překladač jednoduchého programovacího jazyka

maturitní práce 2015

jan špaček

Abstract

We present a simple programming language Spiral and its implementation. Spiral is an impure untyped functional language inspired by Scheme, featuring first-class functions, tail-calls, module system and both mutable and immutable data structures. We first translate the source language to Spine, a continuation-passing style functional language. Spine is then transformed into low-level imperative language Grit. Several optimization passes operate on this intermediate language, including global value analysis and inlining. As the last step of the compilation pipeline, we generate assembly, translate it with an external assembler and link the resulting object file with the runtime library. The runtime, written in C++, defines core functions and objects and manages the heap with a moving garbage collector. The implementation includes a basic standard library.

Abstrakt

V práci je představen jednoduchý programovací jazyk Spiral a jeho implementace. Spiral je funkcionální jazyk inspirovaný Scheme, který zahrnuje funkce první kategorie, koncová volání, systém modulů a měnitelné i neměnitelné datové struktury. Zdrojový jazyk je nejprve přeložen do Spine, funkcionálního jazyka založeného na CPS (continuation-passing style). Spine je poté transformován do nízkoúrovňového imperativního jazyka Grit, na kterém operuje několik optimalizačních fází, včetně globální optimalizace hodnot a inliningu. Jako poslední krok překladu je vygenerován externí assembler a přeložen externím programem. Výsledný objektový soubor je slinkován s podpůrnou knihovnou. Podpůrná běhová knihovna je napsaná v C++, definuje základní funkce a objekty a spravuje paměť pomocí přemisťovacího garbage collectoru. Součástí implementace je i základní standardní knihovna.

Prohlášení

Prohlašuji, že jsem svou práci vypracoval samostatně, použil jsem pouze podklady (literaturu, SW atd.) uvedené v přiloženém seznamu a postup při zpracování a dalším nakládání s prací je v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) v platném znění.

V ..... dne .....

Podpis: .....

Obsah

1 Úvod

Procesory počítačů pracují se sadou jednoduchých instrukcí, které lze rychle a efektivně vykonávat. Programování v těchto základních stavebních jednotkách je ovšem natolik pomalé, obtížné a náchylné k chybám, že se rychle po sestrojení prvních počítačů vyvinuly rovněž první programovací jazyky, které umožňují programy vytvářet v podobě, která je pro člověka jednodušší. Vývoj v takovém programovacím jazyku je nesrovnatelně efektivnější než v kódu počítače a programy v něm napsané je často jednodušší použít i na jiných počítačích, než na které byly původně určeny.

Aby program napsaný ve vyšším programovacím jazyku mohl být spuštěn, musí pro tento jazyk existovat interpret nebo kompilátor (překladač). Interpret je program, který přečte zdrojový kód interpretovaného jazyka a sám vykonává operace v něm obsažené. Překladač naproti tomu přeloží program v programovacím jazyku do strojového kódu, takže je poté tento program možno spouštět přímo na procesoru počítače, k čemuž již není potřeba původní zdrojový kód ani překladač. Vytvořit interpret je většinou poměrně jednoduché, ovšem interpretovaný program zpravidla dosahuje horšího výkonu než program přeložený přímo do strojového kódu.

Programovací jazyky se většinou soustředily na organizaci kódu a průběhu výpočtu, ovšem menší pozornost byla věnována problému organizace paměti. Dnes proto existují v zásadě dva typy jazyků: nízkoúrovňové jazyky jako C a C++, kde je správa paměti plně v režii programátora, a všechny ostatní jazyky, kde je paměť spravována pomocí garbage collectoru, který paměť uvolňuje automaticky bez zásahu programátora. Oba tyto přístupy však mají své problémy. Ruční správa paměti je extrémně náchylná k chybám, které je obtížné odhalit, a navíc často představují bezpečnostní riziko (asi nejznámější bezpečnostní chybou tohoto typu byl Heartbleed [9]). Na druhou stranu garbage collectory zbavují programátora možnosti nakládat volně s pamětí a ubírají programům na výkonu.

1.1 Struktura překladače

Struktura běžného překladače vypadá přibližně takto: [10, 5]:

- Lexikální a syntaktická analýza zdrojového jazyka, jejímž výsledkem je syntaktický strom.
- Sémantická analýza zdrojového jazyka (spojení jmen a jejich definic, kontrola typů). Tato část překladače je spolu s lexikální a syntaktickou analýzou označována jako front-end.
- Generování přechodného jazyka (*intermediate language*), který je jednodušší a pro další zpracování vhodnější než zdrojový jazyk.
- Analýza a optimalizace přechodného jazyka.
- Překlad přechodného jazyka do assembleru nebo objektového souboru (back-end).

Jako příklad lze uvést překladač GHC [14, 4], který nejprve zdrojový jazyk Haskell přeloží do jazyka Core [12], nad kterým probíhají optimalizace [13, 16]. Core je poté přeložen do jazyka STG [11], ze kterého je následně generován kód.

Často je možné několik zdrojových jazyků překládat do jednoho přechodného jazyka a sdílet tak společný back-end. Jedním z takových projektů je LLVM [2], na kterém je založen například překladač Clang [1] jazyků C a C++ nebo překladač jazyka Rust [3].

2 Popis jazyka Spiral

Samotný jazyk je velmi prostý, jelikož poskytuje pouze nástroje pro definici modulů, tvorbu funkcí a proměnných, základní větvení a literály. Primitivní operace (například sčítání čísel či přístup k prvkům pole) jsou definovány ve standardní knihovně, kde jsou implementovány jako externí volání podpůrné knihovny. Za zmínku stojí, že identifikátory jako `+` a `*` nejsou zabudované operátory, ale klasické proměnné.

Proměnné jsou neměnitelné, neexistuje proto žádný příkaz přiřazení. Některé objekty ze standardní knihovny, například pole, je ale možno interně upravovat. Funkce, které mění nějaký objekt, je zvykem pojmenovávat s vykřičníkem na konci (`array-set!`).

Volání funkcí v koncových pozicích (*tail-calls*) jsou implementována tak, že umožňují neomezené množství rekurzivních volání bez hrozby přetečení zásobníku, což umožňuje efektivně vyjádřit všechny iterace pomocí rekurze.

Funkce jsou hodnoty podobně jako čísla nebo řetězce, je s nimi tedy možno bez omezení pracovat jako v jiných funkcionálních jazycích. Ve funkci je možno použít proměnné definované vně těla funkce. Tyto proměnné se nazývají zachycené (*captured*) a jejich hodnoty jsou spolu s adresou funkce uloženy v paměti v objektu, který tuto funkci reprezentuje (*closure*).

2.1 Gramatika

Syntaxe jazyka Spiral je založena na syntaxi jazyka Scheme [17]. Program je tak zapsán ve formě uzávorkovaných prefixových výrazů (*s-expressions*). Tento způsob zápisu je jednoduchý na parsování a příjemný na psaní.

Programy a moduly

```
<top-level>   = <program> | <module>
<program>     = (program <stmt>...)
<module>      = (module <ident> <decl>...)
<decl>        = (export <ident>...) | <stmt>
```

Program obsahuje seznam příkazů (*statements*, `<stmt>...`), které jsou postupně vykonány. Modul obsahuje deklarace, které kromě příkazů zahrnují i exporty, které modul poskutuje ostatním modulům a hlavnímu programu.

Příkazy

Příkazy slouží k definici importovaných jmen, pojmenovaných funkcí i obyčejných proměnných.

```
<stmt>        = (import <import-def>...)
              | (fun <ident> (<ident>...) <stmt>...)
              | (var <ident> <expr>)
              | <expr>
<import-def>  = <ident>
              | (only <import-def> <ident>...)
              | (except <import-def> <ident>...)
              | (prefix <import-def> <ident>)
```

`(import <import-def>...)` Příkaz importu načte modul a umožní používat jeho exportované proměnné v aktuálním kontextu. Je možno importovat vybraná jména pomocí `(only ...)`, nebo importovat všechny kromě zadaných `(except ...)` nebo všem importovaným jménům předřadit prefix `(prefix ...)`.

Příklady importů:

- `(import std std.math)` importuje všechna jména z modulů `std` a `std.math`.
- `(import (only std + - *))` importuje ze `std` pouze jména `+`, `-` a `*`.

- `(import (except std.math sin cos))` importuje z `std.math` všechna jména kromě `sin` a `cos`.
- `(import (prefix (only std.math div mod) m.))` importuje z modulu `std.math` jména `div` a `mod` a přidá jim prefix `m.` (takže se na ně bude možno odkazovat jako `m.div` a `m.mod`).

`(fun <fun-name> (<arg>...) <body-stmt>...)` definuje pojmenovanou funkci. Funkce, které jsou takto definovány těsně za sebou, mohou být vzájemně rekurzivní, jelikož funkce definovaná v pořadí dříve „vidí“ i definice pozdějších funkcí (včetně sebe samotné). Kromě toho je ve funkcích možno použít veškeré předem definované proměnné.

`(var <var-name> <value>)` definuje proměnnou. Proměnná, narozdíl od funkce, odkazovat na sebe samu nemůže, jelikož její hodnota může být použita až po její definici.

`<expr>` – pokud je na místě příkazu nalezen výraz, je vyhodnocen. Pokud je na posledním místě v posloupnosti příkazů, je jeho hodnota použita jako hodnota celé posloupnosti, jinak je výsledek zahozen.

Výrazy

Všechny zbývající konstrukce jazyka jsou výrazy (*expressions*), což znamená, že vrací hodnotu. Pokud není žádná rozumná hodnota, kterou by měl výraz vrátit (například prázdný výraz `(begin)`), vrátí výraz hodnotu `false`.

```
<expr>        = (if <expr> <expr> <expr>)
              | (cond (<expr> <stmt>...)...)
              | (when <expr> <stmt>...)
              | (unless <expr> <stmt>...)
              | (do ((<ident> <expr> <expr>)...) (<expr> <stmt>...) <stmt>...)
              | (and <expr>...)
              | (or <expr>...)
              | (begin <stmt>...)
              | (let ((<ident> <expr>)...) <stmt>...)
              | (lambda (<ident>...) <stmt>...)
              | (<expr> <expr>...)
              | (extern <ident> <expr>...)
              | <integer-literal>
              | <float-literal>
              | <string-literal>
              | <character-literal>
```

`<integer-literal>`, `<float-literal>` jsou číselné konstanty, které se vyhodnotí na číslo, které reprezentují. Číslice je možno oddělovat podtržítky a v zápisu desetinného čísla je možno použít exponent.

`<character-literal>` je znaková konstanta, uzavřená v jednoduchách uvozovkách, která se vyhodnotí na číselnou hodnotu znaku.

`<string-literal>` je řetězcový literál uzavřený v uvozovkách, ve kterém je možné použít escape sekvence z jazyka C (například `\n` je znak nového řádku, `\"` je uvozovka).

`(if <condition> <then> <else>)` je podmíněný příkaz, který nejprve vyhodnotí podmínku, a pokud je pravdivá, vyhodnotí první výraz, jinak vyhodnotí druhý výraz. Za pravdivé jsou považovány všechny hodnoty kromě `false`.

`(when <condition> <body-stmt>...)` vyhodnotí podmínku a pokud je pravdivá, vyhodnotí následující příkazy.

`(unless <condition> <body-stmt>...)` je obdobou `when`, ale příkazy vyhodnotí pouze pokud je podmínka nepravdivá.

`(cond (<condition> <stmt>...)..)` postupně zkouší vyhodnocovat podmínky a vyhodnotí příkazy u první, která je pravdivá.

`(and <expr>...)` vyhodnocuje výrazy zleva a vrátí hodnotu prvního, který je nepravdivý, čímž implementuje logické „a“.

`(or <expr>...)` vyhodnocuje výrazy zleva a vrátí hodnotu prvního, který je pravdivý, což je obdoba logického „nebo“.

`(let ((<var> <expr>)...) <body-stmt>...)` přiřadí proměnným hodnoty daných výrazů a poté vyhodnotí následující příkazy.

`(lambda (<arg>...) <body-stmt>...)` vytvoří anonymní funkci s danými argumenty a příkazy, které tvoří její tělo. Uvnitř funkce je opět možno používat proměnné z vnějšku funkce.

`(<fun> <arg>...)` je zápis volání. Nejprve se vyhodnotí všechny argumenty zleva doprava a poté funkce, která je poté zavolána. Pokud se první výraz nevyhodnotí na funkci, program skončí s chybou.

`(extern <fun-name> <arg>...)` je volání externí funkce (definované v jazyku C) s danými argumenty. Obvykle se tato volání objevují pouze ve standardní knihovně. Počet argumentů ani existenci funkce není možno zkontrolovat, korektnost je tedy závislá na programátorovi.

`(begin <body-stmt>...)` vyhodnotí všechny příkazy a vrátí hodnotu posledního.

`(do ((<var> <init> <next>)...) (<exit-condition> <exit-stmt>...) <body-stmt>...)` je výraz cyklu. Do všech proměnných je neprve přiřazena počáteční hodnota (`<init>`). Poté je vyhodnocena podmínka (`<exit-condition>`) a pokud je pravdivá, vyhodnotí se následující příkazy (`<exit-stmt>...`) a cyklus se ukončí. V opačném případě jsou vyhodnoceny příkazy v těle cyklu (`<body-stmt>...`), poté jsou do proměnných přiřazeny hodnoty `<next>` a cyklus se opakuje novou kontrolou podmínky.

Například následující program spočte sto čísel Fibonacciho posloupnosti, přičemž každé vypíše, a na konci vypíše `done`:

```
(program
  (import std)
  (do ((f1 0 f1)
       (f2 1 (+ f1 f2))
       (i  1 (+ i 1)))
    ((>= i 100)
      (println "done"))
    (println f1)))
```

2.2 Standardní knihovna a základní typy

Standardní knihovna definuje základní funkce. Jelikož program ve Spiral nemá implicitně definovaná žádná jména, je import standardní knihovny prakticky nutností.

`std.core` definuje základní operace s čísly (`+`, `mod`, `<`, `==`, ...), predikáty ekvivalence (`eqv?`, `equal?`) a základní funkci pro výstup (`println`). Rovněž definuje logické proměnné `true` a `false` (jako hodnoty výrazů `(and)` a `(or)`).

`std.array` poskytuje funkce pro práci s poli (`array-new`, `array-push!`, `array-get`, `array-empty?`, ...). Pole jsou indexována celými čísly od nuly a jejich délka i obsah jsou měnitelné.

`std.tuple` umožňuje používat n-tice (*tuples*), a to v délce od 0 do 8 prvků. Modul definuje konstruktory (`tuple-2`, ...), funkce pro přístup k jednotlivým prvkům (`get-0`, `get-2`) a predikáty (`tuple?`, `tuple-2?`, `tuple-n?`, ...). Prvky n-tic jsou neměnitelné.

`std.cons` definuje funkce pro práci s páry (*cons*), které se převážně používají ve formě spojových seznamů, kdy první prvek páru (*car*) ukládá hodnotu prvního prvku seznamu a druhý prvek (*cdr*) odkazuje na zbytek seznamu. Konec seznamu reprezentuje speciální hodnota (*null* či *nil*), pro kterou je ve Spiral použito `false`. Tento modul definuje základní funkce pro práci s páry (`cons`, `car`, `cdr`, `cons?`, ...) i pro manipulaci se seznamy (`list?`, `list-len`, `list-append`, `list-reverse`, ...). Páry jsou neměnné, což znamená, že není možno vytvořit kruhový seznam.

`std.string` exportuje funkce pracující s řetězci (`str-len`, `str-get`, `stringify`, `str-cat-2`, `str-cat-3`, ...). Řetězce jsou neměnitelné a složené z bytů.

`std.io` slouží pro vstupní a výstupní operace (*input/output*) se soubory a standardními proudy (`io-file-open`, `io-close`, `io-stdin`, `io-write-line`, `io-read-line`, `io-read-number`, ...).

`std.env` poskytuje programu přístup k argumentům z příkazové řádky (`env-get-argv`) a k proměnným prostředí (`env-get-var`).

`std.math` implementuje základní matematické funkce jako `abs`, `neg`, `sin` nebo `atan-2`.

`std.test` je minimalistická knihovna pro tvorbu jednotkových testů.

Modul `std` reexportuje základní definice (například `+` nebo `car`) a je poskytován pro pohodlí programátora.

2.3 Ovládání překladače z příkazové řádky

Překladač je možno ovládat pomocí argumentů na příkazové řádce:

`-o, --output` umožňuje určit výstupní soubor. Výstup je jinak umístěn do stejného adresáře jako vstupní soubor se stejným jménem a příponou určenou podle typu výstupu.

`-I, --include` přidá adresář, ve kterém se hledají `import`ované moduly.

`-e, --emit` nastaví typ výstupu: `sexpr` přečte vstupní s-výraz a vypíše jej ve formátované podobě, `spiral` vypíše syntaktický strom Spiral v interní podobě, `spine` a `grit` vypíšou program v odpovídajícím mezijazyku jako s-výraz, `asm` vypíše vygenerovaný assembler v interní podobě, `gas` vypíše assembler jako vstup pro GNU Assembler a `exec` program přeloží a slinkuje s běhovou knihovnou (což je výchozí chování).

`-t, --runtime` určí soubor s běhovou knihovnou, se kterou bude program slinkován.

`--link-cmd` nastaví linkovací program. Výchozí je `clang`, který sám zavolá systémový linker tak, aby program správně slinkoval se standardní knihovnou jazyka C (`libc`).

`--gas-cmd` umožňuje změnit assembler místo výchozího `as`.

`-O, --opt-level` umožňuje nastavit úroveň optimalizace od 0 do 3.

2.4 Příklady

Na závěr uvádíme několik drobných programů napsaných v jazyce Spiral.

Počítání prvočísel

Následující program vypíše prvních tisíc prvočísel. Funkce `make-prime-gen` vytvoří generátor, tedy funkci, která při každém dalším zavolání vrátí další prvočíslo. Tato funkce má interně uloženo pole doposud nalezených prvočísel.

```
(program
  (import std)
  (import std.array)
  (import std.test)

  (fun make-prime-gen ()
    (var primes (array-new))
    (lambda ()
      (fun find-next-prime (x)
        (fun check-prime (i)
          (var p (array-get primes i))
          (cond
            ((> (* p p) x) true)
            ((== (mod x p) 0) false)
            (true (check-prime (+ 1 i)))))
        (if (check-prime 1) x (find-next-prime (+ 2 x))))
      (cond
        ((== (array-len primes) 0)
          (array-push! primes 2) 2)
        ((== (array-len primes) 1)
          (array-push! primes 3) 3)
        (true
          (var next-prime (find-next-prime (+ 2 (array-last primes))))
          (array-push! primes next-prime)
          next-prime))))

  (var gen (make-prime-gen))
  (do ((i 0 (+ i 1)))
    ((>= i 1000))
    (println (gen))))
```

Počítání celých mocnin a odmocnin

Tento program aproximuje $\sqrt[n]{x}$ Newtonovou metodou (funkce `root`) a počítá x^n pomocí binárního umocňování (funkce `power`). Oba algoritmy jsou poté otestovány pomocí knihovny `std.test`.

```
(program
  (import std)
  (import std.math)
  (import std.test)

  (var min-del 0.000001)
  (fun root (n a)
    (fun iter (x)
      (var b (/ a (power (- n 1) x)))
      (var x-del (/ (- b x) n))
      (if (< (abs x-del) min-del) x (iter (+ x x-del))))
    (iter (* a 1.0)))
  (fun power (n a)
    (cond
      ((== n 0) 1)
      ((== n 1) a)
      ((== (mod n 2) 0) (square (power (div n 2) a)))
      (true (* a (square (power (div n 2) a))))))
  (fun square (a) (* a a))
```

```
(var eps 0.00001)
(test "powers" (lambda (t)
  (assert-near-eq t eps (power 2 10) 100)
  (assert-near-eq t eps (power 3 2) 8)
  (assert-near-eq t eps (power 1 33) 33)))
(test "roots" (lambda (t)
  (assert-near-eq t eps (root 3 1000) 10)
  (assert-near-eq t eps (root 2 49) 7)
  (assert-near-eq t eps (root 4 256) 4)))
(test "powered roots" (lambda (t)
  (assert-near-eq t eps (power 2 (root 2 2)) 2)
  (assert-near-eq t eps (power 3 (root 3 100)) 100)
  (assert-near-eq t eps (power 2 (root 2 13)) 13)
  (assert-near-eq t 0.01 (power 5 (root 5 12345)) 12345))))
```

3 Postup překladu

Vstupní zdrojový kód je nejprve přečten parserem s-výrazů do obecné struktury, ze které je poté dekódován syntaktický strom pro Spiral. Následně jsou všechny moduly a hlavní program společně přeloženy do *continuation-passing style* v jazyku Spine, který slouží jako první přechodný jazyk. Z jazyka Spine je pak celý program přeložen do imperativního jazyka Grit, na kterém proběhne optimalizace. Z jazyka Grit je pak již vygenerován assembler, čímž postup překladu končí.

3.1 S-výrazy

Parsování s-výrazů provádí jednoduchý ručně psaný parser. Výsledná datová struktura je poté dále zpracovávána. Rovněž je možné tuto datovou strukturu zpětně zapsat do textové formy (*pretty-print*). Kromě jazyka Spiral je v s-výrazech možno zapsat i přechodné jazyky Spine a Grit, což je využito pro testování, a to na čtení i výpis programů.

3.2 Spiral

Syntaktický strom jazyka Spiral je dekódován z načteného s-výrazu, což je jednoduchý, leč nezáživný proces. Zde jsou odhaleny a oznámeny syntaktické chyby, které se v programu nacházejí. Po přečtení následuje průchod stromem s cílem odhalit všechny importované moduly, které jsou posléze rovněž načteny a zpracovány. Moduly se posléze topologicky seřadí podle vzájemné závislosti, aby mohly být zpracovány. Pokud žádné takové seřazení neexistuje, tedy pokud graf závislosti není acyklický, překlad skončí s chybou.

3.3 Spine

Prvním přechodným jazykem je Spine. Tento jazyk je silně inspirován jazykem $\lambda^{U}_{\mathrm{CPS}}$ [15] a je založen na λ-kalkulu a *continuation-passing style*.

Continuation je speciální λ-funkce, která je lokální pro danou funkci a nikdy nevrátí hodnotu, volání *continuation* tedy modeluje skok na *basic block*, se kterým bychom se mohli setkat v imperativním jazyce nebo SSA formě (*single static assignment*). Při volání funkce je kromě funkce samotné a jejích argumentů třeba specifikovat i *continuation*, které bude výsledek volání funkce předán. Návrat z funkce má podobu skoku do její návratové *continuation*.

```
<program>   = (program <cont-name> <term>)
<term>      = (letcont <cont-def>... <term>)
            | (letfun <fun-def>... <term>)
            | (letobj <obj-def> <term>)
            | (call <val> <cont-name> <val>...)
            | (extern-call <extern-name> <cont-name> <val>...)
            | (cont <cont-name> <val>...)
            | (branch <boolval> <cont-name> <cont-name>)
<fun-def>   = (<var> <cont-name> (<var>...) (<var>...) <term>)
<cont-def>  = (<cont-name> (<var>...) <term>)
<obj-def>   = (string <var> <string-literal>)
            | (double <var> <double-literal>)
<val>       = <var> | <int-literal> | (true) | (false)
<boolval>   = (is-true <val>) | (is-false <val>)
```

Program je definován jako jeden velký výraz (*term*) a ukončovací *continuation*, po jejímž zavolání program skončí.

`(cont <cont-name> <arg>...)` skočí do zadané *continuation* a předá jí určené argumenty (počet argumentů musí odpovídat její definici).

`(branch <boolval> <then-name> <else-name>)` vyhodnotí pravdivostní výrok `<boolval>` a podle toho skočí na jednu z uvedených *continuations* (které nesmí očekávat argumenty).

`(call <fun-val> <return-cont> <arg>...)` zavolá danou funkci se zadanými argumenty a poté skočí do `<return-cont>`, které předá návratovou hodnotu. Pokud je `<return-cont>` návratovou *continuation* volající funkce, je toto volání koncové.

`(extern-call <extern-name> <return-cont> <arg>...)` zavolá externí funkci danou svým jménem a s jejím výsledkem skočí do `<return-cont>`. Toto voláni nikdy není koncové.

`(letcont (<cont-name> (<arg>...) <body>)... <term>)` definuje *continuations*, které budou viditelné ve vnořeném výrazu. Každá může přijímat libovolný počet argumentů (`(<arg>...)`). Tyto *continuations* mohou být vzájemně rekurzivní, takže je možno implementovat cykly.

`(letfun (<fun> <ret-cont> (<capture>...) (<arg>...) <body>)... <term>)` definuje funkce, které je možno použít ve vnořeném výrazu. Uvnitř těla funkci (`<body>`) není možné použít *continuation* z aktuálního kontextu a veškeré zachycené (*captured*) proměnné musí být specifikovány v definici funkce (`(<captured>..)`). Návrat z funkce bude proveden jako skok do *continuation* `<ret-cont>`.

`(letobj <obj-def> <term>)` definuje objekt (řetězec nebo reálné číslo).

Všechny hodnoty (`<val>`) jsou atomické (jednoduché proměnné nebo konstanty), takže jejich vyhodnocení nestojí žádné výpočetní úsilí a je možno je libovolně kopírovat.

Překlad z jazyka Spiral

Při překladu výrazu z jazyka Spiral do Spine je třeba převést program z *direct style* do *continuation-passing style*. Základem překladu jsou dvě funkce, `translate-expr(spiral-expr) -> (onion, spine-val)` a `translate-expr-tail(spiral-expr, spine-cont-name) -> spine-term`.

Funkce `translate-expr` přeloží předaný výraz ve Spiral tak, že vrátí „cibuli“[1] (*onion*) a hodnotu. „Slupka cibule“ je tvořena výrazy `letcont`, `letfun` a `letobj`. Uvnitř této slupky je výsledek výrazu reprezentován danou hodnotou.

Druhá funkce `translate-expr-tail` pak přeloží výraz do podoby výrazu ve Spine, který s výslednou hodnotou skočí do předané *continuation*. Tímto způsobem jsou přeložena například koncová volání (*tail-calls*).

Pro ilustraci si ukážeme příklad, jak je přeložena tato funkce ve Spiral:

```
(fun big-enough? (x)
  (if (< x 0)
    (println "small")
    (println "ok")))
```

Nejprve vygenerujeme jméno návratové *continuation* pro funkci `big-enough?`, například `r`. Tělo funkce musí být přeloženo tak, aby volání v koncových pozicích byla koncová, tedy aby vracela do `r`. Proto na výraz `(if (< x 0) (println "small") (println "ok"))` použijeme `translate-expr-tail` s `r`.

Při vyhodnocení výrazu `if` musíme nejprve vyhodnotit podmínku `(< x 0)`. Tu přeložíme pomocí `translate-expr`, čímž dostaneme „cibuli“ `(letcont (lt-ret (lt-result) ?) (call < lt-ret x 0))`. Otazník označuje místo, kde obdržíme výsledek v proměnné `lt-result` („díru“).

Obě alternativy v příkazu `if` přeložíme opět pomocí `translate-expr-tail` s *continuation* `r`, abychom zachovali koncová volání. Obdržíme Spine výrazy `(letobj (string s1 "small") (call println r s1))` a `(letobj (string s2 "ok") (call println r s2))`.

Samotné větvení provedeme výrazem `branch`, na to ale potřebujeme *continuation*, do které můžeme skočit. Ty si proto vytvoříme (nazveme je `on-true` a `on-false`) a vložíme do nich přeložené výrazy z obou větví zdrojového výrazu `if`. Výsledek pak vypadá takto:

[1] Na obranu komunity počítačových vědců je třeba dodat, že tento název vymyslel autor sám.

```
(letcont (lt-ret (lt-result)
           (letcont (on-true ()
                      (letobj (string s1 "small") (call println r s1)))
                    (on-false ()
                      (letobj (string s2 "ok") (call println r s2)))
             (branch (is-true lt-result) on-true on-false)))
  (call < lt-ret x 0))
```

Je vidět, že pořadí vyhodnocování a přenos informací z původního programu je nyní explicitně vyjádřeno. Nejprve se zavolá funkce `<` s argumenty `x` a `0`. Ta svůj výsledek předá v proměnné `lt-result` do `lt-ret`. Ta následně tuto proměnnou prozkoumá, a pokud je její hodnota pravdivá, skočí do `on-true`, jinak do `on-false`. V `on-true` a `on-false` pak definujeme patřičný řetězec a následně zavoláme funkci `println`, které tento řetězec předáme. Funkce `println` vrátí svůj výsledek do `r`, což je ale návratová *continuation* volající funkce `big-enough?`, takže toto volání bude přeloženo jako koncové.

3.4 Grit

Dalším jazykem v pořadí je Grit. Tento jazyk je již poměrně nízkoúrovňový a blízký assembleru. Definice všech funkcí a objektů jsou globální a proměnné jsou zapisovatelné a jsou pojmenované pouze čísly. Funkce se skládají z bloků, které obsahují sekvenci operací a jsou ukončeny skokem.

```
<program>   = (program <fun-name> <fun-def>... <obj-def>...)
<fun-def>   = (fun <fun-name> <int> <int> <int> <label> <block>...)
<obj-def>   = (string <obj-name> <string-literal>)
            | (double <obj-name> <double-literal>)
<block>     = (<label> <op>... <jump>)

<op>        = (call <var> <callee> <val>...)
            | (extern-call <var> <extern-name> <val>...)
            | (alloc-clos (<var> <fun-name> <val>...)...)
            | (assign (<var> <val>)...)
<jump>      = (goto <label>)
            | (return <val>)
            | (tail-call <callee> <val>...)
            | (branch <boolval> <label> <label>)
<callee>    = (combinator <fun-name>)
            | (known-closure <fun-name> <val>)
            | (unknown <val>)

<val>       = (var <int>)
            | (arg <int>)
            | (capture <int>)
            | (combinator <fun-name>)
            | (obj <obj-name>)
            | (int <int>)
            | (true)
            | (false)
            | (undefined)
<boolval>   = (is-true <val>) | (is-false <val>)
```

Jednotlivé operace reprezentují vše, co program může vykonávat:

`(call <var> <callee> <arg>...)` zavolá funkci specifikovanou v `<callee>` se zadanými argumenty a její výsledek zapíše do proměnné. Hodnoty `<callee>` mohou být:

`(combinator <fun-name>)` je volání kombinátoru, tedy funkce, která nemá žádné uložené proměnné z prostředí. Taková funkce je staticky alokovaná, není proto třeba žádná proměnná na její uchování. U tohoto volání je navíc možno zkontrolovat počet argumentů předem, takže volaná funkce je již kontrolovat nemusí.

`(known-closure <fun-name> <val>)` je volání známé funkce, která má uložené proměnné z prostředí (*closure*), takže ji musíme volat spolu s její funkční hodnotou. Pokud by hodnota `<val>` nebyla funkce `<fun-name>`, program by selhal. I zde je zaručeno, že počet argumentů odpovídá, takže volaná funkce je nekontroluje.

`(unknown <val>)` je volání, u kterého není staticky zjištěno, kterou funkci bude volat. Za běhu je proto nutno zkontrolovat, že `<val>` je skutečně funkce, a tato pak ještě sama kontroluje počet předaných argumentů.

`(extern-call <var> <extern-name> <arg>...)` zavolá externí funkci s předanými argumenty a výsledek zapíše do dané proměnné.

`(alloc-clos (<var> <fun-name> <capture>...))` alokuje funkce se zadanými názvy a určeným seznamem zachycených hodnot. Tyto funkce jsou zapsány do proměnných ještě předtím, než jsou zachycené hodnoty zapsány, což umožňuje funkci uložit referenci na sebe samu. Rovněž je takto možné získat hodnotu kombinátoru, pokud je seznam zachycených hodnot prázdný – v tom případě se nic nealokuje.

`(assign (<var> <val>)...)` zapíše do proměnných odpovídající hodnoty. Toto přiřazení proběhne najednou, takže na pravé straně je možno použít proměnné z levé strany, jejichž hodnota bude odpovídat hodnotě před operací `assign`.

Skoky mohou být rovněž několika druhů:

`(goto <label>)` skočí na daný blok.

`(return <val>)` vrátí z funkce určenou hodnotu.

`(tail-call <callee> <arg>...)` provede koncové volání `<callee>` (podobně jako v operaci `call`) s argumenty. Tímto se místo na zásobníku zabrané volající funkcí uvolní ještě před vstupem do volané funkce.

`(branch <boolval> <then> <else>)` skočí na jeden z bloků podle určené pravdivostní hodnoty.

Paleta hodnot je širší než ve Spine, ovšem stále platí, že každá hodnota je konstantní nebo ji lze snadno načíst z paměti:

`(var <index>)` je hodnota proměnné.

`(arg <index>)` je hodnota argumentu.

`(capture <index>)` je hodnota zachycené hodnoty z aktuální funkce.

`(combinator <fun-name>)` je konstantní hodnota staticky alokovaného kombinátoru.

`(obj <obj-name>)` je konstantní hodnota staticky alokovaného objektu.

`(int <int>)` je celočíselná konstanta.

`(true)`, `(false)` jsou pravdivostní konstanty.

`(undefined)` je nedefinovaná hodnota. Tuto hodnotu mohou vytvořit optimalizace, například při čtení z proměnné, do které nebylo zapsáno. Při zápisu nedefinované hodnoty do paměti či registru během překladu do assembleru není vygenerována žádná instrukce.

Překlad z jazyka Spine

Překlad ze Spine do Gritu je poměrně přímočarý. Volání *continuation* je přeloženo jako přiřazení do proměnných vygenerovaných jako její argumenty a skok do bloku, kterým tělo *continuation* začíná. Ostatní konstrukce ve Spine mají v jazyku Grit přímý ekvivalent.

Optimalizace

Jelikož je jazyk Spiral poměrně chudý, nabízí se relativně málo možností optimalizace. Proto se první optimalizační fáze aplikují až na nízkoúrovňový Grit a jedná se převážně o zjednodušující a zeštíhlující proces.

Optimalizace známých hodnot

V této fázi, která operuje nad celým programem, je určen odhad hodnot, kterých může nabýt každá proměnná, zachycená proměnná a návratová hodnota. Tyto informace jsou poté využity několikerým způsobem:

- Optimalizace známých volání nahrazením `(call (unknown ...) ...)` za `(call (known-closure ...) ...)` nebo `(call (combinator ...) ...)`.
- Nahrazení větvení nepodmíněným skokem, pokud je možné dokázat, že testovací hodnota je vždy pravdivá nebo vždy nepravdivá.
- Propagace konstant odstraní proměnné, jejichž hodnota je konstantní (číslo, pravdivostní hodnota nebo staticky alokovaný objekt).

Odstranění nepotřebných hodnot

Tato fáze je opět globální a jejím účelem je převážně zrušit nepotřebné zachycené hodnoty, ovšem zároveň je možno odstranit i zápisy do nevyužitých proměnných a alokaci zbytečných funkcí. Často se stane, že funkce přijde o všechny zachycené hodnoty, čímž se stane kombinátorem. Její použití pak již není spojeno s alokací paměti.

Inlining funkcí

Během fáze inliningu jsou volání vybraných funkcí nahrazena přímým vložením volané funkce do funkce volající. Tato optimalizace je pro funkcionální programy zcela klíčová, protože se většinou skládají z velkého počtu malých funkcí. Jejich expanzí na místo volání se ušetří instrukce pro manipulaci se zásobníkem a pro samotné volání a návrat z funkce. Zároveň je pak možné provést další optimalizace, převážně proto, že jsou nyní zpravidla k dispozici lepší informace o argumentech volané funkce. Na druhou stranu takto dochází k duplikaci kódu a většinou tak i k nárůstu délky výsledného programu.

Pro inlining jsou nyní vybírány funkce, které jsou dostatečně malé, jsou kombinátory a nevolají jiné funkce kromě externích. Do této kategorie spadá většina funkcí standardní knihovny, které obvykle pouze obalují externí volání podpůrné běhové knihovny.

Odstranění nepotřebných definic

Tato optimalizace je známá rovněž jako odstranění mrtvého kódu (*dead code elimination*) a spočívá ve vynechání funkcí a staticky alokovaných objektů, které nejsou referencovány z hlavní funkce a jsou tedy zbytečné. Kromě nepoužitých definic z importovaných modulů jsou obvykle odstraněny i funkce, jejichž volání byla všude inlinována.

Pořadí optimalizací

U optimalizačních fází je velmi důležité pořadí, v jakém jsou na program aplikovány. Optimalizace známých hodnot zanechá část proměnných a zachycených hodnot bez využití. Tyto proměnné jsou následně odstraněny ve fázi odstranění nepotřebných hodnot. Zároveň se takto zvýší počet kombinátorů, které můžeme inlinovat. Po inliningu pak dostaneme řadu nepotřebných definic, které jsou kandidáty na odstranění.

Úroveň optimalizací je možno ovlivnit z argumentů příkazové řádky. Na úrovni 0 neprobíhá žádná optimalizace. Úroveň 1 zapne všechny optimalizace ve výše uvedeném pořadí kromě

inliningu, který je umožněn až od úrovně 2. Při úrovni 3 pak po inliningu následuje ještě jednou optimalizace známých hodnot a odstranění nepotřebných hodnot.

3.5 Alokace slotů

Program v Gritu typicky pracuje s velkým množstvím proměnných. Bylo by ovšem značným plýtváním pamětí, pokud bychom každé proměnné alokovali na zásobníku zvláštní prostor (slot), jelikož doba života proměnné je obvykle krátká a její slot by byl většinu času nevyužit. Proto je žádoucí několika proměnným přiřadit stejný slot, ovšem samozřejmě tak, aby nedošlo ke ztrátě informace, tedy aby zápis do proměnné nepřepsal hodnotu jiné proměnné, ze které se bude později číst.

Této fázi v běžných překladačích odpovídá alokace registrů, ovšem náš generátor kódu je zjednodušený, takže všechny proměnné umisťuje do paměti. Podobně jako při alokaci registrů však můžeme použít přístup s užitím barvení grafu interference [8, 7, 6], ovšem postup je jednodušší díky tomu, že počet barev (registrů) nemáme omezen, pouze se jej snažíme minimalizovat. Proto stačí pouze sestavit graf interference a ten pak hladovým algoritmem obarvit. U hladového algoritmu je klíčové pořadí, v jakém jsou vrcholy barveny. V našem případě vrcholy barvíme v pořadí podle počtu vycházejících hran sestupně.

3.6 Assembler

Posledním krokem překladu je generování assembleru pro architekturu IA-32 z jazyka Grit. Výsledný soubor obsahuje kód všech funkcí, staticky alokované objekty a řetězce. Kvůli podpoře koncových volání mají funkce jazyka Spiral jinou volací konvenci (*calling convention*) než funkce jazyka C. Argumenty pro funkci v C jsou totiž umístěny na zásobníku v oblasti volající funkce, pro koncová volání je však potřeba, aby z volající funkce na zásobníku nic nezbylo, argumenty proto musí volaná funkce dostat do své oblasti.

Zásobník ve funkci s `N` sloty vypadá takto (relativně vůči vrcholu zásobníku v registru `%esp`):

```
4*N+4  :  return address
  4*N  :  slot 0 (argument 0)
4*N-4  :  slot 1 (argument 1)
          ...
    8  :  slot (N-2)
    4  :  slot (N-1)
    0  :  closure value
```

Volaná funkce dostane argumenty ve svých prvních slotech, které se nachází těsně pod návratovou adresou (zapsanou instrukcí `call`). Ostatní sloty jsou umístěny níž. Hodnota funkce je umístěna v registru `%ecx`, počet argumentů u neznámých volání v `%eax`. Hodnotu z `%ecx` funkce zapíše na začátek svého zásobníku, aby bylo možno projít zásobník při úklidu paměti. Návratovou hodnotu pak funkce zpět na místo volání předá v registru `%eax`.

Volání funkce proběhne tak, že se argumenty umístí pod prostor obsazený volající funkcí a instrukcí `call` se na zásobník uloží návratová adresa. Při koncovém volání argumenty přepíšou sloty volající funkce, zásobník se posune zpátky nahoru a do funkce se provede skok (instrukcí `jmp`). Návratová adresa tak zůstane nezměněna a při návratu z koncové funkce dojde ke skoku do původní volající funkce.

Jelikož všechny proměnné jsou uloženy v paměti, může generátor kódu použít pevně dané registry pro dočasné uložení hodnot. Registry `%eax` a `%edx` se používají při přesunech hodnot v paměti, v registru `%ecx` je uložena hodnota aktuální funkce, která se využívá při přístupu k zachyceným hodnotám. V jistých speciálních případech je rovněž využit i registr `%ebx`. Dále funkce přistupuje k registru `%edi`, ve kterém je uložen ukazatel na aktuální pozadí (`Bg*`) běhové knihovny.

4 Běhová knihovna

Přeložený program potřebuje ke svému běhu podpůrnou běhovou knihovnu (*runtime*). V této knihovně jsou definovány základní funkce, které využívá standardní knihovna, a rovněž se zde nachází kód pro automatickou správu paměti.

Je špatnou programovací praxí používat nekonstantní globální proměnné, proto jsou veškerá „globální“ data uložena v objektu nazývaném pozadí (*background*, kvůli všudypřítomnosti zkráceno až na *bg*). Odkaz na tento objekt je skrz funkce ve Spiral přenášen v registru `%edi`, v běhové knihovně pak explicitně jako první argument většiny funkcí.

4.1 Reprezentace hodnot

Všechny hodnoty jsou reprezentovány jako 32-bitové číslo. Hodnoty celých čísel jsou reprezentovány přímo, a to vynásobené dvěmi, tudíž jejich nejnižší bit je nulový. Takto je možno uložit 2^{31} čísel, takže rozsah celých čísel v jazyku Spiral je od -2^{30} do $2^{30} - 1$. Ostatní hodnoty jsou uloženy jako ukazatel do paměti a jsou od celých čísel odlišeny tak, že jejich nejnižší bit je nastaven na jedničku. Aby bylo možno rychle rozpoznat funkce, mají datové objekty (tedy všechny, které nejsou funkce) poslední dva bity nastaveny na 01, funkce pak na 11 (binárně). Původní ukazatel obdržíme snadno bitovou operací *and*.

```
 <int>0 ... integer
<ptr>01 ... data object
<ptr>11 ... function
```

Všechny objekty v paměti mají v prvních 4 bajtech uložen ukazatel na statickou tabulku objektu (*object table* neboli *otable*). Pomocí této tabulky lze dynamické objekty rozlišovat a pracovat s nimi. Tabulka kromě ukazatelů na funkce sloužící ke správě paměti obsahuje rovněž ukazatele na funkce pro převod objektu na řetězec a pro porovnávání objektů. Tyto funkce jsou využity pro implementaci funkcí `stringify`, `eqv?` a `equal?` standardní knihovny, které musí být schopny pracovat se všemi typy objektů.

```
struct ObjTable {
  const char* type_name;
  void (*stringify_fun)(Bg* bg, Buffer* buf, void* obj_ptr);
  auto (*length_fun)(void* obj_ptr) -> uint32_t;
  auto (*evacuate_fun)(GcCtx* gc_ctx, void* obj_ptr) -> Val;
  void (*scavenge_fun)(GcCtx* gc_ctx, void* obj_ptr);
  void (*drop_fun)(Bg* bg, void* obj_ptr);
  auto (*eqv_fun)(Bg* bg, void* l_ptr, void* r_ptr) -> bool;
  auto (*equal_fun)(Bg* bg, void* l_ptr, void* r_ptr) -> bool;
};
```

Obsah úseku paměti po ukazateli na tabulku objektu je již závislý na typu objektu. Je zde například umístěn ukazatel na data a délka řetězce nebo prvky n-tice. Funkce mají po tabulce objektu uloženu adresu začátku kódu funkce. Před kódem je umístěna tabulka funkce (*fun table* či zkráceně *ftable*), která obsahuje informace o počtu argumentů, slotů a zachycených hodnot, které jsou potřeba pro korektní čtení zásobníku při úklidu paměti.

```
struct FunTable {
  uint32_t slot_count;
  uint32_t arg_count;
  uint32_t capture_count;
  const char* fun_name;
  uint8_t padding_0[16];
};
```

4.2 Správa paměti

Paměť objektů je strukturovaná do úseků (*chunks*), které tvoří spojový seznam. Když program požádá o paměť na uložení objektu, běhová knihovna použije prostor prvního úseku. Pokud v něm již není dostatek prostoru, může knihovna od operačního systému získat další úsek a zařadit jej na začátek seznamu. Pokud však program již alokoval velké množství paměti, je spuštěn garbage collector, který všechny živé objekty zkopíruje do nových úseků paměti a staré úseky poté uvolní k dalšímu použití.

Za živé jsou považovány všechny objekty, které jsou odkazovány z proměnných v daném okamžiku běhu programu a všechny objekty, na které jiné živé objekty odkazují. Nejprve jsou proto evakuovány (tedy zkopírovány ze staré oblasti do nové a zachráněny před zničením) všechny objekty ze zásobníku. Poté jsou evakuované objekty prohledány (*scavenged* – doslova „hledat v odpadcích“) a všechny objekty, na které odkazují, jsou rovněž evakuovány. Aby objekt nebyl evakuován vícekrát a nebyl pokaždé zkopírován, je na jeho místo po evakuaci zapsán ukazatel (*forward pointer*) odkazující na nové umístění původního objektu. Zbylé objekty ze starých úseků paměti jsou zahozeny (*dropped*).

Některé objekty (například pole nebo řetězce) využívají pro uložení svého obsahu další oblasti paměti mimo paměť objektů. Tyto objekty se o správu své paměti starají samy a když jsou zahozeny, svou paměť uvolní.

4.3 Rozhraní mezi programem a knihovnou

Při volání externí funkce v programu v jazyce Spiral se zavolá céčková funkce odpovídajícího jména. Jejími argumenty je odkaz na pozadí (tedy obsah registru `%edi`), vrchol zásobníku (z registru `%esp`) a poté argumenty předané na místě volání. Jelikož runtime je napsaný v C++, jsou funkce určené pro volání ze standardní knihovny umístěny v blocích `extern "C" { ... }`, jelikož překladač C++ jména funkcí mění (kvůli podpoře overloadingu). Některé proměnné, například objekty `true` a `false`, by bylo nepraktické takto definovat, proto se na ně z jazyka Spiral odkazujeme prostřednictvím jejich změněných (*mangled*) jmen.

5 Implementace

Celá implementace je dostupná v gitovém repozitáři ze serveru GitHub na adrese `https://github.com/honzasp/spiral`. V adresáři `src/` se nachází zdrojové kódy samotného překladače, adresář `rt/` obsahuje implementaci běhové knihovny, v adresář `stdlib/` jsou definovány moduly standardní knihovny jazyka Spiral a v adresáři `tests/` jsou umístěny celkové testy. Všechny soubory jsou uvolněny do public domain (viz soubor `UNLICENSE`).

5.1 Překladač

Překladač je napsaný v jazyce Rust [3] a s více než 6000 řádky kódu[2] (včetně jednotkových testů) se jedná o největší část implementace. Celý program je rozdělen do modulů a toto rozdělení úzce souvisí s rozdělením postupu překladu na jednotlivé mezijazyky.

`main` obsahuje hlavní funkci, která vykoná všechny akce zadané uživatelem na příkazové řádce.

`args` zodpovídá za dekódování argumentů z příkazové řádky.

`sexpr::syntax` definuje datové typy s-výrazů.

`sexpr::parse` poskytuje parser s-výrazů.

`sexpr::pretty_print` převádí s-výrazy z interní podoby opět do textového formátu v lidsky čitelné podobě.

`sexpr::to_spiral` umožňuje z s-výrazu získat syntaktický strom jazyka Spiral.

`sexpr::to_spine` z s-výrazu načte definici programu v jazyku Spine (což se využívá pro testování).

`sexpr::to_grit` přečte z s-výrazu definici programu v jazyku Grit (opět využito pro testování).

`spiral::syntax` definuje typy pro syntaktický strom jazyka Spiral.

`spiral::env` poskytuje pomocný objekt prostředí (*environment*) pro průchod syntaktického stromu.

`spiral::imported` slouží pro získání seznamu všech importovaných modulů z jednoho modulu či programu.

`spiral::to_spine` překládá program ze Spiral do Spine, k čemuž využívá:

`spiral::to_spine::mods` k překladu celých modulů,

`spiral::to_spine::decls` k překladu deklarací,

`spiral::to_spine::stmts` k překladů příkazů a

`spiral::to_spine::exprs` k překladu výrazů.

`spine::syntax` definuje syntaktický strom jazyka Spine.

`spine::env` poskytuje prostředí pro průchod tímto stromem.

[2] Započítány jsou pouze řádky, které obsahují alespoň dva neprázdné znaky.

spine::check implementuje kontrolu korektnosti (*well-formedness*), která je využívána pro testování.

spine::eval umožňuje vyhodnocovat (interpretovat) programy v jazyku Spine, čehož se opět využívá při testování.

spine::free umožňuje zjistit všechny volné proměnné ve výrazu, čehož se využívá pro zachycení pšech proměnných v definici funkce.

spine::onion poskytuje typ „cibule“ pro překlad ze Spiral.

spine::to_grit překládá program ze Spine do jazyka Grit.

spine::to_sexpr převede Spine na s-výraz.

grit::syntax definuje syntaxi jazyka Grit.

grit::optimize_dead_vals implementuje optimalizaci nevyužitých hodnot.

grit::optimize_dead_defs zastřešuje optimalizaci nepoužitých definic funkcí a objektů.

grit::optimize_values poskytuje optimalizaci známých hodnot.

grit::optimize_inline implementuje inlining.

grit::interf zkonstruuje pro funkci graf interference.

grit::slot_alloc pomocí grafu interference alokuje proměnným sloty.

grit::to_asm přeloží Grit do interní podoby assembleru.

grit::to_sexpr převede Grit na s-výraz.

asm::syntax definuje interní syntaxi assembleru.

asm::simplify implementuje jednoduchou optimalizaci na úrovni instrukcí.

asm::to_gas generuje vstup pro GNU Assembler.

5.2 Běhová knihovna

Běhová knihovna je implementována v C++ a nemá rozsah ani 2000 řádků. C++ bylo místo čistého C použito především kvůli podpoře jmenných prostorů a `auto`-deklarace proměnných. Možnosti jazyka vyžadující skrytý kód generovaný překladačem (výjimky, RTTI, virtuální funkce, ...) jsou záměrně potlačeny[3] a použití standardní knihovny je omezeno na část odpovídající jazyku C.

5.3 Testy

Jednotkové testy kontrolující funkčnost vybraných částí překladače jsou umístěny přímo v kódu pomocí nástrojů podporovaných jazykem Rust. Hlavní porce testů má podobu malých programů ve Spiral, které jsou automaticky přeloženy a spuštěny. Jejich výstup je poté porovnán s očekávaným výstupem a pokud se liší, je ohlášena chyba. Většina z objemu testů testuje jednotlivé oblasti standardní knihovny, další větší skupinou jsou zátěžové testy, které jsou převážně zaměřené na testování správy paměti.

[3] Pomocí přepínačů `-fno-rtti` a `-fno-exceptions` překladače `clang++`.

6 Závěr

Představili jsme funkční implementaci netriviálního programovacího jazyka včetně běhového prostředí a standardní knihovny. Podporovány jsou plnohodnotné funkce, koncová volání i jednoduchý systém modulů.

Původní autorův záměr byl vytvořit překladač jazyka Plch[4], který měl být prakticky stejný jako Spiral. Jeho první mezijazyk ${}^{p}\lambda_{\chi}$ byl velmi podobný Spine, umožňoval však přímo pracovat s celými čísly a funkcemi a měl proto i jednoduchý typový systém. Druhý mezijazyk `tac` (*three address code*), jehož zjednodušením vznikl Grit, se pak překládal do assembleru za použití skutečného alokátoru registrů, generovaný kód byl tedy poměrně kvalitní. Snaha o vytvoření tohoto překladače ale nakonec selhala.

Rozvržení překladu přes jazyky Spine a Grit tak původně počítalo s tím, že většina optimalizací bude probíhat v „typově bezpečném“ jazyku Spine a Grit bude sloužit jen jako multiplatformní předstupeň assembleru, stejně jako jako jeho neúspěšný předchůdce `tac`. Ukázalo se však, že optimalizovat strukturovaný Spine je obtížnější než jednoduchý Grit, navíc některé důležité aspekty (známá a neznámá volání) nelze v bezpečném netypovaném jazyce vyjádřit stejně přímo jako v Gritu.

I když je jazyk Spine v některých aspektech kvalitnější než jisté široce používané jazyky (například „jazyk“ PHP), trpí některými fundamentálními nedostatky:

- Autor je přesvědčen, že dynamicky typované jazyky dosáhly své popularity především díky absenci vysokoúrovňového praktického jazyka s kvalitním statickým typovým systémem a bezpečnou správou paměti. Na tuto mezeru na trhu je zaměřen právě jazyk Rust, který ji dle autora může zaplnit velmi důstojně. Spiral je jako dynamický jazyk ve vývoji programovacích jazyků dnes již naprosto nezajímavá větev, jelikož se zde tísní spolu s Pythonem, Luou, JavaScriptem a všemi ostatními deriváty Scheme a Lispu vůbec.

- V jazyce chybí možnost pracovat se strukturami (záznamy, objekty, slovníky, tabulkami, ...), tedy n-ticemi, k jejichž prvkům lze přistupovat pomocí symbolických jmen. Takovéto objekty buď vyžadují statické typování (`struct` v C) nebo implementaci pomocí hešovací tabulky či jiné mapovací datové struktury (jako jsou objekty v JavaScriptu, Ruby nebo Pythonu), což je však dle autora příliš velká cena za takto elementární operaci.

Kromě těchto vad, které jsou důsledkem návrhu jazyka, pak implementace trpí množstvím dalších nedodělků, které by pro praktické použití musely být napraveny:

- Kvůli nízké kvalitě generátoru kódu, který ani není schopen umístit proměnné do registrů, a kvůli nutnosti volat externí funkce pro každou elementární operaci, včetně sčítání dvou celých čísel, jsou přeložené programy velmi pomalé. Neblahý vliv na výkon má i nutnost alokovat paměť pro každé použité reálné číslo. Pro nedostatek místa a času není součástí práce reálné srovnání rychlostí programu ve Spiral a programů v jiných jazycích, ovšem výsledky by jistě byly pro Spiral velice nepříznivé. Nekvalitní garbage collector, který při každém úklidu paměti kopíruje všechny alokované objekty, a naivní (a tedy pomalé) algoritmy použité v samotném překladači už jen katastrofickou situaci z pohledu výkonu dovrší.

- Při parsování programu není nikde uchována informace o pozicích ve zdrojovém souboru, takže když nastane chyba při překladu nebo za běhu, není implementace schopna určit ani v náznaku místo, kde chyba nastala, což činí vývoj jakéhokoli jen trochu většího programu prakticky nemožným.

- Výše zmíněný generátor kódu je nejen děsivě neefektivní, ale zároveň velmi nepřehledný a špatně napsaný, a to i ve srovnání se zbytkem překladače.

[4] `https://github.com/honzasp/plch`

- Standardní knihovna je pro praktické použití příliš minimalistická, chybí například funkce pro práci se souborovým systémem nebo základní datové struktury jako haldy nebo hešovací tabulky.
- Jazyk neumožňuje používat funkce s proměnným počtem argumentů, takže standardní knihovna musí exportovat palety funkcí jako `tuple-0` až `tuple-8` nebo `str-cat-0` až `str-cat-8`.
- Zdrojový kód neobsahuje ani jeden komentář, jazyk Spiral dokonce syntaxi na zápis komentářů úplně postrádá.

Na závěr lze tedy říci, že jednoduchý programovací jazyk byl sice podle zadání z názvu práce přeložen, ovšem přínos práce končí tímto konstatováním. Byly použity pouze běžné a v literatuře mnohokrát popsané postupy, navíc často notně zjednodušené.

Literatura

[1] *clang: a C language family frontend for LLVM*. `http://clang.llvm.org/`, [Online; accessed 6.4.2015].

[2] *The LLVM Compiler Infrastructure*. `http://llvm.org/`, [Online; accessed 6.4.2015].

[3] *The Rust Programming Language*. `http://www.rust-lang.org/`, [Online; accessed 6.4.2015].

[4] *Haskell 2010 Language Report*. Technická zpráva, Haskell Comittee, 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

[5] Appel, Andrew W: *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[6] Briggs, Preston, Keith D Cooper a Linda Torczon: *Improvements to graph coloring register allocation*. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(3):428–455, 1994.

[7] Chaitin, Gregory J: *Register allocation & spilling via graph coloring*. ACM Sigplan Notices, 17(6):98–101, 1982.

[8] Chaitin, Gregory J, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins a Peter W Markstein: *Register allocation via coloring*. Computer languages, 6(1):47–57, 1981.

[9] Codenomicon: *Heartbleed Bug*, 2014. `http://heartbleed.com/`.

[10] Grune, Dick, Kees van Reeuwijk, Henri E Bal, Ceriel JH Jacobs a Koen Langendoen: *Modern compiler design*. Springer Science & Business Media, 2012.

[11] Jones, Simon L Peyton: *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*. Journal of functional programming, 2(02):127–202, 1992.

[12] Jones, Simon L Peyton: *Compiling Haskell by program transformation: A report from the trenches*. V *Programming Languages and Systems—ESOP'96*, strany 18–44. Springer, 1996.

[13] Jones, Simon L. Peyton a André L. M. Santos: *A transformation-based optimiser for Haskell*, 1997.

[14] Jones, SL Peyton, Cordy Hall, Kevin Hammond, Will Partain a Philip Wadler: *The Glasgow Haskell compiler: a technical overview*. V *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, svazek 93. Citeseer, 1993.

[15] Kennedy, Andrew: *Compiling with continuations, continued*. V *ACM SIGPLAN Notices*, svazek 42, strany 177–190. ACM, 2007.

[16] Santos, André Luís de Medeiros: *Compilation by Transformation in Non-Strict Functional Languages*. disertace, Univerzity of Glasgow, 1995.

[17] Shinn, Alex, John Cowan a Arthur A. Gleckler: *Revised^7 Report on the Algorithmic Language Scheme*. Technická zpráva, 2013.