F-Bounded Polymorphism - Összefoglaló

Rusznák Demeter

Az F-kötött Polymorphism a polimorf tulajdonság típus helyessé tétele generikus függvények esetében, objektum orientált környezetben fordítási időben.
Ezt az F-kötött kvantálás használatával éri el eme tulajdonságot. Ami a gyakorlatban annyit jelent, hogy az $\forall t \subseteq F[t].\sigma$ kifejezésben a típus megkötés nem csak a bemenő paraméterekre vonatkozik, hanem az eredmény típusában is megkötésre kerül.
Ezen módszer használatával biztosítható, hogy az egyes generikusan írt függvények eredményei is az általunk elvárt típusúak legyenek. Hogyha csak a bemenetre teszünk megkötést akkor előállhat olyan eset amelynél az eredmény típusa esetlegesen a bemenetnek valamely altípusa lesz, ezt a fordító nem jelzi mert a bemenet típusa a kötöttségeknek eleget tesz, a kimenet típusa meg elég tág határok között mozog ahhoz, hogy futási időben problémákat okoznak.
Természetesen bizonyos esetekben megírhatók manuálisan az altípusokra a megfelelő ellenőrzések, de ez legtöbbször bonyolult, hosszadalmas, vagy a típusok sokasága miatt "lehetetlen". Ezért is jobb ha már az egyes metódusok definiálásakor már tudunk kikötéseket tenni, így már fordítási időben ellenőrizhető, hogy ténylegesen a nekünk kellő típusúak az eredmények. Így téve egyszerűbbé és biztonságosabbá az ilyen kódok írását.

Elméleti háttér

Objektumok, Rekordok és Rekurzív típusok

Az objektum-orientált programozás leggyakoribb modelljei azon az ötleten alapszanak, miszerint gondoljunk az egyes objektumokra úgy mint olyan rekordokra amelyek komponensei olyan függvények amelyek az egyes metódusokat reprezentálják.
Ebben a modellben egy üzenet küldés nem más mint a megfelelő komponens kiválasztása. Ezen rekord típusokal le lehet írni az objektumok protokolljait vagy interfészeit. Ezek nem mások mint egy olyan hozzárendelés megadása amely címkéket rendel az egyes típusokhoz. Vegyünk egy rekordot amely az $l_1,\ldots, l_j$ cimkékből és az $\sigma_1,\ldots, \sigma_j$ értékekből áll: $\{l_1{:}\sigma_1,\ldots,l_j\ \sigma_j\}$. Ekkor az egyes mezők a mező típusával és visszatérési értékével együtt az egyes üzenetek leírásának tekinthetők.
Még a legegyszerűbb objektum-orientált példák esetében is könnyen belefutottunk rekurzívan definiált rekordokba. Például egy egyszerű síkbéli pont definíciójánál:

Point = {
x:void → Real
y:void→ Real
move: Real x Real → Point

equal: Point → Boolean
}
A Point objektum move metódusa egy ugyan olyan Point típusú értékkel tér vissza mint az eredeti pont, valamint az equal metódus bemeneti paramétereként is megjelenik. Mivel az equal egy bináris operátor a Point típuson ezért p és q pontra a p.equal(q) kifejezés értelmes.
Jelöljük a rekurzívan definiált típusokat a következőképpen: Rec t.A. Ez azt jelenti, hogy Rec t.A egy olyan típus ahol t=A ahol A tartalmazza a t változót. Ezen jelöléssel a Point definíciója a következő:

Point = Rec pnt. {
x: void → Real
y: void → Real
move: Real x Real -> pnt
equal: pnt -> Boolean
}

Mivel a pnt típus változót köti Rec ezért pnt-t átnevezhetjük anélkül, hogy a definíció értelmét megváltoztatnánk.

Rekordok és altípusaik:

Az altípusok bevezetése fontos erős eszköze lehet a polimorfizmusnak. Ennek egyik alaptétele a következő képp írható le:

$\{x_1:\sigma_1,\ldots,x_k:\sigma_k,\ldots,x_l:\sigma_l\} \subseteq \{x_1:\sigma_1,\ldots,x_k:\sigma_k\}$

Az az elképzelés, ha van egy r rekord amely $x_1:\sigma_1,\ldots,x_k:\sigma_k$ és $x_{k+1}:\sigma_{k+1},\ldots,x_l:\sigma_l$ mezőkkel is rendelkezik. Ekkor minden olyan művelet amely működik az $\{x_1:\sigma_1,\ldots,x_k:\sigma_k\}$ típuson, elvárható hogy működjön az $\{x_1:\sigma_1,\ldots,x_k:\sigma_k,\ldots,x_l:\sigma_l\}$ típuson is. Ezen alaptétel általánosítása a típus kikövetkeztetés szabálya.

$$\frac{\sigma 1 \subseteq p1,\ldots,\sigma k \subseteq pk}{\{x1:\sigma 1,\ldots,xk:\sigma k,\ldots,xl:\sigma l\} \subseteq \{x1:p1,\ldots,xk:pk\}}$$

amely figyelembe veszi az egyes mezők altípusait is.
Az általános szabály a függvény altípusosság szabálya:
$(\sigma' \subseteq \sigma\ \tau \subseteq \tau') / (\sigma \to \tau \subseteq \sigma' \to \tau')$
Figyeljük meg, hogy az $\sigma' \subseteq \sigma$ a többinek az ellenkezője, amely azt mondja, hogy a nyíl konstruktor *kontravariáns* az első argumentumában. A rekurzív típusokra vonatkozó egyik altípusos szabály a következő:

$$\frac{\Gamma, s \subseteq t \vdash \sigma \subseteq}{\Gamma \vdash Rec\ s.\ \sigma \subseteq\ Rec\ t.r}$$

Ez nagyjából azt jelenti, ha feltételezzük, hogy , t-nek egy altípusa s akkor abból belátható hogy σ egy altípusa τ-nak, akkor a Rec.s σ rekurzív típus altípusa Rec.t τ-nak
Például az alábbi típus egy szupertípusa a Point típusnak:
Movable = Rec mv {move : Real x Real → mv}

Mert a Point definíciója altípusa a Movable-nek.

Kötött Kvantálás

Ahhoz hogy pontosabban tudjuk modellezni a polimorfizmust az eddig ismertetett rendszerben, be kell vezetnünk a kvantorokat (az exisztenciális és univerzális kvantort) hogy ezen keresztül pontosabb tipizálást tegyünk lehetővé. Ugyanis ezek nélkül az eddigi rendszer nagyon szigorú, azzal hogy explicit megkötjük a változók típusát. - nem lehet általánosítani anélkül hogy minden egyes típusra külön külön meg ne írnánk az adott metódust. a Kvantorok bevezetésével lehetőségünk nyílik ugyanazt a metódust egyszerre több típus felett is definiálni típus változók bevezetésével.
Példa:

```
value  id = all[a] fun(x:a) x
```

Ebben az esetben adott az identitás függvény id, amelynek definíciójában a egy típus változó, és azzal hogy all[a] -t használunk azt jelezzük, hogy eme függvény típustól függetlenül ugyanazt fogja végrehajtani. De ahhoz hogy egy konkrét típusra végrehajtható legyen nekünk kell megadni a bemenő paraméter típusát - az olyan függvényeket amelyekben van típus változó generikus függvénynek nevezzük.
Alkalmazásra példa:

```
SimplePoint = {x:int, y:int}
```

Vegyük észre hogy ez az egyszerű pont nem tartalmaz olyan metódusokta amelyeknek a bemenete, vagy visszatérési értéke egyszerű pont lenne, így nem rekurzív típus. Definiálhatunk egy olyan függvényt ami egyszerű pontokat 'mozgat' Kötött kvantálással. A Move típusa: ∀ t SimplePoint. t → Real x Real → t és a következőképpen definiálhatjuk:

```
move : ∀t ⊆ SimplePoint. t → Real  x Real → t
        = Fun[t ⊆ SimplePoint] fun(p:t) fun(dx,dy:Real) p with {x = p.x +dx, y= p.y+dy}
```

Az Fun[t ⊆ SimplePoint] azt jelzi, hogy a move függvény első argumentuma mindenképpen SimplePoint egy altípusa kell hogy legyen. A második argumentuma egy adott típusú érték. A harmadik argumentum egy számpár amely a mozgatás távját jelöli. Az eredmény úgy keletkezik, hogy egy új rekordot képzünk, amely mezői az eredeti altípus értékeit tartalmazzák, de az x és y komponenseit felül írjuk a mozgatott értékekkel. Ezen keletkezett rekord ugyanolyan típusú mint az eredeti mozgatni kívánt rekord.
Minden a SimplePoint-hoz tartozó altípus behelyettesíthető a move függvény bemeneti értékeinek típusába. Például:

```
SimpleColoredPoint = {x:int, y:int, color:int }
```

egy érvényes argumentuma lesz a move-ak és az eredmény típusa is egy SimpleColoredPoint lesz.

Rekurzív típusok és Kötött Kvantálás

Láthattuk, hogy egyszerű nem rekurzívan definiált típusokra a kötött kvantálás működik és helyes eredményt ad.
Most azt vizsgáljuk mi van akkor ha rekurzív típusokat használunk. Megmutatjuk, hogy a Kötött kvantálás nem elég rugalmas ahhoz hogy sikerrel alkalmazzuk rekurzív típusok esetében.
Két nagyobb probléma adódik, attól függően, hogy a rekurzív típusban hol helyezkedik el a rekurzív változó. Ahhoz, hogy ezekről beszélhessünk, szükségünk lesz a logikából ismert polaritásra. Az $\sigma \to \tau$ kifejezésben a τ rész kifejezés pozitívan szerepel, a σ pedig negatívan. Ha egy σ' kifejezés valamilyen polaritással szerepel a σ akkor ellenkező polaritással fog szerepelni a $\sigma \to \tau$ kifejezésben.
De a τ részkifejezései megtartják polaritásukat. Példa: t pozitív a $(t \to \sigma) \to \tau$ -ban, de negatív a $t \to (\rho \to \sigma)$ kifejezésben. A rekord típusú kifejezésekben a polaritás megőrződik: szóval egy {put: $t \to$ void, get: void $\to$ s} kifejezésben t pozitív, s pedig negatív.

Pozitív rekurzió

Amikor a rekurziós típusban szereplő rekurzív változó pozitív, akkor az altipizálás nem garantálja, az általunk elvárt típust. Vegyük az fentebb rekurzívan definiált Movable típust. Az mv változó csak pozitívan szerepel a Movable definíciójában.

Movable = Rec mv {move: Real $\times$ Real $\to$ mv}

Ennek felhasználásával definiálhatunk egy olyan függvényt (translate) amely egy Movable típusú értéket mindkét irányban tud mozgatni.

translate = fun(x: Movable) x.move(1.0, 1.0)

Bár ez minden olyan értékre működik amely a Movable típusnak altípusa, de a végeredmény típusa mindig Movable típusú lesz, tehát nem feltétlen egyezik meg a bemenetként szolgáló érték tényleges típusával. Ezért jobb lenne ha lenne egy olyan translate függvényünk amely ha kap egy t típust amely a Movable-nek egy tetszőleges altípusa, akkor a kimenet ugyanolyan t típusú értéket ad vissza.Megmutatható, hogy amennyiben a translate mostani definícióját használjuk, az nem így viselkedik:

R = {move: Real $\times$ Real $\to$ Movable, other: A}

Könnyű belátni, hogy R ténylegesen a Movable egyik altípusa, de ha alkalmazzuk rá a translate függvényt akkor egy Movable típusú értéket kapunk, nem pedig R típusút mint ahogy elvárnánk. Ennek alapján elmondhatjuk, hogy a legbiztosabb amit állíthatunk ha kötött

kvantálást használunk az ∀r ⊆ Movable. r → Movable. Tehát nem jutottunk közelebb a megoldáshoz.
Igaz ezt a translate függvényt kvantálás nélkül is lehetne definiálni, de ez csak azért működik mert az x változó csak egyszer szerepel a definícióban. De ennél bonyolultabb típusok esetében ez már nem lenne elég:

choose = fun(b:bool) fun(x:Movable) if b then x.move(1.0, 1.0) else x.

Negatív rekurzió

Azon rekurzív típusokból ahol a rekurziós változó negatívan szerepel, nem tudunk az eddigi módon , csak mezők hozzáadásával altípusokat gyártani, mert a keletkezett típusok nem állnak altípusossági viszonyban az eredeti típussal. Ennek következménye, hogy a kötött kvantálás nem alkalmazható ezen típusokra. Legyen egy PartialOrder típusunk amely a következőképpen néz ki:

PartialOrder = Rec po {lesseq: po → bool }

Definiáljunk ezen típus felett egy polimorf minimum függvényt:

minimum ∀t ⊆ PartialOrder. t → t → t

Ezen minimum függvény visszaadja a két bemenetként kapott érték közül a kisebbet, úgy hogy az elsőt összeveti a másodikként kapottal. Feltételezhetnénk hogy Number és String típusú bemenetekre működnie kéne eme függvénynek, mivel mind kettőnek van lesseq művelete. Tekintsük a Number-t mint rekurzívan definiált típust:

Number = Rec num. {…, lesseq: num → bool,…}

Ennek ellenére minimum[Number] típus helytelen, mert a Number nem altípusa a PartialOrder típusnak. Ha megpróbáljuk leszármaztatni a Number típust a PartialOrder típusból, úgy hogy kicsomagoljuk az egyes típusokat azt kapjuk hogy:

{…,lesseq: Number → bool,…} ⊆ {lesseq: PartialOrder → bool}

Amelyből azt az eredményt kapjuk, hogy PartialOrder ⊆ Number. Ez az ellenkezője annak amit meg akartunk mutatni, ami azt mutatja, hogy Number ⊆ PartialOrder nem származtatható hacsak Number = PartialOrder.
Egy, a PartialOrderhez tartozó típus:

Rec t. {…,lesseq: PartialOrder → bool,…}

Ezen típusba tartozó értékeket össze tudjuk hasonlítani más a PartialOrder-hez tartozó értékekkel, de mivel a PartialOrder típus nem rendelkezik olyan mezőkkel amelyen egy ilyen összehasonlítást el tudnánk végezni, ezért nem túl hasznos. Azokban az esetekben ahol

léteznek további mezők, végre tudjuk hajtani az összehasonlítást, de továbbra sem tudjuk a minimum függvényünket, az általunk szeretett polimorfizmus-nak megfelelően alkalmazni.

F-kötött kvantálás

Az F-kötött kvantálás lehetővé teszi hogy a fenti példák esetében a kimenet típusa biztosan a kívánt típusú legyen típus ellenőrzéssel együtt. Azt mondjuk hogy egy univerzálisan kvantált típus F-kötött ha felírható a következő alakban::

$\forall t \subseteq F[t].\sigma$

ahol F[t] egy olyan kifejezés tartalmazza t változót.
Az F-kötött polimorf típusok abban különböznek az eddigi kötött típusoktól, hogy két helyen köti meg a típus változót: egyszer magában a típus megkötésban F[t]-ben, ahogy eddig, és még egyszer az eredmény típus σ-ban. Ha F[t] egy típus amelyre igaz: F[t] = {ai:σi[t]} akkor az A ⊆ F[A] feltétel azt mondja hogy A-nak rendelkeznie kell az ai metódusokkal, és ezen metódusok argumentumainak meg kell felelniük az σi[A] kötésnek, amelyek az A függvényében vannak definiálva. A a legtöbb esetben egy rekurzív típus lesz, amely azt mutatja, hogy az F-kötés és a rekurzív típusok szorosan kapcsolódnak egymáshoz.
De az $\forall t \subseteq (\text{Rec } r.F[r])\sigma(t)$ ként rekurzívan definiált kötött típus, és az $\forall t \subseteq F[t]\sigma(t)$ típus nagyon nem ugyan az.

Pozitív rekurzió

Ahogy azt korábban láthattuk, amikor polimorf ként próbáltuk kiterjeszteni a translate metódust pontokra: translate[Point] az egy Point → Movable és nem Point → Point típusú függvényt eredményezett. Azért definiáltuk az F-kötött kvantálást mert szeretnénk egy egyszerű módot találni arra hogy hasonló esetekben a kívánt típust kapjuk.

A translate példáján keresztül a megoldástól, visszafelé haladva, jutunk el az F-kötés szükségességéhez. A probléma a következő: szeretnénk egy olyan feltételt találni tetszőleges t típus esetén, hogy bármely x értékre amely t típusú, alkalmazva a x.move(1.0,1.0) metódust t típusú értéket kapjunk vissza. Egy olyan elégséges feltételt keresünk t típusra, hogy a keresett típus származtatható legyen:

$x{:}t \vdash x.\text{move}(1.0,1.0) : t$

Alkalmazva az (APP)[1] és (SEL) tipizálási szabályokat a fenti állítás a következőre redukálódik:

$x : t \vdash \{\text{move} : \text{Real} \times \text{Real} \to t\}$

az altípizálás szabályát használva származtatható:

[1] Az (APP) és (SEL) szabályok valamint az egyéb tipizálási szabályok megtalálhatók:
Luca Cardelli. Structural subtyping and the notion of power type.
John C. Mitchell. Type inference and type containment.

$$\frac{\subseteq \{move: Real \times Real \to t\}}{x{:}t \vdash x{:}}$$

Mivel a τ típus máshol nem szerepel ezért egyszerűsíthetünk:
t ⊆ {move: Real × Real → t}
Amit nem tudunk bizonyítani további feltételek nélkül. Ezt a feltételt t ⊆ F-Movable[t] -vel fejezzük ki ahol

F-Movable[t] = {move: Real × Real → t}

Ez az alak már megfelel az F-kötött kvantálásnak. Erre alapozva definiálhatunk egy F-kötöt polimorfikus függvényt:

translate = Fun[t ⊆F-Movable[t]] fun(x:t) x.move(1.0,1.0)

Amelynek a típusa, egy F-kötött polimorf típus::

translate: ∀t ⊆ F-Movable[t]. t → t

Mivel a Point típusra igaz hogy Point ⊆ F-Movable[Point], ezért így már a translate[Point] típus helyes lesz, és Point → Point típusú függvényt fog eredményezni. Természetesen az így tipizált translate minden olyan típusra működni fog amelyre igaz hogy t ⊆ F-movable[t] mint például a ColoredPoint:

```
ColoredPoint = Rec pnt {
  x:void → Real
 y: void → Real
 c: void → Color
move: Real × Real → pnt
}
```

Érdemes még megemlíteni hogy az F-Movable típus függvény kapcsolódik ahhoz a rekurzív típushoz amellyel nem tudtuk a megfelelő típust előállítani. F-Movable-t úgy konstruáltuk, hogy figyelembe vettük a rekurzív típus kifejezést mint típus függvényt.

Negatív rekurzió

Előzőleg láthattuk, hogy a Number típus nem altípusa a PartialOrder típusnak. Ezen felül a Number , a String és a PartialOrder-nek is van egy olyan művelete, hogy lesseq. Így minden olyan esetben ahol x és y azonos típusba tartoznak az x.lesseq(y) értelmes és típus helyes, akkor azonban nem ha x és y különböző típushoz tartoznak. Viszont csak Kötött-kvantálással nem tudtuk elérni, hogy egy polimorf minimum függvény megfelelően működjön mind a Number, String és PartialOrder típusokra. Most megmutatjuk hogy az F-kötött kvantálás lehetővé teszi egy típus helyes polimorf minimum függvény definiálását.

ami azért fontos előrelépés mert az eddig létező megoldások nagyon erősen korlátozzák a bináris műveleteket.

A közös strukturális elemet a PartialOrder a Number és a String típusok között a a PartialOrder rekurzív definíciójából származó tipus függvény adja meg:

F-PartialOrder[t] = {lesseq: t → bool }.

Ezt alkalmazva a Number típusra:

F-PartialOrder[Number] = {lesseq: Number → bool}

ebből következik:

Number ⊆ F-PartialOrder[Number]

Azt tudjuk, hogy a Number típus nem áll altípusossági kapcsolatban a PartialOrder típussal, viszont az F-PartialOrder[Number] típissal már igen, és így már elő tudjuk állítani azt a minimum függvényt amire szükségünk volt:

minimum = Fun[t ⊆ F-PartialOrder[t]] fun(x:t) fun(y:t) if x.lesseq(y) then x else y

amelynek a típusa:

minimum = Fun[t ⊆ F-PartialOrder[t]]. t → t → t

Ezzel sikerült a polimorfizmus egy olyan tulajdonságát amelyet eddig nem sikerült modellezni a kötött kvantálással. Bár a Pozitív és Negatív polaritást külön néztük, de az F-kötött kvantálás esetén mindkét estben sikerrel levezethető a kívánt típus. Az F-kötött kvantálás egy olyan típust jellemez amelynek a struktúrája rekurzív, hasonló a Rec t.F[t] típushoz. Azt mondhatjuk hogy F[A] olyan értelmes műveleteket fog magába amelyek az A típusú értékeken végrehajthatóak, vagy A típusú értékellel térnek vissza. Az A-ba tartozó típusok mind rendelkeznek ezen műveletekkel mert minden elemére igaz hogy A ⊆ F[A]. Az A = Rec t.F[t] rekurzív típus mindig kielégíti az A ⊆ F[A] kifejezést. Általánosabban: ha adott egy G[t] típus kifejezés, és G[t] ⊆ F[t] minden t esetén, akkor A = Rec.tG[t] típus szintén kielégíti A ⊆ F[A]-t. Ez abból következik, hogy ∀t G[t] ⊆ F[t] akkor A = G[A] ⊆ F[A]. Érdemes még megjegyezni, hogy F-től függően lehetnek más típusok is amelyek kielégítik a t ⊆ F[t] kifejezést, olyanok is amelyek nem kifejezetten ilyen formájúak.
Ezt észben tartva elmondhatjuk, itt külön válik az altípusosság és az öröklés objektum orientált környezetben, mivel létezik olyan t1 és t2 típusok amelyek kielégítenek egy F-kötést, de nem állnak altípus kapcsolatban (nem igaz hogy t1 ⊆ t2-t vagy hogy t2 ⊆ t1-t), hanem valamilyen közös strukturális elemnél teremti meg a kapcsolatot.

“Anyagok” és “Formák”

Most, hogy már tudjuk miért kell és miért jó az F-kötött polimorfizmus, vannak esetek ritka esetek amikor más is kell a sikeres típusok kalkulálásához:

Példaként hozhatunk egyes bináris metódusokat amelyek (pl összeadás vagy az összehasonlítás) megkövetelik hogy a bemenő paraméterek ugyanolyan típusú legyenek, és az F-kötéssel ez elérhető. Így az egyes típusok között nem lép fel olyan probléma, hogy két különböző típusú értéket akarunk összehasonlítani, csak mert megvan mindkettőban ugyan az a metódus. A Probléma abból származik, hogy hogy teszünk különbséget az egyes típusok között. Például Java esetében az egyenlőséget objektum szintem határozták meg, és mivel minden osztály az objektum osztályba tartozik, ezért nem lehet azonnal eldönteni hogy biztosan végrehajtható-e az adott művelet hanem először castolnunk kell a bemeneti paramétereket a megfelelő típussal.

A fő gond az F-kötöttséggel az, hogy nagyon támaszkodik a rekurzívitásra.. Például egy String ami implementálja a Comparable<String>, akkor az örökölt típus az öröklő típus függvényében van definiálva. Ez jó de a gyakorlatban a generikus típusok esetében előfordulhat variancia. Például egy List<Stringet> kezelhetjük úgy mint List<Objectet> ezért néhány nyelv megengedi hogy a List-et covariant deklarálják.ennek duálisaként az olyan dolgokat amelyeket tetszőleges objektumokkal összehaszonlíthatunk akár integer ekkel is, szóval az összehasonlító művelet (Comparable) contravariant. Viszont ha kombináljuk a varianciát és a rekurzív öröklést olyan komplex tipizálási szabályokat kaphatunk ahol nem minden esetben eldönthető, hogy az egyes objektumok mely altípushoz is tartoznak.
Ahhoz, hogy ismét visszanyerjük az eldönthetőséget, korlátozni fogjuk a rekurzív öröklést, oly módon ami a gyakorlatban már egyébként is jelen van.
Bevezetjük az anyagok (materials) és formák (shapes) elnevezéseket. Azon osztályokat és interfészeket amelyek felhasználásra kerülnek rekurzív öröklésnél (mint pl a Comparable) formáknak nevezzük, azért mert egy magasabb rendű tulajdonságot írnak le a rekurzív öröklés segítségével. A tapasztalat azt mutatja hogy az ebbe a csoportba tartozó osztályok és interfészek szinte soha nem szerepelnek paraméter típus, return típus, mezőtípus vagy argumentum típusként. Minden más osztály és interfész anyagnak minősül, és ezek azok az osztályok amelyek a fent említett esetekben a típusokat adják. Ha ez a két csoport diszjunkt akkor (és a felmérés azt mutatta, hogy egy két kivételtől eltekintve amelyek amúgy tervezési hibából adódtak) akkor a típus eldöntése sokkal egyszerűbbé válik, és definiálni lehet egy olyan típus rendszert amely megtartja az eldönthetőséget.
Ezt Material-Shape Separation-nak nevezzük.

Típus következtetés:
Tekintsük a:
Class Tree extends ArrayList<Tree>{}

Listát, ami egy mutable címkézetlen fát implementál. Azzal hogy az ArrayList implementálja a Lista típust, azzal kapunk egy helyes egyenlőségi implementációt fákra. De ha megpróbálunk összehasonlítani két fát akkor a fordító StackOverflowError-t dob. Az hogy miét okoz nehézséget a típusellenőrzőnek a típus kikövetkeztetése, a generikus programozas egyik nagy kihívása. Ahhoz hogy az egyenlőséget ellenőrizni tudja a típus ellenőrzőnek először ellenőriznie kell hogy a bal oldalon álló valami a jobb oldalon álló valamivel megegyező egyenlőséget implementál-e. Ebben az esetben ez arra vezet hogy ellenőrizni kell hogy a Tree altípusa-e az Equitable<Tree> típusnak. Ennek eredményeképp keletkezik az alábbi végtelen típus redukció:

$$
\begin{array}{r}
\texttt{Tree} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{ArrayList<Tree>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{List<Tree>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{Equatable<List<Equatable<Tree>>>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{contravariance}) \\
\texttt{Tree} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{ArrayList<Tree>} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{List<Tree>} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{covariance}) \\
\texttt{Tree} <: \texttt{Equatable<Tree>} \\
\vdots
\end{array}
$$

Amint azt látjuk az első lépés ugyanaz mint az utolsó, tehát egy végtelen ciklust kaptunk. A meglepő dolog az egészben hogy minden lépés érvényes, így egy érvényes ellenőrzést kaptunk, és ezen végteéen ciklusok miatt válik a típusok kikövetkeztetése nehézzé, mert bár az algoritmus minden lépése helyes, soha nem fog a végére érni.
Ha viszont alkalmazzuk a fent említett Material-Shape Separation akkor belátható hogy minden altípus bizonyítás véges, és biztosan megáll.

Alternatív megoldás: Virtuális Típusok

Az F-kötés mellett a egy másik lehetőség a polimorfizmus megvalósítására a Virtuális típusok bevezetése. Itt a típus változók nem mint osztály paraméterek kerülnek átadásra, hanem mint az osztálynak egy attribútuma. A virtuális jelző azt jelenti, hogy mint a virtuális metódusok esetében, felüldefiniálhatók a leszármazottakban. A virtuális típusok bevezetésével nagyjából ugyan úgy lehet polimorfikus viselkedést elérni mint a parametrikus esetben, és például a varianciát jobban kezeli mint az F-kötött eset. Ugyanakkor vannak szituációk amit virtuális típussal nehéz vagy éppen lehetetlen lenne kifejezni.

A parametrikus esetben legtöbbször létezik egy, a strukturából származó, kapcsolat a generikus osztály és annak specializációja között, és emiatt a kapcsolat miatt lehet F-kötést alkalmazni. míg a virtuális típusok esetén ez nincs meg, így abban az esetekben amikor definiálni lehet egy F-kötött szerkezetet virtuális típusokkal, sok esetben, egy redundáns, hosszabb kódot kapunk.
A virtuális típusok, pont mert a típus változók attribútumok ként vannak definiálva, ugyanis az egyszer rögzített típus, kívülről hozzáférhető, így a varianciából adódó típus következtetéseket végre tudják hajtani, ugyanis csak ki kell olvasniuk az a megfelelő attribútumot.

Összefoglalás

Ebben az összefoglalóban az F-kötött polimorfizmus-ról volt szó. Amit azért találtak ki hogy objektum orientált környezetben írhassak olyan generikus függvényeket, interfészeket, amelyek megfelelő tipizálása fordítási időben eldönthető.
Szó volt arról hogyan kezdték modellezni az objektum orientált nyelveket, (rekordok, λ-kalkulus) majd hogyan bővült a modell a követelményeknek megfelelően amikor is szerettünk volna a generikus tulajdonságot bevezetni (kötött kvantálás) és, hogy miért nem volt elég az így kapott eszköz készlet. Míg végül megérkeztünk Az F-kötött polimorfizmus-hoz amely azáltal hogy lehetővé teszi a definiálni kívánt valami, felhasználását mint kötési kritériumot, ezzel megadja a hiányzó információt ahhoz hogy sikeres típuskövetkeztetést hajtsunk végre.
A végén szó volt arról hogy az F-kötés a gyakorlatban milyen problémákat vet fel, valamint egy javaslat arra, hogy hogyan lehet javítani az F-kötés néhány hiányosságát a típusok kikövetkeztetésénél, az anyagok és formák bevezetésével. Majd röviden megemlítettük a virtuális típusokat mint alternatívát, amennyiben generikus kódok típus helyességét szeretnénk ellenőrizni fordítási időben. A virtuális típusoknál az típus és altípus közötti kapcsolat nem a struktúrából származik, igy vannak olyan esetek amikor F-kötéssel egyszerűbb “megfogalmazni” amit szeretnénk.Éppen ezért születtek tanulmányok a két technika összevonására, amelyben a két rendszer erősségeit ötvözik, de amennyire én láttam nem jutott el a gyakorlati megvalósításig.

Felhasznált irodalom:

1:Luca Cardelli, Peter Wegner - On Understanding Types,Data Abstraction, and Polymorphism
2:Peter Canning, William Cook, Walter Hill, Walter Oltho, John C. Mitchell - F-Bounded Polymorphism for Object-Oriented Programming
3:Ben Greenman, Fabian Muehlboeck, Ross Tate - Getting F-Bounded Polymorphism into Shape
4:Kresten Krab Thorup, and Mads Torgersen - Unifying Genericity Combining the Benefits of Virtual Types and Parameterized Classes

5: Kim B. Bruce, Martin Odersky, and Philip Wadler - A Statically Safe Alternative to Virtual Types
6: Kresten Krab Thorup - Genericity in Java with Virtual Types