Okruhy ke zkoušce X36OBP

Okruhy ke zkoušce X36OBP 1
Znovupoužitelnost 2
Požadavky 2
Dědičnost 2
Rozhraní 2
Polymorfizmus 3
Method dispatch 3
Způsob vyhledávání: 3
Dědičnost 3
Principy abstrakce konceptuálního modelování 3
Použití dědičnosti 4
Abstraktní třídy 4
Druhy dědičnosti 4
Dědičnost vs. Include 5
Vícenásobná dědičnost 5
Podmínky dědění (spíš doporučení) 6
Dynamická dědičnost 6
Datový typ 6
Definice 6
Proč používat datové typy? 7
Typová kontrola 7
Generické typy 7
Rozdělení jazyků podle datových typů 8
Typová inference (automatické určení typu) 8
Jak udělat typovost jazyka? 9
Introspection 9
Metadata 9
Reflection (reflexe) 10
Nevýhody reflexe 11
Metamodel 11
Garbage Collector 11
Použití 12
Počítání referencí 12
Sledování (tracking) 12
Mark & Sweep 12
Generační algoritmus (Copying collectors) 12
Optimalizace 13
Object features implementation 13
Jak vypadá objekt v paměti v Javě? 13
Virtuální funkce v C++ 13
Reprezentace bloků 14
Prototypy 14
Bytecode a JIT 14
Co je bytecode? 14
Interpreter (Virtual machine) 15
JIT – Just In Time 15
APL (smalltalk) 15
Klasifikace JIT 16
Ukázka bytecode Javy 16
Optimalizace 16
Překlad do nativního kódu 16
Functional concepts 17
Lazy evaluation (delay evaluation) 17
Funkce vyšších řádů (high-order) 17
Currying 18
Pure functions (pure expressions) 18
Rekurze 18

Znovupoužitelnost

Znovupoužitelnost v OOP je dána hlavně:

dědičností, polymorfizmem

Požadavky

Při návrhu je potřeba dbát na to, že se časem mohou měnit požadavky zákazníka/technologie apod. a čas, který jsme věnovali dobrému návrhu se nám vrátí několikanásobně zpátky. Každopádně si vždycky musíme stanovit nějaké požadavky. Např. tyhle byli v paperu (hledání nejkratší cesty Dijkstrou):

- abstrakce nad daty – různé typy grafů, různé uložení dat, různé datové typy
- různé uzlové a hranové atributy
- cachování vzdálenosti
- libovolný formát vstupu/výstupu
- řízení algoritmu (zastavení/pauza), zpětná vazba pro např. pro animace
- robustnost
- atd.

Dědičnost

Dědičnost je mechanismus oop, který umožňuje jednoduchou znovupoužitelnost existujícího kódu a má také důležitou funkci z pohledu polymorfizmu.

Od každé třídy lze oddědit třídu novou. Tato nová třída má přístup ke všem členům svých předků pokud nejsou označeny přístupovým modifikátorem private. Dědit lze vícekrát a často existuje speciální třída, která je automaticky předkem všech dalších tříd.

Oddděděná třída může přidat další data, rozšířit funkčnost svého předka přidáním dalších metod, nebo modifikovat funkcionalitu předka *překrytím* metod s modifikátorem abstract nebo virtual. Překrytí metody spočívá ve vytvoření metody se stejným prototypem, ale s jiným kódem. Význam tohoto kroku je plně vysvětlen v sekci o polymorfizmu.

S dědičností je příbuzné také téma rozhraní.

Rozhraní

Rozhraní představuje soubor deklarací metod a vlastností. Třída pak může takové rozhraní implementovat. To znamená vytvořit implementaci pro **každou** (není možné některé metody vynechat) metodu v rozhraní deklarovanou. Třída může implementovat více než jedno rozhraní a implementace se obvykle zapisuje stejně jako dědění.

Skutečný význam rozhraní je vysvětlen v sekci o polymorfizmu.

Polymorfizmus

Polymorfizmus je umožněn mechanismem dědičnosti a rozhraní.

Instanci třídy lze přetypovat na jakéhokoliv svého předka nebo na rozhraní, které implementuje. Klíčový je pak způsob volání překrytých členů a metod rozhraní: při použití virtuální metody nebo

abstraktního člena se použije verze definovaná na původní úrovni instance. Při volaní metody rozhraní se pak vykonává kód její implementace v dané třídě. Voláním stejné metody se tak provádí různý kód v závislosti na skutečném typu instance.

Rozdíl mezi dědičností a systémem rozhraní je v tom, že dědičností se rozšiřují možnosti třídy, kdežto interface slouží pouze jako nástroj pro unifikaci komunikace mezi heterogeními typy.

Method dispatch

http://en.wikipedia.org/wiki/Dynamic_dispatch

C++ používá virtual table (vtable) – tabulka pointerů na metody. Je komplikovanější pokud pokud se používá dědění z více předků. Princip je takový, že pokud je metoda označená jako virtual, tak se mrkne do tabulky a zavolá se metoda z tabulky. http://en.wikipedia.org/wiki/Virtual_table

U „normálních“ metod se používá statické volání – adresy metod se určí za překladu.

Smalltalk a hodně interpretovaních jazyků (Python, Ruby, Objective-C) používá dispatcher – je to pomocný nástroj, který v případě řízení od potomka projde jeho metody a když tam nenajde nic vhodného, tak postupuje výš. Není to tolik efektivní jako metoda u C++.

Javascript používá velmi jednoduchou a docela efektivní způsob – ukládá metody jako instanční proměnné, takže když je je možnost přepsat tuto proměnnou bez problému.

Způsob vyhledávání:

C++ - mrkne do tabulky a ví – tabulka se vytváří při překladu.

Smalltalk a spol. - Dispatcher prohledává objekty a hledá nejvhodnější metodu pro zavolání od potomka k předkovi. Po nalezení se zastaví. Pokud by se nezastavil (jiné jazyky) můžeme hovořil to multiple-dispatch.

Javascript – zavolá se metoda, která je uložena v instanční proměnné – žádné hledání.

Dědičnost

Principy abstrakce konceptuálního modelování

- Classification/Instantiation
- Aggregation/Decomposition
- Generalization/Specialization
- Grouping/Individualization

Použití dědičnosti

- **Implementace** (stornování (přepis prázdnou metodou), optimalizace (přepis lepším řešením), konvence (např. doplnění metod, aby třída odpovídala očekávanému)
- **Kombinace** (dědíme za účelem získat potom s kombinací vlastností rodičů)
- **Obalení** (Např. Smalltalk nemá moduly a pokud bychom chtěli třídy nějak spojit dohromady můžeme je podědit od společného předka)

- **Zobecnění** (někdy se dá dědit i pro to, abychom mohli zobecnit třídu (normálně je to naopak))

Abstraktní třídy

Z abstraktní třídy nemůže být vytvořena instance. Ne všechny jazyky tento mechanismus umí – např. u Smalltalku záleží na jeho implementaci. Při dědění z abstraktní třídy rozdělujeme:

- Dědění s kompletní implementací (potomek definuje všechny abstraktní metody)
- Dědění s částečnou implementací (potomek nedefinuje všechny abstraktní metody -> taky nelze vytvořit jeho instanci)

Druhy dědičnosti

- Třídní dědičnost (class-based) – ta, kterou známe z Javy, C++, Smalltalku...
- Prototypová (objekt-based) – používá se v jazycích, ve kterých nejsou třídy, ale rovnou objekty – vytvoření „instance“ se potom dělá tak, že vytvoří kopie daného objektu.

Další rozdělení:

- Jednoduchá (dědí se z jedné třídy)
- Vícenásobná

Dědičnost vs. Include

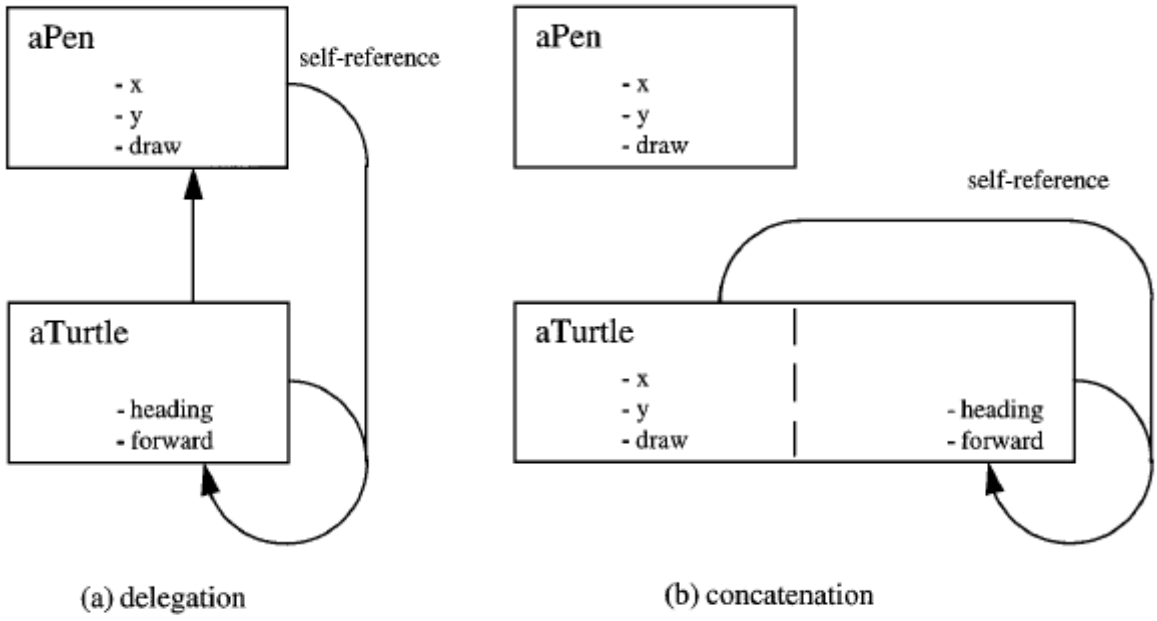


Figure 9. Delegation versus concatenation.

Dva způsoby implementace dědičnosti – jeden je vytvoření reference na předka a druhá je zkopírování všech položek z předka do potomka. Self-refence (this) odkazovala na aPen, ale

změnila se na potomka aTurtle. V (b) je self reference pořád v rámci aTurtle. Příklady (a) class-based jazyky C++, Smalltalk, (b) – např. Simula – tzv. prefixové dědění.

Vícenásobná dědičnost

http://www.builder.cz/art/cpp/cpp_vicededic.html

Vse by bylo krásné nebýt několika velmi nepříjemných problémů, které vícenásobná dědičnost přináší. Problémy:

- Konflikt jmen

  Při vícenásobné dědičnosti může nastat situace, že třída dědí z nadtříd, které mají stejné názvy atributů, nebo metod. Který potom zdědit?

- Opakovaná dědičnost

  Nějaký atribut, nebo metoda mohou být zděděné "vícekrát". Opakovaná dědičnost nastává, jestliže v třídním diagramu existuje mezi 2 třídami více než jedna cesta. Například třída B dědí ze třídy A. Třída C dědí ze třídy A. Třída D dědí ze třídy C a B. Třída D vlastně dvakrát dědí ze třídy A. Má mít třída D položky třídy A dvakrát, nebo jen jednou?

- Volání konstruktorů a destruktorů

  Při vytváření instance se implicitně zavolají konstruktory předka, při rušení se volá implicitně destruktor potomka. Ve vícenásobné dědičnosti je ale bezprostředních předků více. V jakém pořadí se tedy mají volat konstruktory a destruktory?

Dalším problémem často se projevujícím u vícenásobné dědičnosti je správné (spíše špatné) přetypování potomka na předka. Jedná se ale spíše o problém s přetypováváním, proto se mu nebudu věnovat ve článcích o vícenásobné dědičnosti, ale až někdy v budoucnu ve článku věnovanému přetypování v C++.

Podmínky dědění (spíš doporučení)

Vyhnout se vícenásobnému dědění viz. výše. Důvod je takový, že při vícenásobném dědění vzniká orientovaný acyklický graf. Pokud dědíme pouze jednoduše vzniká strom – lépe se to zpracovává a hlavně je to lepší na orientaci programátora. Proto některé jazyky vícenásobnou dědičnost nepodporují – její použití většinou značí chybu v návrhu.

Tipy:

- Používat zapoudření – to, co by nemělo být vidět ven, schovat
- Přemýšlet o rozdílu dědičnost/kompozice
- Virtuální metody jsou obecně pomalejší – používat tam, kde je potřeba

Dynamická dědičnost

Každý objekt má nějaký svůj logický stav – např. Dveře jsou zavřené a otevřené. Dynamická

dědičnost má tu myšlenku, že pro každý stav by existovala podtřída (subclass) např. : OtevřenéDveře, ZavřenéDveře a tyto podtřídy by měly specifické metody pro svoje stavy např. : OtevřenéDveře.zavřít(); ZavřenéDveře.otevřít(); A to všechno runtime. Měl by to umět Smalltalk a např. CLOS. (nativně)

Do ostatních jazyku lze doprogramovat Dispatcher, který by tuto funkcionalitu zajišťoval.

Datový typ

http://en.wikipedia.org/wiki/Data_type

Definice

Datový typ je atribut dat, který říká překladači (počítači) a programátorovi o jaký druh dat se jedná.

Rozdělujeme:

- Jednoduché
  - celočíselné (integer), boolean (0/1)
  - reálné (floating-point)
  - textové řetězce
- Složené
  - objekty (objects)
  - třídy (classes)
  - pole (arrays)
  - záznamy (records)
  - struktury (structures)

Každý typ má svůj rozsah, každý programovací jazyk má různé typy.

Složené typy se skládájí z jednoduchých datových typů a nebo z se složených.

Proč používat datové typy?

Datové typy se zavádějí pro to, abychom mohli kontrolovat kód za překladu. Tzn. že pokud napíše programátor nějakou chybu, přijde se na ní ihned. Z datového typu lze také vyčíst jaké data lze asi v proměnné očekávat, což je výhoda hlavně při práci v týmu a při studiu cizího kódu.

Typová kontrola

http://cs.wikipedia.org/wiki/P%C5%99eklada%C4%8D

Sémantický analyzátor zpracovává syntaktický strom a provádí analýzu významu jednotlivých operací. Na rozdíl od syntaktického analyzátoru, který provádí kontrolu správnosti struktury programu čili zápis programového kódu, sémantický analyzátor provádí kontrolu správnosti

operací. V tomto mechanismu se odehrává veškerá typová kontrola a různé konverze. Prací sémantického analyzátoru je např. analýza typů výrazů na levé a pravé straně přiřazovacího příkazu, přičemž musí zabránit např. přiřazení hodnoty s plovoucí řádovou čárkou do proměnné s pevnou řádovou čárkou.

Vstupem sémantického analyzátoru je syntaktický strom, který vygeneroval syntaktický analyzátor, výstupem je opět syntaktický strom, ale s doplněnými dodatečnými informacemi a s provedenými konverzemi.

Generické typy

http://en.wikipedia.org/wiki/Generic_programming

- Šablony C++
- Generické programování Java (5.0), C#, VB .NET (.NET 2.0), Delphi (rok 2008), Ada, Eifell

C++ : Při překladu je nahrazen typ T v šabloně... <int>max(20,3);

```
template <typename T>
T max(T x, T y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

```
int max(int x, int y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

V Javě se tento toto nahrazení typu jmenuje *type erasure* v po překladu není typ šablony k dispozici. Proto typ List<String> je List – stejně jako List<Int>.

V C# (a .NETu) se používá jiná metoda, která umožňuje reflexi i nad generickými typy. Navíc lze využít klíčové slovo *where*, kterým se dají určit omezení pro typ použitý v parametru generického typu. Např., že typ musí mít bezparametrický konstruktor.

V javě, C#, C++ .. všude můžeme udělat složené generické typy: `Dictionary<string, List<int>>`

Rozdělení jazyků podle datových typů

- Staticky typované (C++, C, C#, Java, Pascal, ...)
  - typová kontrola se provádí za překladu
  – výhody: bezpěčnost (přetypování), optimalizace, dokumentace, abstrakce od dat
- Dynamicky typované (PHP, Smalltalk, Perl, Python, Ruby...)
  - typová kontrola se provádí za běhu programu

Typová inference (automatické určení typu)

http://en.wikipedia.org/wiki/Type_inference

Automatické určení datového typu je typické pro funkcionální jazyky (Haskell, Scheme), ale v současné době se vyskytuje i jako nová vlastnost silně staticky typovaných jazyků např. C# 3.0, VB .NET 9.0...

U každé proměnné je určen i datový typ, který proměnná má.

Příklad:

```
addone(x) {
    val result;  /*inferred-type result */
    val result2; /*inferred-type result #2 */

    result = x+1;
    result2 = x+1.0;  /* tady by normálně byla chyba */
    return result;
}
```

Pokud je vstupní proměnná *x* třeba int, tak *result* bude taky int. Přiřazení do *result2* by normálně skončilo s chybou (nelze sčítat int+float - chyba přesnosti), ale v jazycích tohoto typu se hojně používají implicitní typové konverze.

Aby tohle fungovalo musíme mít ohodnocené atomické hodnoty proměnných jako třeba:
True je boolean, 11 je integer, 1.123 je float atd. Potom lze vysledovat jakého typu mají proměnné být.

Algoritmus (trocha angličtiny ještě nikoho nezabila :))

The common algorithm used to perform type inference is the one now commonly referred to as Hindley–Milner, Damas–Milner algorithm. It has been referred to in the past as polymorphic type checking or Algorithm W.

The origin of this algorithm is the type inference algorithm for the simply typed lambda calculus, which was devised by Haskell B. Curry and Robert Feys in 1958. In 1969 Roger Hindley extended this work and proved that their algorithm always inferred the most general type. In 1978 Robin Milner [1], independently of Hindley's work, provided an equivalent algorithm, Algorithm W. In 1982 Luis Damas [2] finally proved that Milner's algorithm is complete and extended it to support systems with polymorphic references.

Příklad:

```
var x := 5;     // (1)  (x is an integer)
var y := "37"; // (2)  (y is a string)
x + y;          // (3)  (?)
```

Javascript by dal 375. Obecně hodně záleží na přetíženém operátoru – např. PHP kvůli tomu používá pro spojování řetězců '.' (tečku).

Jak udělat typovost jazyka?

Pokud to chápu správně, tak rozšířit vstupní gramatiku, upravit lexikální, syntaktickou a sémantickou analýzu.

Introspection

http://en.wikipedia.org/wiki/Introspection_(computer_science)

- *zkoumání svého nitra, sebepoznání*

Ve smyslu objektů to znamená sebepopisnost objektů v run-timu. Může být použita k polymorfizmu. Prostě se dá zjistit, co za je daný objekt za třídu.

Operátory pro porovnání, jestli je objekt určité třídy:

C# - klíčové slovo is

Java – klíčové slovo instanceof

C++ - operátor typeid(), použitelný taky dynamic_cast<>

Metadata

Metadata v OOP jsou chápána tak, že každá třída (objekt) vrací informace o tom, jakého je typu, jaké má metody (a ty jaké mají parametry), vlastnosti (jakého typu). Třídy lze dekorovat pomocí atributů (např. C#) a potom za běhu tyto atributy číst. Metadata se využívají hlavně pro reflexi – viz dále.

Metadata se získávají od objektu pomocí nějaké metody, kterou má každý objekt. Většinou se jedná o instanci třídy, která popisuje strukturu třídy a umožňuje s ní určité činnosti (viz dále) .

Získání tříd typu v různých jazycích: C# - Object.GetType()

Java – Object.getClass()[1]

Delphi – TObject.ClassName

Smalltalk – class

Reflection (reflexe)

http://en.wikipedia.org/wiki/Reflection_(computer_science)#C.2B.2B

Způsob, kdy je program měnit sám sebe za běhu. Mluvíme o metaprogramování nebo reflexivním programováním. Jedná se o techniku, která se používá více ve vyšších programovacích jazycích (C#, Java, Smalltalk...) v programovacím jazyku C++ lze získat informace o typu (RTTI), ale reflexe možná není (ale jsou cesty).

1 C++ a Delphi pouze s RTTI (Run-time type information) – podpora těchto symbolů musí být zakompilována v kódu

Reflexivní programování rozšiřuje OOP paradigma a zahrnuje „sebepopisnost“, „sebemodifikaci“ a „sebereplikaci“ - program může modifikovat sám sebe.

Příklad, jak vytvořit třídu za běhu:

```
// Without reflection
Foo foo = new Foo();
foo.hello();

// With reflection
Class cls = Class.forName("Foo");
Object foo = cls.newInstance();
Method method = cls.getMethod("hello", null);
method.invoke(foo, null);
```

Java

```
"Without reflection"
Foo new hello

"With reflection"
(Compiler evaluate: 'Foo') new perform: #hello
```

Smalltalk[2]

K čemu to je dobrý? Jeden příklad za všechny:

OR-Mapování – máme relační DB a v ní tabulku např. osob. V programu načteme řádek tabulky a podle názvu sloupce voláme settery ve třídě, kterou jsme si předtím vytvořili. Např:

ID	Name	Age
1	Pepa	12

```
class Person
{
   setID(int id);
   setName(string n);
   setAge(int i
}
```

Dalším příkladem by mohlo být nahrazení switch-e – pokud potřebujeme vytvořit třídu na základě jména (stringu) (návrhový vzor Factory). Kdybych těch položek ve switch-i mělo být hodně (nebo neznámo moc) vyplatí se vytvořit třídu reflexí na základě jména třídy.

Nevýhody reflexe

Jedinou nevýhodou reflexe je její malá rychlost. Proto se nepoužívá na místech choulostivých na čas. Jinak je to moc dobrá vymoženost. ;)

2 Ve smalltalku je to znatelně jednodušší než v Javě

Metamodel

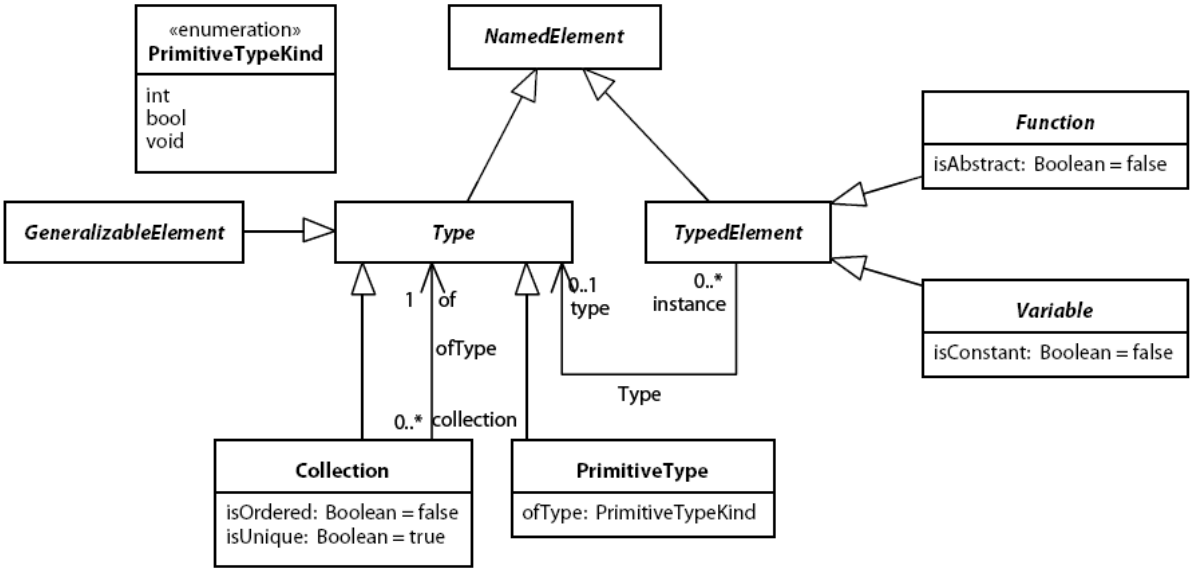


Figure 5: Types diagram

Tady je obrázek nějakého metamodelu datového typu. Jak je vidět – je to hodně podobné gramatice -> názvy tříd by byly názvy pravidel a tak nějak... prostě to je tak nějak +- stejné. :)

Garbage Collector

http://cs.wikipedia.org/wiki/Garbage_collector

Garbage collector (GC) je obvykle část běhového prostředí (programovacího) jazyka, nebo přídavná knihovna, podporovaná kompilátorem, hardware, operačním systémem, nebo jakoukoli kombinací těchto tří. Má za úkol automaticky určit, která část paměti programu je už nepoužívaná, a připravit ji pro další znovupoužití.

Použití

Zřejmě nejznámějším jazykem, který používá garbage collector je jazyk Java - ten v dnešní době (JDK 1.5) používá rovnou čtyři druhy garbage collectorů, přičemž všechny jsou generační. Další známou platformou je .NET, který používá obdobu algoritmu *Mark & Sweep*. Smalltalk (uplně první verze) používal počítání referencí nyní používá kombinací metody *Mark & Sweep* a dalších.

Počítání referencí

Vůbec první algoritmus pro garbage collector se jmenoval *počítání referencí* (reference counting). Funguje následovně: Ke každému objektu je přiřazen čítač referencí. Když je objekt vytvořen, jeho čítači je nastavena hodnota 1. V okamžiku, kdy si nějaký jiný objekt nebo kořen programu (kořeny jsou hledány v programových registrech, v lokálních proměnných uložených v zásobnících jednotlivých vláken a ve statických proměnných) uloží referenci na tento objekt, hodnota čítače je zvětšena o 1. Ve chvíli, kdy je reference mimo rozsah platnosti (např. po opuštění funkce, která si referenci uložila), nebo když je referenci přiřazena nová hodnota, čítač je snížen o 1. Jestliže je hodnota čítače některého objektu nulová, může být tento objekt uvolněn z paměti. Když je

uvolňován z paměti, všem objektům, na něž má objekt referenci, se sníží hodnota o 1 - tedy uvolnění jednoho objektu může vést k uvolnění dalších objektů. Nevýhoda této metody spočívá ve faktu, že neumí detekovat cykly. Cyklus nastává v okamžiku, kdy dva a více objektů ukazují samy na sebe, například když rodičovská třída ukazuje na svého potomka a ten má referenci zpátky na rodiče. Tyto dva objekty nebudou mít nikdy čítač roven nule, ačkoli jsou nedosažitelné z kořene programu. Další nevýhoda spočívá v režii, která je nutná pro zvyšování a snižování čítačů u každého objektu. Kvůli těmto nedostatkům se reference counting v dnešní době nepoužívá jako univerzální garbage collector.

Sledování (tracking)

Sledovací algoritmy tzv. *zastaví svět* (v tomto smyslu tedy běh programu) a začnou vyhledávat objekty. Začínají v kořenové množině programu a pokračují po referencích, dokud neprozkoumají všechny dosažitelné objekty. Algoritmy, založené na tomto principu, se používají téměř výlučně pro implementaci garbage collectorů v dnešních programovacích jazycích. Jako první byla tato metoda použita v jazyce Lisp v roce 1960, kde ji využíval algoritmus nazvaný *Mark & Sweep*.

Mark & Sweep

Algoritmus nejdříve nastaví všem objektům, které jsou v paměti, speciální příznak *navštíven* na hodnotu *ne*. Poté projde všechny objekty, ke kterým se lze dostat. Těm, které takto navštívil, nastaví příznak na hodnotu *ano*. V okamžiku, kdy se už nemůže dostat k žádnému dalšímu objektu, znamená to, že všechny objekty s příznakem *navštíven* majícím hodnotu *ne* jsou odpad - a mohou být tedy uvolněny z paměti.

Generační algoritmus (Copying collectors)

Při použití garbage collectorů se dají empiricky vypozorovat dva důležité fakty. Tím prvním je, že mnoho objektů se stane odpadem krátce po svém vzniku. Tím druhým je skutečnost, že jen malé procento referencí ve „starších“ objektech ukazuje na objekty mladší.

Generační garbage collector využívá těchto skutečností a rozděluje si paměť programu do několika částí, tzv. *generací*. Objekty jsou vytvářeny ve spodní (nejmladší) generaci a po splnění určité podmínky, obvykle stáří, jsou přeřazeny do starší generace. Pro každou generaci může být úklid prováděn v rozdílných časových intervalech (obvykle nejkratší intervaly budou pro nejmladší generaci) a dokonce pro rozdílné generace mohou být použity rozdílné algoritmy úklidu. V okamžiku, kdy se prostor pro spodní generaci zaplní, všechny dosažitelné objekty v nejmladší generaci jsou zkopírovány do starší generace. I tak bude množství kopírovaných objektů pouze zlomkem z celkového množství mladších objektů, jelikož většina z nich se již stala odpadem.

Optimalizace

Optimalizaci na základě stáří objektu používá generační algoritmus – čerstvě vytvořené objekty jdou do mladší generace. Starší se přesouvají do starší generace... Šlo by i přesouvat na základě četnosti použití nebo posledního použití objektu.

Dobrou implementací cache a vylepšení GC lze získat vylepšení cache-miss o 20 – 40 % a zrychlení programu o zhruba 14 – 31 %. Podle technik uvedených v paperu **Reducing garbage collector cache misses,** Hans-J. Boehm, 2001.

Object features implementation

Jak vypadá objekt v paměti v Javě?

Taková tabulka:

```
class MyClass {
    byte a;
    int c;
    boolean d;
    long e;
    Object f;
}
```

```
[HEADER:  8 bytes]  8
[a:       1 byte ]  9
[padding: 3 bytes] 12
[c:       4 bytes] 16
[d:       1 byte ] 17
[padding: 7 bytes] 24
[e:       8 bytes] 32
[f:       4 bytes] 36
[padding: 4 bytes] 40
```

Hlavička obsahuje hash identifikující typ, pak flagy age, lock a reference na objekt třídy.

Virtuální funkce v C++

Je tam hezky vidět ta vtable:

```
class foo {
  virtual void a(int);
  virtual void b(float);
  int c;
  float d;
};
```

```
struct foo_vtable {
  typeid * type_id_ptr;
  int offset;
  void (* ptr_to_a)(int);
  void (* ptr_to_b)(float);
};

struct foo {
  foo_vtable * vtable;
  int c;
  float d;
};
```

Reprezentace bloků

Zatímco v C++ se bloky kódu převedou do strojového jazyka kompilací např. u Smalltalku to tak není. Blok je reprezentován meta-objektem.

Seznam Smalltalkových meta-objektů:

1. ***Structure***:
   `Behavior, ClassDescription, Class, Metaclass, ClassBuilder`
2. ***Semantics***:
   `Parser, Compiler, Decompiler, ProgamNode, ProgramNodeBuilder, CodeStream`
3. ***Behavior***:

```
CompiledMethod, CompiledBlock, Message, Signal, Exception
```

4. ***Control State***:
   `Context, BlockContext, Process, BlockClosure, ProcessorScheduler`
5. ***Resources***:
   `ObjectMemory, MemoryPolicy, WeakArray`
6. ***Naming***:
   `SystemDictionary, NameScope, PoolDictionary`
7. ***Libraries***:
   `MethodDictionary, ClassOrganizer, SystemOrganizer`
8. ***Environment***:
   `Browser, Inspector, Debugger`

Prototypy

http://en.wikipedia.org/wiki/Prototype-based_programming

Je to trochu jiný způsob objektového programování než class-based. Každý objekt je definován rovnou včetně hodnot a další „instance“ se děla pomocí klonování. Tuto techniku používají třeba třeba Self, Javascript nebo Lua.

Bytecode a JIT

Co je bytecode?

http://en.wikipedia.org/wiki/Bytecode

Bytecode (mezikód) se používá pro uložení instrukcí pro nějaký interpretr (virtuální mašinu). Je to mnohem rychlejší načítat instrukce z bytecode než znovu parsovat zdrojáky. Název je odvozen od zpravidla jednobytových instrukcí. Bytecode pak může být přeložen do strojového kódu a tím ještě urychlit činnost programu.

Interpreter (Virtual machine)

http://en.wikipedia.org/wiki/Virtual_machine

Virtual machine – efektivní kopie reálného stroje

Jedná se o interpretr instrukcí na vyšší úrovni (než je strojový kód). Často podporuje JIT (Just In Time) kompilaci do nativního kódu procesoru pro rychlejší vykonávání.

Příklady: JVM (Java Virtual Machine) - Java, CLR (Common Language Runtime) - .NET, Smalltalk má taky svojí VM

Interpret bývá implementován jako zásobníková ISA, protože je to rychlý způsob.

JIT – Just In Time

Překlad až v době spuštění programu (od cca. 60. let), ve Smalltalku od 80. let

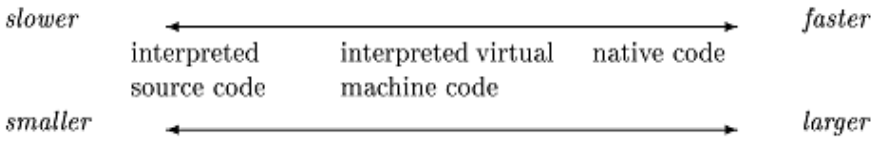


Fig. 1. The time-space tradeoff.

Dvě cesty – kompilace do nativního kódu – rychlejší program, interpretace bytecode – menší programy

Java

- statický překlad do bytecode (JVM)
- po roce 2001 se JVM kód začíná měnit na registr-based reprezentaci
- interpretace je pomalá → nebývalý výzkum v oblasti JIT (i díky komerčním tlakům na rychlou Javu)
- hledá se optimum a kompromisu mezi dobou spouštění a rychlostí běhu
- myšlenka: Optimalizační informace si předpočítat a uložit jako anotace pro JVM přímo do bytecode

APL (smalltalk)

- netypový jazyk (jako Smalltalk)
- „Drag-along“ je metoda odkládající spuštění bloku kódu co to jde (dnes se tomu říká Lazy
- Evaluation)
- „Beating“ metoda snižující množství manipulací s daty

Při přidání metody je tato zkompilována do kódu VM. Je použito kešování kódu pro VM a „lazycompilation“.

Klasifikace JIT

- invocation → implicitní vs. explicitní (uživatel musí něco konkrétního udělat)
- executability
  - JIT systém musí rozumět 2 jazykům (mezi nimiž překládá). Klidně to může být jeden a ten
    samý (pak se jedná jen o optimalizaci programu).
  - „monoexecutable“ → JIT systém umí spustit jen 1 z jazyků
  - „polyexecutable“ → JIT systém umí spustit více než 1 jazyk (může si tedy volit, zda spustí
    originální program, nebo až jeho překlad – respektive jejich části (bloky))
- concurency → Zda se JIT kompilátor pouští paralelně s programem, nebo zda musí být běh
- programu přerušován, kvůli kompilaci dalších částí. Paralelní postupy jsou teprve ve vývoji.

Ukázka bytecode Javy

`public static int Add(int a, int b)`	Code:

`{`
`  return a+b;`
`}`
	0: iload_0
	1: iload_1
	2: iadd
	3: ireturn

Ja na tom vidět ta zásobníková ISA.

Optimalizace

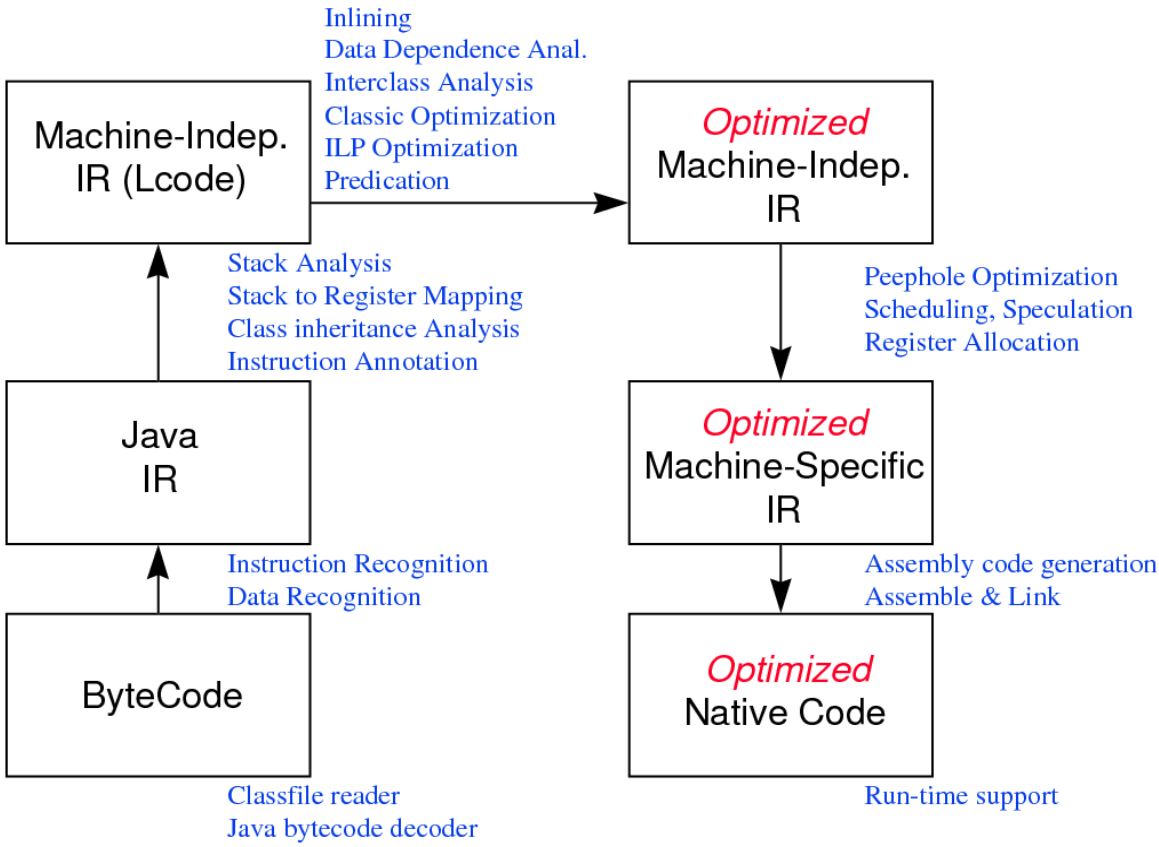
Způsoby:

- Vložením bytecode otevřeného příkazu pro rychlost (např. ve smalltalku FalseBlock)
- Může se použít inlining – tzn. místo volání metody se kód metody vloží přímo na místo volání
- Může se použít VM pro dynamický překlad z bytecode do bytecode pro optimalizaci

Překlad do nativního kódu

Záleží na jaký systém bytecode převádíme – pokud např. na klasické PC (které má v procesoru registry) musíme převést zásobníkové operace na operace s registry.

Zásobník (bytecode):	Registrové operace (nativní kód):
Push A Push B Add	Push A Push B r1 Pop(B) r2 Pop(A) r3 := r1 + r2 Push r3



Ilustrace 1: Průběh transformace bytecode -> nativní kód (java)

Functional concepts

http://en.wikipedia.org/wiki/Functional_programming

Lazy evaluation (delay evaluation)

http://en.wikipedia.org/wiki/Lazy_evaluation

zpožděné vyhodnocení – je technika zpoždění výpočtu až do doby, než opravdu potřebujeme výsledek. Díky tomu se program zrychlí, zabrání se chybám ve vyhodnocení výrazů, jsme schopni vytvořit „nekonečné“ datové struktury a můžeme podmínky udělat jako normální funkce a ne vestavěně. Tohle používá např. Haskell.

Opačná technika je eager (strict) evaluation, kterou používají běžné jazyky.

Lazy evaluation se používá i v Javě, C++ apod. -> jako to ternální operátor '?'. Jinak třeba v C# 3.0, Python... jsou to lambda funkce atd.

Funkce vyšších řádů (high-order)

Funkce vyšších řádů mohou jako svůj parametr přijímat funkce a vracet je jako výsledek. Jako příklad by mohla být derivace *d/dx (funkce),* která přijatou funkci zderivuje a vrátí jako výsledek. Podobné jsou first-class funkce, které mají stejný význam, ale je to termín spíše pro programování. High-order funkce jsou spíše ve spojení s matematikou a čísly.

Tato technologie umožňuje currying.

Currying

Je to technika, která slouží k transformaci funkcí. Z jedné funkce, která má x parametrů se udělá x funkcí, které mají jeden parametr.

Pure functions (pure expressions)

Jedná se o funkce nebo výrazy, které nemění žádná data a slouží pouze k výpočtu. Díky tomu můžeme optimalizovat:

- Pokud se nikdy nepoužije její výsledek, můžeme ji vynechat
- Pokud má parametry a žádné jiné datové závislosti, tak můžeme cachovat dvojici parametr-výsledek
- Pokud nemá jiné datové závislosti, tak je to super na paralelní operace

Rekurze

Pro urychlení operací lze rekurzi rozbalit do delšího řetězce výrazů.