Typové systémy O-O programovacích jazyků

Tomáš Burger

Úvod – situace typových systémů objektově orientovaných jazyků

Standardní kompilované programovací jazyky se snaží během kompilace, kdy se provádí systematická inspekce programového textu, provést i některé kontroly – snaží se tak odhalit některé programátorské chyby, které lze identifikovat jen na základě programového textu. Tradiční takovou chybou jsou „typové kolize“, kdy programový text kombinuje entity různých typů nedovoleným způsobem.

Typickým příkladem z běžných programovacích jazyků je přiřazování řetězců do proměnných číselného typu a podobně. Klasické „před-objektové“ programovací jazyky definují relativně malý soubor takovýchto pravidel a proto typová kontrola nepředstavuje zásadní problém.

Situace se ale komplikuje, pokud programovací jazyk obsahuje objektově-orientované konstrukce; co se typové kontroly týče, klíčovou roli hraje vlastnost objektově orientovaných systémů, zvaná polymorfismum.

Polymorfismus znamená, že entita určitého typu může přijímat hodnoty nikoliv jen stejného typu, ale i všech „podtypů“. Právě termín „podtyp“, a co všechno může být za podtyp považováno, je základním problémem typových systémů objektově-orientovaných programovacích jazyků.

Zejména budeme studovat dvě metody, které umožňují konstrukci podtypu: dědičnost a genericitu.

Dědičnost je vztah mezi typy, kdy potomek (podtyp) zdědí typ svého předka (nadtypu) a následně může některé vybrané elementy typu předefinovat. – Možnosti předefinovávaní jsou omezeny právě požadavkem, aby podtyp byl nadále kompatibilní se svým typem-předkem. Objektově-orientované programovací jazyky kombinují dědičnost typů s dědičností tříd (třída je základní konstrukce objektově-orientovaných jazyků, která definuje programový kód i typ objektu) – u objektů pak používáme terminologii teorie množin a mluvíme o podtřídách, nadtřídách, inkluzi a o objektech, které jsou prvkem určitých tříd; je-li objekt prvkem určité třídy, je prvkem i všech jejích nadtříd. Dědičnost pak definuje v rámci výpočetního systému algebraickou strukturu svazu.

Genericita je možnost programovacího jazyka definovat místo typů vlastně jen „vzory typů“, kde určité typy, použité v definici typu, jsou vyvedeny vně definice jako parametry a jsou určeny později. Základním užitím genericity jsou třídy kontejnerů, které jsou určené k udržování skupin objektů určitého typu, například vzor třídy Seznam je definován vlastně jako Seznam[G], kde G je typ objektů, které mohou být do seznamu vloženy (kouzlo genericity vynikne pak v kombinaci s dědičností, kdy do seznamu mohou být vloženy nejen objekty typu G, ale i objekty všech možných podtypů typu G). Konkrétní typ, použitelný v textu programovacího jazyka, pak vzniká, když G nahradíme skutečným existujícím typem. Předmětem našeho zkoumání bude, jaký můžeme očekávat vztah typů Seznam[G] a Seznam[H], pokud známe vztah typů G a H (například jako že typ G je podtypem typu H).

Standardní programovací jazyky

Nejprve se podívejme na standardní kompilované objektově-orientované programovací jazyky, které jsou používány praktickými programátory. – Jako zástupce si vezměme programovací jazyk Java, který je navržený relativně nedávno (alespoň ve srovnání s ostatními jazyky této kategorie jako C++ nebo tradiční objektové klony Pascalu) a lze ho tedy považovat za relativně moderní.

Java disponuje samozřejmě dědičností – a to jak dědičností typů (zvaných v Javě interface), tak dědičností tříd. Dědičnost typů je násobná (jeden typ může mít více předků), dědičnost tříd proti tomu jednoduchá (jedna třída může mít jen jednu nadtřídu), třída ale kromě jednoduché „třídové“

dědičnosti může dědit i z několika dalších typů. Potomci v Javě nemohou měnit zděděný typ, mohou pouze přidávat nové prvky a případně předefinovat zděděný kód tříd. Zajímavou vzhledem k další diskusi je vlastnost, že je-li nějaká třída podtypem určitého interfacu, pak i podtřídy této třídy jsou podtypy tohoto interfacu. Přestože se jedná o velmi přirozenou vlastnost, její popření se nabídne později jako východisko z některých typových problémů.

Polymorfismus v Javě je tak relativně přímočarý: entita určitého typu může přijímat i hodnoty všech svých podtypů; a protože podtyp může obsahovat jen „cosi navíc“ – a to „navíc“ zůstane v „polymorfní situaci“ neviditelné – nemůže nastat vlastně žádná kolize.

Nic zajímavějšího a složitějšího vlastně tradiční objektově-orientované kompilované programovací jazyky nenabízejí. C++ sice nabízí genericitu, ale bez jakéhokoliv vztahu k polymorfismu.

Pro další úvahy se teda musíme obrátit k teoretickým konceptům, případně více méně teoretickým programovacím jazyků – počínaje ještě relativně rozšířeným jazykem Eiffel a konče ryze teoretickými jazyky, jako PolyTOIL apod.

Základní definice a notace

Zaveďme proto notaci, připomínající spíše matematický zápis, než programovací jazyk (notace programovacího jazyka je přeci jen vedena ještě jinými potřebami než je výstižný typový systém). Budeme-li psát, že entita A je typu T, napíšeme A:T. Vztah podtyp-nadtyp budeme vyjadřovat symbolem <: - píšeme T <: U jestliže typ T je podtypem typu U.

Pokud zatím mluvíme pouze o základních typech, kterými by měl být programovací jazyk vybaven (jako jsou různé číselné, logické nebo řetězcové typy), relace má zatím velmi jednoduchou podobu, protože obsahuje jen identitu (tj. jestliže T je základní typ, platí T <: T).

Kromě typů samotných entit (proměnných a hodnot), budeme zkoumat i typy funkcí a objektů.

Funkce má n-tici vstupních a n-tici výstupních parametrů určitého typu, píšeme tedy U = (Ti1, ..., Tin) → (To1, ..., Tom), tj. typ U je typem funkce s n vstupními parametry s typem Ti1, ..., Tin a s m výstupními parametry To1, ..., Tom. V příkladech budeme ovšem většinou zmiňovat jednodušší příklady funkcí s jedním vstupním a jedním výstupním parametrem, což nám umožní vynechat indexy i závorky a psát V = T→U, nebo rovnou F:T→U, kde funkce F má „bezejmenný“ typ s jedním vstupním parametrem T a jedním výstupním parametrem U. Pokud funkce nemá žádné vstupní, nebo výstupní parametry, budeme používat klíčové slovo Void.

Pokud funkce má pouze výstupní parametr, tj. F:Void→U, můžeme vynechat Void i šipku a psát F:U. V takové notaci zanikne rozdíl mezi proměnnou a funkcí, což je konec konců jeden z klíčových požadavků teoretických úvah o objektově-orientovaných programovacích jazycích (zvaný „Uniform Access Principle“ nebo „Space-Time Dilemma“), přestože například notace Javy (a ostatně i jiných programovacích jazyků) tento požadavek nerespektuje.

Na typech funkcí můžeme definovat obdobnou relaci jako na typech entit. Pokud jsou V1 = T1→U1 a V2 = T2→U2, můžeme psát V1 <: V2, pokud T2 <: T1 a U1 <: U2 (pokud bude funkce mít několik vstupních či několik výstupních parametrů, počty parametrů musí být v obou typech stejné a relace musí platit po složkách). Všimněme si obrácené relace u vstupního typu – pokud typ funkce V1 má být podtypem typu V2, musí typy vstupních parametrů funkce typu V1 být nadtypy odpovídajících vstupních parametrů funkce typu V2 (vrátíme se k tomu později v příkladu, na kterém tento zdánlivý paradox bude ilustrován a objasněn, aby se vzápětí vrátil zpátky jako přirozený požadavek v diskusi o tzv. matchingu). Relace výstupních parametrů kopíruje relaci funkcí, což odpovídá Uniform Access Principle, protože přesně taková bude relace mezi typy proměnných – a my nechceme činit rozdíl mezi proměnnou a funkcí bez vstupních a s jedním výstupním parametrem.

Objektový typ, který budeme studovat ne jen jako celek, ale také co se týče jeho struktury, je vlastně uspořádaná n-tice typů funkcí. Tyto funkce nazýváme „členské funkce“ třídy. Tato definice

pomíjí dva aspekty, běžné ve standardních programovacích jazycích: že objekty mají kromě členských funkcí i členské atributy a že existují také statické členské funkce a atributy.

Atributy můžeme v našich úvahách pominout, protože pokud chceme získat hodnotu atributu, je s ohledem na „Uniform Access Principle“ lhostejno, jedná-li se o atribut anebo o unární funkci, a pokud chceme atributu hodnotu přiřadit, lze to udělat tzv. setter funkcí – některé programovací jazyky přímo nabízejí konstrukce, které přiřazení hodnoty do atributu samy nahradí voláním setter funkce.

Statické funkce také nepředstavují problém, protože v běžném smyslu se vlastně nejedná o funkce objektu (tj. instance třídy), ale o funkce samotné třídy – a pokud vyhovíme častému požadavku považovat třídu za objektu typu Třída, můžeme jednu třídu považovat vlastně za definici dvou typů – „objektového typu“ a „třídového“ typu, které si mezi sebe rozdělí instanční a statické funkce.

Nad objektovými typy budeme definovat relaci <: tak, že pro objektové typy C a D platí, že C <: D, pokud pro každou funkci F typu V v D existuje odpovídající funkce F typu U v C tak, že U <: V. Všimněme si dvou věcí – za prvé relace se váže přes jména funkcí, tj. funkce se v C i v D musí jmenovat F, a za druhé, podmínka hovoří o všech členských funkcí objektového typu D, ale jen o některých členských funkcí objektového type C. Členské funkce C, které jsou jaksi navíc, nehrají v definici relace roli.

Takto můžeme rekurzivně definovat relaci <: nad všemi typy programovacího jazyka. Začneme identitou základních typů a postupně rekurzivně relaci dodefinujeme nad všemi ostatními typy.

Na závěr úvahy si připomeňme, že standardní objektově-orientované programovací jazyky redukují relaci <: na funkcích na identitu, čímž odstraní většinu zásadních problémů typových systémů. Jinými slovy, pokud jsou C a D objektové typy a C <: D, pak pro každé F type V v D musí existovat odpovídající funkce F v C stejného typu V, nikoliv jeho podtypu.

Uveďme si příklad, na kterém půjdou demonstrovat některé základní problémy. Představme si následující hierarchii typů:

```
Object {}      // banální základní typ

Engine = Object + {fuel:String}
               // Engine je podtyp typu Object, má jednu unární funkci,
               která vrací string, tj. prostý text
CoalEngine = Engine + {fuel:String="Coal"}
DieselEngine = Engine + {fuel:String="Diesel",litres:Integer,full:Boolean}

Vehicle = Object + {engine:Engine}
DieselVehicle = Vehicle + {engine:DieselEngine}
               // DieselVehicle přepsal funkci engine, která teď vrací
               podtyp DieselEngine původního typu Engine – DieselVehicle
               je ale stále podtyp typu Vehicle
```

A teď kus programového kódu (v jakémsi náznakovém programovacím jazyce):

```
d:DieselVehicle = new DieselVehicle;
               // vytvořme nový objekt typu DieselVehicle
v:Vehicle = d; // typický polymorfismu: protože je DieselVehicle podtyp
               typu Vehicle, můžeme objekty typu DieselVehicle přiřadit
               entitě typu Vehicle
e:Engine = v.engine;
               // funkce engine vrací objekt typu DieselEngine, ale
               protože Engine je nadtyp typu DieselEngine, můžeme hodnotu
               díky polymorfismu uložit do entity typu Engine
```

Problém kovariance

Pokud třída mění typy parametrů zděděných funkcí na podtypy, nazývá se taková změna „kovariantní“, změna v opačném směru se nazývá „kontravariantní“. Pokud ke změně nedojde, mluvíme o tzv. „novarianci“.

Jak už víme, dědění ve standardních programovacích jazycích je přísně novariantní. Dále víme z předešlé sekce, že kovariantní změna výstupních parametrů a kontravariantní změna vstupních parametrů je v souladu s očekáváními (alespoň z pohledu jednoduché matematizace).

Problém nastává při kovariantních změnách vstupních parametrů děděných funkcí. A to navzdory tomu, že přesně taková změna je z hlediska „selského rozumu“ ta jediná správná (a naopak: najít srozumitelný příklad pro kontravariantní změnu vstupních parametrů není vůbec jednoduché). Pokračujme v definicích tříd:

```
FuelSource = Object + {provide:Engine→Void }
               // funkce provide naplní nádrž objektu typu Engine, nemá
               žádnou návratovou hodnotu
DieselSource = FuelSource + {provide:DieselEngine→Void}
               // DieselSource přebírá definici typu FuelSource a mění jen
               typ funkce provide

function DieselSource.provide(de:DieselEngine) =
    {while (not de.empty) de.litres++ }
               // pro typovou kontrolu není konec konců podstatné, jak je
               vlastně funkce provide implementována, ukázka ale pomůže
               naznačit, že funkce provide objektového typu DieselEngine
               potřebuje některé funkce, které má jen a jen DieselEngine a
               tudíž se nemůže spokojit s obyčejným typem Engine
```

Navzdory tomu, jak přirozeně to vypadá, DieselSource není vinou kovariantí změny argumentu podtypem typu FuelSource. Proč? Uvažujme následující kód:

```
Train = Vehicle + {engine:CoalEngine}

t:Train = new Train;
ds:DieselSource = new DieselSource;

function provide_fuel:(fs:FuelSource, v:Vehicle) =
     {fs.provide(v.engine)}

provide_fuel(ds, t);     // <-- tady nastává typová chyba
```

Volání funkce fs.provide očekává typ Engine a v.engine je tohoto typu, to znamená, že vše je jakoby v pořádku, ale ve skutečnosti je fs typu DieselSource a tak jeho funkce provide očekává DieselEngine a v je ve skutečnosti typu Train a jeho funkce engine vrací CoalEngine, který není kompatibilní s DieselEngine, což znamená, že kód obsahuje typovou chybu.

Takováto typová chyba ovšem je patrná z programového kódu pouze v místě, kde se z typu FuelSource odvozuje type DieselSource. Samozřejmě existuje mnoho programového kódu, kde by se mohlo s objekty typu DieselSource zacházet, jakoby byly i objekty typu FuelSource, ale my chceme mít naprostou jistotu jen na základě momentálního programového kódu – a protože dané definice typů umožňují zapsat typově nekorektní kód, celá konstrukce je typovou kontrolou odmítnuta.

Tedy, jak už jsme jednou definovali výše, aby DieselSource byl podtypem typu FuelSource, musí být DieselSource.provide <: FuelSource.provide a k tomu musí platit Engine <: DieselEngine, což neplatí. Platí přesný opak – DieselEngine <: Engine. Proto, aby platilo DieselSource <: FuelSource,

musí být vstupní parametr funkce provide objektové typu DieselSource typu Engine nebo nějakého jeho nadtypu – tedy Object.

Poznamenejme, že toto pravidlo nemá žádné věcné zdůvodnění, je to pouze důsledek našeho snažení o statickou typovou kontrolu na základě samotného programového textu.

Pokud si představíme, o čem se v příkladu hovoří, je nad slunce jasné, že čerpací stanice, poskytující naftu, umí obsloužit jen naftové motory, a nikoliv uhláky lokomotiv. Na druhou stranu je na místě se ptát, nakolik je nám nastíněná situace zřejmá díky jisté představivosti a intuici a znalosti; konec konců není zde žádný důvod, na základě kterého by zřejmost dané situace plynula pro počítač bez představivosti a intuice a znalosti.

Z hlediska programovacího jazyka máme vlastně dvě základní možnosti, jak danou situaci řešit – postavit polymorfismus a typovou kontrolu na jiném konstruktu než je podtyp (to znamená, že typová kontrola nějak pozná, že DieselSource lze kombinovat jen s DieselEnginem) nebo vybavit programovací jazyk prostředky, které umožní nějak reagovat na typový konflikt v době běhu programu (to znamená, že funkce provide objektového typu DieselSource bude sice mít vstupní parametr Engine, ale bude schopná detekovat, jaký Engine to je a obsloužit jen ty, co jim chutná nafta).

Problém s genericitou

Problém velmi podobný předešlému je spojený s genericitou. Uvažujme následující definice:

```
Person = Object + {name:String}
Student = Person + {school:String}

List[G] = Object + {size:Integer,item:Integer→G,add:G→Void}

PersonList = List[Person]
StudentList = List[Student]
```

Máme dva objektové typy: typ Person a jeho podtyp Student. Pak máme generický typový vzor List, který se chová vlastně jako indexovaný seznam objektů typu G. Definovali jsme si dva objektové typy, odvozené z tohoto vzoru, seznam objektů typu Person a seznam objektů typu Student.

Pak uvažujme následující funkci:

```
function HumanList(list:PersonList) =
     {from i = 1 until i > list.size loop list.item(i).name.print)}
```

Funkce přijímá seznam objektů typu Person a vypíše jejich jména.

Následující kód je typově v pořádku:

```
pl:PersonList = new PersonList

pl.add(new Student(Jarda, MFF UK))
pl.add(new Student(Franta, VUT Brno))

HumanList(pl)
```

Díky vztahu Student <: Person můžeme klidně do seznamu typu PersonList uložit jen samé studenty a funkce HumanList bez problémů vypíše „Jarda, Franta“, protože každý student je taky jenom člověk. Problém ale nastane, zeptáme-li se, je-li i každá skupina studentů také skupinou lidí, nebo-li je také správně následující kód?

```
sl:StudentList = new StudentList
```

```
sl.add(new Student(Jarda, MFF UK))
sl.add(new Student(Franta, VUT Brno))

HumanList(sl)
```

Pokud by programovací jazyk nedisponoval typovou kontrolou, byl by uvedený kód v pořádku, funkce HumanList by prostě prošla seznamem studentů, a protože každý z nich je i člověkem a má tudíž i jméno, nenastal by žádný problém. Tato úvaha ale předpokládá, že víme, co se odehrává uvnitř funkce HumanList. Představme si, že by funkce HumanList vypadala takto:

```
function HumanList(list:PersonList) = {
    list.add(new Person(Lojza));
    from i = 1 until i > list.size do list.item(i).name.print}
```

Na funkci samotné není nic špatného: entita list je typu PersonList (=List[Person]) a lze tedy do ní zcela jistě přidat Lojzu, taktéž typu Person. Ale představme si takovouto funkci použitou právě v předešlém příkladě, totiž že zavoláme funkci HumanList a předáme mu seznam typu StudentList (= List[Student]). Lojzův kontravariantní pokus vetřít se mezi studenty by tak skončil typovou chybou.

Problém je právě ve funkci add, která mění kovariantně svůj vstupní argument. Chceme-li, aby StudentList <: PersonList, musí platit daný vztah i pro všechny členské funkce. Funkce size se nemění a size:Integer <: size:Integer platí, protože Integer <: Integer; aby platilo item:Integer→Student <: item:Integer→Person, musí opět platit Integer <: Integer a pak také Student <: Person, což oboje také platí; aby platilo add:Student→Void <: add:Person→Void, musí platit Void <: Void a Person <: Student – což ovšem neplatí!

Narážíme tady opět na stejný problém – pokud chceme pro generickou třídu L[G] používat pravidlo „A <: B implikuje L[A] <: L[B]“, musíme zajistit, aby v typovém vzoru L se typ G vyskytoval pouze jako výstupní parametr členských funkcí. Pokud se bude vyskytovat pouze jako parametr vstupních funkcí, můžeme naopak použít pravidlo „A <: B implikuje L[B] <: L[A]“. Jak jsme ale viděli před chvílí na typovém vzoru List, ani jedno takové omezení není praktické a obvyklý typový vzor bude zcela přirozeně využívat typ G jako vstupní i jako výstupní parametr svých funkcí.

Problém s MyType

Třetí obvyklou objektově orientovanou typovou obtíží, která je ale nakonec jen variací na známou notu, je konstrukce s odkazem na vlastní typ (PolyTOIL jí řeší pomocí klíčového slova MyType, Eiffel používá zobecněnou techniku tzv. „anchored“ typování a MyType píše jako „like Current“).

Představme si, že rozšíříme typ Person o funkci bestFriend:Person. Pak, protože existuje nějaká stavovská hrdost, předefinujeme funkci v typu Student na bestFriend:Student. A tak dále, pro každý podtyp typu Person budeme předefinovávat funkci bestFriend, aby její návratová hodnota byla stejného typu. To je ovšem velmi neefektivní a pracné. Proto bude vhodné definovat funkci jako bestFriend:MyType, kde MyType není nějaký konkrétní typ, ale pro objekty vlastního typu Student (tj. takové objekty, které jsou typu Student, ale nepatří do žádného jeho podtypu) to bude typ Student, pro objekty vlastního typu Employee to bude typ Employee apod.

Taková konstrukce jistě šetří programový kód a je chvályhodná, má ale opět problém se statickým typováním. Ukažme si to na příkladu: zaveďme ještě jednu funkci do typu Person, totiž talkTo:MyType→Void. Pak aby platilo Student <: Person, musí platit Student.talkTo <: Person.talkTo, tj. Student→Void <: Person→Void, tedy Person <: Student. Protože tedy z Student <: Person plyne Person <: Student, platí, že Person = Student. Všimněme si, že statické typování je odkázáno na text programu a v textu programu, díky konstrukci s MyType, jsou skutečně typy Person a Student stejné, ale to rozhodně není to, co jsme sledovali!

Možná řešení – přetypování

Jaké jsou možnosti řešení daného problému?

První řešení, které používají všechny standardní programovací jazyky, je zakázat jakékoliv takovéto problematické konstrukce. Přepisování typů parametrů zděděných funkcí se ve standardních programovacích jazycích prakticky nevyskytuje (a to ani povoleným směrem), konstrukce s MyType tím pádem také nemá žádné opodstatnění, a genericita se sice vyskytuje (jak v C++, tak ve slibované Java 1.5), ale rozhodně je prosta jakýchkoliv vazeb na polymorfismus.

Znamená to ale, že problémy, které jsme se pokoušeli řešit staticky zkontrolovanou konstrukcí jazyka, musíme vyřešit až za chodu programu, nejspíše nějakým násilným přetypováním, nebo alespoň pokusem o něj.

Takto se dá vyřešit problém s typem DieselSource: metoda provide prostě vstupní parametr Engine zkusí přetypovat na DieselEngine, a půjde-li to, bude pokračovat, nepůjde-li to, neudělá nic. Samozřejmě bude tak trochu spoléhat, že logika programu, který metodu provide volá, si pojistí, že ji nebude volat s ničím jiným, než s parametry typu DieselEngine, ale na to se dá samozřejmě spoléhat jen dokud na programovém kódu pracuje omezená a trvale informovaná skupina lidí (nejlépe jeden programátor) – v jakémkoliv rozsáhlejším projektu je takové očekávání iluzorní.

Přetypování je také jediným řešením pro polymorfní vztahy mezi generickými kontejnery, které nabízejí standardní programovací jazyky, jako například Java, kde jsou všechny kontejnery zcela obecné (a tudíž vlastně vzájemně kompatibilní) a skutečný typ se získává přetypováním jednotlivých položek kontejneru. To, že dané přetypování lze provést, je ponecháno plně v režii aplikace a na kontextu.

Možná řešení – CAT funkce

Jinou možnost nabízí programovací jazyk Eiffel. Eiffel povoluje při dědění přepisovat typy vstupních a výstupních parametrů i takovým způsobem, že výsledná podtřída nebude podtypem své nadtřídy ve smyslu naší definice – v Eiffelu se každá podtřída považuje i za podtyp (důsledné sjednocení typů a tříd je jedním ze základních designových principů Eiffelu), jestli je podtyp v rozporu s polymorfismem se zjišťuje při kompilaci typovou kontrolou na takzvané CAT funkce. CAT je zkratka z „Changing Availability or Type“ a označuje funkce, které při dědění změnily typy svých vstupních parametrů nebo změnily svojí viditelnost (další z neobvyklých možností Eiffelu, která obdobně komplikuje polymorfismus: podtřída může některé funkce, které byly původně public, přeznačit při dědění na privátní funkce).

Takové CAT funkce se pak nesmějí vyskytovat v situacích, kdy by mohly kolidovat s polymorfismem – jejich výsledek nesmí být například přiřazen formálnímu parametru jiné metody.

Dochází tak k rozdělení světa na dva neslučitelné tábory: na polymorfní a CAT tábor a typy z jednoho tábora nesmějí být kombinovány s typy druhého tábora. Je samozřejmě otázka, jestli takovéto schizma není příliš vysokou cenou, zaplacenou za některé vymoženosti Eiffelu, které nakonec nejsou ani zásadní, produkují ale zásadní problémy v typovém systému.

Nakonec sám Bertrand Meyer, autor Eiffelu, připouští konstrukci s přetypováním jako jedinou možnou, pokud nechceme přijít o polymorfismus, jakožto klíčovou vlastnost objektově-orientovaných systémů.

Možná řešení - Matching

Řešení, orientované především na situaci s MyType, nabízí teoretický výzkum. Vžil se pro něj termín „matching“ a jedná se o jiný druh polymorfismu, který není založený na podtypech. Cílem matchingu je poskytnout techniku v rámci programovacího jazyka, která umožní právě popsané

konstrukce nejen zapsat a následně zkompilovat a provést, ale hlavně staticky typově zkontrolovat na základě programového kódu.

Základním předpokladem úvah o matchingu je oddělení třídové a typové hierarchie. Přesněji řečeno, dědičnost se aplikuje jen na třídy a koncentruje se zejména na znovupoužitelnost už jednou napsaného programového kódu. Zděděná podtřída negeneruje nutně podtyp. Podtyp vzniká jaksi implicitně a typová hierarchie je odpoutaná od programového kódu a hierarchie tříd.

Například v jazyku PolyTOIL se definují zvlášť obyčejné typy a třídové typy (že třídový typ je podtypem obyčejného typu je zřejmé z textu definice obou typů, explicitně to není řečeno) a třídy jsou konstantní hodnoty třídových typů nebo dokonce typové funkce, vracející hodnoty určitého třídového typu; dědičnost se používá při definici tříd, tedy až v momentě, kdy se daný třídový typ doplňuje implementací, tj. programovým kódem.

Bohužel, stejně jako je teoretický výzkum matchingu, jsou teoretické i jeho výsledky. Práce Abadiho a Cardelliho „On Subtyping and Matching“ nabízí dvě definice matchingu pomocí F-bounded a higher-order subtypingu, které se liší matematickými vlastnostmi, jako je reflexivita a tranzitivita, které ale při své matematické složitosti nedávají odpověď na základní otázku programátora nad programovým kódem: co udělat, aby můj program neobsahoval typovou chybu?

(připomeňme si, že ale i sama tato otázka je značně teoretická, protože prakticky používané programovací jazyky konstrukce, které vedou k výše popsaným typovým problémům, prostě neobsahují)

Východisko – typová chyba jako běhová chyba (?)

Je ale všechno toto úsilí nutné. Neexistuje jiný možný přístup k celému problému, totiž ignorovat celý problém statického typování?

Pokud nahlédneme interakci mezi objekty (a nebavme se teď o typech těchto objektů) klasickým způsobem, můžeme se vrátit ke klasickému termínu „zasílání zpráv“. Smalltalk, objektově-orientovaný interpretovaný programovací jazyk úplně bez typové kontroly, je proslulý chybovým hlášením „Objekt nerozumí zaslané zprávě“. Obhájci statického typování dávají právě Smalltalk za odstrašující příklad, proč se programovací jazyk, určený pro solidní programování, nemůže obejít bez statického typování. Ale je tomu skutečně tak?

Skutečně jsme schopni v momentě kompilace odhadnout všechny možné interakce objektů v systému a roztřídit je podle toho do typů a staticky zkontrolovat, že všechno vychází? Jistě, svět, kde platí jen několik, dopředu známých a vlastně velmi jednoduchých pravidel, na základě kterých se ti, co se mohou setkat, vždycky jisto jistě domluví, by mohl být velmi krásný. Ale realita je jiná – nakonec skončíme u toho, že i ti, co by se vlastně domluvit mohli, se nesmějí setkat, protože to jakási pravidla nedovolují. (Což byl náš příklad s funkcí HumanList, která by bez problémů mohla obsloužit skupinu studentů, ale statická typová kontrola to nedovolí).

Nemohli bychom statické typování nahradit nějakou technikou, která by umožnila aplikačnímu programátorovi efektivně obsloužit nedorozumění, když už nastane, ale apriori by nediskriminovala žádný pokus o interakci objektů? Konec konců, v dnešním světě distribuovaných systémů a Internetu je těžké předpokládat v době kompilace něco o objektu, který se bude nacházet na opačném konci Zeměkoule a jako správný objekt nám nabídne jen svojí identitu, ale všechno ostatní si ponechá jako svoje tajemství (nejen seznam funkcí, které nabízí, ale třeba i programovací jazyk, ve kterém je napsán) – můžeme mu poslat nějakou zprávu a on nám buď odpoví anebo taky ne. Statická typová kontrola má panickou hrůzu z takovéto situace, protože je svázaná jedním programovacím jazykem a často i jedním jediným systémem a jediným monolitickým výsledným kompilátem.

Studovaná literatura

- Leontiev, Tamer Özsu & Szafron: On Type Systems for Object-Oriented Database Programming Languages, ACM 2002 – přehledová práce, na jedné straně vypisující seznam požadavků, které jsou relevantní v typových systémech objektově-orientovaných jazyků, a na druhé straně testující, které z těchto požadavků jsou k dispozici ve známých objektově-orientovaných programovacích jazycích současnosti od C++ až po PolyTOIL
- Meyer: Object-Oriented Software Construction, Prentice Hall 1997 – univerzální práce o objektově-orientované metodě, která krok za krokem buduje objektově-orientovaný systém, založený na jazyku Eiffel
- Cardelli: Semantics of Multiple Inheritance, Information and Computation 1988 – práce, uvádějící základy kalkulu s dědičností a podtypy
- Abadi & Cardelli: On Subtyping and Matching, ACM 1996 – základní teoretická práce, konstruující matematický model pro matching
- Bruce, Schuett & van Gent: PolyTOIL: A type-safe polymorphic object-oriented language, ACM 2003 – přehledový článek o PolyTOILu, včetně definice matchingu a důkazu úplnosti typového systému jazyka