítása váltja ki.

```
signature PROMISE =
sig
    type ’a promise
    type ’a t = ’a promise
    exception Promise
    val promise : unit -> ’a promise
         (* Új ígéret létrehozása *)
    val future : ’a promise -> ’a
         (* Az ígéretbe ágyazott jövő *)
    val fulfill : ’a promise * ’a -> unit
         (* Az ígéret beváltása *)
    val fail : ’a promise * exn -> unit
         (* Az ígéret meghiúsítása *)
end
```

Az ígéretek egy lehetséges használata az adatszerkezetek top-down felépítése:

```
fun append (l1,l2) =
    let
        fun iter (nil,   p) = fulfill (p, l2)
          | iter (x::xs, p) =
              let
                  val p’ = promise ()
              in
                  fulfill (p, x::future p’);
                  iter (xs, p’)
              end
        val p = promise ()
    in
        iter (l1, p); future p
    end
```

1.4 Meghiúsult jövők

Egy jövő kiszámítása közben esetlegesen hibák, kivételek léphetnek föl. Ilyenkor a jövőből meghiúsult jövő lesz. A meghiúsult jövő nem blokkolja az értékét használó függvényeket, hanem ezeken a helyeken a meghiúsulást eredményező kivétel újra kiváltódik.

Ígéreteket a `fail` függvénnyel explicit módon lehet meghiúsítani.

```
- val p : int promise = promise ();
- fail (p, Domain);
- future p;
val it : int = _failed|Domain|
- 1 + future p;
uncaught exception Domain
- val x : int = lazy raise Domain;
val x: int = _lazy
- x + 1;
uncaught exception Domain
```

1.5 Jövők használata (*requesting futures*)

A fenti példákban is látszott már, hogy egy jövőre való hivatkozás nem mindig minősül a jövő használatának olyan értelemben, hogy a művelet nem blokkolódott, a lusta kifejezés nem értékelődött, ki és a meghiúsult jövő kivétele nem váltódott ki.

A jövőt akkor használjuk, ha szigorú (*strict*) műveletek argumentumaként szerepel. Szigorú műveletek a következők:

- a mintaillesztés a vizsgált értékre;
- a függvényalkalmazás a függvényváltozóra;
- kivétel kiváltása a kivételváltozóra;
- azon primitív műveletek, amelyeknek hozzá kell férniük egy argumentumuk értékhez, a megfelelő argumentum értékére (pl. `op+`, `op=`, pickling);
- a funktor alkalmazása a funktorértékre;
- a kicsomagolás a csomag szignatúrájára és a célszignatúrára.

1.6 Jövőkhöz kapcsolódó egyéb struktúrák

Ref: a `ref` típust és műveleteit definiálja. Ez egy megváltoztatható (frissíthető) értékre vonatkozó referencia.

- `datatype 'a ref = ref of 'a`: a konstruktorfüggvény.
- `! : 'a ref -> 'a`: a hivatkozott értéket adja vissza.
- `:= : 'a ref * 'a -> unit`: a hivatkozott értéket állítja be.
- `exchange : 'a ref * 'a -> 'a`: egyetlen oszthatatlan lépésben átállítja a referenciát és visszaadja az eredeti értékét. Konkurens szálak szinkronizálására használható.

Future: a jövők létrehozása és nyomonkövetése. Explicit szinkronizációs és állapotlekérdező függvényeket tartalmaz.

Thread: szálak kezelésére (indítás, leállítás, felfüggesztés, várakozás) nyújt módot. A `spawn` kulcsszó is egy itt definiált `thread` típusú szálat indít.

Lock: monitor jellegű szinkronizáció megvalósítása.

- `type lock`: a monitor típusa.
- `lock : unit -> lock`: egy új monitor létrehozása.
- `sync : lock -> ('a -> 'b) -> ('a -> 'b)`: visszaad egy függvényt, amely pontosan azt számítja ki, mint a paraméterül adott, de a monitor része. Az egy monitorba felvett függvények közül egyszerre csak egynek a kiértékelése folyhat. A zárolás *nem* reentráns.

1.7 A konkurencia buktatói

A konkurencia miatt a nem tisztán funkcionális minták (pl. `ref` kifejezések) illesztése nem várt viselkedéshez vezethet. Egy SML alatt kimerítő mintaillesztés Alice alatt nem feltétlenül lesz az, mivel a mintaillesztés nem atomi művelet!

A lusta kiértékelésű kifejezésekben szereplő mellékhatásos függvények is gondot okozhatnak, hiszen ezek a hatások is „késleltetve vannak” a teljes kifejezés kiértékeléséig.

```
- fun f (ref false) = 1 | f (ref true)  = 2
val rec r = ref (lazy (r := false; true))
- f r
```

2 Csomagok

Az Alice-ben lehetőség van folyamatok (távoli gépek) közötti adat- és kódátvitelre. Ehhez viszont elengedhetetlen a kód futásidejű mozgatása, ami egyben futásidejű típusellenőrzést is jelent. A csomagok (*packages*) épp ezt teszik lehetővé.

Egy csomag létrehozása:

```
val változó = pack struktúra :> szignatúra
```

Egy csomag kicsomagolása:

```
structure struktúra = unpack csomag : szignatúra
```

Példák:

```
val p = pack Word8 :> WORD
structure Word' = unpack p : WORD
Word'.toInt (Word'.+ (Word'.fromInt 4, Word'.fromInt 2))

val p = pack struct fun f l = (hd l, hd (tl l)) end
                :> sig    val f : 'a list -> 'a * 'a  end
structure Intlist2pair = unpack p
                : sig val f : int list -> int * int end
```

Ha a kicsomagolásnál nem lehet a csomag szignatúráját illeszteni a megadott célszignatúrára, az Alice `Mismatch` kivételt generál.

Fontos megjegyezni, hogy az áttetsző szignatúrakötés miatt egy struktúra típusai nem feltétlen lesznek kompatibilisek a struktúra be-, majd kicsomagolásával kapott másik struktúrával. Például a következő kifejezés nem fordul le:

```
Word’.toInt (Word8.fromInt 12)
```

A probléma megoldható, ha a csomagot a következőképpen csomagoljuk ki:

```
structure Word8’ =
    unpack p : (WORD where type word = Word8.t)
Word8’.toInt (Word8.fromInt 12)
```

Ez a módszer „ismeretlen” típusokra is használható. Feltéve, hogy `S1` és `S2` érvényes szignatúrakifejezések, viszont se `t_s1`, se `t_s2` típusa nem ismert, csupán az, hogy egyenlők, az alábbi kódrészlet helyes:

```
structure X1 = unpack p1 :  S1
structure X2 = unpack p2 : (S2 where type t_s2 = X1.t_s1)
```

3 Szerializáció

Pickle: egy érték szerializált, fájlba írt reprezentációja. Az Alice a fentiekben bemutatott csomagokból képes elő állítani pickle-ket. A típushelyességet a csomagok megvalósítása garantálja. Egy érték szerializálásánál az általa hivatkozott objektumok tranzitív lezártja íródik a fájlba.

Az értékek viselkedése szerializációnál: egyes esetekben a szerializáció nem triviális. Bizonyos típusú értékek eltérő mód viselkednek a szerializáció során.

- A funkcionális (*functional*) értékek – amelyek nem tartalmaznak megváltoztatható objektumokat – szerializálása problémamentes. A visszatöltött érték az eredetileg elmentettől megkülönböztethetetlen objektum lesz.
- Az állapotos (*stateful*) értékek – amelyek viszont megváltoztatható objektumokat tartalmaznak, mint pl. `'a ref` – szerializálása szintén lehetséges. A funkcionális értékekkel szemben visszatöltéskor az eredeti, megváltoztatható objektumnak csak egy, a szerializálás pillanatában érvényes másolatával rendelkeznek. Egy pickle-n belül az állapotos értékek közös hivatkozásai a visszatöltés után is közösek maradnak.

- Helyhezkötött (*sited*) erőforrást tartalmazó értékek szerializációja nem lehetséges. Erőforrásnak számít minden olyan objektum, amely nem értelmezhető azon folyamaton kívül, amelyben létrehozták (pl. megnyitott fájlok, szálak.) Továbbá helyhezkötöttnek számítanak az erőforrásokat létrehozó függvények is.

  A tranzitív lezárás miatt a helyhezkötött értékek könnyen meghiúsíthatják egy olyan csomag szerializálását is, amely látszólag nem tartalmaz efféle változókat. Az ilyen hibákat a rendszer egy `Sited` paraméterű `IO.io` kivétellel jelzi.

- *Jövők* nem szerializálhatók. Ellenben a szerializáció a jövő használatának minősül, így meg kell várni annak végleges kiértékelődését.

Megvalósítás: a `Pickle` struktúra segítségével. Ennek fontosabb elemei:

- `val extension : string`: az alapértelmezett kiterjesztés a szerializált fájlokhoz.
- `val save : string * package -> unit`: egy csomag szerializálása.
- `val load : string -> package`: egy fájl beolvasása. Hibás formátumú fájl esetén egy `Corrupt` paraméterű `IO.io` kivétel váltódik ki.

4 Elosztott programozás

Az Alice elosztott programozási modelljében több folyamat kommunikál egymással pickle-k segítségével. A folyamatokat az Alice-ben szokás inkább helyeknek (*sites*) nevezni. Az adattovábbítás kommunikációs portok segítségével történik, a szükséges függvényeket pedig a `Remote` struktúra tartalmazza.

4.1 Jegyek

Egy hely explicit módon felajánlhat egy struktúrát a többiek számára:

```
offer : package -> ticket
```

A függvény hatására az Alice megnyit egy TCP-portot, amelyen egy HTTP szerver lesz elérhető. A megadott struktúrát szerializálja a rendszer, és az eredményt egy dokumentumként elérhetővé teszi a szerveren. A visszaadott jegy (*ticket*) egy `string`, amely az ezen dokumentum eléréséhez szükséges URL-t tartalmazza. Ennek ismeretében a többi hely letöltheti a csomagot:

```
take : ticket -> package
```

A jegy eljuttatásának módját a kliensekhez a felhasználónak kell kitalálnia. (Viszonylag egyszerű megoldás egy, a többi hely számára is ismert webcímen szöveges fájlként közzétenni.)

4.2 Proxyk

Bármely függvényből létrehozható egy proxy a `proxy` függvény segítségével:

```
proxy : ('a -> 'b) -> ('a -> 'b)
```

A visszaadott függvény pontosan ugyanazt az értéket fogja kiszámítani, mint az eredeti, csupán két különbség lesz közöttük:

1. A proxy mindig szerializálható, akkor is, ha az eredeti függvény nem volt az.
2. Az `f` egy `f' x` proxy függvényének kiértékelési lépései a következők:
   - Létrejön `x'`, az `x` egy szerializált klónja.
   - Az `x'`-t a rendszer átküldi arra a helyre, ahol az `f'` proxyt létrehozták.
   - Ez a hely, az `f'` otthona, kiszámítja az `y = f x'`-t. (`y` lehet egy kivétel is.)
   - Létrejön `y'`, az `y` egy szerializált klónja.
   - Az `y'`-t a rendszer átküldi az eredeti helyre, oda, ahol az `f'`-t meghívták.

4.2.1 A proxyk néhány fontos tulajdonsága

- A proxy értékének kiszámítása az otthoni helyén mindig konkurensen történik.
- Egy proxy függvénynek mind az argumentumát, mind az eredményét szerializálni kell, így egyik sem tartalmazhat helyhezkötött értéket.
- A paraméter klónozása miatt meg kell várni a benne szereplő jövők kiértékelését. Ha egy jövő meghiúsul, akkor kivétel váltódik ki – és ez lesz a proxy eredménye is.
- Egy proxy kiértékelése a hagyományos függvényekkel ellentétben több hibalehetőséget rejt magában, ezért többféle kivételt is okozhat. Közülük néhány fontosabb:

A hiba oka	*Kivétel*
Az argumentum helyhez kötött értéket tartalmaz.	`Proxy(SitedArgument)`
Az eredmény helyhez kötött értéket vagy olyan meghiúsult jövőt tartalmaz, amelyben meghiúsult jövőt tartalmazó kivétel szerepel.	`Proxy(SitedResult)`
A proxyt létrehozó folyamat terminált.	`Proxy(Ticket)`

5 Egy összetett példa: számkitalálós játék

A játék kliens–szerver felépítésű. A szerver gondol egy számra, a kliensek pedig egymással versengve megpróbálják kitalálni.

A klienseknek kétféle lehetőségük van: feltehetnek egy eldöntendő kérdést, illetve rákérdezhetnek a megoldásra. E célra a szerver két függvényt definiál (mindkettőt proxyn keresztül):

```
ask :  (int -> bool)  -> bool
solve: (int * string) -> bool
```

Az első függvény paramétere a gondolt számra vonatkozó kérdés, a másodiké pedig a feltételezett megoldás és a kliens neve. Mindkettő a szerveren fut (így láthatják a gondolt értéket), és a klienseknek csak egy igaz/hamis választ adnak át.

A játék nyertese az a kliens, amelyik elsőként jön rá a helyes megoldásra.

5.1 A szerver, a kliens és a szerver által felkínált kérdéscsomag szignatúrája

```
signature GAMESERVER =
sig
    val min : int
    val max : int
    exception Finished
    val play : int -> Remote.ticket * string
end
signature GAMEPLAYER =
sig
    val play : (Remote.ticket * int * int) -> int option
end
signature GAMECONNECTION =
sig
    exception Finished
    val ask   : (int -> bool)  -> bool
    val solve : (int * string) -> bool
end
```

5.2 A szerver kódja

```
structure Gameserver :> GAMESERVER =
struct
  (* A lehetséges legkisebb és legnagyobb gondolt szám *)
  val min = 0
  val max = 100

  (* Játék végét jelző kivétel *)
  exception Finished

  (* play secret = (ticket, winner), ahol:
   *  - winner annak a kliensnek a neve, amelyik elsőnek
        találja ki a secret számot;
   *  - ticket egy URL, amin át a kliensek elérhetik a
        szerver találgató függvényeit.

   * Ha secret nem a [min,max] intervallumból való, Domain
     kivételt eredményez.
  *)
```

```
fun play secret =
let
      (* A nyertes neve *)
    val winner : string Promise.promise = Promise.promise()
      (* A nyertessé avatások számának korlátozására
         szolgáló változó
      *)
    val semaphor = ref 1

      (* win name = true, és a nyertes neve name lesz.
         Ha már van nyertes, Finished kivételt eredményez
      *)
    fun win name =
    let
        val s = Ref.exchange(semaphor, 0)
    in
        if s = 0 then raise Finished
                 else Promise.fulfill(winner, name); true
    end
```

```
    (* A kliensek találgató függvényei *)
  fun ask’   f          = if ! semaphor = 0
                          then raise Finished
                          else f secret
  fun solve’ (x, name) = if ! semaphor = 0
                          then raise Finished
                          else if x = secret
                          then win name
                          else false

    (* A kliensek függvényeiből képzett struktúra *)
  structure Connection =
      struct
          val ask   = Remote.proxy ask’
          val solve = Remote.proxy solve’
          exception Finished = Finished
      end
  val t = Remote.offer(pack Connection :> GAMECONNECTION)

in
```

```
      if (secret < min orelse secret > max)
      then raise Domain
      else (t, Promise.future winner)
  end
end
```

5.3 Az okos kliens kódja

```
structure Smartplayer :> GAMEPLAYER =
struct
      (* play (ticket, low, high) = secret, ahol:
       *  - ticket a szerver által adott kérdező csomag címe
       *  - low a legkisebb lehetséges szám
       *  - high a legnagyobb lehetséges szám
       *  - secret a megfejtés, ha ez a kliens nyerte a játékot,
            egyébként NONE
      *)
    fun play (t, low, high) =
      let
```

```
  (* A kérdező függvényeket tartalmazó struktúra *)
structure Connection = unpack(Remote.take t) :
  (* A végső megoldást a szerverrel közlő függvény *)
fun tell i = if Connection.solve(i,"Smartplayer")
             then SOME i
             else NONE
             handle Connection.Finished => NONE

  (* A megoldást kereső függvény:
   * bináris kereséssel próbálja megtalálni a helyes
     értéket
  *)
fun try (l, h) =
let
    val m = (l + h) div 2
in
    if l = h
    then tell l
    else if Connection.ask(fn x => m < x)
    then try(m+1, h)
```

```
            else try(l, m)
            handle Connection.Finished => NONE
         end
      in
         try(low, high)
      end
end
```

5.4 A csaló kliens kódja

```
structure Cheater :> GAMEPLAYER =
struct
      (* play (ticket, low, high) = secret, ahol:
       *  - ticket a szerver által adott kérdező csomag címe
       *  - low a legkisebb lehetséges szám
       *  - high a legnagyobb lehetséges szám
       *  - secret a megfejtés, ha ez a kliens nyerte a játékot,
            egyébként NONE
      *)
    fun play (t, _, _) =
```

```
        let
              (* A kérdező függvényeket tartalmazó struktúra *)
            structure Connection = unpack(Remote.take t) :
              (* A végső megoldást a szerverrel közlő függvény *)
            fun tell i = if Connection.solve(i,"Cheater")
                         then SOME i
                         else NONE
                         handle Connection.Finished => NONE
              (* Egy ígéret, amelybe a csaló kilopja a szerver
                 titkát *)
            val secret : int Promise.promise = Promise.promise()
              (* A titkot ellopni hivatott függvény *)
            fun steal x = (Promise.fulfill(secret, x); true)
              (* A ténylegesen kérdésként feltett függvény *)
            val question = Remote.proxy steal
        in
            (Connection.ask question; tell(Promise.future secret))
            handle Connection.Finished => NONE
        end
end
```