AZ SML-MODULNYELV

Típusmegosztás specifikálása

- Ebben a részben *modulok szimmetrikus összekapcsolásával* foglalkozunk.
- A különböző modulokban (akár azonos néven) specifikált absztrakt típusok mind különbözők. Általában ezt akarjuk. De nem mindig.
- A különböző modulokban specifikált típusok azonosságát az ún. *típusmegosztási előírással* (type sharing constraint) adhatjuk meg.
- A következő példák egy mértani elemeket megvalósító programból valók.
- Csupán két térbeli elemet valósítunk meg: a pontot és gömböt.

```
signature GEOMETRY =
sig
    structure Point : POINT
    structure Sphere : SPHERE
end
```

- A mértani elemek ábrázolását a vektorra és a pontra alapozzuk.

Példa: mértani alapelemek ábrázolása (VECTOR, POINT)

- A `VECTOR` szignatúra egy vektor skalárral való szorzatát (`scale`), két vektor összegét (`add`) és skalárszorzatát (`dot`), továbbá a vektorösszeadás egységelemét (`zero`) specifikiálja.

```
signature VECTOR =
sig  type vector
     val zero : vector
     val scale : real * vector -> vector
     val add : vector * vector -> vector
     val dot : vector * vector -> real
end
```

- A `POINT` szignatúra egy pont eltolását egy vektor mentén (`translate`) és egy végpontjaival megadott vektor előállítását (`ray`) specifikiálja.

```
signature POINT =
sig  structure Vector : VECTOR
     type point
     val translate : point * Vector.vector -> point
     val ray : point * point -> Vector.vector
end
```


Példa: mértani alapelemek ábrázolása (SPHERE)

- A gömböt a középpontjával és a sugarával adjuk meg.
- A gömböt létrehozó függvényt (`sphere`) az alábbi szignatúra specifikiálja:

```
signature SPHERE =
sig
    structure Vector : VECTOR
    structure Point : POINT
    type sphere
    val sphere : Point.point * Vector.vector -> sphere
end
```

- Emlékeztető: a típusneveket és az értékneveket különböző névterek tárolják, ezért a `sphere` azonosító egyszerre használható típusnévként és értéknévként.
- Vegyük észre, hogy tér dimenziója nem része a specifikációnak!
- A dimenziót majd csak a modul megvalósításakor rögzítjük, ezzel elősegítjük a specifikáció *újrafelhasználását*.

Különböző modulokban specifikált absztrakt típusok különböző volta

- Két- és háromdimenziós mértant így kezdődő struktúra-deklarációkkal valósíthatunk majd meg:

```
structure Geom2D :> GEOMETRY = ...
structure Geom3D :> GEOMETRY = ...
```

- Az *áttetsző* szignatúrakötésnek köszönhetően a két struktúrának *különböző* lesz a látható szignatúrája: a típusellenőrzés gondoskodik róla, hogy pl. a háromdimenziós térben ábrázolt `Geom3D.Sphere.sphere` középpontja ne lehessen a kétdimenziós térben ábrázolt `Geom2D.Point.point` pont.
- Ez jó dolog, növeli a programozás biztonságát.
- Sajnos, nemcsak `Geom2D` különbözik `Geom3D`-től, hanem pl. `Geom2D.Sphere.Vector` is különbözik `Geom2D.Point.Vector`-tól!
- Ezért típushibát jelez a fordító a következő sor fordításakor (ahol `p` és `q` adott, `Geom2D.Point.point` típusú pontok):

```
Geom2D.Sphere.sphere (p, Geom2D.Point.ray (p, q))
```

- `Geom2D.Point.ray (p, q)` eredménye `Geom2D.Point.Vector.vector` típusú, `Geom2D.Sphere.sphere` ugyanakkor `Geom2D.Sphere.Vector.vector` típusú értéket vár. Ezt nyilvánvalóan nem akarjuk. Mi lehet az oka, hogyan küszöbölhetjük ki?

Különböző modulokban specifikált absztrakt típusok megosztása (SPHERE)

- Az ok az, hogy a szignatúrákban specifikáltuk azokat az alstruktúrákat, amelyektől e szignatúrák függnek, így a pont-absztrakciót *két*, a vektor-absztrakciót *három példányban* hoztuk létre!
- Mivel áttetsző szignatúrakötést használunk, a látható szignatúráik mind különböznek!
- *Általában ezt akarjuk, néha nem.* Az SML-ben előírhatjuk, hogy két alstruktúra valamely absztrakt típusa legyen azonos. Erre való a *típusmegosztási előírás* (type sharing constraint).
- `SPHERE` módosított specifikációja (a módosítást **félkövér szedés** jelöli):

```
signature SPHERE =
sig  structure Vector : VECTOR
     structure Point : POINT
     sharing type Point.Vector.vector = Vector.vector
     type sphere
     val sphere : Point.point * Vector.vector -> sphere
end
```

- A *típusmegosztási előírás* egy változatával, a *struktúramegosztási előírással* (structure sharing constraint) előírhatjuk, hogy két alstruktúra *összes* absztrakt típusa azonos legyen.

```
... sharing Point.Vector = Vector ...
```


Különböző modulokban specifikált absztrakt típusok megosztása (GEOMETRY)

- `GEOMETRY` módosított specifikációja (két változatban, a módosítást **félkövér szedés** jelöli):

```
signature GEOMETRY =
sig
    structure Point : POINT
    structure Sphere : SPHERE
    sharing type Point.point = Sphere.Point.point
    sharing type Point.Vector.vector = Sphere.Vector.vector
end
```

```
... sharing Point = Sphere.Point
    sharing Point.Vector = Sphere.Vector ...
```

- A megosztási előírás tehát garantálja, hogy
  - a típusegyenletek mindig teljesüljenek, amikor a `GEOMETRY` szignatúrát és összes komponensét megvalósítjuk;
  - a megosztási előírás által érintett összes absztrakt típus azonos legyen.
- `VECTOR`-t és `POINT`-ot *egy példányban* valósítjuk meg, és e példányokat *újra felhasználjuk* a magasabb szintű absztrakció során (ld. a következő fóliákon).

Példa: VECTOR, POINT, SPHERE és GEOMETRY egy 3D-s megvalósítása

- `structure Vector3D : VECTOR = ...`
-
```
structure Point3D : POINT =
struct
    structure Vector : VECTOR = Vector3D
        ...
end
```

-
```
structure Sphere3D : SPHERE =
struct
    structure Vector : VECTOR = Vector3D
    structure Point : POINT = Point3D
        ...
end
```

-
```
structure Geom3D :> GEOMETRY =
struct
    structure Point = Point3D
    structure Sphere = Sphere3D
end
```

- Fordítási idejű típushibához vezetne egyes 2D-s elemek alkalmazása a 3D-s megvalósításban. Példa:

```
... structure Sphere = Sphere2D ...
```

A típusmegosztás elkerülése, a megosztási előírások számának csökkentése

- Fölvethető a kérdés, hogy a típusmegosztás elkerülhető-e a pont- és a vektor-absztrakció következtében létrejött példányszámok csökkentésével.
- A válasz: igen; azon az áron, hogy az egész programstruktúrát erőszakosan megváltoztatjuk.
- Első lépésként SPHERE-ben `Vector.vector`-t `Point.Vector.vector`-ra cseréljük:

```
signature SPHERE =
sig  structure Point : POINT
     type sphere
     val sphere : Point.point * Point.Vector.vector -> sphere
end
```

- Ezzel GEOMETRY-ben `sharing Point.Vector = Sphere.Vector` feleslegessé vált, a megosztási előírások száma eggyel csökkent:

```
signature GEOMETRY =
sig  structure Point : POINT
     structure Sphere : SPHERE
     sharing Point = Sphere.Point
end
```


A típusmegosztás elkerülése, a megosztási előírások számának csökkentése (folyt.)

- Ha a `Point` alstruktúra specifikációját is sikerülne feleslegessé tenni SPHERE-ben, egyáltalán nem kellene megosztási előírás. Ekkor ez maradna SPHERE-ből:

```
signature SPHERE =
sig
     type sphere
     val sphere : Point.point * Point.Vector.vector -> sphere
end
```

- Most SPHERE-ből hiányzik a `Point` specifikálása. Ha `Point` már definiálva van, akkor `Point` lefordítható.
- Csakhogy ettől kezdve a SPHERE szignatúra a `Point` struktúrától, azaz a POINT szignatúra *egy megvalósításától* függ. Pl. a 2D-s megvalósítástól, ami által a szignatúra dimenziótól való függetlensége megszűnt, az absztrakció csorbát szenvedett.
- Ez az út tehát járhatatlan, de semmiképpen nem javasolható.
- Az eset hasonló ahhoz, amikor egy függvénynek nem paraméterként, hanem globálisként adunk át egy értéket.
- A `Point` struktúra a SPHERE szignatúra *paraméterének* tekinthető az ismertetett értelemben.

A típusmegosztás elkerülése, a megosztási előírások számának csökkentése (folyt.)

- Egyetlen lehetőség maradt, hogy megszabaduljunk a megosztási előírásoktól, az, hogy a GEOMETRY szignatúrából töröljük a `Point` alstruktúrát. Ez lehetséges éppen, csak nem megoldás, mert csupán elhalasztja a problémát, de nem oldja meg a következők miatt.

  Pl. egy mértani elemeket megvalósító igazi programban több olyan elem van, amely a pont-fogalomra épít. Ezeknek feltétlenül szükségük lesz a megosztási előírásra.
- Lássunk egy további példát ennek alátámasztására, a SEMI_SPACE specifikációját. A `side` predikátummal vizsgálható meg, hogy *egy adott pont a tér melyik felében* van – feltéve, hogy ez lehetséges: ezért választjuk a `bool option` típusú eredményt.

```
signature SEMI_SPACE =
sig
    structure Point : POINT
    type semispace
    val side : Point.point * semispace -> bool option
end
```

- SEMI_SPACE-ből, SPHERE-hez hasonlóan, nem törölhetjük a `Point` alstruktúra specifikációját, ezért `Point`-ból mégiscsak két újabb példányt hozunk létre, azonosságukat pedig majd megosztási előírással kell kifejeznünk a GEOMETRY szignatúra új változatában.

A típusmegosztás elkerülése, a megosztási előírások számának csökkentése (folyt.)

- A `Point` alstruktúrák azonosságát tehát megosztási előírással kell kifejeznünk EXTD_GEOMETRY-ben, GEOMETRY kiegészített, de a `Point` alstruktúra nélküli változatában:

```
signature EXTD_GEOMETRY =
sig
    structure Sphere : SPHERE
    structure SemiSpace : SEMI_SPACE
    sharing Sphere.Point = SemiSpace.Point
end
```

- Amiről itt szó van, nem más, mint a moduláris programozás alapvető belső ellentmondása.
  - Egyrészt el akarjuk *szigetelni* egymástól a modulokat, hogy egymástól függetlenül legyenek kezelhetők, és ne befolyásolja az egyik modul megváltoztatása a többit. (A globális értékektől, változóktól való függés is az elszigetelés ellen hat!)
  - Másrészt a modulok kombinálásával programokat hozunk létre; ekkor a különböző modulok egyes programelemeinek azonosságát (SML: megosztási előírásokkal) ki kell kötnünk.
- A megosztási előírás olyan *eszköz*, amely valaminek a bekövetkezése (ti. a specifikáció rögzítése) után hoz létre kapcsolatot a különböző absztrakciók között, és teremti meg az egész program koherenciáját. Ez a megközelítés az SML – más nyelvekben ismeretlen – sajátossága.

Paraméterezhető, más néven generikus modulok

- Az újrafelhasználhatóságot segíti elő a *paraméterezhető*, más néven *generikus* modul, azáltal hogy a megvalósítás egy vagy több elemét specifikálatlanul hagyja. A spefifikálatlan elemek specifikálása a modul egy *példányát* hozza létre. A közös részt *csak egyszer* kell megírni.
- Az SML-ben az ilyen generikus modult *funktornak* (`functor`) nevezik. A funktornak struktúra a paramétere is, az eredménye is. A funktor egy *példányát* úgy hozzuk létre, hogy alkalmazzuk egy (létező) struktúrára.
- A *funktordeklarációnak* (vagy *funktorkötésnek)* két változata van, az *átlátszó*:

```
functor funid (decs) : sigexp = strexp
```

- és az *áttetsző*:

```
functor funid (decs) :> sigexp = strexp
```

- A funktor típusának ellenőrzéséhez a fordító megvizsgálja, hogy a funktor törzse megfelel-e a szignatúra-kötés által előírt szignatúrának, feltéve hogy a funktor paramétereinek megfelelő a szignatúrája.
- Mint tudjuk, az *áttetsző* szignatúra-kötés a megadott szignatúrát eredményezi, az *átlátszó* szignatúra-kötés pedig ennek a törzsszignatúra szerinti típusokkal bővített változata.

Példa: polimorf szótár megvalósítása funktorral

- Egy korábbi példánk olyan polimorf szótár volt, ahol a keresési kulcsot és a rajta végrehajtható műveleteket egy *alstruktúra* specifikálta. A félkövér szedés a változatok közötti eltérésre utal.

```
structure StringDict :> DICT where type Key.t = string =
struct
    structure Key : ORDERED = LexString
    datatype 'a dict = Empty
                     | Node of 'a dict * Key.t * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) =                NONE
      | lookup (Node (dl, l, v, dr), k) =
              if Key.lt (k, l) then       lookup (dl, k)
              else if Key.lt (l, k) then lookup (dr, k)
              else                        SOME v
end
```

- Látható, hogy különbség csak a keresési kulcs típusában és a rajta végrehajtható műveletekben van; mindezt az `ORDERED` szignatúra specifikálja.

Példa: polimorf szótár megvalósítása funktorral (`DictFun`)

- Az alstruktúrát, ha funktort deklarálunk, paraméterként adhatjuk át. A struktúradeklaráció és a funktordeklaráció közötti különbségeket most is félkövér szedéssel emeljük ki.

```
functor DictFun (structure K : ORDERED) :>
                  DICT where type Key.t = K.t =
struct
    structure Key : ORDERED = K
    datatype 'a dict = Empty
                     | Node of 'a dict * Key.t * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) =                NONE
      | lookup (Node (dl, l, v, dr), k) =
              if Key.lt(k, l) then        lookup (dl, k)
              else if Key.lt (l, k) then lookup (dr, k)
              else                        SOME v
end
```

- A `DICT where type Key.t = K.t` szignatúra az `'a dict` típus absztrakt voltát megőrzi. A `Key.lt` összehasonlító műveletet a paraméterként átadott struktúra valósítja meg.

Funktoralkalmazás szignatúrája, funktor generativitása, ill. applikativitása

- A funktoralkalmazás *`funid (binds)`* alakú kifejezés, ahol *`binds`* a funktorargumentumok kötésének egy sorozata.
- Egy *funktoralkalmazás szignatúrája* a következő eljárással határozható meg. Feltesszük, hogy ismerjük a funktorparaméterek szignatúráját, valamint a funktor látható szignatúráját (a megadottat áttetsző, a bővítettet átlátszó szignatúrakötés esetén).
  - Minden argumentum szignatúráját illesztjük a funktor megfelelő paraméterének szignatúrájára. Ezzel minden argumentumra megkapjuk a paraméterszignatúrák egy bővített változatát.
  - Ha az eredmény szignatúrája hivatkozik a funktorparaméter valamely típuskomponensére, akkor a bővített paraméterszignatúrában lévő típusdefiníciónak meg kell jelennie az eredmény szignatúrájában.
- Az így előállított szignatúra átlátszó kötéssel kapcsolódik a funktoralkalmazáshoz. Ez azt jelenti, hogy ha a funktor eredményszignatúrája egy típust absztraktként specifikál, akkor e funktor minden alkalmazása e típusból egy új példányt hoz létre. Ezt a viselkedést a funktor *generativitásának* nevezzük. (Ezzel szemben a funktor *applikativitása* azt jelenti, hogy a funktor összes példánya megosztva „használja” ugyanazt az absztrakt típust.)

Példa: polimorf szótár (`LtIntDict`, `LexStringDict`, `DivIntDict`)

- A szótár három változatát a `DictFun` funktor alkalmazásával könnyű előállítani:

```
structure LtIntDict = DictFun (structure K = LessInt)
structure LexStringDict = DictFun (structure K = LexString)
structure DivIntDict =  DictFun (structure K = DivInt)
```

- Idézzük föl `LexString`, `LessInt` és `DivInt` egy-egy megvalósítását:

```
structure LexString : ORDERED =
struct  type t = string
    val lt = (op <)
    val eq = (op =)
end
```

```
structure LessInt : ORDERED =
struct  type t = int
    val lt = (op <)
    val eq = (op =)
end
```

```
structure DivInt : ORDERED =
struct  type t = int
    fun lt (m, n) = (n mod m = 0)
    fun eq (m, n) = lt (m, n) andalso lt (n, m)
end
```

- Például `LessInt` bővített szignatúrája ez: `ORDERED where type t = int`.
- Ha a `K` paraméter aktuális értéke `LessInt`, akkor `K.t` és `int` ekvivalensek lesznek, és így `DictFun` aktuális szignatúrája ez: `ORDERED where type Key.t = int`.

Funktorok és a típusmegosztás specifikálása egy példán

- Korábban specifikáltuk a `GEOMETRY` szignatúrát és komponenseit.
- Mivel a célunk többféle (2D és 3D) megvalósításuk, érdemes őket funktorként definiálnunk.

```
functor PointFun
    (structure V : VECTOR) : POINT = ...
```

```
functor SphereFun
    (structure V : VECTOR
     structure P : POINT) : SPHERE =
struct
    structure Vector = V
    structure Point = P
        ...
end
```

```
functor GeomFun
    (structure P : POINT
     structure S : SPHERE) : GEOMETRY =
struct
    structure Point = P
    structure Sphere = S
end
```


Funktorok és a típusmegosztás specifikálása egy példán (folyt.)

- Az előző funktordefiníciókkal a 2D-s programcsomag így valósítható meg:

```
structure Vector2D : VECTOR = ...
```

```
structure Point2D : POINT =
     PointFun (structure V = Vector2D)
```

```
structure Sphere2D : SPHERE =
    SphereFun (structure V = Vector2D and P = Point2D)
```

```
structure Geom2D : GEOMETRY =
    GeomFun (structure P = Point2D and S = Sphere2D)
```


Funktorok és a típusmegosztás specifikálása egy példán (folyt.)

- Egyetlen baj van csupán: `SphereFun` és `GeomFun` típushibás! A hiba javítható: típusmegosztást kell előírnunk.

```
functor SphereFun
    (structure V : VECTOR
     structure P : POINT
     sharing P.Vector = V) : SPHERE =
struct
    structure Vector = V
    structure Point = P
        ...
end
```

```
functor GeomFun
    (structure P : POINT
     structure S : SPHERE
     sharing P.Vector = S.Vector
     sharing P = S.Point) : GEOMETRY =
struct
    structure Point = P
    structure Sphere = S
end
```

A típusmegosztás elkerülése funktor használatakor

- Most is felvetődik a kérdés: elkerülhető-e típusmegosztás?
- Igen, de azon az áron, hogy erőszakosan megváltoztatjuk a program szerkezetét.
- A típusmegosztás fő erénye, hogy közvetlenül és tömören fejezi ki az elvárt összefüggéseket, de a paraméterek szignatúrájának definiálásakor még nem kell velük foglalkozni. Ez a tulajdonság nagyon megkönnyíti az „előre gyártott” programrészek újrafelhasználását, hiszen ilyen esetekben a típusmegosztás konkrét igényét előre (azaz pl. a paraméterek definiálásakor) lehetetlen megmondani.
- Vegyük elő a már látott példát, amelyben a megosztási specifikációk számát egyre csökkentettük.

```
signature EXTD_GEOMETRY =
sig
    structure Sphere : SPHERE
    structure SemiSpace : SEMI_SPACE
    sharing Sphere.Point = SemiSpace.Point
end
```


A típusmegosztás elkerülése funktor használatakor (folyt.)

- Az `EXTD_GEOMETRY` szignatúrát ezzel a funktorral valósítjuk meg:

```
functor ExtdGeomFun
    (structure Sp : SPHERE
     structure Ss : SEMI_SPACE
     sharing Sp.Point = Ss.Point) =
struct
    structure Sphere = Sp
    structure SemiSpace = Ss
end
```

- Ahhoz, hogy a megosztási előírást elhagyhassuk a funktor paraméteréből, gondoskodnunk kell arról, hogy az `EXTD_GEOMETRY` szignatúra által előírt típusmegosztás teljesüljön.
- Megoldás lehet a `POINT` szignatúrát megvalósító struktúra „kiemelése”.
- Kétféle módon járhatunk el.

A típusmegosztás elkerülése funktor használatakor (folyt.)

- Az első lehetőség az, hogy `ExtdGeomFun` paraméterként `SPHERE` és `SEMI_SPACE` egy-egy megvalósítása helyett a közös elem, azaz `POINT` egy megvalósítását kapja, és a funktor törzsében hozza létre `SPHERE` és `SEMI_SPACE` egy-egy megvalósítását.
- Ehhez a `SphereFun` és a `SemiSpaceFun` funktorokat is megfelelően kell paraméterezni:

```
functor SphereFun
    (structure P : POINT) : SPHERE =
struct
    structure Vector = P.Vector
    structure Point = P
    ...
end

functor SemiSpaceFun
    (structure P : POINT) : SEMI_SPACE =
struct
    ...
end
```


A típusmegosztás elkerülése funktor használatakor (folyt.)

- Az `ExtdGeomFun` funktor első változatával, `ExtdGeomFun_1`-gyel több gond van:

```
functor ExtdGeomFun_1
    (structure P : POINT) : GEOMETRY =
struct
    structure Sphere = SphereFun (structure P = Point)
    structure SemiSpace = SemiSpaceFun (structure P = Point)
end
```

  - `ExtdGeomFun_1`-ben alstruktúra-definícióban fordul elő `SphereFun` és `SemiSpaceFun`, és ez olyan paraméterekre korlátozza `ExtdGeomFun_1`-et, amelyek e két funktorral állíthatók elő. – Ez erős korlátozás `ExtdGeomFun`-hoz képest, amely `SPHERE`, ill. `SEMI_SPACE` *bármely* megvalósításának alkalmazását lehetővé teszi.
  - `ExtdGeomFun_1`-nek paraméterként meg kell kapnia komponenseinek közös elemét, ill. elemeit (a példában a `POINT` szignatúrát megvalósító `P` struktúrát). Ez a megoldás kényelmetlenné válik, ha a programunk sok rétegből áll: a „ legtávolabbi” elemtől kezdve a teljes hierarchiát fel építenünk minden alkalommal.
  - Nincs igazi indok arra, hogy `ExtdGeomFun_1` miért éppen `POINT` egy megvalósítását kapja paraméterként. (Nem elég nyomós) oka az, hogy `ExtdGeomFun_1`-nek fel kell építenie az említett hierarchiát a megosztási előírások kielégítéséhez.

A típusmegosztás elkerülése funktor használatakor (folyt.)

- A második lehetőség az, hogy a paraméterként átadott közös elem, azaz `POINT` megvalósításához kötjük `SPHERE`, ill. `SEMI_SPACE` (ugyancsak paraméterként átveendő) megvalósítását, és így érjük el a típusegyenletek teljesülését.
- `ExtdGeomFun_2` alábbi deklarációja kielégíti a követelményeket, de láthatóan nincs semmi előnye a megosztási előírásokat tartalmazó kiinduló változathoz, `ExtdGeomFun`-hoz képest.

```
functor ExtdGeomFun_2
    (structure P : POINT
     structure Sp : SPHERE where Point = P
     structure Ss : SEMI_SPACE where Point = P) =
struct
    structure Sphere = Sp
    structure SemiSpace = Ss
end
```

- Inkább hátránynak tekinthető, hogy egy harmadik paramétert vezettünk be, amelynek az egyetlen szerepe az, hogy elhagyhassuk a megosztási előírást.

A típusmegosztás elkerülése funktor használatakor (folyt.)

- `ExtdGeomFun_2` rafináltabb változata `ExtdGeomFun_3` és `ExtdGeomFun_4`. Mindkettőnek csak két paraméterre van szüksége azáltal, hogy a kettő közül valamelyiket bizonyos értelemben kiemeltük, és előírtuk, hogy a másiknak vele kompatíbilisnek kell lennie.

```
functor ExtdGeomFun_3
    (structure Sp : SPHERE
     structure Ss : SEMI_SPACE where Point = Sp.Point) =
struct
    structure Sphere = Sp
    structure SemiSpace = Ss
end

functor ExtdGeomFun_4
    (structure Ss : SEMI_SPACE
     structure Sp : SPHERE where Point = Ss.Point) =
struct
    structure Sphere = Sp
    structure SemiSpace = Ss
end
```


A típusmegosztás elkerülése funktor használatakor (folyt.)

- E két utóbbi megoldásnak megvannak ugyanazok az előnyei, mint a megosztási előírást alkalmazó megoldásnak. De meg kellett törnünk a megoldás természetes szimmetriáját. Ez mondanivalónk lényege:
- A megosztási előírással a programozó *szimmetrikus helyzetet szimmetrikus módon* oldhat meg.
- A programozó helyett a fordítóprogram töri meg a szimmetriát, amikor ilyen vagy olyan megvalósítást választ. A programozó nem kényszerül arra, hogy önkényes, semmivel alá nem támasztható döntést hozzon.