

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 vykonávání,
- případná podpora paralelních výpočtů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a použil jen pramenů, které cituji a uvádím v příloženém 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 17. září 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

Klíčová slova

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	11
0.1 Procedurální paradigma	11
0.2 Objektově orientované paradigma - OOP	11
0.3 Funkcionální paradigma - FP	11
0.4 Cíl práce	12
1 Tacit programming	13
1.1 Principy a odlišnosti od klasického paradigmatu	14
1.2 Debugging	16
1.3 Rešerše existujících implementací - APL	17
2 DSL - principy a využití	19
2.1 Web a enterprise	19
2.2 Grafické DSL	19
2.3 Ostatní DSL	20
3 Návrh vlastního DSL	21
3.1 Vysvětlení gramatiky jazyka	22
4 Implementace interpretu navrženého DSL	23
4.1 Implementace pomocí LLVM	23
4.2 Parsec a kombinátory	23
4.3 Lexer a hlavní datové typy	24
4.4 Parser	27
4.5 Abstraktně syntaktický strom - AST	27
4.6 Testování	27
4.7 Template Haskell	27
5 Ověření použitelnosti (testování funkčnosti, praktické příklady využití)	29
5.1 Praktické využití	29
6 Závěr	31
7 Citace	33

Úvod

Programovací paradigma je způsob myšlení a přístupu k návrhu, strukturování a implementaci počítačových programů. Definuje sadu pravidel, postupů, technik a konceptů, které určují způsob, jakým se programy píší a organizují. Paradigma poskytuje rámec pro definici a řešení problémů v programování.

Některé z nejznámějších programovacích paradigmat zahrnují:

0.1 Procedurální paradigma

Zaměřuje se na sekvenci instrukcí, které jsou vykonávány postupně. Program je rozdělen na procedury a funkce, které provádějí určité operace. Příkladem takového paradigmatu je jazyk C, GOlang a Assembly. Zde se programátoři často setkávají s nutností manuální správou paměti (*malloc*, *free*).

0.2 Objektově orientované paradigma - OOP

Klade důraz na objekty a jejich interakce. Program je strukturován kolem tříd, které obsahují data (atributy) a metody (funkce), které s těmito daty pracují. Toto paradigma je obohaceno o **polymorfismus**. Vývojáři si mohou OOP představit jako nadmonžinu Procedurálního paradigmatu.

V závislosti na programovacím jazyce, vývojáři mohou využívat automatickou správu paměti díky *garbage collectoru*, kde tuto správu paměti využívají C# a Java. Pokud je zapotřebí manuální správa paměti, je zde C nebo Rust. Rust je zajímavý tím, že využívá *borrow checker* a má napodobit chování smart pointerů.

0.3 Funkcionální paradigma - FP

Jedná se o deklarativní způsob programování, kde funkce jsou považovány za základní stavební bloky programu. Funkcionální jazyky mají za cíl minimalizovat mutaci dat a preferovat neměnné *immutable* struktury. To přispívá ke stabilitě, zjednodušené paralelizaci a eliminaci některých typů chyb. Jazyky jako Lisp a jeho dialekty, oCaml, Closure, F# a Haskell jsou běžnými příklady funkcionálního programování.

V jednotlivých funkcionálních jazycích je povolena různá míra mutace dat. Například jazyk F# umožňuje mutaci dat kdekoli v programu, což je částečně záměrem, aby oslovil uživatele jazyka C#. Na druhou stranu, v jazyce Haskell jsou mutace omezeny v IO monádě a data musí být uloženy ve specifických typech jako IORef, STRef nebo MVar. Takto Haskell pomáhá udržet jasnou separaci mezi čistým funkcionálním kódem a kódem, který se zabývá měnícím se stavem nebo interakcí s okolím.

Je důležité si uvědomit, že míra povolené mutace dat se může lišit mezi jednotlivými funkcionálními jazyky a je závislá na jejich návrhu a filozofii. Každý jazyk si volí kompromis mezi funkcionality a striktností v oblasti mutace, aby splňoval požadavky svých uživatelů a cílů, které si klade.

0.4 Cíl práce

Cílem práce je vytvoření doménově specifický konkurenční jazyk, který usnadní přehled business logiky alepší vývojářské prostředí (developer experience) se zaměřením na *tacit* - "bezpečkové" paradigma. V práci budou ukázky, jak *tacit* programming vypadá a budou vyzdvýženy argumenty proč s *tacit* programmingem vůbec pracovat. Do tohoto paradigmatu spadají jazyky APL rodiny. Ukázky v této práci potvrdí, že jazyky které nebyly primárně navrženy jako "bezpečkové" 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 jako je Parsec a MTL (Monad Transformer).

1 Tacit programming

Tacit programming je programovací styl, který klade důraz na skládání a řetězení funkcí a není založen na explicitní specifikaci parametrů funkcí. Pro základní ukázky bude využit JavaScript jelikož se jedná o jeden z nejpobulárnějších jazyků. Základní principy funkcionálního a tacit programování jsou v jazyce JavaScript, jelikož se jedná o jeden z nejvíce pobulárních programovacích jazyků a v základu má již 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 JavaScript 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 debugingu 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 transformuje existující 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 přijímá 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 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

```

int sum(int* arr, size_t numElements)
{
    int acc = 0;

    for(int i = 0; i < numElements; 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 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

```
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 vrácení akumulátoru, tak vrací průběžně vypočtené hodnoty.

1.2 Debugging

Debugging je zásadní činností při vývoji softwaru, která umožňuje identifikovat, analyzovat a odstraňovat chyby ve zdrojovém kódu. Proces debuggování je obzvláště důležitý v imperativních a objektově orientovaných jazycích, které často disponují vyspělými debugovacími nástroji. V těchto jazycích je očekáváno sekvenční vykonávání instrukcí, což usnadňuje postupné sledování jejich provádění. Inspekce zásobníku volání představuje další přirozenou součást debuggingu v těchto jazycích.

V případě lenivého jazyka Haskell však debugging přináší značné obtíže. Haskell využívá mechanismu lenivého vyhodnocování, což znamená, že hodnoty jsou vypočteny až ve chvíli, kdy jsou skutečně potřeba. Tato vlastnost komplikuje proces sledování výpočtu a identifikaci chyb. I přes existenci několika debuggovacích nástrojů pro Haskell může debugging pro zkušeného vývojáře představovat opravdovou výzvu. Zmatek může vznikat zejména při určování, kde a jak byla konkrétní proměnná získána, neboť její hodnota je vypočítána až v okamžiku, kdy je použita.

Naštěstí Haskell nabízí možnost využití REPL (Read - Eval - Print - Loop) prostředí, které umožňuje interaktivní evaluaci výrazů a postupné zkoumání jejich chování. REPL tak může sloužit jako užitečný nástroj pro rychlé testování a experimentování s funkcemi a výrazy. Přítomnost REPL v Haskellu zčásti kompenzuje obtíže spojené s debuggingem a poskytuje prostředí pro analýzu a ladění kódu.

1.3 Rešerše existujících implementací - APL

Jazyk APL je jazyk orientovaný na pole nebo-li *array oriented programming language* se syntaxí zaměřenou na tacit programming. V APL jsou všechny data reprezentovány jako pole a všechny pole jsou skládány ze skalárů a nad nimi jsou dělané matematické operace s poněkud netradiční syntaxí. Totiž každá předem jazykem definovaná funkce je přidělená ke speciálnímu UTF-8 charakteru. Například funkce pro přirozený logaritmus je '⊗' nebo zaokrouhlení nahoru či dolů jsou '⌈' '⌊'. Každá funkce se dá zapsat monadicky či dyadicky.

Monadický zápis má argument operandu na pravé straně: `-5 ⍝ monadic`

Dyadický zápis má argumenty na obou stranách operandu: `10-7 ⍝ dyadic`

Jelikož všechny pole jsou tvořeny skaláry, tak operace "ignorují" strukturu pole a pracují přímo s obsahem pole.

```

    ⍲        ⍝ 6 integers starting from the origin.
0 1 2 3 4 5
    1+⍲      ⍝ Add 1 to the 6 integers starting from the origin.
1 2 3 4 5 6
    2×⍲      ⍝ Multiply by 2 the 6 integers starting from the origin.
0 2 4 6 8 10

```

Pokud pole jsou u dyadického zápisu stejné délky, tak jsou hodnoty vypočteny jako jednotlivé skaláry.

```

    100 0 1 × 2 3 4
200 0 4

```

Operátory jsou vyhodnocovány z prava do leva.

```

    24 ÷ 12 6 - 4 2    ⍝ -> 24 ÷ 8 3
3 6

```

Booleany jsou reprezentovány jako 1 (True) a 0 (False). Zde je ukázka algoritmu, kde se sečte sekvence přirozených čísel (včetně nuly, ale při součtu nemá vliv), která jsou dělitelná buď třemi nebo pěti.

```

    Euler1 ← {+ / ((0=3|ω) ∨ (0=5|ω)) / ω}

    Euler1 ⍲1000
234168

```

Funkce `(0=3|ω)∨(0=5|ω) ⌘ '|' - modulo, '=' is equal operator` zjišťuje zda-li je číslo dělitelné třemi nebo pěti a vrací 0 nebo 1. Operátor `/` funguje jako replicate (pokud je spojené s dalším operátorem, tak funguje jako reduce). Jelikož je vyhodnocování z prava do leva, tak pokud argument funkce je například 7 a 12 tak:

```
((0=3|7 12)∨(0=10 | 7 12))/7 12 ⌘ Result: 12.
```

Toto celé funguje jako filtrování, kombinací operátorů `+/` se udělá suma výsledků.

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 Purpose Language). Zároveň při potřebě expertů na specializovaný obor, 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 svoji práci o mnohem efektivněji. [2]

2.1 Web a enterprise

Jedním z nejrozšířenějších 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.

Těž 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, který je vygenerován automaticky.

2.2 Grafické DSL

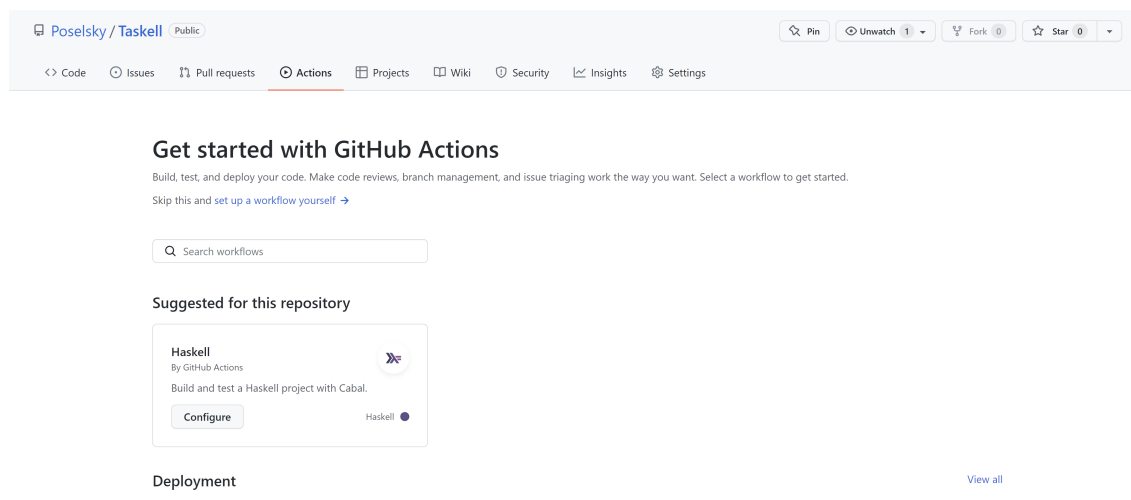
DSL se též týká programů co běží na grafických kartách. Renderovací programy jsou též známé pod pojmem *shader*. Každá grafická knihovna má vlastní DSL jazyk, který jsou velmi podobné jazyku C. Všechny renderovací jazyky prochází takzvanou *graphic pipeline*. OpenGL a Vulkan využívají pro rendering OpenGL Shading Language (GLSL). OpenGL kompiluje GLSL během runtime programu zatímco Vulkan využívá předkompilovaného GLSL bytecode nazývaný SPIRV.

Grafické karty se nevyužívají pouze pro rendering jelikož mají širokou škálu využitelnosti. Třeba *CUDA* vyvinutý firmou NVIDIA využívá dnešní architekturu grafických karet pro paralelní výpočet velkoobjemných dat, kde tuto techniku využívají dnešní algoritmy pro strojové učení.

2.3 Ostatní DSL

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 kompilaci projektů existuje **makefile** a je nejčastěji spárován s C/C++. Jsou zde 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 se dají sepsat konfigurační soubory na testování a deployment.

Obrázek 2.1: Výstřížek 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 neoptimálnější. 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 daného 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 composition* | 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 vygenerována 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 částí 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

Návrh jakéhokoliv DSL (a nejen DSL, ale i programovacího jazyka celkově) zahrnuje **Lexer**, **Parser** a **Abstraktní syntaktický strom (AST)**.

Lexer má za úkol přečíst soubor a najít jednotlivé tokeny v daném souboru. Tyto tokeny mohou obsahovat metadata, jako jsou řádek a sloupec, kde se token nachází, jaký token předcházel a jaký následuje atd... Tyto tokeny jsou zpracovány **parserem**, který má za úkol přečíst tokeny a hledat mezi nimi dle předem definované gramatiky vztahy a zpracovat je do abstraktního syntaktického stromu. AST je výsledek parsování a obsahuje všechny definice jazyka.

4.1 Implementace pomocí LLVM

Prvopočáteční implementace zahrnovala využití knihovny LLVM, která má za následek převzít AST a převést tento jazyk do LLVM intermediate representation (IR). Bohužel LLVM se spíše hodí na vytvoření generického programovacího jazyka, než na vytvoření DSL. DSL se dá touto knihovnou vytvořit, ale pro každou vygenerovanou funkci se musí vytvořit binding mezi IR, jazykem C a Haskell. Toto řešení je rozhodně možné, ale zvyšuje to komplexitu projektu a jednotlivé bindings mohou též být zdrojem nechtěných bugů. Proto se sešlo od implementace pomocí LLVM a místo toho ho nahradil mechanismus Template Haskell.

4.2 Parsec a kombinátory

V tradičních imperativních jazycích parsování probíhá pomocí načtení souboru do takzvaného *streamu* a poté se ze streamu načítají jednotlivé znaky na zpracování tokenů. Díky Haskell knihovnám jako je Parsec, AttoParsec nebo Megaparsec, není potřeba zpracovávat jednotlivé znaky, ale stačí definovat kombinátory, které tyto tokeny vrací. Haskelyzer využívá knihovnu Parsec. Zde je následující příklad jednoduchého kombinátoru.

```
import qualified Text.Parsec.Token as Tok
type CustomParser a = Parsec String () a
```

```
data Variant = Double | Integer | String deriving Show
```

```
emptyLexer :: Tok.GenTokenParser String () Identity
```

```
emptyLexer = Tok.makeTokenParser Tok.emptyDef
```

```
integer :: CustomParser Variant
```

```
integer = Tok.integer emptyLexer
```

```
float :: CustomParser Variant
```

```
float = Tok.float emptyLexer
```

```
parseToVariant :: CustomParser [Variant]
```

```
parseToVariant = many $ integer <|> float <|>
```

```
  (lexeme $ (manyTill alphaNum $ try space))
```

Cílem výše zmíněného příkladu je, přečíst soubor a rozparsovat ho na list, který obsahuje 3 různé datové typy a to na *integer*, *double*, *string*. Typový synonym `CustomParser a` je synonymem pro `Parsec String () a`, což znamená, že Parsec čte soubor do streamu typu `String`, nemá žádný stav (Parsec je *Monad Transformer*, který zároveň má v sobě zahrnutou stavovou monádu) a po úspěšném parsování vrací generický typ `a`.

Je nutné zmínit funkci `parseToVariant`. Díky kombinačnímu operátoru `<|>` lze pouze definovat vysokoúrovňovou parsovací logiku bez potřeby manuální jednotlivých tokenů. Tento operátor funguje na bázi **alternativa** nebo **fallback**. Pokud selže parser `integer`, tak se pokusí parsovat `float`, pokud i ten selže, tak vždy se rozparsuje poslední možnost a tou je jakýkoliv text.

4.3 Lexer a hlavní datové typy

Na začátku je důležité si nadefinovat výsledek parsování. Parsec zprostředkovává jednoduchý token parser, který definuje například komentáře, operátory a rezervovaná jména.

```
haskelyzerLexer :: Tok.GenTokenParser String () (IndentT Identity)
```

```
haskelyzerLexer =
```

```
  Tok.makeTokenParser Tok.emptyDef
```

```
  {
```

```
    Tok.commentStart = "{#",
```

```
    Tok.commentEnd = "}#",
```

```
    Tok.commentLine = "##",
```

```
    Tok.reservedOpNames = ops,
```



```

Tok.reservedNames = names,
Tok.identStart = letter,
Tok.identLetter = letter,
Tok.opStart = oneOf " !#$%&*+./<=>?@\\^|_~",
Tok.opLetter = oneOf " !#$%&*+./<=>?@\\^|_~",
Tok.nestedComments = True,
Tok.caseSensitive = True
}
where
  ops = ["+", "*", "-", ";", "->" , ":", ",", "|"]
  names = ["let"]

```

Lze si všimnout, že tato funkce má poněkud zajímavý typ `IndentT Identity`. Jelikož jazyk nepoužívá středníky tak místo toho využívá odsazení, proto se zde využívá pomocný monad transformer `IndentT` z knihovny *indents*.

Dále je zapotřebí si definovat jednoduché binární a unární operátory.

```

data BinOp
  = Plus
  | Minus
  | Times
  | Divide
  deriving (Eq, Ord, Show)

data UnaryOp
  = Not
  | Nroot -- Like SquareRoot but N
  | Exponent
  deriving (Eq, Ord, Show)

```

Nejzajímavější částí je definice datových typů, které jsou zároveň rekurzivní.

```

type OptionalColumnNameWithType = (Maybe String, CsvDataType)
data Schema = Schema VarNamePath [OptionalColumnNameWithType]
  deriving (Show, Eq, Ord)

data Expr
  = BinOp BinOp Expr Expr
  | CsvDataType CsvDataType
  | UnaryOp UnaryOp Expr
  | Var Name [HaskelyzerFunction]

```

```
| SchemaExpr Schema
| LiteralExpr Literal
deriving (Eq, Ord, Show)

data HaskelyzerFunction =
  HaskelyzerFunction Name [Name] -- Function name args
  | Concurrent [[HaskelyzerFunction]]
  deriving (Show, Ord, Eq)

data CsvDataType =
  CsvFloat
  | CsvInt
  | CsvString
  deriving (Show, Ord, Eq)

data Literal =
  Float Double
  | Int Integer
  | String String
  deriving (Show, Ord, Eq)
```

`Data Literal` jsou pouze primitivní datové typy stejně jako `CsvDataType`, rozdílem je že `CsvDataType` předem definuje, zda-li všechny sloupce v CSV souboru jsou očekávaným datovým typem.

Velmi důležitou součástí je `data HaskelyzerFunction`. První část je pouze funkce a její argumenty, druhá část je `Concurrent [[HaskelyzerFunction]]`. Ve výše zmíněných příkladech bylo poukázáno, jak každá vygenerovaná funkce může vytvořit další konkurentní funkce a v nich další vnořené konkurentní funkce. Díky této vnořenosti je potřeba uchovávat funkce v listu listů.

`Var Name [HaskelyzerFunction]` vygeneruje funkci s názvem "Name" a pořadí funkcí v `[HaskelyzerFunction]` určují, v jakém pořadí budou funkce aplikovány, v tomto případě jsou aplikovány zleva do prava.

4.4 Parser

4.5 Abstraktně syntaktický strom - AST

4.6 Testování

4.7 Template Haskell

5 Ověření použitelnosti (testování funkčnosti, praktické příklady využití)

Abychom si ověřili využití Haskelyzeru, hodil by se nějaký praktický příklad.

5.1 Praktické využití

S rozšiřujícím se kódem a funkcionalitou projektu se zvyšuje obtížnost určení, kde se nachází problém a jak jsou data distribuována v daném systému. Jednou z nejkomplicovanějších výzev při psaní softwaru je jeho škálovatelnost. Haskallyzer má výhodu oproti tradičnímu způsobu psaní kódu v tom, že nejkritičtější části kódu jsou odděleny od business řešení a nabízejí širší pohled na tok dat. To může být z jedním hlavních argumentů, proč toto DSL využít pro větší projekty, protože usnadňuje jeho škálování. Vývojáři mají možnost určit, které části jsou nejdůležitější pro daný cíl a zapsat je do Haskallyzeru.

Konkurence má výhodu v tom, že obecně zvyšuje škálovatelnost softwaru, ale zároveň přináší složitější stav a zvyšuje riziko výskytu chyb. Haskallyzer není pouze DSL pro vnější zápis kritických částí softwaru, ale také umožňuje zápis konkurentních výpočtů v snadno čitelné formě. Tím vzniká určitá forma samo-dokumentace, kterou lze statickým jazykovým analyzátozem například převést do UML zápisu.

6 Závěr

7 Citace

[3] [4]

Bibliografie

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

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

Katuščák, Dušan, Barbora Drobíková a Richard Papík (2008). *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>.