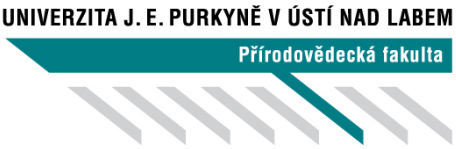
Univerzita Jana Evangelisty Purkyně
v Ústí nad Labem
Přírodovědecká fakulta



Tacit programming - návrh doménově specifického jazyka a implementace jeho interpretu

BAKALÁŘSKÁ PRÁCE

Vypracoval: Oleg Musijenko

Vedoucí práce: Mgr. Jiří Fišer, Ph.D.

Studijní program: Aplikovaná informatika

Studijní obor: Informační systémy

Ústí nad Labem 2023

Cíl bakalářské práce

Cílem bakalářské práce je ukázat výhody a nevýhody tacit přístupu k programování. Výstupem práce bude návrh vlastního doménově specifického jazyka (DSL), který bude využívat tacit programming, a navazující pilotní implementace jeho interpretu. Návrh jazyka by se měl soustředit na následující body: • přehledná syntaxe, • možnosti použití vysokoúrovňových nástrojů pro překlad a podporu běhu programu (např. LLVM v Haskellu) včetně parsování jazyka (např. Parsec v Haskellu), • efektivita při vykonávaní, • případná podpora paralelních výpočtů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a použil jen pramenů, které cituji a uvádím v přiloženém seznamu literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., ve znění zákona č. 81/2005 Sb., autorský zákon, zejména se skutečností, že Univerzita Jana Evangelisty Purkyně v Ústí nad Labem má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona,s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Jana Evangelisty Purkyně v Ústí nad Labem oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

V Ústí nad Labem dne 6. července 2023 Podpis:

Děkuji vedoucímu práce Mgr. Jiřímu Fišerovi, Ph.D.
za neocenitelné rady a pomoc při tvorbě bakalářské práce.

Abstrakt

Tacit programming - návrh doménově specifického jazyka a implementace jeho interpretu

Abstrakt shrnuje základní motivaci práce (kontext), hlavní cíl a následně jednotlivé autorské kroky k jeho splnění (co bylo uděláno od úvodních rešerší, přes návrh, implementaci k případnému nasazení. Minimální rozsah je 800 znaků (maximální půl strany).

Klíčová slova

seznam klíčových slov (obecných termínů vystihujících téma práce) v počtu dva až deset

---

Abstract

Tacit programming - design of a domain specific language and implementation of it's interpreter

Translation of Czech abstract.

Key words

Translation of czech key words.

Obsah

Úvod

Tohle potřebuju kurva přepsat, protože to nedává hlavu ani patu. Kurňa, tohle nedává vůbec nic, jen to jen píčovina.

Existují různá programovací paradigma s kterými se může softwarový inženýr setkat. Mezi nejznámější paradigma patří strukturované, kde počátek tohoto paradigmatu se datuje v 1967 s Dijkstrovo článkem "Goto statement considered harmful". Dijkstra díky 'goto' výrazům nemohl určit správnost programu, proto se 'goto' nahradil za struktury typu 'if else', 'while', atd... Poté se můžeme setkat s objektově orientovaným paradigmatem, který je v dnešní době využíván od menších aplikací až po enterprise systémy. Toto paradigma dává programátorům možnosti využití polymorfismu, není zapotřebí využívát ukazatele na funkce a pomocí dnešních nástrojů je jednodušší se zorientovat v zdrojovém kódu. Jako finální paradigma je zde funkcionální. Funkcionální jazyky omezují primárně omezují mutaci existujících proměnných a zároveň se zaměřují na kompozici funkcí. Existují samozřejmě i další paradigmata, ale mnohdy je na ně pohlíženo jako na kuriozity. Třeba mezi takové patří deklarativní jazyk Prolog, který ověřuje pravdivost výroku v závislosti na předchozích relací.

Hlavně se budeme zaměřovat na tacit - "beztečkové" paradigma. Do tohoto paradigmatu spadá jazyky APL rodiny. Ukážeme si, že i jazyky, které nebyly navržené jako "beztečkové" umožňují v tomto stylu psát.

Tato bakalářská práce předpokládá, že čtenář zná základy funkcionálních jazyků a obzvlášť Haskellu, protože návrh je vytvořen v Haskellu pomocí knihoven Parsec a LLVM.

1 Tacit programming

Tacit programming je programovací styl, který klade důraz na skládání a řetězení funkcí a není založen na explicitní specifikaci parametrů funkcí. Základní principy funkcionálního a taci programování jsou v jazyce JavaScript, jelikož se jedná o jeden z nejvíce populárních programovacích jazyků a v základě má funkcionální možnosti. Detailnější principy jsou psány v Haskellu.

```
fetch("APIURL")
.then(x => fancyFunction(x))
.then(x => console.log(x))
.catch(e => console.error(e))
```

Zde se řetězí funkce zpětného volání (”Callbacks”).

Tento postup je běžný u JavasSript programátorů, ale bohužel má jednu malou nevýhodu. Tvoří se zde zbytečná anonymní funkce (”arrow function nebo-li šipková”) a pokud bychom prohlubovali čím dál víc zásobník volání, mohou nám tyto anonymní funkce zabírat paměť a během debuggingu nám tento styl zápisu ”znečišťuje” zásobník volání.

```
fetch("APIURL")
.then(fancyFunction)
.then(console.log)
.catch(console.error)
```

Přepsaná ukázka je logicky ekvivalentní k té předešlé. Zásadní rozdíl je ten, že se nemusí na paměťový zásobník ukládat kontext anonymní funkce a explicitně se nepředávají parametry funkce. Tudíž se jedná o *tacit* zápis.

Následující úryvek ukazuje, jak funguje **currying** a proč souvisí s tacit programováním.

```
const curry = (f) => a => b => f(a,b);
const sayHello = (a, b) = `Hello ${a} from ${b}`;
const applyToFunctionArray = (input,...args) => args.map(a => a(input))
const partiallyAppliedData = ["A", "B", "C"].map(curry(sayHello));
// [(b) => "Hello A from ${b}",
//  (b) => "Hello B from ${b}",
//  (b) => "Hello C from ${b}"]
```

```
const partiallyAppliedData2 = ["A", "B", "C"].map(curry(sayHello)(1));
// ["Hello A from 1",
//  "Hello B from 1",
//  "Hello C from 1"]
```

Curry funkce transfomuje existujícé funkci tak, že máme pro každý argument vlastní vracející funkci. Z funkce **f(a,b,c,d)** vzniká funkce **f(a)(b)(c)(d)** [1]. V čem je toto výhodné? Například je zde uvedené pole, které se skládá z částečně aplikovaných funkcí. Takto může programátor naiterovat odpověď ze serveru do objektu z předchozí ukázky, které je závislé na třeba na uživatelském vstupu.

Zajímavější část je u *partiallyAppliedData2*. Curryovaná funkce vrací funkci, jež očekává vstupní parametr, aby byla vyhodnocena. Tento princip je důležitý pro lenivé vyhodnocení, který využívá Haskell.

Může zde padnout argument, že v našem případě se curryování nachází pouze pro funkci, která prijímá dva argumenty. Zde je definice funkce, která převádí jakoukoliv funkci na curryovanou.

```
const curry = (f) => (..args) => args.length >= f.length ?
  f.apply(this, args) : (...args2) => curry.apply(this, args.concat(args2));
```

1.1 Principy a odlišnosti od klasického procedurálního paradigmatu

Procedurální paradigma se zaměřuje na psaní procedurálních instrukcí. Typickým příkladem tohoto paradigmatu je programovací jazyk C, protože se jedná o standard, tak v následujících příkladech budu porovnávat jazyk C s jazykem Haskell. Haskell je primárně funkcionální jazyk, tento jazyk umožňuje psát funkce v ”beztečkovém” stylu.

Následující příklad sumace:

Haskell

```
sumCustom:: (Traversable t, Num a) => t a -> a
sumCustom = foldr (+) 0
```

C

```c
int sum(int* arr, size_t numOfElements)
{
    int acc = 0;

    for(int i = 0; i < numOfElements; i++)
    {
        acc += *(arr + i);
    }

    return acc;
}
```

Na příkladu jde vidět, že beztečkový styl zápisu je opravdu kompaktní. V Haskellu není třeba zasahovat do parametrů funkcí. Tento příklad je založen na podstatě tacit programmingu. Co se týče algoritmizace, tacit programming je známý pro vytváření algoritmických řešení pomocí pouze jednoho řádku kódu.

Na dalším příkladě si ukážeme fibonnacciho posloupnost. **Haskell**

```haskell
-- Haskell je lenivý jazyk a proto je možné vytvořit nekonečnou
-- fibonnacciho posloupnost a z té si vzít jen potřebný počet čísel
fibonacci:: Num a => Int -> [a]
fibonacci = (flip take) fibonacciInfinite
  where
    fibonacciInfinite:: Num a => [a]
    fibonacciInfinite = scanl (+) 0 (1:fibonacciInfinite)
```

C

```c
void fibonacci(int* arr, size_t numOfElements)
{
    if(numOfElements > 0)
    {
      arr[0] = 0;
    }
    if(numOfElements > 1)
    {
      arr[1] = 1;
    }
    for(int i = 2; i < numOfElements; i++)
    {
      arr[i] = arr[i - 1] + arr[i - 2];
    }
}
```

Z pohledu imperativního programátora implementace v C je zcela jasná. Funkce přijímá ukazatel na pole a modifikuje toto pole. Zatímco v Haskellu tato implementace může být matoucí. Funkce scanl je velice podobná funkci foldl, jen místo vracení akumulátoru, tak vrací průběžně vypočtené hodnoty.

1.2 Rešerše existujících implementací

2 DSL - principy a využití

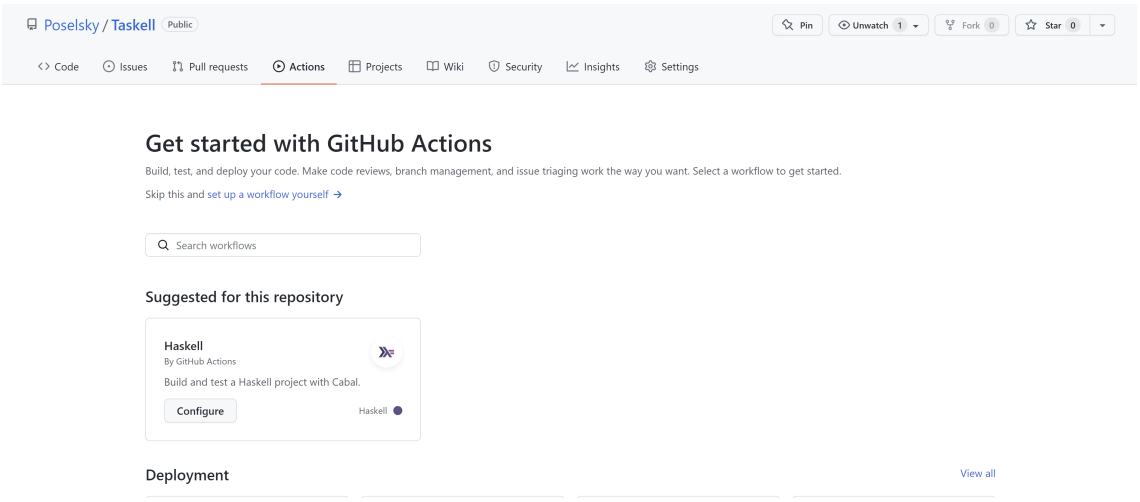
DSL (Domain Specific Language) jsou jazyky, které se zaměřují na specifickou doménu problematiky. Obecně DSL jazyky jsou mnohem jednodušší než jejich plnohodnotné protějšky. Výhodou je, že náročnost učení je mnohem nižší než u GPL (General Purpouse Language). Zároveň při potřebě expertů na specializovaný obor, jako jsou například doktoři nebo architekti, tak ti nepotřebují znát detaily implementace algoritmů, ale místo toho pokud budou mít přístup rovnou k DSL - výpočet šikmosti stěny budovy, hodnota cukrů v krvi pacienta, tak mohou plnit svojí práci o mnohem efektivněji. [2]

Jedním z nejrozšířenejších DSL jazyků je ze světa webu a to **HTML a CSS**. HTML se zaměřuje na vytvoření rámce pro zobrazení textu, zatímco CSS se zaměřuje na stylizaci webu pomocí DOM selectorů. Pravdou je, že pro CSS se nenachází žádný protocol a proto v různých webových enginech, můžete dostat různé výsledky. Příkladem z praxe je zpracování fontů.

Též existují jazyky DSL, které jsou specifické pouze pro jednu dannou enterprise aplikaci, kde její implementace často spočívá na bázi XML nebo podobného formátu jako je např YAML. Zde DSL slouží například pro zjednodušení UI nebo business logiky. Třeba pro porovnání **XAML** pro .NET platformu zjednodušuje logiku, stylizuje UI a zároveň zbavuje potřeby tvoření ”glue” kódu.

Další jazyk který je velice využíván v hardwarovém prostředí je **VHDL** nebo **Verilog**. Tyto DSL jsou zaměřená na simulaci obvodů pomocí FPGA (hradlových polí). Pro building C/C++ projektů existuje **makefile**. Jedním ze zajímavějších DSL je DSL pro ”continuous integration and deployment”. Různé firmy co nabízejí online repositáře se v tomto budou trochu lišit, ale většina z nich poskytují jakousi formu automatizace vydání programu do oběhu. Toto poskytují firmy jako je GitHub, GitLab nebo Azure Dev Ops. Na GitHubu pomocí YAMLu můžete sepsat konfigurační soubor na testování a deployment.

Obrázek 2.1: Výstřižek z GitHub Actions



3 Návrh vlastního DSL

Pro návrh DSL je hlavní vědět o jakou doménu problematiky se jedná. Zatím neexistuje žádná DSL implementace pro konkurenci či paralelizaci vysokého objemu dat. Příkladem vysokého počtu dat je vzorek signálu a detailnější zpracování takového vzorku je časově velice náročné. Tato časová náročnost může být vyřešena právě zmíněnou konkurencí, či paralelizací problému. Toto DSL je pojmenované jako **Haskallyzer**. Pro řešení této problematiky byl zvolen Haskell, jelikož se zdá jako nejoptimálnější. Pro rozbor jazyka byly komunitami vytvořené knihovny (Parsec, MegaParse, AttoParsec), obsahuje mechaniky tacit programmingu, je staticky silně typovaný a díky monádám, řešení okrajových případů je snadné.

Návrh danného jazyka:

```
[CompileTime]
{
  let exampleCSV = "example.csv" :
    (a,Int)
    (b,Float)
    (c,String)
}

let exampleConcurrentProcess = exampleCSV | kalmanFilter
                                          | gaussianFilter


let exampleNestedConcurrentProcess = exampleCSV | kalmanFilter | sum
                                                               | product
                                                | gaussianFilter

let exampleGUIMainLoop = mainLoop | calculateMainState -> writeToEventQueue
                                  | gatherEventQueue -> fireEvents
```

3.1 Vysvětlení gramatiky jazyka

Celý proces je závislý na *Template Haskell* mechanismu. Díky tomuto mechanismu jsou k dispozici části kompilátoru, které umožní generovat kód dle specifikace.

Vytvoří se funkce *exampleCSV*, která vrací obsah csv souboru. Při procesu kompilace se provádí kontrola, zda v csv souboru existuje dvojice ”(a, Int)”, kde ”a” představuje název sloupce a všechny hodnoty ve sloupci ”a” jsou typu ”Int”. Díky atributu *CompileTime* je možné vytvořit funkci *exampleCSV* bez nutnosti použití IO monády. Jednou z nevýhod této metody je, že při spuštění programu se zaplní paměť, protože obsah csv souboru je součástí samotného spustitelného programu. Nicméně, díky tomu není nutné používat IO monádu a obsah csv souboru je k dispozici kdekoliv v programu.

Funkce *exampleConcurrentProcess* vytvoří funkci typu `IO ([a],[b])` a předpokládá, že v programu jsou definované a implementované funkce `kalmanFilter:: CSV -> [a]` a `gaussianFilter:: CSV -> [a]`. Výsledné IO monádě se nejde vyhnout, jelikož se jedná o konkurentní proces, kde vznikají vlákna v jež jsou provedeny výpočty.

Pro vytvoření konkurentního výpočtu je zapotřebí využít *concurrent pipe compostion* `|` operátoru. Každý další *pipe operátor* vytváří další vlákno na kterém je prováděný výpočet. Celá syntaxe je závislá na odsazení, tudíž všechny *pipe operátory* musí mít stejné odsazení.

Příklad s funkcí `let exampleNestedConcurrentProcess` ukazuje, že *pipe operátory* se dají vnořovat. To znamená, že funkce sum i product musí mít typ `sum:: (Num a, Num b) => [a] -> b`. Výsledná funkce bude vygenerováná jako typ `exampleConcurrentProcess:: (Num a, Num b) => IO((a,b), [c])`.

Poslední příklad s `let exampleGUIMainLoop` poukazuje, že není potřeba využít toto DSL pouze pro analýzu dat, ale i pro definování kritických business části programu. Nedílnou součástí GUI aplikací je *EventQueue*, kde se zaznamenávají všechny interakce uživatele a program může s těmito interakcemi pracovat.

4 Implementace interpretu navrženého DSL

Prvopočáteční implementace zahrnovala převedení jazyka

5 Ověření použitelnosti (testování funkčnosti, praktické příklady využití)

Čím více kódu a fíčur v projektu tím obtížnější je definovat, kde se nachází problém a jak se distribujují data. Haskallyzer má výhodu oproti konvenčnímu psaní kódu tu, že ty nejkritičtější části kódu jsou sepsány mimo business řešení a nabízí širší pohled na *dataflow*.

6 Závěr

7 Citace

[3] [4]

Bibliografie

Currying partials (2021). Unknown, Open source. URL: https://javascript.info/currying-partials.

Federico, Tomasetti (2021). *Domain Specific Languages*. Itálie, Strumenta S.R.L. URL: https://tomassetti.me/domain-specific-languages/.

Katuščák, Dušan, Barbora Drobíková a Richard Papík (c2008). *Jak psát závěrečné a kvalifikační práce. jak psát bakalářské práce, diplomové práce, dizertační práce, specializační práce, habilitační práce, seminární a ročníkové práce, práce studentské vědecké a odborné činnosti, jak vytvořit bibliografické citace a odkazy a citovat tradiční a elektronické dokumenty*. Nitra: Enigma. ISBN: 978-80-89132-70-6.

Lattner, Chris (2008). *Intro to LLVM*. Erice, Sicily: Chris Lattner. URL: https://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf.