GENERICKÉ PROGRAMOVÁNÍ: CÍLE A MOŽNOSTI IMPLEMENTACE

Miroslav Virius
KSE FJFI ČVUT v Praze, virius@km1.fjfi.cvut.cz

ABSTRAKT:
V tomto příspěvku se zabýváme jednak cíli generického programování, jednak porovnávám jeho implementací ve třech nejrozšířenějších programovacích jazycích, které ho podporují. Vedle roviny syntaxe jazyka rozebíráme i způsob jeho implementace a jeho důsledky pro jeho použitelnost.

KLÍČOVÁ SLOVA:
Generické programování, C, C++, Java 2 JDK 5, C# 2.0, šablona, kontejner, makro, prostředí .NET

1. ÚVOD

Generické programování je poměrně široký pojem, jehož význam se může v různých programovacích jazycích dosti značně lišit. Pravděpodobně prvním rozšířenějším jazykem, který poskytl možnost generického programování, byla již v 80. letech Ada; do povědomí širší programátorské veřejnosti však generické programování výrazněji proniklo až ve druhé polovině 90. let s nástupem šablon v C++. V současné době se s ním – vedle Ady a C++ – setkáme např. v jazycích C# (verze 2), Eiffel, Haskel, Java (verze 5) a ML. Vzhledem k rozšíření se zde budu zabývat pouze C++, C# a Javou; zmíníme se také o jazyku C, i když ten generické konstrukce vlastně nepodporuje..

Pokud jde o C++, se šablonami – specifickou implementací generického programování v tomto jazyce – se setkáváme poprvé ve verzi AT&T 2.1. Prvním široce dostupným překladačem, který je implementoval, byl Borland C++ 3.1. Přes různé změny ve specifikaci šablon v průběhu standardizace jazyka C++ v 90. letech minulého století zůstal základní princip implementace stejný.

V Javě se nástroje pro generické programování objevily v JDK 5 uvolněném firmou Sun Microsystems na podzim roku 2004.

V C# – a v celé platformě .NET – je podpora generického programování bezesporu nejvýraznější novinkou verze 2.0 uvolněné koncem roku 2005. Číslování se zde týká jak jazyka C#, tak i platformy .NET; ve skutečnosti je genericita podporována i v jazyce IL, do něhož se z vyšších programovacích jazyků překládají všechny programy pro tuto platformu.

2. CO JE GENERICKÉ PROGRAMOVÁNÍ

Často se setkáváme s potřebou implementovat omezenou nebo neomezenou množinu funkcí, metod nebo datových typů, které se navzájem liší pouze datovými typy parametrů funkcí nebo hodnotami určitých konstant.

2.1 Příklad: Neomezená množina funkcí

Typickým příkladem může být úloha implementovat funkci (statickou metodu) pro určení větší ze dvou hodnot. Pro běžný celočíselný typ může ve kterémkoli z uvedených jazyků vypadat např. takto:

```
int max(int a, int b)
{
   return a < b ? b : a;
}
```

Implementace analogické funkce pro jiné číselné typy – nebo pro jakýkoli datový typ, pro nějž je definován operátor < – se liší jen typy parametrů.

Vzhledem k možnostem přetěžování operátorů v C++ a v C# je množina možných funkcí, které zde může programátor požadovat, prakticky neomezená, i když algoritmus je ve všech případech stejný.

2.2 Příklad: Neomezená množina tříd

Tradiční příklad, který se používá jako motivace pro zavedení genericity, představují kontejnery – třídy, jako jsou seznamy, fronty atd., jejichž instance slouží pro ukládání jiných dat. Bez generického programování máme dvě základní možnosti:

- **Můžeme implementovat beztypové kontejnery**. Tento postup je zpravidla založen na polymorfizmu v důsledku dědění – všechny datové typy, jejichž instance se do kontejneru ukládají, jsou odvozeny od téže třídy, typicky jménem `Object`. (V neobjektových programovacích jazycích, jako je C, se místo toho používají ukazatele bez doménového typu.)
  Výsledkem je univerzální kontejner, do kterého lze uložit cokoli – číselnou hodnotu vedle součásti grafického rozhraní programu. Uložením do beztypového kontejneru se ovšem ztrácí informace o typu uložené hodnoty, takže při vyjímání dat z kontejneru je třeba mít tuto informaci k dispozici (někde mimo kontejner). Případné chyby nemůže odhalit překladač, projeví se až za běhu, a to při použití vyjmuté hodnoty. *Chyba se tedy projeví jindy a na jiném místě, než kdy a kde vznikla.*
  Poznamenejme, že v jazycích, jako je C# verze 1 nebo Java verze 1.4 a starší, bylo k dispozici pouze toto řešení. Narazíme na ně i v některých proprietárních knihovnách dodávaných s překladači C++ z doby před zavedením šablon.
- **Můžeme implementovat jednoúčelové kontejnery s typovou kontrolou.** Takový kontejner umožňuje ukládání hodnot jediného typu (nebo množiny objektových typů odvozených od společného předka). Případné chyby – například uložení čísla místo prvku grafického uživatelského rozhraní – dokáže odhalit a přesně lokalizovat již překladač. Na druhé straně takovéto kontejnery jsou jednoúčelové; máme-li k dispozici kontejner na grafické objekty, nemůžeme ho použít k ukládání hodnot typu `int`, i když kontejner na typ `int` má zcela stejnou strukturu a používá zcela stejné algoritmy jako kontejner na grafické objekty a jediné, v čem se liší, je právě typ ukládaných hodnot.

Zde tedy potřebujeme nástroj, který umožní deklarovat množinu datových typů, které se liší jen v datových typech některých složek. I tato množina je prakticky neomezená.

2.3 Generické programování

Předchozí příklady ukazují potřebu parametrizovaných neboli generických konstrukcí – datových typů, metod, případně dalších součástí programů. Přitom „parametrizace“ zde znamená, že měnitelným parametrem je datový typ.

Smyslem generického programování tedy je zavést do programování další vrstvu abstrakce, která umožní vyhnout se psaní opakujícího se kódu, a to především pomocí parametrizace na základě datových typů.

Současné generické programování si klade následující cíle [6]:

- Poskytnout programátorovi způsob, jak vyjádřit algoritmy s minimální závislostí na použitých datových strukturách.
- Poskytnout programátorovi způsob, jak vyjádřit datové struktury s minimální závislostí na algoritmech.
- Poskytnout programátorovi možnost implementovat algoritmy co nejobecněji, aniž by při přechodu ke konkrétním typům došlo ke ztrátě efektivity.
- V případě, že zcela obecná forma algoritmu je v některých speciálních případech neefektivní nebo není použitelná, dát programátorovi možnost tyto speciální případy implementovat zvlášť.
- V případě, že obecná datová struktura pro jisté speciální případy nevyhovuje, poskytnout programátorovi možnost implementovat tyto speciální případy zvlášť.
- V případě, že k řešení určitého problému existuje několik rovnocenných algoritmů, poskytnout programátorovi možnost implementovat je všechny a dát tak uživateli na vybranou podle dalších kritérií.

2.4 Příklad generické funkce: výpočet maxima

Učebnicovým příkladem generické implementace algoritmu je šablona funkce počítající minimum. Obecný případ vypadá v C++ takto:

```
template <typename T>   //  Obecný tvar algoritmu
T min(T a, T b)
{
     return a < b ? a : b;
}
```

Uvedenou implementaci ovšem nelze použít pro lexikografické porovnávání znakových řetězců, takže je nezbytné deklarovat také specializaci – zvláštní implementaci – pro typ `char*`.

```
template<>               // Zvláštní případ pro určitý
char *min(char *a, char *b)              // datový typ
{
     return strcmp(a, b) < 0 ? a : b;
}
```

Pro řešení téhož problému v Javě 5 musíme použít statickou metodu:

```
class Pomocná
{
    public static <T extends Comparable> T max(T a, T b)
    {
       return (a.compareTo(b) > 0) ? a : b;
    }
    // a další složky této třídy
}
```

Řešení v C# 2.0 je na první pohled velmi podobné:

```
class Pomocná
{
    public static T Max<T>(T a, T b) where T : IComparable
```

```
    {
       return a.CompareTo(b) < 0 ? b : a;
    }
    // a další složky této třídy
}
```

Máme-li dvě proměnné

```
int a, b;
```

použijeme šablonu v C++ zápisem

```
int c = max(a, b);            // Volání v C++
```

ve zbývajících dvou jazycích musíme napsat

```
int c = Pomoc.max(a, b);      // Volání v C# a v Javě
```

neboť `max<>()` v nich představuje statickou metodu.

Výsledkem bude ve všech případech volání funkce, jež spočítá požadované maximum.

Jak však dále uvidíme, přes povrchní syntaktickou podobnost se pod těmito zápisy skrývají velmi rozdílné mechanizmy.

3. ZPŮSOBY IMPLEMENTACE GENERICKÉHO PROGRAMOVÁNÍ

Generické programování lze implementovat nejméně třemi různými způsoby.

- Použijeme nástroj podobný preprocesoru jazyka C, v němž lze vytvářet makra s parametry. Kontejner nebo funkci naprogramujeme jako makro, pro v němž budou některé datové typy vyjádřeny parametry, jejichž hodnoty dosadíme před použitím.

- Zavedeme syntaktickou konstrukci, která oznámí překladači, že u hodnot vkládaných do kontejneru požadujeme silnější typovou kontrolu. To znamená, že dostaneme jedinou kontejnerovou třídu obsahující odkazy na typ `Object` (nebo na jiný objektový typ, který je společným předkem typů hodnot ukládaných do kontejneru), překladač však bude vědět, že určitá instance obsahuje hodnoty jednoho přesně určeného typu a podle toho s ním bude zacházet. Podobně můžeme zavést konstrukci, která vhodným způsobem zajistí typy parametrů předávaných funkci či metodě.

- V prostředích, kde se o běh programu stará běhový systém („virtuální stroj“), může generickou konstrukci převzít tento běhový systém a instance podle okolností vytvářet za běhu. Správnost parametrů nebo ukládaných hodnot přitom může kontrolovat již překladač.

První dvě možnosti představují různá pojetí statické genericity, která se uplatňuje pouze v době překladu. Třetí možnost představuje dynamickou genericitu.

Nyní se podívejme, jak genericitu implementují uvedené programovací jazyky. Jednotlivé jazyky uvedeme v abecedním pořadí.

3.1 Jazyk C podle standardu ISO 9899 – 1999

Jazyk C genericitu jako takovou nepodporuje. Jeho preprocesor sice dovoluje vytvářet makra, jež umožňují simulovat genericitu podle prvního z uvedených přístupů, nejedná se ale o genericitu v pravém smyslu slova.

Nejnovější verze tohoto jazyka ovšem obsahuje v hlavičkovém souboru `<tgmath.h>` předdefinovaná *typově generická makra*, jež umožňují používat stejný identifikátor pro matematické funkce s reálnými a komplexními parametry. Jde v podstatě o náhradu přetěžování funkcí, které jazyk C nepodporuje.

3.1 Jazyk C++ podle standardu ISO 14882–2003

Jazyk C++ používá první z uvedených možností.

Šablona v C++ představuje neomezenou množinu objektových typů, metod nebo volných funkcí, které v případě potřeby vytvoří již překladač. Takto vytvořené typy, metody nebo funkce se nazývají *instance šablony* a pro různé hodnoty parametrů představují různé typy, resp. různé metody nebo volné funkce (odpovídá jim různý strojový kód). Šablona v C++ tedy existuje jen ve zdrojovém kódu; přeložený program obsahuje pouze instance vytvořené překladačem.

Tento přístup má některé zajímavé důsledky:

- Požadavky na vytvoření instancí musejí být známy v době překladu.
- Parametry šablon v C++ mohou být nejen jakékoli datové typy, ale i konstantní číselné hodnoty, ukazatele, reference a dokonce i jiné šablony.
- Datový typ, použitý jako skutečný parametr šablony, *nelze* v C++ syntakticky omezit – to znamená, že *z deklarace šablony nejsou zřejmé požadavky kladené na typy skutečných parametrů*. Jejich porušení se pak při překladu často projeví podivnými syntaktickými chybami, zdánlivě nesouvisejícími se skutečnou příčinou.
- Nástroje pro generické programování poskytují možnost přesnější specifikace algoritmu nebo datové struktury pro speciální hodnoty parametrů (parciální a explicitní specializace, přetěžování šablon volných funkcí atd.)
- Šablony v C++ lze využít jako výpočetně úplný nástroj k programování překladače (generické metaprogramování, viz [4]).
- Instance šablon funkcí lze vytvářet implicitně.
- Implementace šablon klade mimořádné nároky na tvůrce překladače. V současné době např. téměř žádný překladač nevyhovuje plně standardu [1].
- Při šíření šablon je třeba dodávat i zdrojový kód.
- Současné implementace jazyka C++ zpravidla neumožňují oddělený překlad šablon. To znamená, že nejen deklarace, ale i definice šablony musí být v místě použití viditelná. Úplná syntaktická kontrola šablony probíhá až při překladu jejího použití, tedy při vytváření instance. (Standard [1] jazyka C++ sice zavádí klíčové slovo `export`, jehož použití by mělo umožnit oddělenou kompilaci, ale s plnou implementací se setkáme jen výjimečně, neboť vede ke značným problémům.)

3.2 Jazyk Java 2, JDK 5

Jazyk Java 5 se opírá o druhý z výše uvedených přístupů k implementaci generického programování. Překlad parametrizované třídy, rozhraní nebo metody (volné funkce Java neobsahuje) je založen na procesu *vymazání* (erasure), který lze zjednodušeně popsat takto:

- Ze záhlaví třídy (resp. metody) se odstraní lomené závorky s parametry představujícími typ (tzv. formální typy).
- Všechna použití formálních typů v těle třídy se nahradí typem `Object` (popřípadě typem uvedeným v omezení parametrů generické konstrukce).

Výsledkem je *surová* třída (rozhraní, metoda), která se objeví v bajtovém kódu. Jméno surového typu vznikne ze jména parametrizovaného typu vypuštěním formálních typů; podobně jméno surové metody vznikne vypuštěním formálních typů ze jména parametrizované metody.

Výsledkem je jediná třída, resp. jediná metoda se zesílenou typovou kontrolou. Vraťme se ke kontejnerům: Při vkládání překladač kontroluje, zda jsou předávané hodnoty odpovídajícího typu, a při vyjímání připojí automaticky přetypování z typu `Object` na typ daný typovým parametrem.

Podívejme se na některé vlastnosti genericity v Javě. Mnohé z nich jsou přímým důsledkem zvoleného přístupu (tedy procesu vymazání typů):

- Formální typ *lze* v Javě 5 syntakticky omezit – to znamená, že z deklarace parametrizovaného typu nebo metody jsou zřejmé požadavky kladené na typy skutečných parametrů. Ve srovnání s C++ to výrazně zpřehledňuje diagnostiku.
- Proces překladu je ve srovnání s C++ podstatně jednodušší.
- Podobně jako v C++, ani v Javě nemají generické konstrukce přímý obraz v přeloženém programu, zde v bajtovém kódu. V důsledku toho lze vytvořený bajtový kód převést na bajtový kód pro starší typy virtuálního stroje Javy. (První verze překladačů Javy 5 dokonce produkovaly bajtový kód, který bylo možno spouštět na starších verzích JVM.)
- Parametry generických konstrukcí mohou být pouze objektové datové typy. Primitivní typy je třeba nahradit jejich obalovými třídami.
- Generické konstrukce založené na stejném surovém typu nelze rozlišit pomocí nástrojů reflexe.
- Generické metody nelze přetěžovat.
- Generické třídy a třídy obsahující generické metody lze samostatně překládat (plná syntaktická kontrola generické konstrukce není závislá na kontextu použití). V důsledku toho může knihovna obsahovat přeložené generické třídy.
- Formální datové typy nelze – na rozdíl od C++ – použít jako typy statických datových složek nebo jako typy ve statických metodách.
- V parametrizovaných třídách nelze – opět na rozdíl od C++ – vytvářet instance formálních typů.
- Nemáme k dispozici mechanizmus parciálních ani úplných specializací.
- Typové parametry lze při volání generických metod za jistých okolností vynechat. To můžeme považovat za na analogii implicitního vytváření instancí v C++.
- Mechanizmus parametrizovaných typů v Javě 5 zná tzv. žolíky. S jejich pomocí lze např. deklarovat metodu, jejímž skutečným parametrem bude instance parametrizovaného typu s libovolným parametrem.

3.3 C# verze 2.0

Implementace nástrojů pro generické programování v C# verze 2.0 vychází v podstatě z třetí možnosti, tedy z dynamické genericity. Není však v tomto směru zcela důsledná.

Mezijazyk IL (bajtový kód prostředí .NET), který vznikne překladem generického typu nebo metody, obsahuje metadata, jež identifikují daný typ jako generický. Způsob použití se pak liší podle toho, zda programátor použije při vytváření instance jako skutečný parametr hodnotový nebo referenční (odkazový) typ.

- V případě hodnotových typů vytvoří běhový systém jednu instanci pro každý jednotlivý typ parametru, a to v okamžiku použití – tedy např. deklarace instance.
- V případě odkazových typů vytvoří běhový systém v okamžiku prvního použití jednu instanci generického typu, která bude společná pro všechny odkazové typy.

Řešení pro odkazové typy je velmi podobné jako v Javě 5. Důvodem je, že tak lze snadněji udržet kontrolu nad počtem instancí.

Podívejme se opět na některé důsledky tohoto řešení.

- Generické konstrukce lze používat i pro atomické typy.
- Formální typ *lze* – podobně jako v Javě 5 – syntakticky omezit. Lze předepsat omezení na třídy, na hodnotové typy, na typu s bezparametrickým konstruktorem, na typy odvozené od daného typu nebo na typy implementující jeden nebo několik rozhraní.
- Nemáme k dispozici mechanizmus parciálních ani úplných specializací.
- Zavedení generických typů do C# (a do platformy .NET) vyžadovalo zásah jak do jazyka IL, tak i do celého společného běhového systému CLR.
- Instance generických typů lze rozlišit pomocí nástrojů reflexe.
- Při volání generických metod lze vynechat typové parametry, pokud si je dokáže překladač odvodit.
- Generické třídy nebo třídy obsahující generické metody lze překládat samostatně, odděleně od použití.
- Generické metody mohou být přetěžovány na základě počtu typových parametrů.

C# je tedy jediný ze srovnávaných jazyků, v němž generické konstrukce existují i za běhu programu.

4. SHRNUTÍ

Cílem tohoto příspěvku nebylo dokazovat, že některý z programovacích jazyků je v nějakém smyslu lepší než jiný, nebo dokonce porovnávat generické programování s jinými paradigmaty. Ani programovací jazyky, ani programovací paradigmata nestojí v protikladu, ale navazují na sebe a jejich společným clem je maximálně usnadnit práci programátora.

Každý ze zde uvedených jazyků přistupuje ke generickému programování jiným způsobem, přičemž odlišnosti se týkají především způsobu překladu. Srovnání nejdůležitějších charakteristik nabízí následující tabulka, v níž porovnáváme C++, C# a Javu..

	C++	C# 2.0	Java 5
Oddělený překlad	ne	ne	ano
Explicitní specializace	ano	ne	ne
Parciální specializace	ano	ne	ne
Explicitní omezení typů parametrů	ne	ano	ano
Implicitní vytváření instancí	ano	ano	ano
Základní typy jako parametry	ano	ano	ne
Žolíky	ne	ne	ano
Přetěžování generických funkcí	ano	ano	ne

Poděkování

Tento příspěvek vznikl v rámci grantů MŠMT Kontakt ME 492 a 1P04LA211.

LITERATURA

1. *International Standard ISO/IEC 14882:2003*. Programming Languages — C++.
2. *Dokumentace k JDK 5.* http://java.sun.com/j2se/1.5.0/download.jsp
3. *Generics in the Runtime (C# Programming Guide).* http://msdn2.microsoft.com/en-us/library/f4a6ta2h
4. M. Virius: *Fascinující svět šablon v C++*. Ve sborníku *Objekty 2004*, ed. D. Ježek a V. Merunka, Česká zemědělská univerzita v Praze 2004, str. 305.

5. M. Virius: *Generické programování v C++, Javě a C#.* Ve sborníku *Objekty 2005*, ed. V. Snášel, VŠB – Technická univerzita Ostrava 2005, str. 202.
6. R. Garcia J., Järvi. A., Lumsdaine, J. Siek, J. Willcock: *A Comparative Study of Language Support for Generic Programming.* Ve sborníku *OOPSLA'03*, str. 115–134.
7. Hall James, Merunka Vojtěch, Polák Jiří et al., *Accounting information systems - Part 4: System development activities* , 4th ed., Thomson South-Western New York 2004, ISBN 0-324-19202-9