zh-elmélet-1

Határidő jan 9, 18:00 **Pont** 50 **Kérdések** 5
Elérhető jan 9, 16:00 - jan 9, 18:00 körülbelül 2 óra **Időkorlát** 120 perc

Instrukciók

Válaszolj minden kérdésre legalább 1000-2000 karakter terjedelemben! A vizsga ezen részéhez semmilyen (írott, nyomtatott, digitális, illetve hálózaton elérhető) segédanyagot nem lehet használni, saját kútfőből kell a válaszokat megfogalmazni.

Elégségest 20 ponttól,

közepest 25 ponttól,

jót 33 ponttól,

jelest 40 ponttól adok.

Ezt a kvízt ekkor zárolták: jan 9, 18:00.

Próbálkozások naplója

	Próbálkozás	Idő	Eredmény
LEGUTOLSÓ	1. próbálkozás	74 perc	40 az összesen elérhető 50 pontból

Ezen kvíz eredménye: **40** az összesen elérhető 50 pontból
Beadva ekkor: jan 9, 17:22
Ez a próbálkozás ennyi időt vett igénybe: 74 perc

1. kérdés — 10 / 10 pont

Mik az altípusosság szabályai függvénytípusokra?

A beküldött megoldás:

Legyen A , A', B, B' típusok és tekintsük az A -> B és A' -> B' függvényeket. A <: reláció pedig jelölje az altípusosságot, úgy, hogy A <: B jelentése, hogy "A altípusa vagy leszármazottja B-nek"

Ekkor, ha A <: A' és B' <: B, akkor az A->B <: A' -> B', azaz a paraméterre kontravariancát követelünk meg (Az alítpus paramétere csak általánosabb lehet, mint a bázis típusé), a visszatérési értékre pedig kovariancát engedünk meg (azaz az alítpus visszatérési értéke lehet speciálisabb, mint a bázis típusé). Ezen szabállyal a Liskov helyettesítési elv teljesül

Ez azonban csak a matematikai definíció, a valóságban nem ez a reláció áll fenn, a paramaméterre is megengedjük a kovarianciát (eiffelben), más nyelvekben pedig általában invarianciát követelünk meg (pl. C#-ban lehet kontravariáns paramétert is megkövetelni genericekkel).
A kovariáns paraméter előnye a rugalmasság, a like Current konstrukcióval eiffelben adaptívan tudjuk a paraméter típusát változtatni, így egy altípusban tudjuk hivatkozni az altípus feature-jeit is.
Hátránya, hogy CAT problémához vezethet egy polimorf referencia esetén.
Pl:

class FOOD (...)
class MILK inherit FOOD (...)
class GRASS inherit FOOD (...)

class Animal (...) feature {ANY} feed (food: FOOD) (...) end

class CAT (...) feature {ANY} feed (food: MILK) (...) end

ekkor, ha deklarálunk egy Animal referenciát
animal: ANIMAL (...)
grass: FOOD
{GRASS}grass.make()
{CAT}animal.make()
animal.feed(grass)

Ekkor a fordító nem fog semmilyen hibát jelezni, mindazonáltal futásidőben hibát fogunk kapni, ugyanis a CAT típus egy MILK-et várna

2. kérdés

7 / 10 pont

Hogyan szabályozza az Eiffel a láthatóságot? Miben különbözik ez a más nyelvekben megszokottól? (Itt sok mindenről lehet beszélni! Ez talán a legnagyobb terjedelmű választ igénylő kérdés ebben a feladatsorban!)

A beküldött megoldás:

Az eiffelben a láthatóság 3 féle módon szabályozható:

A láthatóság szabályozása több kontextusban is előfordulhat, mindazonáltal 3 alapesetre bontható a szabályozás mechanizmusa

Vegyük B deklarált típust, mely altípusa A típusnak

tehát class B inherit A

Ekkor beszélhetünk a láthatóság szabályozásáról feature klózokban:

- feature {NONE}: privát láthatóság, az itt definiált featureök (attribútumok, függvények, eljárások és azok speciális esetei) csak B számára érhetőek el, az őt változóként, vagy attribútumként hivatkozó osztályok nem. Fontos megjegyezni, hogy a privát láthatóság esetén csak az objektum láthatja belső állapotát, tehát más nyelvekkel ellentétben osztályon belüli featureben sem érhető el privát feature

-feature {B}: titkos láthatóság, más nyelvekben a protected kulcsszó működését modellezi, az ezen klózban definiált featureök csak B és B altípusai számára láthatóak

-feature {ANY}: publikus láthatóság, ebben a klózban definiált featureök elérhető bármely objektum számára, mely altípusa B-nek, vagy B típusú változót, paramétert, attribútumot használ

Valójában a 3 öröklődési mód ugyanazon mechanizmus 3 specializált esete, ugyanis a {} szekcióban lévő rész azt írja le, hogy mely típus altípusai számára érhetőek el a klózban definiált tagok, az ANY típusnak pedig minden típus altípusa, NONE-nak pedig semelyik típus sem altípusa

Ezen mechanizmus megfeleltethető a C++, C#, vagy java láthatósági eseteinek

Az öröklődés kontextusában szintén ezen három esetet különböztethetjük meg

- inherit A {NONE} : privát öröklődés, ekkor a megörökölt attribútumok, eljárások és függvények csak a deklaráló típus számára láthatóak

- inherit A {B}: titkos öröklődés, a protected kulcsszónak feleltethető meg, A tagjai csak B és B altípusai számára láthatóak

-inherit A {ANY}: publikus öröklődés, A tagjai bármely A-t használó, típus számára, vagy A altípusai számára láthatóak.

Ilyen mechanizmus a C++-ban érhető még el, ott private, protected és publikus öröklődésként érhető el, C#, vagy Java esetében ilyen mechanizmusra nincs lehetőség, csak publikus öröklődésre

Ezen kívül az öröklődés esetében vannak további lehetőségek A láthatóság szabályozására. Az export klózban meg lehet változtatni a láthatóságát bármely featurenek egymástól függetlenül. Fontos, hogy a láthatóság növelhető és csökkenthető is, tehát A-ban {ANY} láthatóságú featureök B-ben minden további nélkül lehet {NONE} láthatóságú. Az export klózban egyesével is megnevezhetjük a featureöket, vagy használhatjuk az "all" kulcsszót, mellyel minden nem megnevezett feature láthatóságát szabályozzuk

PI:

```
class A
feature {ANY} foo: INTEGER do Result := 0 end
feature {A} bar: BOOLEAN do Result := false end

class B
inherit A
        export {A} foo
        export {ANY} all
(...)
```

Hasonló mechanizmusra van lehetőség más nyelvekben, azonban a láthatóság csak bővíthető minden esetben, akár a java-ban, ahol pl egy protected metódus az altípusban overrideolva lehet már publikus, vagy C#-ban a new kulcsszó ad ugyanilyen lehetőséget.

Az oka, hogy más nyelvekben nincs lehetőség a láthatóság csökkentésének, hogy CAT problémához vezethet, egy polimorf referencia esetén nem tudhatjuk fordítási időben, hogy egy altípus csökkentette a láthatóságot és futási időben hibát kapunk, hogy a bázistípusban publikusként deklarált, de az altípusban már pl privateként exportált featuret akarjuk meghívni

3. kérdés

10 / 10 pont

Mik a ciklus konstrukció szerződés-összetevői? Mikor felel meg a ciklus a szerződésének?

A beküldött megoldás:

A ciklus szerződés összetevői:

from {INIT}

invariant {INV}

variant {VAR}

until {COND}

(do {BODY}, ez nem tartozik feltélen a szerződés összetevők közé, de {BODY}-ként hivatkozni fogom később}

A from egy inicializációs lépés, ahol beállítjuk a ciklus által felhasznált változókat kezdeti értékükre. A from klóz végén az invariánsnak teljesülnie kell.
Hoare hármassal a következőképp formalizálhatjuk: {true} INIT {INV}

Az invariant klózban írhatjuk le, hogy a ciklusmagnak minden lépés során milyen tulajdonságokat kell megőriznie. A ciklusmag végrehajtása során az invariáns "elromolhat", mindazonáltal a ciklusmag végére igaznak kell lennie Hoare hármassal: {INV ^ (not COND)} BODY {INV}

A variant klózban definiálnunk kell egy variáns függvényt, aminek az értéke szigorúan monoton csökkenő kell legyen és értéke csak pozitív lehet (vagy nulla)

Hoare hármasokkal: {INV ^ NOT COND} BODY {VAR >= 0}
illetve minden v-re : {INV ^ NOT COND ^ VAR >= 0} BODY { v > v-1} (ez a levezetési szabály nem teljesen pontos, de ez írja le, hogy a ciklusmag végrehajtása után a variáns függvény értéke legalább 1-el csökken)

Az until klóz a ciklus kilépési feltételét határozza meg, ha ennek kiértékelt értéke igaz, akkor a ciklus végrehajtása leáll. Az invariánsnak ekkor is teljesülnie kell, ideális esetben a variáns függvény ekkor veszi fel a 0 értéket

A hoare hármasokkal leírt állítások közvetlen összeköttetésben áll a ciklus 5 levezetési szabályával, azzal a kivétellel, hogy nincs megkötés elő és utófeltételre, ezért egy levezetési szabály nincs feltűntetve, illetve az előfeltétel szimplán true a from klózban, tehát a ciklus akkor felel meg a szerződésnek, ha a levezetési szabályokat teljesíti

4. kérdés

6 / 10 pont

Mik az expandált típusok, miben rejlik a specialitásuk?

A beküldött megoldás:

Az expandált típusok a C#-ban megszokott "value type"-oknak megfelelő típuskonstrukció.

Az expandált típusok a referencia típusokkal ellentétben nem a heapben kerülnek allokálásra, hanem a stacken, így közvetlenül elérhetőek, nincs szükség dereferálásra felhasználásukkor. Ugyanebből fakadóan, hogy nem referencián keresztül érhetőek el, hanem a program közvetlen memóriaterületéről, nem lehet értékük void.
Ugyanezen tulajdonságukból következik, hogy felszabadításuk a határkörön kívülre kerülve automatikusan megtörténik, nincs szükség szemétgyűjtésre a felszabadításukhoz.

Származtatáskor lehet belőlük származtatni, de csak {NONE} láthatósággal, így altípusosság nem jön létre expandált típusból származtatva, mindazonáltal az expandált típusok származhatnak referenciatípusokból, így az expandált típusok lehetnek altípusok

5. kérdés

7 / 10 pont

Mit értünk sekély-, illetve mély másoláson?

A beküldött megoldás:

Sekély másoláskor a régi objektum összes adattagját sekélyen másoljuk az új objektumba. Amennyiben az adattag referencia típusú, akkor a referenciát állítjuk be, amennyiben expandált típusú, akkor az értékét másoljuk át. A sekély másolás eiffelben a standard_copy (frozen) feature-el használható, végrehajtása után a standard_equals igaz lesz az új és régi objektum között.

Mély másoláskor a régi objektum összes adattagját mélyen másoljuk az új objektumba, itt minden esetben minden adattagot érték szerint másolunk az új objektumba, akár referencia, akár expandált típusú. Eiffelben a deep_copy (frozen) feature-el érhető el, végrehajtása után a deep_equals igaz lesz a régi és új objektum között

Kvízeredmény: **40** az összesen elérhető 50 pontból