FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Transpilátor z jazyka PHP do jazyka Go
Student:	Lukáš Simulík
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2021/22

Pokyny pro vypracování

Nastudujte jazyky PHP a Go a principy týkající se konstrukce překladačů.
Navrhněte, analyzujte a implementujte transpilátor projektů z jazyka PHP do jazyka Go. Transpilátor přeloží veškeré základní řídící konstrukce, standardní operátory, oblasti platnosti proměnných a také další nezbytné konstrukce jazyka PHP a také detekuje těžko přeložitelné konstrukce a upozorní na potřebné ruční úpravy výsledného kódu.
Implementovaný transpilátor testujte na jednoduchých projektech.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 28. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Transpilátor z jazyka PHP do jazyka Go

Lukáš Simulík

Katedra teoretické informatiky
Vedoucí práce: Ing. Radomír Polách

4. června 2020

Poděkování

Děkuji panu inženýrovi Poláchovi za návrhy, jakým způsobem převádět konstrukce jazyka PHP do Go, a tipy, jakým způsobem tento projekt dál rozšířit. Rovněž děkuji panu inženýrovi Bohuslávkovi za postřehy při realizaci transpilátoru a objevením projektů, zabývající se překladem do jazyka Go.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. června 2020 . . . . . .

Odkaz na tuto práci

Simulík, Lukáš. *Transpilátor z jazyka PHP do jazyka Go.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Práce se zabývá tvorbou programu, který převádí projekty pro skriptovací jazyk PHP do jazyka Go. Popisuje historii PHP a jeho vývoj, rozebírá způsoby, kterými lze kód převádět a zrychlovat. Popisuje již existující projekty zabývající se touto problematikou. Zahrnuje popis konstrukce kompilátoru a jeho jednotlivé části. Popisuje realizaci nástroje, způsoby testování, a srovnává výkon původního řešení s novým, přeloženým. Analyzují se způsoby, jakými šlo jednotlivé konstrukce zdrojového jazyka převést do cílového.

Klíčová slova transpilátor, jazyk Go, jazyk PHP, kompilovaný jazyk, dynamický jazyk, webová technologie

Abstract

The thesis deals with the creation of a program which converts projects for the PHP scripting language into the Go language. Describes the history of PHP and its development, discusses ways how to convert and speed up code. Describes existing projects dealing with this issue. Includes a description of the compiler design and its individual parts. Describes implementation of the tool, methods of testing, and compares the performance of the original

solution with the translated one. Methods how the individual constructions could be translated were analyzed.

Keywords transpiler, Go language, PHP language, compiled language, dynamic language, web technology

Obsah

Seznam obrázků

Úvod

Transpilace je procesem, ve kterém se převádí zdrojový kód z jednoho do druhého. Jde o událost ve světě internetových technologií velmi rozšířenou. Jde rozdělit do dvou kategorií, podle místa použití.

První z nich souvisí se zobrazováním na straně klienta, s internetovým prohlížečem. Místo toho, aby programátor používal jazyk prohlížeče, pracuje s jiným, který má větší možnosti, lepší strukturu, kratším zápisem, nebo obsahuje konstrukce známé z jiných jazyků. Často si jsou oba jazyky velmi podobné, nový nějakým způsobem vychází z původního. Toto rozhodnutí používat jinou technologii s sebou nese problém v podobě nutnosti kód překládat do očekávané podoby. Nemusí jít pouze o rozšíření možností jazyka pomocí nového dialektu, jde i o zpětnou kompatibilitu. Jazyky se vyvíjejí, ale ne všechni používají nejnovější prohlížeče.

Druhá kategorie souvisí se stranou serveru a tou se bude zabývat i tato práce. Jedny z nejrozšířenějších jazyků v této oblasti mají dvě významné vlastnosti. Je velmi jednoduché v nich program napsat a změnu jde okamžitě vidět, protože jsou interpretované. To s sebou nese dvě nevýhody. Kvůli interpretu nejsou tolik výkonné, zdroj se totiž musí pokaždé překládat. Nejde o jedinou věc zpomalující běh, jazyky jsou také často velmi dynamické, datové typy proměných se mohou měnit každou chvíli.

Tato vlastnost zjednodušuje psaní, ale snižuje rychlost běhu a způsobuje, že kontrola nad programem je nižší. Kvůli tomu se zvyšuje šance, že se program nebude chovat tak, jak má. Projektů na zrychlení kódu vzniklo velké množství. Nepřímo se zabývají všemi problémy interpretovaných jazyků.

Tato práce se bude zabývat skriptovacím jazykem PHP. Patří mezi ty nejrozšířenější a má všechny neduhy zmíněné výše. Práce se bude krátce zabývat vznikem tohoto jazyka, částečně proto, aby byly obhájeny některá jeho implementační rozhodnutí. Jazyk je kritizován pro některé jeho vlastnosti, většina z nich je ale dána historicky, jeho původním záměrem.

Dalším bodem bude popsání způsobů, jak ke zrychlení jazyka může dojít.

Nejdůležitějším ze všech bude transpilace, každý projekt ji v nějaké míře obsahuje. Prací řešící tuto problematiku je hned několik, některé z nich budou zmíněny zde. U převodu konstrukcí poslouží jako inspirace pro implementaci vlastního řešení. Posledním krokem před realizací je popis jednotlivých částí překladače, což je koncept, kterým se celá práce zabývá.

Cílem této práce bude převést vybranou množinu konstrukcí nacházející se v jazyce PHP a převést je do jazyka Go tak, aby se nezměnil výsledek, a pokud možno vylepšily jeho vlastnosti – rychlost běhu programu a čas odezvy, pokud bude spuštěn jako server. Ke zrychlení bude docházet jen díky změně technologie, optimalizace se nebudou provádět, struktura programu by měla být co nejvíce podobná zdrojovému skriptu. Převádět se bude jen kód splňující dané požadavky, mezi ně například patří jednoznačnost datového typu v daném místě programu. Testováním se zkontroluje úspěšnost převádění. Práce bude mít celou řadou příkladů, u kterých budou popsány důvody, proč jdou převést, či nikoliv. Úspěšnost se nebude dokazovat pouze na příkladech, dojde k pokusům o přeložení již fungujících projektů.

Kapitola 1

PHP

Kapitola se bude zabývat historií webových technologií, chronologicky popíše jejich vývoj po příchod PHP. Zmiňuje se o původním záměru tohoto jazyka.

1.1 Historie WWW

V roce 1989 se v laboratořích CERN objevuje návrh projektu, který ve svém názvu nese dvě významné fráze. Jsou jimi WorldWideWeb a HyperText. Cílem bylo obejít omezení spojená s hielarchickým uspořádáním souborů. Nebylo totiž možné se jednoduchým způsobem dostat na jiný soubor, aniž by se muselo celou stromovou strukturou znovu procházet. Podle Tima Berners-Leeho má „*Stromová struktura praktické výhody v pojmenování každého uzlu unikátním jménem. Na druhou stranu ale neumožňuje systému popsat reálný svět.*“ [1, překlad autora] Odpovědí na tento problém měly být odkazy. Díky nim půjde lépe popsat vztah mezi jednotlivými dokumenty, mohou mezi nimi vzniknout cyklické vazby, dát jim stejnou váhu nehledě na umístění ve stromu a tak podobně.

Žádost se odložila a znovu se začne řešit v květnu přístího roku. Pro implementaci byly použity dvě nové technologie, HyperText Markup Language a HyperTextTransfer Protocol. Obě tyto technologie stále existují a jsou dále vyvíjeny konsorciem W3C, které pro ně zároveň vydává i standardy [2].

Do listopadu 1993 nebyla tato technologie příliš populární, což bylo zapříčeněno nižší atraktivitou prohlížečů HTML stránek. Změna přišla s grafickým webovým prohlížečem Mosaic. Jde o projekt NCSA patřící pod univerzitu Illinois. Úspěchu vděčí za implementaci věcí, které do té doby žádný jiný prohlížeč neměl, „*jde například o ikony, záložky, poutavé uživatelské prostředí – to udělalo software jednoduchým pro používaní a lákavé i pro osoby, které nejsou technicky zaměřené*“ [3, překlad autora].

Požadavky na schopnosti stránek se neustále zvětšovaly, dalším krokem bylo zařídit, aby byly interaktivní. Pro to ze začátku sloužilo Common Gateway Interface – CGI. Jedná se o prostředníka mezi serverem a programem

generující stránku. Díky tomu se rozšířily možnosti, jak odpověď bude vypadat, již nemuselo jít o kopii HTML stránky uloženou na serveru [4].

1.2 Historie PHP

CGI bylo jedním z důvodů, proč se objevil projekt, pojmenovaný jako „Personal Home Pages“. Byl napsán Rasmusem Lerdorfem a jeho cílem bylo vytvořit velmi jednoduchý šablonovací systém, který bude vypadat jako HTML, jen bude mít navíc párovou značku, která se nahradí dynamickým obsahem [5]. Do té doby programy na druhé straně CGI byly napsané v C nebo v Perl, a měly podobu standardního programu, bylo proto těžké rozpoznat, že výsledkem bude HTML výstup [6].

Ke vzniku tohoto systému došlo v roce 1994 a nyní je znám pod zkratkou „PHP“. Veškeré výpočty se měly provádět v jiném jazyce, autorovou vizí bylo používat jazyk C, vytvořil proto rozhraní pro komunikaci mezi těmito dvěma jazyky. V porovnání s původním záměrem a vzhledem jazyka dnes je vidět, že se zaměření změnilo, možnosti jazyka se zvětšily. PHP již není známé tím, že by bylo šablonovacím jazykem, vykreslování a výpočty jsou spojené dohromady [5].

Tato změna postavení jazyka způsobila, že se v něm objevují konstrukty, které jsou pro jiné jazyky se stejným zaměřením neobvyklé. Většina z nich bude zmíněna v kapitole 5, protože s nimi cílový jazyk neumí pracovat.

Kapitola 2

Způsoby zrychlování kódu

Tato kapitola popíše způsoby, jakými lze již existující programy zrychlit. Otázka rychlosti je spojena s PHP od jeho vzniku. Skripty v jazyce Perl byly několikanásobně rychlejší než ty v PHP, ale díky správným a rychlým rozhodnutím tento jazyk nakonec nad konkurentem zvítězil [5].

Způsoby se od sebe liší velikostí zásahu do původní struktury a velikostí zrychlení.

2.1 Transpilace

Kompilátor je program, který převede programovací jazyk do strojového kódu nebo do formátu CLI. Jde o standardizovaný formát nezávislý na zdrojovém kódu. Podporuje objektově orientované programování, generické datové typy. Cílem je mít kód nezávislý na platformě, ke spuštění programu stačí mít systém, který tento formát umí zpracovat a spustit [7]. Tímto způsobem například funguje .NET Framework od společnosti Microsoft [8].

Transpilátor je speciální případ kompilátoru, který převádí zdrojový kód mezi dvěma vysokoúrovňovými programovacími jazyky. Tento proces se používá například pro převod ze staršího jazyka do nového. Dalším důvodem je modernizace již existujícího kódu, používání nových funkcí a tak podobně. Dalším důvodem může být optimalizace zdrojového kódu. Jde o sérii transformací, rozdělené do několika kroků podle jejich náročnosti. V prvních fázích se odstraní nepoužité proměnné, nevykonatelné bloky kódu a tak podobně. Pak přijde na řadu odstraňování zbytečných přiřazování i když hodnota je jasně daná konstantou, úpravy cyklů a řídících konstrukcí, či aplikování znalostí o argumentech funkce [9].

2.2 Jiný interpret

Dalším ze způsobů jak vylepšit čas je použitím jiného interpretu pro daný jazyk. Tento způsob bude ze všech nejméně používaný. Dáno je to zejména tím, že problém rychlosti to neumí řešit tak dobře jako jiné zmiňované varianty. Je zde stále problém se zpomalením, který si s sebou nese každý interpret. I přesto interprety vznikají, jejich využití ale není určeno pro finální produkt. Slouží k jednoduššímu testování, je možné sledovat výsledný kód v reálném čase v případě transpilace a díky tomu snadněji objevovat chyby [10], [11].

2.3 JIT kompilátor

Aktuálně nejúspěšnějším způsobem je použitím JIT kompilátoru. Jde o zkratku pro „just-in-time“, termín do češtiny nepřekládaný. Pro program to znamená odkládání kompilace, k prvnímu překladu funkce do strojového kódu dojde až když je poprvé volána. Při dalším volání této funkce se již bude používat varianta přeložená, čas strávený voláním funkce tak bude mnohonásobně menší [12].

Pokud se funkce volá často, bude docházet k dalším úpravám kódu. Optimalizace se odkládá, protože kdyby se každá funkce volala jen jednou, místo ke zrychlení by došlo ke zpomalení. Cílem je zachovat vlastnosti podobné běžným kompilátorům, uživatel nesmí na čase poznat, že se provádí optimalizace.

Dojde k optimalizacím podobným těm, co lze provádět během transpilace zmíněné v sekci 2.1. Protože cílem je strojový kód, mohou se ve funkcích provádět další úpravy, narušující původní význam. Mezi ně patří odstranění zapouzdřování, běžné pro objektově orientované programování, nebo snížení přístupů do tabulky virtuální metod. Oproti statickému kompilátoru má tento velkou výhodu v podobě většího množství informací o prostředí ve kterém se program vykonává. Díky tomu se může lépe přizpůsobit a optimalizovat kvalitněji daný druh problému. Optimalizace se provádí v několika fázích. Jsou k tomu dva důvody. Prvním z nich je snaha příliš nezpomalit první kompilaci. Druhým je ten fakt, že optimalizace nemusí být vždy úspěšná a může dojít ke zhoršení vlastností. Kompilátor má proto schopnost provést i inverzní postup k optimalizaci [13].

Kapitola 3

Projekty zabývající se úpravou PHP

Úpravou PHP se zabývalo velké množství projektů. Většina z nich nějakým způsobem řešil problém dynamického typování. Sekce budou obsahovat použitou technologii, popis, jaké problémy se při převodu řešily, a jak.

V kapitole jsou dvě sekce, které se problémem dynamického typování přímo nezabývají. Jde o novější verzi interpretu PHP a jeden nový jazyk, který má skriptovací jazyk za inspiraci.

Protože všechny transpilátory řeší stejné problémy, způsoby, jakým je vyřešily, slouží jako inspirace pro vlastní řešení.

3.1 Phalanger

Tento projekt si jako cílovou technologii vybral prostředí .NET. Skripty se překládají do CLI, což je formát, který se na této platformě používá. Efektivity se dosahuje změnou technologie, ta umí provádět další optimalizace.

Text, co není ve funkci, nebo je mimo značky značící oblast PHP skriptu, se přeloží jako statická metoda. Rozhraní a třídy napsané v PHP se definují jako nové typy v CLI. Funkce a příkazy se přeloží jako jeden velký statický typ CLI. Tento typ obsahuje všechny statické metody, včetně textu mimo skript.

Mimo tuto strukturu se navíc definuje kontext ve kterém se skript právě teď nachází, obsahuje informace o stavu, ve kterém se nacházejí různé proměnné, dynamická konfigurace a další. Každá funkce má k těmto údajům přístup.

Stejně jako PHP umožňují dynamické deklarace funkcí a tříd. Každá deklarace má svoji verzi. S touto strukturou je možné provádět redeklarace, čímž by se pak lišili od původního jazyka, tak se toto chování zakázalo, může se provést jen jedna taková deklarace a ta bude aktivní po celou dobu trvání programu. Cílový jazyk dynamicky deklarovat neumí, proto se generují všechny

verze, pro přístup k požadované verzi se používá hašovací tabulka. Tabulka není pokaždé potřeba, ta má opodstatnění jen u dynamických deklarací. Proto když existuje jen jedna verze funkce, jde pro volání funkcí a vytváření objektů používat právě jedna existující definice. Tento krok se provede během kompilace [14, s. 33].

Příkaz `include` se přeloží jako funkce `eval`. To má nevýhodu v podobě časové náročnosti a slabé kontroly nad skriptem, může přinést bezpečnostní riziko. Pokud volání příkazu jde převést do statické podoby a kód za tímto příkazem jde rovněž staticky zpracovat, od funkce `eval` se ustoupí a kód se na dané místo vloží stejným způsobem, jakým to dělá PHP.

Lokální proměnné se ukládají na zásobník, toto pravidlo ale vždy neplatí. I k lokální proměnným se někdy bude přistupovat pomocí tabulky. To nastává například u vykonávání funkce `eval` a příkazu `include`. Globální proměnné se dávají do hašovací tabulky, mohou se totiž kdykoli změnit a během kompilace ještě nemusí ani existovat. Nepřímé volání v PHP reprezentované „proměnnou proměnných“ se řeší přepínačem hledajícím shodu. Pokud se zde nenajde, hledá se v haš tabulce. Nepřímé volání tak nemusí způsobovat problémy. Tento způsob hledání se musí provádět u každé metody, je to způsobené magickými funkcemi `__get` a `__set`. To nedeklarovaným položkám dává nový význam.

Analýza typů se neprovádí, typ se tu reprezentuje jen dvěma způsoby. Jde o obecný objekt a referenci, která se používá například u globálních proměnných. Jde o obalující typ, obsahuje objekt obsahující hodnotu proměnné [14, s. 34].

Pro každou metodu vznikají dvě přetížené varianty, lišící se svým použitím. Jedna slouží pro případ kdy je vše známé během kompilace, druhá, se stejným předpisem, s argumenty přímo nepracuje – volající argumenty umístí na zásobník a funkce si je zpracuje, zkontroluje [14, s. 32 – 34].

Starší verze PHP podporovaly neobvyklé deklarace metod, mohly být například abstraktní a statické zároveň, privátní a abstraktní. Tyto definice v páté verzi začaly mizet [15], překladač si s nimi ale umí poradit pomocí dodatečných atributů v CLI.

Zkompilované třídy jde použít v jiných technologiích používající stejnou platformu, stačí splnit dané rozhraní. Jde použít i převrácený postup, ještě v nezkompilovaném PHP jde použít knihovny z .NET [14, s. 35 – 36].

Toto řešení dosáhlo v testovacích případech více než dvojnásobného zrychlení oproti referenčním verzím PHP [14, s. 37].

3.1.1 PeachPie

Phalanger již není dále vyvíjen, jeho pokračovatelem je PeachPie. Má podobné cíle jako předchůdce, tedy zakomponování C# modulů do PHP. Stejně jako předtím se kód překládá do vnitřní struktury pro .NET, jde ale také celý projekt zpracovat do jazyka C# nebo Visual Basic.

Pro kompilování používají projekt Roslyn [16]. Ten si dal za cíl zjednodušit generování kódu, transformace, tvorbu doménově specifických jazyků. Nabízí také prostředky pro syntaktickou a sémantickou analýzu [17].

3.2 P9

Jde o práci, která pro svoji implementaci používá již existující framework – kompilátor od IBM. Pro ten se dají napsat generátory kódu a optimalizátor. Výsledný kód je určen pro Java Virtual Machine. Kompilátoru předávají vlastní strukturu kódu, provádějí s ní další optimalizace, ale optimalizace specifické pro Java se nepoužívají. Zrychlení se dosahuje za použití JIT kompilace [18, s. 121 – 123].

Datovým typům se dala struktura, nyní se rozlišují tři základní druhy. Prvním z nich jsou primitivní datové typy, další jsou objekty a další samostatnou skupinu tvoří další těžko identifikovatelné struktury, mezi které patří objekty zabývající se databázovým připojením, prací se souborem a další. S tímto rozdělením jsou více konkrétní než PHP a díky tomu šetří pamětí, alokace jsou více konkrétní.

Co jednotlivé základní operace vykonávají je dáno pravým a levým operandem. Toto komplikované přetěžování operátorů brání vytvoření jedné instrukce pro kompilátor, proto se všechny takové operace převedly na pomocné funkce.

Příkaz `include` v PHP vkládá kód na dané místo a umí pracovat s proměnnými, které se před tímto příkazem objevily. Zde to převedli do podoby volání funkce při zachování původního chování, k těmto proměnným má kód přístup a vše mají pod kontrolou [18, s. 124].

V překládaném jazyce je možné se k proměnné dostat několika způsoby, interpret pro všechny používá stejný způsob, a tím je asociativní pole. To s sebou nese problémy v podobě časové náročnosti, zde se toto zpomalení řeší komplikovanější strukturou. Lokální proměnné na zásobníku se indexují. Je snahou jednotlivé přístupy dále v kódu najít hned v tomto indexu. Pokud se neuspěje, používá se stejná technika jakou má interpret, tedy asociativní pole [18, s. 124].

Pro P9 jde napsat rozšiřující knihovny, bylo vytvořeno rozhraní, přes které knihovny mohou komunikovat [18, s. 125].

Zrychlení oproti interpretu PHP se liší podle použití, na malých testovacích skriptech jde až o desetinásobné zlepšení, u webových aplikací jde o hodnotu mezi dvaceti až třiceti procenty [18, s. 125 – 128].

3.3 Project Zero

Projekt vznikl pod záštitou společnosti IBM. Zabýval se spojením jazyku PHP s jazykem Java. Má vlastnosti interpretu a JIT kompilátoru: Project Zero je

pro PHP interpretem, jednotlivé elementy tohoto skriptovací jazyka vidí jako třídy v Java. Ty si pak převede do své vnitřní reprezentace a nakonec se zkompiluje do formátu pro virtuální stroj Java běžný – bytekód.

V některých oblastech se od PHP liší. Například zde neexistují superglobální proměnné – pro získání hodnot se bude používat funkce `get()`, která je součástí prostředí Zero.

Rozšíření pro PHP jsem přepsána do jazyku Java a stejně jako standadní interpret pro PHP i tento umožňuje přidat rozšíření. Mohou být napsány v C nebo v Java [19], [20].

Jakého zrychlení dosáhl v porovnání s původním řešením není známo, hlavním cílem bylo umožnit používat prostředky z obou jazyků zároveň.

3.4 Quercus

Quercus stejně jako Zero přichází s interpretem PHP a velkým množstvím rozšíření. Obě technologie jde volně kombinovat, PHP skript ale automaticky začne používat lepší řešení pocházející z Java, například efektivnější práce s databázovými připojeními.

Quercus pro některé aplikace dosáhl zrychlení, uvádí se, že až čtyřnásobné [21].

3.5 HipHop Compiler for PHP

Tento projekt od společnosti Facebook se zabývá transpilací PHP do kompilovaného, staticky typovaného jazyka, v tomto případě C++. Zrychlení je získáno omezením dynamických kontrol typů a díky optimalizacím, které umí kompilátor provádět [11, s. 576].

Převádějí většinu konstrukcí PHP, nepodporují se funkce `eval`, protože přináší bezpečnostní riziko, a neprovádí se implicitní převod mezi číselnými datovými typy, pokud má dojít k přetečení [11, s. 577 – 578].

Cílem je převést pokud možno co největší množství kódu do statické podoby, kde se nebudou muset řešit problémy dynamického typování. Dosahuje toho tím, že agresivním způsobem se snaží typy odvodit. U primitivních datových typů je tento proces velmi úspěšný. Díky tomu bude velké množství funkcí možné volat bez kontrol na datový typ [11, s. 578 – 579].

Vše, co nelze jasným způsobem převést, bude uloženo do nějaké tabulky. Pro funkci přijímající větší množství datových typů se vygenerují všechny možné kombinace a dostane unikátní jméno. Tabulka zařídí, aby se použila ta správná. Konstanty definované za běhu programu a globální proměnné rovněž nejde udělat staticky, budou uloženy v tabulce globálních symbolů. Ta obsahuje navíc statické proměnné ve třídách a informaci o jejich dostupnosti – zda byl do skriptu vložen soubor pomocí příkazu `include`. Lokální proměnné nemožné odvodit staticky rovněž skončí v tabulce. Ta má ale menší rozsah,

náleží pouze dané funkci. Přístup k dynamické proměnné znamená hledat v tabulkách lokálních a globálních symbolů. Nejednoznačné typy tu je další tabulka, ta namapuje názvy položek k hodnotám [11, s. 581].

Pokud něco nejde zavolat staticky – vytvářet objekty, vkládat skripty s nedefinovaným názvem – jsou na místě ještě vyvolávací tabulky. Chování je podobné interpretu PHP, vše se ale tvoří během kompilace, díky procesu statické analýzy.

Aby se zachovalo původní chování, simuluje se dynamické načítání souborů. Tímto způsobem funkce `class_exists` a podobné mají v cílovém jazyce stále význam. Zkompilovaný program o všech třídách již ví, ale protože ještě neproběhl `include`, předstírá, že třída neexistuje.

Kompilátor se skládá z šesti částí. První vytvoří abstraktní syntaktický strom pro zdrojové soubory. V dalším kroku se strom projde, zjistí se, co vše je definováno a kolikrát. Pro unikátní konstrukce bude možné později vytvořit lépe optimalizovaný kód. Vytvoří se graf závislostí. S touto informací bude možné paralelizovat. Třetím bodem je provedení prvních jednoduchý optimalizací. Ignorují se datové typy, odstraňují se zbytečná volání funkcí v PHP používané kvůli rozdílným verzím interpreta, mizí nepoužité funkce. Způsoby, jakými lze řetězce spojovat se převádí na jeden unifikovaný. Dalším krokem je odvození datových typů. Čím více konkrétní, tím lepší, bude se provádět během běhu programu menší množství kontrol. Pak přichází další optimalizace, které se nedaly bez typů provést. Posledním bodem je kód vygenerovat. Každá třída bude ve vlastním souboru. Každý soubor bude obsahovat funkci podobné `main`, která se zavolá během vkládání souboru [11, s. 579 – 581].

Oproti původnímu řešení je zde více než pětinásobné zrychlení, u některých úloh je zrychlení více než dvou set násobné [11, s. 582 – 584].

O tomto projektu se na svém blogu zmínil i autor PHP. Uvádí, že zrychlení neřeší všechny problémy spojené s generováním stránek. Výpočetní výkon není to, co webové stránky zdržuje, jde spíš o množství dotazů na server, zpoždění kvůli databázovému připojení a další problémy, které transpilace nevyřeší [22].

Pro projekt se používá zkratka „HPHPc“ – HipHop compiler. Aby se během testování neztratila výhoda interpretovaného jazyka, vznikl pro tento projekt i interpret, „HPHPi“. Je šestkrát pomalejší než kompilátor, v některých případech je proto horší než původní interpret. Jeho použitelnost je proto velmi nízká, navíc se chová jinak než kompilovaná verze [23, s. 778].

3.6 HipHop Virtual Machine

Jde o další z projektů společnosti Facebook zabývající se problémem rychlosti PHP. I přesto, že výsledky HPHPc byly velice dobré, stále tu byly limitace spojené s větším množstvím komplikovaných operací v případech, kdy datový typ nebylo možné odvodit. Neznámé typy způsobují, že kompilace nebude

schopná nikdy typ odvodit a průchody programem nejsou jednoznačné, což brání dalším optimalizacím [23, s. 778].

Jde o virtuální stroj chovající se jako zásobníkový počítač, který má vlastní množinu příkazů – instrukční sadu. Připomíná proto CLI. Pro tuto vnitřní strukturu se používá označení „HHBC“, HipHop ByteCode. HHVM pracuje s větší znalostí programu. Frontend virtuálního stroje převede PHP do HHBC a vše dále řeší JIT kompilátor. Zná aktuální hodnoty, proto dokáže vytvářet kód jasně definovaný, datový typ je vždy znám a díky tomu lze kód dále zrychlit [23, s. 778 – 779].

Bytekód obsahuje kromě sady příkazu ještě metadata, které pomáhají zakódovat funkce a třídy. Obsahují čísla řádku, tabulky kódující identifikátory, mapování na zdrojové soubory a další důležité informace pro to, aby mohl stroj fungovat. Většinou je kód bez typů, odkazuje se na většinu konstrukcí názvy. Reference z PHP se snaží eliminovat. Absence datových typů způsobuje, že volání funkce je těžké, musí se před ním provést dva kroky. Identifikuje se volající, zjistí se, zda funkce existuje, a jak vypadají její argumenty, co s nimi bude dělat. Druhým krokem je rozpoznat, co se bude předávat referencí, a co hodnotou [23, s. 779 – 780].

Odvozování typů probíhá za běhu a předpokládá se jejich svázanost s argumenty funkce, nebo že se nemění v průběhu vykonávání tak často. Kód se nedělí na statické a dynamické funkce, tady je dělba a následná optimalizace víc konkrétní. Dochází k rozkládání na úseky, které jsou nějakým způsobem významné svým dynamickým chováním. Pro tyto úseky se používá označení „tracelet“. Tracelet je identifikován přijímaným datovým typem. Pokud typ na vstupu je stejný jaký úsek očekává, stroj může optimalizovat, nebo nemusí nic dělat. K rekompilaci dojde pouze tehdy, když se typy liší. Tyto bloky jde řetězit, všechny za sebou v řadě budou stoprocentně specializované pro daný datový typ. Úseky jsou malé, proto při nutnosti překompilování nenástává velké zdržení. A pokud již k tomuto kroku dojde, původní verze se nezničí, takže návrat k předchozímu typu bude rychlejší. Malé úseky s sebou nesou i jednu nevýhodu v podobě menšího množství optimalizací. Je možné, že s větším množství kontextu by byl kód výkonnější než teď, složený z několika malých bloků. Pokud se budou typy měnit často, čas bude podobný jako v interpretu PHP [23, s. 780 – 782].

HHBC je stále vysokoúrovňová reprezentace zdrojového kódu. Neobsahoval informace o typu, volání funkce je náročné. Tracelet se proto dále převádí do ještě nižší podoby, nazývané jako „HHIR“. Jde o typovaný formát, který umí všechny běžné optimalizace a navíc ovládá ty, které jsou specifické pro PHP. Je přenosný, nezávislý na cílové platformě. Lépe pracuje s pamětí než PHP interpret, tvorba tříd je úspornější, a vlákna umí mezi sebou dobře spolupracovat sdílením mezipaměti [23, s. 782 – 783].

Pro tento projekt není třeba vytvářet interpret jako tomu bylo u HipHop kompilátoru. Rozdílné chování mezi testovacím a produkčním kódem tu je, jde ale jen o množství optimalizací [23, s. 778].

Zrychlení oproti HPHPc je u některých aplikací více než dvojnásobné, u komplikovaných projektů jde o padesátiprocentní zrychlení oproti běžnému PHP interpretu [23, s. 785 – 786].

3.7 Hack

Jde o jiný druh projektu než ty zmiňované výše, které se zabývaly primárně rychlostí vykonávání, případně rozšíření možností jazyka.

Hack je jazykem, který začal jako více striktní dialekt PHP. Přechod mezi nimi měl být velmi snadný a spousta zdrojových kódů už měla být validní kódem napsaným v Hack. Takovou myšlenku jde například pozorovat u TypeScript [24].

Přestal podporovat konstrukce neshodující se s myšlenkou statického typování, již není možné používat konstrukci „proměnná proměnných“. Vše může být staticky typované a množina možných datových typů je velmi velká. Typy mohou být generické, rozsah hodnot rozšířen o hodnotu `null`.

Do jazyku se přidaly kolekce s rozhraním běžné pro jiné jazyky, typům jde dát alias. Tvorba anonymních funkcí je jednodušší [25].

Zmenšilo se množství klíčových slov. Některé nemají alternativu, například slovní logické operátory se liší od standardních svojí nižší prioritou. Příkaz ukončující cyklus už není možné používat s číslicí, není proto možné jednoduchým způsobem ukončit z vnitřního cyklu i vnější. Cykly a podmínky není možné ukončit pomocí klíčového slova složeného z „end“ a názvu řídící konstrukce.

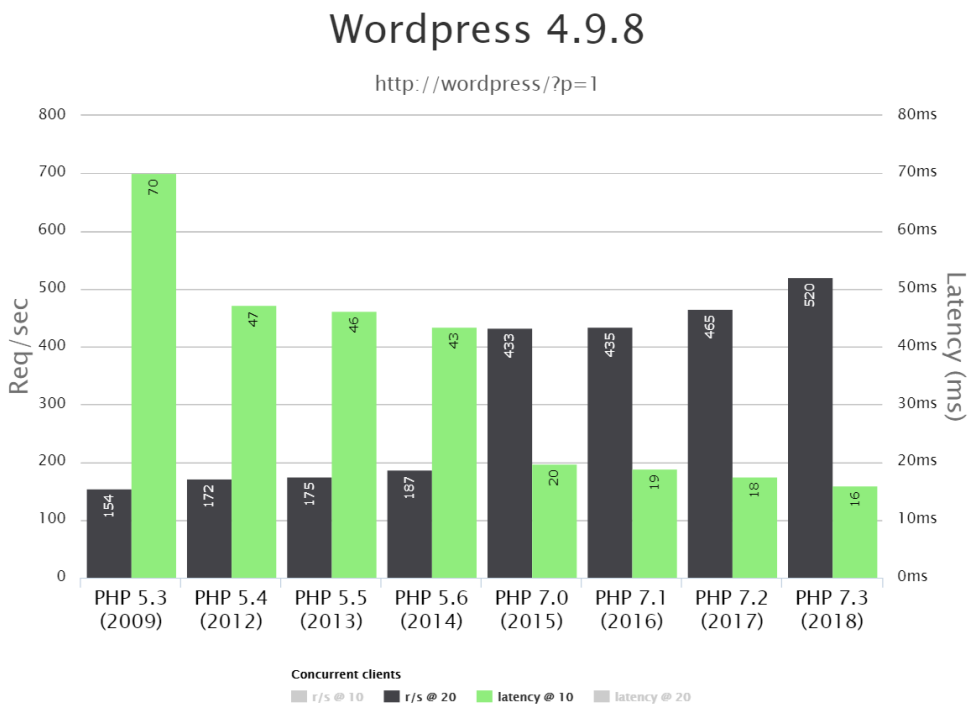
Byly omezeny způsoby, jakými lze třída a funkce definovat. Nelze ji provádět dynamicky a přestaly se používat magické metody.

Neexistují tu žádné globální ani superglobální proměnné. Kód nemůže existovat bez funkce, v kódu je jasným způsobem definován start, funkce, který celý kód spustí, alternativa k funkci `main` v jiných jazycích [26].

Dalšími drobnostmi je respektování velikosti písmen u funkcí, inkrementující a dekrementující příkazy nově nevrací hodnotu, nedají se proto použít během indexaci, přiřazování a dalších operací. To způsobilo, že oba druhy inkrementace splynuly dohromady, v jazyce ale nadále zůstávají obě varianty [27].

3.8 PHP 7.1.

Všechny projekty zmiňované výše pracovaly s PHP verze 5. V té době interpret nebyl na takové úrovni, jako je dnes, s verzí 7. Ta je o dva roky mladší než poslední projekt zabývající se zrychlováním kódu v podobě HHVM v sekci 3.6. Počet požadavků, které interpret zpracuje za sekundu, se zdvojnásobil, zpoždění mezi žádostí a odpovědí je dvakrát menší. Nároky na paměť jsou šestkrát menší [5].

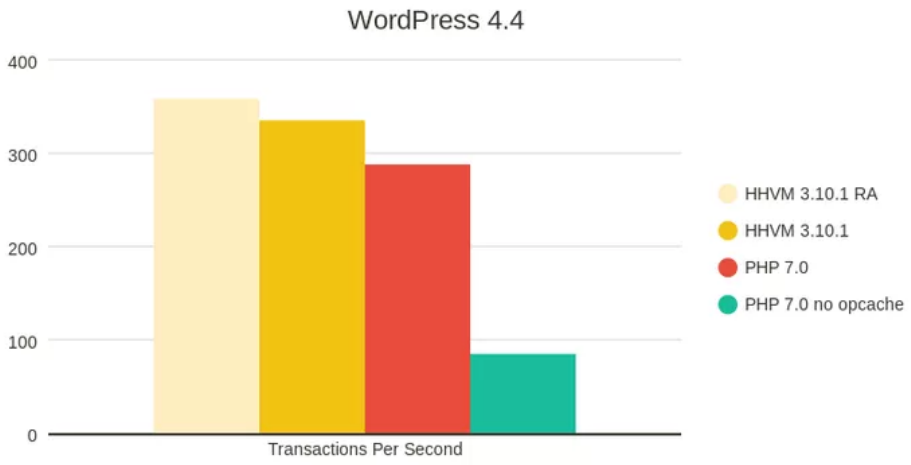


Obrázek 3.1: Zlepšení počtů požadavků za sekundu a čas odezvy napříč verzemi PHP [5]

Z tohoto důvodu je proto otázka zda používat HHVM nebo standardní PHP obtížnější otázkou na zodpovězení. Již existují aplikace, ve kterých je virtuální stroj od společnosti Facebook překonán. HHVM je ale stále efektivnější na zdroje, dokáže provést více transakcí za jednotku času [28]. Problém přichází s ukončením podpory pro PHP, nyní bude možné používat jen Hack, nedá se již proto dále tvrdit, že si přímo konkurují [29].

Nová verze rovněž přidává možnost definovat návratové typy funkcí. Možnosti tu nejsou tak rozsáhlé jako v jazyce Hack, ve kterém je například možné jasným způsobem popsat vracenou anonymní funkci. Tato nová vlastnost bude využívána během transpilace u vlastního řešení.

Jeden z plánů jak dále zrychlit PHP je začít s používáním JIT kompilátoru. Tento požadavek prošel hodnocením a objeví se ve verzi PHP číslo osm. Časy zrychlení se velmi liší, u reálných aplikací jde o hodnoty pod pět procent, skript používáný pro měření výkonu má dvojnásobné zrychlení [30].



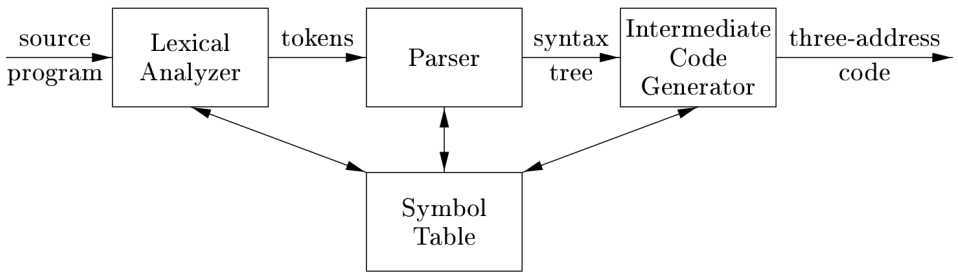
Obrázek 3.2: Porovnání PHP 7.0 a HHVM 3.10.1 [28]

Kapitola 4

Konstrukce kompilátoru

Kompilátor je složitým programem, provádějí hned několik kroků. Jeho cílem je přečíst zdrojový program a převést jej do jiného, ekvivalentního. Pomáhá tak vyřešit problém který přinesly vyšší programovací jazyky. Ty přichází s vlastními operacemi aby udělaly programování jednodušší a odolnější na chyby. Počítače stále pracují s binární informací reprezentující strojový kód a moderní jazyky tak nemají přímou reprezentaci pro cílový počítač. Kompilátor tuto mezeru mezi cílovým jazykem a strojovým jazykem pomáhá překlenout, zdroji navíc přidává přenositelnost – cílem mohou být různé architektury. Kompilátor přichází s celou řadou optimalizací. Jsou dvojího druhu, první skupina upravuje kód bez znalosti cílové architektury, kroky jsou podobné těm při optimalizaci transpilace v 2.1, tedy redukce zbytečných přiřazení a další. Druhá skupina je komplikovanější, závislé na architektuře. To přináší velké množství možností jak optimalizovat, počítače umožňují provádět paralelní výpočty a struktura paměti na složitější hielarchii [31].

Tato práce optimalizace neprovádí, cílem je co nejvěrnějším způsobem zachytit zdrojový kód. Práce zároveň neimplementuje většinu ze zmíněných konstrukcí, pro analýzu zdrojového skriptu se používá knihovna.



Obrázek 4.1: Struktura kompilátoru [31]

Další kapitoly se budou zabývat jednotlivými částmi kompilátoru pro realizaci zajímavé, těmi jsou lexikální, syntaktická a sémantická analýza spolu s generováním kódu. Analýzy jsou založené na teorii automatů, proto se pro popis jednotlivých částí použije právě teorie formálních jazyků a gramatik.

4.1 Lexikální analýza

Lexikální analýza je první komponenta kompilátoru. Převádí vstupní zdrojový kód reprezentovaný jednotlivými znaky na části pro jazyk běžné, kterými jsou například klíčová slova, aritmetické operace, identifikátory a další. Pro jejich konstrukci se používají konečné automaty [32, s. 2]. Jde o model, jehož úkolem je rozhodnout, zda vstupní řetězec symbolů přijme, nebo ne. Všechny sekvence přijaté automatem se nazvou jazykem. Automat se skládá z pěti částí. Jsou jimi vnitřní stavy, vstupní symboly, počáteční stav a množina koncových stavů. Počáteční a koncové stavy jsou z podmnožiny vnitřních stavů. Mezi jednotlivými stavy jsou přechody, které jsou definované překladovou funkcí. Ta podle aktuálního vnitřního stavu a vstupního symbolu změní vnitřní stav automatu [32, s. 51 – 52].

Pro účely analýzy je automat celkem dvakrát upraven. První úpravou je změna vracené hodnoty. Místo toho, aby pouze zodpovídal otázku zda je vstupní řetězec z daného jazyka, vrací „tokeny“. Tímto způsobem se řetězci dal nějaký další význam, již je známo, zda jde o aritmetickou operaci či jinou komponentu zdrojového kódu. Pracuje se s dvěma druhy tokenů. U literálních je význam jasně dán druhem tokenu, lexikální pouze kategorizují vstupní sekvenci, například jako číslo.

Druhá úprava se provádí u ukončovacích stavů. Automat má jen jeden, zde jich bude hned několik, pro každý token jeden. Se zdrojovým kódem se pak nepracuje jako s jednou vstupní sekvencí, ale postupně se dělí na menší logické celky. Díky tomu se na vstupní kód bude nahlížet jako na dlouhou sekvenci tokenů. Pokaždé, co se nalezne nějaký token, se automat spustí znovu.

Ve většině případů se mezery a konce řádků ignorují, v některých jazycích se jim ale dává větší význam, například konec řádku v případě jazyka Go značí konec příkazu. Pokud jazyk podporuje vnořování komentářů do sebe nebo víceslovné proměnné, nejde pro překlad použít konečný automat, protože nemá žádnou vnitřní pamět vhodnou pro počítání počtu vnoření.

U analýzy mohou nastat dva problémy. Mohou totiž existovat tokeny, které začínají stejnými znaky. Vracení prvního výskytu by znamenalo, že by nemohlo vzniknout číslo větší než devět. Analyzátor proto bude hledat tu nejdelší nejdelší sekvenci. Pokud se dojde do nějakého koncového stavu, před vrácením tokenu se ještě zkontroluje, zda nejde sekvenci pomocí pravidla v přechodové funkci prodloužit.

Tento postup funguje dobře u čísel, u komplikovanějších sekvencí ale může nastat stav, kdy již další konečný stav neexistuje, a automat právě teď není

v jednom z konečných stavů. To by vedlo k chybě v překladu, i když tam nemusí být – jeden z vnitřních stavů použitých k překladu mohl být ukončující. Pokud k tomu dojde, místo chyby se vrátí nejdelší sekvence stavů, která končí právě takovým stavem [33, s. 59 – 66].

Velké množství klíčových slov splývá s identifikátory. Je tedy důležité některým identifikátorům dát větší význam. To jde provést dvěma způsoby. Každé klíčové slovo bude mít speciální průchod automatem. To je komplikované, zejména v souvislosti s problémy zmíněné výše, řešení chyby v překladu. Druhý způsob je vložit jej do tabulky symbolů a po každém nálezu identifikátoru jej zpracovat. Symbol má v tabulce informaci o svém speciálním vyznamu. Tabulka symbolů je strukturou, která se používá během celého procesu kompilace, hlídá redeklarace a další problémy, pro lexikální analýzu nepodstatné [31, s. 85 – 86].

4.2 Syntaktická analýza

Regulární gramatika je schopná popsat většinu konstrukcí, nelze s ní ale popsat aritmetické operace. Pro asociativitu a prioritu operátorů se používá bezkontextová gramatika. Jejich přepisovací pravidla jsou více komplexní než u regulárních gramatik a to umožní aritmetické operace jednoznačně popsat [32, s. 194].

Pro popis jazyka se v tomto případě používají derivace. Jde o proces, během kterého se neterminál přepíše pomocí pravidla na kombinaci terminálů a neterminálů, jak je dáno pravidlem. Sekvence symbolů do jazyka patří, pokud existuje taková posloupnost derivací, během které se z počátečního symbolu vytvoří vstupní řetězec. Neterminály se přepisují pomocí pravidel do té doby, dokud se řetězec neskládá pouze z terminálů [32, s. 175].

Derivační strom je grafickou reprezentací tohoto procesu. Startovací symbol je kořenem stromu, listy jsou terminály, vnitřní uzly jsou neterminály. Listy čtené zleva doprava dávají zpracovaný řetězec. Abstraktní syntaktický strom je speciální variantou derivačního stromu vhodného pro uchovávání zdrojového kódu. Je jednodušší, neobsahuje například informace o všech čtených symbolech, závorky a středníky jsou významné jen pro správnou formulaci stromu [32, s. 183 – 184], [31, s. 201 – 202].

Ne každá gramatika je vhodná pro použití v kompilátoru. Stejně jako u lexikální analýzy, i zde je důležitá jednoznačnost. Nejednoznačná gramatika je taková, která pro vstupní řetězec může větší množství rozdílných derivačních stromů. Tato vlastnost je nežádoucí, může vést k nefunkčním prioritám operátorů, což je jeden z důvodů, proč se tato gramatika používá [31, s. 203 – 204]. V programovacích jazycích se používá jedna nejednoznačná konstrukce. Jde o podmínky, každá nemusí mít alternativní větev, a pokud se podmínky do sebe vnořují, nemusí být jasně dané, ke které podmínce blok patří. Pracuje se tu proto s předpokladem, že blok bude patřit předchozí

podmínce [33, s. 101]. Tento problém nenastává v jazyce Go, každá podmínka se musí skládat z celého bloku, nejenom jednoho příkazu.

Všechny gramatiky nejsou jednoznačné, jsou ale způsoby, kterými je jednoznačné udělat. U parsování shora dolů je nutné mít pravidla bez levé rekurze. Pokud symbol v pravidle co nejvíce nalevo je stejný jako ten právě teď převáděný, není možné deterministicky určit, kolik těchto pravidel je potřeba použít, protože symbol, podle kterého se proces může řídit, se nemění. Rekurze lze z pravidel odstranit bez změny jazyka [31, s. 67].

Pro rozhodování, které pravidlo má gramatika použít, se používají dvě pomocné množiny. Každý neterminál tuto dvojici používá. Popisují jednu z vlastností pravidel, a tou je první neterminál, který se na levé straně pravidla může vyskytnout. V množině se může nacházet i prázdný řetězec. Druhá množina uchovává informace o terminálech, které se nachází za daným neterminálním symbolem na pravé straně pravidla. Podle této množiny se rozhoduje v prípadech, kdy první z nich obsahuje prázný řetězec [33, s. 145 – 148]. Množny nesmí obsahovat konflikty, to by znamenalo, že nejde o deterministrický proces. Stejně jako u rekurze, lze provádět úpravy, aby se nejednoznačnost odstranila.

Parserům, které postupují shora dolů a používají tyto množiny se nazývají LL(„k“). Jde o zkratku, první znak značí, že se vstup čte zleva doprava, druhý znamená hledání nejlevější derivace. Proměnná udává, kolik neterminálních symbolů se používá pro určení pravidla, které bude v dalším kroku použito [31, s. 217 – 226].

Kapitola 5

Realizace

Tato kapitola bude zabývat konstrukcí vlastního překladače. Bude rozdělena na čtyři celky. V první popíši požadavky na transpilátor a použité technologie. V další se zmíním o použité knihovně která zjednodušila proces lexikální a syntaktické analýzy na úrovni PHP. Třetí a zároveň největší z nich se bude zabývat problémy, na které jsem během tvorby narazil, a popisuje možnosti, jakými se tyto komplikace vyřešili, případně mohly řešit. Výčet bude seřazen podle významnosti a času, kdy se na těžce převoditelnou konstrukci narazilo. Poslední sekce se bude zabývat způsobem, jakým byla správnost transpilace kontrolována.

5.1 Požadavky

Pro konstrukci transpilátoru jsem použil programovací jazyk Go. Jsou pro to celkem dva důvody, nikterak závažné, dané zejména prostředím, ve kterém se pohybuji. Jde pro mne o druhý nejpoužívanější jazyk, je rovněž na druhé příčce v používanosti ve firmě, ve které pracuji. Vynechávám tu jazyky které se pro psaní překladače nehodí jako jsou HTML a CSS. Druhým důvodem je poukázání na možnosti tohoto jazyka a úmyslné vyhnutí se PHP, kterého se má transpilátor zbavovat.

Transpilátor bude schopný přeložit všechny základní řídící konstrukce nacházející se v PHP. Komplikovanější konstrukce budou fungovat pouze v omezené míře, kdy jejich implementace byla nepřímo vyžadována pro převod jiných konstrukcí.

U všech výrazů se bude vyžadovat, aby bylo možné odvodit jejich datový typ. Podobný krok se provádí u HPHPc, zde je ale nejednoznačnost zakázána a povede k chybě během převodu. Některé konstrukce proto nebude možné přeložit v dané podobě, budou vyžadovány takové úpravy, aby šlo jejich vlastnosti odvodit již během transpilace.

Dynamické vlastnosti jazyka budou zakázány, alternativní definice funkce dané průchodem programu nebude možné provádět. Výjimkou jsou příkazy

pro vkládání skriptů.

Překládat se bude validní PHP kód pro verzi 7.1. Tento požadavek je spojen s novými vlastnostmi tohoto skriptovacího jazyka, tou nejvýznamnější je možnost definovat typy argumentů funkce a její návratový typ. To bude velmi nápomocné pro začátek transpilace, vyžadování této informace zjednoduší deklaraci funkcí a díky ní bude skript jednoznačný.

Jazyk Go je staticky typovaný a tuto vlastnost do jisté míry očekávám i od skriptu. Tohoto omezení se budu moci do jisté míry zbavit až bude transpilace pracovat i s komentáři. Ty mohou dále specifikovat jakého typu vstup je, PHP je u definice polí velmi skromné, nepopisují vnitřní strukturu, používá se jen klíčové slovo `array`. To je něco, co komentář umí upřesnit. Lze je použít i pro inverzní postup, definice bude méně striktní a funkce bude moci vracet/přijímat větší množství typů. Je zde tedy velký prostor pro další rozšíření transpilace a mohlo by tak dojít ke zmírnění požadavku na verzi PHP. Druhým způsobem vedle komentářů je provést větší množství průchodů programem, což je krok, který je možné pozorovat u HPHPc.

5.2 Lexikální a syntaktická analýza PHP

Transpilátor se nezabývá kontrolou PHP kódu a jeho převodem do struktury, se kterou se bude dále dobře pracovat. Pro tuto problematiku bylo použito externí knihovny, která se tímto problémem zabývá. Vytváří abstraktní syntaktický strom, ve kterém je každá řídící konstrukce PHP jednoznačně popsána. Pokud kód není syntakticky validní, je zde možnost dostat se ke všem problémům a tímto způsobem donutit uživatele kód opravit. Strom obsahuje všechny znaky které byly součástí zdrojového skriptu, díky kterým bude možné pracovat s komentáři a ponechat původní formátování programu [34]. Informace o umístění prvků v souboru se ignorují, pro vypsání programu do souboru se používá formátování pro jazyk Go.

Pro analýzu se používají nejnovější pravidla, to například znamená, že `fn` je klíčovým slovem [35].

5.3 Struktura

Program se skládá ze tří balíčků, každý z nich plní jeden specifický úkol.

Pro převedený skript je nejdůležitější balíček „std“, který obsahuje funkce simulující chování PHP. Většina z nich je definována i v Go, ale liší se svým předpisem a návratovou hodnotou. Dále se zde nachází funkce reprezentující implicitní konverze, ty cílový jazyk neprovádí. Navíc neobsahuje všechny konverze jako je tomu v PHP, například z binární hodnoty nelze udělat číslici, balíček toto omezení eliminuje. SQL připojení a asociativní pole se převádí do podoby struktur. Go připojení i asociativní pole implementuje, jeho chování a použití ale není identické. Slouží pro zjednodušení převodu, například v PHP

se připojení k databázi provádí ve dvou krocích, v Go jen v jednom. Struktura připojení k databázi simuluje do podoby dvou funkcí.

Další balíček slouží pro reprezentaci abstraktního syntaktického stromu. Používají se tři druhy uzlů, liší se svými schopnostmi, rozhraním, které musí splňovat. Nejjednodušším z nich je výraz nevracející hodnotu. Do této kategorie spadá přiřazení, inkrementace a dekrementace. Přiřazení má ale jiné postavení, v průběhu transpilace se s ním pracuje jako s uzlem obsahující hodnotu, dochází k jeho úpravám v případě řetězení přiřazení, deklaracím v podmínkách a tak podobně. Další je výraz, který navíc obsahuje informaci o svém návratovém typu. Jde o největší kategorii, spadají do ní nejjednodušší konstrukce jazyka jakými jsou binární operace, volané funkce, primitivní datové typy. Poslední z nich jsou bloky. Jde o kontejnery, které mohou obsahovat další, jiné uzly, a mohou obsahovat definice proměnných. To simuluje chování tabulky symbolů. Do této kategorie spadají funkce, řídící konstrukce, bloky. Pro reprezentaci stromu se používají další dva uzly, které jsou speciálním případem bloků. Splňují rozhraní, ale většina metod neovlivňuje vnitřní stav. Jeden z nich je kořenem celého projektu, obsahuje informace o všech souborech a globálních proměnných. Soubor je uzlem obsahující informace o použitých balíčcích, všech funkcích, a odkazem na funkci symbolizující `main`.

Poslední balíček je pro parser. Ten zdroj převede do podoby stromu, použije k nim uzly popsané výše. Jeho součástí jsou struktury pro překlad názvů. Další obsahuje předpisy všech funkcí, které je možné během překladu použít. Používá se pro výběr správné funkce a vrácení jejího předpisu, navíc do souborů podle potřeby vkládá odkazy na balíčky, aby byl výsledek validní. Z toho důvodu je velmi úzce provázán s oběma balíčky výše, předpisy funkcí jsou definovány textově pouze tady, proto změna standardní knihovny může způsobit, že výsledný program nebude fungovat.

5.4 Konflikty v názvech

PHP má specifický způsob definování proměnných a s funkcemi se váže jeden problém nekonzistence, který skriptovací jazyk ignoruje, ale v cílovém jazyce bude způsobovat problémy. K vyřešení se použije překladová tabulka, která pro frázi v PHP najde vhodný ekvivalent pro Go. Pokud tento název není volný, přidá se na konec číslice, která se bude navyšovat do té doby, dokud fráze nebude unikátní. Tabulky se od liší definicí konfliktů a množinou slov, kterým se musí vyhýbat, projekt přichází již s některými předdefinovanými funkcemi a proměnnými.

Go má velmi malé množství klíčových slov – dvacet sedm [36]. Kroků pro nalezení vhodného názvu by proto nemělo být mnoho. Číslice výše je příliš optimistická, skutečná hodnota nepoužitelných frází je vyšší, ještě se budu vyhýbat názvům balíčků, které se budou používat. Počet se tak navýší až na třicet pět u transpilace do serverové podoby.

5.4.1 Proměnné

Žádná proměnná v PHP nemůže mít konflikt v názvech, například s jmenným prostorem, klíčovým slovem, funkcí či makrem. Jedinou výjimkou jsou superglobální proměnné, tam nějaký konflikt vzniká, ale ne na takové úrovni, aby program přestal fungovat, jen se změní obsah proměnné. Všechny tyto problémy jsou vyřešené dolarem na začátku názvu. To je jedna z věcí, která v Go nejde aplikovat, proměnná nesmí začínat dolarem. Překladová tabulka vrátí správný název proměnné. Konflikty se začnou objevovat právě tehdy, když se proměnná pojmenuje klíčovým slovem v Go. K velkému množství překladů bude poté docházet pokud se klíčové slovo bude vyskytovat v kódu vícekrát, jen s přidáním číslice na konec fráze jako přípona.

5.4.2 Funkce

Nejtriviálnějším konfliktem v názvu je definování funkce `main`. Pokud se program používá jako server, je tu navíc ještě funkce obstarávající spouštění skriptu v podobě konzolové aplikace. Pro překladač funkcí to znamená jen jedno další klíčové slovo, kterému se bude muset vyhýbat. Zajímavějším chováním je přístup k funkcím ze strany PHP – nerespektuje se velikost písmen. Dá se proto tvrdit, že není možné jednoznačně rozhodnout, jak se funkce jmenuje. Název funkce bude dán její definicí a tímto způsobem se vše sjednotí. Před voláním funkce se proto název funkce převede na malá písmena a teprve pak se hledá, zda je funkce definovaná.

5.4.3 Návěstí

Konflikty v těchto identifikátorech pro příkaz `goto` by neměly nastávat, jsou ale transpilátor uměle tvořeny během snahy přeložit klíčová slova `continue` a `break` s číselným identifikátorem velikosti skoku. Postup generování nového názvu je nejjednodušší ze všech, pravidla pro jeho pojmenování jsou v obou jazycích stejná.

5.5 Rozsah platnosti proměnné

Snad každý kompilovaný jazyk váže životnost proměnné k nějakému bloku. Proměnná pro blok o úroveň níže je neznámá a každý potomek bloku o této proměnné ví, pokud si blok nezadefinuje proměnnou s tímto názvem znovu. V PHP se blokové definice ignorují, proměnná existuje i mimo blok. Existují asi tři způsoby, jak toto chování převést do Go:

- Toto chování zakázat. Přístup nejjednodušší a do jisté míry korektní, u složitějších konstrukcí se ignorování bloků nemusí vyplatit a může nastat, že se kód nebude chovat korektně, protože proměnná obsahuje něco jiného, než se očekává.

- Každá proměnná bude definovaná na nejnižší úrovni. Tímto způsobem bude ve staticky typovaném kompilovaném jazyce jasně vidět, za proměnné se používá a jakého datového typu jsou. Na druhou stranu se může hezký skript stát v kompilovaném jazyce stát neatraktivním, což je něco, co není žádoucí, a první způsob má v tomto případě navrh: trestá špatný kód, tento i z dobrého dělá nehezký.

- Použít stejný způsob jako PHP, jen převedený blokového chování. Pokud se nastaví proměnná v bloku pro aktuální blok neznámý, podívá se na všechny ostatní a pokusí se definici najít a vyjmout ji tak, aby ji mohl používat i on. Rozsah platnosti proměnné proto bude tak velký, jak jej vyžaduje PHP. Pokud by se tento způsob aplikoval na všechny proměnné, mohlo by dojít ke stejnému chování jaké je definované v druhém postupu.

Pro transpilaci se používá kombinace všech tří způsobů. Jsou případy, kdy je ignorování bloků velmi špatné, a tím je nastavování proměnné v podmínce. V jedné větvi se proměnná vytvoří, v druhé ne. Nebo se jim nastaví jiný datový typ. To je problém neřešitelný během transpilace a z toho důvodu se vůbec nepřekládá. Pomocí reflexe jde pravou hodnotu proměnné zjistit, ale to by kód udělalo velmi nepřehledným. Navíc by měla být oprava tohoto chování v PHP poměrně jednoduchá.

Přesouvání na nejnižší úroveň se používá v případě globálních proměnných. Každá proměnná, co není umístěná ve funkci, je globální proměnnou, je možné ji používat za pomoci klíčového slova `global`, nebo k ní přistoupit přes superglobální proměnnou – asociativní pole `$GLOBALS`. Vše, co je na nejnižší úrovni, neumístěné v dalším bloku, bude bráno jako proměnná v tomto rozsahu. Výjimkou jsou proměnné určené pro získávání dat z SQL dotazů zmíněné v 5.12.

Pro všechna jiná umístění proměnné v kódu se používá třetí způsob. V průběhu implementace tohoto chování jsem objevil celou množinu případů, kdy k vyjímání musí dojít. Bylo to způsobené mým přístupem k vytváření nových proměnných. Jakmile se zadefinovala ve vnořeném bloku proměnná se stejným jménem ale jiným datovým typem, tranpilátor k ní přistupuje jako k nové proměnné existující pouze v tomto bloku. Mohly proto nastat případy, kdy i na první pohled validní kód mohl dávat špatné výsledky, protože hodnota s daným datovým typem existovala pouze v rozsahu bloku, jak je reprezentováno v příkladu 5.1.

Pro vyřešení této nekonzistence se provádí zpětné hledání i mimo rodičovské bloky. Pokud se v blocích dříve proměnná nachází, její deklarace se přesune na nižší úroveň, prvního společného rodiče dvou bloků. K vyjímání bude docházet jen tehdy, když se proměnná ještě needitovala. Přepsání proměnné v bloku tento proces zastaví.

Toto zpětné hledání není v porovnání s blokovým vyhledáváním výhodné. Místo kontroly na výskyt proměnné v mapě aktuálního prvku a přesunu na

```
$a = 0;
{
    $a = "1";
    echo $a; // "1"
}
echo $a; // "1"
$a = 2;
echo $a; // 2
```

Obrázek 5.1: Příklad neočekávaného chování

rodiče se proces rozšířil na hledání ve všech blocích. Spolu s ním je spojen problém předávání proměnných referencí, nejde provést převedení na správný datový typ a získat odkaz na proměnnou. V některých případech to proto může způsobit chybu během kompilace.

5.6 Vedlejší efekty příkazů

V původním jazyce jsou operace, které mají vedlejší efekty, ale nemají přímou reprezentaci v cílovém jazyce. To vede ke konstrukcím, které se od zdroje v některých případech dost liší, některé z nich jsou zakázané pro svoji komplikovanost.

Nejjednodušším případem, který se nedá do nového jazyka převést, je přiřazení. To zde nevrací hodnotu. Z toho důvodu se nedají řetězit a nemohou se použít v podmínkách.

Go tento problém řeší přidáním možnosti inicializace do podmínky. Předpis tak bude podobný jako u cyklu. Pokud se přiřazení v podmínce objeví, vyjme se a umístí se do inicializace. Přiřazení se odstraní, nahradí se jen proměnnou. Tento postup se používá jen u jednoho přiřazení ve výrazu, složitější operace se nezpracovávají, vedou k pádu programu, nebo k chybě během převodu.

Řetězení na jiném místě než v řídících cyklech je možné – rozdělí se na jednotlivá přiřazení.

Další výraz s vedlejším efektem je inkrementace a dekrementace. V Go nejde o výraz vracející hodnotu, proto nelze používat například pro indexaci, v operacích a další. Oba druhy těchto operací zde splývají podobně jako v Hack, nedají se pro tento účel použít a vedou k chybě transpilace. Aby se chování zachovalo, musela by se použít anonymní funkce. Ta ale nyní není přítomna a její implementace by vedla k možným problémům spojeným s blokovou definicí proměnných.

5.7 Break a continue

Příkazy `break` a `continue` mohou navíc obsahovat číselný argument, který definuje, kolik cyklů se přeruší. Varianty bez čísla jsou identické v cílovém jazyka, pro číselné verze se musí zvolit jiná reprezentace.

Toto chování se bude simulovat dvojicí `goto` a návěstí. Ty se generují podle potřeby, každý s unikátním názvem, který se již nikde jinde nepoužije. Z aktuální bloku se po rodičích ve stromě jde k žádoucímu elementu, a za něj se umístí název návěstí.

5.8 Asociativní pole

Asi nejpoužívanější datovou strukturou ve zdrojovém jazyce je pole, respektive asociativní pole – mapa. Protože jazyk je dynamicky typovaný, jsou rozdíly mezi těmito dvěma druhy pro transpilaci takřka nerozpoznatelné. Tato struktura má dvě vlastnosti, které není možné nějakým jednoduchým způsobem přepsat do Go. Jde o zachovávání pořadí vkládání a jeho vlastnost typická pro primitivní datový typ, přiřazováním se vytváří nová instance, nejde o pouhou referenci [37].

Mnou vytvořená standardní knihovna obsahuje struktury, které lze indexovat stejným způsobem jako tomu je v PHP a uchovává pořadí vkládaných prvků. Vyžaduje se jednoznačnost uchovávaného datového typu, může obsahovat jen hodnoty daného typu. Ve většině případů toto omezení nebude limitací, zejména proto, že pro nejkomplikovanější převáděnou strukturu – výsledek z SQL dotazu – se používá jiný způsob. Funkce používané pro práci s poli se převádí do metod, transpilátor volání těchto funkcí převede do podoby volání metody.

Každý typ má svoji strukturu, pro jejich generování je napsán program. Funguje na jednoduchém principu, doplňuje na správná místa název typu pocházející z příkazového řádku.

Indexovat se dá jakýmkoliv typem, ten se převede do textové podoby. Z desetinných čísel se dělají celá a navzájem se mohou přepisovat stejným způsobem jako je tomu v případě PHP, například není rozdíl mezi indexováním pomocí binární hodnoty `true` nebo textu `"1"`.

Výsledná struktura je velmi pomalá. Mapa se používá pro indexy pole, hodnota odkazuje na hodnotu v poli udržující pořadí vkládaných prvků. Asi nejlepší variantou je snažit se používání této struktury vyhnout. Pokud nebude třeba uchovávat pořadí vkládání, použije se mapa, v případech, kdy indexace není podstatná, se použije běžné pole. Informace o vzhledu pole by se mohla získávat z komentářů.

5.9 Globální proměnné

Řešení globálních proměnných z 5.5 bylo vhodné jen pro překlad kódu, který se nepoužíval jako server. Globální proměnné se v případě serverového chování navzájem přepisují, což vede ke špatným výsledkům, chybám z důvodu nesplnění HTTP protokolu či pádům programu. Tento problém se obchází vytvořením struktury, která všechny globální proměnné obsahuje. Před každým průchodem programu se taková instance struktury vytvoří, a bude se používat jen ta. Každá funkce, která používá globální proměnné, bude součástí této globální struktury. Překladač nad tímto chováním nemá absolutní kontrolu, s metodami se nepracuje přímo, jde jen o vložení textu na dané místo, není to podložené žádnou vlastností datového typu. Alternativním řešením je globální kontext předávat jako argument, zvolil jsem ale první variantu, protože se nemění argumenty funkce. Nemusí se proto upravovat jejich definice a již existující volání funkce zůstávají stejné. Tato varianta jen mění název funkce a to lze upravit během výpisu. K položkám datové struktury je dobrý přístup, vytváří se v kořeni syntaktického stromu jako každá jiná proměnná. Jejich adresování je ale komplikované, dočasně se vytvoří proměnná s názvem složeným ze struktury a položky, ze které se vytvoří výraz, který je používaný stejným způsobem jako jiné proměnné.

V PHP je možné definovat globální proměnné kdekoli, tranpilátor tuto možnost zakazuje, jen ty na nejnižší úrovni nenacházející se ve funkce se stanou globálními. Deklarace pomocí `global` je zakázána, protože není možné odvodit datový typ, a informace o této deklaraci nepůjde získat během jednoho průchodu stromem nebo by vedla k nejednoznačným výsledkům.

5.10 Funkce a proměnný počet argumentů

Funkce v PHP mohou být definované s proměnným počtem parametrů. Toto chování nelze do cílového jazyka jednoznačně převést, protože funkce tímto způsobem používat nelze. Pokud by se zadefinovala funkce se stejným názvem, došlo by k chybě během kompilace, nehledě na jiný zápis v podobě menšího počtu argumentů. Funkce tu jsou od sebe rozlišovány pouze svým názvem, ne celým předpisem.

Pokud se během překladu objeví tato funkce, vygenerují se všechny její varianty s unikátním názvem. Umístí se do kontejneru se všemi funkcemi, identifikovány jsou původním jménem. Každá varianta s kratším seznamem argumentů má stejné tělo funkce, obsahují volání další funkce v řetězci s doplněným argumentem.

Pro zavolání správné varianty funkce se provede porovnání počtu argumentů, se kterými se funkce volá, s úplným počtem. Rozdíl těchto dvou čísel identifikuje její umístění v poli se všemi variantami funkce.

5.11 Vkládání skriptů

Transpilátor právě teď nepracuje s možností překládat větší množství souborů, celé adresáře. Výjimkou jsou soubory dostupné pomocí příkazů `require`, `require_once`, `include`, `include_once`, ty se rovněž přeloží, aby vznikl jeden funkční skript.

Mezi jednotlivými příkazy jsou v PHP rozdíly. Ty končící na `_once` lze vložit jen jednou, pojmenované `require` vyžadují, aby soubor existoval, jinak nastane chyba v průběhu spuštěného skriptu.

Protože dochází ke kompilaci, problémy s neexistencí souboru nemohou nastat. Hlavním důvodem, proč by mělo jít něco vložit jen jednou, je zamezení redeklarací. To ve převedeném skriptu nenastane, proto ke všem případům přistupuji identickým způsobem. Použití těchto výrazů je pro transpilovaný program aliasem pro volání funkce obsahující veškeré konstrukce nenacházející ve funkcí. Stejný způsob zvolil projekt P9 (3.2). Oproti referenčnímu řešení se kontext neuchovává, funkce má proto přístup jen ke globálním proměnným.

Protože se nepřekládají celé projekty, jde o jediný soubor, vkládaný skript musí být jednoznačně identifikován jednoduchým textovým řetězcem. Pokud se daný soubor ještě nepřeložil, dojde k procesu transpilace. Se správnou posloupností těchto příkazů proto může dojít k tomu, že se přeloží celý adresář. Výsledkem celého překladu je jeden další soubor, zvětšení množiny funkcí které lze použít a počtu globálních proměnných. Příkaz je nahrazen voláním funkce v daném souboru reprezentující `main`, příkazy mimo funkci.

Všechny soubory tímto způsobem přeložené bude možné spustit pomocí starého názvu. Pokud se program spustí jako server, bude skript spuštěn stejným způsobem jakým se chová interpret PHP.

5.12 SQL

Převádění SQL dotazů bylo prováděno vlastním, specifickým způsobem, a do transpilátoru to tak přineslo velké množství řídících konstrukcí zabývající se jen touto problematikou. Jde o první případ, kdy se do kódu přidává vlastní anonymní struktura, její definice se získává z komentářů umístěných v kódu. Protože se ve všech případech tato nová struktura deklaruje, je zakázáno do ni přiřazovat jiný datový typ a redeklarovat v rámci bloku. Všechny případy nejsou ošetřené, může to proto vést v chybám v kompilaci výsledného transpilovaného programu.

Tvoření spojení s databází se provádí v PHP ve dvou krocích – nejdříve se zadají přihlašovací údaje, pak se vybere tabulka. Toto chování nelze do Go převést, protože obě akce se provádějí najednou a tvoří se množina připojení. Databázové připojení se reprezentuje strukturou ve vlastní knihovně. První funkce `mysqli_connect` se přeloží jako konstruktor struktury, až při výběru

```
<?php

$link = mysqli_connect("localhost", "root", "");
mysqli_select_db($link, "test");

$r = mysqli_query($link, "SELECT * FROM `table`");
/** @var $a array{id: int, data: string} */
while ($a = mysqli_fetch_array($r, MYSQLI_ASSOC)) {
    echo "$a[id]: $a[data]";
}
```

Obrázek 5.2: SQL připojení v PHP s definovaným návratovým typem

tabulky se vytvoří databázové připojení. Po každém výběru tabulky se navíc zavolá funkce uzavírající připojení.

Chování připojení je podobné tomu v PHP, jen se volání funkcí převádí do podoby metody, prvním argumentem bude identifikátor připojení nebo výsledky dotazu. Pro získání řádků z SQL dotazu jsem ve všech případech používal jednu z funkcí, jejíž volání výsledky, dokud jsou, vrací jako asociativní pole. Podporuje i jiné návratové typy, podporuje se ale jen jedna podoba.

Pokud již výsledek žádné záznamy neobsahuje, vrací se jen binární hodnota. Pro dynamicky typovaný jazyk jiný druh návratového typu nezpůsobuje žádný problém, zde tento postup nejde použít. Pro zjištění, zda dotaz ještě obsahuje nějaké další výsledky, se proto používá jiná metoda. Pokud se metoda vracející výsledek z SQL objeví v podmínkách co vyžadují binární hodnotu, použije se místo tohoto výrazu specializovaná metoda pro tento účel. Metoda se přesune do těla řídící konstrukce.

Aby transpilátor měl úplnou kontrolu nad vraceným výsledkem, musí se návratové typy SQL dotazu někde definovat. Jde o jediný bod v překladu, který nelze vyřešit jinak než připsáním pomocných frází do původního skriptu. Pro tento účel se použijí komentáře, s běžnou strukturou jaká je vidět u definic proměnných ve třídách 5.2. Tento zápis se převede na definici proměnné a ta se bude moci dále použít pro získání hodnot z dotazu. Pokud je typ proměnné pole, převede se do podoby anonymní struktury. Protože se používá externí knihovny, názvy proměnných se tu převádějí do takové podoby, aby byly viditelné z každého balíčku. V Go funkce a proměnné začínající velkým písmenem jsou aliasem pro klíčové slovo `public` z jiných programovacích jazyků. Na zdrojový skript zde není požadavek na pojmenovávání klíčů asociativního pole velkým písmenem, k tomuto převodu bude docházet během transpilace.

Zadefinovaná proměnná bude dále používána pro získávání dat. Balíček „sqlx“ pomocí reflexe výsledek umístí do předané struktury [38].

Komentáře se dají použít i pro definici jiných datových typů. Díky tomu bude jednodušší řešit rozsah platnosti proměnných u podmínek.

5.13 Generování kódu

Každý prvek, který vznikne během transpilace, lze nějakým způsobem vypsat. Ve většině případů jde o postup přímočarý, neovlivněný kontextem, který se vyřeší až po celém procesu zpracování zdrojového skriptu.

Existují zde tři druhy uzlů v abstraktním syntaktickém stromu, jejíž podoba se v průběhu transpilace může měnit.

Prvním z nich je proměnná. Její výpis je závislý způsobu jejího používání. Pokud v průběhu transpilace bude změněn její typ, dojde k převodu na generický `interface{}`, její používání bude navíc obsahovat přetypování na jasně definovaný typ který proměnná v daný moment obsahuje.

Další je funkce, jejíž vzhled je ovlivněm používanými prostředky. Ke změně předpisu nedojde tehdy, pokud nepoužívá globální proměnné a bude proto třeba ji změnit na metodu. Pokud se skript převádí do serverové podoby, i výpisy způsobí, že bude vyžadován globální kontext.

Uzlem pracující na největší úrovni abstrakce je inkrementace a dekrementace. Pokud je proměnná definovaná jako číselný datový typ, vzhled je stejný jako u jiných programovacích jazyků. Pokud se pracuje s neznámým datovým typem, použije se přiřazení, u kterého nenastává problém s přetypováním. Pokud je proměnná textovým řetězcem, použije se rovněž přiřazení, ale místo sčítání se použije specializovaná funkce z mé standardní knihovny určená pro tento datový typ. Jde o jeden ze dvou uzlů, jehož textová podoba může i po úspěšné transpilace způsobit, že kód nebude funkční. Prvním z nich je volání funkce pracující s argumentem předávaným referencí.

5.14 Testování a porovnávání výkonu

Tato sekce se bude zabývat testováním chování transpilátoru a kontrolou výstupu. Dále se bude zkoumat jeho výkonnost v porovnávání s řešením v původním jazyce.

5.14.1 Testování

Testování se provádí na třech úrovních. Na první z nich se provádí jednoduché kontroly zda se vytváří všechny logické prvky používané v kompilátoru správně, nastavení rodičů, správný počet proměnných v prázdném projektu a tak podobně.

Další kontrola se zabývá kontrolou výstupu z programu. Jde o textové porovnávání výstupu z kompilátoru a očekávaným výstupem. To pomohlo odhalit komplikovanější konstrukce, pro které je porovnávání na úrovni prvků příliš komplikované. Tímto způsobem se odhalovaly problémy spojené s vyjímaním proměnných aby splňovaly blokové deklarace.

Oba způsoby výše se nezabývaly výstupem transpilovaného programu. Pro to slouží poslední způsob, který se skládá ze tří dílčích činností. První

```
<?php

$start = microtime(true);

$c = 0;
for ($i = 0; $i < 10; $i++) {
    for ($j = 0; $j < 32000; $j++) {
        for ($k = 0; $k < 32000; $k++) {
            $c++;
            if ($c > 50) {
                $c = 0;
            }
        }
    }
}

$end = microtime(true);
$time = $end - $start;
printf("Execution time of script = %f sec.\n", $time);
```

Obrázek 5.3: Skript použitý pro porovnávání času

z nich spustí interpretr PHP se zdrojovým souborem a jeho výsledek si uloží pro pozdější porovnávání. V dalším kroku se soubor zpracuje transpilátorem a spustí se i ten. Stejně jako u PHP, i zde se uloží jeho výsledek a porovná se s výstupem zdroje. Porovnávání je časově nejnáročnější, ale je schopné odhalit velké množství problémů, které předchozí způsoby nezvládly kvůli absenci kompilace.

5.14.2 Porovnávání výkonu

Zkoušení výkonností nového řešení bylo prováděno celkem dvěma způsoby, lišící se způsobem, jakým byl program spuštěn. Pro časově náročné operace se používalo konzolového chování, bez spouštění serveru. Pro jednodušší věci byl použit program Apache HTTP server benchmarking tool [39]. Ten pomohl odhalit problémy spojené se souběhem u globálních proměnných zmíněné v 5.9.

Protože pro porovnávání běhů programu není jeden počítač s velkým množství jiných spuštěných procesů dobrým testovacím prostředím, spouštěly se skripty, u kterých byla časová náročnost řádově vyšší, nemohlo proto jít o jeden špatný běh skriptu.

Interpretovaná podoba skriptu 5.3 trvala v průměru sto šedesát sekund, transpilovaná verze je desetkrát rychlejší. U jiných testů, například cyklické přiřazování do textového řetězce nebo řazení pole, je převedený skript po-

malejší. Problémy s asociativním polem lze vyřešit jejich lepší implementací, nebo je vůbec nepoužívat – adresování typické pro mapy není vždy používáno. Problémy spojené s řetězci by byly řešené podobným způsobem. Řetězce jsou v PHP editovatelné, nevytváří se kopie, proto práce s nimi v porovnání s naivní implementací v Go tak rychlá. S větší znalostí kódu by se místo řetězců mohly používat specializované struktury, které kopie nevytváří. Jejich využití způsobí, že skript se vykoná v podobném čase, rozdíl je zhruba desetiprocentní ve prospěch transpilované verze.

Apache benchmarking tool byl použit pro časově méně náročné operace, kde výstupem byla HTML stránka. Pro její tvorbu se neprovádí velké množství výpočtů. Cílem bylo vyzkoušet, jak se server bude chovat při zpracovávání většího množství požadavků najednou. Aby výsledky byly pokud možno co nejvíce izolované, nepoužívá se databáze, pracuje se jen s `GET` parametry. Práce s tímto nástrojem byla zkomplikována rozdílným chováním serverů při zahlcení požadavky. Transpilovaná verze požadavky odmítá, integrovaný PHP server zpracovává všechny, což vedlo k neporovnatelným výsledkům.

Pro porovnávání se použil skript, který vypíše do tabulky `GET` parametry, klíč i hodnotu. K vyřešení tohoto problému byla použit vlastní definice asociativního pole a cyklus, tudíž vlastní řešení v nějaké omezené míře ovlivňuje výsledný čas. Pokud se synchronně spustí sto tisíc požadavků, časy jsou obdobné. Pokud bude požadavků paralelně padesát, čas je dvakrát lepší u transpilované verze.

Závěr

Cílem bylo převést vybranou množinu konstrukcí nacházející se v jazyce PHP a převést je do jazyka Go tak, aby se nezměnil výsledek. K optimalizacím nedocházelo, struktura programu zůstala pokud možno co nejvíce podobná původnímu skriptu.

Všechny funkcionality běžné pro staticky typovaný jazyk byly převedeny, zkompilovaný program připomíná chování interpretu PHP. Většina dynamického chování je zakázána, nelze proto například definovat funkce v průběhu programu nebo používat „proměnnou proměnných“. Hodnota `null` v převedeném jazyce neexistuje, předpokládá se, že je vše definované. Ze stejného důvodu jde funkce `isset` a `unset` používat jen u polí. Vkládání skriptů lze provádět kdekoli, musí mít ale podobu jednoduchého textového řetězce, aby došlo k přeložení. U proměnných se vyžaduje jednoznačné určení datového typu, jinak transpilace končí neúspěchem. Serverové chování má zjednodušený přístup ke `GET` parametrům, každý z nich bude jednoduchým textovým řetězcem, nepůjde o pole, jak k tomu může při správném pojmenování klíče dojít.

Lze transpilovat SQL připojení, což je jeden z bodů, co do základních konstrukcí nepatří, jde o rozšiřující knihovnu pro interpretr. K implementaci došlo, aby bylo možné kompletně převést již existující projekt.

U některých aplikací dojde až k desetinásobnému zrychlení. Projekt je připraven na přidávání dalších funkcí obsažených v PHP, které ještě nejsou implementované.

Převedený zdrojový kód umožňuje odhalit konstrukce, které se dají optimalizovat. Mezi ty hlavní patří databázové připojení, které lze v cílovém jazyce ve většině případů provést jen jednou.

Byla popsána historie PHP a jeho vývoj. Byly zmíněny způsoby, jakými lze interpretovaný jazyk zrychlit. Popsaly se projekty zabývající se touto problematikou. Před implementací transpilátoru došlo k popisu jeho konstrukce.

Práce se dále může vyvíjet několika směry. Hlavním cílem bude rozšířit možnosti překladu, teď se zcela ignorují jmenné prostory, ve standardní knihovně nejsou funkce pro autentifikaci. Dále chybí překlad tříd. Výsledný kód

nimi pracuje, neumí je ale zpracovat. Dalším způsobem jak projekt upravit, je přidání dalšího cílového jazyka. Právě teď na to transpilátor není vůbec připraven, před tímto krokem bude nutné provést úpravu ve výpisu a vnitřní reprezentaci zdrojového kódu.

Bibliografie

1. BERNERS-LEE, Tim. *HyperText and CERN* [online]. 1989 [cit. 2020-05-01]. Dostupné z: `https://www.w3.org/Administration/HTandCERN.txt`.
2. W3C. *STANDARDS* [online] [cit. 2020-03-29]. Dostupné z: `https://www.w3.org/standards`.
3. SUPERCOMPUTING APPLICATIONS (NCSA) AT THE UNIVERSITY OF ILLINOIS, National Center for. *NCSA MOSAIC™* [online] [cit. 2020-04-23]. Dostupné z: `http://www.ncsa.illinois.edu/enabling/mosaic`.
4. D. ROBINSON, K. Coar. *The Common Gateway Interface (CGI) Version 1.1* [online]. RFC Editor, 2004 [cit. 2020-04-23]. Dostupné z DOI: `10.17487/RFC3875`. RFC. RFC Editor.
5. LERDORF, Rasmus. *25 Years of PHP (by the Creator of PHP)* [online]. 2019 [cit. 2020-04-23]. Dostupné z: `https://www.youtube.com/watch?v=wCZ5TJCBWMg`.
6. KOSEK, Jiří. 1. Úvod. In: *PHP – tvorba interaktivních internetových aplikací* [online]. 1999, s. 17–22 [cit. 2020-04-23]. ISSN 80-7169-373-1. Dostupné z: `https://www.kosek.cz/php/php-tvorba-interaktivnich-internetovych-aplikaci.pdf`.
7. INTERNATIONAL, ECMA. I.6 Overview of the Common Language Infrastructure. In: *Standard ECMA-335 - Common Language Infrastructure (CLI)* [online]. 2010 [cit. 2020-04-23]. Dostupné z: `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.
8. Common Language Runtime (CLR) overview. In: *Microsoft Docs* [online]. 2019 [cit. 2020-04-20]. Dostupné z: `https://docs.microsoft.com/en-us/dotnet/standard/clr`.

9. LOVEMAN, David B. Program Improvement by Source-to-Source Transformation. *J. ACM*. 1977, roč. 24, č. 1, s. 121–125. ISSN 0004-5411. Dostupné z DOI: 10.1145/321992.322000.

10. Introduction. In: *Trustworthy Compilers* [online]. John Wiley & Sons, Ltd, 2010, kap. 1, s. 4 [cit. 2020-04-17]. ISBN 9780470593387. Dostupné z: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470593387.

11. ZHAO, Haiping et al. The HipHop Compiler for PHP. *SIGPLAN Not.* [online]. 2012, roč. 47, č. 10, s. 575–586 [cit. 2020-04-15]. ISSN 0362-1340. Dostupné z DOI: 10.1145/2398857.2384658.

12. Trustworthy Optimizations. In: *Trustworthy Compilers* [online]. John Wiley & Sons, Ltd, 2010, kap. 6, s. 170–173 [cit. 2020-04-17]. ISBN 9780470593387. Dostupné z: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470593387.

13. Runtime, JIT, and AOT Compilation. In: *Trustworthy Compilers* [online]. John Wiley & Sons, Ltd, 2010, s. 203–211 [cit. 2020-04-17]. ISBN 9780470593387. Dostupné z: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470593387.

14. BENDA, Jan; MATOUSEK, Tomas; PROSEK, Ladislav. Phalanger: Compiling and running PHP applications on the Microsoft .NET platform. *.NET Technologies 2006* [online]. 2006, s. 31–38 [cit. 2020-04-15]. Dostupné z: http://oot.zcu.cz/NET_2006/Papers_2006/!Proceedings_Full_Papers_2006.pdf.

15. PHP 5 ChangeLog. In: *PHP* [online]. 2019 [cit. 2020-04-20]. Dostupné z: https://www.php.net/ChangeLog-5.php.

16. PEACHPIE. *Use cases* [online] [cit. 2020-04-23]. Dostupné z: https://www.peachpie.io/usecases.

17. SDK, The .NET Compiler Platform [online]. 2017 [cit. 2020-04-23]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk.

18. TATSUBORI, Michiaki; TOZAWA, Akihiko; SUZUMURA, Toyotaro; TRENT, Scott; ONODERA, Tamiya. Evaluation of a Just-in-Time Compiler Retrofitted for PHP. *SIGPLAN Not.* [online]. 2010, roč. 45, č. 7, s. 121–132 [cit. 2020-04-23]. ISSN 0362-1340. Dostupné z DOI: 10.1145/1837854.1736015.

19. SLATTERY, Ed; TAYLOR, Judy. Get started with Project Zero and PHP. In: *developerWorks* [online]. 2007 [cit. 2020-04-17]. Dostupné z: https://www.ibm.com/developerworks/web/tutorials/wa-pz-php/wa-pz-php-pdf.pdf.

20. NICHOLSON, Rob. Project Zero. In: *international PHP 2007 conference* [online]. 2007 [cit. 2020-04-17]. Dostupné z: https://www.ibm.com/developerworks/collaboration/uploads/phpblog/zeroPHPforIPC.pdf.

21. quercus: php in java. In: *Caucho* [online] [cit. 2020-04-17]. Dostupné z: https://www.caucho.com/resin-3.1/doc/quercus.xtp.

22. RASMUS, Lerdorf. *Rasmus' Toys Blog*. HipHop PHP - Nifty Trick? [online]. 2010 [cit. 2020-05-13]. Dostupné z: https://toys.lerdorf.com/hiphop-php-nifty-trick.

23. ADAMS, Keith; EVANS, Jason; MAHER, Bertrand; OTTONI, Guilherme; PAROSKI, Andrew; SIMMERS, Brett; SMITH, Edwin; YAMAUCHI, Owen. The Hiphop Virtual Machine. *SIGPLAN Not.* [online]. 2014, roč. 49, č. 10, s. 777–790 [cit. 2020-04-15]. ISSN 0362-1340. Dostupné z DOI: 10.1145/2714064.2660199.

24. Migrating from JavaScript. *TypeScript* [online]. 2020 [cit. 2020-05-13]. Dostupné z: https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html.

25. VERLAGUET, Julien; MENGHRAJANI, Alok. Hack: a new programming language for HHVM. In: *Facebook Engineering* [online]. 2014 [cit. 2020-04-17]. Dostupné z: https://engineering.fb.com/developer-tools/hack-a-new-programming-language-for-hhvm.

26. PHP, Differences from [online] [cit. 2020-04-17]. Dostupné z: https://github.com/facebookarchive/hack-langspec/blob/master/spec/23-differences-from-php.md.

27. EXPRESSIONS; INCREMENTING, Operators: DECREMENTING [online] [cit. 2020-04-17]. Dostupné z: https://docs.hhvm.com/hack/expressions-and-operators/incrementing-and-decrementing.

28. ARSENAULT, Cody. PHP 7 vs HHVM - Which One Should You Use? In: *KeyCDN* [online]. 2014 [cit. 2020-05-13]. Dostupné z: https://www.keycdn.com/blog/php-7-vs-hhvm.

29. EMMOTT, Fred. *HHVM*. Ending PHP Support, and The Future Of Hack [online]. 2018 [cit. 2020-05-13]. Dostupné z: https://hhvm.com/blog/2018/09/12/end-of-php-support-future-of-hack.html.

30. STOGOV, Dmitry; SURASKI, Zeev. PHP RFC: JIT. In: *PHP* [online]. 2019 [cit. 2020-05-13]. Dostupné z: https://wiki.php.net/rfc/jit.

31. AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. 1. Introduction. In: *Compilers: Principles, Techniques, and Tools (2nd Edition)* [online]. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, s. 1–23 [cit. 2020-04-12]. ISBN 0321486811. Dostupné z: http://ce.sharif.edu/courses/94-95/1/ce414-2/resources/root/Text%20Books/Compiler%20Design/Alfred%20V.%20Aho,%20Monica%20S.%20Lam,%20Ravi%20Sethi,%20Jeffrey%20D.%20Ullman-Compilers%20-%20Principles,%20Techniques,%20and%20Tools-Pearson_Addison%20Wesley%20(2006).pdf.

32. HOPCROFT, John E.; MOTWANI, Rajeev; ULLMAN, Jeffrey D. *Introduction to automata theory, languages, and computation* [online]. 3rd ed. Pearson/Addison Wesley, 2007 [cit. 2020-04-12]. ISBN 9780321455369. Dostupné z: http://ce.sharif.edu/courses/94-95/1/ce414-2/resources/root/Text%20Books/Automata/John%20E.%20Hopcroft,%20Rajeev%20Motwani,%20Jeffrey%20D.%20Ullman-Introduction%20to%20Automata%20Theory,%20Languages,%20and%20Computations-Prentice%20Hall%20(2006).pdf.

33. BARRETT, William A. Compiler Design [online]. 2000 [cit. 2020-04-12]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.462.9894&rep=rep1&type=pdf.

34. SLIZOV, Vadym. *PHP Parser* [software]. Dostupné také z: https://github.com/z7zmey/php-parser.

35. MORRISON, Levi; WEINAND, Bob. *PHP*. PHP RFC: Arrow Functions [online]. 2014 [cit. 2020-05-13]. Dostupné z: https://wiki.php.net/rfc/arrow_functions.

36. LANGUAGE, The Go Programming. *The Go Programming Language Specification* [online]. 2020 [cit. 2020-04-23]. Dostupné z: https://golang.org/ref/spec.

37. Arrays. *PHP* [online] [cit. 2020-05-18]. Dostupné z: https://www.php.net/manual/en/language.types.array.php.

38. MOIRON, Jason. *sqlx* [software]. Dostupné také z: https://github.com/jmoiron/sqlx.

39. APACHE. *ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4* [software]. Dostupné také z: https://httpd.apache.org/docs/2.4/programs/ab.html.

Příloha **A**

Seznam použitých zkratek

PHP PHP: Hypertext Preprocessor

AST Abstract syntax tree

JIT just-in-time

HPHPc HipHop compiler

HPHPi HipHop interpreter

HHVM HipHop Virtual Machine

HHBC HipHop bytecode

HHIR HipHop intermediate representation

WWW World Wide Web

SQL Structured Query Language

CGI Common Gateway Interface

CLI Common Language Infrastructure

NCSA National Center for Supercomputing Applications

Příloha B

Obsah přiloženého CD

- /
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `src`
    - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `php2go` ..... zdrojové kódy implementace
    - `examples` ..... příklady
      - `readme.md` ..... popis jednotlivých příkladů
      - `cli`
      - `server`
      - `benchmark`
  - `dist` ..... kompilované transpilátory pro různé architektury