

Az Alice nyelvről

Jövők, csomagok és kódistribúció

Szoboszlai Dániel

Kiegészítette, átdolgozta: Hanák Péter

2004. okt. 18. – Rev. 2005. szept. 26. - okt. 2.

1

1 Jövők

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

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

Négyféle jövő 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 érték lusta kiértékelése: a `lazy` (= lusta) kulcsszóval írjuk elő.

```
- val x = lazy 3+2;           - val f = lazy fn y => y+2;
val x : int = _lazy         val f : int -> int = _lazy
- x + 1;                    - f 1;
val it : int = 6           val it : int = 3
- x;                         - f;
val it : int = 5          val it : int -> int = _fn
```

Az Alice az értéket egészen addig nem számítja ki, amíg az értékre szükség nem lesz. (Ez egy informális meghatározás, később pontosabban is definiáljuk.) A „lustaság” jelen formájában csak egyetlen alkalomra szól: ha függvényértéket definiálunk, a függvény alkalmazásai már nem lusták!

Lusta kiértékelésű függvény létrehozása: két egyenértékű szintaxis létezik:

```
- fun f x = lazy x + 1       - fun lazy f x = x + 1
val f : int -> int = _fn    val f : int -> int = _fn
```

Az így definiált függvényt az Alice mindig lustán értékeli ki.

```
- f (3 + 2);
val it : int = _lazy
- it + 1;
val it : int = 7
- f 3 + 2;
val it : int = 6
```

Ha a `map` függvényt a `lazy` kulcsszóval hívjuk, lustává tesszük a kiértékelését.

```
- val ls = lazy map (fn x => x + 1) [1, 2, 3];
val ls : int list = _lazy
- hd ls;
val it : int = 2
- ls;
val it : int list = [2, 3, 4]
```

A lista fejének lekérdezésénél szükség lett a számítás eredményére, ezért az végrehajtott, méghozzá *az egész listára*.

Ezzel szemben a lustának definiált `mapz` függvény – amely a lustaságán kívül mindenben megegyezik a jól ismert `map`-pel – csak a listának azt a részét dolgozza fel, amelyre ténylegesen szüksége is van – a példában csupán az első elemét.

```
- fun lazy mapz f [] = nil
      | mapz f (x::xs) = f x :: mapz f xs;
val mapz : ('a -> 'b) -> 'a list -> 'b list = _fn

- val ls = mapz (fn x => x + 1) [1, 2, 3];
val ls: int list = _lazy
- hd ls;
val it : int = 2
- ls;
val it : int list = 2 :: _lazy
- tl ls;
val it : int list = _lazy
```

Mindez egyben azt is jelenti, hogy lusta függvényekkel egyszerűen generálhatók végtelen sorozatok nyílt végű listákba.

A `zip` függvény (amely két listából párok listáját állítja elő) lusta verziója:

```
- fun lazy zipz (x::xs, y::ys) = (x, y) :: zipz (xs, ys)
      | zipz _ _ = nil;
val zipz : 'a list * 'b list -> ('a * 'b) list = _fn
```

Szép lenne, ha `mapz`-vel és `zipz`-vel a Fibonacci-számok listája a következő módon lenne előállítható:

```
- val rec fibs =
      1 :: 1 :: (lazy mapz op+ (zipz (fibs, tl fibs)));
```

Itt a `lazy` kulcsszó explicit használatára a rekurzív értékdefiníció miatt lenne szükség, nem lenne elég, hogy mind `mapz`, mind `zipz` lusta kiértékelésű.

Sajnos, ez a példa, amely az Alice-dokumentáció *A Tour to Wonderland* c. részében található, nem fordítható le az `alice`-szel:

2.6-2.54: recursive declaration's right-hand side is not a value

1.2 Konkurens jövő

Egy kifejezés párhuzamos (új szálon futó) kiértékelését írja elő a `spawn` (magyarul ivadék, származék; ered) kulcsszó:

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

Megjegyzés: `spawn 3`-at olyan gyorsan értékeli ki az Alice, hogy észre se vesszük, párhuzamos szálon futott.

Konkurens kiértékelésű függvény létrehozására két egyenértékű szintaxis van:

```
- fun f x = spawn x + 1;   - fun spawn f x = x + 1;
val f : int -> int = _fn   val f : int -> int = _fn
```

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

A konkurencia megvalósítása könnyű súlyú (light weight): a rendszer képes szálak százezreit is kezelni.

A párhuzamos kiértékelést olyan példán figyelhetjük meg, amely elég sokáig fut. Nézzük a Fibonacci-számokat elágazó rekurzióval kiszámító függvényt:

```
fun fib (0 | 1) = 1
  | fib n      = fib (n-1) + fib (n-2);
- val n = spawn fib 36;
val n : int = _future
```

Értékeket az *Inspector*al figyelhetünk meg az Alice-ben:

```
inspect [1,2,3];
```

A `spawn` hatása elég nagy Fibonacci-számok kiszámításakor már jól érzékelhető az Alice-szel:

```
val n = spawn fib 35; inspect n;
val m = spawn fib 34; inspect m;
val x = 3; inspect x;
```

A helyzet érdekesebb, ha még több párhuzamos szálat indítunk el egyszerre, pl.

```
inspect (List.tabulate (10, fn i => spawn fib (i+25)));
```

1.3 Ígéreték

A lusta és a konkurens jövő használatakor előre meg kell mondanunk az Alice-nek, hogyan számítsa ki az eredményt. Ez néha nem elég flexibilis.

Az ígéret egy olyan érték, 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.

Ígéreték létrehozására a `Promise` struktúra ad módot, szignatúrája és a függvények leírása a következő diákon láthatók.

```
signature PROMISE =
sig
  type 'a promise
  type 'a t = 'a promise

  exception Promise (* többszöri beváltás kísérlete *)

  val promise : unit -> 'a promise
                (* új ígéret és jövő létrehozása *)
  val future : 'a promise -> 'a
                (* az ígérethez rendelt jövő *)
  val fulfill : 'a promise * 'a -> unit
                (* az ígéret beváltása *)
  val fail : 'a promise * exn -> unit
                (* az ígéret meghiúsítása *)

  val isFulfilled : 'a promise * 'a -> unit
                (* a beváltás vagy meghiúsulás vizsgálata *)
end
```

exception Promise

Raised on multiple attempts to fulfill the same promise.

promise ()

Creates a new promise and an associated future. Returns the promise.

future p

Returns the future associated with p. If p has already been fulfilled with value v, that value is returned.

fulfill (p, v)

Replaces the future associated with p with the value v. If v is the future itself, the exception Future.Cyclic is raised instead. If p has already been fulfilled or failed, the exception Promise is raised.

fail (p, ex)

Requests the exception ex and fails the future associated with the promise p with ex. If p has already been fulfilled or failed, the exception Promise is raised. Equivalent to (Future.await ex; fulfill (p, spawn raise ex))

isFulfilled p

Returns true if p has already been fulfilled or failed, false otherwise. Note that a result of true does not necessarily imply that future p is determined, since p may have been fulfilled with another future.

Ígéreték többek között adatszerkezetek *top-down* felépítésére használhatók. Például a két listát összefűző append alábbi változata jobbrekurzív:

```
fun append (ls1, ls2) =
  let fun append' (nil, p) = fulfill(p, ls2)
      | append' (x::xs, p) =
          let
            val p' = promise()
          in
            ( fulfill(p, x::future p')
              ; append'(xs, p')
            )
          end
      val p = promise()
  in
    ( append'(ls1, p)
      ; future p
    )
  end;
```

1.4 Meghiúsult jövők

Egy jövő kiszámítása közben hibák, kivételek léphetnek föl. Ilyenkor a jövőből megghiúsult jövő lesz.

A megghiúsult jövő nem blokkolja az értékét használó függvényeket, hanem ezeken a helyeken a megghiúsulást okozó kivétel újra fellép.

Ígéreteket a `fail` függvénnyel explicit módon lehet megghiúsítani.

```
- 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 példákból is látszik, hogy a jövő használata nem mindig blokkolja a műveletet, nem mindig értékeli ki a lusta kifejezést, és a megghiúsulása nem mindig okoz kivételt.

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;
- az olyan primitív műveletek esetén, amelyeknek hozzá kell férniük egy argumentumuk értékéhez, a hozzáférés az adott argumentum értékéhez (pl. `op+`, `op=`, `pickling`);
- a funktor alkalmazása a funktorértékre;
- a kicsomagolás a csomag szignatúrájára és a célszignatúrára.

1.6 Jövőkhöz kapcsolódó egyéb struktúrák

Ref: az `'a ref` típust és műveleteit definiálja, többek között:

- `datatype 'a ref = ref of 'a`: megváltoztatható (frissíthető) érték típusa.
- `ref : 'a -> 'a ref`: konstruktorfüggvény.
- `! : 'a ref -> 'a`: a hivatkozott értéket visszaadja.
- `:= : 'a ref * 'a -> unit`: a hivatkozott értéket beállítja.
- `exchange : 'a ref * 'a -> 'a`: egyetlen oszthatatlan lépésben átállítja a referenciát és visszaadja az eredeti értékét. Konkurens szálak szinkronizálására (kölsönös kizárásra, lockolásra) használható.

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

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

A nem tisztán funkcionális minták (pl. `ref` kifejezések) illesztése nem várt viselkedéshez vezethet.

Egy SML-ben kimerítő mintaillesztés Alice alatt nem feltétlenül az, mivel a minta-illesztés nem atomi művelet.

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

```

- fun f (ref false) = 1 | f (ref true) = 2;
1.4-1.21: warning:
match is not exhaustive, because e.g.
    ref _
is not covered
val f : bool ref -> int = _fn

- val r = ref false; r := (lazy (r := false; true));
val r : bool ref = ref (_lazy)
val it : unit = ()

- f r;
Uncaught exception
    Match

- f r;
val it : int = 1

```

2 Csomagok (packages)

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

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 ls = (hd ls, hd (tl ls)) 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:

```
- structure Int' = unpack p : INTEGER;  
Uncaught exception  
  Mismatch (_val)
```

Ha absztrakt típust használunk egy struktúrában, a struktúra be-, majd kicsomagolásával előálló típus különbözni fog az eredetitől:

```
- Word'.toInt (Word8.fromInt 12);  
1.0-1.30: mismatch on application: expression type  
  word  
does not match function's argument type  
  Word'.word  
because type  
  Word8.word  
does not unify with  
  Word'.word
```

A típusok *átlátszó* szignatúrakötés és típusmegosztás alkalmazásával azonossá tehetők:

```
- val p = pack Word8 : WORD  
  structure Word8' =  
    unpack p : (WORD where type word = Word8.t)  
    Word8'.toInt (Word8.fromInt 12);  
val it : int = 12
```

Áttetsző szignatúrakötéssel azonban a típusmegosztás nem alkalmazható:

```
- val p = pack Word8 :> WORD  
  structure Word8' =  
    unpack p : (WORD where type word = Word8.t);  
Uncaught exception  
  Mismatch (_val)
```

A típusmegosztás módszere két struktúra absztrakt típusainak megosztására is használható. Legyenek S1 és S2 érvényes szignatúrakifejezések, ugyanakkor mind t_{s1} , mind t_{s2} olyan absztrakt típusok, amelyekről csak azt tudjuk, hogy egyenlők:

```

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

```

A modulok az Alice-ben lokálisak is lehetnek. Példák:

```

fun g(p1, p2) =
  let
    signature S = sig type t
                      val x : t
                      val f : t -> int
                    end
    structure X1 = unpack p1 : S
    structure X2 = unpack p2 : S where type t = X1.t
  in
    X2.f X1.x
  end;

```

```

val p1 = pack struct type t type t=int val x = 4 fun f z = z * 3
                end : (type t val x : t val f : t -> int)

val p2 = pack (type t type t=int val x = 5 fun f z = z * 4) :
                sig type t val x : t val f : t -> int end

structure P1 = unpack p1 : sig type t val x : t val f : t -> int
                        end

structure P2 = unpack p2 : (type t val x : t val f : t -> int)

P1.f P1.x;
P2.f P1.x;
P1.f P2.x;
g(p1, p2);
g(p2, p1);

```

3 Konzerválás (pickling, serialization)

Konzerv (pickle): egy érték – tetszőleges adatszerkezet, magasabbrendű érték, modul – szerializált¹ és zárt reprezentációja, amely fájlba írható, onnan visszaolvasható, processzek között átadható–átvehető stb.

A konzerválás² típushelyes, mivel a *konzerv* mindig egy csomag – egy értékből és a *típusából* álló pár – szerializált változata.

A konzerválás szemantikája. Egy konzervált érték az általa hivatkozott összes objektum tranzitív lezártját tartalmazza. Háromféle értéket különböztetünk meg:

- Funkcionális (*functional*) érték: megváltoztatható objektumot nem tartalmaz, a konzerválása ezért problémamentes. A visszatöltött érték az eredetileg elmentettől megkülönböztethetetlen objektum lesz.
- Állapottal rendelkező (*stateful*) érték: megváltoztatható objektumot, pl. 'a ref típusú értéket tartalmaz; a konzerválása lehetséges, azonban visszatöltésekor az

¹Szerializálás = egy objektum, érték átalakítása byte- vagy karaktorsorozattá.

²Eredetileg: pickling = pácolás, savanyítás, besózás.

eredeti, megváltoztatható objektumnak a konzerválás pillanatában érvényes másolatát tartalmazza. Egy konzerven belül a visszatöltés után is közősek maradnak az állapottal rendelkező értékek közös hivatkozásai.

- Helyhez kötött (*sited*) érték: erőforrást tartalmaz, ezért konzerválása nem lehetséges. Erőforrásnak számít minden olyan objektum, amely nem értelmezhető azon a folyamaton kívül, amelyben létrehozták (pl. megnyitott fájlok, szálak.) Helyhez kötötnnek számítanak az erőforrásokat létrehozó függvények is.

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

Jövőből nem lehet konzerv. Egy jövő konzerválása a benne lévő értékek kiszámítását és konzerválását váltja ki.

A konzerválás megvalósítása: a `Pickle` struktúrával. Fontosabb elemei:

- `val save : string * package -> unit`: egy csomag konzerválása.
- `val load : string -> package`: egy csomag betöltése. Hibás formátumú fájl egy `Corrupt, Native` stb. paraméterű `IO.io` kivételt vált ki.

4 Elosztott programozás (distributed programming)

Komponensekkel³ és konzervekkel már megvalósítható az elosztott programozás, mivel ezek akár az interneten keresztül is átadhatók és átvehetők tetszőleges helyek között. Az Alice-nek azonban magasabb szintű kommunikációs eszközei is vannak.

Az Alice-ben az adattovábbítás kommunikációs portok segítségével valósul meg két hely (*site*) között. A szükséges függvényeket a `Remote` struktúra tartalmazza, amelyet használata előtt importálni kell:

```
import structure Remote from "x-alice:/lib/distribution/Remote"
```

4.1 Jegyek (tickets)

Egy hely explicit módon felajánlhat egy csomagot a többi hely számára:

```
Remote.offer : package -> ticket
```

³Komponens (az mosml-ben: unit) = önállóan lefordítható programrész.

A függvény hatására az Alice megnyit egy TCP-portot, amelyen keresztül egy HTTP szerver érhető el. A megadott struktúrát az Alice konzerválja, az eredményt fájlként teszi elérhetővé az adott szerveren. A visszaadott jegy (*ticket*) valójában egy `string`: a fájl eléréséhez szükséges URI (Universal Resource Identifier).

A jegy ismeretében a többi hely átveheti a csomagot:

```
Remote.take : ticket -> package
```

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

4.2 Közvetítők (proxies)

A jegy tulajdonképpen csak a kezdeti kapcsolat felvételéhez szükséges a helyek között, a további kommunikáció a felajánlott és átvett csomag függvényeivel valósul meg. A függvények azonban a HTTP-szerveren maradnak, azaz a megfelelő programokat a szerver hajtja végre, a távoli helyek *közvetítőket* (proxies) hívnak meg, amelyek a hívó és a hívott közötti kommunikációt lebonyolítják.

Bármely függvényhez létrehozható közvetítő a `Remote.proxy` függvénnyel:

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

A közvetítő függvény pontosan ugyanazt az értéket fogja kiszámítani, mint az eredeti.

1. A közvetítő függvény mindig konzerválható, akkor is, ha az eredeti nem.
2. Legyen val $f' = \text{Remote.proxy } f$ az f közvetítő függvénye az A helyen (az A hely az f' otthona). Ekkor az $f' x$ alkalmazás kiértékelési lépései a következők:
 - Az A helyen létrejön x' , az x egy konzervált klónja.
 - Az A hely átküldi x' -t a HTTP-szervernek.
 - A HTTP-szerver kiszámítja az $y = f x'$ értéket (y kivétel is lehet).
 - A HTTP-szerver létrehozza y' -t, az y egy konzervált klónját.
 - A HTTP-szerver átküldi y' -t az A helyre, vagyis oda, ahol az f' -t meghívták.

4.2.1 A közvetítők néhány fontos tulajdonsága

- A közvetítő függvény értékét az otthonában a rendszer mindig konkurens módon számítja ki.
- Egy közvetítő függvénynek az argumentumát is, az eredményét is konzerválni kell, így egyik sem tartalmazhat helyhez kötött értéket.
- A paraméterek klónozása miatt meg kell várni az esetleg bennük lévő jövők kiértékelését. Ha egy jövő meghiúsul, akkor kivétel lép fel – és ez lesz a közvetítő eredménye is.
- Egy közvetítő kiértékelése a hagyományos függvényekkel ellentétben több hibalehetőséget rejt magában, ezért többféle kivételt is okozhat. Közülük néhány fontosabb:

<i>A hiba oka</i>	<i>Kivétel</i>
Az argumentum helyhez kötött értéket tartalmaz.	<code>Proxy(SitedArgument)</code>
Az eredmény helyhez kötött értéket vagy olyan meghiúsult jövőt tartalmaz, amelyben meghiúsult jövőt tartalmazó kivétel szerepel.	<code>Proxy(SitedResult)</code>
A közvetítőt létrehozó folyamat befejeződött.	<code>Proxy(Ticket)</code>

5 Egy összetett példa: „Gondoltam egy számot!”

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, vagy rákérdezhetnek a megoldásra. E célra a szerver két függvényt definiál (a kliensek mindkettőt közvetítőn keresztül használhatják):

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

Az `ask` függvény paramétere a kitalálandó számra vonatkozó, eldöntendő kérdés (tesztelő függvény formájában), a `solve` függvényé pedig a feltételezett megoldás és a kliens neve. Mindkettő a szerveren fut (így ismerik a kitalálandó számot), és a klienseknek csak igaz/hamis választ adnak vissza.

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

5.1 A szerver, a kliens és a felkínált kérdéscsomag szignatúrája

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

5.2 A szerver kódja

```
structure Gameserver :> GAMESERVER =
struct
  (* A kitalálható számok tartománya *)
  val min = 0
  val max = 100

  (* A játék végét jelző kivétel *)
  exception Finished

  (* play secret = (ticket, winner), ahol:
   - secret = a kitalálható szám;
   - winner = annak a kliensnek a neve, amelyik elsőnek
     találja ki a secret számot;
   - ticket = egy URI, amelyen át a kliensek elérhetik a
     szerver találgató függvényeit.
   Ha secret nem a [min,max] intervallumból való, Domain
   kivételt eredményez.
  *)
```

```
fun play secret =
let
  (* winner = a nyertes neve *)
  val winner : string Promise.promise = Promise.promise()

  (* !gameOver = true, ha már valaki kitalálta a számot
  *)
  val gameOver = ref false

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

```

    (* A kliensek találgató függvényei *)
    fun ask' f
        = if !gameOver
          then raise Finished
          else f secret

    fun solve' (x, name) = if !gameOver
                          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 (* struct *)

    (* a Connection csomag felajánlása *)

```

```

    val t = Remote.offer(pack Connection :> GAMECONNECT)
in
    if (secret < min orelse secret > max)
    then raise Domain
    else (t, Promise.future winner)
end (* let *)
end (* struct *)

```

5.3 Az okos kliens kódja

```

structure Smartplayer :> GAMEPLAYER =
struct
    (* play (ticket, low, high) = secret, ahol
       - ticket = a szerver által felajánlott kérdéscsomag URI-ja
       - 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) : GAMECONNECT
    (* A megoldást a szerverrel közlő függvény *)
fun tell i = if Connection.solve(i, "Smartplayer")
    then SOME i
    else NONE
    handle Connection.Finished => NONE
    (* A helyes értéket felezéssel kereső függvény *)
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 (* let *)
in try(low, high)
end (* let *)
end (* struct *)

```

5.4 A csaló kliens kódja

```

structure Cheater :> GAMEPLAYER =
struct
    (* play (ticket, low, high) = secret, ahol
        - ticket = a szerver által felajánlott kérdés csomag URI-ja
        - 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) : GAMECONNECT
  (* A megoldást a szerverrel közlő függvény *)
  fun tell i = if Connection.solve(i, "Cheater")
                then SOME i
                else NONE
                handle Connection.Finished => NONE
  (* Ígéret, amelybe a csaló belopja a szerver titkát *)
  val secret : int Promise.promise = Promise.promise()
  (* A titkot ellopó függvény *)
  fun steal x = (Promise.fulfill(secret, x); true)
  (* A lekérdezésre ténylegesen használandó függvény *)
  val question = Remote.proxy steal
in
  (Connection.ask question; tell(Promise.future secret))
  handle Connection.Finished => NONE
end (* let *)
end (* struct *)

```

5.5 A program letölthető változata, továbbfejlesztési feladatok

A bemutatott program kismértékben módosított és kiegészített változata letölthető a tárgy aktuális félévi honlapjáról: <http://dp.iit.bme.hu/mfp/mfp05a>.

- A `Gameserver.play` függvény egy `(ticket, winner, result)` hármast ad eredményül: a jegyet, a győztes nevét és a kitalált számot.
- A `refCheater.aml` fájlban ígéret helyett referenciát használunk a titkos szám ellopására.
- A `game.aml` fájl egy keretprogram, amellyel az egyes programrészeket lehet betölteni és meghívni.

Továbbfejlesztési feladatok:

- Olyan megvalósítás, amely mellett a kitaláló függvény flexibilis maradhat, de amely megakadályozza a titkos szám ellopását.
- Olyan bővítés, amellyel egy kliens új játszmát kérhet a szervertől, mégpedig úgy, hogy egy véletlenszerűen kiválasztott számot kelljen kitalálnia.
- Olyan bővítés, amikor a szerver utasíthatja a klienseket egy-egy játszma lejátszására.