VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLAD PODMNOŽINY JAZYKA PHP DO C++

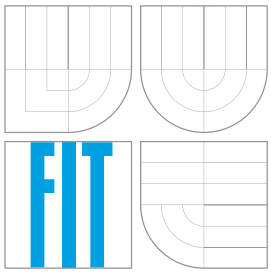
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

STANISLAV NECHUTNÝ

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLAD PODMNOŽINY JAZYKA PHP DO C++

TRANSLATION OF PHP LANGUAGE SUBSET INTO C++

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — STANISLAV NECHUTNÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYNĚK KŘIVKA, Ph.D.
SUPERVISOR

BRNO 2016

Abstrakt

Tato práce se zaměřuje na návrh a tvorbu nástroje pro automatizovaný překlad funkcí napsaných v podmnožině jazyka PHP do C++. Vygenerovaný zdrojový kód je možno zkompilovat jako rozšíření PHP a zavést stejným způsobem jako například MySQL, PDO, GD apod. Ve výsledku je tedy možno zavolat tyto funkce z PHP, jako by se jednalo o původní interpretovanou funkci. Předpokladem je však rozdíl v rychlosti vykonávání, protože odpadá analýza zdrojových kódů, jejich interpretace, či režie způsobená správou paměti. Vytvořený nástroj provádí převod zdrojového kódu do abstraktního syntaktického stromu, staticky jej analyzuje pro určení datových typů proměnných, a následně provádí generování C++ kódu. Výsledné zrychlení pak záleží na charakteristice překládaného kódu a praktické použití je prozatím komplikované kvůli implementaci podmnožiny PHP.

Abstract

My work is focused on design and execution of an automated translation for functions written in PHP into C++. Generated code may be compiled as a PHP extension and loaded the same way MySQL, PDO, GD or so. As a result these functions may be called from PHP as if they were the initial interpreted functions. Since there is no need for source code analysis, interpretation, nor staging by Garbage Collector general assumption would be a significant speed difference. Created tool executes source code transfer into abstract syntactic tree which is followed up by a static analysis of variable types and consequently generates C++ code. Final speed increase then depends on the particular code being translated and its practical use is slightly elaborate at the moment - owing to the implementation of PHP subset.

Klíčová slova

PHP, C++, C++11, překlad, modul, rozšíření, optimalizace, PHP-CPP, xdebug, datové typy, analýza kódu, PHC, HPHPc, HHVM, Testy řízené programování, generování kódu, tokeny, výrazy, precedenční analýza, konverze datových typů, transformace

Keywords

PHP, C++, C++11, translation, module, extension, optimization, PHP-CPP, xdebug, data types, code analysation, PHC, HPHPc, HHVM, Test-driven development, code generating, tokens, expressions, precedence analysis, data type conversion, transformation

Citace

Stanislav Nechutný: Překlad podmnožiny jazyka PHP do C++, bakalářská práce, Brno, FIT VUT v Brně, 2016

Překlad podmnožiny jazyka PHP do C++

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Křivky.

. . . . . .
Stanislav Nechutný
15. května 2016

Poděkování

Děkuji svému vedoucímu panu doktoru Křivkovi za nasměrování správným směrem a doporučení literatury. Dále pak Bc. Marku Milkoviči, za konzultace k jazyku C++ a analýze zdrojových kódů.

Obsah

Kapitola 1

Úvod

Skriptovací jazyk PHP je velmi oblíbeným prostředkem pro tvorbu dynamických webových systémů i pro různé skripty. Z původně jednoduchého jazyka pro tvorbu stránek s dynamickým obsahem vznikl jazyk se stovkami funkcí umožňující rychlou implementaci téměř jakékoliv úlohy.

V důsledku přesouvání mnoha služeb do prostředí webového prohlížeče výrazně narůstá počet a náročnost prováděných operací v PHP. Od generování PDF souborů[1], přes podepisování[2] a komprimování souborů až po náročné operace jako je například detekování osob v fotografiích[3] a podobně. Právě takové knihovny jsou často psané v PHP pro jednoduchost vývoje. PHP však umožňuje zavádět rozšíření s funkcemi v nativním kódu, jejichž vykonávání je výrazně rychlejší.

Cílem této práce je řevod knihoven v PHP na ekvivalent v jazyce C++ připravený pro zkompilování jako rozšíření PHP. Zadání vychází ze skutečnosti, že častým způsobem vývoje aplikace je použití jedné konkrétní verze knihovny, v které není třeba provádět úpravy. V důsledku pak výhody plynoucí z interpretace nejsou třeba, a naopak nevýhody znamenají propad výkonu.

1.1 Motivace

Modelový případ použití je pak tvorba PHP aplikace. Vývojář chce použít knihovnu například pro detekci obličeje ve fotografii. Jelikož takovou knihovnu nechce upravovat, ale pouze použít některé funkce z ní, tak může provést kompilaci pro její zrychlení. Knihovní funkce je pak vykonána rychleji a není narušena možnost rychlého a pohodlného vývoje aplikace. Tento koncept řeší nedostatky, na které naráželo podobné řešení HPHPc navržené společností Facebook, o kterém se zmiňuji v kapitole 2.2.

1.2 Funkcionalita

Zamýšlenou funkcionalitou je nástroj generující z adresáře se zdrojovými kódy v jazyce PHP výstup v jazyce C++. Výsledný kód by měl implementovat stejnou funkcionalitu a zachovat alespoň podobnou souborovou strukturu. Po zkompilování výstupu vznikne dynamická knihovna (.so, či .dll), kterou je možné nastavit v konfiguraci php prostřednictvím direktivy `extension`, či načíst během interpretace vestavěnou funkcí `dl()`.

Takto transformované načtené funkce a třídy bude možné v interpretovaném PHP používat zcela stejně jako v případě zdrojových – stejný způsob volání, stejné názvy, třídy půjde rozšiřovat apod.

[1] <http://www.mpdf1.com/mpdf/index.php>
[2] <http://phpseclib.sourceforge.net/>
[3] <https://github.com/mauricesvay/php-facedetection>

1.3 Cíle

Cílem práce je navrhnout a implementovat překladač funkcí zapsaných v podmnožině jazyka PHP, jehož výstupem bude C++ kód doplněný o potřebné konstrukce umožňující integraci do interpretru PHP.

Podmnožina jazyka PHP byla vybrána po konzultaci s vedoucím práce a zaměřuje se primárně na podporu výrazů a operací s proměnnými. Dále jsou podporovány základní konstrukce jako výpis hodnot, podmínky, cykly, funkce. Podrobněji jsou tyto konstrukce rozebrány v kapitole 3.

Po implementaci překladače pro požadovaný rozsah se zaměřím na jeho testování a hledání slabých míst. Následně se pokusím najít konstrukce, která nefungují korektně či jsou pomalé, a navrhnout řešení.

Kapitola 2

Konkurenční řešení

Provedl jsem analýzu současných konkurenčních řešení a nepodařilo se mi nalézt nástroj implementující takovouto funkcionalitu. Funkcionalitou podobný je nástroj phc[1], který slouží primárně pro kompilaci PHP kódu do spustitelných binárních souborů, které nepotřebují interpret. Dalším podobným a z technického hlediska zajímavým řešením je HPHPc od společnosti Facebook, kterým se budu zabývat v následující podkapitole a poslouží i jako zajímavá inspirace řešení.

2.1 phc

Phc je kompilátor podmnožiny jazyka PHP do spustitelných binárních souborů. Odstraňuje závislost na interpretru PHP a tak umožňuje pohodlnou distribuci CLI skriptů. Vývoj tohoto nástroje započali v roce 2005 vývojáři Edsko de Vries a John Gilbert. Postupně se do vývoje zapojilo na 21 vývojářů, poslední verze byla vydána v roce 2011[1]. Od té doby se pouze množí chybové reporty a projekt není udržován[2].

Nástroj podporoval také funkcionalitu, která je cílem této bakalářské práce, a bylo tedy možné transformovat PHP knihovny do rozšíření pro následné použití v PHP. Dle dokumentace na webu projektu nebyla však podpora pro objektové programování a chyběla implementace novějších jazykových konstrukcí[1].

S nejstarší v současné době podporovanou verzí PHP již není možné aplikaci ani přeložit, i přes mnohé pokusy o úpravy zdrojových kódů pro zajištění kompatibility.

Při analýze tohoto řešení jsem se rozhodl kontaktovat vývojáře a provést s nimi krátký rozhovor, v kterém bych rád zjistil odpovědi na několik otázek. Zamýšlel jsem zjistit v jakém stavu se tento projekt nachází, jaké byly plány do budoucna a proč došlo k jeho pozastavení. Dále mne zajímalo, zda i dnes považuje zainteresovaná osoba tento koncept za prakticky použitelný a jaké by mohli být překážky. Kontaktoval jsem postupně 3 bývalé vývojáře tohoto projektu, ale bohužel se mi nepodařilo získat odpovědi. Poslední kontaktovaný přislíbil odpovědi, ale e-mail s dotazy již zůstal bez reakce.

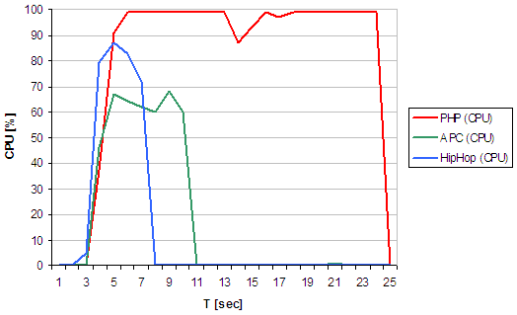
2.2 HPHPc

Tento projekt od společnosti Facebook byl započat v roce 2008 a měl za cíl snížit výkonnostní nároky aplikací psaných v PHP. To se také vývojářskému týmu podařilo a dosáhli snížení vytížení procesoru na polovinu[35]. V roce 2010 došlo k uvolnění zdrojových kódů projektu pod licencí PHP

[1] <http://www.phpcompiler.org/>

License a umožnění zapojení komunity do vývoje. V ruce 2013 došlo k ukončení tohoto projektu a nahrazení nástupcem Hip Hop Virtual Machine (HHVM).

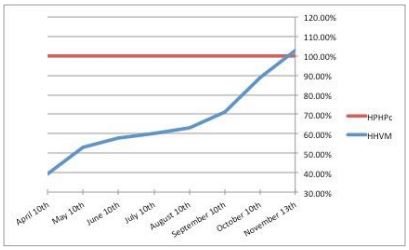
Testy provedené v dubnu 2011 ukazují na velký potenciál překladu PHP do nativního kódu[22]. Například práce s binárními stromy do hloubky 20 úrovní byla dle měření 4x rychlejší. Zajímavější byl ovšem test Hip Hop na překladu redakčního systému Drupal 7. Tento test ukázal na reálné aplikaci pětinásobné zrychlení oproti PHP 5.3.3[23]. Graf porovnávající Drupal 7 pod PHP 5.3.3, použití opcode cache APC a Hip HOP revize 806ee06 můžeme vidět na obrázku 2.1. Na grafu je zobrazeno procentuální vytížení procesoru a doba zpracování 300 požadavků s konkurencí 4.



Obrázek 2.1: Drupal 7.0 na PHP 5.3, APC a Hip Hop[23]

HPHPc provádělo kompilaci celých PHP aplikací do jednoho velkého spustitelného binárního souboru. Nasazování nových verzí tak bylo zpomaleno o kompilaci a distribuci až několik GB velkých souborů na produkční servery. Jedná se tedy o jiné cílové použití, než na jaké se zaměřuje tato práce. Nebylo možné využít výhod verzovacích systémů, kdy jsou distribuovány jen malé rozdíly v souborech, a v neposlední řadě nebyla podpora pro některé dynamické konstrukce, jako například `eval()`. Hlavní nevýhodou tohoto řešení byl problémový vývoj aplikace. Vývojář by byl nucen po každé změně provádět kompilaci, což by bylo časově náročné u takto rozsáhlé aplikace. V důsledku této komplikace vznikla implementace vlastního interpretru používajícího stejné implementace funkcí HPHPi, ale i tak docházelo k výskytu chyb na produkčních serverech, které se u vývojářů neobjevovaly. Důvodem bylo rozdílné prostředí. V důsledku těchto nevýhod bylo rozhodnuto, že se projekt vydá cestou JIT (Just in time) virtuálního stroje[29], který využívá bytecode generovaný ze zdrojových kódů PHP, jako můžeme znát třeba z jazyka Java.

Po sedmi měsících vývoje dosáhla implementace HHVM (Hip Hop Virtual Machine) lepší výsledků, než HPHPc a 19. února 2013 byla commitem odebrána ze zdrojových kódů podpora pro kompilování HPHPc[21]. Na grafu 2.2 můžeme vidět srovnání výkonu obou implementací v průběhu času.



Obrázek 2.2: Srovnání rychlosti HPHPc a HHVM v čase[29]

Řešení, které je navrženo a implementováno v této práci ponechává možnost pohodlné změny aplikace a cílí na překlad jednotlivých knihoven/funkcí. Klasický vývojový cyklus zahrnuje použití knihovny, z které jsou pouze volány funkce a není třeba provádět její úpravy. Mé řešení odstraňuje potíže vzniklé s kompilací při vývoji, nebo dvojím prostředím, u řešení HPHPc.

Kapitola 3

Vybrané konstrukce PHP

Vkládání prostřednictvím konstrukcí `include`, `require` a jejich variant pro unikátní vložení bude z důvodu komplexnosti implementováno jen v omezené míře. Prvním logickým omezením vycházejícím z návrhu tohoto nástroje je schopnost provést vložení pouze souborů, které jsou dostupné v momentě překladu a jejichž cesta je známá. Nebude tedy možná transformace funkce `includeNe` uvedené v kódu 3.1, ani funkce `includeNe2` i v případě existence souborů `prvni.php` a `druhy.php`.

Druhým omezením vkládání souborů je pouze částečná implementace varianty `include_once` a `require_once`, kdy nebude možné zcela zajistit neopakované vložení v době běhu aplikace. V době transformace PHP do C++ bude toto chování kontrolováno a dojde pouze k prvnímu vložení. V době vykonávání zkompilované funkce v prostředí interpretru PHP však nebude možné zamezit vložení, jelikož systém rozšíření PHP neumožnuje ovlivňovat tyto části.

```
1  function includeNe($soubor)
2  {
3    include $soubor;
4  }
5
6  function includeNe2($co)
7  {
8    $soubor = $co ? "prvni.php" : "druhy.php";
9
10   include $soubor;
11 }
```

Kód 3.1: Vložení souboru

Další implementovanou částí je podpora konstrukcí `echo`, `exit` a další uvedené v tabulce 3.1.

Typ	Token	Zápis
Komentář	T_COMMENT	`//`, `#`, `/* ... */`
	T_DOC_COMMENT	`/** ... */`
Výstup	T_PRINT	`print()`
	T_ECHO	`echo`
Vkládání	T_INCLUDE	`include`
	T_INCLUDE_ONCE	`include_once`
	T_REQUIRE	`require`
	T_REQUIRE_ONCE	`require_once`
Ukončení	T_EXIT	`exit()`, `die()`
Názvy	T_STRING	funkce, konstanty...
Proměnné	T_VARIABLE	`$a`

Tabulka 3.1: Tabulka tokenů konstrukcí

PHP umožnuje přistoupit k proměnné, jejíž název je uložen v jiné proměnné prostřednictvím konstrukce `$$promenna`, nebo `${$promenna}`. Tento zápis v současné implementaci nebude podporován, jelikož je třeba omezit implementovaný rozsah, a jedná se o potencionálně nebezpečnou konstrukci, stejně jako konstrukce `extract()` [13].

Při transformaci je podporován zápis konstant ve zdrojových souborech. Podporovaný je zápis čísel v celočíselném i desetinném formátu. Na základě těchto typů je přihlédnuto k volbě datových typů ve výstupním kódu. Dále je pak podporován zápis řetězců i s možností použití proměnných. Podporované tokeny pak podrobněji ukazuje tabulka 3.2

Řetězce je možné konkatenovat s čísly, poli a dalšími hodnotami. Tato funkcionalita prostřednictvím operátoru `.` je implementována. Dále se v řetězcích uvozených prostřednictvím znaku `"` mohou vyskytovat vložené proměnné, s případným přístupem k indexu pole. Tento zápis (tokeny T_ENCAPSED_AND_WHITESPACE a T_NUM_STRING) již nejsou podporovány z důvodu volby podmnožiny jazyka.

Typ	Token	Zápis
Desetinné číslo	T_DNUMBER	`1.2345`
Celé číslo	T_LNUMBER	`93`, `0xC0FFEE`
Řetězec	T_CONSTANT_ENCAPSED_STRING	`"retezec"`
Promenná v řetězci	T_ENCAPSED_AND_WHITESPACE	`"$a"`
Pole v řetězci	T_NUM_STRING	`"$a[0]"`

Tabulka 3.2: Tabulka tokenů konstant

Stěžejní částí aplikace je vyhodnocování výrazů. Pro implementaci byly zvoleny všechny tokeny pro matematické operace, porovnávání, bitové posuvy i přetypování viz tabulka 3.3.

Typ	Token	Zápis
Porovnání	T_IS_EQUAL	==
	T_IS_NOT_EQUAL	!=, <>
	T_IS_IDENTICAL	===
	T_IS_NOT_IDENTICAL	!==
	T_IS_GREATER_OR_EQUAL	>=
	T_IS_SMALLER_OR_EQUAL	<=
	T_SPACESHIP	<=>
Přiřazení	T_AND_EQUAL	&=
	T_CONCAT_EQUAL	.=
	T_DIV_EQUAL	/=
	T_MINUS_EQUAL	-=
	T_MOD_EQUAL	%=
	T_MUL_EQUAL	*=
	T_OR_EQUAL	|=
	T_PLUS_EQUAL	+=
	T_POW_EQUAL	**=
	T_SL_EQUAL	»=
	T_SR_EQUAL	«=
	T_XOR_EQUAL	^=
Logické členy	T_BOOLEAN_AND	&&
	T_BOOLEAN_OR	||
	T_LOGICAL_AND	and
	T_LOGICAL_OR	or
	T_LOGICAL_XOR	xor
Bitové posuvy	T_SL	»
	T_SR	«
Matematické operace	T_POW	**
	T_DEC	--
	T_INC	++
Přetypování	T_ARRAY_CAST	(array)
	T_BOOL_CAST	(bool)
	T_INT_CAST	(int)
	T_DOUBLE_CAST	(real), (float), (double)
	T_OBJECT_CAST	(object)
	T_STRING_CAST	(string)

Tabulka 3.3: Tabulka tokenů výrazů

Další podstatnou částí podmnožiny je podpora řídících konstrukcí. Podporovanými výrazy je zápis podmínek i cyklů. Z cyklů je podporovaný `for`, `while`, `do while` i `foreach`, který slouží pro iteraci nad všemi prvky v poli. Jednotlivé tokeny jsou uvedeny v tabulce 3.4.

Typ	Token	Zápis
Podmínka	T_IF	`if`
	T_ELSE	`else`
	T_ELSEIF	`elseif`
	T_ENDIF	`endif`
Cykly	T_DO	`do`
	T_WHILE	`while`
	T_ENDWHILE	`endwhile`
	T_FOR	`for`
	T_ENDFOR	`endfor`
	T_FOREACH	`foreach`
	T_AS	`as`
	T_ENDFOREACH	`endforeach`
	T_BREAK	`break`
	T_CONTINUE	`continue`

Tabulka 3.4: Tabulka tokenů řídících konstrukcí

3.1 Funkce

Podpora definice funkcí s i bez návratové hodnoty. Podpora argumentů, nepovinné argumenty s výchozí hodnotou.

- T_FUNCTION `function`
- T_RETURN `return`

3.2 Třídy

V důsledku udržení rozumného rozsahu podmnožiny byla vypuštěna objektově orientovaná část konstrukcí jazyka. Není tedy možné v překládaném kódu definovat ani vytvářet instance tříd, dědit, implementovat interface. Nejsou podporovány logicky ani jmenné prostory, modifikátory viditelnosti a statické metody. Pro úplnost uvádím tabulku nepodporovaných tokenů 3.5. Podpora OOP části jazyka je však plánována v budoucnu.

Typ	Token	Zápis
Definice	T_CLASS	`class`
	T_EXTENDS	`extends`
	T_IMPLEMENTS	`implements`
Konstanta	T_CONST	`const`
	T_INTERFACE	`interface`
Namespace	T_NAMESPACE	`namespace`
	T_NS_SEPARATOR	`\`
	T_USE	`use`
	T_NEW	`new`
Modifikátory	T_PRIVATE	`private`
	T_PUBLIC	`public`
	T_PROTECTED	`protected`
	T_STATIC	`static`
Přístup	T_DOUBLE_COLON	`::`
	T_OBJECT_OPERATOR	`->`

Tabulka 3.5: Tabulka tokenů tříd

Kapitola 4

Rozdíly PHP a C++

I přes velkou podobnost je rozdílů mezi oběma jazyky celá řada a je třeba se s nimi vypořádat.

4.1 Výrazy

Tabulka 4.1 na straně 13 obsahuje operátory jazyka PHP s jejich asociativitou seřazené dle priority – priorita se snižuje shora dolů. Druhou podstatnou tabulkou je 4.2 na straně 14, která obsahuje priority a asociativitu operátorů jazyka C++. Priorita je opět seřazena od nejvyšší po nejnižší. Při podrobnějším prozkoumání tabulky narazíme na několik rozdílů, se kterými se je třeba vypořádat.

Prvním je absence operátoru konkatenace `.` ze zdrojového jazyka PHP ve výsledném C++. Dále chybí operátor mocniny `**`.

Dalším chybějícím operátorem je `@`, který slouží pro potlačení případného chybového výstupu. Jeho použití je v kódu PHP nedoporučované,[19] [30] [34] [24] i přesto je možné provést implementaci tohoto chování, čímž se zabývám v kapitole 6.2.9 na straně 22.

Logické operátory `OR`, `AND`, `XOR` v PHP a `||`, `&&`, `^^` v C++ mají rozdílnou prioritu. Proto byla v kapitole 6.2.1 navržena transformace s explicitním závorkováním.

4.1.1 Porovnávání

Rozdíl v porovnávání proměnných v jazycích, kterými se zabývá tato práce, je poměrně zásadní. Zdrojový jazyk obsahuje dva typy porovnání, kdy je možné porovnávat odlišné datové typy. Konstrukce `"5"== 5` pak bude v PHP pravdivá, kdežto v C++ nikoliv. Pro porovnání i s přihlédnutím k typu je v PHP operátor === [7].

Problematickým chováním může být přetypovávání při porovnání. Nejen, že v případě, že jeden z operandů porovnání je číslo, může dojít k přetypování druhého operandu typu řetězec na číslo, ale i v případě porovnávání dvou řetězců, které mohou mít význam čísla. Například v PHP je pravdivé toto porovnání `'1e3' == '1000'` [28] [25].

Porovnávání instancí tříd prostřednictvím operátoru == je pravdivé, pokud jsou instancemi stejných tříd a všechny jejich vlastnosti mají stejnou hodnotu. Při porovnání === je porovnání pravdivé jen v případě totožných instancí [6].

[9]

Asociativita	Operátor	Další informace
neasociativní	clone, new	clone a new
levá	[	Pole
pravá	**	Výpočet
pravá	++, –, (int), (float), (string), (array), (object), (bool), @	(in/de)krementace, typy, utišení chyby
neasociativní	instanceof	Typy
pravá	!	Logické výrazy
levá	*, /, %	Výpočet
levá	+, -, .	Výpočty a řetězec
levá	«, »	Bitové operace
neasociativní	<, <=, >, >=	Porovnání
neasociativní	==, !=, ===, !==, <>, <=>	Porovnání
levá	&	Bitové operace a reference
levá	^	Bitové operace
levá	|	Bitové operace
levá	&&	Logické výrazy
levá	||	Logické výrazy
pravá	??	Porovnání
levá	?:	Porovnání
pravá	=, +=, -=, *=, **=, /=, .=, %=, |=, ^=, «=, »=, =>	Přiřazení
levá	and	Logické výrazy
levá	xor	Logické výrazy
levá	or	Logické výrazy
levá	,	Mnoho významů

Tabulka 4.1: Tabulka priority výrazů PHP[20]

Asociativita	Operátor	Další informace
levá	::	Scope resolution
pravá	++, –	Prefix inkrementace a dekrementace
	+, -	Unární plus a mínus
	!, ~	Logická negace a binární negace
	(type)	Přetypování (C styl)
	*	Dereference
	&	Reference
	sizeof	Size-of
	new	Vytvoření instance
	delete	Zrušení instance
levá	., ->	Ukazatel na člena
levá	*, /, %	Násobení, dělení, zbytek po celočíselném dělení
levá	+, -	Sčitání a odčítání
levá	«, »	Bitový posuv vlevo a vpravo
levá	<, <=	Porovnání - menší, menší nebo rovno
	>, >=	Porovnání - větší, většíší nebo rovno
levá	==, !=	Porovnání - rovno, nerovno
levá	&	Bitový AND
levá	^	Bitový XOR (exkluzivní OR)
levá	|	Bitový OR
levá	&&	Logický AND
levá	||	Logický OR
pravá	?:	Ternární operátor
	throw	Operátor výjimky
	=	Přímé přiřazení
	+=, -=	přiřazení součtem/odečtem
	*=, /=, %=	Přiřazení násobku, dělení, zbytku po celočíselném dělení
	«=, »=	Přiřazení výsledku bitového posuvu vlevo/vpravo
	&=, ^=, |=	Přiřazení bitového AND, XOR, OR
Levá	,	Čárka - mnoho významů

Tabulka 4.2: Tabulka priority výrazů C++[32]

4.2 Proměnné

Proměnné v PHP musejí dle dokumentace začínat znakem z rozsahu `{a-z,A-Z,_, 0x7f-0xff}` a další znaky mohou obsahovat také číslice[15]. I přesto existují způsoby, jak je možné definovat proměnnou nesplňující tuto podmínku. Například `$a = 5; $$a = 1;` umožní definovat proměnnou s názvem „5“ a hodnotou 1. Jazyk C++ vyžaduje, aby název proměnné začínal znakem z množiny `{ a-z, A-Z, _}`, další znak může být také číslo. Název nesmí být klíčové slovo (uvedeny v specifikaci C++11 kapitola 6.3.3.1). Oba jazyky rozlišují velikost písmen proměnných[32].

Z těchto specifikací názvů proměnných vyplývá několik omezení, které je třeba vyřešit. Použití znaků z rozsahu s kódem `0x7f - 0xff` není v kódu PHP příliš časté, a můžeme tedy pro účely nástroje zobrazit uživateli upozornění, že tento název není podporován. Název proměnné PHP, který je v jazyce C++ klíčovým slovem, či začíná číslicí, je možno upravit prefixem. Tím dojde i k zpřehlednění kódu, kdy bude jasně rozlišitelné, které proměnné jsou ekvivalentem proměnných z PHP

a které byly přidány v rámci transformace. Pro tyto účely byl zvolen prefix `phpVar_`.

Další částí, kterou je třeba vyřešit, je rozdílný obor platnosti proměnných. Specifikace jazyka C++ definuje tzv. Lokální obor, který má rozsah bloku uvnitř `{` a `}`. Tedy proměnná definovaná v těle funkce je dostupná uvnitř dalších vnořených bloků až do konce těla funkce. V případě těla cyklu nebo podmínky je pak proměnná definovaná v tomto těle dostupná do jeho konce. Dále je v C++ rozsah výrazu (Statement scope), kdy proměnná definovaná uvnitř `(` a `)` u konstrukcí `for`, `while`, `if` a `switch` je platná do konce těla[32]. Jazyk PHP obsahuje lokální obor funkce, kdy všechny proměnné, které jsou definovány uvnitř bloku, nebo by v C++ měly platnost pouze v rozsahu výrazu, jsou od daného místa dostupny dále i v nadřazených blocích[15].

Tento rozdílný obor platností proměnných je třeba řešit přesunem deklarace proměnných do lokálního oboru funkce. Proto na začátku těla funkce bude generován blok se všemi proměnnými použitými uvnitř funkce, vnořených bloků a výrazů.

Kapitola 5

Integrace do PHP

Interpret PHP umožňuje zavádět dynamické knihovny tzv. moduly, které rozšiřují dostupné funkce a třídy v interpretovaném prostředí. Načtení je možné prostřednictvím konfigurační direktivy `extension=nazev.so` v konfiguračním souboru `php.ini`, či v době běhu funkcí `dl("nazev.so")`. Rozšíření obsahují tzv. entry point, tedy adresu, kam v momentě požadavku na načtení modulu skočí tok interpretru a očekává se vrácení datové struktury, která obsahuje informace o dostupných funkcích a třídách. Zend Engine[1], jádro PHP, je napsáno primárně v jazyce C, takže je třeba dodržet kompatibilní způsob volání a navrácení výsledku. V případě této práce bude z důvodu jazyka C++ nutno použít konstrukci `extern "C"`[31][10] [11].

Po inicializaci PHP jsou zavolány inicializační funkce rozšíření, které umožňují provést alokaci paměti, otevření deskriptorů k souborům apod. V tomto kroku je například vhodné, aby rozšíření pro logování otevřelo soubor, do kterého bude zapisovat, jelikož bude pro všechny požadavky stejný a je zbytečné jej otevírat a zavírat při každém požadavku.

PHP proces čeká na příchozí požadavek k odbavení. Při jeho přijetí nastaví potřebné proměnné a spustí interpretaci skriptu. Spolu s nastavením proměnných je zavolán další callback z rozšíření, který může připravit k načtení již konkrétní hodnoty pro daný požadavek - může tím být například vlastní implementace session.

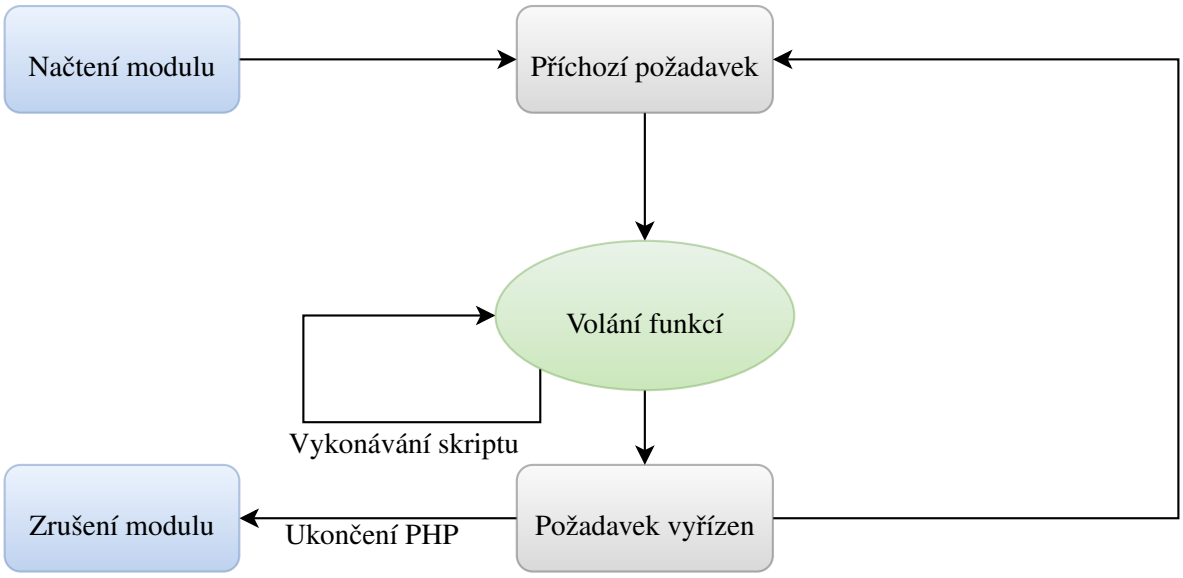
Následně dochází k případným voláním funkcí při vykonávání skriptu obsluhujícího požadavek. Po dokončení požadavku je zavolán další callback, který umožňuje provést uvolnění paměti a další operace pro funkcionalitu rozšíření. Z důvodu optimalizace nedochází po obsloužení požadavku k ukončení procesu PHP, ale proces čeká na další příchozí požadavek. Pokud dojde k vypnutí webového serveru, tak je zavolán další callback, který by měl provést úklid zbylých zdrojů - protiklad k prvně volanému callbacku. Celý tento proces znázorňuje diagram na obrázku č. 5.1.

5.1 Zend engine

Datová struktura `zend_module_entry` obsahuje následující hodnoty [10]:

- Verze Zend API, pro kterou je modul zkompilován.
- Seznam závislostí modulu na jiných modulech.
- Název modulu.
- Ukazatel na pole struktur se seznamem funkcí (`zend_function_entry`).

[1] <http://www.zend.com/en/community/php>



Obrázek 5.1: Životní cyklus rozšíření od startu PHP s obsluhou N požadavků po ukončení

- Ukazatel na funkci volanou při zavedení modulu.
- Ukazatel na funkci volanou při rušení modulu.
- Ukazatel na funkci volanou při každém příchozím požadavku na stránku (při použití PHP jako SAPI modul webového serveru Apache).
- Ukazatel na funkci volanou při konci požadavku na stránku.
- Funkce vracející informace při volání `phpinfo()`.
- Verze modulu.
- Globální proměnné modulu.

Struktura `zend_function_entry` pak obsahuje [5]:

- Jméno, pod jakým bude funkce dostupná v PHP.
- Ukazatel na funkci.
- Argumenty funkce - zda mají být předány referencí.

Tyto hodnoty je třeba nastavit v každém rozšíření, aby interpret mohl exponovat funkce do interpretovaného prostředí.

zval

Tato struktura je využívána Zend enginem pro data proměnné. Vytvoření instance struktury je zjednodušeno makrem `MAKE_STD_ZVAL` a následné přiřazení hodnoty určitého typu je možno pomocí různých maker – například `ZVAL_DOUBLE`. Tyto struktury obsahují několik prvků - C `union` prvků pro uložení různých datových typů (`long`, `double`, `char*`, `HashTable`, `zend_object_value`), dále počet refencí na hodnotu (PHP implementuje z důvodu zlepšení rychlosti tzv.

líné kopírování), další položkou je typ hodnoty a posledním členem struktury jsou příznaky pro správu paměti.[26] [16] [17]. Podrobněji zkoumat tyo struktury naplněné daty je možné pomocí funkce `xdebug_debug_zval` z rozšíření xdebug.

5.2 PHP-CPP

Tato knihovna přidává abstrakci pro definici rozšíření a práci s hodnotami z Zend Engine. Přináší třídu s přetíženými operátory, které ulehčují vývoj, ale interně jsou stále použity zval struktury, takže dochází k zpomalení způsobenému zpomalením přístupu k hodnotě. Při načítání hodnot do registru musí procesor načíst položku značící datový typ. V případě rozdílného datového typu, než jaký je požadován, je nucen provést konverzi. Poté může přistoupit k položce union struktury a provést změnu hodnoty. Tento popsaný postup má za následek výpadek hodnot z registru procesoru, a proto se ukázal propad výkonu u výpočtu faktoriálu o několik řádů.

V případě použití nativních datových typů jazyka C byl výsledek výrazně lepší. Pro faktoriál z 100 000 000 000 byl průměrný čas na PHP 5.6.15 73s a 0.35s C++. Testovaný počítač byl vybaven procesorem Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz, 16 GB RAM 1 600 MHz, SSD s čtením 540 MB/s. Pro překlad bylo použito GCC 4.9.2 zkompilovaný Red Hatem s optimalizací -O2. Linuxové jádro 4.1.13-100 x64 na distribuci Fedora 21.

Dále se podrobněji touto knihovnou zabývám v kapitole 7.1.1.

Kapitola 6

Návrh transformace

Při převodu abstraktního syntaktického stromu na C++ zdrojový kód je třeba navrhnout pravidla a šablony, podle kterých bude transformace a generování kódu probíhat. Při tvorbě těchto pravidel bylo třeba podrobně prozkoumat dokumentaci a specifikaci obou jazyků a chybějící informace získat a ověřit experimentálně.

6.1 Jazykové konstrukce

6.1.1 Foreach

Tato konstrukce umožňuje v PHP iterování nad prvky pole. `Php::Value` implementuje iterátor, takže je možné pole projít pomocí nové vlastnosti z C++11 – `for( auto item : array ) { ... }`.

6.1.2 Konstanty

Jazyk PHP umožňuje definovat konstanty pomocí konstrukce `define`. Ty jsou pak dostupné po celý zbytek kódu pod zvoleným jménem a není možno měnit jejich hodnoty. Knihovna PHP-CPP opět přidává abstrakci a umožňuje tyto operace. Pro definování konstanty se tedy používá `Php::define("nazev", hodnota)` a pro získání (token `T_STRING`) volání `Php::constant(nazev)`.

6.1.3 Nepovinné argumenty

Funkce v PHP mohou mít nepovinné argumenty. Ty mají v zápise přiřazenou výchozí hodnotu a pokud není hodnota argumentu při volání specifikována, tak je použita tato výchozí. Knihovna PHP-CPP předává všechny argumenty jako ukazatel na instanci třídy `Php::Parameters`, což je objekt implementující přístup polem. Nepovinné argumenty jsou tedy řešeny pomocí ternárního operátoru. Použitá šablona pro generování takové proměnné je zde uvedena.

```
1 phpVar_<argName> = (args.size() >= <argPosition> ? args[ <argPosition> ] : <
      defaultValue>;
```

Kód 6.1: Šablona pro generování nepovinných argument funkce

6.1.4 Podmínky

V PHP je možno zapsat u podmínek `elseif` větev, jejíž chování je stejné jako `else if` v C++. Transformace tohoto výrazu je tedy přímá.

6.1.5 Návrat hodnoty z funkce

V PHP není povinnost, aby funkce vracela hodnotu z funkce. Dokonce je možné, aby hodnotu vrátila jen v jedné z podmíněných větví. Při překladu do C++ to je problém, protože pokud je definováno, že funkce vrací určitý datový typ, tak ho musí opravdu vracet. Pro řešení problému nám nahrává chování PHP, kdy pokud funkce nevrací výsledek, ale i přesto je výsledek volání přiřazen do proměnné, tak je „vrácenou“ hodnotou `NULL`. Postačí tedy na konec každé funkce doplnit `return nullptr;` a není třeba analyzovat větvení funkce a návratové hodnoty v nich.

6.2 Operátory

I přes podobnost operátorů v obou jazycích je třeba při překladu řešit konverzi datových typů ve výrazech a je tedy třeba definovat pravidla transformace.

6.2.1 Logické operátory

Jak je uvedeno v tabulce 4.1 priorit výrazů PHP na straně 13, tak jazyk PHP obsahuje pro logické operace dvě možnosti zápisu operací. Prvním způsobem zápisu s vyšší prioritou je zápis prostřednictvím operátorů `&&`, `||` a `^`. Tento zápis má vyšší prioritu než přiřazení. Při zápisu v kódu 6.2 na první řádce tedy bude výsledná hodnota v proměnné `$a` nepravdivá. Opakem je však zápis na řádku 2, kdy bude hodnota proměnné pravdivá, jelikož se s vyšší prioritou provede přiřazení a až s výsledkem přiřazení se provede AND, jehož výsledek se zahodí.

```
1 $a = TRUE &&  FALSE; //  $a = (TRUE AND FALSE)  -> $a == FALSE
2 $a = TRUE AND FALSE; // ($a = TRUE) AND FALSE   -> $a == TRUE
```

Kód 6.2: PHP ukázka rozdílných priorit operátorů

Jazyk C++ však obsahuje pouze operátory s vyšší prioritou než přiřazení. Kromě priority operátorů se chování neliší[12]. Je tedy třeba kromě nahrazení zápisu ve formě slova za funkčně odpovídající zápis z jazyka C++ také provést uzavření levé a pravé strany do závorek.

Výsledná transformace tedy bude $< op1 > OR < op2 > \rightarrow (< op1 >)||(< op2 >)$

6.2.2 Matematické operátory

Matematické operátory jako `+`, `-`, `*` a další jsou řešeny bez nutnosti výrazné úpravy kódu. Pouze je doplněna případná konverze datových typů na základě výsledků analýzy. Výjimkou je dělení a mocnina, které jsou popsány v následujících oddílech.

6.2.3 Dělení

V případě dělení čísel v PHP se neprovádí celočíselné, ale i pro výraz `5/4` je výsledkem desetinné číslo. Při transformaci operátorů dělení bude tedy třeba provést změnu typu na float/double. V jazyce C++ je rozdíl mezi výrazy `5/4` a `5.0/4.0`, kdy v prvním případě je výsledkem `1` a v druhém `1.2`. Při transformaci tedy bude třeba oba operandy dělení přetypovat na desetinné číslo. K tomu je použita funkce `php2cpp:to_float`, která je popsána v kapitole 7.4.

6.2.4 Mocnina

Pro výpočet mocniny čísla je možné nahradit operátor `**` za volání funkce `pow` ze standardní knihovny `math.h`[3]. Opět je třeba rozhodnout o případném doplnění převodu na číslo na základě analýzy typů.

Konverze tedy bude dle pravidla $< op1 > ** < op2 > \rightarrow pow(< op1 >, < op2 >)$.

6.2.5 Porovnání

Jazyk PHP obsahuje dva rozdílné způsoby porovnání hodnot. Jednodušší je porovnání prostřednictvím operátorů `===` a `!==`, kdy je prvně porovnán datový typ, a pokud se shoduje, tak hodnota. Složitější je porovnání pomocí operátorů `==`, `!=`, `>`, `<`, `>=` a `<=`, které provádějí automatickou typovou konverzi. Je tedy možné porovnat například řetězec `"4.5"` s číslem `0x05`.

PHP pro porovnání zval struktur s konverzí používá funkci `compare_function(result, a, b TSRMLS_CC);`. Ta napřed provede sjednocení datových typů hodnot. Pokud se jedná o řetězec, tak je použita funkce `zend_is_numeric`, která detekuje v řetězci případné celé číslo, nebo desetinné a vrátí výsledek. Tato funkce je navržena jako konečný automat přecházející mezi stavy. Následně jsou v závislosti na typu provedeny konverze na společný datový typ a provedeno porovnání. Informace o fungování a návrhu porovnávání byla zjištěna analýzou zdrojových kódů interpretru PHP.

Jak již bylo zmíněno, tak pro účely této implementace byla použita knihovna PHP-CPP, která obsahuje třídu `Php::Value` s přetíženými operátory. Mezi nimi je i přetížení operátoru `==`, a je tedy možné provádět porovnávání s konverzí typů klasickým způsobem. Pro porovnání bez konverze je pak třeba zkontrolovat datové typy obou operandů, a pokud jsou shodné, tak porovnat hodnoty.

6.2.6 Konkatenace

`std::string` obsahuje metodu `append`[4]. Je tedy možné transformovat kód 6.3 na 6.4.

```
1 $a = "a";
2 $b = "b";
3 $a .= $b;
```

Kód 6.3: PHP ukázka konkatenace

```
1 std::string a = "a";
2 std::string b = "b";
3 a.apend(b);
```

Kód 6.4: Transformace konkatenace

6.2.7 Pole

Pro práci s poli je použita instance třídy `Php::Value` z knihovny PHP-CPP, která umožňuje práci s poli jako v PHP. Interně je tato funkcionalita implementována přetíženými operátory a hodnoty jsou ukládány do `zval` struktur. Při přístupu k prvku pole (v PHP je pole hashovací tabulka) je vrácena instance třídy `Php::HashMember`, která drží referenci na prvek v poli, a přiřazení do této třídy provede pomocí přetíženého operátoru = změnu její hodnoty. Pokud je zamýšleno použití pro další operace, tak je třeba metodou `value` získat instanci `Php::Value`, která je kopií dat.

Z výše popsaných důvodů je transformace takového výrazu pak složitější. Při generování je třeba zohlednit kontext a podle toho použít dva způsoby zápisu.

Výraz $\$a[0] = \$a[1] = \$a[\text{"}a\text{"}] + 5$ pak ukazuje transformaci. Pravidlo pro přepis je $< var > [< index >] <= expr > \rightarrow (< var > [< index >] <= expr >).value()$. Výsledek pro část $\$a[\text{"}a\text{"}] + 5$ je $(a[\text{"}a\text{"}]).value() + 5$. Pro část $\$a[1] = \$a[\text{"}a\text{"}] + 5$ pak $(a[1] = (a[\text{"}a\text{"}]).value() + 5).value()$ a obdobně pro zbytek výrazu.

6.2.8 Volání funkce

Volání funkce vestavěné v PHP i uživatelem definované (interpretované) je poměrně jednoduché pomocí `Php::call`, kdy prvním argumentem je název a další argumenty jsou argumenty funkce. Argumenty mohou být typu `Php::Value` či jakýkoliv jiný, který je na něj možno převést. Tato funkce umožňuje provázání nejen směrem z interpretovaného skriptu do zkompilovaného (volání přeložených funkcí), ale i opačně – přeložený a zkompilovaný kód může volat funkci, která je definována v interpretovaném skriptu. V případě, že je volána nedefinovaná funkce, tak je zobrazena zcela stejná chyba, jako by byla volána z interpretovaného kódu. To umožňuje překládat jen určité části knihoven, které jsou náročné, či je není třeba upravovat.

Transformace takového kódu je tedy jednoduché pravidlo: $< name > (< args >) \rightarrow Php :: Call(< name >< , args >)$

6.2.9 Potlačení chyby

Pro potlačení chyby je třeba část ovlivněnou potlačením vyjmout do vlastní části kódu, přiřadit výsledek do pomocné proměnné a obalit voláním funkcí z jádra PHP viz kód 6.5. Z důvodu, že takové potlačení by ovlivnilo pouze volání vestavěných funkcí (překládaný kód nevyvolává PHP chyby) a jeho použití je silně nedoporučováno, jak již bylo zdůvodněno v předchozí části, tak tento operátor není prozatím podporován.

```
1 void zend_do_begin_silence(znode *strudel_token TSRMLS_DC);
2 // volaný kód
3 void zend_do_end_silence(const znode *strudel_token TSRMLS_DC);
```

Kód 6.5: Potlačení chybové hlášky[30]

Kapitola 7

Implementace

Pro implementaci byl zvolen jazyk PHP, v kterém je možné provádět rychle implementaci, a poskytuje potřebné funkce. Velkou výhodu pro implementaci je `token_get_all()`, která umožňuje naparsovat zdrojový kód a převést jej na tokeny. Po získání tokenů je provedena analýza rekurzivním sestupem shora dolu. Při analýze je generován abstraktní syntaktický strom pro generátor kódu. Parser využívá volání precedenční analýzy, která provádí kontrolu výrazů a provádí také tvorbu derivačního stromu, kterého je následně využito pro tvorbu abstraktního syntaktického stromu výrazů. Po získání všech potřebných informací ze zdrojového kódu je přistoupeno k analýze výrazů pro detekci datových typů a následně provedeno generování C++ kódu.

Prvně jsem provedl analýzu několika hotových řešení parsování zdrojových kódů, ale zjistil jsem, že rychlejším a pohodlnějším pro další vývoj bude tvorba vlastního řešení. Prozkoumaná řešení byla často velmi obecná pro pokrytí co největší části jazyka PHP a nedetekovala pak některé chybné konstrukce. Dále pak jejich výstup neobsahoval dostatek potřebných informací pro určování datových typů a vhodné sestavování C++ kódu. Mezi zkoumanými řešeními bylo i PHP-Parser[1], jehož autorem je Nikita Popov, který také přispívá do PHP interpretru a je jedním z autorů knihy „PHP Internals Book“, zabývající se tvorbou PHP rozšíření. Zajímavým a vhodným řešením s dobrým výstupem se ukázal PHP Analyzer[2] od společnosti Scrutinizer, který provádí velmi podrobnou analýzu. Problémem však byla vysoká cena, komplikující záměr poskytnout výsledné řešení jako open source produkt dostupný zdarma.

7.1 Knihovny

Při implementaci jsem se rozhodl použít open source knihovny poskytující některé potřebné funkce, než aby docházelo k jejich opakované tvorbě.

7.1.1 PHP-CPP

PHP-CPP[3] je knihovna od společnosti Copernica, která umožňuje tvorbu rozšíření PHP v jazyce C++ a přidává abstrakci nad interními Zend strukturami. Knihovna je šířena pod licencí Apache 2.0 a je ji tedy možno použít pro komerční účely a distribuovat i s případnými změnami.

Použitím třídy odpadá nutnost používat struktury rozšíření Zend enginu, které popisuji v kapitole 5, a je možné použít objektovou abstrakci. Dále umožňuje volání vestavěných funkcí PHP inter-

[1] <https://github.com/nikic/PHP-Parser>
[2] <https://scrutinizer-ci.com/docs/tools/php/php-analyzer/>
[3] <http://www.php-cpp.com/>

pretru, a není tedy potřeba jejich reimplementace. Volané funkce jsou v interpretru PHP implementovány v jazyce C s častým použitím inline assembleru pro vysokou optimalizaci. Jejich voláním tedy nedochází k zpomalení. Další, podstatnější výhodou je možnost použít třídu `Php::Value`, která implementuje přetížený konstruktor a přetížené operátory pro většinu datových typů a operací včetně porovnání s automatickou konverzí a implementuje i funkcionalitu polí.

Nevýhodou tohoto řešení je však jeho pomalost. Třída `Php::Value` je interně velmi provázána s Zend enginem a pro vnitřní uložení dat jsou použity tzv. `zval` struktury, které vnitřně používá i PHP interpret.[27] Ty obsahují mimo jiné i informace pro fungování garbage collectoru a tak, i když nejsou tyto informace v kompilovaném kódu použity pro správu paměti, konzumuje jejich generování procesorový čas.[26] Z tohoto důvodu, pokud je to možné, není tato třída použita pro proměnné a jsou použity primitivní datové typy `bool`, `long`, `double` a `std::string`.

7.2 Analýza zdrojových kódů PHP

Pro analýzu je třeba provést parsování zdrojových kódů do abstraktní struktury. Pro tento krok je možné použít funkce, které PHP nabízí. Konkrétně se jedná o funkci `token_get()` vracející tokeny uvedené v dokumentaci na adrese `<http://php.net/manual/en/tokens.php>`.

Při programování se však ukázalo, že tato funkce není dostatečně konkrétní pro zamýšlené potřeby implementace. Častým problémem bylo označení velkého množství konstrukcí tokenem `T_STRING`. Například v tabulce tokenů 3.3 na straně 9 nejsou uvedeny operátory +, -, /, * a další. To není z důvodu jejich absence ve zvolené podmnožině, ale absencí konkrétních tokenů. V tabulce priority operátorů 4.1 jsou uvedeny a jsou dle ní překládány. Jelikož výrazy jsou důležitou částí, tak byla doplněna další úroveň abstrakce, která tyto tokeny přeznačí z `T_STRING` na odpovídající vlastní tokeny - `T_PLUS`, `T_MUL` apod. Obdobný postup byl zvolen u dalších chybějících typů tokenů pro středník, složené závorky a další.

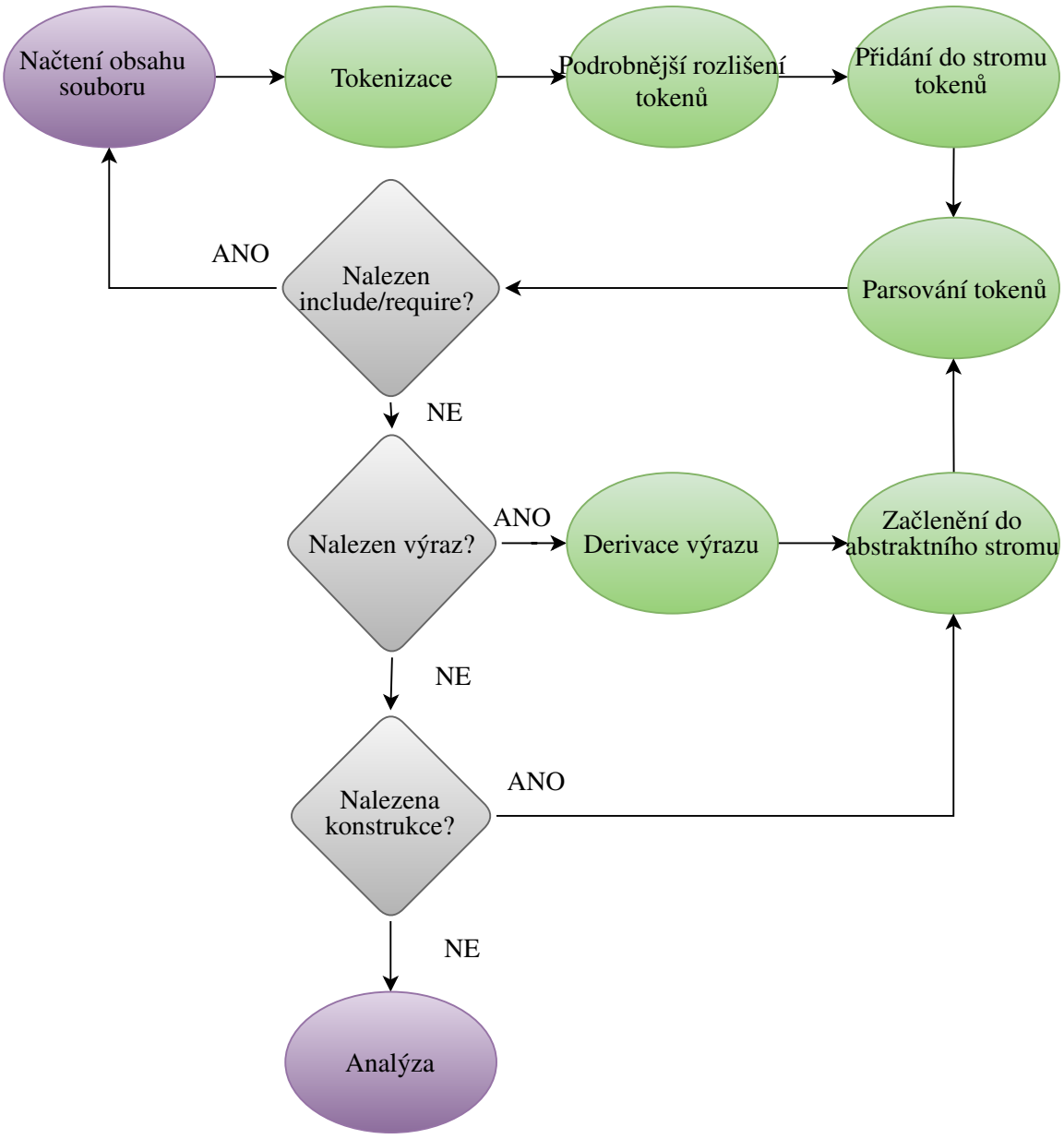
Po získání těchto tokenů ze zdrojového kódu se pokračuje jejich analýzou. V průběhu analýzy může být nalezena jedna z konstrukcí `include`, `include_once`, `require`, `require_once`. Pokud je možné v době překladu získat cestu k souboru (tedy neobsahuje například proměnné), tak je provedeno jejich vložení - získání tokenů a vložení na odpovídající místo již známých tokenů. V případě, že není možné provést vložení, je druh vyvolané chyby obdobný jako v době interpretace. Konstrukce `include` a `include_once` vedou pouze k vypsání chyby na standartní chybový výstup a pokračuje se v přikladu, kdežto u `require` a `require_once` je kromě výstupu také ukončen překlad[14].

Vlastní parser využívá metody rekurzivního sestupu shora dolů. Při průchodu sestavuje abstraktní syntaktický strom. Když narazí na výraz, tak je použita zásobníková precedenční analýza. Její derivační strom, který je výstupem, je pak opět převeden na abstraktní syntaktický strom a začleněn do stromu vzniklého v parseru.

Pro uložení informací o proměnných - datové typy, použití apod. jsou použity instance třídy `Variable` uložené v kontejneru, který tvoří instance třídy `Scope`. Informace jsou získávány z výstupu precedenční analýzy, jelikož v PHP může být použita proměnná ve výrazu i bez definice. Pak dojde, v závislosti na konfiguraci, k vypsání varování a hodnota takové proměnné je rovna `NULL`.

Po dokončení parsování tokenů zdrojového kódu je předáno řízení analyzátoru, který provádí detekci datových typů. Této detekci se podrobněji věnuji v následující podkapitole 7.3.

Postup této části analýzy zdrojových kódů je znázorněn na diagramu 7.1 na straně 25, který začíná načtením obsahu překládaného souboru a končí přechodem k analýze abstraktního syntaktického stromu pro detekci datových typů.



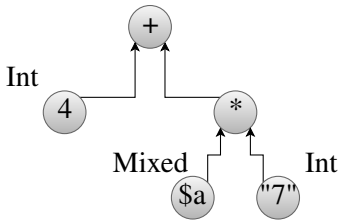
Obrázek 7.1: Analýza vstupního souboru

7.3 Detekce datových typů

Detekce datových typů je založena na statické analýze výrazů a jejím výsledkem je uložení 2 hodnot k položce stromu. Operuje se zde s vstupním datovým typem - tedy jakého typu je zdrojová hodnota, a druhým je výstupní typ – ten je výsledkem analýzy výrazu a použit pro konverzi a datové typy. Analýza je provedena dvěmi průchody stromem.

Při prvním průchodu metodou Inorder je analyzován strom výrazu a jsou doplněny vstupní datové typy listů stromu. Tato detekce je založena na jednoduché detekci datových typů na základě hodnot.

Pokud se jedná o číselnou konstantu, tak je doplněn typ `Int`. Obdobné je to s typem `Float` pro desetinná čísla, `Bool` pro logickou hodnotu (`True`, `False`), nebo typ `NULL`.



Obrázek 7.2: Detekce datových typů – krok 1

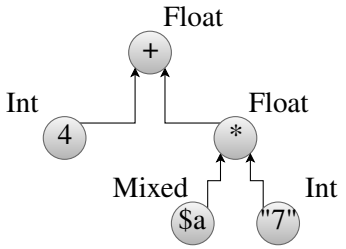
Při nalezení řetězce je postup složitější. Hodnota řetězce je načtena do proměnné v jazyce PHP a provedeno porovnání pomocí výrazu `$varValue == (int)$varValue`, čímž je provedena detekce, zda je hodnota v řetězci celým číslem. Pokud porovnání uspěje, tak je nastaven vstupní typ na `Int`, obdobná detekce se provede pro `Float` porovnáním zapsané hodnoty s hodnotou přetypovanou na `float`. Pokud není tato detekce úspěšná, tak je nastaven datový typ na `String`. Tato detekce vstupních hodnot v řetězcích je z důvodu optimalizace, aby se případně eliminovala potřeba převádět řetězec na číslo, jelikož to může být důvodem zpomalení výsledného kódu.

Pokud se jedná o proměnnou, tak je datový typ označen automaticky za `Mixed`, jelikož její vstupní datový typ může být odvozen pouze z předchozí práce s touto proměnnou (přiřazení), a není tedy možno s informacemi z tohoto jednoho výrazu detekovat typ. Výsledek prvního průchodu stromem výrazu `4 + $a * "7"` s označenými vstupními datovými typy je diagram 7.2.

Druhý průchod již operuje s informacemi z první iterace a je proveden metodou Postorder. Při tomto průchodu se analýza zaměřuje na operátory, které spojují jednotlivé větve stromu. Je zde aplikována sada pravidel založená na typu operátoru a operandů.

7.3.1 Pravidla

Pro detekci datových typů výrazů jsou použita tato pravidla. Pokud není možné aplikovat žádné z pravidel, tak je typ nastaven na `Mixed`. Na diagramu 7.3 je znázorněna vnitřní struktura po aplikaci těchto pravidel na operátory. Tato pravidla byla sestavena na základě dokumentace PHP, experimentů s interpretrem PHP 5.6.15 a PHP 7.0.1 a dalších knih [33][20][8].



Obrázek 7.3: Detekce datových typů – krok 2

Porovnání

Výstupní typ: `Bool`
Operátor: { `!`, `==`, `===`, `!=`, `!==`, `<=`, `>=`, `<`, `>`, `OR`, `AND`, `XOR`, `||`, `&&`, `^^`, `||=`, `&&=`, `^^=`}
Operandy: Libovolný typ

Matematické operace - celočíselné

Výstupní typ: `Int`

Operátor: { +, -, *, **, +=, -=, *=, **=, ++, - }
Operandy: {NULL, Bool, Int, String}

Matematické operace - desetinné

Výstupní typ: Float
Operátor: { +, -, *, **, +=, -=, *=, **=, ++, - }
Operandy: Alespoň jeden {Float, Mixed}

Matematické operace

Výstupní typ: Float
Operátor: { /, /= }
Operandy: Libovolný typ

Matematické operace

Výstupní typ: Int
Operátor: { %, %= }
Operandy: Libovolný typ

Konkatenace

Výstupní typ: String
Operátor: { ., .= }
Operandy: Libovolný typ

Přiřazení

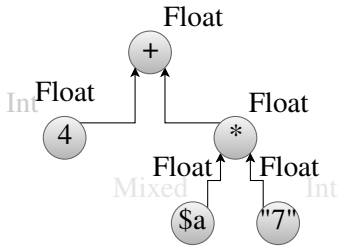
Operátor: { = }
Kopíruje datový typ z pravého operandu do levého.

Volání funkce

Součástí nástroje je seznam některých vestavěných funkcí PHP s datovými typy jejich návratových hodnot. Pokud je vracen pouze jeden datový typ, tak je nastaven jako výstupní. V opačném případě je použit Mixed.

7.3.2 Propagace výsledku druhéh kroku

Při získání typů operandů je jejich výsledek propagován listům, pokud je typem Int, Float, nebo String. Tím je docíleno konverze generovaného kódu na potřebné datové typy. Výsledek propagace můžeme vidět na diagramu 7.4, kde černou barvou jsou zapsány výstupní typy a šedou barvou vstupní. Detekovaný typ Float u proměnné $a je uložen k informacím o proměnné spolu s cestou v abstraktním syntaktickém stromu. Tyto hodnoty jsou následně zohledněny při generování kódu.



Obrázek 7.4: Detekce datových typů – krok 3

7.4 Konverze datových typů

Jelikož může být použit například výraz `5 . (4*6)`, kdy se prvně provede konkatenace dvou řetězců, a následně se provádí matematické operace, tak bylo třeba implementovat funkce pro konverzi mezi datovými typy. Při návrhu konverze na řetězec se zdálo jako vhodné řešení `std::to_string`. Následně se však ukázalo, že absence podpory konverze instance třídy `Php::Value` není elegantně řešitelný problém. Specifikace jazyka C++ v kapitole 17.6.4.2.1 zakazuje přidávat funkce do jmenného prostoru `std` [32]. Vznikla tedy nutnost vytvořit tuto funkci ve vlastném jmenném prostoru, kdy pro většinu datových typů bude pouze volat originální funkci. Při testování se ukázalo, že je také třeba předefinovat převod typu `double` na řetězec, jelikož PHP interpretru odpovídá `printf` formátovací řetězec `%G`, místo původního `%f`[18]. Výsledkem je soubor `cpp/PhpValString.cpp` implementující funkci `to_string` v jmenném prostoru `php2cpp`.

Druhou konverzí je pak převod na desetinné číslo – například pro výraz `5 * ("4.".6)`. Implementace opět proběhla v jmenném prostoru `php2cpp` a pro zachování konvence byla pojmenována `to_float`. Opět se jedná o přetíženou funkci, aby bylo možné převádět různé datové typy. Většina konverze je řešena C++ operátorem přetypování `(double)` a u řetězců pak funkcí `std::stod`. Tím je dosaženo rozumné rychlosti.

7.5 Generování kódu

Generování kódu z abstraktního syntaktického stromu doplněného o výsledky analýzy datových typů a informací o proměnných je prováděno prostřednictvím několika tříd. Každá třída implementuje část generování.

`FileGenerator` je použit pro generování výstupního C++ souboru. Na začátek souboru doplňuje potřebné konstrukce pro vložení hlavičkových souborů a dalších C++ funkcí a tříd vytvořených pro potřeby překladu. Dále volá generování kódu funkcí a následně ve funkci `get_module` exponuje jednotlivé přeložené funkce interpretovanému prostředí.

`FunctionGenerator` generuje kód funkce. Obsahuje informace o názvu funkce, argumentech a lokálních proměnných. V struktuře abstraktního syntaktického stromu generovaného kódu je jednou z větví `FileGenerator` a obsahuje jednu instanci třídy `CodeGenerator`. Úkolem této třídy je vygenerování C++ kódu funkce s argumenty, vygenerování kódu pro definici a inicializaci lokálních proměnných s hodnotami argumentů. Pro generování proměnných se správnými datovými typy použit `VariableGenerator`.

`CodeGenerator` je reprezentací jednoho bloku kódu. Může reprezentovat funkci, podmíněnou část kódu či tělo cyklu. Obsahuje posloupnost konstrukcí s operandy, které mohou být `ExprGenerator` pro generování výrazu, nebo `CodeGenerator` pro další blok kódu. Ukázka použité struktury je uvedena pseudokódem 7.1. Položka command rozlišuje, o jakou konstrukci se jedná a jak budou zpracovány argumenty. V tomto případě se jedná o reprezentaci cyklu for. Instance třídy `CodeGenerator` provede vygenerování začátku zápisu `for(` a zavolá metodu na prvním argumentu typu `ExprGenerator` pro získání kódu inicializátoru. Následně doplní středník a předá generování kódu další instanci generátoru výrazu pro podmínku a poté pro inkrement. Dalším krokem je zavolání metody pro generování kódu těla cyklu posledním argumentem – instancí třídy `CodeGenerator`, která může obsahovat obdobné zanoření.

```
1 [
2   'command' => 'for',
3   'args' => [
4     ExprGenerator,
5     ExprGenerator,
```

```
6       ExprGenerator,
7       CodeGenerator
8     ]
9   ]
```

Kód 7.1: Struktura interní reprezentace konstrukce zdrojového jazyka PHP

`ExprGenerator` generuje výraz na základě výsledku z precedenční analýzy a sady pravidel zmíněných v kapitole 6. Rekurzivně projde strom výrazu a vrátí kód.

7.6 Testování

Při implementaci jsem se rozhodl pro programování řízené testy (Test-driven development). Tedy tvorba testů pro překlad, které napřed nejsou úspěšné, ale postupně se s prací na projektu stav zlepšuje. Výhodou takového postupu je kontrola prováděných změn v komplexním řešení, kterým překladač bezesporu je. Automatizované testování takto integrovaného řešení bylo poměrně komplikované a proto jsem se nakonec rozhodl pro manuální spouštění testů a vyhodnocování.

Testy jsou umístěny u zdrojových kódů v adresáři „tests“ a rozčleněny podle specifické části, kterou testují. Testy pokrývají základní konstrukce – výstup, komentáře, konstanty, funkce, vkládání, výrazy, argumenty. Dále jsou testy zaměřené na pole, cykl for, foreach, while, funkce, podmínky, precedenční analýzu nebo práci s řetězci. Mezi další testy testy v jazyce C++ pro ověření funkčnosti části implementované v C++ a testy algoritmů – soubory s algoritmem implementovaným v PHP pro testování výkonu a komplexnějšího bloku kódu.

7.7 Experimentování

Po dokončení zamýšleného rozsahu jsem se zaměřil na testování překladu na připravených algoritmech i částech používaných knihoven, nebo částech samotného řešení. Najít však knihovnu, či aplikací, jejíž část by bylo možné přeložit, se ukázalo jako velmi obtížné. Příčinou byl objektový návrh většiny knihoven, či časté používání referencí, anonymních funkcí a globálních proměnných, které nejsou v současné implementaci překladače podporovány.

Veškeré experimenty byly prováděny na počítači vybaveným procesorem Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz, 16 GB RAM 1 600 MHz, SSD s zápisem 480 MB/s a čtením 540 MB/s. Operačním systémem je Fedora 21 s jádrem verze 4.1.13-100.fc21.x86_64. Verze PHP interpretru je 5.6.15 s Zend Engine v2.6.0 a Xdebug v2.3.3. Jako kompilátor přeložených C++ zdrojových kódů bylo použito GCC 4.9.2 20150212 (Red Hat 4.9.2-6). Pro kompiler byly použity přepínače `-Wall -c -O2 -std=c++11 -fpic -o`.

Bohužel se ukázalo, že podmnožina jazyka PHP nebyla vybrána zcela optimálně a absence konstrukce `isset` a `unset` znemožnila překlad většiny knihoven. Pro experimenty a testování jsem proto použil některé algoritmy a vzorce, které zastupují knihovní funkce.

Faktoriál

Pro výpočet faktoriálu byla použita implementace s cyklem typu for provádějící násobení s uložením mezivýsledku do proměnné. Pro výpočet byl zvolen faktoriál čísla 5 000 000. Kód pro měření byl použit 7.2.

Čas prvních 3 testů zkompilované verze je: 50.831 s, 51.555 s, 50.380 s. Interpretovaná verze: 52.301 s, 49.888 s, 50.123 s. Po úpravě datových typů (změna `Php::Value` za `double`) byly výsledky testu 2.149 s, 2.144 s, 2.139 s. Výsledek tohoto testu ukázal zpomalení 0.3% zkompilované

verze. Zajímavější je ovšem výsledek verze upravené o správné datové typy, který potřeboval pro svůj běh pouze 4.2% času interpretované verze. Tato verze ušetří 95.7% původně potřebného času a bylo třeba provést jednoduchou změnu na 2 řádcích vygenerovaného kódu.

```
1 $start = microtime(TRUE);
2 for($i = 0; $i < 100; $i++)
3 {
4   factorial(5000000);
5 }
6 echo microtime(TRUE)-$start;
```

Kód 7.2: Test pro měření rychlosti výpočtu faktoriálu

Fibonacciho číslo

Algoritmus pro výpočet n-tého čísla Fibonacciho řady byl zvolen záměrně neefektivní formou rekurzivního algoritmu vycházejícího z definice 7.1

$$F(x) = \begin{cases} 0, & \text{pro } n = 0; \\ 1, & \text{pro } n = 1; \\ F(n-1) + F(n-2) & \text{jinak.} \end{cases} \tag{7.1}$$

Pro test byl zvolen výpočet 27. prvku řady. Provedeno 100 výpočtů a změřen čas kódem 7.3. Čas zkompilované verze: 84.709 s, 84.177 s, 84.243 s. Intepretovaná verze: 54.155 s, 53.285 s, 53.332 s. V tomto případě zafungovala detekce datových typů správně a nebyla tedy vytvořena verze obsahující ručně zoptimalizované datové typy. V případě takto časté rekurze by ovšem mohlo být vhodné zoptimalizovat volání funkce – nevolat funkci prostřednictvím konstrukce `Php::Call`, ale volat přímo její C++ implementaci. Proto vznikla optimalizovaná verze tímto postupem a byla změřena. Časy této implementace jsou: 84.702 s, 83.288 s, 83.765 s.

Přeložená verze dosáhla zpomalení o 57.4% oproti interpretované verzi. Ukázalo se, že přeložené funkce s velkým množstvím rekurze jsou pomalejší, než interpretované. Do optimalizace, kdy bude z volání funkce vynechána knihovna PHP-CPP, resp. Zend engine, jsem vkládal naděje na zrychlení. Ukázalo se, že tato implementace je stále výrazně pomalejší, než interpretovaná i když bylo dosaženo zrychlení o 0.6% oproti pouze přeložené verzi.

```
1 $start = microtime(TRUE);
2 for($i = 0; $i < 100; $i++)
3 {
4   fibonacci(27);
5 }
6 echo microtime(TRUE)-$start;
```

Kód 7.3: Test pro měření rychlosti výpočtu fibonacciho čísla

Arcus sinus

Algoritmus pro výpočet byl použit taylorův polynom s vzorcem 7.2, který zejména pro hodnoty limitně se blížící 1 má pomalou aproximaci výsledku. Pro přesnost byla použita konstanta $1^{-}15$.

$$arcsin(x) = x + \frac{1}{2} * \frac{x^3}{3} + \frac{1*3}{2*4} * \frac{x^5}{5} + \frac{1*3*5}{2*4*6} * \frac{x^7}{7} + \ldots = \sum_{n=0}^{\inf} \left(\frac{(2n)!}{2^{2n}(n!)^2} * \frac{x^{2n+1}}{2n+1} \right), |z| < 1 \tag{7.2}$$

Pro měření byl použit kód 7.4, který provedl 100 výpočtů hodnoty arcus sinus pro číslo 0,99999. Potřebný čas pro těchto 100 výpočtů změří s dostatečnou přesností. Výsledky pro 3 měření zkompilované verze jsou: 133.469 s, 128.166 s, 133.269 s a pro interpretovanou verzi 126.870 s, 126.772

s, 126.957 s. Po úpravě vygenerovaného zdrojového kódu, která spočívala v změně datových typů (změna `Php::Value` za `double`) byly výsledky testu 113.505 s, 114.399 s, 114.408 s.

V případě zlepšení detekce datových typů je tedy možnost u tohoto algoritmu dosáhnout zkrácení výpočetního času na 89.94 % původního času. Současná implementace překladače se ukázala jako nejpomalejší a naopak přidává zpomalení ve výši 3.75 %.

```
1 $start = microtime(TRUE);
2 for($i = 0; $i < 100; $i++)
3 {
4   my_asin(0.99999);
5 }
6 echo microtime(TRUE)-$start;
```

Kód 7.4: Test pro měření rychlosti výpočtu Arcus sinus

7.8 Návrhy na zlepšení

Ukázalo se, že předpokládané úzké hrdlo ve formě třídy `Php::Value` využívající interně `zval` struktury a funkce interpretru je reálným problémem. Proto jsem započal implementaci vlastní třídy s přetíženými operátory využívající cachování výsledků při konverzi typů a detekci celých a desetinných čísel v řetězcích. Částečně implementovaná třída je již dostupná v zdrojových kódech projektu v souboru `cpp/PhpValue.cpp`. Bohužel kód není natolik kompletní, aby bylo možné změřit dopad na rychlost.

Dalším problémem byla nepřesná detekce datových typů. V případě zpřesnění výsledků by došlo k generování optimálnějšího kódu a zlepšení výsledků, což ukázal experiment. Pro zlepšení detekce datových typů se nabízí několik několik řešení. Prvním je rozšíření pravidel o další pravidla, pro zpřesnění detekce. Tento postup bude vyžadovat další podrobné zkoumání zdrojových kódů PHP, pro zjištění přesného chování operátorů i v okrajových situacích. Další možností je zavedení fuzzy logiky do detekce datových typů a následné generování několika variant kódu – pro pravděpodobnější kombinaci datových typů by byl vygenerován optimalizovaný kód s datovými typy a pro méně pravděpodobný pak obecý kód s použitím `Php::Value`. Obdobné řešení jsem z počátku zavrhl pro vysoký počet možných kombinací, ale v případě generování jen pro více pravděpodobné kombinace by tato optimalizace mohla mít smysl.

Zajímavým způsobem, jak detekovat datové typy v kódu by mohlo být vytvoření vlastního analytického rozšíření PHP, které by sledovalo vykonávání interpretované aplikace. Tento nástroj by se zaměřil na hodnoty uložené v zval strukturách interpretru PHP při běžném provozu PHP aplikace. Po nasbírání dostatečně relevantního vzorku by byla provedena analýza typů a výsledek využit pro generování kódu. S přihldénutím k pravděpodobnosti, že ojde k zavolání některé funkce s argumentem jiného datového typu, než jaký byl v analyzovaných datech by byla opět generována i verze využívající `Php::Value`.

Navržené, a v sekci experimentování otestované, řešení zamýšlející vynechat PHP-CPP z procesu volání funkce, v případě volání přeložené funkce z přeloženého kódu, se ukázalo jako neopodstatněné. Provedené měření ukázalo zrychlení o pouhých 0.6% původního času.

Kapitola 8

Závěr

Výsledkem této práce je překladač, který je schopen přeložit podmnožinu jazyka PHP do C++. Některé přeložené konstrukce jsou rychlejší, ale bohužel práce s poli a volání funkcí je stále pomalejší. Při práci na projektu byla vytvořena sada pravidel pro detekci datových typů na základě výrazů a navrženo několik dalších způsobů detekce datových typů původně slabě typovaných proměnných. Dalším výsledkem práce je navržení a odzkoušení pravidel pro transformaci kódu řešící rozdíly obou jazyků.

Praktické použití projektu v současném rozsahu je komplikované, jelikož výsledný kód není vždy rychlejší a není podporována dostatečně velká množina jazyka, která je zastoupena v reálných aplikacích. Ukázalo se, že i když se při výběru podmnožiny zdál výběr dostatečně široký, tak pro reálné aplikace je značně nedostačující.

Současný stav ovšem plánuji změnit, jelikož nástroj vzniklý v rámci této práce hodlám zveřejnit zdarma jako open source projekt pod licencí Apache 2.0 s placenou technickou podporou. V době dokončování této práce je již na adrese `<www.php2cpp.net>` rozpracovaný web pro prezentaci řešení. Zájem o tento nástroj projevilo hned několik zástupců firem na studentské konferenci Excel@FIT, na které jsem demonstroval své řešení. Výsledek má tedy, v případě pokračujícího vývoje, komerční potenciál. V blízké době mám v úmyslu zlepšit detekci datových typů za použití postupů navržených v této práci a dokončit novou implementaci třídy pro dynamické datové typy. Zamýšlím také prezentovat toto řešení na dalších konferencích zaměřených na PHP vývojáře. Jednou z takových akcí, na níž je účast již v jednání, je PoSobota, na které se neformálně scházejí vývojáři, nejen používající Nette Framework, a přednášejí o zajímavých technických řešeních.

Literatura

[1] Contributing to phc. <http://www.phpcompiler.org/contribute.html>, 2009-12-15 [cit. 2015-10-25].

[2] Issues. <https://github.com/pbiggar/phc/issues>, [cit. 2015-10-25].

[3] pow. <http://www.cplusplus.com/reference/cmath/pow/>, [cit. 2015-11-08].

[4] std::string::append. <http://www.cplusplus.com/reference/string/string/append/>, [cit. 2015-11-08].

[5] Achour, M.; Betz, F.; Dovgal, A.; aj.: Declaration of the Zend Function Block. <http://www.itmnetworks.com.br/suporte/manuais/php/zend.structure.function-block.html>, 2005-03-01 [cit. 2015-11-09].

[6] Achour, M.; Betz, F.; Dovgal, A.; aj.: Comparing Objects. <http://php.net/manual/en/language.oop5.object-comparison.php>, [cit. 2015-11-08].

[7] Achour, M.; Betz, F.; Dovgal, A.; aj.: Comparison Operators. <http://php.net/manual/en/language.operators.comparison.php>, [cit. 2015-11-08].

[8] Achour, M.; Betz, F.; Dovgal, A.; aj.: Operator Precedence. <http://php.net/manual/en/language.operators.precedence.php>, [cit. 2015-11-08].

[9] Achour, M.; Betz, F.; Dovgal, A.; aj.: PHP type comparison tables. <http://php.net/manual/en/types.comparisons.php>, [cit. 2015-11-08].

[10] Achour, M.; Betz, F.; Dovgal, A.; aj.: The zend_module structure. <http://php.net/manual/en/internals2.structure.modstruct.php>, [cit. 2015-11-08].

[11] Achour, M.; Betz, F.; Dovgal, A.; aj.: Life cycle of an extension. <http://php.net/manual/en/internals2.structure.lifecycle.php>, [cit. 2015-11-09].

[12] Achour, M.; Betz, F.; Dovgal, A.; aj.: PHP Logic operators. <http://us2.php.net/manual/en/language.operators.logical.php>, [cit. 2015-12-25].

[13] Achour, M.; Betz, F.; Dovgal, A.; aj.: PHP: extract. <http://php.net/manual/en/function.extract.php>, [cit. 2016-05-07].

[14] Achour, M.; Betz, F.; Dovgal, A.; aj.: PHP: require. <http://php.net/manual/en/function.require.php>, [cit. 2016-05-09].

[15] Achour, M.; Betz, F.; Dovgal, A.; aj.: PHP: Basics. <http://php.net/manual/en/language.variables.basics.php>, [cit. 2016-05-10].

[16] Achour, M.; Betz, F.; Dovgal, A.; aj.: PHP: Introduction to Variables. <http://php.net/manual/en/internals2.variables.intro.php>, [cit. 2016-05-11].

[17] Achour, M.; Betz, F.; Dovgal, A.; aj.: PHP: Reference Counting Basics. <http://php.net/manual/en/features.gc.refcounting-basics.php>, [cit. 2016-05-11].

[18] Achour, M.; Betz, F.; Dovgal, A.; aj.: PHP: Floating point numbers. <http://php.net/manual/en/language.types.float.php>, [cit. 2016-05-12].

[19] Caulfield, G.: Suppress error with @ operator in PHP. <http://stackoverflow.com/a/960288>, 2009-06-11 [cit. 2015-11-08].

[20] Gilmore, W. J.: *Velká kniha PHP a MySQL 5*. Zoner press, 2006, iSBN 80-86815-53-6.

[21] Golemon, S.: Remove support for buildling HPHPc. <https://github.com/facebook/hhvm/commit/fc5b95110ff75110ad55bb97f7c93a8c4eb68e3b>, 2013-02-19 [cit. 2015-10-25].

[22] Graniszewski, A.: HipHop for PHP: Benchmark. <http://php.webtutor.pl/en/2011/04/02/hiphop-for-php-bechmark-english-version/>, 2011-04-02 [cit. 2015-10-25].

[23] Graniszewski, A.: Drupal 7: HipHop for PHP vs APC - benchmark. <http://php.webtutor.pl/en/2011/05/17/drupal-hiphop-for-php-vs-apc-benchmark/>, [cit. 2016-05-09].

[24] Grudl, D.: Jak psát kód: Nepřežeňte to s komentáři. <http://php.vrana.cz/jak-psat-kod-neprezente-to-s-komentari.php#d-32276>, 2013-06-14 [cit. 2015-11-08].

[25] Hansen, R.: Magic Hashes. <http://blog.whitehatsec.com/magic-hashes/>, 2015-05-11 [cit. 2015-11-08].

[26] Julien Pauli, N. P., Anthony Ferrara: Zvals: Casts and operations. <http://www.phpinternalsbook.com/zvals/casts_and_operations.html>, [cit. 2016-05-09].

[27] Kuznik, P.: Working with variables. <http://www.php-cpp.com/documentation/variables>, [cit. 2016-05-09].

[28] Lengstorf, J.: Strict vs. Loose Comparisons in PHP. <http://www.copterlabs.com/blog/strict-vs-loose-comparisons-in-php/>, [cit. 2015-11-08].

[29] Paroski, D.: Speeding up PHP-based development with HHVM. <https://www.facebook.com/notes/facebook-engineering/speeding-up-php-based-development-with-hiphop-vm/10151170460698920>, 2012-11-29 [cit. 2015-10-25].

[30] Rethams, D.: Five reasons why the shut-op operator (@) should be avoided. <http://derickrethans.nl/five-reasons-why-the-shutop-operator-should-be-avoided.html>, 2009-06-03 [cit. 2015-11-08].

[31] Sklar, D.: *PHP 5 – moduly, rozšíření, akcelerátory*. Zoner press, 2005, iSBN 80-86815-19-6.

[32] Stroustrup, B.: *The C++ Programming Language – Fourth Edition*. Addison-Wesley, 2013, iSBN 0-321-56384-0.

[33] Vrána, J.: *1001 tipů a triků pro PHP*. Computer press, 2010, iSBN 978-80-251-2940-1.

[34] Woodward, J.: Why you shouldn't suppress errors in PHP using the @ operator. <http://josephwoodward.co.uk/2014/02/why-you-shouldnt-suppress-errors-in-php-using-operators/>, 2014-02-12 [cit. 2015-11-08].

[35] Zhao, H.: HipHop for PHP: Move Fast. <https://www.facebook.com/notes/facebook-engineering/hiphop-for-php-move-fast/280583813919>, 2010-02-02 [cit. 2015-10-25].