VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

LÍHNUTÍ KÓDU A VÝPOČET ZA DOBY KOMPILACE V PROGRAMOVACÍCH JAZYCÍCH

CODE HATCHING AND COMPILE-TIME COMPUTATION IN PROGRAMMING LANGUAGES

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE **Bc. DANIEK ČEJCHAN**
AUTHOR

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

BRNO 2018

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Klíčová slova

líhnutí kódu, beast, programovací jazyk, vykonávání kódu během kompilace

Keywords

code hatching, beast, programming language, compile-time function execution, CTFE

Citace

ČEJCHAN, Daniek. *Líhnutí kódu a výpočet za doby kompilace v programovacích jazycích.* Brno, 2018. Semestrální projekt. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Líhnutí kódu a výpočet za doby kompilace v programovacích jazycích

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Daniek Čejchan
19. října 2018

Obsah

Kapitola 1

Úvod

Tato publikace navazuje na autorovu bakalářskou práci [1], kde je představen a rámcově navržen programovací jazyk Beast a koncept líhnutí kódu. Její přečtení se doporučuje, není však vyžadováno pro porozumění tohoto dokumentu.

Koncept líhnutí kódu je novým konceptem navrženým současně s programovacím jazykem Beast a sjednocuje funkcionalitu spjatou s vykonáváním kódu za doby kompilace, jako například generické programování, reflexe jazyka, volání funkcí během kompilace apod. Koncept je v této publikaci představen znovu, tentokrát z trochu jiné perspektivy, a jeho možnosti jsou srovnávány vůči ostatním kompilovaným programovacím jazykům.

Součástí této práce je také postup v implementaci referenčního kompilátoru Dragon pro jazyk Beast. Zatímco pro bakalářskou práci byla implementována jen minimální funkcionalita nutná k demonstraci základních vlastností konceptu líhnutí kódu, zde rozšiřujeme možnosti jazyka a kompilátoru natolik, aby mohla být efektivně srovnávána a demonstrována síla konceptu líhnutí kódu s ostatními jazyky. K práci je přiložena i aktualizovaná specifikace jazyka v anglickém jazyce.

Kapitola 2

Programovací jazyk Beast

2.1 Stručné představení programovacího jazyka Beast

2.2 Změny oproti bakalářské práci

Kapitola 3

Koncept líhnutí kódu

Kapitola 4

Implementace konceptu líhnutí kódu

Kapitola 5

Výpočty za doby kompilace v jiných programovacích jazycích

V této kapitole srovnáme možnosti konceptu líhnutí kódu se silou ostatních programovacích jazyků, co se výpočtů za doby kompilace týče.

5.1 D

Z doposud prozkoumaných jazyků vykazuje jazyk D možnosti nejblíže podobné těm, které má Beast. Ačkoli se nejedná o všeobecně rozšířený jazyk, je aktivně vyvíjen a udržován od roku 2001[1] a ve své infrastruktuře ho využívají i společnosti jako Ebay a Facebook[2]. Jedná se o silně staticky typovaný kompilovaný jazyk podobný C++, C# nebo Javě. Hlavními rozdíly oproti C++ jsou modulový systém podobný Javě (netřeba psát hlavičkové soubory, deklarace symbolu nemusí předcházet jeho použití v kódu), *garbage collector* a vícenásobná dedičnost omezená pouze na rozhraní.

5.1.1 Generické programování

Generické programování v D je koncepčně podobné tomu v C++. Tedy, šablony zavádějí speciální seznam pro generické parametry; na základě hodnot generických parametrů se vytvoří speciální *instance* dané šablony, která má (v případě funkcí) další, *runtime* seznam parametrů. Takto se dají vytvořit jak generické funkce, tak generické třídy a struktury. Syntakticky je D rozdílné v tom, že nevyužívá klíčového slova **`template`**, ale seznam parametrů je ve standardních kulatých závorkách umístěn v definici za identifikátor definovaného symbolu; u generických funkcí jsou za sebou tedy dva seznamy parametrů, první generický a druhý *runtime*. Při aplikaci se před seznam generických argumentů (který je opět v kulatých závorkách) přidává vykřičník.

[1] Dle https://wiki.dlang.org/Language_History_and_Future
[2] Dle https://dlang.org/orgs-using-d.html

```d
void genericFunction(int genericArgument)(int runtimeArgument) {
  writeln(genericArgument, " ", runtimeArgument);
}

struct GenericType(int genericArgument, GenericTypeArgument) {
  GenericTypeArgument[genericArgument] arrayField;
}

void main() {
  // Also can write genericFunction!3(5);
  genericFunction!(3)(5); // Prints "3 5\n"

  GenericType!(3, int) var;
  var[0] = 1;
  var[2] = 5;
}
```

(5.1) Demonstrace generického programování v jazyce D

Za zmínku ještě stojí také konstrukce **`template`**[3], která by se dala přirovnat ke generickému **`namespace`**. Díky této konstrukci se dá pod jednu sadu generických parametrů spojit více symbolů. Pokud je v *template* symbol se stejným identifikátorem jaký má *template* sám, nabývá výraz instanciace *template* namísto jmenného prostoru *template* přímo sémantiku daného symbolu[4]; to efektivně umožňuje definovat generické proměnné nebo výčty (*enums*), ačkoli přímo ty generický seznam parametrů nepodporují.

```d
template Template(Type) {
  Type var;
  void func() {
    writeln(Type.stringof);
  }
}

template TemplateVar(Type) {
  Type TemplateVar;
}

void main() {
  Template!int.var = 5;
  Template!short.var = 10;

  Template!short.func(); // Prints "short\n"

  writeln(Template!int.var, " ", Template!short.var); // Prints "5 10\n"

  TemplateVar!int = 5;
  writeln(TemplateVar!short); // Prints "0\n"
}
```

(5.2) Demonstrace konstrukce **`template`** v jazyce D

Jako generické parametry lze v jazyce D předávat typy, hodnoty struktur a primitivních typů. Dále lze pomocí klíčového slova **`alias`** předávat sémantiku výrazu zadaného jako argument (sémantikou se myslí cokoli, co výraz v kontextu argumentu zastupoval – typ, proměnnou, přetíženou funkci, ...). Mimo konstruci **`alias`** nelze generickými parametry pře-

[3] https://dlang.org/spec/template.html
[4] https://dlang.org/spec/template.html#implicit_template_properties

dávat data obsahující reference (tedy nelze předávat třídy ani struktury obsahující třídy nebo ukazatele).

```
1  void f(string str) {
2    writeln("string ", str);
3  }
4
5  void f(int x) {
6    writeln("int ", x);
7  }
8
9  template T(alias f) {
10   void T() {
11     f(3);
12     f("asd");
13   }
14 }
15
16 void main() {
17   T!f(); // Prints "int 3\nstring asd\n"
18 }
```

(5.3) Demonstrace konstrukce **alias** v jazyce D

Specializace šablon

Kromě konvenčních omezení typem (např. **template** T(**int** x)) nabízí jazyk D další prostředky pro specializaci šablon. V ostatních jazycích neznámým prostředkem je konstrukce **if**(cond)[5], která se přidává před tělo definice symbolu. Výraz cond v této konstrukci se vykonává za doby kompilace po vyhodhocení generických parametrů; pokud výraz nabývá hodnoty false, daná definice je vyřazena z rezolučního procesu.

```
1  void foo(T)() if(T.sizeof < 2) {
2    writeln("A ", T.sizeof);
3  }
4
5  void foo(T)() if(T.sizeof > 3) {
6    writeln("B ", T.sizeof);
7  }
8
9  void main() {
10   foo!byte(); // Prints "A 1"
11   foo!int(); // Prints "B 4"
12   foo!short(); // Error
13 }
```

(5.4) Demonstrace konstrukce **if**(cond) v kontextu specializace generických konstrukcí

Druhým prostředkem je konstrukce param : expr. Tato konstrukce se aplikuje na deklaraci generického parametru (param zastupuje původní deklaraci). Výraz expr může být buď konkrétní hodnota očekávaná u daného parametru nebo výraz pro typový *pattern matching*[6].

[5] https://dlang.org/spec/template.html#template_constraints
[6] Vychází z https://dlang.org/spec/expression.html#is_expression

```
struct S(alias a_, alias b_) {
  alias a = a_;
  alias b = b_;
  int x;
}

void foo(T : int)() {
  writeln("A");
}

void foo(T : Item[], Item)() {
  writeln("B ", Item.stringof, " ", T.stringof);
}

void foo(T : S!(a, b), int a, int b)() {
  writeln("C ", a, " ", b, " ", T.stringof);
}

void foo(T : S!(a, b), int a : 5, alias b)() {
  writeln("D ", T.stringof);
}

void main() {
  foo!(int)(); // Prints "A\n"
  foo!(int[])(); // Prints "B int int[]\n"
  foo!(S!(2,3))(); // Prints "C 2 3 S!(2,3)\n"
  foo!(S!(5,"asd"))(); // Prints "D S!(5,"asd")\n"
}
```

(5.5) Demonstrace konstrukce `param : expr` pro specializaci generických konstrukcí v jazyce D

Srovnání s jazykem Beast

Generické typy jsou v jazycích D a Beast principiálně totožné, u generických funkcí se ale přístupy rozcházejí. Primárním rozdílem je to, že v jazyce Beast generické parametry sdílejí stejné závorky s *runtime* parametry a nejsou striktně oddělené pořadím (tedy generické a *runtime* parametry mohou být vzájemně prokládány). Nové možnosti uspořádání mohou někdy lépe reflektovat sémantický tok významu parametrů.

```
Void glVertices(Buffer(Int)? buffer, @ctime Int dim) {
  for(Int i = 0; i < buffer.size / dim; i++)
    #resolve("glVertex" + dim.to(String) + "i")(...buffer[i * dim .. (i+1) *
       dim].to(StaticArray(Int, dim)).to(Tuple));
}

Void main() {
  Buffer(Int) buf(1, 2, 3, 4, 5, 6);
  glVertices(buffer, 2); // Calls glVertex2i(x, y) for each pair in the buffer
  glVertices(buffer, 3); // Calls glVertex3i(x, y, z) for each triplet in the
     buffer
}
```

(5.6) Nastínění případu, kdy může být intiutivnější umístění generických parametrů za *runtime* parametry

Syntaktické oddělení generických a *runtime* parametrů může být komplikací také ve chvíli, kdy se v existujícím kódu nahrazuje varianta funkce, která má nějaký z parametrů

generický, za variantu, kdy je daný parametr *runtime* a naopak. V takovém případě je kód třeba ručně přepisovat; současná IDE automatizaci tohoto druhu refaktorizace neumožňují.

Tento nový přístup vyžaduje i revizi automatického odvozování typů a *pattern matchingu*. Zatímco oddělený seznam generických parametrů umožňuje například automatické odvození typu u funkce `max`:

```
auto max(T)(T a, T b) {
  return a >= b ? a : b;
}

auto i = max(3, 5); // T is automatically inferred
```

(5.7) Vzorová implementace generické funkce `max` v jazyce D

...ve sjednoceném seznamu parametrů něco takového možné není. Naivním řešením by tedy bylo:

```
auto max(@ctime Type T, T a, T b) {
  return a >= b ? a : b;
}

auto i = max(Int, 3, 5);
```

(5.8) Naivní implementace funkce `max` v jazyce Beast

Tato implementace by ale vyžadovala explicitní určení typu ve všech případech. Jazyk Beast proto zavádí klíčové slovo **`auto`** i do parametrů funkce, takže je možné definovat funkci `max` takto:

```
auto max(auto a, a.#type b) {
  return a >= b ? a : b;
}
```

(5.9)

Sjednocení seznamů parametrů také vyžaduje přehodnocení konstrukcí pro *pattern matching*. Jazyk Beast tento problém explicitně neřeší. V jazyce Beast je také zavedena konstrukce **`if`**`(cond)`, kde podmínka `cond` rozhoduje, zda se daný symbol bude uvažovat v procesu rezoluce. Funkčnost *pattern matchingu* je v jazyce Beast do jisté míry nahrazena tím, že jsou při rezoluci parametru funkce k dispozici data všech předchozích parametrů. V některých případech je tento přístup dokonce silnější:

```
struct Z {
  alias Y = int;
  int x;
}

void foo(T)(T t, T.Y u) {
  writeln(u);
}

void main() {
  Z z = Z(10);
  foo!Z(z, 5); // Ok
  foo(z, 5); // Error: cannot deduce
}
```

(5.10) Příklad selhání *pattern matchingu* v jazyce D

```
1  class Z {
2    @static @ctime Type Y = Int;
3    Int! x;
4  }
5
6  Void foo(auto t, t.Y u) {
7    print(u);
8  }
9
10 Void main() {
11   Z z;
12   z.x = 10;
13   foo(z, 10); // Ok
14 }
```

(5.11) Příklad funkčního ekvivalentního kódu v jazyce Beast

5.1.2 Vykonávání funkcí za doby kompilace

Kompilátor jazyka D obsahuje interpret, který umožňuje vykonávat funkce za doby kompilace (dále CTFE – *Compile Time Function Execution*). Limity interpretovaného kódu nejsou nijak výrazné – lze volat funkce, i rekurzivně, i dynamicky alokovat paměť; nicméně volání I/O funkcí během interpretace vyvolá chybu. Výsledky interpretace lze předávat jako generické parametry, ale s již dříve zmíněnými omezeními, že předávaná data nemohou obsahovat ukazateles. Ačkoli I/O funkce nelze volat během kompilace ani v Beastu, koncept líhnutí kódu netrpí omezením indirekcí v generických parametrech.

Jádrem tohoto omezení je, že D umí během kompilace (s výjimkou běhu interpretu) ukládat a pracovat s daty pouze jako s literály. V jazyce D tedy nelze mít nekonstantní *compile-time* proměnné. Tuto funkčnost do jisté míry realizuje existence konstruktu **`static foreach`**, která umožňuje během kompilace projít prvky zadaného iterovatelného výrazu.

5.1.3 Reflexe, typové proměnné

Jazyk D, stejně jako Beast, podporuje reflexi za doby kompilace v dostatečné míře pro praktické aplikace. Jsou k dispozici např. konstrukty `__traits(allMembers, Class)` navracející členy zadané třídy, `__traits(getOverloads, f)` navracející jednotlivá přetížení dané funkce a další, popsané v dokumentaci jazyka[7] a standardní knihovny[8].

Práci s těmito konstrukty v jazyce D ale ztěžuje fakt, že typy (a potažmo *aliases*; aliasem rozumíme jakýkoli výraz se sémantikou) nejsou *first-class citizens*, tedy se s nimi nedá pracovat jako s proměnnými, nedají se ukládat do polí apod. V jazyce D existuje kontejner pro aliasy: jedná se o generiku `AliasSeq`[9]; práce s kontejnerem `AliasSeq` je však značně omezená. Nad takovýmto kontejnerem se nedají používat standardní knihovní funkce a operace nad ním se musí implementovat pomocí rekurzivních šablon. Díky tomu vznikají velké syntaktické rozdíly mezi běžným programováním a prací s reflexí.

Demonstrujme tento rozdíl na příkladu: mějme pole číselných hodnot **`int`**`[] arr` a chceme rozhodnout, zda po každém výskytu čísla `5` následuje číslo `3`. Implementace funkce by vypadala takto:

[7] https://dlang.org/spec/traits.html
[8] https://dlang.org/phobos/std_traits.html
[9] https://dlang.org/phobos/std_meta.html#AliasSeq

```
bool test(int[] arr) {
  int previous = arr[0];
  foreach(int current; arr[1..$]) {
    if(previous == 5 && current != 3)
      return false;

    previous = current;
  }

  return previous != 5;
}
```

(5.12)

Podobným způsobem ale v jazyce D nelze řešit například ověření, zda v `AliasSeq` typů **`int`** vždy následuje **`string`**; pro řešení stejným způsobem bychom museli být schopni uchovávat měnit hodnotu `previous`. Možné řešení je toto:

```
// Single-item AliasSeq
template test(Item) {
  enum test = !is(Item == int);
}

// Multiple-item AliasSeq
template test(Item1, Item2, Items...) {
  static if(is(Item1 == int) && !is(Item2 == string))
    enum test = false;
  else
    enum test = test!(Item2, Items);
}

pragma(msg, test!(AliasSeq!(int, string, int, string))); // true
pragma(msg, test!(AliasSeq!(int, string, int))); // false
```

(5.13)

Je zřejmé, že rozdíl v syntaxi těchto dvou příkladů je markantní. V jazyce Beast lze díky konceptu líhnutí kódu řešit obojí stejným způsobem, který se neliší od běžného psaní kódu:

```
@ctime Bool test(Type[] arr) {
  Type previous = arr[0];
  for(Type current : arr[1 .. $]) {
    if(previous == Int && current != String)
      return false;

    previous = current;
  }

  return previous != Int;
}
```

(5.14)

TODO: mixins, template mixins, variadické funkce

5.2 Stručný přehled

Schopnost	**C++**	**Java**	**D**	**Zig**	**Beast**
Generické programování					
Compile-time proměnné *runtime shadowing*[10]					
CTFE[11]					
Typové proměnné					
Reflexe					
Mixins[12]					

[10] *Runtime shadowing*: předávání *compile-time* proměnných za běhu odkazem
[11] *Compile Time Function Execution*: Vykonávání funkcí za doby kompilace
[12] *Mixins*: vyhodnocování textových řetězců jakožto platný kód jazyka

Kapitola 6

Závěr

Literatura

[1] Čejchan, D.: Překladač nového modulárního programovacího jazyka. 2017.