AZ SML-MODULNYELV

Modulfogalmak és elnevezések

- Már találkoztunk két alapkonstrukció (a szignatúra és a struktúra) fogalmával:
  - szignatúra (`signature`): a struktúra *specifikációja*, „*típusa*”,
  - struktúra (`structure`): a szignatúra *megvalósítása*.
- Az SML még egy alapkonstrukciót ismer:
  - funktor (`functor`): olyan generikus konstrukció, amelynek *struktúra* a paramétere;
  - akkor használjuk, amikor a polimorfizmus kevés újrafelhasználható algoritmusok írásához.
- Az MOSML további két elnevezést használ (ld. *Moscow ML Language Overview):*
  - modul *(module):* a struktúra és a funktor *közös* megnevezése;
  - (fordítási, ill. lefordított) *egység* (compilation, ill. compiled *unit):* egy struktúra vagy szignatúra lefordítható, ill. lefordított (tárgykódú) változata.
- Állománynév-kiterjesztések
  - `.sml` (SML, opcionális): struktúra vagy funktor,
  - `.sig` (SML, kötelező): szignatúra,
  - `.ui` (MOSML, kötelező): szignatúra lefordított változata (unit interface code),
  - `.uo` (MOSML, kötelező): struktúra lefordított változata (unit object code).

Szignatúra-illesztés

- Mikor tekinthető egy struktúra egy szignatúra megvalósításának?

  Akkor, ha a struktúra *az összes olyan komponenst definiálja* és az *összes olyan típusdefiníciót kielégíti*, amelyeket a szignatúra elvár. Másszóval definiálja a szignatúrában megadottal

  - ekvivalens típusú *kivételkomponenseket*,
  - kompatíbilis (azaz legalább annyira általános) típusú *értékkomponenseket*,
  - azonos aritású (paraméterszámú) és – ha definiálja – ekvivalens definíciójú *típuskomponenseket*.

- Csakhogy egy struktúra a szignatúrájához képest, többek között

  - *több komponenst* definiálhat;
  - *általánosabb típusú értékeket* definiálhat (ami az újrafelhasználást segíti elő);
  - `datatype`-*deklarációt* használhat `type`-deklaráció helyett, *értékkonstruktort* definiálhat érték helyett (ami az adatabsztrakciót teszi lehetővé);
  - a *deklarációk sorrendje* tetszőleges lehet (ami növeli a flexibilitást).

- A feltett kérdésre ezért pontosabb a következő válasz:

  Egy struktúra akkor és csak akkor tekinthető egy szignatúra megvalósításának, ha a struktúra ún. *törzsszignatúrája* illeszthető az adott szignatúrára.

Törzstípus, törzsszignatúra

- Egy érték *törzstípusa* (principal type) az adott értékhez rendelhető *legáltalánosabb* típus.
- Minden jóldefiniált értéknek van törzstípusa.

  Pl. ha `fun I x = x`, akkor `I : 'a -> 'a`.
- Egy struktúra *törzsszignatúrája* (principal signature) a komponenseihez rendelhető *legspecifikusabb* leírás.
- Minden jóldefiniált struktúrának van törzsszignatúrája.
- A típusellenőrzéshez elegendő a törzsszignatúra ismerete, a struktúrát többé nem kell vizsgálni.
- Egy struktúra törzsszignatúrája a következőképpen állítható elő: ha a deklaráció

  - `type (`$tyvar_1$`, ..., `$tyvar_n$`) `*tycon*` = `*typ* alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
  - `datatype (`$tyvar_1$`, ..., `$tyvar_n$`) `*tycon*` = `con_1` of `typ_1` | ... | `con_k` of `typ_k alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
  - `exception` *id* `of` *typ* alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
  - `val` *id* `=` *exp* alakú, akkor a törzsszignatúra a `val` *id* `:` *typ* specifikációt tartalmazza, ahol *typ* az *exp* kifejezés törzstípusa.

Szignatúra-illesztés (folyt.)

- Egy *szignatúra-jelölt* (candidate signature) akkor és csak akkor illeszthető egy *célszignatúrára* (target signature), ha az összes olyan komponenst és típusdefiníciót tartalmazza, amelyet a *célszignatúra* specifikál.
- Pontosabban, a szignatúra-jelöltnek tartalmaznia kell a célszignatúra
  - összes típuskonstruktorát, mégpedig azonos aritással (paraméterszámmal) és – ha definiálja – ekvivalens definícióval;
  - összes `datatype`-deklarációját, mégpedig úgy, hogy az adatkonstruktoroknak ekvivalens típusúaknak kell lenniük;
  - összes `exception` deklarációját, mégpedig úgy, hogy az argumentumaiknak, ha vannak, ekvivalens típusúaknak kell lenniük;
  - minden értékdeklarációját, mégpedig úgy, hogy a típusuknak legalább annyira általánosnak kell lenniük, mint a célszignatúrában.
- A szignatúra-jelöltnek a célszignatúránál lehet több komponense, és több típusdefiníciót tartalmazhat, de nem lehet benne kevesebb egyikből sem.
- A szignatúra-jelölt a célszignatúra *gyengítése*, mivel a célszignatúra összes tulajdonsága igaz a szignatúra-jelöltre is.

Szignatúra-illesztés: `QUEUE`, `QUEUE_WITH_EMPTY`, `QUEUE_AS_LISTS`

```
signature QUEUE =
sig  type 'a queue
     exception Empty
     val empty : 'a queue
     val insert : 'a * 'a queue -> 'a queue
     val remove : 'a queue -> 'a * 'a queue
end

signature QUEUE_WITH_EMPTY =
sig  include QUEUE
     val is_empty : 'a queue -> bool
end

signature QUEUE_AS_LISTS =
     QUEUE where type 'a queue = 'a list * 'a list
```

- `QUEUE_WITH_EMPTY` illeszthető `QUEUE`-ra, mert kielégíti `QUEUE` összes elvárását.

  `QUEUE` azonban a hiányzó `is_empty` miatt nem illeszthető `QUEUE_WITH_EMPTY`-re.
- `QUEUE_AS_LISTS` illeszthető `QUEUE`-ra, csak abban különbözik tőle, hogy `'a queue`-t specifikálja.

  `QUEUE` azonban nem illeszthető `QUEUE_AS_LISTS`-re, mert a `QUEUE`-beli `'a queue` nem ekvivalens `'a list * 'a list`-tel.

Szignatúra-illesztés: QUEUE, QUEUE_AS_LIST

- ```
  signature QUEUE =
  sig  type 'a queue
       exception Empty
       val empty : 'a queue
       val insert : 'a * 'a queue -> 'a queue
       val remove : 'a queue -> 'a * 'a queue
  end

  signature QUEUE_AS_LIST =
  sig  type 'a queue = 'a list
       exception Empty
       val empty : 'a list
       val insert : 'a * 'a list -> 'a list
       val remove : 'a list -> 'a * 'a list
  end
  ```
- Úgy vélhetjük, hogy `QUEUE_AS_LIST` nem illeszthető `QUEUE`-ra, annyira különbözik tőle.
- Csakhogy az előzővel ekvivalens az alábbi definíció:

  ```
  signature QUEUE_AS_LIST =
       QUEUE where type 'a queue = 'a list
  ```

  és az utóbbi nyilvánvalóan illeszthető `QUEUE`-ra.

Szignatúra-illesztés: MERGEABLE_QUEUE, MERGEABLE_INT_QUEUE

- Említettük, hogy a szignatúra-jelöltben az értékek típusa általánosabb lehet, mint a célszignatúrában.
- A szignatúra-illesztés együtt járhat azzal, hogy a polimorf típusokat konkrét típusokra cseréljük.
- ```
  signature MERGEABLE_QUEUE =
  sig
      include QUEUE
      val merge : 'a queue * 'a queue -> 'a queue
  end

  signature MERGEABLE_INT_QUEUE =
  sig
      include QUEUE
      val merge : int queue * int queue -> int queue
  end
  ```
- A `MERGEABLE_QUEUE` szignatúra-jelölt illeszthető a `MERGEABLE_INT_QUEUE` célszignatúrára, mert az előbbiben specifikált polimorf `merge` függvény típusát leíró típuskifejezés típusváltozója leköthető az `int` típussal.

Szignatúra-illesztés: RBT_DT, RBT

```
signature RBT_DT =
sig  datatype 'a rbt = Empty
                     | Red of 'a rbt * 'a * 'a rbt
                     | Black of 'a rbt * 'a * 'a rbt
end

signature RBT =
sig  type 'a rbt
     val Empty : 'a rbt
     val Red : 'a rbt * 'a * 'a rbt -> 'a rbt
end
```

- Az RBT_DT szignatúra-jelölt illeszthető az RBT célszignatúrára, mert az RBT_DT-ben a `datatype` deklarációval specifikált típus és adatkonstruktorai illeszthetők az RBT-ben specifikált `'a rbt` absztrakt típusra és a két értékspecifikációra (`Empty` és `Red`). Fordítva nem igaz.
- RBT_DT ugyanis a következő típust, ill. adatkonstruktorokat specifikálja:

```
type 'a rbt
con 'a Empty : 'a rbt
con 'a Red : 'a rbt * 'a * 'a rbt -> 'a rbt
con 'a Black : 'a rbt * 'a * 'a rbt -> 'a rbt
```


Szignatúra-illesztés (folyt.)

- Most már még pontosabban válaszolhatunk a kérdésre:
- Mikor tekinthető egy *struktúra-jelölt* egy *célszignatúra* megvalósításának?
- Akkor és csak akkor, ha a struktúra-jelölt *törzsszignatúrája* illeszthető a célszignatúrára.
- Nyilvánvaló, hogy minden struktúra kielégíti a törzsszignatúráját (az illesztési reláció reflexív).
- Bármely szignatúra, amelyet egy struktúra megvalósít, *gyengébb* az adott struktúra törzsszignatúrájánál.
- A törzsszignatúra ezért a *legerősebb* szignatúra, amelyet egy struktúra megvalósíthat.

Szignatúra-kötés

- *Szignatúra-kötéssel* (signature ascription) írjuk elő, hogy egy struktúra valósítson meg egy szignatúrát.
- A szignatúra-kötés *gyengíti* a struktúra szignatúráját az összes további felhasználás számára.
- Kétféle szignatúra-kötés van az SML-ben:
  - *átlátszó* vagy *leíró* (transparent, descriptive): a struktúra *látható szignatúrája* az adott struktúrában definiált típusokkal *bővített* célszignatúra lesz,
  - *áttetsző* vagy *korlátozó* (opaque, restrictive): a struktúra *látható szignatúrája* a célszignatúra lesz, bővítés nélkül.
- A szignatúra-kötés mindkét változata elrejti azokat a komponenseket, amelyek a célszignatúra nem specifikál.
- A moduláris programozás biztonsága megköveteli a típusinformációt gondos kezelését. A láthatóvá tételnek és az elrejtésnek egyformán fontos a szerepe.
- Az áttetsző szignatúra-kötéssel a típusinformáció láthatóságát korlátozzuk.
- Az átlátszó szignatúra-kötéssel a típusinformációt láthatóvá tesszük.

Struktúra-deklaráció szignatúra-kötéssel

- Már láttuk, hogy egy struktúra-deklarációban hogyan alkalmazzuk a kétféle szignatúra-kötést:
  - átlátszó: `structure` *`strid`* `:` *`sigexp`* `=` *`strexp`*
  - áttetsző: `structure` *`strid`* `:>` *`sigexp`* `=` *`strexp`*
- A típusellenőrzés lépései szignatúrához kötött struktúra-deklaráció esetén a következők:
  - *strexp* megvalósítja-e *sigexp*-et? Ennek eldöntéséhez a fordító
    - meghatározza *strexp* $sigexp_0$ törzsszignatúráját, és megpróbálja illeszteni a *sigexp* célszignatúrára; valamint
    - előállítja a bővített *sigexp'* szignatúrát úgy, hogy *sigexp*-et bővíti a $sigexp_0$-ban lévő típusdeklarációkkal;
  - a struktúranévhez köti a szignatúrát a kötés előírt módja szerint: a struktúra látható szignatúrája
    - átlátszó szignatúra-kötés esetén *sigexp'*,
    - áttetsző szignatúra-kötés esetén *sigexp* lesz.
- A leírtakból is kitűnik, hogy az átlátszó szignatúra-kötés az áttetsző szignatúra-kötés *speciális esete:* a bővített szignatúrát a programozó maga is előállíthatná (technikai nehézségektől eltekintve, ui. néha nem férhet hozzá a szükséges információhoz).

Struktúra-deklaráció szignatúra-kötéssel (folyt.)

- Idézzük föl a kétféle szignatúra-kötést:
  - átlátszó: `structure` *`strid`* `:` *`sigexp`* `=` *`strexp`*
  - áttetsző: `structure` *`strid`* `:>` *`sigexp`* `=` *`strexp`*
- A szignatúrához kötött struktúra-deklaráció kiértékelését a fordító így folytatja:
  - kiértékeli *strexp*-et;
  - előállítja az eredményül kapott érték egy *nézetét* úgy, hogy eldobja azokat az értékeket, amelyeket a *sigexp* célszignatúra nem tartalmaz;
  - a *strid* nevet ehhez a nézethez köti.

Struktúra-deklaráció áttetsző szignatúra-kötéssel: `QUEUE`, `Queue_as_lists`

- Az áttetsző szignatúra-kötés legfontosabb célja az adatabsztrakció elősegítése.
- Nézzük ismét a már látott példát:

```
signature QUEUE =
sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
end

structure Queue_as_lists :> QUEUE =
struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (nil, nil) = raise Empty
      | remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
end
```

Struktúra-deklaráció áttetsző szignatúra-kötéssel (folyt.)

- Az áttetsző szignatúra-kötés garantálja, hogy a `'a Queue_as_lists.queue` *absztrakt*, és így *kizárólag* az `empty`, `insert` és `remove` műveleteket lehessen alkalmazni ilyen típusú értékekre.
- A programozó *nem használhatja ki*, hogy most a `'a Queue_as_lists.queue` típust listákból álló párral valósítjuk meg.
- Ezért a szignatúrát megvalósító struktúra szabadon, a többi programrész konzisztenciájának megsértése nélkül módosítható.

Struktúra-deklaráció áttetsző szignatúra-kötéssel (folyt.)

- A típusinformáció elrejtése a reprezentáció (ábrázolás) invariánsait is elszigeteli az absztrakció megvalósításától.
  - Az `'a Queue_as_lists.queue` típust egy olyan absztrakt gép állapottípusának tekinthetjük, amelynek csak három parancs adható: `empty` (amely a kezdőállapotot hozza létre), `insert` és `remove`.
  - A `Queue_as_lists` struktúrán belül invariáns állításokkal jellemezhetjük az absztrakt gép belső állapotát.
  - Az adatabsztrakció elegáns eljárást nyújt az invariáns állítások alkalmazásához; az *assume–ensure* vagy *rely–guarantee* néven ismert eljáráshoz két követelményt kell kielégíteni:
    - minden inicializáló parancsnak *garantálnia kell* az invariáns teljesülését a végrehajtása után;
    - minden állapotmódosító parancs *felteheti*, hogy az invariáns teljesül a parancs végrehajtásának kezdetén, és minden ilyen parancsnak *garantálnia kell* az invariáns teljesülését a végrehajtása után.
  - Teljes indukcióval belátható, hogy az invariáns állítás az összes állapotra teljesül, azaz valóban invariáns!

Struktúra-deklaráció áttetsző szignatúra-kötéssel: prioritási sor

- Olyan absztrakt prioritásisor-típust akarunk létrehozni, amely tetszőleges típusú elemekből állhat.
- A műveletek (függvények) nem lehetnek politípusúak, mert az elemek relatív prioritását összehasonlítással tudjuk megállapítani. Ezt függőséget fejezi ki az alábbi szignatúra:

```
signature PQ =
sig
    type elt
    val lt : elt * elt -> bool
    type queue
    exception Empty
    val empty : queue
    val insert : elt * queue -> queue
    val remove : queue -> elt * queue
end
```

- Egy lehetséges megvalósítás vázlatát mutatja a következő példa, ahol az elemek `string` típusúak.

Struktúra-deklaráció áttetsző szignatúra-kötéssel: prioritási sor (folyt.)

- A megvalósítástól független absztrakt típus *áttetsző szignatúrát* igényel:

```
structure PrioQueue :> PQ =
struct  type elt = string
        val lt : string * string -> bool = (op <)
        type queue = ...
end
```

- Csakhogy így `PrioQueue.queue` mellett `PrioQueue.elt` is absztrakt típus lett, így nem tudunk `PrioQueue.elt` típusú értéket létrehozni, és pl. nem hívhatjuk a `PrioQueue.insert` függvényt. Ezért `PrioQueue.elt`-nek nem kellene absztrakt típusnak lennie.
- Egy lehetséges megoldás az, hogy a PQ szignatúrát bővítjük, és a bővített szignatúrát kötjük a struktúrához:

```
signature STRING_PQ = PQ where type elt = string
structure PrioQueue :> STRING_PQ = ...
```

vagy

```
structure PrioQueue :> PQ where type elt = string = ...
```

- A tanulság: megfontolást igényel, hogy mely típusokat válasszuk absztraktnak, és melyeket ne.

LUSTA LISTÁK

Lusta lista

- Olyan lista, amelynek a farka függvény, ezáltal késleltetjük a kiértékelését.
- Ily módon *végtelen listákat* hozhatunk létre.
- A lusta listának hátrányai, veszélyei is vannak, pl.
  - egy lusta lista bármely részét megjeleníthetjük, de sohasem az egészet;
  - két lusta lista elemeiből páronként képezhetünk egy harmadikat, de nem számíthatjuk ki egy lusta lista elemeinek az összegét, nem kereshetjük meg benne a legkisebbet, nem fordíthatjuk meg az elemek sorrendjét;
  - úgy kell rekurziót definiálnunk, hogy nincs alapeset;
  - egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény tetszőleges véges része véges idő alatt előáll.
- A lusta listát sorozatnak *(sequence)* nevezzük, és a `seq` típusoperátort használjuk a létrehozására.

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Lusta lista (folyt.)

- Egy sorozat fejét adja eredményül a `head` függvény; abortál, ha üres sorozatra alkalmazzuk.

```
(* head : 'a seq -> 'a
*)
fun head (Cons(x, _)) = x
```

- Egy sorozat farkát adja eredményül a `tail` függvény; abortál, ha üres sorozatra alkalmazzák.

```
(* tail : 'a seq -> 'a seq
*)
fun tail (Cons(_, xf)) = xf()
```

A sorozat farka `unit -> 'a seq` típusú *függvény*, erre illesztjük az `xf` mintát `tail` fejében; `tail` törzsében `xf`-et a `()` argumentumra kell alkalmazni.