Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Překladač funkcionálního programovacího jazyka
na platformě GraalVM

2023

Bc. Ivo Horák

Vedoucí práce:
Mgr. Petr Krajča, Ph.D.

Studijní program: Aplikovaná informatika,
Specializace: Vývoj software

Bibliografické údaje

Autor:	Bc. Ivo Horák
Název práce:	Překladač funkcionálního programovacího jazyka na platformě GraalVM
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	Mgr. Petr Krajča, Ph.D.
Počet stran:	79
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Bc. Ivo Horák
Title:	Functional Programming Language Compiler on GraalVM Platform
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	Mgr. Petr Krajča, Ph.D.
Page count:	79
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

V rámci diplomové práce byl vytvořen interpret jazyka Scheme na platformě GraalVM, který svou rychlostí ve vybraných benchmarkových testech překonává implementace jako Guile nebo Racket. Zatímco většina interpretů překládá kód do bytecode, jazyky implementované nad platformou GraalVM jsou překládány do abstraktních syntaktických stromů. Tímto bylo možné implementovat jazykové vlastnosti, jako například optimalizaci koncových volání, které JVM nepodporuje. Dále interpret podporuje interoperabilitu s jazyky implementovanými na platformě GraalVM, a tím umožňuje polyglotní programování.

Synopsis

In this master thesis, an interpreter of Scheme language was developed on the GraalVM platform that outperforms implementations such as Guile or Racket in selected benchmarks. While most implementations compile code into bytecode, languages implemented on the GraalVM platform are transformed into abstract syntax trees. This approach makes it possible to implement language features, such as Tail Call optimization, which is not natively supported by JVM. Furthermore, the interpreter supports interoperability with languages implemented on the GraalVM platform, thereby enabling polyglot programming.

Klíčová slova: GraalVM; TruffleScheme; polyglotní programování; partial evaluation; Truffle

Keywords: GraalVM; TruffleScheme; polyglot programming; partial evaluation; Truffle

Chtěl bych poděkovat Mgr. Petru Krajčovi, Ph.D. za cenné rady, testování a věcné připomínky při konzultacích a během tvorby této diplomové práce.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

1 Úvod **8**
- 1.1 Motivace . . . 9
- 1.2 Cíle a rozsah práce . . . 9
  - 1.2.1 Podpora funkcionálních prvků . . . 10
  - 1.2.2 Rychlost . . . 10
  - 1.2.3 Podpora primitivních procedur a speciálních forem . . . 10
  - 1.2.4 Podpora polyglotního programování . . . 10

2 Platforma Java **12**
- 2.1 Virtuální stroj Javy . . . 12
- 2.2 Just-in-time kompilace . . . 14
  - 2.2.1 Klientský překladač . . . 14
  - 2.2.2 Serverový překladač . . . 14
  - 2.2.3 Vícevrstvá kompilace . . . 15
  - 2.2.4 Optimalizace realizované JIT překladači . . . 16
    - 2.2.4.1 Alokace registrů . . . 16
    - 2.2.4.2 Rozbalení programových smyček . . . 17
    - 2.2.4.3 Výpočet konstantních výrazů a jejich propagace . . . 17
  - 2.2.5 Způsoby JIT kompilace . . . 18
    - 2.2.5.1 Meta-tracing . . . 19
    - 2.2.5.2 Partial evaluation . . . 20
    - 2.2.5.3 Meta-kompilace při psaní interpretů . . . 23
    - 2.2.5.4 Rozdíl mezi meta-tracing a partial evalution . . . 23

3 Platforma GraalVM **25**
- 3.1 Framework Truffle . . . 25
- 3.2 Sebeoptimalizace AST interpretů . . . 26
- 3.3 Truffle API . . . 27
  - 3.3.1 Node . . . 27
  - 3.3.2 VirtualFrame . . . 28
  - 3.3.3 RootNode . . . 29
  - 3.3.4 CallTarget . . . 29
- 3.4 Specializace . . . 29
  - 3.4.1 Specializace a výkon aplikace . . . 32
  - 3.4.2 Specializace a partial evaluation . . . 35
- 3.5 Truffle DSL . . . 38

4 Scheme na platformě GraalVM **41**
- 4.1 Datové typy . . . 41
- 4.2 Primitivní procedury . . . 42
- 4.3 Speciální formy . . . 42
- 4.4 Makra . . . 43

5 Implementace interpretu **44**
5.1 Architektura interpretu . . . 44
5.2 Lokální proměnné . . . 44
5.3 Globální proměnné . . . 47
5.4 Uživatelsky definované procedury . . . 49
5.5 Primitivní procedury . . . 51
5.6 Speciální formy . . . 53
5.7 Makra . . . 54
5.8 Optimalizace koncových volání . . . 57
5.8.1 Implementace optimalizace koncové rekurze . . . 58
5.8.2 Implementace optimalizace koncových volání . . . 62

6 Podpora interoperability **63**
6.1 Použití InteropLibrary knihovny . . . 63
6.2 Podpora objektů z cizích jazyků . . . 65
6.3 Polyglotní API . . . 66

7 Srovnání s ostatními implementacemi **70**
7.1 Srovnání s Guile implementací . . . 70
7.2 Srovnání s Racket implementací . . . 71

Závěr **74**

Conclusions **75**

A Obsah elektronických dat **76**

Literatura **77**

Seznam obrázků

1 Porovnání rychlosti jazyků z roku 2017 [10] . . . 11
2 Schéma znázorňující vztah mezi JDK, JRE a JVM . . . 12
3 Příklad překladu jazyka Java do bytecode a následně do strojového kódu [14] . . . 13
4 Růst výkonu s využitím vícevrstvé kompilace [15] . . . 15
5 Příklad uživatelského programu (levý sloupec), jeho neoptimalizovaný záznam (sloupec uprostřed) a optimalizovaný záznam (sloupec vpravo) [20] . . . 19
6 Schéma funkce s dvěma argumenty n a k . . . 20
7 Aplikování partial evaluation na funkci f za předpokladu, že hodnota argumentu k je konstantní . . . 21
8 První Futamorova projekce . . . 24
9 Schéma architektury GraalVM . . . 25
10 Uzel reprezentující operaci sčítání se všemi možnými přechody od neinicializovaného stavu `U` až do generického stavu `G` . . . 31
11 Porovnání rychlosti s rostoucím počtem specializací. V případě, že počet specializací je nízký, může dojít k náhlému propadu ve výkonu a způsobit tzv. performance cliff [23] . . . 32
12 Varianty vrstvení dvou specializací k zamezení náhlému propadu ve výkonu (performance cliff) [23] . . . 34
13 Použití anotace `@CompilationFinal` u vlastnosti způsobí eliminaci větvení, a tudíž výsledný strojový kód je výrazně kratší a efektivnější [24] . . . 35
14 Aplikace PE na program reprezentující sčítání tří hodnot. Postupně jsou eliminovány jakékoliv známky interpretu [24] . . . 36
15 Na základě profilujících dat se uzly specializují (1). Jakmile je strom stabilní a počet volání metody překročí hranici pro JIT kompilaci, je na specializovaný strom nejprve aplikována PE a poté je přeložen do strojového kódu (2) [25] . . . 37
16 Jestliže spekulace stability stromu je porušena, dochází k deoptimalizaci, strojový kód je zahozen a AST strom se interpretuje (1). Uzly se opět specializují na základě profilujících dat (2), a jakmile jsou podmínky pro JIT kompilaci znovu splněny, dochází k další kompilaci do strojového kódu (3) [25] . . . 37
17 Architektura interpretu . . . 44
18 Pole argumentů při volání procedury. Argument na indexu `i` se získá pomocí vzorce `i + 1` . . . 49
19 Velikosti zásobníku roste s každým dalším voláním procedury v případě, že není aplikována optimalizace koncových volání [7] . . . 58
20 Velikost zásobníku je konstantní s každým dalším voláním procedury, v případě, že je aplikována optimalizace koncových volání [7] . . . 59

21 Výkonnostní srovnání Guile, Racket a TruffleScheme. TruffleScheme je možné buďto spustit nad JVM, nebo využít technologii SubstrateVM . . . 72
22 Interní reprezentace (Graal IR) těla procedury `tak` . . . 73

Seznam tabulek

1 Význam operátorů použitých pro definovaní gramatik . . . 42
2 Výsledky měření jednotlivých implementací v milisekundách . . . 70

Seznam vět

1 Definice (Meta-kompilace) . . . 18
2 Definice (Koncová pozice) . . . 57

Seznam zdrojových kódů

1 Výpočet konstantních výrazů a jejich propagace . . . 18
2 Partial evalution aplikovaná na funkci `pow` . . . 22
3 Příklad implementace uzlu pro sčítání argumentů typu `int` . . . 28
4 Příklad implementace uzlu pro sčítání dvou čísel bez využití Truffle DSL [26] . . . 39
5 Implementace uzlu pro sčítání dvou čísel s využitím Truffle DSL . 40
6 Uzel zapisující lokální proměnnou . . . 46
7 Jazykový kontext obsahující mapu pro ukládání globálních proměnných . . . 48
8 Uzel volající uživatelsky definované procedury . . . 51
9 Uzel volající primitivní procedury . . . 53
10 Uzel reprezentující speciální formu `if`, která neobsahuje náhradníka 54
11 Uzel reprezentující volání makra . . . 56
12 Procedura `trampoline` v tomto případě bude sloužit jako trampolína (vnější procedura), která postupně bude volat vnitřní procedury `foo`, `bar` a `baz` . . . 60
13 Uzel, který místo volání procedury vyhodí výjimku . . . 61
14 Interní reprezentace primitivní procedury, využívající InteropLibrary knihovnu . . . 64
15 Implementace operace `car` podporující objekty z cizích jazyků . . 66
16 Implementace Takeuchi funkce v jazyce Scheme . . . 72

1 Úvod

S rostoucí oblibou funkcionálních jazyků vzniká stále větší množství nových jazyků, které umožňují programátorům vyvíjet rychleji a efektivněji, zejména v oblastech webového vývoje, strojového učení nebo umělé inteligence. Obecně existují dva způsoby, jak implementovat nový interpretovaný jazyk. Buďto je vytvořen nový virtuální stroj, který je navržený přesně na míru vlastnostem daného jazyka, nebo je možné implementovat jazyk nad již existujícím virtuálním strojem. Často je volena právě druhá varianta, protože tvorba nového virtuálního stroje je velice náročná a mnoho částí (např. správa paměti) je již mnohokrát implementováno. Clojure [1], Scala [2] nebo Ruby [3] lze uvést jako příklady jazyků, které jsou implementovány nad již existujícím virtuálním strojem Javy (JVM). Tento přístup však s sebou nese i mnoho úskalí. Například JVM bylo vytvořeno pro staticky typovaný jazyk, a proto implementace dynamicky typovaného jazyka musí obsahovat mnoho typových kontrol, což má negativní dopad na výkon interpretu daného jazyka.

Dále je pro funkcionální jazyky například typická *optimalizace koncových volání* (ang. tail call optimalization), kterou JVM v současné době vůbec nepodporuje. To není problém pro imperativní jazyky, jako je Java, které obsahují cykly. Nicméně mnoho čistě funkcionálních jazyků nahrazuje cykly rekurzí. V takovém případě by mohla být vyčerpána maximální velikost zásobníku a program by skončil chybou. Proto některé implementace musí dělat kompromisy. Například Clojure představil speciální klíčové slovo `recur` [4], které značí koncovou rekurzi (metoda volá sama sebe v koncové pozici). Ta je během překladu převedena na smyčku, čímž je eliminováno riziko vyčerpání zásobníku. Je důležité poznamenat, že Clojure tímto vyřešil jen koncovou rekurzi, což je speciální případ koncového volání. Projekt Loom by potenciálně mohl přidat nativní podporu koncového volání do JVM, ale v době psaní diplomové práce není jisté, kdy a v jaké podobě bude podpora implementována [5].

Všechna tato úskalí jsou řešitelná pomocí nového frameworku zvaného Truffle od společnosti Oracle, který je součástí platformy GraalVM. Jedná se o framework, který slouží pro implementaci nových jazyků nad platformou GraalVM. Ve své podstatě se jedná o sadu tříd, rozhraní a anotací, které vývojář při implementaci nového jazyka využívá. Součástí platformy je i nový serverový překladač zvaný *Graal* (odtud název GraalVM), který Truffle anotacím rozumí a společně tak během JIT kompilace generují velice efektivní strojový kód. Jelikož Truffle nepřekládá kód do Java bytecode, mohou být výše zmíněná omezení JVM buďto úplně, nebo alespoň částečně odstraněny. Jazyky implementované pomocí frameworku Truffle jsou reprezentovány jako abstraktní syntaktické stromy (AST), které jsou následně interpretovány. Interpret je standardní Java aplikace překládaná do bytecode, přičemž jazyk sám do bytecode překládán není, jako je tomu v případě JRuby nebo Scala. Během interpretace jazyka Truffle sbírá profilující data a na základě nich se optimalizují (specializují) jednotlivé uzly abstraktního syntaktického stromu.

Hlavním úkolem vývojáře jazyka je převést zdrojový kód *hostovaného jazyka* (jazyk hostovaný na platformě GraalVM, neboli jazyk, který vývojář implementuje) do abstraktního syntaktického stromu. O správu paměti, optimalizace a mnoho dalšího se pak stará Truffle (spolu s GraalVM) automaticky. To je žádoucí, jelikož právě ony optimalizace jsou jedna z nejnáročnějších částí vývoje jazyka. Další výhodou je fakt, že optimalizace jsou jazykově nezávislé, což znamená, že budoucí zlepšení jednotlivých optimalizací zlepšuje všechny jazyky implementované pomocí frameworku Truffle.

1.1 Motivace

V současné době existuje již několik jazyků implementovaných pomocí frameworku Truffle. Jazyky jako JavaScript, Python nebo Ruby jsou implementovány přímo společností Oracle, zatímco Pascal nebo SOM[1] jsou jazyky implementované komunitou. I přesto, že jazyky Python nebo JavaScript obsahují funkcionální prvky, v době psaní diplomové práce neexistuje čistě funkcionální jazyk implementovaný s využitím frameworku Truffle. Výjimkou by mohl být Mumbler, minimalistický dialekt jazyka Common Lisp, jehož implementace je však příliš zjednodušená a neefektivní [6]. Další nevýhodou je fakt, že implementace je 10 let stará, a tudíž v dnešní době téměř nepoužitelná jako demonstrační příklad implementace funkcionálního jazyka nad platformou GraalVM. Další implementací funkcionálního jazyka je TruffleClojure, která vznikla jako diplomová práce na Univerzitě Johanna Keplera v Linci roku 2015 [7]. Jelikož tato práce vznikla ve spolupráci se společností Oracle, která nepovolila zveřejnění zdrojového kódu, je těžké posoudit úplnost, rychlost a funkčnost interpretu. Podle informací, které mi byly poskytnuty zaměstnancem společnosti Oracle, Christianem Humerem, není zdrojový kód v publikovatelné formě a interně se na něm dále nepracuje.

V rámci této diplomové práce byl proto vyvinut interpret jazyka Scheme, na kterém je demonstrována implementace funkcionálních prvků jako je optimalizace koncové rekurze, optimalizace koncových volání nebo implementace maker. Zároveň bylo vyvinuto API pro práci s dalšími jazyky podporovanými platformou GraalVM, což umožňuje tzv. *polyglotní programování*.

1.2 Cíle a rozsah práce

Hlavním cílem diplomové práce je vytvořit interpret funkcionálního jazyka Scheme na platformě GraalVM. Vzhledem k omezené dostupnosti aktuálních zdrojů týkajících se používání frameworku Truffle se diplomová práce soustředí na detailní popis základních principů tohoto frameworku, včetně teoretických základů a praktických ukázek použití.

[1] SOM (Simple Object Machine) je minimalistický dialekt jazyka Smalltalk, který byl původně vyvinut pro účely výuky a výzkumu virtuálních strojů v rámci institutu Hasso Plattner.

1.2.1 Podpora funkcionálních prvků

Jelikož neexistuje žádná veřejně dostupná implementace jazyka, která by obsahovala funkcionální prvky, jako jsou například makra nebo optimalizace koncových volání, bylo cílem zjistit, zda a v jaké podobě je možné tyto prvky implementovat. Během implementace těchto prvků byl kladen důraz na výslednou rychlost interpretu. Například výše zmíněný interpret Mumbler již triviálně implementuje optimalizaci koncových volání, ale s jejím zavedením se rychlost výrazně snížila.

1.2.2 Rychlost

Jednou z největších předností platformy GraalVM je rychlost implementovaných jazyků. Na grafu 1 je možné vidět porovnání rychlosti jazyků na platformě GraalVM s ostatními implementacemi. Například JavaScript na platformě GraalVM je téměř stejně rychlý jako na enginu V8 [8]. Je nutné podotknout, že rychlost enginu V8 vyžadovala obrovské investice a všechny optimalizace, které překladač provádí, jsou jazykově závislé. Pokud popularita JavaScriptu upadne nebo bude nahrazen jiným jazykem (například WebAssembly je zajímavou technologií), všechny tyto investice do rychlosti nebude možné přímo použít.

Naopak platforma GraalVM provádí optimalizace nezávislé na interpretovaném jazyku. To znamená, že přidání nové či zlepšení existující optimalizace do platformy GraalVM by se mělo dotknout všech implementovaných jazyků a potenciálně je zrychlit. Důležité je zmínit, že optimalizace ve virtuálním stroji nevyžadují téměř žádnou změnu v implementaci jazyka, naopak optimalizace ve frameworku Truffle většinou jisté změny v implementaci vyžadují.

Proto byl během implementace jazyka Scheme kladen důraz na rychlost a cílem bylo zjistit, jak rychlá bude implementace v porovnání s již existujícími implementacemi. Z tohoto důvodu tato implementace nepodporuje REPL,[2] jelikož primárně slouží k rychlému prototypování, kde rychlost není důležitým faktorem.

1.2.3 Podpora primitivních procedur a speciálních forem

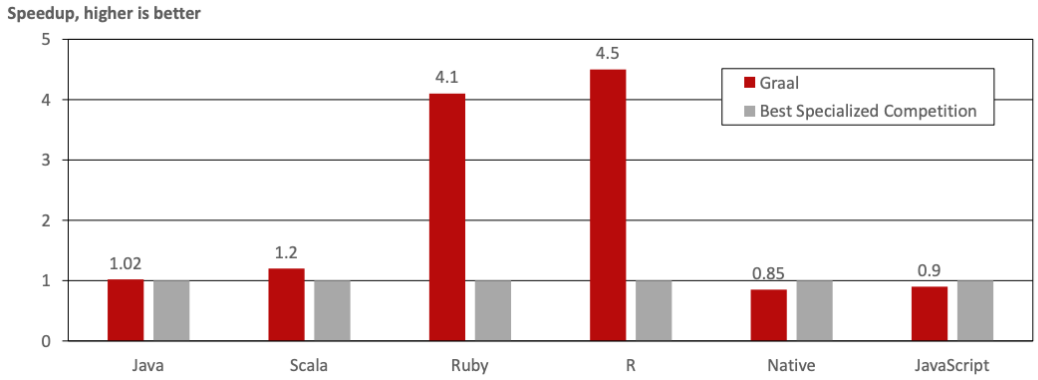
Vzhledem k rozsahu této práce nebylo možné implementovat celou specifikaci jazyka Scheme. Byly proto implementovány jen základní primitivní procedury a speciální formy, které jsou potřeba k vyhodnocení vybraných benchmarků. Podrobný seznam všech podporovaných primitivních procedur a speciálních forem je k dispozici v kapitole 4.

1.2.4 Podpora polyglotního programování

Dalším cílem této práce bylo umožnit interoperabilitu s dalšími jazyky, které jsou podporované platformou GraalVM, a tím umožnit polyglotní programování. Muselo proto být navrženo generické API, které umožní například vyhodnocovat

[2] Zkratka pro Read-Eval-Print-Loop někdy také známý jako jazyková příkazová řádka (ang. language shell). Jedná se o jednoduché prostředí pro programování, kde uživatel může napsat výraz, který je vyhodnocen a výsledek se okamžitě vrátí uživateli. [9]

kód cizích jazyků nebo číst vazby definované v globálním prostředí těchto jazyků. Jelikož interoperabilita na platformě GraalVM je stále aktivní oblastí výzkumu a velmi rychle se vyvíjí, nebyl kladen důraz na rychlost této implementace. Optimalizace rychlosti interoperability byla tedy záměrně ponechána pro potenciální budoucí zlepšení.



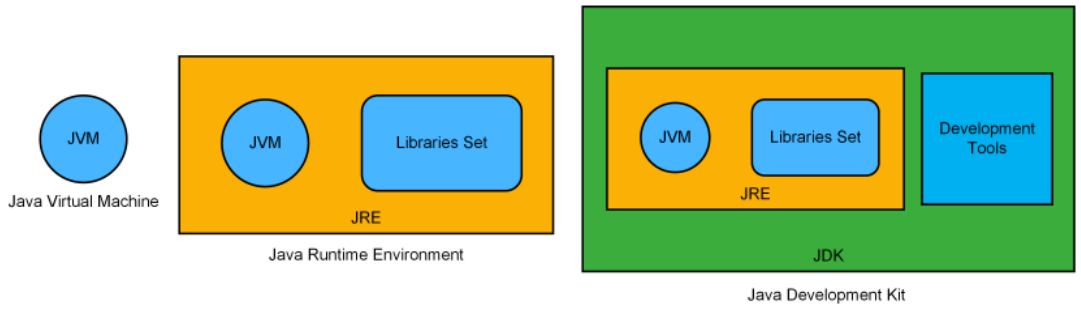
Obrázek 1: Porovnání rychlosti jazyků z roku 2017 [10]

2 Platforma Java

Platforma Java je sada softwarových nástrojů a specifikací pro vývoj a následný běh aplikací napsaných v jazyce Java.[3] Její kořeny sahají do roku 1990, kdy byla vyvinuta jako interní projekt firmy Sun Microsystems, primárně jako alternativa k jazykům C/C++ [11]. Základním stavebním kamenem platformy Java je *běhové prostředí* (ang. Java Runtime environment, dále JRE), které se skládá ze dvou hlavních částí:

- Sada knihoven pro hladký běh aplikací (např. `rt.jar`, která obsahuje implementaci Java Core API).
- Virtuální stroj Javy (ang. Java Virtual Machine, dále JVM) sloužící k interpretaci mezikódu, do kterého jsou aplikace přeloženy.

JRE obsahuje nástroje pro běh Java aplikací, avšak nezahrnuje žádné nástroje pro vývoj, jako například debugger[4] nebo překladač.[5] Tyto nástroje jsou součástí Java Development Kit (dále JDK). Z důvodu, že je JDK závislé na platformě, existují implementace pro Windows, macOS a unixové operační systémy. Schéma 2 zachycuje vztah mezi JVM, JRE a JDK.



Obrázek 2: Schéma znázorňující vztah mezi JDK, JRE a JVM

2.1 Virtuální stroj Javy

Virtuální stroj Javy je abstraktní virtuální stroj sloužící k běhu Java aplikací. Aby aplikace mohly být spuštěny nad JVM, je potřeba je zkompilovat do tzv. *mezikódu*, který na této platformě nazýváme *Java bytecode* (dále pouze bytecode). Takto zkompilovaná aplikace je uložena v `.class` souborech, které následně

[3] V počátku byla platforma Java primárně určena pro vývoj v jazyce Java. V dnešní době existuje mnoho dalších jazyků, které jsou platformou podporovány. Mezi nejznámější patří například Kotlin, Scala nebo Clojure.

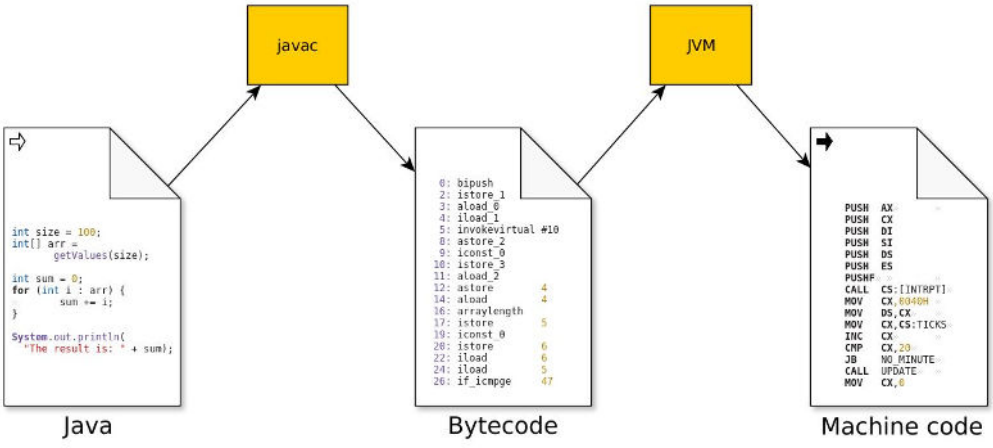
[4] Debugger je softwarový nástroj, který se používá pro hledání chyb při vývoji softwaru. [12]

[5] Například překladač *javac* není součástí JRE, který překládá jazyk Java do bytecode.

interpretuje JVM. Díky tomuto přístupu je pak každá Java aplikace platformě nezávislá a jedinou limitací je přítomnost JRE na daném prostředí.

Jednou z nejdůležitější částí JVM je *správce paměti* (ang. garbage collector), který automaticky provádí alokaci nových objektů a jejich následné uvolnění. Primárním důvodem pro automatickou správu paměti byl fakt, že téměř 70 % chyb v jazycích C/C++ bylo způsobeno chybnou nebo neoptimální prací s pamětí. Existuje hned několik správců paměti (závisí na implementaci JDK), z nichž každý je vhodný pro jiný účel (např. podle velikosti haldy). Správce paměti je pak spuštěn v samostatném vlákně (nebo ve více vláknech), což znamená, že běží paralelně s aplikací. Operace prováděné správcem paměti nejsou nezávislé na běhu aplikace. Například při defragmentaci haldy je nutné uzamknout některé objekty, což může ovlivnit výkon aplikace i na procesorech s více jádry. [13]

Další významnou součástí JVM je *interpret*, jehož hlavním úkolem je interpretovat instrukce bytecode. Jednou z hlavních nevýhod interpretovaných jazyků je jejich nižší rychlost oproti jazykům překládaným přímo do strojového kódu (často až desetkrát pomalejší). Tento problém byl v pozdějších verzích Javy vyřešen využitím tzv. *Just-in-time kompilace* (dále JIT kompilace). Během JIT kompilace jsou části bytecode, jako jsou funkce a bloky kódu, zkompilovány do strojového kódu a uloženy do tzv. *code cache*. Takto zkompilovaný kód již není interpretován, nýbrž volán přímo z code cache. Na obrázku 3 je ilustrován postup překladu jazyka Java nejprve do bytecode a poté do strojového kódu pomocí JIT kompilace. V následující kapitole je detailně popsána JIT kompilace, včetně optimalizací, které jsou v jejím průběhu aplikovány.



Obrázek 3: Příklad překladu jazyka Java do bytecode a následně do strojového kódu [14]

2.2 Just-in-time kompilace

Just-in-time kompilace je technika, která kombinuje interpretaci a kompilaci kódu za účelem zvýšení výkonu aplikace. Program je nejprve interpretován a později překládán do strojového kódu, který je uložen do code cache, odkud je volán. Jelikož code cache má omezenou velikost, nelze takto přeložit celý program. Překládají se pouze části kódu, tzv. *hotspots*,[6] které se často volají.

Interpret při běhu programu sbírá (profiluje) data, která pak využívá pro efektivnější generování strojového kódu. Právě tato profilující data chybějí jazykům, které jsou přímo překládány do strojového kódu, a tudíž nedokážou generovat tak efektivní strojový kód. Optimalizace jsou pak často založeny na spekulacích (ang. *assumptions*), kde program a jeho funkce jsou volány se specifickou konfigurací nebo s konkrétními datovými typy. V současných implementacích JVM jsou k dispozici dva typy JIT překladačů, a to *klient* a *server*. Oba překladače jsou řízeny daty, které při běhu aplikace interpret získal. To znamená, že bytecode je na začátku relativně pomalu interpretován a až ve chvíli, kdy volání funkce nebo bloku kódu (například ve smyčce) překročí určitý práh, je kód přeložen. Mezi nejznámější prahy patří:

- **Počet volání jednotlivých metod** (ang. *invocation count threshold*). Tento práh se používá pro rozhodnutí, zda metoda bude přeložena klientským nebo serverovým překladačem.
- **Počet volání těla smyčky** (ang. *backedge counter treshold*). Tento práh se používá pro rozhodnutí, zda tělo smyčky bude přeloženo klientským nebo serverovým překladačem.

2.2.1 Klientský překladač

Klientský překladač (v implementaci Hotspot JVM nazývaný jako C1 překladač) se vyznačuje tím, že překlad provádí velice rychle a k přeložení bytecode nepotřebuje mnoho dat z běhu aplikace. Z těchto vlastností vyplývá, že se primárně využívá po startu aplikace ke zvýšení její rychlosti, jelikož při překladu provádí optimalizace daleko jednodušší a méně kvalitní v porovnání se serverovým překladačem. Historicky se tento typ překladače používal pro aplikace, které běžely jen krátce a mezi nefunkční požadavky patřil rychlý start.

2.2.2 Serverový překladač

Serverový překladač (v implementaci Hotspot JVM nazývaný jako C2 překladač) provádí optimalizace daleko komplexnější a často založené na spekulacích. Ty jsou nejen časově náročné, ale také je pro jejich realizaci potřeba velké množství dat z běhu aplikace. Jedná se o paradoxní situaci, kdy se pro dlouhotrvající aplikace vyplatí nejprve funkce nebo bloky kódu pomalu interpretovat a jakmile

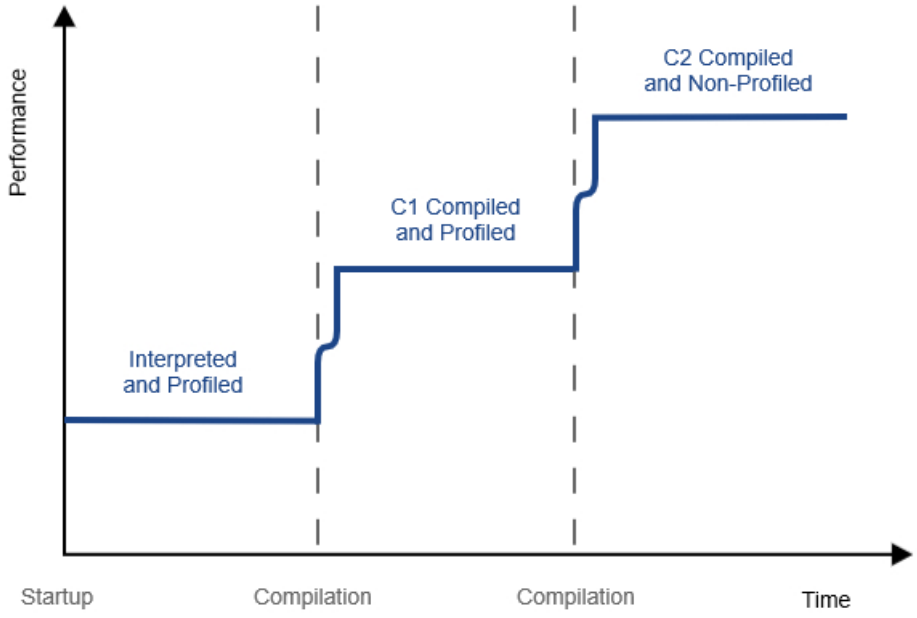
[6] Jedna z nejznámějších implementací JVM, známá jako Hotspot JVM, odtud dostala své jméno.

jsou k dispozici data potřebná k překladu, je tento kód přeložen do velice efektivního strojového kódu. Jak název napovídá, tento překladač se používal pro dlouhotrvající aplikace, primárně pro serverové aplikace. [13]

Optimalizace založené na spekulacích se mohou stát nevalidní v případě, že některá ze spekulací již není pravdivá. V takovém případě dochází k tzv. *deoptimalizaci*, při které je strojový kód zahozen a původní kód je opět interpretován. JVM posléze může provést opětovnou optimalizaci s novými informacemi, které vedly k předešlé deoptimalizaci. Tímto je zamezeno častým deoptimalizacím, které negativně ovlivňují výkon aplikace.

2.2.3 Vícevrstvá kompilace

Do Javy verze 7 bylo nutné zvolit, jaký typ překladače bude aplikace využívat. Java 7 představila tzv. *vícevrstvou kompilaci* (ang. tiered compilation), která využívá klientský překladač k rychlému startu aplikace a serverový překladač k dosažení dlouhodobě vyššího výkonu aplikace.



Obrázek 4: Růst výkonu s využitím vícevrstvé kompilace [15]

Obecný princip vícevrstvého překladu spočívá v tom, že při startu aplikace je všechen bytecode interpretován a profilující data jsou použita k nalezení často volaných funkcí/bloků kódu. Tyto často volané funkce/bloky pak JVM přeloží do strojového kódu pomocí klientského překladače, a tím se přiblíží rychlosti kompilovaných jazyků.[7] Jakmile má JVM dostatek dat z běhu aplikace, přichází

[7]Rychlost bude pomalejší, protože některé části kódu jsou stále interpretovány.

na řadu serverový překladač, který provede komplexnější a déletrvající optimalizace a kód následně přeloží do strojového kódu. Takto optimalizovaný kód je pak mnohem efektivnější než strojový kód kompilovaných jazyků (C/C++).[8] Obrázek 4 znázorňuje, jak výkon aplikace roste během jednotlivých překladů. Nutno poznamenat, že profilování se již neprovádí pouze nad interpretovaným bytecode, ale i nad strojovým kódem přeloženým klientským překladačem. Nedochází tak již k výše zmíněné paradoxní situaci. [15]

2.2.4 Optimalizace realizované JIT překladači

Cílem této kapitoly je seznámení s nejčastějšími optimalizacemi prováděnými klientským nebo serverovým překladačem (nebo oběma). Nejprve jsou popsány obecné oblasti, na které se JIT překladače zaměřují, a poté jsou popsány konkrétní optimalizační metody. Kapitola vychází ze zdroje [16].

1. **Přístup do paměti**: Operace s pamětí jsou obecně velice pomalé. Například čtení z operační paměti trvá přibližně 50-100ns a čtení z registrů trvá přibližně 300ps [17]. Vzhledem k tomu, že přístup k registrům je o tři řády rychlejší, alokace proměnných v registrech namísto operační paměti bude klíčová pro celkový výkon.

2. **Volání metod**: JIT překladače využívají data z běhu programu k hledání často volaných metod. Volání takové metody je následně nahrazeno tělem dané metody, a tím je eliminována režie spojená s jejím voláním. Tato optimalizace se nazývaná *eliminace volání metod* (ang. method inlining).

3. **Eliminace zámků**: JIT překladače se s pomocí technik jako *escape analysis* snaží určit, zda reference lokálních objektů neunikají mimo aktuální metodu nebo kontext. V takovém případě se pokusí zámky buďto úplně eliminovat, nebo alespoň částečně omezit rozsah jejich platnosti.

4. **Eliminace podmíněných skoků**: JIT překladače se snaží co nejvíce eliminovat podmíněné či nepodmíněné skoky, popřípadě kód přeskládat tak, aby prediktory skoků byly schopné odhadnout, zda se skok provede.

2.2.4.1 Alokace registrů

Alokace registrů (ang. register allocation) se primárně pokouší minimalizovat přístup do operační paměti, neboť čtení z operační paměti je výrazně pomalejší. Tento typ optimalizace implementuje jak klientský, tak i serverový JIT překladač, avšak každý využívá odlišné algoritmy. Optimální přiřazení registrů je NP-úplný problém, což znamená, že časová složitost algoritmu je exponenciální.

[8] Za předpokladu, že nedochází k častým deoptimalizacím.

Většina procesorů využívá architekturu založenou na registrech, a nikoliv na zásobníku operandů (tu právě využívá JVM při interpretaci bytecode). To znamená, že JIT překladač při překladu musí vhodně transformovat zásobníkové operandy do registrů dané procesorové architektury. Takových architektur je hned několik, například x86_64, i686 nebo v dnešní době čím dál populárnější ARM. To znamená, že pro každou architekturu je nutné JIT překladače do jisté míry upravit, jelikož počet dostupných registrů a jejich využití se na každé architektuře liší.

Klientský překladač, který je zaměřen více na rychlost překladu než na kvalitu výsledného strojového kódu, používá algoritmus nazývaný *lineární alokace registrů*. Algoritmus nejprve zjistí oblasti života jednotlivých proměnných a následně se jim pokusí přiřadit registry. Pokud nastane situace, že není žádný volný registr k dispozici, pak proměnou uloží na haldu (dochází k tzv. *variable spilling*). Tento přístup je velice rychlý, ale nejedná o optimální řešení, a tudíž negeneruje optimální strojový kód.

Problém přiřazení registrů lze převést na problém barvení grafu, což je algoritmus, který využívá JIT překladač typu server. Jelikož barvení grafu je opět NP-úplný problém, je tento problém řešen heuristickými (přibližnými), časově přijatelnými metodami, jejichž výsledek opět není optimální ve všech případech.

2.2.4.2 Rozbalení programových smyček

Jednou z hlavních optimalizací, které eliminují podmíněné či nepodmíněné skoky, je *rozbalení programových smyček* (ang. loop unrolling). Klíčovým důvodem, proč je tato optimalizace tak důležitá, je tzv. *pipeline architektura*, kterou téměř všechny procesory v dnešní době používají. Pokud se narazí na instrukci skoku, musí se nejen vyřešit problém s rozpracovanými instrukcemi, ale i problém s prediktivním bufferem, který se pokouší dopředu „uhodnout“, jaké instrukce budou následovat. Velmi často tento problém vyžaduje, aby všechny rozpracované instrukce byly zahozeny a pipeline se naplnila znovu. To může mít drastický dopad na výkon, kdy například Pentium 4 má pipeline rozdělenou na 30 řezů, což znamená, že se 30 instrukcí provádí paralelně a všechny tyto instrukce se musí zahodit. Z tohoto důvodu je vhodné mít strojový kód s co nejmenším počtem skoků, tzn. mít *základní blok programu*[9] co nejdelší.

JIT překladač poté využívá heuristik a dat z běhu programu k rozhodnutí, zda je vhodné tuto optimalizaci provést. Rozbalení každé smyčky by mohlo způsobit dramatické zvětšení velikosti výsledného strojového kódu a potenciálně by se mohl zvýšit počet tzv. *cache misses*.

2.2.4.3 Výpočet konstantních výrazů a jejich propagace

Výpočet konstantních výrazů a jejich propagace (ang. Constant Folding and Constant Propagation) je jednou z nejpoužívanějších optimalizačních technik. Lze ji

[9] Základní blok programu je část programu, ve které není žádná instrukce větvení (např. podmíněný či nepodmíněný skok, volání podprogramu nebo synchronní přerušení). [18]

aplikovat i na jazyky, které jsou přímo překládány do jazyka stroje. Operace, u kterých jsou hodnoty operandů konstantní (nemění se při běhu programu), může překladač přímo vypočítat a nemusí tak pro daný výpočet generovat strojový kód. Výsledky pak mohou být dále šířeny a může docházet buďto k dalším výpočtům konstantních výrazů anebo k *eliminaci nedosažitelného či nadbytečného kódu* (ang. dead-code elimination). Odstranění nedosažitelného kódu často vzniká při větvení, kde je podmínka konstantní, a tudíž větvení může být odstraněno. Příklad výpočtu konstantních výrazů a jejich propagace je vidět ve zdrojovém kódu 1.

```
1  // původní blok kódu před optimalizací
2  int x = 14;
3  int y = 7 - x / 2;
4  return y * (28 / x + 2);
5
6  // po propagaci proměnné x
7  int x = 14;
8  int y = 7 - 14 / 2;
9  return y * (28 / 14 + 2);
10
11 // po výpočtu konstantních výrazů a následné propagaci proměnné y
12 int x = 14;
13 int y = 0;
14 return 0;
15
16 // po odstranění zbytečného/nedosažitelného kódu
17 return 0;
```

Zdrojový kód 1: Výpočet konstantních výrazů a jejich propagace

2.2.5 Způsoby JIT kompilace

Drtivá většina JIT překladačů je jazykově závislá, což znamená, že všechny optimalizace prováděny těmito překladači je možné dělat pouze pro jeden konkrétní jazyk nebo pro jednu konkrétní rodinu jazyků. Výborným příkladem je JavaScript, jehož výkon byl v posledním desetiletí zlepšen o několik řádů. Nutno podotknout, že toto zlepšení vyžadovalo obrovské investice, které jsou bohužel zřídka opodstatněné u výzkumných projektů nebo u doménově specifických jazyků (DSL) navržených k řešení specifických problémů. Existují ale i techniky, které umožňují provádět jazykově nezávislé optimalizace. Jednou z nich je *meta-kompilace* (ang. meta-compilation), kterou se tato kapitola bude zabývat.

Definice 1 (Meta-kompilace)
Meta-kompilace je výpočet, který na základě meta-systémových přechodů (ang. metasystem transitions) z výpočetního stroje M vytvoří výpočetní stroj M', který je sémanticky totožný se strojem M [19].

Jak bude možné vidět v následujících kapitolách, meta-kompilace je pouhá transformace, která využívá data z běhu programu a program samotný k vytvoření nového kódu, který bude mít stejnou sémantiku jako kód původní. Meta-systémové přechody z definice 1 odpovídají datům z běhu programu, tzn. transformace je řízena těmito daty. Je důležité podotknout, že pokud každý program bude reprezentován stejnou strukturou (např. pomocí abstraktních syntaktických stromů), je tato transformace jazykově nezávislá. Pokud je pojem transformace nahrazen pojmem optimalizace, což je možné, jelikož jakákoliv optimalizace je pouhou transformací kódu, dostáváme pak jazykově nezávislé optimalizace. V následujících kapitolách jsou popsány dvě nejznámější techniky meta-kompilace *meta-tracing* a *partial evaluation*.

2.2.5.1 Meta-tracing

Meta-tracing je technika meta-kompilace, jejíž základní princip spočívá ve sledování částí kódu, které byly navštíveny během vykonání programu, a jejich následné optimalizace. Jelikož sledování kódu má vliv na výkon aplikace, nesleduje se celý program, ale pouze funkce a cykly, které přesáhly práh pro JIT kompilaci. Výsledkem sledování je *záznam* (ang. trace), který je optimalizován[10] a poté přeložen do strojového kódu. Záznamy obsahují specifické *kontroly* (ang. guards), které kontrolují, zda spekulace, které byly provedeny během optimalizací, jsou stále validní. Pokud ne, dochází ke klasické deoptimalizaci, strojový kód je zahozen a kód je interpretován. Nutno zmínit, že meta-tracing je obecná technika využívána v různých odvětví informatiky, a proto meta-tracing popsán v této kapitole primárně vychází z implementace RPython, což je Python framework pro psaní interpretů.

User program	Trace when x is set to 6	Optimised trace

`if x < 0:`
`  x = x + 1`
`else:`
`  x = x + 2`
`x = x + 3`
	`guard_type(x, int)`
	`guard_not_less_than(x, 0)`
	`guard_type(x, int)`
	`x = int_add(x, 2)`
	`guard_type(x, int)`
	`x = int_add(x, 3)`
		`guard_type(x, int)`
		`guard_not_less_than(x, 0)`
		`x = int_add(x, 5)`

Obrázek 5: Příklad uživatelského programu (levý sloupec), jeho neoptimalizovaný záznam (sloupec uprostřed) a optimalizovaný záznam (sloupec vpravo) [20]

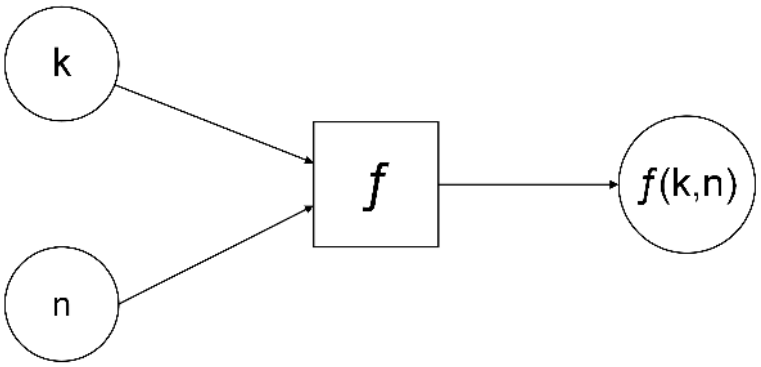
Na obrázku 5 je možné vidět uživatelský program a jeho záznamy. V levém sloupci je uživatelský program v jazyce Python. Pokud tento kód bude vyhodnocen jako vhodný kandidát pro JIT kompilaci (počet volání tohoto kódu překročí určitý práh), v příštím vykonání bude kód sledován a vytvoří se záznam. Sloupec uprostřed je ukázka takového záznamu, kde při vykonávání proměnná x

[10]Optimalizace jsou opět založeny na spekulacích, které vychází z dat sesbíraných během tohoto sledování.

nabývala hodnoty `6`. Je důležité zdůraznit, že hodnota proměnné `x` není uložena v záznamu. Stejný záznam by byl vytvořen pro jakoukoliv kladnou hodnotu `x`. Jakmile je záznam vytvořen, přichází na řadu jeho optimalizace, jejímž cílem je snížit velikost záznamu a vytvořit co nejefektivnější strojový kód. V tomto případě dvě kontroly typu proměnné `x` mohou být odstraněny (jelikož typ byl ověřen hned na začátku záznamu) a poté dvě konstantní operace sčítání mohou být nahrazeny pouze jednou operací sčítání. V pravém sloupci jsou výsledné optimalizované záznamy, které budou převedeny do strojového kódu. [20]

2.2.5.2 Partial evaluation

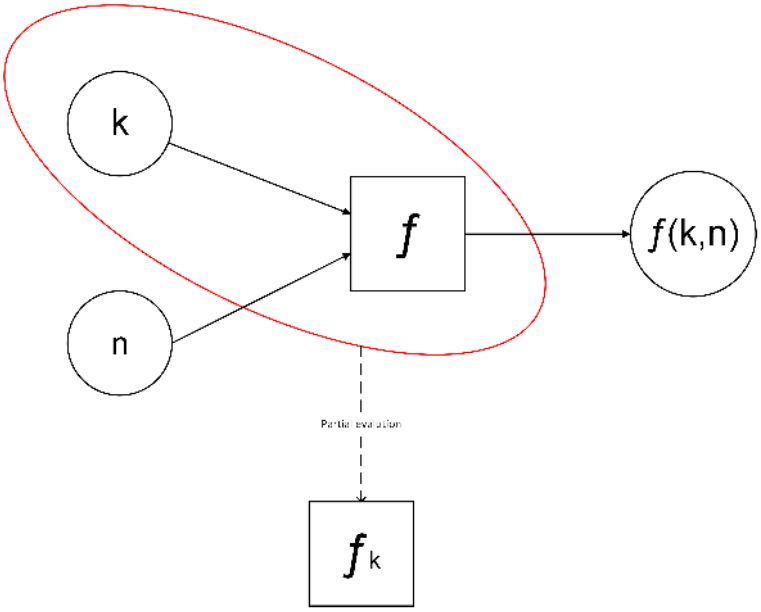
Partial evaluation je další technikou meta-kompilace, jejímž hlavním cílem je, stejně jako u meta-tracing, transformovat kód do menší a jednodušší podoby tak, aby výsledný strojový kód byl co nejefektivnější. Základním principem této techniky je sběr dat z běhu aplikace, která jsou použita k vytvoření efektivnějšího kódu. Tato technika vychází z předpokladu, že program se po určité době stane stabilním, a bude ho tudíž možné transformovat do efektivnější podoby. Tímto je možné například odstranit části původního kódu, které nejsou programem používány, nebo spekulovat nad typem argumentů u jednotlivých funkcí. Opět je nutno zmínit, že partial evaluation je obecná technika, a proto tato kapitola vychází z frameworku Truffle, což je Java framework pro psaní interpretů pro libovolné jazyky nad GraalVM.



Obrázek 6: Schéma funkce s dvěma argumenty n a k

Schéma 6 znázorňuje funkci f s dvěma argumenty n a k. Jak již bylo zmíněno, hlavním cílem partial evalution je transformovat funkci f na kratší a efektivnější funkci s využitím dat z běhu aplikace. Řekněme, že funkce f byla vždy volána se stejnou hodnotou argumentu k. Jakmile JVM vyhodnotí, že funkce f je vhodným kandidátem na JIT kompilaci, partial evaluation nejprve funkci transformuje, a poté se transformovaný kód přeloží do strojového kódu. Schéma této transformace je znázorněno na obrázku 7. Výsledkem je funkce f_k, která je sémanticky totožná s funkcí f. Platí proto následující vztah:

$$f(k, n) = f_k(n)$$



Obrázek 7: Aplikování partial evaluation na funkci f za předpokladu, že hodnota argumentu k je konstantní

Zdrojový kód 2 tuto transformaci zobrazuje na konkrétní funkci `pow`. V tomto příkladu je předpokladem, že funkce byla často volána s hodnotou argumentu `k = 5`. Proces partial evaluation je ve zdrojovém kódu zobrazen tak, že se postupně tvoří nové funkce `pow5`, což se ve skutečnosti samozřejmě neděje. Jak je možné vidět ze zdrojového kódu, partial evaluation s využitím dat z běhu (v tomto případě je známa hodnota parametru k) transformuje kód pomocí již výše zmíněných optimalizací. Nejprve bylo eliminováno volání metod a poté proběhlo odstranění nadbytečného kódu. V případě, že spekulace se stane nevalidní, což znamená, že hodnota argumentu k je jiná než 5, dojde k deoptimalizaci a funkce `pow` bude následně opět interpretována.

Závěrem je nutno zmínit, že tento příklad je pouze ilustrativní a reálně by tato metoda nebyla takto transformována z důvodu, že spekulace nad hodnotou argumentu k je moc silná. Cílem tohoto příkladu bylo demonstrovat, že partial evaluation kód transformuje pomocí dat z běhu programu a tvoří tak kratší a efektivnější kód.

```
1  // před transformací
2  public long pow(int n, int k) {
3      if (k <= 0) {
4          return 1;
5      } else {
6          return n * pow(n, k - 1);
7      }
8  }
9
10 // eliminace if podmínky a výpočet konstantních výrazů
11 public long pow5(int n) {
12     return n * pow(n, 4);
13 }
14
15 // eliminace funkce pow(n, 4) - vložení těla
16 public long pow5(int n) {
17     return n * n * pow(n, 3);
18 }
19
20 ....
21
22 // výsledný kód po eliminaci funkcí pow(n, 3), pow(n, 2), pow(n, 1)
       a pow(n, 0)
23 public long pow5(int n) {
24     return n * n * n * n * n * 1;
25 }
26
27 // odstranění nadbytečného kódu
28 public long pow5(int n) {
29     return n * n * n * n * n;
30 }
```

Zdrojový kód 2: Partial evalution aplikovaná na funkci `pow`

2.2.5.3 Meta-kompilace při psaní interpretů

Předchozí dvě kapitoly se zabývaly meta-tracing a partial evaluation jako obecnými technikami pro transformaci kódu. V této kapitole je popsáno, jak jsou tyto techniky aplikovány při psaní interpretů. Na obrázku 8 můžeme vidět klasické schéma interpretu. Interpret je program se dvěma vstupy:

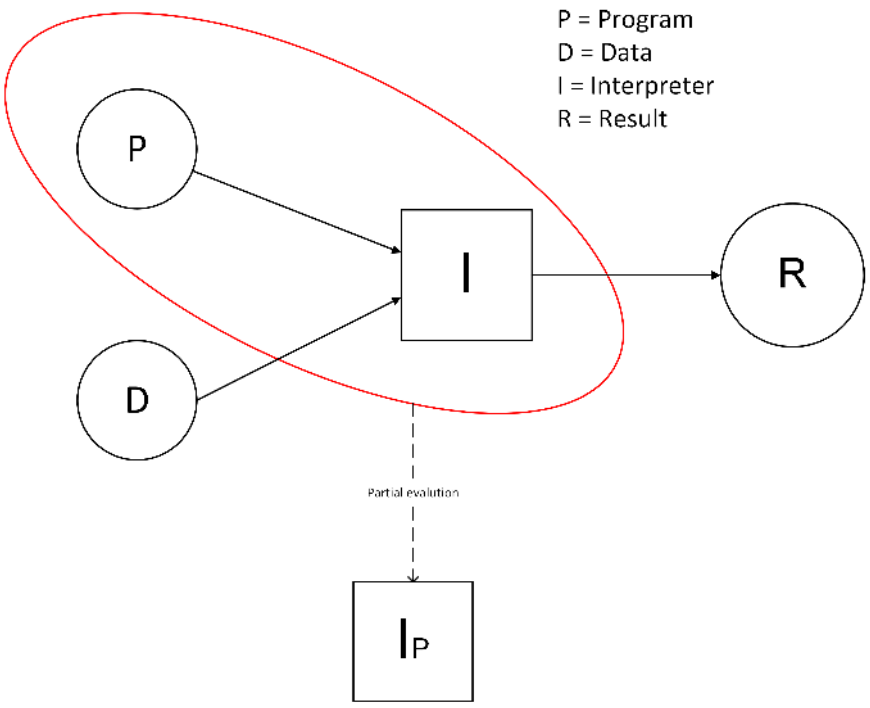
- Program, který se interpretuje.
- Vstupní data interpretovaného programu.

Pokud je aplikována jedna z meta-kompilačních technik na program a interpret (znázorněno červeným kruhem na obrázku 8), vzniká nový interpret, který je přímo optimalizovaný pro vstupní program. Tato transformace se provádí za běhu programu, a proto jsou dostupná i data z běhu programu, která umožňují aplikovat optimalizace založené na spekulacích. Jelikož jsou během transformace aplikovány spekulace, může dojít k situaci, kdy se spekulace stane nevalidní. V takovém případě dochází k deoptimalizaci a kód je interpretován. Meta-kompilační techniky tedy nezahrnují celou sémantiku daného kódu, ale pouze sémantiku, kterou interpretovaný program opravdu využívá. Výpočetní stroj *M'* zmíněný v definici 1 neobsahuje celou sémantiku stroje *M*, ale obsahuje mechanismus (deoptimalizace a návrat k interpretaci), kterým vždy celou sémantiku může pokrýt. Transformace, kdy je aplikována jedna z meta-kompilačních technik na program a interpret, se nazývá *První Futamurova projekce*, na které, jak uvidíme v dalších kapitolách, je založen framework Truffle.

2.2.5.4 Rozdíl mezi meta-tracing a partial evalution

I přesto, že obě techniky se snaží dosáhnout stejného cíle, a to vytvořit krátký a efektivní kód, každá z nich na to jde z opačného konce. Partial evaluation nejprve nashromáždí co nejvíce dat z běhu programu, která posléze aplikuje na program a interpret, a tím vytvoří interpret optimalizovaný pro daný program. Klíčové je zde podotknout, že k tomu, aby tato transformace mohla proběhnout, je nutné, aby daný kód byl několikrát vyhodnocen. Sběr dat z běhu programu je o to důležitější při implementaci dynamických jazyků, jelikož množství informací, které lze získat ze statické analýzy těchto jazyků, je minimální.

Naopak meta-tracing potřebuje pro optimalizaci interpretu pouhé jedno vykonání, během kterého jsou nashromážděny všechny informace o dané *vykonané cestě* (ang. execution path). To teoreticky znamená, že právě po jednom vykonání má meta-tracing dostatek informací k optimalizaci této cesty. Jelikož tato optimalizace má výrazný dopad na výkon aplikace, nelze takto optimalizovat každou cestu. Místo toho se meta-tracing snaží vynechat všechny informace, které by tuto cestu činily příliš specifickou. To již bylo možné vidět v kapitole 2.2.5.1 na obrázku 5, kde se do záznamu neuložila kontrola hodnoty argumentu `x`, ale pouze kontrola, zda je hodnota kladná. Tímto se od sebe jednotlivé techniky



Obrázek 8: První Futamorova projekce

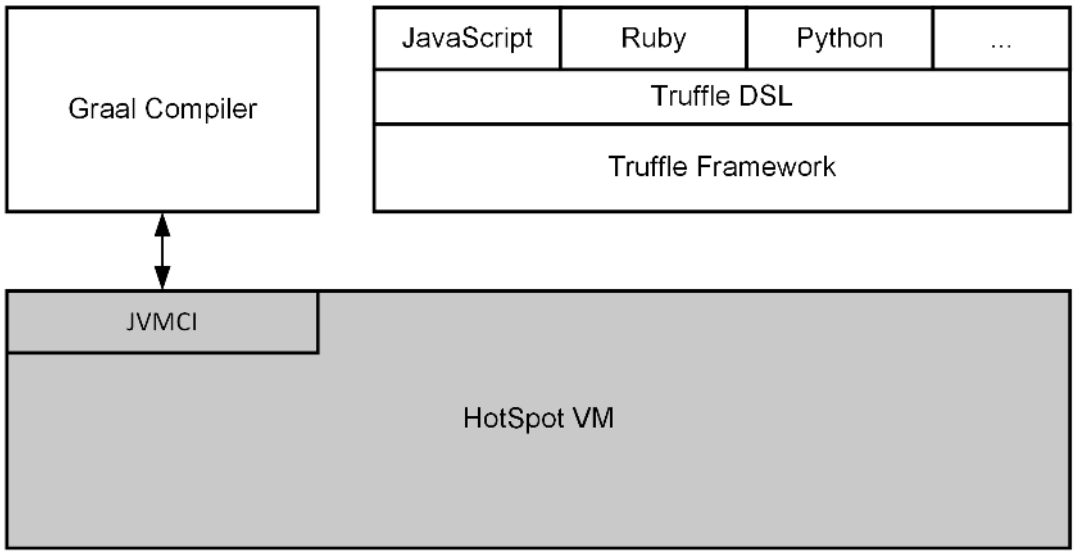
nejvíce liší. Zatímco partial evaluation musí vyhodnotit daný blok kódu několikrát k získání všech potřebných informací, meta-tracing má všechny potřebné informace již po prvním průchodu a využívá heuristik k vynechaní některých informací, které by činily optimalizaci příliš specifickou.

Dalším významným rozdílem jsou části kódu, na které se jednotlivé techniky zaměřují. Partial evaluation se zaměřuje na často volané funkce, na které se následně aplikují optimalizace k vytvoření co největšího kompilačního celku. To znamená, že v optimalizované funkci dochází k eliminaci volání vnitřních funkcí (tzn. volání funkce je nahrazeno jejím tělem), k rozbalování programových smyček nebo k výpočtu konstantních výrazů. Takto je vytvořen co největší kompilační celek, který je následně přeložen do strojového kódu. Naopak meta-tracing se výhradně zaměřuje na často volané smyčky, jejichž těla jsou obdobně optimalizována a opět přeložena do strojového kódu.

3 Platforma GraalVM

GraalVM je virtuální stroj vyvíjený společností Oracle. Jedná se o modifikaci HotSpotVM, kde serverový C2 překladač byl nahrazen *Graal* překladačem. Nahrazení dynamického překladače je možné díky *Java Virtual Machine Compiler Interface* (JVMCI), což je rozhraní, které je k dispozici od Java verze 9 a umožňuje integrovat nové dynamické překladače psané v Javě do JVM.

Překladač Graal jako vstup přijímá Java bytecode a transformuje ho do interní reprezentace zvané Graal IR (ang. Graal Intermediate Representation). Graal IR je založen na orientované grafové struktuře, která modeluje jak datový tok, tak i kontrolní tok programu. Zároveň je i v tzv. *static single assignment form*.[11] Nad touto datovou strukturou jsou následně prováděny veškeré optimalizace. Jakmile jsou všechny aplikovány, je interní reprezentace přeložena do strojového kódu. Ostatní části GraalVM, jako například správa paměti, jsou pak totožné s HotspotVM. Architektura GraalVM je znázorněna na obrázku 9. [22]



Obrázek 9: Schéma architektury GraalVM

3.1 Framework Truffle

Truffle je framework používaný pro implementaci nových jazyků nad platformou GraalVM. Jedná se o sadu tříd, rozhraní a anotací, které vývojář při implementaci nového jazyka používá. Rychlost jazyků implementovaných nad GraalVM je pak téměř stejná (nebo lepší) jako v případě interpretovaných jazyků, kterým

[11] Static single assignment form je vlastnost interní reprezentace, kde každá proměnná je deklarována před jejím použitím a zároveň je deklarována pouze jednou. Nad interní reprezentací, která splňuje tuto vlastnost je výrazně jednoduší provádět optimalizace. [21]

byl vytvořen virtuální stroj na míru. Obecně existují dva způsoby, jakými lze implementovat nový jazyk:

1. Napsat interpret, který je relativně jednoduchý na implementaci, ale výrazně pomalejší.

2. Napsat překladač, který je rychlý, ale mnohonásobně náročnější na implementaci (hlavně při implementaci sofistikovaných optimalizací).

Framework Truffle kombinuje oba výše zmíněné způsoby a bere si z obou to nejlepší. Programy v jazycích implementovaných ve frameworku Truffle (dále hostované jazyky) jsou reprezentovány jako abstraktní syntaktické stromy (AST), které jsou následně interpretovány. Během této interpretace Truffle sbírá profilující data a ukládá je do uzlů abstraktního syntaktického stromu. V případě, že některá z metod hostovaného jazyka je často volána, Truffle zahájí její JIT překlad. Je důležité poznamenat, že sám framework Truffle určuje, kdy a jaké části programu budou překládány. Dochází tak ke dvěma rozdílným JIT překladům. První překlad je řízen frameworkem Truffle a druhý je klasický JIT překlad řízený JVM (interpret je stále pouhá Java aplikace).

Hlavním úkolem vývojáře, který se rozhodne implementovat jazyk nad platformou GraalVM, je převést zdrojový kód hostovaného jazyka do abstraktních syntaktických stromů. O vše ostatní, ať už se jedná o optimalizace nebo generování strojového kódu, se stará framework Truffle. Tento přístup přináší dvě hlavní výhody:

1. Rychlost výzkumných nebo DSL jazyků bude velice podobná rychlosti jejich kompilovaných nebo interpretovaných protějšků (v případě, že by vůbec vznikly). Úsilí vyložené na implementaci těchto jazyků nad GraalVM by bylo ale výrazně menší.

2. Budoucí vylepšení překladače Graal nebo frameworku Truffle přinese vylepšení do všech jazyků, které jsou nad platformou implementovány bez jakékoliv změny kódu v implementaci jazyka.

3.2 Sebeoptimalizace AST interpretů

Při psaní interpretů je běžnou praktikou transformovat zdrojový kód do abstraktního syntaktického stromu (dále AST), který je následně procházen a vyhodnocen. AST interpret vyhodnocuje strom *post-order*, tzn. nejprve jsou vyhodnoceni potomci uzlu a poté uzel samotný. To je intuitivní, jelikož pokud by daný AST strom reprezentoval například funkci sčítání, tak je potomky možné chápat jako argumenty, které musí být vyhodnoceny před vyhodnocením samotného uzlu reprezentujícího sčítání.

Truffle koncept AST interpretace rozšířil o tzv. *sebeoptimalizaci*. Sebeoptimalizací je myšleno, že uzel AST stromu se buďto může specializovat na základě vstupních argumentů, a pokrývat tak pouze část původní sémantiky, nebo může

nahradit sám sebe za nový uzel, který bude optimalizován na základě profilujících dat. Pokud specializovaný uzel není schopný provést operaci s aktuálními typy argumentů, dochází k nové specializaci na základě těchto typů. Tuto opětovnou specializaci lze chápat jako postupné rozšiřování sémantiky. Na začátku specializované uzly pokrývají minimální podmnožinu sémantiky operace a v případě, že stávající uzel nedokáže provést operaci nad aktuálními operandy, je uzel znovu specializován, a tím je sémantika operace rozšířena. Tyto uzly musí splňovat následující podmínky:

1. **Úplnost** – přestože specializovaný uzel pokrývá pouze podmnožinu sémantiky, musí být schopen se znovu specializovat a pokrývat tak zbytek sémantiky operace.

2. **Konečnost** – po konečném počtu opakovaných specializací musí být uzel v generickém stavu (pokrývat celou sémantiku operace). To znamená, že tento uzel již nebude nikdy specializován.

3.3 Truffle API

Truffle API je knihovna určená pro vývojáře jazyků k tvorbě sebeoptimalizujících AST interpretů. Pomocí rozhraní, abstraktních tříd a anotací jsou postupně tvořeny abstraktní syntaktické stromy Truffle, které reprezentují program hostovaného jazyka. Základními prvky knihovny, kterým se bude věnovat tato kapitola, jsou `Node`, `RootNode`, `CallTarget` a `VirtualFrame`.

3.3.1 Node

`Node` je abstraktní třída, ze které musí dědit všechny uzly abstraktních syntaktických stromů Truffle. Na první pohled může být překvapující, že tato abstraktní třída nemá žádné abstraktní metody, i přesto, že uzly jsou následně vyhodnocovány. Naopak vývojář hostovaného jazyka je zodpovědný za vytvoření těchto metod. Primárním důvodem je fakt, že každý uzel může vracet jiný datový typ, tudíž nelze dopředu definovat dostatečně generickou abstraktní metodu. `Node` obsahuje spoustu pomocných funkcí, které slouží k modifikaci abstraktního syntaktického stromu Truffle. Například metody jako `insert` nebo `replace` slouží k vložení či nahrazení potomka. V kapitole 5.7 je pak možné se dočíst, jak je metoda `replace` použita k expanzi maker.

Každá třída, která není abstraktní a dědí z `Node` musí definovat metodu, jejíž jméno musí začínat prefixem `execute`. Potomka tohoto uzlu je pak možné definovat pomocí anotace `@Child`. V případě, že potomků je více, existuje anotace `@Children`, která slouží k definování pole potomků. Potomci nemohou být označeni klíčovým slovem `final`, jelikož může dojít k nahrazení potomka specializovanějším (např. pomoci výše zmíněné metody `replace`). Důležité je zmínit, že všechny vlastnosti třídy s anotací `@Child` jsou překladačem Graal vnímány jako konstanty. Vlastnosti označené `@Children` mají dokonce silnější

sémantiku než `final`, a to takovou, že nejen pole samotné je vnímáno jako konstanta, ale i jeho prvky jsou překladačem Graal vnímané jako konstanty.

Pro úplnost je vhodné zmínit anotaci `@CompilationFinal`. Jak je možné odvodit z názvu, jedná se o anotaci, která má stejnou sémantiku jako klíčové slovo `final`. Vlastnosti, které neobsahují anotaci `@Child`, ale stejně je důležité, aby byly překladačem Graal vnímané jako konstantní, obsahují právě tuto anotaci. Anotace dokonce obsahuje vlastnost `dimensions`, která se používá na vlastnosti typu pole. Definuje, zda i prvky pole mají být vnímány jako konstanty. Ve skutečnosti anotace `@Child` sémanticky odpovídá anotaci `@CompilationFinal` a anotace `@Children` sémanticky odpovídá anotaci `@CompilationFinal(dimensions = 1)`.

Ve zdrojovém kódu 3 je pak možné vidět jednoduchou implementaci uzlu pro sčítání. Uzel dědí z abstraktní třídy `SchemeNode`, která obsahuje onu abstraktní metodu `executeInt`. Uzel obsahuje dva potomky `leftNode` a `rightNode`. V těle metody jsou pak prvky prvně vyhodnoceny a následně sečteny. Metoda obsahuje pouze jeden argument typu `VirtualFrame`, kterému je věnována následující kapitola.

```
1  public final class AdditionNode extends SchemeNode {
2      @SuppressWarnings("FieldMayBeFinal")
3      @Child
4      private SchemeNode leftNode, rightNode;
5
6      public AdditionNode(SchemeNode leftNode, SchemeNode rightNode) {
7          this.leftNode = leftNode;
8          this.rightNode = rightNode;
9      }
10
11     @Override
12     public int executeInt(VirtualFrame frame) {
13         int leftValue = this.leftNode.executeInt(frame);
14         int rightValue = this.rightNode.executeInt(frame);
15         return leftValue + rightValue;
16     }
17 }
```

Zdrojový kód 3: Příklad implementace uzlu pro sčítání argumentů typu `int`

3.3.2 VirtualFrame

`VirtualFrame` je třída odpovídající klasickému zásobníkovému rámci. Jedná se o datovou strukturu, která slouží k ukládání potřebných dat spojených s voláním funkce. Jedná se o data jako například lokální proměnné funkce nebo její parametry. `VirtualFrame` je vytvořen automaticky při zavolání funkce a je vytvořen na základě třídy `FrameDescriptor`. Tato třída může být chápána jako „popisovač“, podle kterého jsou následně vytvořeny instance třídy `VirtualFrame`. Obsahuje informace jako například počet lokálních proměnných a jejich

typ, které jsou následně využity k vytvoření paměťově optimální instance. Lokální proměnné primitivního typu (např. `int`, `double`, `float`) jsou uloženy v poli typu `long[]`, zatímco všechny ostatní proměnné jsou uloženy v poli typu `Object[]`. Stojí za povšimnutí, že všechny primitivní typy je možné převést na hodnoty typu `long`. Například hodnoty typu `double` lze převést na hodnoty typu `long` pomocí statických metod `Double.doubleToRawLongBits()` a `Double.longToDoubleBits`. [7]

3.3.3 RootNode

`RootNode` je speciálním uzlem (potomkem třídy `Node`), který reprezentuje kořen daného stromu. Jedná se opět o abstraktní třídu stejně jako `Node`. Na rozdíl od `Node` obsahuje již abstraktní metodu `execute`, kterou je nutné implementovat. To znamená, že `RootNode` slouží k zapouzdření jednoho nebo více uzlů, které je možné následně vyhodnotit. To nejčastěji reprezentuje části hostovaného jazyka, které lze volat. Může se jednat nejen o procedury či funkce, ale i o vstupní body programu (například v Javě se jedná o statickou metodu `main`). V praxi se ale přímo `RootNode` k volání funkcí nepoužívá. Používá se `CallTarget`, který je popsán v následující kapitole.

3.3.4 CallTarget

`CallTarget` je poslední abstrakcí nad `RootNode`, avšak již se nejedná o abstraktní třídu, ze které by měl vývojář dědit. Místo toho se jedná o rozhraní obsahující metodu `call`, která přijímá libovolný počet argumentů a slouží k vyhodnocení uzlů uložených v `RootNode`. Implementace tohoto rozhraní je pak poskytována aktuálním běhovým prostředím (např. GraalVM Community Edition nebo GraalVM Enterprise Edition). Hlavním cílem této abstrakce je:

1. Sbírat a ukládat profilující informace z běhu aplikace. Nejčastěji se jedná o informace jako počet volání dané metody k určení, zda je vhodným kandidátem pro JIT kompilaci.

2. Vytvořit novou instanci třídy `VirtualFrame`, která bude obsahovat potřebné argumenty k uskutečnění daného volání.

3.4 Specializace

Specializace jsou jednou z klíčových optimalizací frameworku Truffle. V této kapitole je popsán princip specializací spolu s partial evaluation, která je nezbytnou součástí tvorby velice efektivního strojového kódu. Kapitola vychází ze zdroje [23].

Jak již bylo nastíněno v předchozí kapitole, specializace je možné vnímat jako dělení sémantiky operace na více částí (specializací). Hlavními výhodami tohoto dělení je například:

- Rozdělení komplexních operací na menší části. To je obecně dobrou praktikou v softwarovém inženýrství. Tyto menší části lze pak jednodušeji implementovat a testovat.
- Malé specializované metody umožní generovat velice rychlý a kompaktní strojový kód.

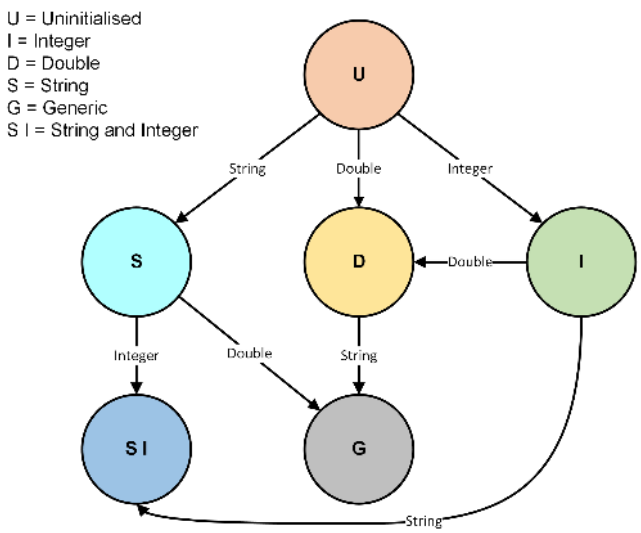
Obecná implementace specializací se skládá ze dvou částí. Tou první je kontrola, zda aktuální specializace pokrývá sémantiku na základě typu vstupních argumentů. Druhou částí je pak samotné provedení logiky operace. Jinými slovy, logika operace je chráněna (ang. *guarded*) typovou kontrolou vstupních argumentů. Pokud typová kontrola vstupních argumentů není úspěšná (specializace nepokrývá sémantiku danou typem vstupních argumentů), může dojít ke dvěma případům:

1. V případě, že se daný uzel pouze interpretuje, tzn. uzel nebyl přeložen do strojového kódu, dojde pouze ke změně jeho vnitřního stavu. Tato změna reprezentuje přidání nové specializace, která pokrývá sémantiku danou vstupními argumenty.
2. V případě, že uzel již byl přeložen do strojové kódu, dochází k deoptimalizaci a strojový kód je zahozen. Uzel se opět začne interpretovat a změní se jeho vnitřní stav, stejně jako v předchozím případě. Jakmile Truffle vyhodnotí, že je uzel opět vhodným kandidátem k JIT kompilaci, dojde opět k překladu do strojového kódu. Takto nově vytvořený strojový kód je rozšířen o sémantiku, která způsobila předchozí deoptimalizaci.

Pokud uzel reprezentuje operaci sčítání, ve většině dynamických jazyků lze vytvořit tři specializace `I`, `D` a `S`. Specializace `I` reprezentuje sčítání čísel typu `integer`, specilizace `D` sčítání čísel typu `double` a specializace `S` reprezentuje konkatenaci řetězců. Všechny specializace dohromady pokrývají celou sémantiku operátoru sčítání. Na obrázku 10 je takový uzel zobrazen se všemi stavy, do kterých se může dostat.

Na začátku je uzel v *neinicializovaném* stavu. Po prvním zavolání se uzel specializuje na základě typu vstupních argumentů a přejde do stavu `I`, `D` nebo `S`. Pokud by specializace dokázala obsloužit všechny budoucí volání (to by znamenalo, že operátor sčítání byl volán pouze s jedním typem argumentů) nazýváme takový uzel *monomorfní*. Takový uzel je co se týče výkonu optimální, ale není možné garantovat, že v budoucnu tento předpoklad nebude porušen.

V případě, že tento předpoklad je porušen, nelze jen jednoduše přejít do jiné specializace a znovu se specializovat jen na základě nových vstupních argumentů, protože by bylo možné dojít do stavu, kdy se alternuje mezi dvěma nebo více stavy, což porušuje podmínku konečnosti. Nutné je zmínit, že deoptimalizace má výrazný vliv na výkon aplikace, tudíž alternací mezi několika stavy by výkon výrazně klesl. Je potřeba, aby uzel byl schopný pokrýt více specializací zároveň,



Obrázek 10: Uzel reprezentující operaci sčítání se všemi možnými přechody od neinicializovaného stavu `U` až do generického stavu `G`

jinými slovy, pokrýt více typů vstupních argumentů. Takový uzel nazýváme *polymorfní*. Implementovat všechny možné kombinace jednotlivých specializací je nepředstavitelné, protože počet kombinací roste exponenciálně vzhledem k počtu specializací. Proto Truffle využívá *zřetězení* (ang. chaining), kde specializace jsou řetězeny za sebou a první specializace, která je schopná pokrýt sémantiku danou vstupními argumenty je použita. Je zřejmé, že s rostoucím počtem specializací klesá výkon aplikace. Uzel, který obsahuje všechny specializace, je označován jako generický.

Pokud ve výše zmíněné operaci sčítání je funkce nejprve volána s argumenty typu `double`, přejde uzel do specializace `D`. Poté bude funkce zavolána s argumenty typu `string`. V takovém případě, jak je možné vidět na obrázku 10, nedojde k přechodu do specializace `S`, ale do generické specializace `G`. Uzel se tak stal polymorfním, a to zřetězením specializace `D` a `S`. Pokud by v tomto případě nebyla zvolena specializace `G`, ale specializace `S` mohlo by dojít k výše zmíněným alternacím a byla by porušena podmínka konečnosti.

Dále je možné vidět na obrázku 10, že nikdy nedojde k zřetězení specializace `D` se specializací `I`, a to z důvodu, že specializace `I` je striktní podmnožinou specializace `D` (celá čísla jsou podmnožinou racionálních čísel). Nemusí tak vždy dojít k řetězení jednotlivých specializací, ale specializace může být zcela nahrazena jinou specializací, která zároveň pokrývá sémantiku předchozí specializace. Všechny možnosti, jak lze jednotlivé specializace kombinovat jsou popsány v kapitole 3.4.1.

Specializace jsou reprezentovány v uzlu jako stavy, do kterých na základě typu vstupních argumentů uzel může přejít. V podstatě se jedná o konečný deterministický automat, který vždy začíná v neinicializovaném stavu. Po prvním přechodu vždy přejde do monomorfního stavu, který je co se týče výkonu op-

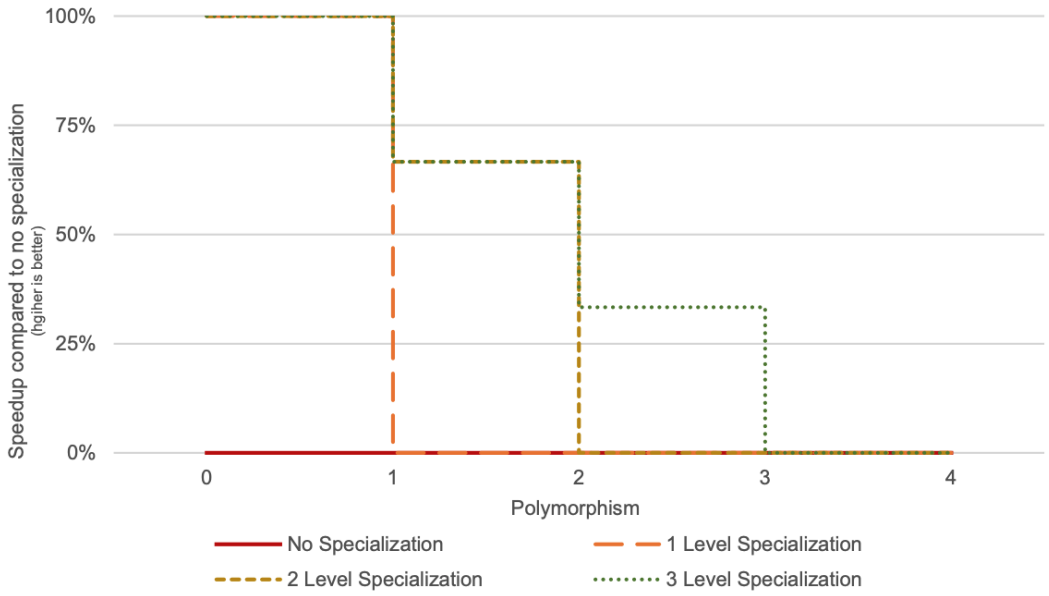
timální. Další řetězení specializací činí uzel polymorfní. Pokud má uzel aktivní všechny specializace, je v generickém stavu a nedojde tak již k žádné další specializaci. Generický stav uzlu je samozřejmě z hlediska výkonu nejhorší.

3.4.1 Specializace a výkon aplikace

Výkonné implementace virtuálních strojů jsou založeny na velice agresivních optimalizacích. Nejčastěji jsou tyto optimalizace založeny na dvou předpokladech:

1. *Hodnota se nezmění.* Na základě tohoto předpokladu jsou proměnné vnímány běhovým prostředím jako konstanty. Použití těchto proměnných lze následně optimalizovat pomocí výpočtu konstantních výrazů a jejich propagace. To nejčastěji vede k odstranění větvení, a tím se otevírá prostor pro další optimalizace.

2. *Typ se nezmění.* Na tomto předpokladu jsou většinou založeny specializace Truffle. Tímto je výrazně redukována velikost generovaného strojového kódu, jelikož je možné odstranit větvení vybírající správnou logiku dle typu vstupních argumentů. Tento typ spekulace je obzvláště důležitý pro dynamické jazyky.



Obrázek 11: Porovnání rychlosti s rostoucím počtem specializací. V případě, že počet specializací je nízký, může dojít k náhlému propadu ve výkonu a způsobit tzv. performance cliff [23]

V předchozí kapitole již bylo zmíněno, že součástí specializací jsou kontroly (ang. *guards*), které zaručí správný výběr logiky na základě typu vstupních argumentů. V případě, že kontroly specializací selžou, je nutné ke stávající specializaci

zřetězit novou specializaci a rozšířit tak sémantiku operace. V nejhorším případě se uzel může dostat do generického stavu, který je nejtěžší na optimalizaci. Přechodem do generického stavu může být způsoben náhlý propad ve výkonu, který je často nazýván *performance cliff*.

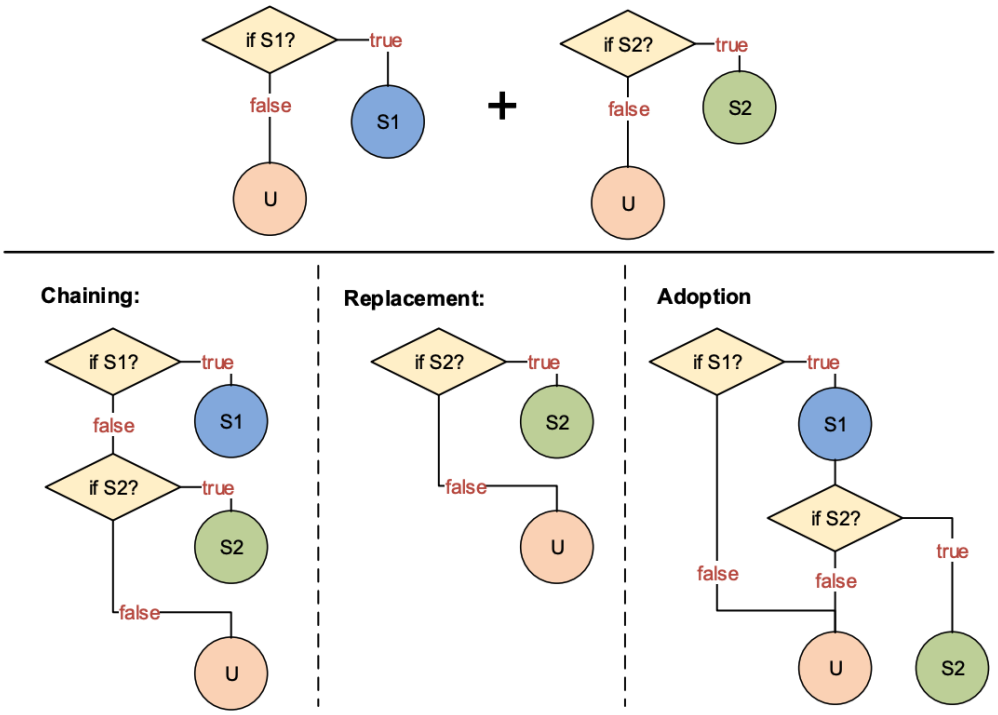
Náhlé změně výkonu lze předejít několika vrstvami specializací. Cílem je předejít přechodu do generického stavu, který je z hlediska výkonu nejhorší. Obrázek 11 ukazuje možný propad výkonu s rostoucím stupněm polymorfismu. Stupeň polymorfismu vyjadřuje, s kolika různými typy vstupních argumentů byla operace již zavolána. Pokud například byla operace sčítání prvně zavolána se vstupními argumenty typu `double` a poté s argumenty typu `string`, je stupeň polymorfismu roven dvěma. Je důležité poznamenat, že nelze určit dle stupně polymorfismu, zda je uzel v generickém stavu, či ne. V příkladu, který je zobrazen na obrázku 10 je vidět, že stav `SI` má stejný stupeň polymorfismu jako generický stav *G*. Stav `SI` je užitečný jen za předpokladu, že je efektivnější než stav `G`. Vrstvením jednotlivých specializací lze předejít náhlým výkonnostním změnám. Existují tři možnosti [23], jakými lze specializace vrstvit (viz obrázek 12).

- **Řetězení** (ang. chaining) sloučí dvě specializace tak, že se nejprve vykonají kontroly první specializace a v případě že selžou, zavolají se kontroly druhé specializace. Je doporučeno, aby množiny vstupních argumentů jednotlivých specializací byly disjunktní. Řetězení je obecná technika, kterou využívá například i *polymorphic inline caches*.

- **Nahrazení** (ang. replacement) sloučí dvě specializace tak, že první specializace je nahrazena druhou specializací. To je možné jen v případě, že vstupní argumentu první specializace jsou podmnožinou vstupních argumentů druhé specializace.

- **Adopce** (ang. adoption) sloučí dvě specializace tak, že druhá specializace se stane potomkem první specializace. V praxi se adopce používá například k rozdělení komplexních operací na menší části. Například jazyk R nemá skalární hodnoty samotné, ale má vektor skalárních hodnot. Je možné vytvořit uzly se specializacemi pro aritmetiku se skalárními hodnotami, které budou potomky uzlů se specializacemi pro práci se samotnými vektory.

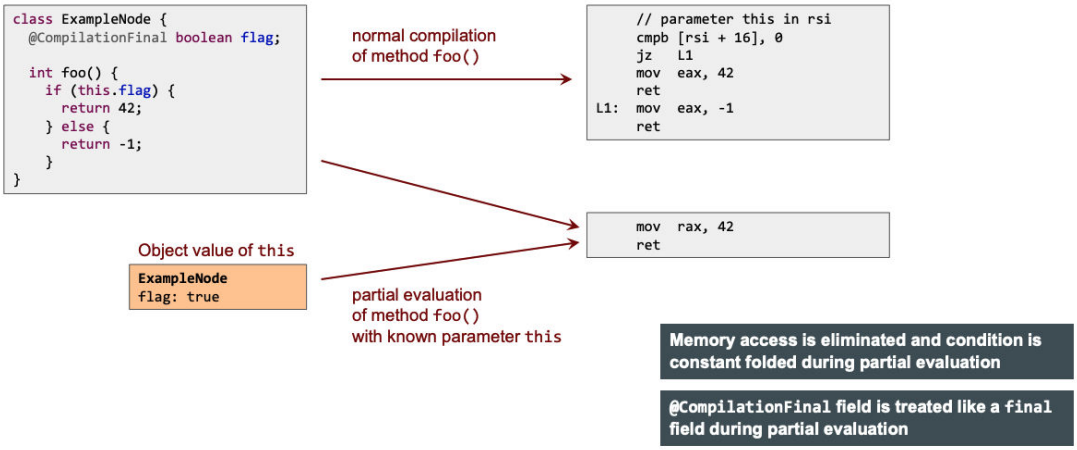
Při psaní interpretu je pak potřeba najít rovnováhu v počtu specializací pro každou operaci. V případě, že specializací je málo, uzel se brzy dostane do generického stavu a vzniká tak výše zmíněný náhlý propad ve výkonu. Pokud naopak bude specializací příliš mnoho, časté deoptimalizace a překlady do strojového kódu se také negativně projeví na rychlosti interpretovaného jazyka.



Obrázek 12: Varianty vrstvení dvou specializací k zamezení náhlému propadu ve výkonu (performance cliff) [23]

3.4.2 Specializace a partial evaluation

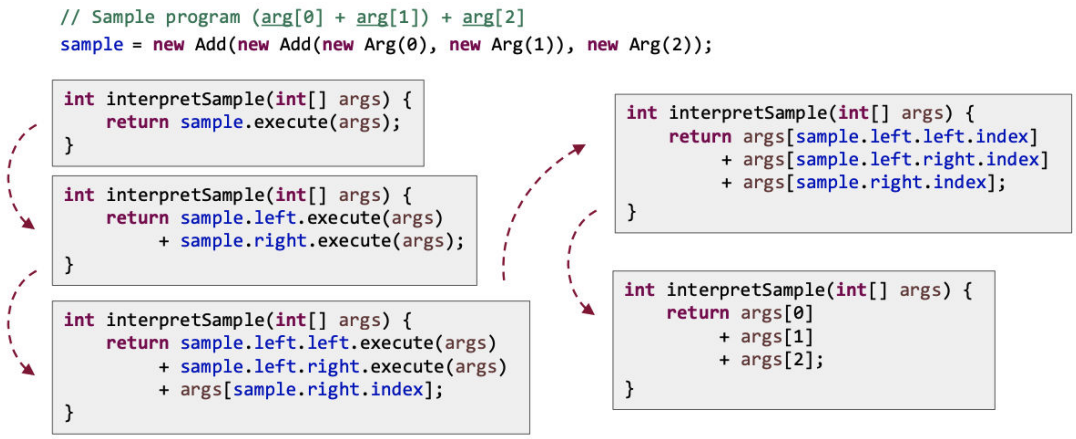
Partial evaluation (dále PE) na platformě GraalVM je ve své podstatě velice agresivní aplikace optimalizací, jako například rozbalování programových smyček, eliminace volání metod nebo výpočet konstantních výrazů na interpret a program samotný (včetně dat z běhu programu). Důležité je zmínit, že vývojář interpretu si je vědom, že na daný kód může být aplikována PE, a tudíž píše kód interpretu tak, aby na něj mohla být PE aplikována jednoduše (píše tzv. *PE-friendly code*). Na obrázku 13 lze například vidět, že použitím anotace `@CompilationFinal` dochází k eliminaci větvení a vzniká tak krátký a efektivní strojový kód.



Obrázek 13: Použití anotace `@CompilationFinal` u vlastnosti způsobí eliminaci větvení, a tudíž výsledný strojový kód je výrazně kratší a efektivnější [24]

Na obrázku 14 je další ukázka toho, jak PE dokáže zefektivnit a zjednodušit kód. V tomto případě se jedná již o program, ve kterém dochází ke sečtení tří hodnot. Nejprve je vytvořen abstraktní syntaktický strom Truffle, který daný program reprezentuje. I přesto, že interpret pracuje nad třídami jako jsou `Add` nebo `Arg`,[12] Překladač Graal má stále přístup k bytecode jednotlivých tříd skrze reflexi. Bytecode je nejprve převeden do Graal IR a následně je aplikována PE. Jak je možné vidět, postupnou aplikací optimalizací (v tom případě výpočet konstantních výrazů a eliminace volání metod) dochází k postupnému vkládání těl jednotlivých `execute` metod. Díky těmto optimalizacím nejprve dochází k odstranění tříd `Add` a `Arg` a následně dochází i k eliminaci všech jejich `execute` metod. Výsledkem je krátký a velice efektivní kód, ve kterém byly eliminovány jakékoliv známky interpretu. Nutno zmínit, že všechny tyto transformace probíhají nad Graal IR, tudíž jednotlivé transformace kódu viditelné na obrázku 14 slouží pouze k demonstračním účelům.

[12]Jedná se o potomky třídy `Node`, tudíž všechny obsahují metodu `execute` a potomci jsou deklarováni pomocí anotace `@Child` nebo `@Children`.



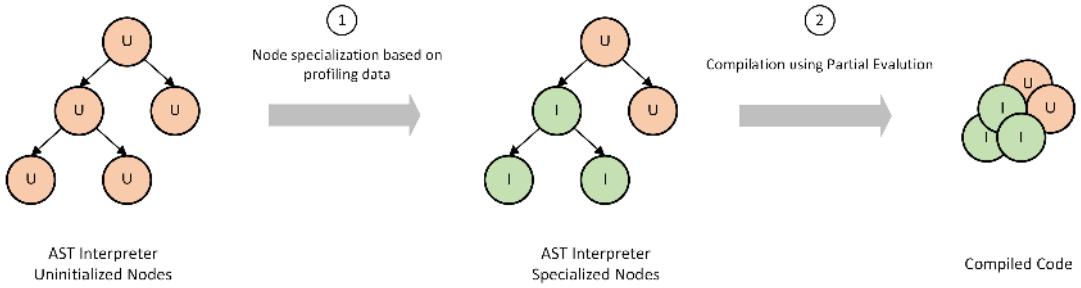
Obrázek 14: Aplikace PE na program reprezentující sčítání tří hodnot. Postupně jsou eliminovány jakékoliv známky interpretu [24]

Jak již bylo popsáno v kapitole 3.1, Truffle si řídí JIT kompilaci metod interpretovaného jazyka. Aby metoda byla vhodným kandidátem pro JIT kompilaci, musí splňovat následující podmínky:

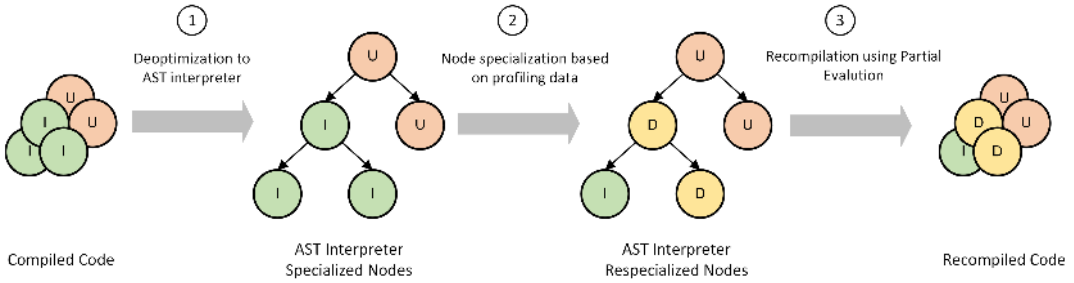
- Počet volání dané metody musí překročit hranici pro JIT kompilaci.
- Abstraktní syntaktický strom Truffle, který reprezentuje danou metodu, musí být stabilní. Strom je stabilní ve chvíli, kdy nedochází k dalším specializacím.

Jakmile metoda splňuje obě podmínky, dochází k JIT kompilaci řízené frameworkem Truffle, jejíž první fází je PE. Touto fází se liší od tradiční JVM řízené JIT kompilace. To znamená, že překladač Graal je rozšířen o tuto fázi vzhledem k C2 překladači a aplikuje ji pouze v případě, že je kompilace řízena frameworkem Truffle. Abstraktní syntaktický strom Truffle, reprezentující kompilovanou metodu, je nejprve převeden do interní reprezentace (Graal IR), na kterou je následně aplikována PE. Výsledkem je efektivnější Graal IR, který již prochází standardními fázemi běžné JIT kompilace. Celý proces je vizualizován na obrázku 15.

V případě, že předpoklad stability stromu je porušen a strojový kód nedokáže pokrýt sémantiku vstupních argumentů, dochází k deoptimalizaci. Tímto dojde k odstranění dříve přeloženého strojového kódu a přechází se k interpretaci. Abstraktní syntaktický strom Truffle je znovu specializován na základě nových vstupních argumentů a v případě, že jsou splněny obě výše zmíněné podmínky, je zahájena nová JIT kompilace, která je opět řízena frameworkem Truffle. Proces deoptimalizace je vizualizován na obrázku 16.



Obrázek 15: Na základě profilujících dat se uzly specializují (1). Jakmile je strom stabilní a počet volání metody překročí hranici pro JIT kompilaci, je na specializovaný strom nejprve aplikována PE a poté je přeložen do strojového kódu (2) [25]



Obrázek 16: Jestliže spekulace stability stromu je porušena, dochází k deoptimalizaci, strojový kód je zahozen a AST strom se interpretuje (1). Uzly se opět specializují na základě profilujících dat (2), a jakmile jsou podmínky pro JIT kompilaci znovu splněny, dochází k další kompilaci do strojového kódu (3) [25]

3.5 Truffle DSL

Sebeoptimalizace pomocí specializací je klíčovou částí frameworku Truffle. Přechod mezi jednotlivými specializacemi je velice repetitivní a náročný proces s vysokou mírou pravděpodobnosti zanesení chyby. Tento problém je řešen pomocí Truffle DSL, což je anotační procesor využívající doménově specifický jazyk sloužící pro řízení přechodů mezi jednotlivými specializacemi. Ve zdrojovém kódu 4 je možné vidět zjednodušený příklad, jak by mohla vypadat implementace sčítání čísel typu `integer` bez použití Truffle DSL. Třída obsahuje vlastnost `specializationState` označenou anotací `@CompilationFinal`, ve které je uloženo, jaká specializace je právě aktivní. Jakmile je na tento uzel aplikována PE, kód neaktivních specializací je eliminován. Jak je možné vidět z implementace, kód pro sčítání dvou čísel typu `integer` je asi 30 řádků dlouhý, a přitom hlavní logika, kterou by se měl vývojář primárně zabývat, zabírá pouhý jeden řádek (sčítání dvou čísel je na řádku 33).

Z tohoto důvodu byl vytvořen Truffle DSL, který jednotlivé přechody mezi specializacemi vygeneruje, což nejen snižuje množství kódu, který vývojář musí během implementace interpretu napsat, ale i snižuje pravděpodobnost zanesení chyby. Další výhodou je fakt, že se vývojář zabývá pouze logikou dané specializace, a tím se zjednodušuje celý proces implementace dané operace. Ve zdrojovém kódu 5 je možné vidět použití Truffle DSL na uzlu reprezentujícím operaci sčítání. Jelikož anotační procesor nesmí modifikovat existující třídy,[13] musí být všechny třídy využívající Truffle DSL abstraktní. Během kompilace je pak vygenerována třída obsahující logiku jednotlivých specializací, včetně logiky starající se o přechody mezi jednotlivými specializacemi. Jak je možné vidět ve zdrojovém kódu, potomci uzlu jsou v tomto případě definováni pomocí anotace `@NodeChild`, která je sémanticky totožná s anotací `@Child`. Stejně tak anotace `@Children` by byla nahrazena anotací `@NodeChildren`. Specializace jsou reprezentovány pomocí metod, které obsahují anotaci `@Specialization`. Pomocí atributu `rewriteOn` je pak možné specifikovat, kdy se má specializace deaktivovat. V tomto případě specializace pro sčítání dvou čísel typu `integer` bude deaktivována, jakmile nastane `ArithmeticException` a bude nahrazena specializací pro sčítání čísel typu `double` (viz `replaces = "addInts"` na řádku 10 ve zdrojovém kódu 5). Nahrazení specializace za novou je zde možné pouze z důvodu, že čísla typu `integer` jsou podmnožinou čísel typu `double`.

[13] Standard anotačního procesoru Java zakazuje modifikaci existujících tříd. Projekt Lombok tento standard porušuje. [27]

```
1  public final class AdditionNode extends ExampleNode {
2  @Child
3  private ExampleNode leftNode, rightNode;
4
5  private enum SpecializationState {UNINITIALIZED, INT, DOUBLE}
6
7  @CompilationFinal
8  private SpecializationState specializationState;
9
10 ....
11
12 @Override
13 public int executeInt(VirtualFrame frame) throws
       UnexpectedResultException {
14     int leftValue;
15     try {
16         leftValue = this.leftNode.executeInt(frame);
17     } catch (UnexpectedResultException e) {
18         this.activateDoubleSpecialization();
19         double leftDouble = (double) e.getResult();
20         throw new UnexpectedResultException(leftDouble +
               this.rightNode.executeDouble(frame));
21     }
22
23     int rightValue;
24     try {
25         rightValue = this.rightNode.executeInt(frame);
26     } catch (UnexpectedResultException e) {
27         this.activateDoubleSpecialization();
28         double rightDouble = (double) e.getResult();
29         throw new UnexpectedResultException(leftValue +
               rightDouble);
30     }
31
32     try {
33         return Math.addExact(leftValue, rightValue);
34     } catch (ArithmeticException e) {
35         this.activateDoubleSpecialization();
36         throw new UnexpectedResultException((double) leftValue +
               (double) rightValue);
37     }
38 }
39
40 private void activateDoubleSpecialization() {
41     this.specializationState = SpecializationState.DOUBLE;
42 }
43
44 // omitted other execute methods for simplicity
```

Zdrojový kód 4: Příklad implementace uzlu pro sčítání dvou čísel bez využití Truffle DSL [26]

```java
@NodeChild("leftNode") @NodeChild("rightNode")
public abstract class AdditionNode extends SchemeExpression {

    @Specialization(rewriteOn = ArithmeticException.class)
    protected int addInts(int leftVal, int rightVal) {
        return Math.addExact(leftVal, rightVal);
    }

    @Specialization(replaces = "addInts")
    protected double addDoubles(double leftVal, double rightVal) {
        return leftVal + rightVal;
    }

    @Specialization
    protected String addStrings(String leftVal, String rightVal) {
        return leftVal + rightVal;
    }
}
```

Zdrojový kód 5: Implementace uzlu pro sčítání dvou čísel s využitím Truffle DSL

4 Scheme na platformě GraalVM

Tato kapitola se bude zabývat popisem jednotlivých funkcionalit, které tato implementace jazyka Scheme podporuje (dále také nazývaná TruffleScheme). Jak již bylo zmíněno v úvodní kapitole, cílem této práce nebylo implementovat co nejvíce ze specifikace jazyka Scheme, ale implementovat prvky, které dosud na platformě nebyly implementované a zároveň umožní vykonání benchmarkových testů.

4.1 Datové typy

V této sekci jsou popsány všechny datové typy, které TruffleScheme podporuje. Zároveň je zmíněno, jak je každý interní typ reprezentován.

- **Pravdivostní hodnota** je interně reprezentována jako Java `boolean`.
- **Celá čísla** jsou interně reprezentována jako Java `long`. V případě, že datový typ `long` není dostačující, je využit `BigInteger`. Nutno zmínit, že třída `BigInteger` není použita přímo, ale je zapouzdřena do třídy nazývané `SchemeBigInt` z důvodu podpory interoperability.
- **Čísla s plovoucí desetinou čárkou** jsou interně reprezentována primitivní hodnotou `double`.
- **Symbol** je interně reprezentován třídou `SchemeSymbol`.
- **Řetězec** je interně reprezentován třídou `TruffleString` z knihovny Truffle API. Implementace obsahuje funkce pro běžnou práci s řetězcem, které na rozdíl od běžné Java implementace podporují partial evaluation.
- **List** je interně reprezentován třídou `SchemeList`.
- **Tečkový pár** je interně reprezentován třídou `SchemePair`.
- **Primitivní procedury** jsou interně reprezentovány třídou `Primitive-Procedure`.
- **Uživatelsky definované procedury** jsou interně reprezentovány třídou `UserDefinedProcedure`.
- **Nedefinovaná hodnota** je interně reprezentována třídou `Undefined-Value`.

4.2 Primitivní procedury

Primitivní procedury jsou základními stavebními bloky, které jsou programátorovi poskytnuty automaticky, bez nutnosti jejich definice. Všechny vazby mezi primitivní procedurou a symbolem jsou vytvořeny automaticky při startu interpretu v globálním prostředí. Jedná se stejné vazby, které programátor může vytvořit, tudíž je lze redefinovat. Podporové primitivní procedury jsou:

- **Aritmetické:** +, -, *, /, modulo.
- **Porovnání čísel:** <, <=, >, >=, =.
- **Práce s listem nebo párem:** car, cdr, cons, append, length, list.
- **Ostatní:** apply, map, not, null?, current-milliseconds, display.

4.3 Speciální formy

Některé konstrukty jazyka nelze implementovat pomocí primitivních procedur. Například při definování nové vazby je důležité zajistit, aby symbol nebyl vyhodnocen před samotnou aplikací speciální formy. Vzhledem k tomu, že primitivní procedury vyhodnocují všechny své argumenty před aplikací procedury, jsou potřeba nové elementy jazyka, které si budou samy řídit vyhodnocení jednotlivých argumentů. Speciální formy jsou právě těmito elementy jazyka. Tabulka 1 vysvětluje význam operátorů, použitých pro definování gramatik jednotlivých speciálních forem.

Operátor	Význam
?	0 až 1 krát
*	0 až n krát
+	1 až n krát

Tabulka 1: Význam operátorů použitých pro definovaní gramatik

Podporované speciální formy a jejich gramatiky jsou:

- **define**: `(define <jméno> <výraz>)`
- **lambda**: `(lambda (<param>*) <tělo>)`
- **let**: `(let (<vazba>+) <tělo>)`
- **letrec**: `(letrec (<vazba>+) <tělo>)`
  - kde `vazba`: `(symbol hodnota)`
- **if**: `(if <test> <důsledek> <náhradník>?)`
- **and**: `(and <test>*)`

- **or**: `(or <test>*)`
- **quote**: `(quote <arg>)`
- **quasiquote**: `(quasiquote <arg>)`
- **define-macro**: `(define-macro <jméno> <výraz>)`

4.4 Makra

Jazyk Scheme umožňuje uživateli definovat vlastní speciální formy pomocí maker. Makra lze obecně rozdělit do dvou skupin, *hygienická* a *nehygienická*. Rozdíl spočívá v tom, zda makro automaticky řeší konflikt jmen lokálních proměnných či parametrů. Jak je z názvu zřejmé, hygienická makra tento problém řeší automaticky, naopak u nehygienických maker se o „hygienu“ musí postarat programátor sám. Hygienická makra je možné najít v jazycích jako Rust, Julia či Dylan, naopak nehygienická makra je pak možné najít v jazycích jako Clojure nebo Common Lisp. I přesto, že ve specifikaci jazyka Scheme není zmínka o nehygienických makrech, existuje mnoho implementací, které tyto makra podporují. Například implementace Racket podporuje nehygienická makra primárně z důvodu kompatibility. Nevýhodou této implementace je fakt, že makro expanze nehygienických maker probíhá ve fázi nazývané *expanze syntaxe*, ve které se nemohou využívat vazby definované uživatelem (jelikož tyto vazby ještě nevznikly). [28]

TruffleScheme implementuje pouze nehygienická makra, a to pomocí výše zmíněné speciální formy `define-macro`. To znamená, že každé makro musí obsahovat tzv. *transformační proceduru*, což je procedura, jejíž výsledkem jsou data, která nahradí místo volání daného makra (to je možné, jelikož data lze chápat jako program, viz kapitola 5.1). Tato transformace se často nazývá *expanze maker*. Implementačními detaily maker se zabývá kapitola 5.7.

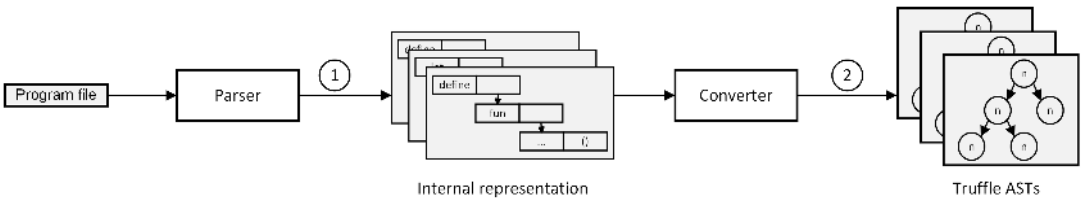
5 Implementace interpretu

V této kapitole je nejprve popsána celková architektura interpretu včetně všech jeho částí a poté je popsána implementace základních prvků, jako jsou například primitivní procedury, speciální formy nebo uživatelsky definované procedury.

5.1 Architektura interpretu

Základní architekturu interpretu je možno vidět na obrázku 17. Vstupním argumentem je soubor obsahující zdrojový kód, který přijímá `parser`. Ten provádí lexikální a syntaktickou analýzu zdrojového kódu pomocí nástroje ANTLR.[14] Výsledkem je *interní reprezentace* neboli vnitřní forma, což je způsob, kterým jsou reprezentována data (1). Stejná data jsou pak výsledkem vyhodnocení výrazů.

`Converter` následně přijímá tuto interní reprezentaci (data) a transformuje je do abstraktních syntaktických stromů Truffle (2). Tyto stromy jsou následně uloženy do `SchemeRootNode`, což je potomek `RootNode`, který reprezentuje kořen daného stromu. To znamená, že celý program si lze představit jako jeden velký abstraktní syntaktický strom Truffle (primitivní a uživatelsky definované procedury jsou samostatné abstraktní stromy, které jsou odtud volány), který je po konverzi vyhodnocen. Důležité je podotknout, že jakákoliv data lze transformovat na abstraktní syntaktické stromy Truffle, což znamená, že program lze chápat jako data nebo data lze chápat jako program.



Obrázek 17: Architektura interpretu

5.2 Lokální proměnné

Jak již bylo zmíněno v kapitole 3.3.2, `VirtualFrame` slouží k ukládaní lokálních proměnných. O zapisování jednotlivých lokálních proměnných se stará uzel pojmenovaný `WriteFrameSlotNode`. Implementace tohoto uzlu (viz zdrojový kód 6) obsahuje hned několik prvků, které již byly popsány, ale nebyly demonstrovány na příkladu. První je vlastnost `guards` v anotaci `@Specialization`, která je detailně popsána v kapitole 3.4. Jedná se o kontrolu, která se provádí před vykonáním samostatné specializace. Výraz, který se nachází v kontrole

[14] ANTLR (ANother Tool for Language Recognition) je nástroj, který na základě definované gramatiky vygeneruje lexikální a syntaktický analyzátor. [29]

(v tomto případě volání funkce `isExpectedOrIllegal`) se musí vyhodnotit na `boolean` hodnotu. V případě, že se výraz v kontrole vyhodnotí na `true`, je specializace vykonána, v opačném případě se přechází na další specializaci v pořadí. Je důležité zmínit, že specializace se vykonávají v pořadí, ve kterém jsou deklarovány ve zdrojovém kódu. Je proto vhodné, aby první specializace byla tou nejvíce pravděpodobnou a naopak poslední specializace byla tou nejméně pravděpodobnou (nebo generickou, která pokrývá celou sémantiku operace). V tomto případě je možné vidět, že poslední specializace je generická, jelikož hodnota, kterou ukládáme, je typu `Object`, což je typ, na který lze v Javě převést jakoukoliv jinou hodnotu. Dále stojí za povšimnutí, že tato generická specializace obsahuje vlastnost `replace`, která nahrazuje všechny předchozí specializace. Uzel se tak dostal do generického stavu a k dalším specializacím již nedojde (viz podmínka konečnosti v kapitole 3.2).

Poslední zajímavostí je aktualizace `FrameDescriptoru`, kterou je možné vidět na řádku 33 ve zdrojovém kódu.[15] Na začátku má každá lokální proměnná nastavený typ na `Illegal` (odtud název funkce `isExpectedOrIllegal`), jelikož typ lokální proměnné není znám. Po prvním uložení hodnoty se `FrameDescriptor` aktualizuje a při dalším zavolání je vytvořen paměťově úspornější `VirtualFrame`. Pokud by lokální proměnná vždy nabývala hodnoty stejného typu, strojový kód vytvořený JIT kompilátorem by byl totožný se strojovým kódem staticky typovaného jazyka.

[15] Aktualizace `FrameDescriptoru` probíhá i v ostatních specializacích, a to v metodě `isExpectedOrIllegal`, jejíž implementace z důvodu stručnosti byla vynechána.

```
1  public abstract class WriteFrameSlotNode extends SchemeNode {
2
3      @CompilationFinal
4      private FrameDescriptor cachedDescriptor;
5
6      private final int frameSlot;
7
8      public abstract void executeWrite(VirtualFrame frame, Object
           value);
9
10     public WriteFrameSlotNode(int frameSlot) {
11         assert frameSlot >= 0;
12         this.frameSlot = frameSlot;
13     }
14
15     @Specialization(guards = "isExpectedOrIllegal(frame, Long)")
16     protected void writeLong(VirtualFrame frame, long value) {
17         frame.setLong(frameSlot, value);
18     }
19     @Specialization(guards = "isExpectedOrIllegal(frame, Boolean)")
20     protected void writeBoolean(VirtualFrame frame, boolean value) {
21         frame.setBoolean(frameSlot, value);
22     }
23
24     @Specialization(guards = "isExpectedOrIllegal(frame, Double)")
25     protected void writeDouble(VirtualFrame frame, double value) {
26         frame.setDouble(frameSlot, value);
27     }
28
29     @Specialization(replaces = { "writeBoolean", "writeLong",
           "writeDouble" })
30     protected void writeObject(VirtualFrame frame, Object value) {
31         /* No-op if kind is already Object. */
32         final FrameDescriptor descriptor =
               getFrameDescriptor(frame);
33         descriptor.setSlotKind(frameSlot, FrameSlotKind.Object);
34
35         frame.setObject(frameSlot, value);
36     }
37
38     //omitted for simplicity
```

Zdrojový kód 6: Uzel zapisující lokální proměnnou

5.3 Globální proměnné

Globální proměnné se neukládají do `VirtualFramu`, ale do tzv. *jazykového kontextu*, který je vytvořen při spuštění interpretu. To je zcela běžné v jazycích implementovaných nad platformou GraalVM, a to z důvodu podpory polyglotního programování. Dost často je nutné přistupovat do globálního prostředí cizího jazyku, a proto se musí jednat o objekt, který podporuje interoperabilitu (což `VirtualFrame` není). Detailněji se problematikou interoperability zabývá kapitola 6.

Implementaci jazykového kontextu je možné vidět ve zdrojovém kódu 7, kde třída obsahuje vlastnost `globalVariableStorage`, což je mapa, ve které jsou uloženy globální proměnné. Tato třída také poskytuje dvě metody pro přidávání a čtení globálních proměnných. Dále zdrojový kód obsahuje dva nové prvky, které dosud nebyly podrobně popsány.

První je anotace `@TruffleBoundary`, což je informace pro překladač Graal, aby na následující metodu neaplikoval partial evaluation. V tomto případě na metody nemůže být aplikovaná partial evaluation, protože obsahují práci s mapou, jejíž implementace (v tomto případě implementace metod `put` a `get`) by způsobila vygenerování příliš mnoho strojového kódu (došlo by k tzv. *code explosion*).

Druhým prvkem je použití třídy `CyclicAssumption` z knihovny Truffle API, která umožňuje vytvářet vlastní spekulace. Ty jsou klíčovým prvkem sofistikovaných optimalizací. V tomto případě je vytvořena spekulace, že globální proměnné jsou konstantní, a tudíž nebudou změněny. Tímto je možné při čtení globální proměnné hodnotu cachovat a výrazně tak zrychlit opětovné čtení těchto proměnných. Jak je možné vidět ve zdrojovém kódu 7 na řádku 28, spekulace je zneplatněna, jakmile dojde k redefinici již definovaného symbolu. Kontrola, zda je spekulace validní a následné cachování hodnoty je implementováno ve třídě `ReadGlobalVariableExprNode`.

Zajímavostí je, že kontrola, zda je spekulace stále validní není obsažena v JIT zkompilovaném kódu, a to díky tomu, jak jsou spekulace ve frameworku Truffle implementovány. Řídí se tzv. *hollywoodským principem.*[16] Místo toho, aby byla prováděna kontrola, zda je spekulace validní před každým čtením globální proměnné, je sama spekulace „připnuta“ ke strojovému kódu, a jakmile je spekulace zneplatněna, je zneplatněn i strojový kód. Kód je následně opět interpretován a nová spekulace muže být vytvořena (proto je použita třída `CyclicAssumption`, která automaticky po zneplatnění spekulace vytvoří novou).

[16] Hollywoodský princip je známý frází: „*Don't Call Us, We'll Call You*“. Čili kód nezjišťuje voláním, jestli je platný. Místo toho je mu to „sděleno“ připnutou spekulací.

```java
public class SchemeLanguageContext implements TruffleObject {

    private final Map<SchemeSymbol, Object> globalVariableStorage;
    public static final CyclicAssumption notRedefinedAssumption =
        new CyclicAssumption("global variable not redefined");
    public final TruffleLanguage.Env env;
    private final PrintWriter output;

    public SchemeLanguageContext(TruffleLanguage.Env env) {
        this.globalVariableStorage =
            PrimitiveProcedureGenerator.generate();
        this.env = env;
        this.output = new PrintWriter(env.out(), true);
    }

    @TruffleBoundary
    public Object getVariable(SchemeSymbol symbol) {
        var value = globalVariableStorage.get(symbol);
        if (value == null) {
            throw new SchemeException(symbol + ": undefined cannot
                reference an identifier before its definition",
                null);
        }
        return value;
    }

    @TruffleBoundary
    public void addVariable(SchemeSymbol symbol, Object
        valueToStore) {
        var shouldInvalidate =
            globalVariableStorage.containsKey(symbol);
        globalVariableStorage.put(symbol, valueToStore);
        if (shouldInvalidate) {
            notRedefinedAssumption.invalidate();
        }
    }
     //omitted for simplicity
```

Zdrojový kód 7: Jazykový kontext obsahující mapu pro ukládání globálních proměnných

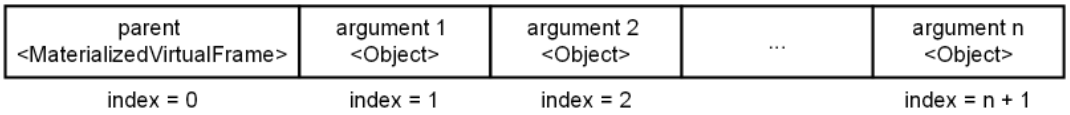
5.4 Uživatelsky definované procedury

Uživatelsky definované procedury jsou v jazyce Scheme *elementy prvního řádu*. Každý element prvního řádu musí splňovat následující podmínky [30]:

- Element může být pojmenován.
- Element může být předán proceduře jako argument.
- Element může vzniknout voláním jiné procedury.
- Element může být obsažen v hierarchických datových strukturách.

Uživatelsky definované procedury jsou ve frameworku Truffle reprezentované pomocí již zmíněného `CallTargetu` (viz kapitola 3.3.4). Ten je zapouzdřen do třídy `UserDefinedProcedure`, což je interní reprezentace, která navíc obsahuje předpokládaný počet argumentů, lexikálního předka nebo jméno dané procedury. Při jejím volání jsou nejprve její argumenty uloženy jako lokální proměnné. To z důvodu, že argumenty jsou předávány skrze pole typu `Object[]` (viz obrázek 18), což by mohlo zapříčinit neustálé *boxování* a *unboxování* vstupních argumentů.[17]

Ve výše zmíněním poli je první prvek vyhrazen lexikálnímu předku, tedy instanci třídy `VirtualFrame`, ve které byla procedura deklarována. Takto uložená instance musí být tzv. *materializována*, jelikož nesmí zaniknout s koncem volání procedury, se kterou byla vytvořena. Materializování `VirtualFramu` je drahá operace, a proto TruffleScheme implementuje optimalizaci, která během parsování detekuje, zda procedura vyžaduje lexikálního předka. Pokud nevyžaduje, lexikální předek je nahrazen hodnotou `null`.



Obrázek 18: Pole argumentů při volání procedury. Argument na indexu `i` se získá pomocí vzorce `i + 1`

I přesto, že `CallTarget` obsahuje metodu `call`, není optimální takto procedury volat. Jakékoliv volání `CallTargetu` by mělo být provedeno skrze uzly `DirectCallNode` nebo `IndirectCallNode`.[18] Jak je patrné z názvu, výběr konkrétního uzlu závisí na stabilitě `CallTargetu`.

[17] Boxování (ang. boxing) je proces, při kterém se hodnota primitivního typu (např. `int`) převede na hodnotu objektového typu (např. `Integer`). Unboxování (ang. unboxing) je opačný proces, jedná se o převod hodnoty z objektového typu na primitivní typ.

[18] `DirectCallNode` a `IndirectCallNode` jsou součástí knihovny Truffle API.

`DirectCallNode` se používá v případě, že `CallTarget` je statický, což znamená, že se procedura vyhodnotí vždy na stejnou proceduru ve stejném *místě volání* (ang. call site). Naopak `IndirectCallNode` se používá v případě, že `CallTarget` je dynamický, a tudíž se procedura může vyhodnotit na jinou proceduru ve stejném místě volání. To je typické pro případy, kdy je procedura argumentem jiné procedury, což je v jazyku Scheme zcela běžné.

Pokud je procedura volána skrze `DirectCallNode`, platforma GraalVM je pak schopná provést následující optimalizace:

- Eliminace volání metod (ang. method inlining).
- Duplikaci abstraktního syntaktického stromu Truffle s ohledem na místo volání procedury (ang. call site sensitive AST duplication).

První zmíněná optimalizace byla popsána v kapitole 2.2.4. V Truffle kontextu se jedná o optimalizaci, kde abstraktní syntaktický strom Truffle uložený v `CallTargetu` je vložen do `CallTargetu` předka. Tato optimalizace probíhá nad Graal IR. To znamená, že ve výsledném grafu bude uzel reprezentující volání procedury nahrazen grafem reprezentující tělo eliminované procedury.

Druhá zmíněná optimalizace vytváří novou kopii `CallTargetu` v neinicializovaném stavu. Tato optimalizace se využívá, jakmile Truffle detekuje, že se procedura volá z více různých míst, kde na každém místě se využívají jiné specializace (polymorfní volání). Originální `CallTarget` pak zůstává v polymorfní podobě, ale nově vytvořená kopie se následně specializuje na základě profilujících dat z nového místa volání (je proto vysoká pravděpodobnost, že nový `CallTarget` bude monomorfní).

Naopak pokud je procedura volána skrze `IndirectCallNode`, GraalVM nemůže provést žádnou výše zmíněnou optimalizaci. Ale i přesto dochází v tomto uzlu k profilování typů vstupních argumentů, což posléze umožní eliminovat mnoho typových kontrol (například u specializací), jelikož typ jednotlivých argumentů je považován za konstantní. Stejné profilování samozřejmě probíhá i u `DirectCallNode`.

TruffleScheme využívá uzel `DispatchUserProcedureNode` k volání uživatelsky definovaných procedur, jehož implementaci je možno vidět ve zdrojovém kódu 8. Tento zdrojový kód využívá dva nové prvky, které dosud nebyly popsány. Jedná se o anotaci `@Cached` a s ní spojený atribut `limit`. Anotace `@Cached` se používá k definovaní konstanty, která je spojená s instancí dané specializace. Konstanta je vytvořena na základě výrazu, který je možno napsat do závorek k anotaci. Pokud výraz není zadán (možné vidět na řádku 25 ve zdrojovém kódu) anotace se bude snažit vyhledat statickou metodu `create()` ve třídě, jejíhož typu je vytvářená konstanta.[19] Konstanty jsou vytvořeny ještě před vyhodnocením kontroly dané specializace, a tudíž je lze použít ve vlastnosti `guards` anotace `@Specialization`. Pokud tak vývojář učiní, musí pak definovat atribut `limit`, který specifikuje, kolik instancí dané specializace se může

[19] Pokud se samozřejmě nejedná o primitivní typ, ten je vytvořen triviálně.

vytvořit. Vzniká tak stejně, jako v případě řetězení specializací, polymorfní inline cache (viz kapitola 3.4.1). Nutné zmínit, že pokud argument obsahující anotaci `@Cached` dědí z třídy `Node`, je vytvořen nový potomek (tzn. že ve vygenerovaném kódu bude vlastnost obsahovat anotaci `@Child`), naopak pokud se jedná například o argument primitivního typu, ve vygenerovaném kódu bude vlastnost obsahovat anotaci `@CompilationFinal`.

To znamená, že v případě `DispatchUserProcedureNode`, každé místo volání smí zavolat pouze tři různé procedury (jelikož `limit = 3`). Pokud je tento počet překročen, je metoda specializace `doCached` nahrazena generickou specializací `doUncached`, která je ovšem výrazně pomalejší.

```
1  public abstract class DispatchUserProcedureNode extends SchemeNode {
2
3      public abstract Object executeDispatch(Object procedure,
           Object[] arguments);
4
5      @Specialization(guards = "proc.callTarget() ==
           procCached.callTarget()", limit = "3")
6      protected static Object doCached(
7              UserDefinedProcedure proc,
8              Object[] arguments,
9              @Cached("proc") UserDefinedProcedure procCached,
10             @Cached("create(procCached.callTarget())")
                   DirectCallNode directCallNode) {
11         var argumentSize = arguments.length - 1;
12         if (argumentSize != procCached.expectedNumberOfArgs()) {
13             CompilerDirectives.transferToInterpreterAndInvalidate();
14             throw SchemeException.arityException(null,
                   procCached.name(),
                   procCached.expectedNumberOfArgs(), argumentSize);
15         }
16         return directCallNode.call(arguments);
17     }
18
19     @Specialization(replaces = "doCached")
20     protected static Object doUncached(
21             UserDefinedProcedure proc,
22             Object[] arguments,
23             @Cached IndirectCallNode indirectCallNode) {
24         return indirectCallNode.call(proc.callTarget(), arguments);
25     }
26 }
```

Zdrojový kód 8: Uzel volající uživatelsky definované procedury

5.5 Primitivní procedury

Primitivní procedury by bylo možné implementovat stejně jako uživatelsky definované procedury, aniž by to ovlivnilo korektnost interpretu. Výsledky ben-

chmarků by byly ve většině případů stejné, a to díky optimalizacím, které framework Truffle automaticky provádí (např. eliminace volání metod nebo duplikaci abstraktního syntaktického stromu Truffle). Problém by však mohl nastat, jakmile by byla interpretována větší aplikace. Každý JIT překladač má nějaký *limit* (ang. budget), který může strávit na optimalizacích. Ve většině případů se jedná o množství zdrojů (ang. resources), které překladač na optimalizace může využít. Překladač se pak snaží hledat nějaký kompromis mezi množstvím prostředků využitých na danou optimalizaci a benefity získanými touto optimalizací. Proto jsou primitivní procedury implementovány jiným způsobem, který se snaží co nejvíce JIT překladači „ulehčit“ práci.

Na rozdíl od uživatelsky definovaných procedur, primitivní procedury neobsahují `CallTarget` ve své interní reprezentaci. Obsahují tzv. *továrnu* (ang. factory), která umí vytvořit abstraktní syntaktický strom Truffle, který reprezentuje tělo primitivní procedury. Primitivní procedury jsou volány skrze uzel `DispatchPrimitiveProcedureNode`, jehož implementaci můžeme vidět ve zdrojovém kódu 9. Klíčový je zde řádek 11, který vytvoří nového potomka typu `AlwaysInlinableProcedureNode` pomocí anotace `@Cached`, což je potomek reprezentující primitivní proceduru. Důležité je podotknout, co je dosaženo touto implementací. Každé místo volání při prvním provedení primitivní procedury vytvoří nového potomka, který je následně vyhodnocen. Při opětovném volání je pak pouze provedena kontrola, zda místo volání tuto instanci primitivní procedury již vidělo a v případě že ano, dojde k pouhému vyhodnocení již vzniklého potomka. V případě, že se instance liší (jedná se o jinou primitivní proceduru), je nastaven atribut `limit` na hodnotu 2, tudíž by vznikl další potomek, který by byl následně vyhodnocen. Pokud by místo volání zaznamenalo ještě jinou primitivní proceduru, byla by specializace zneplatněna a využívala by se generická specializace `doPrimitiveProcedureUncached`, která je samozřejmě pomalejší. To znamená, že touto implementací jsme de facto dosáhli optimalizací, které by JIT překladač musel dělat v případě, že by se jednalo o klasické volání procedur. Bylo eliminováno volání primitivní procedury a zároveň došlo i k duplikaci abstraktního syntaktického stromu Truffle, jelikož v každém místě volání je vytvořeno nové tělo primitivní procedury.

```
1  public abstract class DispatchPrimitiveProcedureNode extends
       SchemeNode {
2
3      public abstract Object execute(PrimitiveProcedure procedure,
           Object[] arguments);
4
5
6      @Specialization(guards = "proc == procCached", limit = "2")
7      protected static Object doPrimitiveProcedureCached(
8              PrimitiveProcedure pr,
9              Object[] arguments,
10             @Cached("proc") PrimitiveProcedure procCached,
11             @Cached("createInlinableNode(procCached)")
                   AlwaysInlinableProcedureNode inlinedMethodNode) {
12         return inlinedMethodNode.execute(arguments);
13     }
14
15     @Specialization(replaces = "doPrimitiveProcedureCached")
16     protected static Object doPrimitiveProcedureUncached(
17             PrimitiveProcedure proc,
18             Object[] args) {
19         return proc.factory().getUncachedInstance().execute(args);
20     }
21
22
23     protected static AlwaysInlinableProcedureNode
           createInlinableNode(PrimitiveProcedure procedure) {
24         return procedure.factory().createNode();
25     }
26 }
```

Zdrojový kód 9: Uzel volající primitivní procedury

5.6 Speciální formy

Implementace speciálních forem je z teoretického hlediska velice podobná primitivním procedurám. V obou případech je eliminováno jakékoliv volání procedur. To je žádoucí, jelikož nejen že volání procedur s sebou nese značnou režii, ale při volání procedur vzniká `VirtualFrame`, kterým je reprezentovaný lexikální předek, kterého primitivní procedury a některé speciální formy vůbec nepotřebují.

TruffleScheme dokonce implementuje všechny podporované speciální formy bez nutnosti vytvoření nového `VirtualFrame`. To se na první pohled může zdát překvapivé, zejména u speciálních forem jako let nebo letrec. Tyto formy nevytvářejí `VirtualFrame`, ale místo toho využívají `VirtualFrame`, ve kterém byly vyhodnoceny. Pokud je například použita speciální forma let uvnitř těla uživatelsky definované procedury, vazby speciální formy se uloží do `VirtualFramu` této procedury. Tímto se eliminuje nutnost tvory nového `VirtualFramu`.

Speciální formy jsou tvořeny během konverze interní reprezentace do abs-

traktních syntaktických stromů Truffle. Pokud se například při konverzi narazí na list, který obsahuje jako první prvek symbol `if`, je tento list vnímán jako volání speciální formy a je vytvořen uzel `IfExprNode`.[20] Jak je možné vidět ze zdrojového kódu 10, uzel neobsahuje žádný mechanismus k detekci, zda vazba speciální formy nebyla přepsána. Důvodem je fakt, že TruffleScheme nepodporuje redefinici speciálních forem. V případě, že se o to uživatel pokusí, je vyhozena výjimka. I přesto, že by podpora redefinice nebyla příliš náročná na implementaci, benefit získaný touto podporou by byl minimální.

```
1  public abstract class IfExprNode extends SchemeExpression {
2
3      @Child private SchemeExpression condition;
4      @Child private SchemeExpression thenExpr;
5
6      public IfExprNode(SchemeExpression condition, SchemeExpression
           thenExpr) {
7          this.condition = condition;
8          this.thenExpr = thenExpr;
9      }
10
11     @Specialization
12     protected Object doIf(VirtualFrame frame,
13                           @Cached BooleanCastNode cast,
14                           @Cached("createCountingProfile()")
                                 ConditionProfile condProfile) {
15         final var conditionAsBool =
               cast.executeBoolean(condition.executeGeneric(frame));
16         if (condProfile.profile(conditionAsBool)) {
17             return thenExpr.executeGeneric(frame);
18         } else {
19             return UndefinedValue.SINGLETON;
20         }
21     }
22 }
```

Zdrojový kód 10: Uzel reprezentující speciální formu `if`, která neobsahuje náhradníka

5.7 Makra

Jak již bylo popsáno v kapitole 4.4, TruffleScheme implementuje pouze nehygienická makra, která lze definovat pomocí speciální formy `define-macro`. Tělo této speciální formy musí obsahovat výraz, který se vyhodnotí na uživatelsky definovanou proceduru. Tato procedura pak reprezentuje transformační

[20] Uzel `IfExprNode` je samozřejmě vytvořen jen v případě, že list reprezentující volání speciální formy `if` má velikost 3 (neobsahuje náhradníka). V případě, že by náhradník byl součástí listu, je vytvořen uzel `IfElseExprNode`. Rozdělení operace na více uzlů je žádoucí, jelikož je tímto eliminováno větvení, které by bylo potřeba provést za běhu programu.

proceduru makra. Jelikož není možné během konverze určit, zda se výraz vyhodnotí na proceduru, je nejprve tento výraz spolu se jménem daného makra uložen do konverzního kontextu.[21] Jakmile dojde k volání tohoto makra, je vytvořen uzel `MacroCallableExprNode`, který obsahuje jako potomka výraz, který byl pro dané makro uložený v konverzním kontextu. Implementaci je možno vidět ve zdrojovém kódu 11. Při interpretaci tohoto uzlu nejprve dojde k vyhodnocení potomka reprezentujícího transformační proceduru. V případě, že se nevyhodnotí na uživatelsky definovanou proceduru, je vyhozena výjimka. Tento případ by znamenal, že speciální forma `define-macro` neobsahuje transformační proceduru. Kontrola, zda se potomkem vyhodnotil na uživatelsky definovanou proceduru, je v tomto případě realizována pomocí anotace `@Fallback`, která obsahuje negované kontroly všech specializací definovaných v daném uzlu. V tomto případě uzel obsahuje jen jednu specializaci s implicitní kontrolou, zda potomek je typu `UserDefinedProcedure`, což znamená, že metoda `fallback` bude přijímat všechny objekty, které nejsou typu `UserDefinedProcedure`.

V případě, že se jedná o uživatelsky definovanou proceduru, je standardně volána skrze uzel `DispatchUserProcedureNode`. Výsledkem tohoto volání jsou data, která jsou pomocí třídy `InternalRepresentationConverter` převedena na abstraktní syntaktický strom Truffle. Tento strom následně nahradí stávající uzel (tj. uzel `MacroCallableExprNode`) pomocí funkce `replace` (řádek 23 ve zdrojovém kódu 11). Makro expanze tak probíhá při prvním volání makra, což znamená, že po tomto volání běhové prostředí již neobsahuje žádné známky maker. Bylo tak odstraněno i omezení týkající se využití uživatelsky definovaných vazeb během makro expanze (viz kapitola 4.4). Jelikož je makro expanze prováděna při prvním volání makra, vazby již existují, a tudíž je možné tyto vazby využít.

[21] Během konverze je vytvořen kontext, ve kterém jsou uloženy informace jako například lokální proměnné, makra nebo lexikální předek.

```java
public abstract class MacroCallableExprNode extends SchemeExpression {

    private final Object[] notEvalArgs;
    private final ConverterContext converterContext;
    private final SchemeSymbol name;
    @Child @Executed
    protected SchemeExpression transformationExpr;

    //constructor omitted

    @Specialization
    protected Object doMacroExpansion(
            VirtualFrame frame,
            UserDefinedProcedure proc,
            @Cached DispatchUserProcedureNode dispatch) {
        CompilerDirectives.transferToInterpreterAndInvalidate();
        if (proc.expectedNumberOfArgs() != notEvalArgs.length) {
            throw SchemeException.arityException(this, name.value(),
                proc.expectedNumberOfArgs(), notEvalArgs.length);
        }

        var args = getArgumentsForMacroExpansion(proc);
        var macroExpandedIR = dispatch.executeDispatch(proc, args);
        var macroExpandedAST =
            InternalRepresentationConverter.convert(macroExpandedIR,
            converterContext, false, false, null);
        return replace(macroExpandedAST).executeGeneric(frame);
    }

    @TruffleBoundary
    @Fallback
    Object fallback(Object object) {
        throw new SchemeException("""
                macro's body has to be evaluated to procedure
                expected: procedure?
                given: %s""".formatted(object), this);
    }

    //omitted for simplicity
}
```

Zdrojový kód 11: Uzel reprezentující volání makra

5.8 Optimalizace koncových volání

I přesto, že jazyk Scheme obsahuje konstrukce pro tvorbu cyklů nebo smyček, jsou tyto konstrukce často nahrazovány rekurzivním voláním procedur, protože eliminují vedlejší efekt,[22] který je častým zdrojem chyb. I když jsou oba způsoby sémanticky totožné, rekurzivně volané procedury zabírají určité místo na zásobníku. Pokud by počet rekurzivně volaných procedur byl příliš vysoký, došlo by k tzv. *přetečení zásobníku* (ang. Stack overflow), což znamená, že velikost zásobníku překročila maximální povolenou velikost. Proto většina funkcionálních jazyků implementuje tzv. *optimalizaci koncových volání* (ang. tail call optimalization), která je aplikována v případě, že procedura je volaná z tzv. *koncové pozice* lambda výrazu.

Definice 2 (Koncová pozice)
Výraz je v koncové pozici lambda výrazu Λ v případě, že splňuje alespoň jednu následující podmínku [32]:

1. Poslední výraz v těle výrazu Λ je v koncové pozici výrazu Λ.
2. Je-li `(if <test> <důsledek> <náhradník>?)` v koncové pozici výrazu Λ, pak `<důsledek>` i `<náhradník>` jsou v koncové pozici.
3. Je-li `(and <test`$_1$`> ... <test`$_n$`>)` v koncové pozici výrazu Λ, pak `<test`$_n$`>` je v koncové pozici. Obdobné platí i pro speciální formu `or`.
4. Je-li `(let (<vazba>+) <výraz`$_1$`> ... <výraz`$_n$`>)` v koncové pozici výrazu Λ, pak `<výraz`$_n$`>` je v koncové pozici. Obdobné platí i pro speciální formu `letrec`.

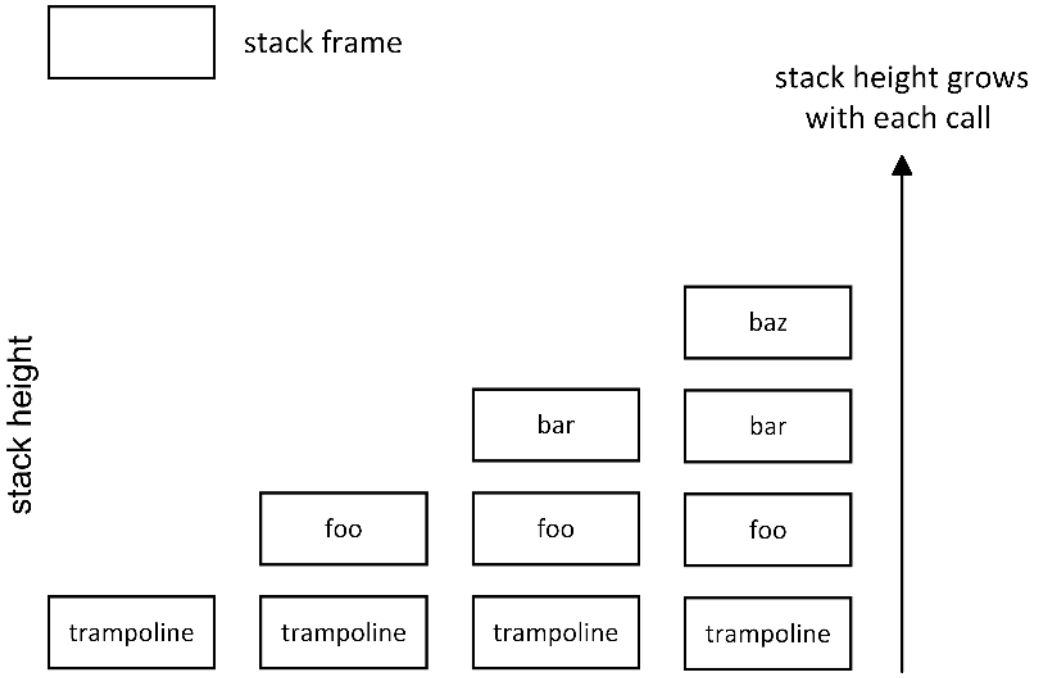
TruffleScheme implementace rozlišuje dvě varianty, které mohou nastat při aplikaci procedury v koncové pozici. Pokud aplikovaná procedura je stejná jako procedura, která ji volá (jinými slovy procedura volá sama sebe v koncové pozici), jedná se o tzv. *koncovou rekurzi*. Naopak pokud aplikovaná procedura je odlišná od procedury, která ji volá, jedná se o klasické *koncové volání*. TruffleScheme optimalizuje každou z variant odlišně.

Před popisem jednotlivých optimalizací je nejprve potřeba popsat obecnou techniku, ze které implementace vychází v obou případech. Jedná se o techniku zvanou *trampolína*. Trampolína je ve své podstatě vnější procedura, která opakovaně volá vnitřní procedury v koncové pozici. Pokud vnitřní procedura chce zavolat jinou proceduru v koncové pozici, namísto jejího zavolání je tato procedura vrácena vnější proceduře (trampolíně), která ji následně zavolá. Tímto je eliminován problém přetečení zásobníku, jelikož v tomto případě je velikost zásobníku konstantní. Rozdíl ve velikosti zásobníku je možné vidět na obrázku 19 a 20,

[22] Vedlejší efekt (též vedlejší účinek, ang. side effect) je situace, kdy funkce nebo výpočetní výraz mění i jiný stav procesu, než je návratová hodnota funkce nebo změna hodnoty parametru odkazovaného přes odkaz referencí [31].

kde je volána procedura `trampoline`, která reprezentuje onu vnější proceduru (trampolínu). Procedury `foo`, `bar` a `baz` pak reprezentují vnitřní procedury, které postupně budou volány procedurou `trampoline`. Implementaci procedury `trampoline` je možné vidět ve zdrojovém kódu 12. [33]

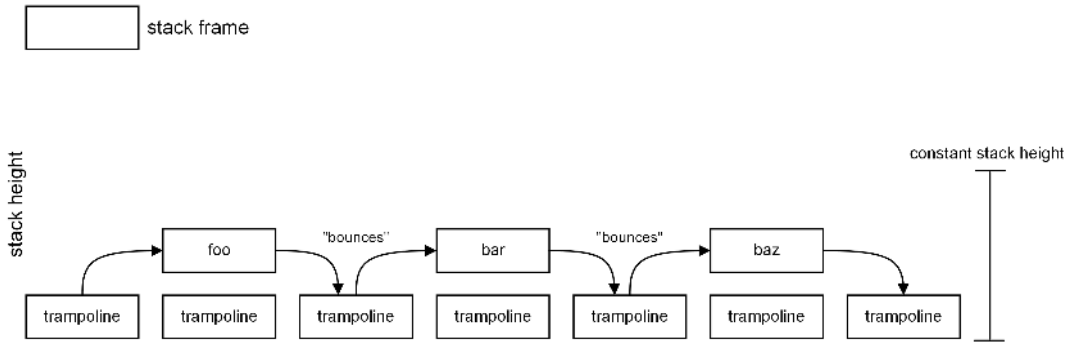
Jak již bylo zmíněno na začátku této kapitoly, TruffleScheme rozlišuje mezi koncovou rekurzí a koncovým voláním. V následující kapitole bude nejprve popsána implementace koncové rekurze, kterou je možné chápat jako speciální případ koncového volání. Poté bude následovat kapitola popisující implementaci koncového volání.



Obrázek 19: Velikosti zásobníku roste s každým dalším voláním procedury v případě, že není aplikována optimalizace koncových volání [7]

5.8.1 Implementace optimalizace koncové rekurze

Detekce koncové rekurze probíhá během konverze interní reprezentace na abstraktní syntaktické stromy Truffle. Pokud je zjištěno, že posledním výrazem v těle právě definované procedury je volání stejnojmenné procedury, je toto volání nahrazeno speciálním uzlem `TailRecursiveThrowerNode`, jehož implementaci je možné vidět ve zdrojovém kódu 13. Z této implementace je možné vidět, že nejprve jsou vyhodnoceny všechny argumenty procedury, poté jsou uloženy do `VirtualFramu` a následně je vyhozena výjimka `TailRecursiveException`. Tato výjimka je potomkem třídy `ControlFlowException`, což je speciální typ výjimky, kterou překladač Graal ve zkompilovaném kódu dokáže odstranit.



Obrázek 20: Velikost zásobníku je konstantní s každým dalším voláním procedury, v případě, že je aplikována optimalizace koncových volání [7]

Dále může být překvapivé, že výjimka neobsahuje žádná data, jako například proceduru, kterou vnější procedura bude volat. Je důležité si uvědomit, že díky tomu, že se jedná o koncovou rekurzi, je tato procedura známa. Jedná se o stejnou proceduru, která je právě volána.

Výjimka je následně odchycena v uzlu `TailRecursiveLoopNode`, který implementuje rozhraní `RepeatingNode`. Třídy implementující toto rozhraní se používají k implementaci smyček. Překladač Graal aplikuje na tyto uzly optimalizaci zvanou *On-Stack Replacement* (dále OSR), která umožňuje nahradit interpretované tělo smyčky strojovým kódem již během vykonávání dané smyčky. To je žádoucí, jelikož bez OSR optimalizace by se nejprve muselo počkat na skončení smyčky a až poté by tělo mohlo být nahrazeno strojovým kódem. OSR optimalizace je proto velice důležitou pro koncovou rekurzi, jelikož v tomto případě by strojový kód těla rekurzivní procedury mohl být nahrazen až po skončení rekurze, což v některých případech může trvat opravdu dlouho. Mělo by být z popisu zřejmé, že koncová rekurze je transformována na smyčku, jejíž tělo je nahrazeno ihned, jakmile je k dispozici optimalizovaný strojový kód. Tímto je odstraněna jakákoliv režie spojená s voláním procedury, jelikož samotné volání bylo nahrazeno smyčkou, obsahující tělo rekurzivní procedury.

```
1  (define trampoline
2    (lambda ()
3      (foo)))
4
5  (define foo
6    (lambda ()
7      (bar)))
8
9  (define bar
10   (lambda ()
11     (baz)))
12
13 (define baz
14   (lambda ()
15     10))
16
17 (trampoline)
```

Zdrojový kód 12: Procedura `trampoline` v tomto případě bude sloužit jako trampolína (vnější procedura), která postupně bude volat vnitřní procedury `foo`, `bar` a `baz`

```java
public abstract class TailRecursiveThrowerNode extends
    SchemeExpression {

    @Children
    private final SchemeExpression[] arguments;

    @Children
    private final WriteFrameSlotNode[] writeFrameSlotNodes;


    public TailRecursiveThrowerNode(List<SchemeExpression>
        arguments, List<WriteFrameSlotNode> writeFrameSlotNodes) {
        this.arguments = arguments.toArray(SchemeExpression[]::new);
        this.writeFrameSlotNodes =
            writeFrameSlotNodes.toArray(WriteFrameSlotNode[]::new);
    }

    @Specialization
    protected Object doThrow(VirtualFrame frame) {
        prepareArguments(frame);
        throw TailRecursiveException.INSTANCE;
    }


    @ExplodeLoop
    private void prepareArguments(VirtualFrame frame) {
        Object[] evalArgs = new Object[arguments.length];

        for (int i = 0; i < arguments.length; i++) {
            evalArgs[i] = arguments[i].executeGeneric(frame);
        }

        for (int i = 0; i < writeFrameSlotNodes.length; i++) {
            writeFrameSlotNodes[i].executeWrite(frame, evalArgs[i]);
        }
    }
}
```

Zdrojový kód 13: Uzel, který místo volání procedury vyhodí výjimku

5.8.2 Implementace optimalizace koncových volání

Implementace optimalizace koncových volání je velice podobná předchozí implementaci. Mělo by být zřejmé, že implementace optimalizace koncových volání je o něco pomalejší než implementace na koncovou rekurzi, jelikož koncovou rekurzi lze považovat za speciální případ koncového volání. K detekci koncového volání opět dochází během konverze interní reprezentace na abstraktní syntaktické stromy Truffle. V tomto případě je volání procedury v koncové pozici nahrazeno uzlem `TailCallThrowerNode`, který opět místo volání procedury vyhodí výjimku. V tomto případě se jedná o výjimku `TailCallException`, která obsahuje uživatelsky definovanou proceduru, kterou bude vnější procedura volat s vyhodnocenými argumenty. Stejně jako v předchozím případě je výjimka potomkem třídy `ControlFlowException`.

Největší rozdíl se pak nachází v místě odchycení výjimky, konkrétně v uzlu `CallableExprNode`, kde v případě, že se jedná o koncové volání dojde k jeho nahrazení uzlem `TailCallCatcherNode`. Toto rozhodnutí nelze provést během konverze, jelikož při konverzi není jisté, zda procedura bude obsahovat volání procedury z koncové pozice či ne (např. volaná procedura ještě není definovaná). To znamená, že nelze předem rozhodnout, zda vytvořit uzel pro běžné volání procedury (tj. `CallableExprNode`), nebo uzel `TailCallCatcherNode`, který slouží jako vnější procedura (trampolína) při koncovém volání. Je důležité poznamenat, že toto nahrazení uzlů nemá téměř žádný vliv na rychlost interpretu, protože již po prvním volání procedury, v případě, že se jedná o proceduru s koncovým voláním, je uzel nahrazen a další volání této procedury již není nijak ovlivněno.

Uzel `TailCallCatcherNode` obsahuje potomka typu `TailCallLoopNode`, který pak opět transformuje koncové volání na smyčku. Jediným rozdílem je fakt, že v těle této smyčky jsou volány vnitřní procedury, které jsou vždy součásti odchycené výjimky. Tento rozdíl však může mít zásadní dopad na výkon aplikace. Jak již bylo zmíněno (viz kapitola 5.4), všechny procedury jsou volány skrze uzel `DispatchUserProcedureNode`, který obsahuje polymorfní inline cache. To znamená, že v případě, kdy vnější procedura zavolá více různých procedur, než je velikost cache, uzel se dostane do generického stavu a nebude moct volání procedur optimalizovat. Je proto nutné najít kompromis mezi velikostí cache a rychlostí aplikace. V případě, že velikost cache bude příliš velká, bude vygenerováno příliš mnoho strojového kódu. Naopak v případě, že velikost cache bude malá, uzel `DispatchUserProcedureNode` se dostane do generického stavu a nebude možné provádět další optimalizace. TruffleScheme implementace má nastavenou velikost této polymorfní inline cache na 3, což se ukázalo jako nejlepší kompromis mezi rychlostí interpretu a velikosti vygenerovaného strojového kódu.[23]

[23]Tato hodnota byla určena na základě vybraných benchmarků. Nejedná se o dost velký reprezentativní vzorek k určení, zda je tato hodnota vhodná pro reálné aplikace.

6 Podpora interoperability

GraalVM spolu s frameworkem Truffle umožňuje běh několika programovacích jazyků ve stejném běhovém prostředí. Hostované jazyky jsou nejprve převedeny do abstraktních syntaktických stromů Truffle, které jsou následně vyhodnocovány. Tento přístup umožňuje mít hned několik abstraktních syntaktických stromů Truffle z různých jazyků ve stejném paměťovém prostoru. K tomu, aby si jednotlivé jazyky mohly předávat data mezi sebou, byl vyvinut speciální protokol, který definuje zprávy, které vývojář jazyka musí implementovat pro všechny objekty interní reprezentace. Jednotlivé zprávy protokolu lze volat pomocí tzv. *Truffle knihoven* (ang. Truffle Libraries), které jazykovým implementacím umožňují provádět *polymorfní volání* (ang. polymorphic dispatch) na základě datového typu příjemce zprávy.[24] Například implementace jazyka JavaScript nad platformou GraalVM obsahuje okolo 20 různých implementací pole, což znamená že uzly pracující s polem musí teoreticky obsahovat minimálně 20 specializací, jelikož s každou implementací pole může pracovat jinak. Velký počet specializací není jediným problémem při práci s velkým počtem typů objektů. Dalšími problémy mohou například být [34]:

- Přidání (nebo změna implementace) nového typu pole vyžaduje úpravu všech operací využívajících dané pole.
- Dochází k porušení principu zapouzdření, jelikož mnoho implementačních detailů jednotlivých interních reprezentací musí být k dispozici k provedení operace. Například pro čtení z pole je potřeba vědět, jakou datovou strukturou je pole reprezentováno.
- Nové implementace nemohou být dynamicky načteny, jelikož typy těchto implementací musí být známy v době překladu.

Všechny tyto nedostatky byly hlavní motivací k tvorbě Truffle knihoven. Detailní popis, jak Truffle knihovny fungují, by byl nad rámec této diplomové práce, proto následující kapitoly popisují jen prvky nutné k implementaci podpory interoperability.

6.1 Použití InteropLibrary knihovny

Výše zmíněný protokol je implementovaný pomocí knihovny zvané *InteropLibrary*, která obsahuje přibližně 150 zpráv, které slouží k identifikaci a k práci s daným objektem. Jedná se o zprávy jako například `fitsInInt`, `fitsInDouble`, `getArraySize` nebo `isExecutable`. To znamená, že každý objekt interní reprezentace jednotlivých jazykových implementací (které interoperabilitu chtějí podporovat), musí využívat tuto knihovnu, aby ostatní jazyky uměly s tímto objektem pracovat. Ve zdrojovém kódu 14 je ukázka použití této knihovny nad

[24] V případě, že čtenář zná (staticky typované) *typové třídy* (ang. type-classes), je možné si Truffle knihovny představit jako „dynamicky typované typové třídy“.

interní reprezentací primitivní procedury. Nejprve je potřeba umístit anotaci `@ExportLibrary` nad danou třídu a specifikovat, jaké knihovny interní reprezentace bude podporovat. V tomto případě se jedná pouze o knihovnu InteropLibrary. Implementace primitivní procedury exportuje čtyři zprávy, kde zprávy `hasLanguage` a `getLanguage` jsou spíše informativní k identifikaci, z jakého jazyka objekt pochází. Zbylé dvě zprávy slouží k volání primitivní procedury. Aby bylo možné exportovat zprávu `execute`, je potřeba nejprve exportovat zprávu `isExecutable`, aby bylo možné zkontrolovat (např. v kontrolách specializací), zda se jedná o objekt, který může být volán. Implementace zprávy `execute` je pak přímočará. Obsahuje pole argumentů, se kterými bude primitivní procedura volána a definuje dva potomky, pomocí anotace `@Cached`, které jsou potřeba k zavolání primitivní procedury.

```
1  @ExportLibrary(InteropLibrary.class)
2  public record PrimitiveProcedure(
3          String name,
4          NodeFactory<? extends AlwaysInlinableProcedureNode> factory
5  ) implements TruffleObject {
6
7      @ExportMessage
8      boolean hasLanguage() {
9          return true;
10     }
11
12     @ExportMessage
13     Class<? extends TruffleLanguage<?>> getLanguage() {
14         return SchemeTruffleLanguage.class;
15     }
16
17     @ExportMessage
18     boolean isExecutable() {
19         return true;
20     }
21
22     @ExportMessage
23     Object execute(Object[] arguments,
24                    @Cached ForeignToSchemeNode toSchemeNode,
25                    @Cached DispatchPrimitiveProcedureNode
                          dispatchNode) {
26         var args = convertToSchemeValues(arguments, toSchemeNode);
27         return dispatchNode.execute(this, args);
28     }
29 }
```

Zdrojový kód 14: Interní reprezentace primitivní procedury, využívající InteropLibrary knihovnu

6.2 Podpora objektů z cizích jazyků

Předchozí kapitola se zabývala využitím InteropLibrary knihovny nad třídami TruffleScheme, které slouží jako interní reprezentace. To znamená, že cizí jazyky jsou nyní schopny vyhodnotit jakýkoliv Scheme kód a následně s výsledkem pracovat. Tato kapitola se bude zabývat problematikou podpory objektů z cizích jazyků, které také musí implementovat InteropLibrary knihovnu. To znamená, že operace musí podporovat nejen typy objektů interní reprezentace jazyka Scheme, ale i typy objektů interní reprezentace cizích jazyků. Podpora cizích objektů by nebyla téměř možná bez Truffle knihoven, jelikož není možné znát všechny typy interních reprezentací cizích jazyků v době překladu (viz zmíněné nedostatky v kapitole 6).

Implementace bude opět demonstrována na příkladu. Ve zdrojovém kódu 15 je možné vidět implementaci operace `car`. První dvě specializace implementují logiku operace pro objekty interní reprezentace jazyka Scheme. Třetí specializace pak slouží k podpoře objektů z cizích jazyků. Prvním argumentem této specializace je příjemce zprávy, který je typu `Object`, jelikož dopředu není možné znát typ tohoto objektu. Druhým argumentem je pak InteropLibrary knihovna, která bude ve vygenerovaném kódu potomkem tohoto uzlu. Všechny Truffle knihovny se vytváří pomocí anotace `@CachedLibrary`,[25] která jako hodnotu vyžaduje příjemce zprávy, na základě kterého se následně bude specializovat. V případě, že se příjemce změní, tzn. bude ze stejného místa volání zavolána procedura `car` s jiným typem objektu, dochází opět ke tvorbě polymorfní inline cache. Velikost této cache je opět stanovena atributem `limit` ve specializaci. To znamená, že každá specializace využívající Truffle knihovny musí definovat atribut `limit`,[26] jelikož musí specifikovat, kolik typů příjemců má specializace přijímat, než se knihovna dostane do generického stavu. Specializace dále využívá `InteropLibrary` knihovnu ke kontrole, zda příjemce reprezentuje strukturu obsahující elementy (může se jednat například o strukturu reprezentující pole nebo list). V takovém případě se stane specializace aktivní a vykoná se její tělo. Tímto způsobem je potřeba rozšířit každý uzel reprezentující Scheme operaci alespoň o jednu specializaci, která bude podporovat objekty z cizích jazyků.

[25] Nejedná se o jediný způsob, jakým je možné vytvářet Truffle knihovny. Existují mezní případy, kdy je potřeba knihovnu vytvořit skrze její vygenerovanou továrnu.

[26] Existuje typ Truffle knihoven nazývaný *Dispatch knihovny* (ang. Dispatch Libraries), kde atribut `limit` není potřeba uvádět. Jelikož tento typ knihovny nebyl v TruffleScheme implementaci použit, není zde popsán. Zájemci se o tomto typu knihovny mohou dozvědět více na https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/library/CachedLibrary.html.

```java
public abstract class CarCoreNode extends SchemeNode {

    public abstract Object execute(Object object);

    @Specialization(guards = "!list.isEmpty")
    protected Object doList(SchemeList list) {
        return list.car;
    }

    @Specialization
    protected Object doPair(SchemePair pair) {
        return pair.first();
    }

    @Specialization(guards = "interop.hasArrayElements(receiver)",
                    limit = "getInteropCacheLimit()")
    protected Object doForeignObject(
            Object receiver,
            @CachedLibrary("receiver") InteropLibrary interop,
            @Cached ForeignToSchemeNode foreignToSchemeNode,
            @Cached TranslateInteropExceptionNode
                translateException) {
        final var foreign = readForeignArrayElement(receiver, 0,
            interop, translateException);
        return foreignToSchemeNode.executeConvert(foreign);
    }
    //some specializations omitted for simplicity
}
```

Zdrojový kód 15: Implementace operace `car` podporující objekty z cizích jazyků

I přesto mohou být některé procedury náročné na implementaci, aby byla zachována správná sémantika operace. Například k určení rovnosti dvou procedur pomocí primitivní procedury `equal?` je nezbytné znát implementační detaily cizího jazyka. V těchto případech byla zvolena nejjednodušší možná implementace, jelikož hlavním cílem práce byla demonstrace implementace interoperability. Proto například výše zmíněná procedura `equal?` porovnává pouze reference u objektů, které pochází z cizích jazyků.

6.3 Polyglotní API

V tuto chvíli je TruffleScheme schopný pracovat s objekty pocházejícími z cizích jazyků a zároveň umožňuje cizím jazykům pracovat s objekty pocházejícími z jazyka Scheme. Pro umožnění polyglotního programování je potřeba rozšířit jazyk Scheme o několik primitivních procedur a speciálních forem, které umožní například vyhodnotit kód cizích jazyků nebo číst jejich globální vazby. V této kapitole jsou popsány všechny primitivní procedury a speciální formy spolu s jejich gramatikami, které byly přidány do jazyka Scheme.

Jelikož v jazyce Scheme neexistují třídy, jako tomu je v objektově orientovaných jazycích, nebylo by se stávající implementací možné například zjistit, jaké vlastnosti daný objekt obsahuje. Bylo proto vytvořeno generické API, které umožní snadnou práci s objekty, které pocházejí z cizích jazyků. Jedná se většinou o vestavěné primitivní procedury, které jsou automaticky definovány v globálním prostředí při startu interpretu. Nově přidané primitivní procedury a jejich gramatiky jsou:

- **eval-source**: slouží k vyhodnocení kódu cizího jazyka.
  - `(eval-source <id-jazyka> <kód>)`
- **read-global-scope**: slouží ke čtení globálních vazeb cizího jazyka.
  - `(read-global-scope <id-jazyka> <jméno>)`
- **has-members?**: slouží k zjištění, zda objekt obsahuje nějaké vlastnosti
  - `(has-members? <objekt>)`
- **member-readable?**: slouží k zjištěním, zda vlastnost lze číst. Zároveň primitivní procedury `has-members?` a `member-existing?` musí vracet pravdu.
  - `(member-readable? <objekt> <vlastnost>)`
- **member-modifiable?**: slouží k zjištění, zda vlastnost lze modifikovat. Zároveň primitivní procedury `has-members?` a `member-existing?` musí vracet pravdu.
  - `(member-modifiable? <objekt> <vlastnost>)`
- **member-insertable?**: slouží k zjištění, zda danou vlastnost lze přidat. Zároveň primitivní procedura `has-members?` musí vracet pravdu a primitivní procedura `member-existing?` musí vracet nepravdu.
  - `(member-insertable? <objekt> <vlastnost>)`
- **member-removable?**: slouží k zjištění, zda vlastnost lze smazat. Zároveň primitivní procedury `has-members?` a `member-existing?` musí vracet pravdu.
  - `(member-removable? <objekt> <vlastnost>)`
- **member-invocable?**: slouží k zjištění, zda vlastnost obsahuje proceduru. Zároveň primitivní procedury `has-members?` a `member-existing?` musí vracet pravdu.
  - `(member-invocable? <objekt> <vlastnost>)`

- **member-writable?**: slouží k ověření, zda je možné vytvořit danou vlastnost přidat (v případě, že neexistuje), nebo upravit (v případě, že existuje). Zároveň alespoň jedna z primitivních procedur `member-modifiable?`, nebo `member-insertable?` musí vracet pravdu.
  - `(member-writable? <objekt> <vlastnost>)`
- **member-existing?**: slouží k zjištění, zda daná vlastnost existuje
  - `(member-existing? <objekt> <vlastnost>)`
- **get-members**: vrací pole obsahující jména všech vlastností objektu
  - `(get-members <objekt>)`
- **read-member**: vrací hodnotu uloženou v dané vlastnosti
  - `(read-member <objekt> <vlastnost>)`
- **write-member**: zapíše hodnotu do vlastnosti objektu
  - `(write-member <objekt> <vlastnost> <hodnota>)`
- **remove-member**: odstraní vlastnost z objektu
  - `(remove-member <objekt> <vlastnost>)`
- **invoke-member**: zavolá proceduru uloženou ve vlastnosti s poskytnutými argumenty
  - `(invoke-member <objekt> <vlastnost> <args>*)`

kde:

- `<id-jazyka>` se musí vyhodnotit na řetězec reprezentující identifikátor jazyka. Například identifikátor `js` značí jazyk JavaScript, identifikátor `python` značí jazyk Python nebo identifikátor `scm` značí jazyk Scheme.
- `<kód>` se musí vyhodnotit na řetězec obsahující kód, který se má vyhodnotit.
- `<vlastnost>` se musí vyhodnotit na řetězec obsahující jméno vlastnosti. Toto jméno je možné získat pomocí primitivní procedury `get-members`.
- `<objekt>` se musí vyhodnotit na objekt, který implementuje *InteropLibrary* knihovnu.

Dále byly přidány dvě speciální formy, které usnadňují základní práci s cizími objekty. Inspirací pro přidání těchto speciálních forem byl jazyk Clojure, který podporuje interoperabilitu s jazykem Java. Jelikož se jedná o speciální formy, tak identifikátory jednotlivých vlastností se nevyhodnocují, tudíž identifikátory nejsou řetězce, ale symboly. Nově přidané speciální formy a jejich gramatiky jsou:

- **set-value!**: modifikuje hodnotu dané vlastnosti. Od primitivní procedury `write-member` se liší v tom, že vyhodí výjimku v případě, že vlastnost neexistuje.
  - `(set-value! <symbol> <objekt> <args>*)`
- **.**: v případě, že vlastnost obsahuje proceduru, je zavolána. Jinak přečte hodnotu vlastnosti.
  - `(. <symbol> <objekt> <args>*)`

7 Srovnání s ostatními implementacemi

Tato kapitola obsahuje výkonnostní srovnání implementace TruffleScheme s implementací Racket (dříve MzScheme) a Guile. Měření rychlosti TruffleScheme bylo provedeno pomocí knihovny *Java Microbenchmark Harness* (dále JMH), která se běžně využívá k měření rychlosti Java aplikací. Pro měření rychlosti Racket a Guile byly využity primitivní procedury `current-milliseconds` a `times`.[27] Před každým měřením bylo provedeno 25 iterací, které slouží k zahřátí jednotlivých virtuálních strojů. Následně bylo provedeno 20 iterací, z jejichž výsledků byl vypočítán průměr, který byl zaokrouhlen. Výsledky měření jednotlivých implementací je možné vidět v tabulce 2. Všechny benchmarky byly provedeny na procesoru M1 Max od společnosti Apple s pamětí RAM 32GB.

	Guile	**Racket**	**TruffleScheme**	**TruffleScheme (SubstrateVM)**
fibonacci	174	51	49	25
factorial	20	12	6	7
quicksort	54	41	29	30
tak	45	12	27	6

Tabulka 2: Výsledky měření jednotlivých implementací v milisekundách

7.1 Srovnání s Guile implementací

Guile od verze 3.0.0 přidala do svého virtuálního stroje JIT překladač, což vedlo ke čtyřnásobnému zrychlení existujících programů [35]. K měření proto byla využita verze 3.0.9, která je nejaktuálnější verzí v době psaní diplomové práce. Z teoretického hlediska je implementace TruffleScheme velmi podobná implementaci Guile, protože obě nejprve interpretují kód a poté aplikují JIT kompilaci na často používané procedury. Existuje mezi nimi však drobný rozdíl. Guile nejprve překládá kód do mezikódu, který je následně interpretován, zatímco TruffleScheme překládá kód do abstraktních syntaktických stromů Truffle, které jsou vyhodnoceny.

Na obrázku 21 jsou zobrazeny výsledky měření. Graf zobrazuje, zda je TruffleScheme rychlejší, či pomalejší v porovnání s Guile. Například TruffleScheme je 3.6krát rychlejší při výpočtu n-tého čísla Fibonacciho řady ve srovnání s Guile. Obecně vyššího výkonu dosahuje TruffleScheme ve všech benchmarcích. Primárním důvodem je způsob, jakým obě implementace nakládají s daty, které je možné získat z běhu programu. Zatímco TruffleScheme (a obecně jazyky implementované nad platformou GraalVM) využívá co nejvíce dat z běhu programu, Guile

[27] TruffleScheme také obsahuje primitivní proceduru `current-milliseconds`. Jelikož rozdíly v měření pomocí JMH knihovny nebo pomocí této procedury byly téměř zanedbatelné, bylo usouzeno, že toto měření je dostatečně spolehlivé, a proto bylo využito při testování rychlosti Racket a Guile implementace.

nepoužívá žádná data. Výjimkou je počet volání jednotlivých procedur, aby bylo možné rozhodnout, zda je procedura vhodným kandidátem k JIT kompilaci [36]. Takovému typu JIT překladače se často říká *Template-Based JIT Compiler*. Jelikož pro přeložení do strojového kódu není potřeba žádná analýza mezikódu, je překlad výrazně rychlejší v porovnání s překladačem Graal. Naopak, jak je možné vidět z výsledku měření, výsledný strojový kód není tak efektivní, jelikož nedokáže provádět tak sofistikované optimalizace.

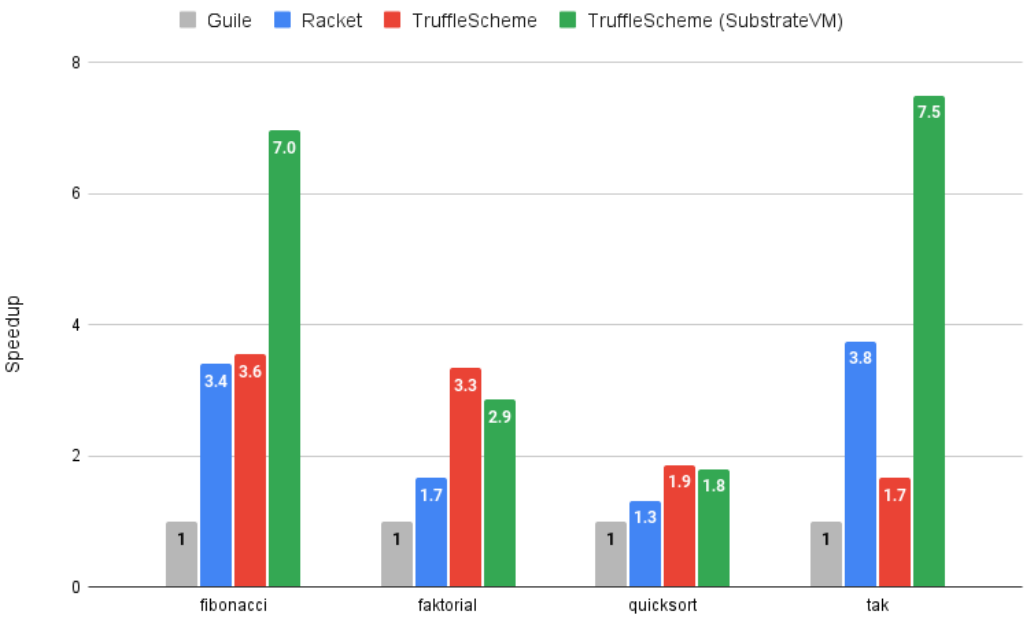
7.2 Srovnání s Racket implementací

Racket (dříve známý jako MzScheme) je považován za potomka jazyka Scheme. Jeho syntaxe je tak téměř identická s jazykem Scheme. Podobně jako Guile, Racket také obsahuje virtuální stroj včetně JIT překladače, nicméně během JIT kompilace využívá data z běhu programu, čímž se ještě více podobá implementaci TruffleScheme. Pro měření byla použita Racket verze 8.8, což je nejaktuálnější verze dostupná v době psaní diplomové práce. Jak je možné vidět z grafu 21, rychlosti jednotlivých implementací jsou daleko podobnější než v předchozím srovnání. Primárním důvodem je již zmíněný JIT překladač, který v obou implementacích využívá data z běhu programu.

Největší zrychlení je možné vidět v benchmarku *faktorial*, kde je TruffleScheme 2krát rychlejší. Hlavním důvodem je optimalizace koncové rekurze, která v implementaci TruffleScheme eliminovala volání rekurzivní procedury, tudíž eliminovala režii spojenou s voláním procedury. Racket pravděpodobně nerozlišuje mezi optimalizací na koncovou rekurzi a optimalizací na koncové volání, jelikož vygenerovaný strojový kód obsahuje volání rekurzivní procedury.

Z grafu je dále možné vidět, že benchmark `tak` je výrazně pomalejší v porovnání s Racket implementací. Možný důvod je vidět na obrázku 22, kde je vizualizována interní reprezentace (Graal IR), která se využívá k určení, jaké optimalizace byly na zdrojový kód aplikovány v různých fázích překladu. V tomto případě obrázek reprezentuje tělo metody `tak` (viz zdrojový kód 16), kde je možné vidět, že koncová rekurze byla transformována na smyčku (uzel 362 reprezentuje začátek smyčky, uzel 2402 ukončení smyčky a uzel 2405 ukončení jedné iterace). Dále je možné vidět, že překladač nebyl schopen eliminovat boxování argumentů (např. uzly 2681, 2682, 2683), které jsou následně uloženy v poli typu `Object[]` a předány k rekurzivnímu volání. Právě tyto uzly jsou důvodem, proč je benchmark pomalejší. Eliminace těchto uzlů ale není v tuto chvíli možná, jelikož se jedná o omezení JVM.

Problém s boxováním lze ale vyřešit pomocí technologie *SubstrateVM*. Interprety jazyků implementovaných nad platformou GraalVM mohou být přeloženy přímo do strojového kódu, tudíž není potřeba JVM pro jejich spuštění. Jedná se o tzv. *ahead-of-time kompilaci* (AOT), jejímž výsledkem je tzv. *native-image*. JVM je zde nahrazeno SubstrateVM obsahující komponenty, které jsou běžně součástí běhového prostředí JVM, jako např. deoptimalizátor, správce paměti nebo správce vláken. Tyto komponenty jsou k dispozici během překladu, a tudíž

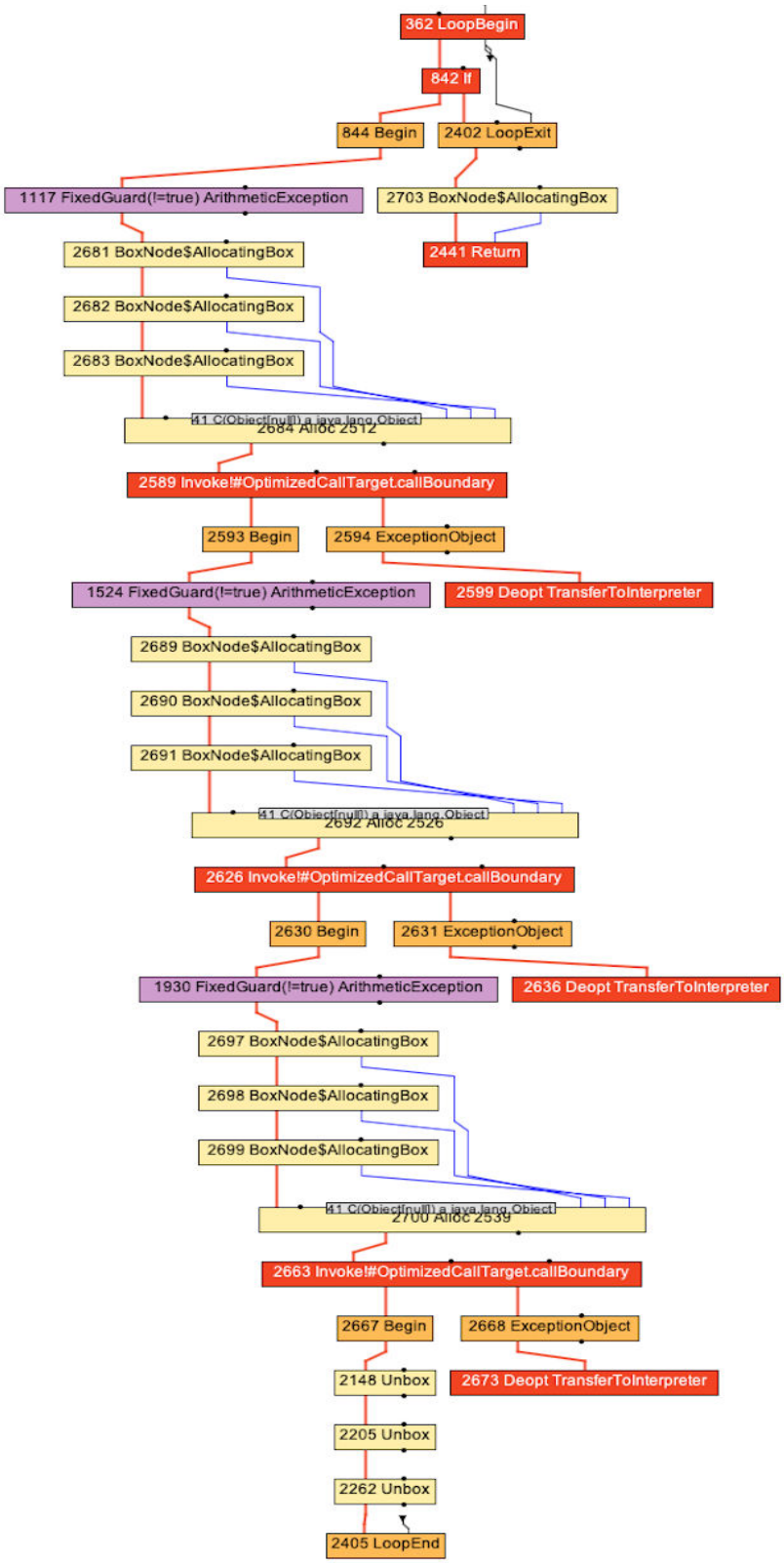


Obrázek 21: Výkonnostní srovnání Guile, Racket a TruffleScheme. TruffleScheme je možné buďto spustit nad JVM, nebo využít technologii SubstrateVM

jsou součástí výsledného strojového kódu. Jelikož byly tyto komponenty tvořeny pro interpretaci jazyků, odstraňují některá omezení JVM. Jedním z nich je výše zmíněné boxování argumentů při volání procedur. Na grafu 21 je možné vidět srovnání implementace TruffleScheme, která je přeložena do strojového kódu a využívá tedy technologii SubstrateVM, s ostatními implementacemi. Benchmarky založené na rekurzivním voláním procedur (výpočet n-tého čísla Fibonacci řady a Takeuchi funkce) dosáhly výrazného zrychlení, jelikož v obou případech bylo eliminováno boxování jednotlivých argumentů.

```
1  (define tak
2    (lambda (x y z)
3      (if (not (< y x))
4          z (tak (tak (- x 1) y z)
5                 (tak (- y 1) z x)
6                 (tak (-  1) x y)))))
```

Zdrojový kód 16: Implementace Takeuchi funkce v jazyce Scheme



Obrázek 22: Interní reprezentace (Graal IR) těla procedury `tak`

Závěr

Cílem práce bylo vytvořit interpret funkcionálního jazyka na platformě GraalVM. Vznikl tak TruffleScheme, interpret jazyka Scheme, který svou rychlostí ve vybraných benchmarcích překonává implementace jako Guile nebo Racket. TruffleScheme je jedna z prvních implementací na platformě GraalVM, která poskytuje optimalizaci koncových volání a optimalizaci pro koncovou rekurzi bez negativního dopadu na rychlost interpretu. Naopak implementace optimalizace pro koncovou rekurzi interpret výrazně zrychlila, jelikož eliminovala volání rekurzivní procedury, čímž odstranila režii spojenou s jejím voláním.

Ačkoliv TruffleScheme není kompletní implementací jazyka Scheme, výsledky benchmarků dokazují, že platforma GraalVM je nejen vhodná pro kompletní implementaci jazyka Scheme, ale i pro ostatní funkcionální jazyky. TruffleScheme je v průměru 2.6krát rychlejší v porovnání s Guile implementací a 1.2krát rychlejší v porovnání s Racket implementací. Nejlepších výsledků bylo dosaženo při použití technologie SubstrateVM, kde TruffleScheme je v průměru 4.7krát rychlejší než Guile implementace a 1.8krát rychlejší než Racket implementace.

Dalším důležitým aspektem implementace byla podpora interoperability s dalšími jazyky dostupnými na platformě GraalVM, čímž TruffleScheme umožňuje polyglotní programování. Pro dosažení tohoto cíle bylo nutné navrhnout generické aplikační rozhraní, které umožňuje například vyhodnocovat kód cizích jazyků nebo číst vazby definované v globálním prostředí těchto jazyků. Rychlost této implementace byla záměrně ponechána pro potenciální budoucí zlepšení.

Cílem tohoto textu bylo nejprve teoreticky vysvětlit, na jakém principu platforma GraalVM spolu s frameworkem Truffle pracuje, a poté popsat implementaci základních prvků jazyka. Text mimo jiné obsahuje mnoho implementačních detailů, včetně konkrétních příkladů, které mohou sloužit jako podklad pro budoucí vývoj jazyků na platformě GraalVM.

Conclusions

This thesis aimed to develop a functional language interpreter on the GraalVM platform. The result is TruffleScheme, which is a Scheme language interpreter that surpasses implementations such as Guile or Racket in selected benchmarks. TruffleScheme is one of the first implementations on the GraalVM platform that supports Tail Call and Tail Recursive optimizations without negatively affecting the speed of the interpreter. Conversely, the implementation of Tail Recursive optimization has significantly sped up the interpreter by eliminating the recursive call, thus eliminating the overhead associated with the call.

Although TruffleScheme is not a complete implementation of the Scheme language, the benchmark results show that the GraalVM platform is not only suitable for a complete implementation but also for other functional languages. TruffleScheme is on average 2.6 times faster than the Guile implementation and 1.2 times faster than the Racket implementation. The best results have been achieved using SubstrateVM, where TruffleScheme is on average 4.7 times faster than the Guile implementation and 1.8 times faster than the Racket implementation.

Another important aspect of this thesis was to support interoperability with languages implemented on the GraalVM platform, which would enable polyglot programming. To achieve this goal, it was necessary to design and implement a new generic API that allows, for example, to evaluate the code of foreign languages or read bindings defined in the global environment of these languages. The speed of this implementation was defined as an explicit non-goal that was intentionally left for potential future improvements.

The goal of this text was first to explain the theoretical principles on which the GraalVM and Truffle framework works and then to describe the implementation of basic elements of the language. This text contains many implementation details, including examples, which can serve as a solid foundation for further language development on the GraalVM platform.

A Obsah elektronických dat

Elektronická data odevzdaná v systému katedry mají následující strukturu:

`text/`
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu (případně v ZIP archivu), tj. zdrojový text textu a příloh, vložené obrázky, apod.

`README.md`
: Instrukce pro sestavení a spuštění interpretu TruffleScheme, včetně všech požadavků pro jeho bezproblémový provoz.

`bin/`
: Binární soubory pro spuštění interpretu pro operační systém Linux (amd64) a macOS (aarch64). Dále složka obsahuje soubor `scm-component.jar`, který je možné využít pro instalaci interpretu jako komponenty pomocí Graal Updater.

`src/`
: Kompletní zdrojový kód interpretu TruffleScheme, rovněž dostupný na https://github.com/horakivo/TruffleScheme.

`examples`
: Zdrojové kódy, které byly použity pro srovnání rychlosti s ostatními implementacemi v kapitole 7 této práce.

Literatura

[1] Hickey, Rich. The Clojure programming language. In. *The Clojure programming language*. 2008, s. 1. Dostupný také z: ⟨http://dx.doi.org/10.1145/1408681.1408682⟩.

[2] Odersky, Martin; Altherr, Philippe; Cremet, Vincent aj. An Overview of the Scala Programming Language. 2008.

[3] Shaughnessy, Pat. *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals*. USA: No Starch Press, 2013. ISBN 1593275277.

[4] *ClojureDocs - recur*. [online]. [cit. 2023-8-1]. Dostupný z: ⟨https://clojuredocs.org/clojure.core/recur⟩.

[5] *Loom - Fibers, Continuations and Tail-Calls for the JVM*. [online]. [cit. 2023-4-12]. Dostupný z: ⟨https://openjdk.org/projects/loom/⟩.

[6] *Writing a Language in Truffle. Part 1: A Simple, Slow Interpreter*. [online]. [cit. 2013-10-13]. Dostupný z: ⟨http://cesquivias.github.io/blog/2014/10/13/writing-a-language-in-truffle-part-1-a-simple-slow-interpreter/#mumbler-language⟩.

[7] *TruffleClojure: A self-optimizing AST-Interpreter for Clojure*. [online]. [cit. 2015-6-10]. Dostupný z: ⟨https://epub.jku.at/obvulihs/download/pdf/501665?originalFilename=true⟩.

[8] Gaikwad, Swapnil; Nisbet, Andy; Luján, Mikel. Performance analysis for languages hosted on the truffle framework. In. *Performance analysis for languages hosted on the truffle framework*. 2018, s. 1–12. Dostupný také z: ⟨http://dx.doi.org/10.1145/3237009.3237019⟩.

[9] *Co je to REPL (Read-Eval-Print-Loop)?* [online]. [cit. 2023-4-29]. Dostupný z: ⟨https://it-slovnik.cz/pojem/repl/?utm_source=cp&utm_medium=link&utm_campaign=cp/?utm_source=cp&utm_medium=link&utm_campaign=cp⟩.

[10] *Truffle/Graal:From Interpreters to Optimizing Compilers via Partial Evaluation*. [online]. [cit. 2020-1-29]. Dostupný z: ⟨https://www.cs.cmu.edu/~aldrich/courses/17-396/slides/truffle.pdf⟩.

[11] Přispěvatelé Wikipedie. *Java (platforma)* [online]. 2023 [cit. 2022-9-18]. Dostupný z: ⟨https://cs.wikipedia.org/w/index.php?title=Debugger&oldid=21202859⟩.

[12] Přispěvatelé Wikipedie. *Debugger* [online]. 2022 [cit. 2023-2-10]. Dostupný z: ⟨https://cs.wikipedia.org/w/index.php?title=Java_(platforma)&oldid=22434781⟩.

[13] *Pohled pod kapotu JVM – základy optimalizace aplikací naprogramovaných v Javě*. [online]. [cit. 2013-9-10]. Dostupný z: ⟨https://www.root.cz/clanky/pohled-pod-kapotu-jvm-zaklady-optimalizace-aplikaci-naprogramovanych-v-jave/⟩.

[14] *JVM JIT optimization techniques.* [online]. [cit. 2016-5-27]. Dostupný z: ⟨https://advancedweb.hu/jvm-jit-optimization-techniques/?fbclid=IwAR0cJJjrIi5m3byd3zohIuV-UwEYovLe29U6nhG5GNo66pz7jmONokYEbRQ⟩.

[15] *Tiered Compilation in JVM.* [online]. [cit. 2021-6-12]. Dostupný z: ⟨https://www.baeldung.com/jvm-tiered-compilation?fbclid=IwAR1tBSyf_Qv5tT6vWzVAjk1aTcor7eiHw2wiZiqkon4T-g6mPMY3YP4w6Mw⟩.

[16] *Pohled pod kapotu JVM – základy optimalizace aplikací naprogramovaných v Javě (5).* [online]. [cit. 2013-10-15]. Dostupný z: ⟨https://www.root.cz/clanky/pohled-pod-kapotu-jvm-zaklady-optimalizace-aplikaci-naprogramovanych-v-jave-5-1/⟩.

[17] *A Quick Introduction to Register Allocation.* [online]. [cit. 2013-10-15]. Dostupný z: ⟨https://hassamuddin.com/blog/reg-alloc/⟩.

[18] *Základní blok programu.* [online]. [cit. 2013-10-15]. Dostupný z: ⟨https://soubor.inf.upol.cz/ws-public/Vyuka-na-Windows/Arno%5C%C5%5C%A1t%5C%20Ve%5C%C4%5C%8Derka/PRKL⟩.

[19] Přispěvatelé Wikipedie. *Metacompilation* [online]. 2022 [cit. 2022-9-19]. Dostupný z: ⟨https://en.wikipedia.org/w/index.php?title=Metacompilation&oldid=1094965480⟩.

[20] Bolz, Carl Friedrich; Tratt, Laurence. The impact of meta-tracing on VM design and implementation. *SCICO.* 2015, s. 408–421. Dostupný také z: ⟨http://dx.doi.org/10.1016/j.scico.2013.02.001⟩.

[21] Přispěvatelé Wikipedie. *Static single-assignment form* [online]. 2022 [cit. 2022-10-11]. Dostupný z: ⟨https://en.wikipedia.org/w/index.php?title=Static_single-assignment_form&oldid=1112620046⟩.

[22] Duboscq, Gilles; Stadler, Lukas; Wuerthinger, Thomas aj. Graal IR: An Extensible Declarative Intermediate Representation. In. *Graal IR: An Extensible Declarative Intermediate Representation.* 2013.

[23] Humer, Christian. *Truffle DSL: A DSL for Building Self-Optimizing AST Interpreters.* 2016. Dostupný také z: ⟨http://dx.doi.org/10.13140/RG.2.2.23639.52646⟩.

[24] *Truffle/Graal: From Interpreters to Optimizing Compilers via Partial Evaluation.* [online]. [cit. 2020-4-7]. Dostupný z: ⟨https://www.cs.cmu.edu/~aldrich/courses/17-396/slides/truffle.pdf⟩.

[25] Würthinger, Thomas; Wimmer, Christian; Wöß, Andreas aj. One VM to Rule Them All. In. *One VM to Rule Them All.* Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, s. 187–204. Onward! 2013. Dostupný také z: ⟨https://doi.org/10.1145/2509578.2509581⟩.

[26] *Graal Truffle tutorial part 2 – introduction to specializations.* [online]. [cit. 2020-11-16]. Dostupný z: ⟨https://www.endoflineblog.com/graal-truffle-tutorial-part-2-introduction-to-specializations⟩.

[27] *Graal Truffle tutorial part 3 – specializations with Truffle DSL, TypeSystem.* [online]. [cit. 2021-1-5]. Dostupný z: ⟨https://www.endoflineblog.com/graal-truffle-tutorial-part-3-specializations-with-truffle-dsl-typesystem⟩.

[28] *Legacy macro support.* [online]. [cit. 2022-3-22]. Dostupný z: ⟨https://docs.racket-lang.org/compatibility/defmacro.html⟩.

[29] *What is ANTLR?* [online]. [cit. 2023-4-29]. Dostupný z: ⟨https://www.antlr.org/⟩.

[30] Jan Konečný, Vilém Vychodil. *Paradigmata programování 1A* [online]. [cit. 2008-3-20]. Dostupný z: ⟨https://phoenix.inf.upol.cz/esf/ucebni/pp1a.pdf⟩.

[31] Přispěvatelé Wikipedie. *Vedlejší účinek (programování)* [online]. 2022 [cit. 2023-2-10]. Dostupný z: ⟨https://cs.wikipedia.org/w/index.php?title=Vedlej%5C%C5%5C%A1%5C%C3%5C%AD_%5C%C3%5C%BA%5C%C4%5C%8Dinek_(programov%5C%C3%5C%A1n%5C%C3%5C%AD)&oldid=21420540⟩.

[32] Jan Konečný, Vilém Vychodil. *Paradigmata programování 1B* [online]. [cit. 2008-3-20]. Dostupný z: ⟨https://phoenix.inf.upol.cz/esf/ucebni/pp1b.pdf⟩.

[33] Schinz, Michel; Odersky, Martin. Tail Call Elimination on the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*. 2001, roč. 59, s. 158–171. Dostupný také z: ⟨http://dx.doi.org/10.1016/S1571-0661(05)80459-1⟩.

[34] *Truffle Library Guide.* [online]. [cit. 2023-3-23]. Dostupný z: ⟨https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/TruffleLibraries/⟩.

[35] *GNU Guile 3.0.0 released.* [online]. [cit. 2020-1-16]. Dostupný z: ⟨https://lists.gnu.org/archive/html/guile-devel/2020-01/msg00080.html⟩.

[36] *Just-In-Time Native Code.* [online]. [cit. 2020-2-18]. Dostupný z: ⟨https://www.gnu.org/software/guile/manual/html_node/Just_002dIn_002dTime-Native-Code.html⟩.

[37] Hlzle, Urs; Chambers, Craig; Ungar, David. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. 1999, s. 21–38.