Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Návrh a implementace asynchronní knihovny pro tvorbu toků dat

Diplomová práce

Autor: Jan Najman
Studijní obor: Aplikovaná informatika

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Hradec Králové březen 2026

Prohlášení:

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a uvedl jsem všechny použité prameny a literaturu. V případě využití nástrojů umělé inteligence jsem plně deklaroval způsob jejich využití při vypracování této práce.

V Hradci Králové dne 15. 3. 2026

Jan Najman

Poděkování:

Děkuji vedoucímu diplomové práce prof. RNDr. PhDr. Antonínu Slabému, CSc. za metodické vedení práce a cenné rady, které mi byly poskytnuty během celého procesu psaní.

Abstrakt

Cílem diplomové práce je navrhnout a implementovat asynchronní knihovnu pro vytváření datových toků v programovacím jazyce Rust. Práce se zaměřuje na vytvoření knihovny, která umožňuje typově bezpečnou tvorbu a kompozici toků. Na základě analýzy principů programování datových toků, asynchronního modelu jazyka Rust a souvisejících technik, jako je polymorfismus, monomorfismus a práce s generickými typy, je navržena architektura knihovny, včetně kontextového systému pro sdílení dat mezi uzly. Následně je implementována a otestována. Součástí práce je testování funkčnosti knihovny a experimenty pro vyhodnocení ovlivnění výkonu a škálovatelnosti toků.

Klíčová slova: Rust, toky, kompozice, vizualizace, asynchronní programovaní, uzel, pipeline, workflow, runtime, framework, knihovna, crate, typová bezpečnost

Abstract

Title: Design and implementation of an asynchronous library for creating data flows

The goal of this diploma thesis is to design and implement an asynchronous library for creating data flows in the Rust programming language. The thesis focuses on creating a library that enables type-safe creation and composition of data flows. It is based on an analysis of data flow programming principles, the asynchronous model of the Rust language, and related techniques such as polymorphism, monomorphism, and working with generic types. Based on the results of the analysis, a library architecture is proposed, including a context system for sharing data between nodes, and subsequently implemented and tested. As a part of the thesis, the functionality of the library is tested, and experiments are conducted to evaluate the impact on the performance and scalability of data flows.

Keywords: Rust, flow, data flow, composition, visualization, asynchronous, node, pipeline, workflow, runtime, framework, crate, library, type safety

Obsah

1 Úvod **1**

2 Teoretická část **2**

2.1 Toky dat 2

2.2 Asynchronní programování 2

2.3 Asynchronní programování v Rustu 3

2.3.1 Enum Poll 3

2.3.2 Trait Future 4

2.3.3 Runtime 4

2.3.4 Syntaxe async/await a asynchronních úloh 4

2.4 Polymorfismus 6

2.4.1 Static dispatch 7

2.4.2 Dynamic dispatch 7

2.5 Monomorfizace 9

2.6 Inlining 10

2.7 Trait Any, downcasting a TypeId 12

2.7.1 TypeId 12

2.7.2 Trait Any 12

2.7.3 Downcast metody 13

2.8 Dyn kompatibilita 13

2.9 Smart pointery 14

2.9.1 Box<T> 14

2.9.2 Pin<Ptr> 15

2.9.3 Arc<T> 15

2.9.4 Mutex<T> 16

2.9.5 RwLock<T> 17

2.10 Tuple listy 18

3 Analýza požadavků **20**

3.1 Funkční požadavky 20

3.1.1 Definice uzlů a toků 20

3.1.2 Skládání toků 20

3.1.3 Spouštění toků 20
3.1.4 Zpracování chyb 21
3.1.5 Konfigurace a inicializace uzlů 21
3.1.6 Typová bezpečnost a generika 21
3.2 Nefunkční požadavky 21
3.2.1 Výkon 21
3.2.2 Dokumentace 22
3.2.3 Struktura kódu 22
3.2.4 Přenositelnost a kompatibilita 22

4 Návrh **23**
4.1 Traity pro uzly a toky 23
4.2 Základní toky 23
4.3 Konstrukce toku 24
4.4 Uložení uzlů v toku 24
4.5 Definice běhu toku 25
4.6 Kontextový systém 26
4.6.1 Úložiště typu klíč-hodnota 26
4.6.2 Kontext za pomocí generického typu 28

5 Implementace uzlů **31**
5.1 Trait Node 31
5.2 Enum NodeOutput 32
5.3 Trait BoxedNode 33
5.4 Makro impl_node_output 33
5.5 Makro node 34

6 Implementace kontextového systému **35**
6.1 Základní traity pro práci s kontextem 35
6.1.1 Trait Fork 35
6.1.2 Trait Update 36
6.1.3 Trait Join 36
6.2 Práce se synchronními a asynchronními úlohami 36
6.2.1 Trait Task 37
6.2.2 Trait SpawnAsync 37

6.2.3 Trait SpawnSync 38
6.3 Lokální úložiště 38
6.3.1 Enum MergeResult 38
6.3.2 Trait Merge 39
6.3.3 Trait LocalStorage 39
6.3.4 Implementace lokálního úložiště 40
6.4 Sdílené úložiště 42
6.4.1 Trait SharedStorage 43
6.4.2 Implementace sdíleného úložiště 44

7 Implementace popisu uzlů a toků **47**
7.1 Enum Description 47
7.2 Struct DescriptionBase 48
7.3 Struct Type 48
7.4 Stuct Edge a enum EdgeEnding 49
7.5 Struct ExternalResource 50
7.6 Privátní funkce remove_generics_from_name 50
7.7 Konfigurovatelný D2 formátovač popisů uzlů a toků 51

8 Implementace toků **52**
8.1 Privátní pomocné typové aliasy 52
8.2 Privátní pomocná struktura enum SoftFailPoll 53
8.3 Privátní pomocný trait ChainDebug 53
8.4 Privátní pomocný trait ChainDescribe 54
8.5 Privátní pomocná makra 54
8.5.1 Makra impl_debug_for_builder a impl_debug_for_flow . . . 54
8.5.2 Makro define_flow 55
8.5.3 Makro define_builder 56
8.5.4 Makro define_flow_and_ioe_conv_builder 57
8.6 Tok SequentialFlow 58
8.6.1 Definice toku 58
8.6.2 Definice builderu 59
8.6.3 Privátní trait ChainRunSequential 60
8.7 Tok OneOfSequentialFlow 60

8.7.1 Definice toku a builderu . . . 61
8.7.2 Privátní trait ChainRunOneOfSequential . . . 62
8.8 Tok ParallelFlow . . . 62
8.8.1 Definice toku . . . 63
8.8.2 Trait Joiner . . . 64
8.8.3 Definice builderu . . . 64
8.8.4 Privátní trait ChainRunParallel . . . 65
8.9 Tok OneOfParallelFlow . . . 66
8.9.1 Definice toku a builderu . . . 66
8.9.2 Privátní trait ChainRunOneOfParallel . . . 67
8.10 Tok Detached . . . 68
8.11 Tok FnFlow . . . 69
8.11.1 Trait Runner . . . 69

9 Testování, dokumentace a dostupnost 71

10 Experimenty 72
10.1 Vliv inliningu a monomorfizace na jediný a samotný uzel . . . 72
10.2 Vliv inliningu a monomorfizace na toky . . . 74
10.2.1 Vliv na skládání toků . . . 75
10.3 Maximální hloubka toku . . . 76
10.4 Maximální šířka toku . . . 77
10.5 Maximální kombinovaná hloubka a šířka toku . . . 77
10.6 Shrnutí výsledků . . . 78

11 Porovnání s existujícími řešeními 79

12 Budoucí vývoj 81

13 Závěr 82

14 Seznam použité literatury 83

15 Seznam obrázků, tabulek a kódů 86

16 Přílohy 91

1 Úvod

V dnešní době je software čím dál komplexnější a komplikovanější. Komplexní software je většinou rozdělen na moduly, které spolu spolupracují a uživateli poskytují jen nezbytná rozhraní pro komunikaci či manipulaci s nimi. Díky tomuto přístupu je možno tyto moduly měnit vnitřně jak je potřeba. Tyto moduly se vnitřně skládají z různých stavebních bloků, jako jsou funkce, různé struktury, primitiva a jejich kombinace.

Jednou z těchto technik je vytváření datových toků, kde každý tok je složen z uzlů, které dělají jen základní jednotku práce, jako je například získání dat ze serveru nebo parsování dat do nějakého formátu. Tyto jednoduché uzly jsou poté skládány do toků, které mají různé mechanismy pro běh a správu těchto uzlů, a tím vytvářejí komplexní operace.

Tato práce se zabývá tvorbou, definicí a limitacemi datových toků, které jsou definovány a jejich požadavky jsou kontrolovány za kompilace programu. Pro dosažení tohoto cíle je vytvořena analýza funkčních a nefunkčních požadavků. Na základě této analýzy je vytvořen návrh pro definici a kompozici toků a uzlů. Implementace poté využívá vytvořeného návrhu a navazuje na něj definicí a implementací různých typů toků, kontextového systému, definicí základních traitů a jednotek.

Tvorba nové knihovny a téma této diplomové práce bylo zvoleno, protože existující knihovny nevyužívají jednu z nejsilnějších vlastností jazyka Rust, což je jeho typový systém, a pro vytváření a kontrolu datových typů používají reflexi za běhu. Toto způsobuje, že chyby, které by mohly být řešeny při kompilaci, jsou řešeny za běhu programu a bez rozsáhlých testů často nejsou zachyceny. Dále také tento přistup způsobuje, že kompilátor není schopen kód programu pořádně optimalizovat, protože nevidí konkrétní datové typy.

2 Teoretická část

Tato část práce se zabývá různými tématy, která tvoří teoretické a kontextové zázemí pro tuto práci. Jsou v ní popsány základní pojmy jazyka Rust, obecné či specifické, které je nutné znát pro následující kapitoly.

2.1 Toky dat

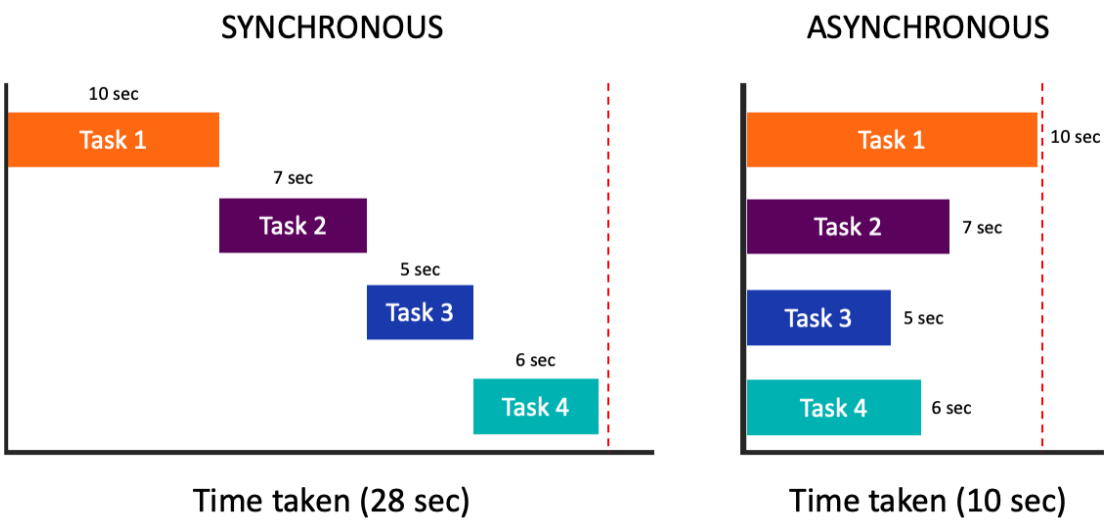
Tok dat je programovací princip, který modeluje program jako orientovaný graf dat proudících mezi operacemi. Díky datovým tokům vznikly lepší, odolnější a škálovatelnější distribuované systémy, lepší kompilátory a lepší databáze. [1, 2]

Klasicky je program modelován jako postup operací, které se dějí v určitém pořadí. Tento postup lze nazvat sekvenčním, procedurálním, řízeným tokem. Oproti tomu programování datových toků klade důraz na pohyb dat a modeluje programy jako řetězce operací. Každé spojení mezi operacemi znamená přesun či kopírování dat do další operace. Každá operace a spojení má jednoznačně definované vstupy a výstupy, které fungují jako černé skříňky. [1, 2]

2.2 Asynchronní programování

Asynchronní programování je efektivní přístup ke zlepšení výkonu systémů s vysokou zátěží, jelikož umožňuje souběžné provádění úloh bez blokování provádění jiných úloh. Na rozdíl od synchronního programování, kde jsou úlohy prováděny postupně, asynchronní programování umožňuje systému zpracovávat více operací současně, což vede k rychlejším odezvám a efektivnějšímu využití prostředků. [3, 4]

Ve srovnání se synchronním programováním nabízí asynchronní programování lepší výkon v aplikacích závislých na I/O operacích, protože umožňuje systému zpracovávat více operací, a to bez nutnosti čekat na dokončení každé z nich. U úloh náročných na výpočetní výkon však může být zlepšení výkonu méně patrné, nebo může dojít ke zpomalení, protože úlohy je stále nutné zpracovávat postupně na jednom jádru procesoru. [3, 4]



Obrázek 1: Porovnání asynchronního a synchronního zpracování [5]

2.3 Asynchronní programování v Rustu

Asynchronní programování v Rustu je provedeno jinak než v ostatních jazycích. Hlavním rozdílem je, že standardní knihovna neposkytuje žádný runtime pro asynchronní úlohy a asynchronní úlohy nejsou okamžitě spuštěny po vytvoření, ale pouze po zavolání `await`. Standardní knihovna poskytuje pouze základní traity a struktury pro vytvoření runtimů a asynchronních úloh. Díky tomuto přístupu si uživatel může vybrat, jaký runtime bude používat a kdy budou asynchronní úlohy spuštěny.

2.3.1 Enum Poll

Kód 1: Definice enumu `Poll` ve standardní knihovně [6]

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Enum `Poll` vyjadřuje, zda asynchronní úloha skončila s nějakým výsledkem, nebo zda je potřeba na tuto úlohu ještě čekat a vzbudit ji později. [6, 7]

2.3.2 Trait Future

Kód 2: Definice trait `Future` ve standardní knihovně [8]

```rust
pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
     ↪  Poll<Self::Output>;
}
```

Trait `Future` je základem pro asynchronní programování v Rustu. Reprezentuje asynchronní úlohu, která vrací nějakou výstupní hodnotu. [7, 8]

Hlavní metodou tohoto traitu je metoda `poll`, která se snaží co nejvíce přiblížit asynchronní úlohu k jejímu dokončení. Tato metoda neblokuje, pokud asynchronní úloha ještě není dokončena, ale místo toho je tato úloha naplánována k probuzení. Kontext předaný metodě `poll` poskytuje takzvaný `Waker`, sloužící k signalizaci, že asynchronní úloha je připravena k probuzení. [7, 8]

2.3.3 Runtime

Runtime je systém, který umožňuje uživatelům spouštět asynchronní kód a má na starost správu a vykonávání asynchronních úloh. Kvůli tomu, že obvykle existuje více asynchronních úloh než je k dispozici jader procesoru, tak runtime plánuje jaké úlohy budou spuštěny a jaké úlohy budou pozastaveny. Je tedy vyžadováno, aby runtime spolupracoval s operačním systémem a bylo tak možné efektivně zpracovat asynchronní I/O a časovače. Obvykle se runtime skládá z reaktoru (nebo event loopu), který komunikuje s operačním systémem, a plánovače, který rozhoduje o tom, kdy a kde jsou úlohy prováděny. [7]

2.3.4 Syntaxe async/await a asynchronních úloh

V Rustu je možno definovat asynchronní úlohu či funkci více způsoby. Definice asynchronní úlohy může obsahovat klíčové slovo `async` nebo vracet nějaký datový type implementující `Futute` trait. Klíčové slovo `async` je možno si představit, jako jednoduchý proces obalení vnitřního kódu bloku či funkce do nové asynchronní úlohy.

Kód 3: Příklad definice bloku vracejícího asynchronní úlohu [Vlastní tvorba]

```
// vytvoření asynchronního bloku pomocí klíčového slova, tento asynchronní blok může obsahovat více čekání asynchronní úlohy, protože je sám o sobě asynchronní úlohou
let blok1 = async {
    ..
    /*nějaká asynchronní úloha*/.await;
    ..
};

// vytvoření bloku bez klíčového slova, tento blok je synchronní a vrací právě jednu asynchronní úlohu a nemůže v něm být použito asynchronní volání
// vše co je obsaženo v tomto bloku bude spuštěno ihned synchronně bez čekání, čekat lze jen na vrácenou asynchronní úlohu
let blok2 = {
    ..
    /*nějaká asynchronní úloha*/
};

// spuštění a čekání na dokončení asynchronních úloh postupně
blok1.await;
blok2.await;
```

Kód 4: Příklad definice asynchronní funkce [Vlastní tvorba]

```
// vytvoření asynchronní funkce pomocí klíčového slova
async fn func1() -> f32 {
    /*nějaká asynchronní úloha*/.await;
    return 0.1;
}

// vytvoření asynchronní funkce bez klíčového slova
fn func2() -> impl Future<Output = f32> {
    // zde lze využít toho, že volání této funkce je synchronní a vrací asynchronní úlohu, díky tomu lze provést nenáročné synchronní úkony a poté vrátit asynchronní úlohu, která závisí na těchto úkonech
    // vrácená asynchronní úloha může být jakákoli například i asynchronní blok
    return /*nějaká asynchronní úloha vracející datový typ f32*/;
    // nebo třeba
    async {
        return 0.1;
    }
}

// vytvoření anonymní asynchronní funkce přiřazené do proměnné func3
let func3 = async || {
    /*nějaká asynchronní úloha*/.await;
    return 0.1;
};

// spuštění a čekání na dokončení asynchronních úloh postupně
func1().await;
func2().await;
func3().await;
```

Dále je také možno pro jakýkoli datový typ implementovat trait `Future` a tím umožnit použití tohoto datového typu jako asynchronní úlohu.

Kód 5: Příklad implementace traitu `Future` pro datový typ [Vlastní tvorba]

```rust
// definice vlastního datového typu
struct MojeAsynchronniUloha {
    hodnota: u8,
}

// implementace traitu Future
impl Future for MojeAsynchronniUloha {
    // definice výstupního typu asynchronní úlohy
    type Output = u8;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
     ↪  Poll<Self::Output> {
        // zde je poté definováno, co má asynchronní úloha dělat
        // v tomto případě jen vrací uloženou hodnotu
        return Poll::Ready(self.hodnota);
    }
}

// vytvoření instance
let instance = MojeAsynchronniUloha {
    hodnota: 26,
};
// spuštění a čekání na dokončení asynchronní úlohy
let res = instance.await;
// kontrola výsledku
assert_eq!(res, instance.hodnota);
```

2.4 Polymorfismus

Polymorfismus v programovacích jazycích je schopnost prezentovat stejné „rozhraní“ pro různé datové typy. Zde je třeba zdůraznit, že pojem „rozhraní“ se v tomto případě vztahuje ke způsobu použití třídy a nemusí se tedy nutně jednat o konkrétní pojmenování, které se vyskytuje v některých objektově orientovaných jazycích. [9, 10, 11]

Polymorfismus lze v Rustu používat dvěma hlavními způsoby, a oba mají své výhody a nevýhody, které je třeba zvážit z hlediska výkonu a velikosti souboru. [11, 12, 13]

2.4.1 Static dispatch

První způsob je tzv. „statický výběr“, který využívá generické typy (tzv. parametrický polymorfismus), monomorfizaci (sekce 2.5) a omezení datového typu (pomocí „traitů“, v jiných jazycích označovaných jako „rozhraní“), které poskytují nezbytnou úroveň flexibility a zachovávají typovou bezpečnost, bez nutnosti vytvářet velké množství duplicitního kódu. [11, 12, 13]

Kód 6: Příklad statického výběru [Vlastní tvorba]

```
trait Update {
    fn update(&mut self)
}

fn update_val<T>(t: &mut T)
where
    T: Update
{
    t.update();
}

fn main() {
    let mut val1: TypeA = todo!();
    let mut val2: TypeB = todo!();

    update_val(&mut val1);
    update_val(&mut val2);
}
```

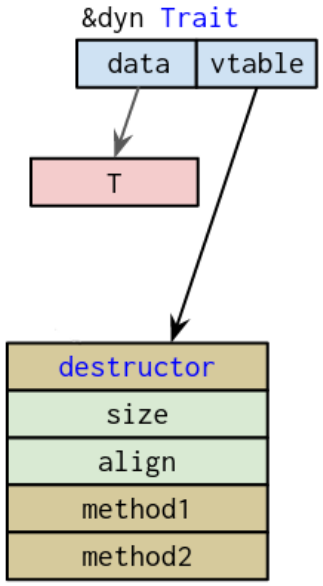
V ukázce výše je vidět, jak lze využít statického výběru. Trait `Update` definuje, co musí konkrétní typy umět. Dále existuje funkce `update_val`, která bere jako parametr měnitelnou referenci na nějaký generický typ `T`, který implementuje trait `Update`. Hlavní metoda poté volá funkci `update_val` na dvou hodnotách s různými typy (konkrétně `TypeA` a `TypeB`), které musí implementovat `Update` trait. V tomto případě by za pomocí monomorfizace (sekce 2.5) byla vygenerována funkce pro každý konkrétní typ.

2.4.2 Dynamic dispatch

Druhý způsob je tzv. „dynamický výběr“, který používá „trait objekty“ k tomu, aby odložil rozhodnutí o tom, který typ je potřebný k vyhovění nějakého polymorfního rozhraní na běh samotného programu. Díky tomuto přístupu se zmenší velikost binárního souboru, protože není použita monomorfizace (sekce 2.5), ale způsobuje snížení výkonu kvůli dodatečnému vyhledávání

během běhu programu. Při tomto přístupu je také explicitně zakázáno používání generických typů. [11, 12, 13]

Trait objekt je ukazatel na část paměti, kde lze najít nejen metody daného typu, ale také i další věci jako je například destruktor. Obvykle se této části paměti říká „tabulka virtuálních metod“ nebo „vtable“. Pro každý konkrétní datový typ s metodami je vytvořena tabulka vtable. Do této tabulky, pak program přistupuje, když potřebuje zjistit, kde se pro daný typ nachází daná metoda. [11, 12, 13]



Obrázek 2: Struktura trait objektu [11]

Kód 7: Příklad dynamického výběru [Vlastní tvorba]

```rust
trait Update {
    fn update(&mut self)
}

fn update_val<T>(t: &mut dyn Update) {
    t.update();
}

fn main() {
    let mut val1: TypeA = todo!();
    let mut val2: TypeB = todo!();

    update_val(&mut val1);
    update_val(&mut val2);
}
```

V ukázce výše je vidět, jak lze využít dynamického výběru. Tato ukázka je velice podobná statickému výběru. Jediné, co se liší je implementace funkce `update_val`, která nyní bere jako parametr měnitelnou referenci na trait objekt `dyn`

`Update`. V tomto případě by byla vygenerována pouze jedna funkce `update_val` pro každý konkrétní typ implementující `Update` trait. Funkce `update_val` poté obsahuje dodatečné vyhledání lokace `update` metody na konkrétním trait objektu za použití vtablu.

2.5 Monomorfizace

Monomorfizace je proces změny generického kódu na specifický kód doplněním konkrétních datových typů. Během tohoto procesu kompilátor hledá všechna místa, kde je generický kód volán, a generuje kopii kódu pro konkrétní datové typy, se kterými je generický kód volán. [14, 15, 16]

Použití monomorfizace je výhodné, protože výsledkem je tzv. „mezijazyk“ (intermediate representation (IR)) se specifickými typy, což umožňuje účinnější optimalizaci. Výsledný kód je většinou rychlejší než dynamický výběr (sekce 2.4.2), ale za cenu delšího času kompilace a větší velikosti binárního souboru. [14, 15, 16]

Kód 8: Příklad kódu před monomorfizací [15]

```
enum Option<T> {
    Some(T),
    None,
}

fn main() {
    let integer = Option::Some(5);
    let float = Option::Some(5.0);
}
```

V ukázce výše je vidět kód před monomorfizací, kde je definovaný vestavěný typ `Option`, který bere jako generický parametr nějaký datový typ `T` a v hlavní metodě je tento datový typ `Option` použit s dvěma různými datovými typy.

Kód 9: Příklad kódu po monomorfizaci [15]

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

```

```rust
fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Po monomorfizaci došlo k vytvoření dvou kopií kódu. Datové typy pro vytvoření kopií byly získány z hlavní metody, kde datový typ `Option` je použitý. Pro datový typ `i32` byla vytvořena kopie `Option_i32` a pro datový typ `f64` byla vytvořena kopie `Option_f64`. Tyto dvě kopie jsou poté použity v hlavní metodě místo generického datového typu `Option`. Tento kód je samozřejmě jen pro ilustraci, ale na této ukázce je nádherně vidět, jak monomorfizace funguje.

2.6 Inlining

Inlining je optimalizace prováděná ručně nebo kompilátorem, při níž se volání funkce nahradí tělem volané funkce. Inlining je podobný makroexpanzi, ale probíhá během kompilace, aniž by došlo ke změně zdrojového kódu. [17, 18]

Funkce bez inliningu často tvoří nemalou část běhu programu. Inlining těchto funkcí odstraňuje vstupy a výstupy (do a z funkcí) a umožňuje kompilátoru provést další optimalizace. Výslednou změnu rychlosti programu lze v nejlepším případě považovat za zanedbatelnou, avšak každé zrychlení se počítá. Nejhorší možný případ by pak byl takový, že je voláno mnoho malých a jednoduchých funkcí (např. `return (a + c) * b` by bylo tělo funkce), kde většina času by byla strávena na vstupech a výstupech těchto funkcí. [17, 18]

Rust pro tento účel obsahuje `inline` atribut, který může mít více významů.

Tabulka 1: Popis `inline` atributu [17, 18]

Atribut	Popis
Bez atributu	Kompilátor sám rozhodne, zda má být funkce inlinenutá. Rozhodnutí závisí jak na nastavení kompilátoru, jako je úroveň optimalizace, tak i na velikosti funkce, zda je funkce generická a zda se funkce nachází v jiné knihovně.
`#[inline]`	Naznačuje kompilátoru, že by funkce asi měla být inlinenutá.

Atribut	Popis
`#[inline(always)]`	Silně naznačuje, že by funkce by měla být vždy inlinenutá, ale kompilátor může požadavek i tak odmítnout.
`#[inline(never)]`	Silně naznačuje, že by funkce by neměla být inlinenutá.

Kód 10: Příklad kódu před inliningem [Vlastní tvorba]

```
fn funkce1(a: i32, b: i32, c: i32) -> i32 {
    return (a + c) * b;
}
fn funkce2(a: i32, b: i32) -> i32 {
    return a * a - b;
}

fn main() {
    let a = 5;
    let b = 12;
    let c = 25;

    let r1 = funkce1(a, b, c);
    let r2 = funkce1(c, a, b);
    let res = funkce2(r1, r2);
}
```

Před inliningem v hlavní funkci jsou volané funkce `funkce1` a `funkce2`, které mají velmi jednoduché tělo funkce a měly by být kompilátorem inlinenuté.

Kód 11: Příklad kódu po inliningu [Vlastní tvorba]

```
fn main() {
    let a = 5;
    let b = 12;
    let c = 25;

    let r1 = (a + c) * b;
    let r2 = (c + b) * a;
    let res = r1 * r1 - r2;
}
```

Po inliningu jsou v hlavní funkci volané funkce nahrazeny tělem těchto funkcí. Zde by samozřejmě kompilátor neskončil a odoptimalizoval i samotné počty, protože již zná všechny operandy (nakonec by v hlavní funkci zbylo jen `let res = 129415`). Tento kód je jen pro ilustraci, na které je vidět, jak inlining funguje.

2.7 Trait Any, downcasting a TypeId

Trait objekty `dyn Any` slouží hlavně k odstínění konkrétního datového typu. Tímto umožňují například vytvoření heterogenních kolekcí a reflexi za běhu programu. Na těchto trait objektech je poté možno použít různé downcast metody, které se pokusí vrátit object s konkrétním datovým typem. K tomuto účelu také slouží `TypeId`, což je identifikátor pro datový typ. [19, 20, 21]

2.7.1 TypeId

TypeId je globálně unikátní identifikátor datového typu. Existuje také funkce `TypeId::of<T>()`, která vrací `TypeId` pro konkrétní datový typ `T`. [20]

Kód 12: Příklad použití `TypeId::of<T>()` funkce [20]

```
use std::any::TypeId;

fn is_string<T: ?Sized + 'static>(_: &T) -> bool {
    TypeId::of::<String>() == TypeId::of::<T>()
}

assert_eq!(is_string(&0), false);
assert_eq!(is_string(&"cookie monster".to_string()), true);
```

2.7.2 Trait Any

Trait `Any` je jedním z hlavních stavebních bloků pro dynamické typování a runtime reflexi. Obsahuje pouze jednu metodu `type_id`, která vrací `TypeId`, konkrétního datového typu. Tato metoda je hlavně využita při použití trait objektu `dyn Any`. Existuje také obecná implementace, která implementuje tento trait pro všechny datové typy, které splňují tzv. `'static` lifetime. [21]

Kód 13: Příklad použití traitu `Any` [21]

```
use std::any::{Any, TypeId};

fn is_string(s: &dyn Any) -> bool {
    TypeId::of::<String>() == s.type_id()
}

assert_eq!(is_string(&0), false);
assert_eq!(is_string(&"cookie monster".to_string()), true);
```

2.7.3 Downcast metody

Downcast metody slouží k získání reference či instance konkretního datového typu z `dyn Any` trait objektu nebo `Box<dyn Any>`. [21]

Tabulka 2: Popis implementováných metod pro varianty traitu `Any` [21]

Metoda	Popis	Implementováno pro
`is<T>`	Vrací, zda je vnitřní datový typ stejný jako datový typ `T`.	`dyn Any`, `dyn Any + Send`, `dyn Any + Send + Sync`
`downcast_ref<T>`	Vrací referenci na vnitřní hodnotu jako datový typ `T`, pokud je datový typ stejný.	`dyn Any`, `dyn Any + Send`, `dyn Any + Send + Sync`
`downcast_mut<T>`	Vrací měnitelnou referenci na vnitřní hodnotu jako datový typ `T`, pokud je datový typ stejný.	`dyn Any`, `dyn Any + Send`, `dyn Any + Send + Sync`
`downcast<T>`	Vrací vnitřní hodnotu jako datový typ `Box<T>`, pokud je datový typ stejný.	`Box<dyn Any>`, `Box<dyn Any + Send>`, `Box<dyn Any + Send + Sync>`
Unchecked downcast varianty	Vrací to samé jako základní varianty, akorát nekontrolují, zda vnitřní datový typ je stejný.	

2.8 Dyn kompatibilita

Dyn kompatibilita se používá u traitů a značí, zda lze trait použít při definici trait objektu. Pokud je trait dyn kompatibilní, tak je možné vytvořit vtable a lze tento trait použít jako trait objekt, ale pokud trait není dyn kompatibilní, tak to znamená, že nelze vytvořit vtable a nelze tento trait použít jako trait objekt. [22]

Kód 14: Příklad dyn kompatibilního traitu [Vlastní tvorba]

```rust
trait DynKompatibilni {
    fn tiskni(&self);
    fn ziskej_hodnotu(&self) -> u32;
    fn nastav_hodnotu(&mut self, val: u32);
}
```

Kód 15: Příklad dyn nekompatibilního traitu [Vlastní tvorba]

```rust
trait DynNekompatibilni {
    fn duplikuj_sebe(&self) -> Self;
      // při použití např. Vec<&dyn DynNekompatibilni> by každá instance typu
      // implementující tento trait vracela jiný výstupní typ

    fn ziskej_typ() -> TypeId;
      // <dyn DynNekompatibilni>::ziskej_typ() není možno zavolat,
      // protože chybí nějaký ukazatel na konkrétní typ (např. &self)
      // a tím pádem není možno přistoupit k vtablu

    fn pridej<T: Into<u64>>(&self, val: T);
      // v tomto případě by vtable musel obsahovat všechny variace této funkce,
      // se kterými tato metoda byla volána,
      // což by způsobilo, že vtable bude obrovský a dále by bylo potřeba, aby
      // kompilátor nalezl všechny invokace přes celý projekt a jeho závislosti
}
```

2.9 Smart pointery

Smart pointery jsou datové struktury, které fungují jako normální ukazatele, ale oproti normálním ukazatelům obsahují dodatečná metadata a funkce. Ve standardní knihovně jazyka Rust je definováno mnoho typů smart pointerů, které poskytují řadu funkcí navíc oproti běžným ukazatelům. [23]

2.9.1 Box<T>

Box je nejspíše ten nejjednodušší typ smart pointeru. Jediné co dělá je, že umožňuje ručně alokovat paměť na heap (haldu). [24, 25, 26]

Funguje skoro stejně jako `malloc` v C, s tím rozdílem, že Box automaticky uvolní paměť, když opustí rozsah bloku či funkce, nebo když je ukončen běh programu. [24, 25, 26]

Kód 16: Příklad použití smart pointeru `Box<T>` [Vlastní tvorba]

```rust
// hodnota u32 na haldě (heap)
let u32_heap = Box::new(1u32);
// hodnota u32 na zásobníku (stack)
let u32_stack = 1u32;
```

2.9.2 Pin<Ptr>

Pin je jednoduchý obal kolem nějakého ukazatele `Ptr` a zajišťuje, že se hodnota ukazatele zafixuje na daném místě v paměti, čímž se zabrání přesunutí hodnoty, na kterou ukazatel odkazuje, nebo jejímu zneplatnění. Toto chování, ale platí jen pro typ neimplementující auto trait `Unpin`, který říká, že s hodnotou můžeme bezpečně hýbat. [27]

Kód 17: Příklad použití smart pointeru `Pin<T>` [27]

```rust
struct AddrTracker { .. }

impl AddrTracker {
    fn check_for_move(self: Pin<&mut Self>) { .. }
}

// vytvoření instance se kterou, ještě můžeme bezpečně hýbat
let tracker = AddrTracker::default();

// vytvoření obalené měnitelné reference na instanci pomocí makra
let mut ptr_to_pinned_tracker: Pin<&mut AddrTracker> = pin!(tracker);
// s instancí již nelze hýbat a může se bezpečně inicializovat
ptr_to_pinned_tracker.as_mut().check_for_move();

// díky pinu jakékoli následující volání funkce `check_for_move` by nikdy
↪  nemělo selhat
ptr_to_pinned_tracker.as_mut().check_for_move();
```

2.9.3 Arc<T>

Arc je ukazatel s atomicky počítanými referencemi, což umožňuje, že alokovaná paměť může mít více vlastníků. Podobně jako Box se paměť alokuje na heap, ale oproti Boxu tato paměť může být bezpečně sdílena mezi vlákny programu. [24, 25, 26]

Kód 18: Příklad použití smart pointeru `Arc<T>` [Vlastní tvorba]

```rust
use std::sync::Arc;

fn main() {
    let sdilena_hodnota = Arc::new("Sdílený řetězec".to_string());

    let sdilena_hodnota_pro_vlakno1 = Arc::clone(&sdilena_hodnota);
```

```
    let vlakno1 = std::thread::spawn(move || {
        println!("{}", *sdilena_hodnota_pro_vlakno1);
    });

    let sdilena_hodnota_pro_vlakno2 = Arc::clone(&sdilena_hodnota);
    let vlakno2 = std::thread::spawn(move || {
        println!("{}", *sdilena_hodnota_pro_vlakno2);
    });

    println!("{}", *sdilena_hodnota);
    vlakno1.join().unwrap();
    vlakno2.join().unwrap();
}
```

2.9.4 Mutex<T>

Mutex se používá pro sdílená data, která je třeba bezpečně upravovat ve více vláknech. Každé vlákno může zamknout tuto sdílenou hodnotu. Zamknutím sdílené hodnoty a držením zámku, vlákno zabraňuje ostatním vláknům v zamknutí a upravě sdílené hodnoty. [24, 25, 26]

Ostatní vlákna mohou zamknout sdílenou hodnotu jen po tom, co vlákno, které drží zámek, odemkne sdílenou hodnotu. Odemknutí sdílené hodnoty nastane, když zámek opustí rozsah bloku či funkce, nebo když uživatel zámek odemkne ručně. [24, 25, 26]

Kód 19: Příklad použití smart pointeru `Mutex<T>` [Vlastní tvorba]

```
use std::sync::Mutex;

fn main() {
    let sdilena_hodnota = Mutex::new(0u8);

    {
        // získání zámku pro hodnotu
        let mut guard = sdilena_hodnota.lock().unwrap();
        // dereference zámku na měnitelnou referenci
        // a přidání 1 k existující hodnotě
        *guard += 1;
        // zámek je na konci tohoto bloku odemčen
    }

    {
        // získání zámku pro hodnotu
        let mut guard = sdilena_hodnota.lock().unwrap();
        // dereference zámku na měnitelnou referenci
        // a přidání 25 k existující hodnotě
        *guard += 25;
        // zámek je odemčen zde ručně
        drop(guard);
        // tento blok poté může pokračovat dále,
        // aniž by zbytečně držel zámek
```

```
    }

    let guard = sdilena_hodnota.lock().unwrap();
    assert_eq!(*guard, 26);
}
```

2.9.5 RwLock<T>

RwLock je zámek používaný pro sdílená data. Tento zámek umožňuje držet libovolní počet referencí pro čtení sdílené hodnoty, nebo maximálně jednu měnitelnou referenci pro úpravu sdílené hodnoty. [24, 25, 26]

Ve srovnání Mutex nerozlišuje mezi čtenáři a zapisovateli, kteří zámek získávají a blokuje tak všechna vlákna čekající na uvolnění zámku. RwLock umožňuje libovolnému počtu čtenářů získat zámek, pokud zámek již nedrží zapisovatel. [24, 25, 26]

Kód 20: Příklad použití smart pointeru `RwLock<T>` [Vlastní tvorba]

```rust
use std::sync::RwLock;

fn main() {
    let sdilena_hodnota = RwLock::new(5u8);

    {
        // získání zámků pro čtení
        let guard1 = sdilena_hodnota.read().unwrap();
        let guard2 = sdilena_hodnota.read().unwrap();
        let guard3 = sdilena_hodnota.read().unwrap();
        // dereference zámků na reference
        // a kontrola, že je součet správný
        assert_eq!(*guard1 + *guard2 + *guard3, 15);
        // zámky jsou na konci tohoto bloku odemčeny
    }

    {
        // získání zámku pro zápis
        let mut guard = sdilena_hodnota.write().unwrap();
        // dereference zámku na měnitelnou referenci
        // a přidání 25 k existující hodnotě
        *guard += 25;
        // zámek je odemčen zde ručně
        drop(guard);
        // tento blok poté může pokračovat dále,
        // aniž by zbytečně držel zámek
        // a blokoval čtenáře či zapisovatele
    }

    let guard = sdilena_hodnota.read().unwrap();
    assert_eq!(*guard, 30);
}
```

2.10 Tuple listy

Tuple list je technika používaná pro vytváření heterogenních polí za pomocí skládání tuplů. Tato technika se používá, protože v Rustu zatím nelze iterovat přes hodnoty nebo definovat trait pro tuply, které mají libovolný počet elementů. Jejich definice je velmi jednoduchá, ale práce s nimi může být matoucí.

Obvykle ve používají tuply o velikosti 2, kde prvnímu elementu se říká hlava (head) a druhému se říká ocas (tail). Tyto tuply se poté dají skládat tak, že vzniknou 4 hlavní variace tuple listů. Typy těchto tuple listů vznikají podle toho, zda má být první element jako první v tuple listu, nebo zda má být první element na první vrstvě tuple listu. Dále ještě musí existovat nějaký typ, který značí konec tuple listu.

Kód 21: Tuple listy reprezentované kódem [Vlastní tvorba]

```
// heterogenní pole za pomocí tuplu
let tuple = (A, B, C, D);
// jako konec tuple listu je zde použit unit typ ()
// první element je na první pozici a první vsrvě
let tuple_list = (A, (B, (C, (D, ()))));
// první element je na první pozici a poslední vsrvě
let tuple_list = (((((), A), B), C), D);
// první element je na poslední pozici a poslední vsrvě
let tuple_list = (D, (C, (B, (A, ()))));
// první element je na poslední pozici a první vsrvě
let tuple_list = (((((), D), C), B), A);
```

Kód 22: Příklad vytištění elementů v tuple listu [Vlastní tvorba]

```
// definice traitu pro vytištění sama sebe
trait Tiskni {
    fn tiskni(&self);
}

// implementace traitu pro všechny elementy,
// které se nachází v tuple listu
impl Tiskni for &'static str {
    fn tiskni(&self) { println!("{self}"); }
}

// implementace traitu pro typ značící konec tuple listu
impl Tiskni for () {
    fn tiskni(&self) {}
}

// implementace traitu pro tuple list
impl<Hlava, Ocas> Tiskni for (Hlava, Ocas)
where
    Hlava: Tiskni,
    Ocas: Tiskni,
```

```
{
    fn tiskni(&self) {
        let (hlava, ocas) = self;
        hlava.tiskni();
        ocas.tiskni();
    }
}

fn main() {
    // sestavení tuple listu
    let tuple_list = ("A", ("B", ("C", ("D", ()))));
    // zavolání traitu na tuple listu
    Tiskni::tiskni(&tuple_list);
}
```

3 Analýza požadavků

V této kapitole je řešena analýza požadavků kladených na navrhovanou knihovnu. Účelem je identifikovat všechny potřeby, které by mělo finální řešení splňovat. Následně jsou definovány funkční a nefunkční požadavky, které tvoří základ pro návrh a implementaci asynchronní knihovny pro tvorbu toků dat.

3.1 Funkční požadavky

Tato kapitola popisuje funkční požadavky kladené na knihovnu. Popisuje konkrétní funkce a chování, které musí knihovna poskytovat pro vytváření asynchronních toků dat. Tyto požadavky definují, jaké funkce by měla knihovna uživatelům poskytnout a jaké operace by měla být schopna provádět.

3.1.1 Definice uzlů a toků

Knihovna musí umožnit definici uzlů a toků, které berou nějaký vstup a vrací nějaký výstup. Definice uzlu a toku by měla obsahovat nějakou funkci (např. `run()`), která umožní uživateli volně pracovat se vstupními daty a transformovat je na výstupní data. Tato funkce musí být asynchronní, aby uživatel mohl pracovat s I/O operacemi efektivně.

3.1.2 Skládání toků

Knihovna musí umožnit skládání uzlů do toků. Tyto toky také musí být možno skládat do dalších toků, a tak vytvářet komplexnější toky. Při skládání uzlů do toku je nutno, aby bylo jasno jak jsou uzly propojeny, a jak uzly budou spouštěny.

3.1.3 Spouštění toků

Spouštění toků by mělo být realizováno pomocí nějaké vhodně nazvané metody, jako je např. `run`. Tato metoda by měla brát vstupní data a vracet asynchronní úlohu, na kterou je možno čekat a získat výsledek, což by měly být výstupní data.

3.1.4 Zpracování chyb

Uzly by měly být schopny vracet chyby, jako jsou výjimky nebo selhání nějaké operace. Tyto chyby musí být součástí výstupního datového typu, aby si jich uživatel byl vědom a řešil jejich možné nastání.

3.1.5 Konfigurace a inicializace uzlů

Knihovna by měla umožnit uživateli definovat vlastní logiku konfigurace uzlu, jako je maximální počet instancí pro tok nebo i pro uzel, nebo sdílení dat mezi uzly. Knihovna nemusí tuto logiku poskytovat, ale měla by jí umožnit. Dále by také knihovna měla umožnit inicializační a demoliční logiku pro uzly.

3.1.6 Typová bezpečnost a generika

Knihovna musí ověřit za kompilace, že toky je možno sestavit a nedojde k tomu, že výstup jednoho uzlu není kompatibilní s očekávaným vstupem následujícího uzlu a celý program spadne. Knihovna by také měla podporovat generické typy pro vstupy a výstupy uzlů, což umožní uzlům pracovat s různými datovými typy.

3.2 Nefunkční požadavky

Tato kapitola se zabývá nefunkčními požadavky, které určují kvalitativní vlastnosti navrhované knihovny. Tyto požadavky nespecifikují konkrétní funkčnost, ale popisují očekávané vlastnosti systému, jako je výkon, spolehlivost a použitelnost.

3.2.1 Výkon

Režie knihovnou definovaných toku by měla být co nejmenší. Toky by měly efektivně využívat systémové zdroje. Neměly by alokovat paměť, která nemusí být využita, a neměly by zbytečně blokovat vlákno a tak zabraňovat dalším úlohám v běhu.

3.2.2 Dokumentace

Knihovna musí mít plnou dokumentaci s ukázkami. Dokumentace a chybová hlášení musí být srozumitelná. Dokumentace prvků knihovny musí vysvětlovat k čemu slouží a jak fungují. Design API knihovny by měl být snadno pochopitelný a intuitivní.

3.2.3 Struktura kódu

Kód knihovny by měl být modulární, dobře strukturovaný a obsahovat testy. Testy by měly testovat funkčnost toků, krajní případy a hlavně by měly selhat již při kompilaci, když nebudou splněny podmínky toků.

3.2.4 Přenositelnost a kompatibilita

Knihovna by měla být kompatibilní s různými operačními systémy, na který Rust běží. Hlavně musí být kompatibilní s operačními systémy Windows a Linux. Knihovna by také měla být kompatibilní pokud možno se všemi asynchronními runtimy v Rustu. Hlavně by měla být kompatibilní s tokio[1] a smol[2] runtimy. Kvůli těmto důvodům by knihovna měla obsahovat co nejméně závislostí, aby bylo možno použít tuto knihovnu pokud možno kdekoliv a s čímkoliv.

[1] https://docs.rs/tokio/latest/tokio/
[2] https://docs.rs/smol/latest/smol/

4 Návrh

Tato kapitola se zabývá prvotním a naivním návrhem základních bloků knihovny, jako je definice uzlu, kontextového systému, struktuře a běhu toků. U těchto témat je navrhována jejich definice, možná omezení a jak by tento návrh vypadal nebo mohl být implementován kódem či jinými vizualizacemi.

4.1 Traity pro uzly a toky

Základním stavebním blokem by měl být nějaký trait `Node`, který obsahuje metodu `run`. Tato metoda musí být asynchronní a brát měnitelnou referenci na typ, který tento trait implementuje, poté musí brát nějaký vstupní typ a vracet nějaký výstupní typ.

Kód 23: Návrh traitu `Node` [Vlastní tvorba]

```
trait Node<Input, Output> {
    async fn run(&mut self, input: Input) -> Output;
}
```

Dále by také mohl existovat trait `Flow`, který by obsahoval specifické metody pro toky. Jednou z takových metod by mohla být metoda `nodes`, která by brala referenci na sama sebe, a vracela by referenci na uzly obsažené v toku.

Zde ale nastává problém - Jaký typ by měla tato metoda vracet? Při přidání uzlu do tohoto toku by se výstupní typ musel změnit, nebo by tato metoda musela vracet list trait objektů `Node`, což není možné protože trait `Node` není dyn kompatibilní (sekce 2.8).

Kód 24: Návrh traitu `Flow` [Vlastní tvorba]

```
trait Flow<Input, Output, Nodes>: Node<Input, Output> {
    fn nodes(&self) -> &Nodes
}
```

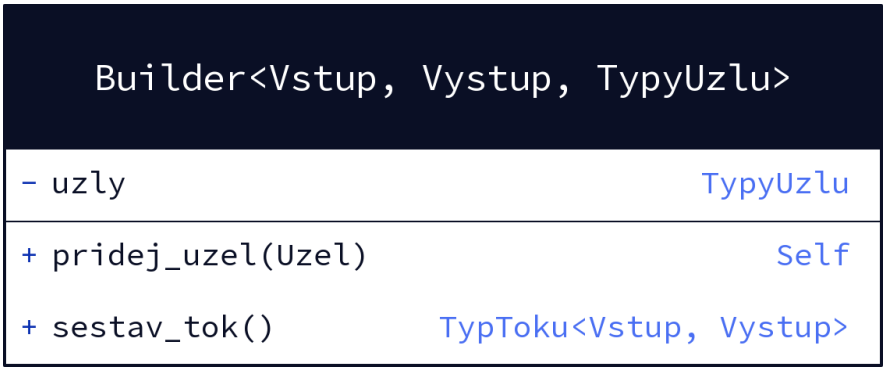
4.2 Základní toky

Knihovna by měla obsahovat základní typy toků, jako je sekvenční, paralelní a fallback tok. Pomocí těchto základních toků je poté možno vytvořit prakticky cokoliv.

4.3 Konstrukce toku

Při konstrukci toku je nutno zajistit, aby uzly na sebe mohly navazovat, bez jakýchkoliv potíží. Dále je nutno zajistit, že vstup do toku lze poslat do prvního uzlu a výstup z posledního uzlu lze vrátit jako výstup z posledního uzlu.

Kvůli těmto požadavkům je nutné měnit datové typy, které tyto toky definují při konstrukci. Za běhu, ale už tyto typy nejsou tak důležité, a proto by konstrukce měla být provedena přes tzv. „builder“. Tento styl konstrukce zajistí, že lze měnit datové typy definice toku a na konci je ukončen metodou, která může vracet úplně jiný typ a tak se například zbavit nadbytečných datových typů v definici, nebo i polí v objektu.



Obrázek 3: Návrh builderu pro toky [Vlastní tvorba]

4.4 Uložení uzlů v toku

Uložení uzlů v toku může být provedeno dvěma hlavními způsoby. První z možností je použití homogenních polí, kde každý uzel by musel mít stejný typ. Druhá možnost je použití heterogenních polí za pomocí trait objektů (`Vec<dyn Node>` nebo `Vec<dyn Any>`). Pak ještě existuje možnost použití tuple listů (sekce 2.10), které umožňují vytváření heterogenních polí s konkrétními typy.

Použití homogenních polí nepřipadá v úvahu, protože by v toku mohl být použit jen jeden typ uzlu, což je pro účely této knihovny nepřípustné.

Použití trait objektu `dyn Node` by znamenalo použití dynamické výběru (sekce 2.4.2), který způsobí, že kompilátor nemůže kód pořádně optimalizovat. Zároveň aktuální návrh `Node` traitu není možně použít jako trait objekt, protože není dyn kompatibilní (sekce 2.8).

Použití trait objektu `dyn Any` přidá režii díky tomu, že se musí každý trait objekt převést na konkrétní typ (sekce 2.7). Typy, na které je potřeba trait objekty převést, je nutno znát předem, což znamená, že musí být v definici toku. Dále při tomto procesu také probíhá kontrola typu a trait objekt `dyn Any` musí být obalen např. v boxu (sekce 2.9.1), což přidává indirekci a režii. Mohly by se použít tzv. „unchecked downcast“ metody (sekce 2.7.3), ale ty jsou bohužel nedostupné na stabilním kanálu a neřeší problém režie boxu.

Metodou eliminace už zbývají jen tuple listy. Jejich hlavní nevýhodou je, že se s nimi pracuje hůře než s normálními listy a vyžadují definici nových traitů pro specifické použití, což není až takový problém, protože každý tok by měl plnit jinou funkci. Dále je také nutno uložit datový typ tuple listu do definice toku.

Kód 25: Návrh struktury objektu toku [Vlastní tvorba]

```
struct NazevToku<Vstup, Vystup, TupleListTyp> {
    uzly: TupleListTyp,
}
```

4.5 Definice běhu toku

Běh toku lze zhruba rozdělit do čtyř částí. První část je vytvoření asynchronních úloh. Druhá část je spuštění asynchronních úloh. Třetí část je čekání na dokončení asynchronních úloh. Poslední čtvrtá část je zpracování výstupů asynchronních úloh.

Toto rozdělení běhu toků do částí s největší pravděpodobností není možno dodržet pro všechny toky. Z tohoto důvodu při implementaci mohou být nějaké nebo všechny tyto části spojeny do jedné. Dále nad těmito částmi by už měla být jen samotná implementace traitu pro toky.

Kód 26: Ukázka návrhu implementace běhu toku pro paralelní tok [Vlastní tvorba]

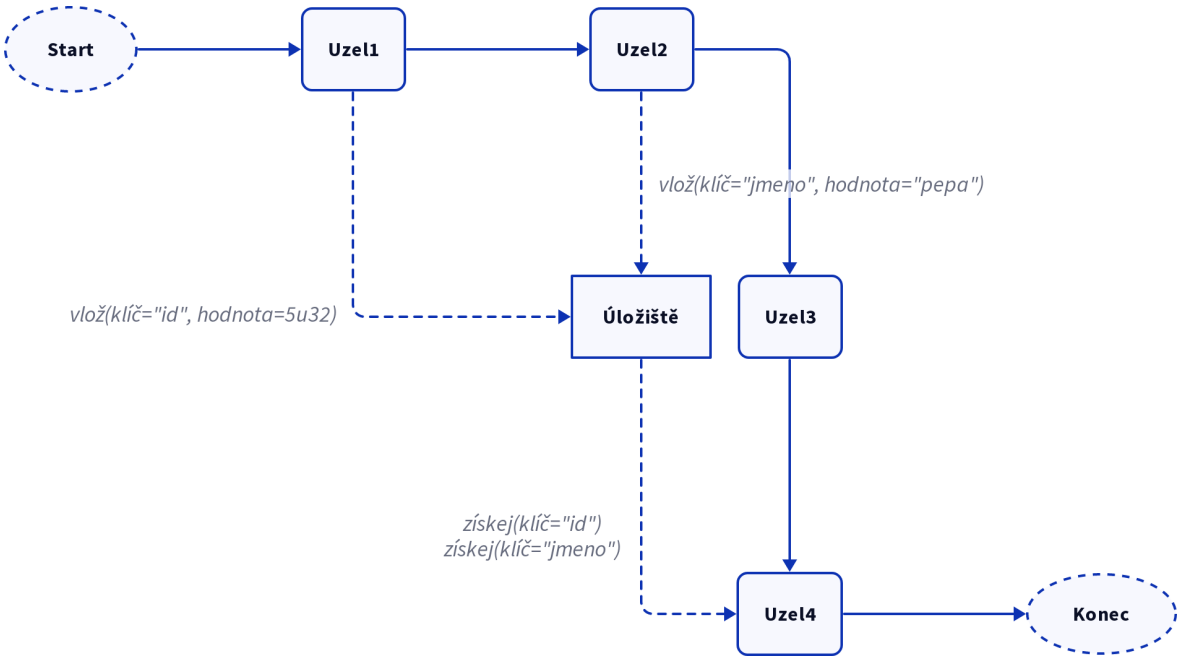
```rust
// pseudokód implementace běhu toku
impl Node for ParallelFlow {
    async fn run(&mut self, input: Input) -> Output {
        // vytvoření asynchronních úloh
        let futures: Vec<NodeFuture> = self.create_futures(input);
        // spuštění asynchronních úloh
        futures.iter().for_each(runtime::spawn);
        // čekání na dokončení asynchronních úloh
        let outputs: Vec<_> =
         ↪  futures.into_stream().map(NodeFuture::await).await;
        // zpracování výstupů asynchronních úloh
        let result = self.handle_outputs(outputs);
        return result;
    }
}
```

4.6 Kontextový systém

Hlavní funkcí kontextového systému je poskytnutí prostředků pro získání hodnot potřebných pro běh uzlů.

4.6.1 Úložiště typu klíč-hodnota

Prvním návrhem pro kontext bylo úložiště typu klíč-hodnota. Toto úložiště by sloužilo ke sdílení dat mezi uzly nebo k zachování dat ze vstupu pro následující uzly.



Obrázek 4: Ilustrace použití úložiště [Vlastní tvorba]

Tento přístup by byl velmi jednoduchý na implementaci, protože by stačilo jen implementovat úložiště typu klíč-hodnota, což může být provedeno pomocí hašovací tabulky, použitím nějaké existující knihovny, nebo i napojením na externí databázi jako je například redis.

Kód 27: Návrh traitu `Node` s úložištěm [Vlastní tvorba]

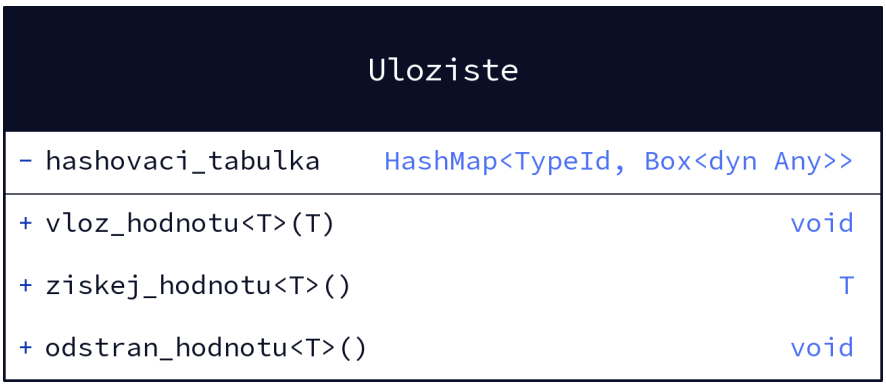
```
use node_flow::storage::Storage;

trait Node<Input, Output> {
    async fn run(&mut self, input: Input, storage: &mut Storage) ->
    ↪  Output;
}
```

Zde ale už nastává problém - Jak má být toto úložiště tedy implementováno a co musí umět? Pokud by se použila jakákoli implementace, která limituje, co může být do úložiště vloženo, tak uživatel knihovny na tuto limitaci určitě narazí a díky tomu, že definice traitu `Node` je neměnná, tak není schopen tuto limitaci jednoduše obejít. Jednou z možností, jak by tuto limitaci mohl obejít, je brát data, která chce sdílet mezi uzly, při konstrukci uzlu. Tato možnost je ale nepraktická, zbytečně komplikovaná a znamenalo by to, že úložiště je tomuto uživateli spíše na potíž než k užitku.

Jednoduchý příklad je implementace pomocí hašovací tabulky, která má klíč `String` a hodnotu `String`. Vše, co do tohoto úložiště bude vloženo, musí být převeditelné na řetězec a získatelné z řetězce. Poté uživatel narazí na problém, že do tohoto úložiště nejde snadno vložit například `Mutex` (sekce 2.9.4), protože ho nelze převést na řetězec.

Tento problém řeší tzv. type mapa, která může být implementována pomocí hašovací tabulky, kde klíčem je `TypeId` (sekce 2.7.1) a hodnotou je `Box<dyn Any>` (sekce 2.7.2). Type mapa řeší tento problém tak, že všechny objekty, které jsou do ní vloženy, převádí na trait objekt `Box<dyn Any>` a když uživatel chce tyto data získat zpět z type mapy, tak stačí jen znát datový typ těchto dat (sekce 2.7.3). Bohužel i toto řešení má své omezení a to, že pro daný datový typ může type mapa obsahovat pouze jednu instanci tohoto datového typu.



Obrázek 5: Návrh úložiště pro trait `Node` [Vlastní tvorba]

4.6.2 Kontext za pomocí generického typu

Implementace za pomocí type mapy řeší většinu, ne-li všechny, problémy s úložištěm, ale neřeší problém kontextu. Uzly by měly mít možnost vyžadovat, aby nějaká hodnota, potřebná k běhu uzlu, byla vždy dostupná. Pro tento účel nelze použít jen jednoduché úložiště. Pokud by se použilo jen úložiště, tak může nastat stav, kde daná hodnota v úložišti nebude a uzel tento problém musí řešit za běhu programu. Znamenalo by to, že uzel by musel vracet chybu, která musí být uživatelem řešena, za běhu nebo v horším případě by mohl také celý program spadnout.

Tento problém lze jednoduše ilustrovat na uzlu, který bude stahovat data z webu a potřebuje přihlašovací údaje. Přihlašovací údaje by mohl tento uzel brát v konstruktoru, ale to by znamenalo, že když je tento uzel použit v nějakém toku, tak pro každý přihlašovací údaj je potřeba vytvořit nová instance uzlu, která je poté použita k sestavení nového toku.

Řešením je přidání stupně volnosti na kontext tím způsobem, že místo konkrétního typu pro kontext se použije generický parametr, který může být libovolně omezen. Tímto způsobem se docílí toho, že uzly mohou definovat traity, které kontext musí implementovat. Tyto traity poté obsahují metody, které souží k poskytnutí libovolné hodnoty potřebné k běhu uzlu.

Kód 28: Návrh traitu `Node` s generickým kontextem [Vlastní tvorba]

```rust
trait Node<Input, Output, Context> {
    async fn run(
        &mut self,
        input: Input,
        storage: &mut Context
    ) -> Output;
}
```

Kód 29: Příklad použití traitu `Node` s generickým kontextem [Vlastní tvorba]

```rust
// trait pro omezení kontextu
trait HowMuch {
    fn how_much(&self) -> u8;
}

// definice kontextu, který bude použit
struct MyContext {
    by_how_much: u8,
}

// implementace omezovacího traitu pro kontext
impl HowMuch for MyContext {
    fn how_much(&self) -> u8 {
        return self.by_how_much;
    }
}

// definice uzlu
struct AddNode;

// implementace Node traitu pro uzel,
// kde kontext musí implementovat omezovací trait
impl<Context> Node<u8, u8, Context> for AddNode
where
    // omezení kontextu je definováno zde
    Context: HowMuch
{
    async fn run(
        &mut self,
        input: u8,
        context: &mut Context
    ) -> u8 {
        return input + context.how_much()
    }
}

async fn main() {
    // vytvoření instance uzlu a kontextu
    let mut node = AddNode;
    let mut context = MyContext {
        by_how_much: 5,
    };
    // definice vstupu
    let input = 12;

    // spuštění uzlu a čekání na výsledek
    let output = node.run(input, &mut context).await;
    // kontrola výsledku
```

```
    assert_eq!(output, input + context.by_how_much);
}
```

Kód 30: Chyba, která nastane při kompilaci, když nejsou splněny požadavaky na kontext [Vlastní tvorba]

```
error[E0277]: the trait bound `MyContext: HowMuch` is not satisfied
  --> src/main.rs:47:27
   |
47 |     let output = node.run(input, &mut context).await;
   |                       --- ^^^^^ unsatisfied trait bound
   |                       |
   |                       required by a bound introduced by this call
   |
help: the trait `HowMuch` is not implemented for `MyContext`
  --> src/main.rs:7:1
   |
 7 | struct MyContext {
   | ^^^^^^^^^^^^^^^^
help: this trait has no implementations, consider adding one
  --> src/main.rs:2:1
   |
 2 | trait HowMuch {
   | ^^^^^^^^^^^^^
note: required for `AddNode` to implement `Node<u8, u8, MyContext>`
  --> src/main.rs:23:15
   |
23 | impl<Context> Node<u8, u8, Context> for AddNode
   |               ^^^^^^^^^^^^^^^^^^^^     ^^^^^^^
...
26 |     Context: HowMuch
   |              ------- unsatisfied trait bound introduced here
```

5 Implementace uzlů

V této kapitole je popsána implementace jak základní jednotky pro tvorbu uzlů, tak i ostatních traitů, struktur a maker souvisejících s prací s uzly a jejich tvorbou.

5.1 Trait Node

Kód 31: Implementace traitu `Node` [Vlastní tvorba]

```rust
pub trait Node<Input, Output, Error, Context> {
    fn run(
        &mut self,
        input: Input,
        context: &mut Context,
    ) -> impl Future<Output = Result<Output, Error>> + Send;

    fn describe(&self) -> Description where Self: Sized { .. }
}
```

Tento trait slouží jako základní stavební blok pro definici uzlů a toků. Vychází z návrhu pro tento trait s generickým kontextem (kód 28) a rozšiřuje ho o generický parametr `Error` a metodu `describe`.

Dále se liší v definici metody `run` explicitní definicí výstupního typu, který musí implementovat trait `Future` (sekce 2.3.2) a tento typ také musí implementovat auto trait `Send`. Auto trait `Send` je vyžadován proto, aby bylo možno vzniklou asynchronní úlohu odeslat do jiného vlákna a docílit tak mnohem větší konkurence.

Bylo také nutno do tohoto traitu přidat metodu `describe`, která slouží k popisu uzlu. Vrací informace jak o vstupních a výstupních typech, tak i dodatečné informace jako je popis uzlu. Tato metoda se nachází v tomto traitu, protože bylo nutno definovat výchozí implementaci pro všechny uzly. Původně tato metoda měla být ve vlastním traitu, ale Rust zatím nedovoluje vytvořit výchozí implementaci pro všechny datové typy implementující trait `Node`, kterou by poté uživatel mohl přepsat.

Výchozí implementace metody `describe` vrací pouze název typu uzlu, vstupního, výstupního, chybového a kontextového typu. Z názvu výstupního typu odstraňuje `NodeOutput<>`, aby výsledný popis byl více přehledný. Pro získání takzvané `DescriptionBase` se používá pomocná metoda, která je popsána ve vlastní kapitole (sekce 7, 7.2).

5.2 Enum NodeOutput

Kód 32: Implementace enumu `NodeOutput` [Vlastní tvorba]

```rust
pub enum NodeOutput<T> {
    SoftFail,
    Ok(T),
}

impl<T> NodeOutput<T> {
    pub fn ok(self) -> Option<T> { .. }
    pub fn ok_or<E>(self, err: E) -> Result<T, E> { .. }
    pub fn ok_or_else<E>(self, err: impl Fn() -> E) -> Result<T, E> { .. }
}
```

Enum `NodeOutput` slouží k signalizaci takzvané „soft chyby“. Tato chyba vyjadřuje, že operace selhala, ale ne katastrofálně. Když nějaký uzel signalizuje tuto chybu, znamená to, že tok může pokračovat dále (pokud je jak), jako kdyby se nic nedělo. Uzel může signalizovat tuto chybu například při nesplnění vstupních požadavků, kde vstup neodpovídá přesně požadavkům tohoto uzlu, ale může existovat nějaký další uzel, který s těmito vstupními daty bude spokojen a dokáže je zpracovat. Díky tomuto enumu lze definovat takzvané „fallback vodopády“. Tyto vodopády umožňují pokračovat další větví toku, když předchozí větev vrátí `SoftFail`.

Pro tento enum jsou také implementovány pomocné metody `ok`, `ok_or` a `ok_or_else` pro jednoduchou konverzi do typů `Option` a `Result`. Tyto metody jsou inspirovány metodami nacházejícími se ve standardní knihovně Rustu.

5.3 Trait BoxedNode

Kód 33: Implementace traitu `BoxedNode` [Vlastní tvorba]

```rust
#[async_trait::async_trait]
pub trait BoxedNode<Input, Output, Error, Context> {
    async fn run_boxed(&mut self, input: Input, context: &mut Context) ->
     ↪  Result<Output, Error>
    where
        Input: 'async_trait,
        Output: 'async_trait,
        Error: 'async_trait;

    fn describe(&self) -> Description;
}
```

Trait `BoxedNode` je dyn kompatibilní obal (sekce 2.8) pro trait `Node`, který obsahuje veškeré metody, jako trait `Node`. Díky tomuto obalu je možno pracovat s uzly ve stejném stylu, jako kdyby trait `Node` byl dyn kompatibilní. Přidává tedy další stupeň volnosti pro uživatele této knihovny. Tento obal hlavně existuje proto, aby si uživatel mohl vybrat, zda chce pracovat se statickým výběrem nebo s dynamickým.

Pro tento obal je také vytvořená výchozí implementace pro jakýkoli typ implementující trait `Node`. Metoda `run` vypadá komplikovaně, ale jen obaluje vytvořenou asynchronní úlohu z `Node` traitu do pin-boxu (sekce 2.9.2, 2.9.1) a přemění typ této úlohy trait object `dyn Future + Send`. Metoda `describe` volá metodu se stejným jménem z `Node` traitu a vrací tuto nepozměněnou hodnotu, protože trait `BoxedNode` nemění chování ani vstupy či výstupy uzlu.

5.4 Makro impl_node_output

Toto makro slouží k jednoduchému přidání implementace traitu `Node`, která má úspěšný výstupní typ obalen v `NodeOutput` enumu. Díky tomuto makru uživatel nemusí psát zbytečně opakující se kód. Bere typ uzlu, vstupu, výstupu, chyby a možná dodatečná omezení pro generické typy a pomocí těchto typů a omezení definuje novou implementaci `Node` traitu.

Kód 34: Příklad použití makra `impl_node_output` [Vlastní tvorba]

```
use node_flow::{impl_node_output, node::{Node, NodeOutput}};

struct ExampleNode;

impl<Context: Send> Node<i32, i32, String, Context> for ExampleNode {
    async fn run(&mut self, input: i32, _context: &mut Context) ->
     ↪  Result<i32, String> {
        Ok(input + 1)
    }
}

// Přidá implementaci Node<i32, NodeOutput<i32>, String, _> pro
 ↪  ExampleNode.
impl_node_output!(ExampleNode, i32, i32, String);
```

5.5 Makro node

Kód 35: Implementace makra `node` [Vlastní tvorba]

```
#[macro_export]
macro_rules! node {
    ($input:ty, !$output:ty, $error:ty, $context:ty) ⟹ {
        impl $crate::node::Node<$input, $output, $error, $context> +
         ↪  Clone + Send + Sync
    };
    ($input:ty, $output:ty, $error:ty, $context:ty) ⟹ {
        $crate::node!($input, !$crate::node::NodeOutput<$output>, $error,
         ↪  $context)
    };
}
```

Makro sloužící pro jednoduchou definici vstupního či výstupního typu, který implementuje `Node` trait. Bere vstupní, výstupní, chybový a kontext typ pro uzel a vrací `impl Node<Vstup, NodeOutput<Výstup>, Error, Kontext> + Clone + Send + Sync`. Typ výstupu může mít před sebou vykřičník a tím říká, že výstupní typ nemá být obalen do `NodeOutput` enumu.

Kód 36: Příklad použití makra `impl_node_output` [Vlastní tvorba]

```
// Funkce vytvařející tok, který vrací nějaký typ implementující Node
 ↪  trait s vstupním typem u8, výstupním typem u64, chybovým typem String
 ↪  a generickým typem kontextu Context
fn build_flow<Context: Send>(..) -> node_flow::node!(u8, !u64, String,
 ↪  Context) {
    ..
}
```

6 Implementace kontextového systému

Implementovaný kontextový systém vychází z návrhu kontextového systému pomocí generického typu (sekce 4.6.2). Hlavní myšlenkou je dovolit uživateli omezovat kontext pro každý uzel zvlášť podle jeho potřeby. Díky této implementaci mohou uživatelé této knihovny definovat vlastní omezovací traity, které nejsou součástí této knihovny.

V této kapitole jsou probírány a popsány implementace základních traitů pro práci s kontextem, traity pro vytváření synchronních a asynchronních úloh a traity pro práci s lokálním a sdíleným úložištěm. Dále jsou zde také popsány implementace konkrétních úložišť, které implementují traity pro lokální nebo sdílené úložiště.

6.1 Základní traity pro práci s kontextem

Tato kapitola obsahuje základní traity, které by měly být implementovány pro jakýkoliv datový typ, který lze použít jako kontext v uzlu. Tyto traity slouží jako základ pro vytváření komplexnějších toků.

6.1.1 Trait Fork

Kód 37: Implementace traitu `Fork` [Vlastní tvorba]

```rust
pub trait Fork {
    #[must_use]
    fn fork(&self) -> Self;
}
```

Trait `Fork` slouží k vytvoření nové instance kontextu z již existující instance, která byla uzlu předána. Hlavní využití tohoto traitu je vytvoření na sobě nezávislých instancí kontextu, které poté mohou být použity ve větvích toků. Tento trait umožňuje například „fallback vodopády“, kde pokud by nějaká větev toku selhala „soft chybou“, tak tok může pokračovat další větví vytvořením nové instance kontextu z originálu. Tímto se zajistí, že větev co selhala nezpůsobí selhání dalších větví tím, že nějakým nevratným způsobem upravila kontext.

6.1.2 Trait Update

Kód 38: Implementace traitu `Update` [Vlastní tvorba]

```
pub trait Update {
    fn update_from(&mut self, other: Self);
}
```

Trait `Update` komplementuje trait `Fork` tím, že umožňuje aktualizaci nějaké instance kontextu z jiné instance kontextu stejného typu. Díky tomuto traitu je možno v tocích, které například vrací výsledek první úspěšné větve, aktualizovat originální kontext z kontextu vytvořeného pomocí `Fork` traitu.

6.1.3 Trait Join

Kód 39: Implementace traitu `Join` [Vlastní tvorba]

```
pub trait Join: Sized {
    fn join(&mut self, others: Box<[Self]>);
}
```

Trait `Join` komplementuje trait `Fork` tím, že umožňuje aktualizaci jedné instance kontextu z mnoha instancí kontextu stejného typu. Tento trait umožňuje tokům, které například vrací výsledek všech úspěšných či neúspěšných větví, sloučit kontexty vytvořené pomocí `Fork` traitu do originálního kontextu.

Zde by samozřejmě šlo použít trait `Update` vícekrát, ale tento přístup by způsobil, že sloučení kontextů by bylo závislé na pořadí. Pokud by tento přístup byl použit, tak by jistě nastal případ, kdy by několik větví vkládalo nějaká data se stejným klíčem do úložiště a při slučování by v tomto případě v úložišti s největší pravděpodobností zůstaly ta data, která byla sloučena jako poslední. Mnohem lepší přístup je rozlišit tyto případy a řešení jak bude slučování probíhat v těchto případech ponechat na uživatele.

6.2 Práce se synchronními a asynchronními úlohami

V této kapitole jsou definovány a popsány traity potřebné pro vytváření a práci se synchronními a asynchronními úlohami. Tyto traity slouží hlavně k abstrakci konkrétních implementací v asynchronních runtimech (sekce 2.3.3).

6.2.1 Trait Task

Kód 40: Implementace traitu `Task` [Vlastní tvorba]

```rust
pub trait Task<T>: Future<Output = T> {
    fn is_finished(&self) -> bool;
    fn cancel(self);
}
```

Trait `Task` je nástavba nad traitem `Future` (sekce 2.3.2), která přidává dvě velmi důležité metody a slouží k reprezentaci vzniklé asynchronní úlohy. Tento trait hlavně slouží jako abstrakce nad konkrétními implementacemi asynchronních úloh v různých asynchronní runtimech (sekce 2.3.3). Metoda `is_finished` vrací, zda úloha byla dokončena, bez ohledu na to, jestli úspěšně či neúspěšně. Metoda `cancel` konzumuje instanci úlohy a signalizuje úloze, že byla zrušena. V případě signalizace zrušení úlohy by úloha měla přerušit jakékoli právě běžící úkony a zastavit se.

6.2.2 Trait SpawnAsync

Kód 41: Implementace traitu `SpawnAsync` [Vlastní tvorba]

```rust
pub trait SpawnAsync {
    fn spawn<F>(fut: F) -> impl Task<F::Output>
    where
        F: Future + Send + 'static,
        F::Output: Send + 'static;
}
```

Trait `SpawnAsync` poskytuje rozhraní pro vytváření asynchronních úloh. Tento trait je abstrakcí nad různými asynchronními runtimy (sekce 2.3.3) pro vytváření a práci s asynchronními úlohami. Obsahuje pouze jednu metodu `spawn`, která bere asynchronní úlohu a vrací nějaký datový typ implementující trait `Task`. Tato asynchronní úloha musí být `Send + 'static`, aby bylo možno tuto úlohu poslat do jiných vláken pro zpracování.

6.2.3 Trait SpawnSync

Kód 42: Implementace traitu `SpawnSync` [Vlastní tvorba]

```
pub trait SpawnSync {
    fn spawn_blocking<F, O>(func: F) -> impl Task<O>
    where
        F: Fn() -> O + Send + 'static,
        O: Send + 'static;
}
```

Trait `SpawnSync` poskytuje rozhraní pro vytváření synchronních úloh. Tento trait je synchronní verze `SpawnAsync` traitu a poskytuje možnost vytváření potenciálně časově či výpočetně náročných úloh bez blokování asynchronních runtimu (sekce 2.3.3). Stejně jako trait `SpawnAsync` obsahuje pouze jednu metodu se jménem `spawn_blocking`, která bere synchronní úlohu a vrací nějaký datový typ implementující trait `Task`. Pro tuto synchronní úlohu také platí, že musí být `Send + 'static`, aby bylo možno tuto úlohu poslat do jiných vláken.

6.3 Lokální úložiště

Lokální úložiště je navrženo pro správu dat, která by měla být sdílena mezi uzly ve stejné větvi, ale ne napříč větvemi v tocích. Instance lokálního úložiště jsou tedy na sobě nezávislé a unikátní. Každá instance tohoto úložiště může existovat samostatně a práce s ní neovlivňuje ostatní instance.

6.3.1 Enum MergeResult

Kód 43: Implementace enumu `MergeResult` [Vlastní tvorba]

```
pub enum MergeResult<T> {
    KeepParent,
    ReplaceOrInsert(T),
    Remove,
}
```

Enum `MergeResult` reprezentuje výsledek sloučení více instancí určitého datového typu během slučování kontextů. Tento enum se používá v traitu `Merge` k určení, jak by mělo sloučení ovlivnit originální hodnotu. Rozhodnutí zda originální hodnota bude ponechána, nahrazena nebo vložena či odstraněna záleží na implementaci uživatele této knihovny.

6.3.2 Trait Merge

Kód 44: Implementace traitu `Merge` [Vlastní tvorba]

```
pub trait Merge: Sized {
    fn merge(parent: Option<&Self>, others: Box<[Self]>) ->
     ↪  MergeResult<Self>;
}
```

Trait `Merge` slouží k definici, jak mají být instance stejného typu sloučeny. Obsahuje pouze jednu funkci `merge`, která bere neměnnou referenci na originální instanci, pokud existuje, a pole ostatních instancí stejného typu. Implementace této funkce definuje, co se má stát s instancí, která se momentálně nachází v úložišti. Toto rozhodnutí je ponecháno na uživateli knihovny.

6.3.3 Trait LocalStorage

Kód 45: Implementace traitu `LocalStorage` [Vlastní tvorba]

```
pub trait LocalStorage {
    fn get<T>(&self) -> Option<&T>
    where
        T: 'static;

    fn get_mut<T>(&mut self) -> Option<&mut T>
    where
        T: 'static;

    fn insert<T>(&mut self, val: T) -> Option<T>
    where
        T: Merge + Clone + Send + 'static;

    fn remove<T>(&mut self) -> Option<T>
    where
        T: 'static;
}
```

Trait `LocalStorage` definuje typové lokální úložiště, které není sdíleno s žádnou jinou větví v toku a vychází z návrhu pro úložiště typu klíč-hodnota (sekce 4.6.1, obrázek 5). Umožňuje ukládání a získávání hodnot podle jejich datového typu. Tato implementace umožňuje mít pro každý datový typ `T` uloženou pouze jednu instanci v úložišti.

Obsahuje metody pro získání reference na instanci datového typu, jak měnitelnou (`get_mut`), tak neměnitelnou (`get`), odstranění instance (`remove`) z úložiště a vložení (`insert`). Instance které mají být vloženy do tohoto úložiště musí implementovat trait `Merge`.

6.3.4 Implementace lokálního úložiště

Kód 46: Implementace lokálního úložiště - struct `LocalStorageImpl` [Vlastní tvorba]

```
pub struct LocalStorageImpl {
    inner: HashMap<TypeId, Box<dyn StorageItem>>,
    changed: HashSet<TypeId>,
}
```

Implementace lokálního úložiště vychází z návrhu pro úložiště typu klíč-hodnota (obrázek 5). Tento návrh je rozšířen o atribut `changed`, který značí, jaké hodnoty byly změněny, vloženy či odstraněny a je využit při slučování kontextů. Dále je namísto trait objektu `dyn Any` využit privátní trait objekt `dyn StorageItem`, který není dostupný uživateli knihovny, ale pouze této implementaci lokálního úložiště.

Trait StorageItem

Kód 47: Implementace lokálního úložiště - trait `StorageItem` [Vlastní tvorba]

```
trait StorageItem: Any + Send {
    fn duplicate(&self) -> Box<dyn StorageItem>;
    fn merge(
        &self,
        parent: Option<&dyn StorageItem>,
        others: Box<[Box<dyn StorageItem>]>,
    ) -> MergeResult<Box<dyn StorageItem>>;
}
```

Trait `StorageItem` je nástavba nad traitem `Any` a slouží jako dyn kompatibilní (sekce 2.8) verze kombinace traitů `Merge + Clone`. Metoda `duplicate` slouží k naklonování hodnoty, stejně jako trait `Clone`, ale vrací `Box<dyn StorageItem>` místo `Self` typu. Metoda `merge` slouží ke sloučení vícero hodnot stejného typu, stejně jako trait `Merge`, ale bere a vrací nějakou verzi trait objektu `dyn StorageItem` místo `Self` typu.

U metody `merge` je důležité zajistit, že všechny trait objekty `dyn StorageItem` mají vnitřně stejný datový typ, aby bylo možno tento trait považovat za ekvivalentní ve funkčnosti jako trait `Merge`. Dále do originální funkce `merge` byla přidána reference na sama sebe, která je nutná aby trait `StorageItem` mohl být dyn kompatibilní (kód 15 - funkce `ziskej_typ`).

Trait `StorageItem` je implementován pro jakýkoli typ `T`, který implementuje `Merge + Any + Send + Clone`. Metoda `duplicate` pouze volá metodu `clone` z traitu `Clone` a výslednou hodnotu obalí do boxu a vrátí ji. Přetypování na trait objekt `dyn StorageItem` probíhá automaticky. Metoda `merge` nejprve převádí všechny vstupní hodnoty trait objektů `dyn StorageItem` na konkrétní datový typ `T` (sekce 2.7.3) a poté volá funkci `merge` z `Merge` traitu s těmito hodnotami. Výsledná hodnota je vrácena nepozměněná, akorát u `MergeResult::ReplaceOrInsert` je potřeba tuto hodnotu obalit do boxu, aby přetypování mohlo převést konkrétní datový typ `T` na trait objekt `dyn StorageItem`.

Implementace základních traitů

Trait `Fork` je implementován, tak že type mapa obsahující data je naklonována a pro atribut `changed` je vytvořen nový hashset. Tímto se docílí toho, že instance lokálního úložiště jsou na sobě nezávislé a při slučování kontextů proběhne sloučení pouze změněných hodnot.

K naklonování type mapy je nutno, aby hodnoty implementovaly trait `Clone` pro `Box<dyn StorageItem>` a z tohoto důvodu byla přidána implementace, která volá metodu `duplicate` a vrací její výsledek.

Trait `Update` má velmi jednoduchou implementaci, protože stačí jen originální type mapu nahradit type mapou z instance, která má aktualizovat originální, a rozšířit originální hashset změněných hodnot.

Trait `Join` je implementován, tak že nejprve získá všechny typy (klíče type mapy) změněných hodnot a pak pro každý změněný typ získá hodnoty tohoto typu ze všech úložišť včetně originálního. Poté pro každý typ zavolá metodu `merge` a podle vrácené hodnoty upraví hodnotu v originálním úložišti. Implementace traitu

`Join` má jednoduchou myšlenku, ale samotná implementace je velmi zdlouhavá a řeší se v ní plno krajních případů.

Implementace traitu LocalStorage

Kód 48: Implementace lokálního úložiště - implementace traitu `LocalStorage` - metoda `get_mut` [Vlastní tvorba]

```
fn get_mut<T>(&mut self) -> Option<&mut T> where T: 'static {
    self.inner.get_mut(&TypeId::of::<T>()).map(|val| {
        self.changed.insert(TypeId::of::<T>());
        let any_debug_ref: &mut dyn Any = &mut **val;
        any_debug_ref.downcast_mut::<T>().unwrap()
    })
}
```

Implementace traitu `LocalStorage` je oproti implementaci traitu `Merge` pro datové typy, které mohou byt uloženy v úložišti, a traitu `Join` pro toto úložiště velmi triviální. Všechny metody pracují s type mapou a downcast metodami (sekce 2.7.3), aby získaly hodnotu konkrétního typu v úložišti. Metody `get_mut` a `remove` navíc ještě, oproti metodě `get`, označují datový typ v úložišti za změněný, pokud se v úložišti nachází, a metoda `insert` označuje datový typ za změněný vždy.

6.4 Sdílené úložiště

Sdílené úložiště podobně jako lokální úložiště je navrženo pro správu dat, která by měla být sdílena mezi uzly, ale na rozdíl od lokálního úložiště data jsou sdílena jak mezi uzly ve stejné větvi, tak i napříč větvemi. Každá instance sdíleného úložiště je tedy závislá na ostatních instancích se stejným kořenem. Pro každou instanci také platí, že může existovat samostatně, ale práce s ní ovlivňuje i ostatní instance.

6.4.1 Trait SharedStorage

Kód 49: Implementace traitu `SharedStorage` [Vlastní tvorba]

```rust
pub trait SharedStorage {
    fn get<T>(&self) -> impl Future<Output = Option<impl Deref<Target = T>>> + Send
    where
        T: 'static;

    fn get_mut<T>(&mut self) -> impl Future<Output = Option<impl DerefMut<Target = T>>> + Send
    where
        T: 'static;

    fn insert<T>(&mut self, val: T) -> impl Future<Output = Option<T>> + Send
    where
        T: Send + Sync + 'static;

    fn insert_with_if_absent<T, E>(
        &self,
        fut: impl Future<Output = Result<T, E>> + Send,
    ) -> impl Future<Output = Result<(), E>> + Send
    where
        T: Send + Sync + 'static,
        E: Send;

    fn remove<T>(&mut self) -> impl Future<Output = Option<T>> + Send
    where
        T: 'static;
}

```

Trait `SharedStorage` definuje typové sdílené úložiště, které je sdíleno napříč všemi větvemi v toku a vychází z návrhu pro úložiště typu klíč-hodnota (sekce 4.6.1, obrázek 5). Tento trait je velmi podobný traitu `LocalStorage` ve funkcích, co nabízí. Stejně jako trait `LocalStorage` umožňuje ukládání a získávání hodnot podle jejich datového typu. Tato implementace také umožňuje mít pro každý datový typ `T` uloženou pouze jednu instanci v úložišti.

Stejně jako trait `LocalStorage`, tento trait obsahuje metody pro získání reference na instanci datového typu, jak měnitelnou (`get_mut`), tak neměnitelnou (`get`), odstranění instance (`remove`) z úložiště a vložení (`insert`), ale oproti traitu `LocalStorage` má ještě navíc metodu `insert_with_if_absent`, která vloží hodnotu do úložiště, jen pokud se v úložišti pro daný datový typ nenachází žádná hodnota. Dalším rozdílem je, že trait `SharedStorage` vrací asynchronní úlohy pro práci s úložištěm, což je nutno, aby nebyl blokován asynchronní

runtime (sekce 2.3.3) při přístupu k úložišti z vícero větví současně. Instance datového typu, které mají být vloženy do tohoto úložiště musí implementovat trait `Send + Sync`, aby je bylo možno sdílet mezi vlákny.

Od implementací tohoto traitu se očekává, že budou správně pracovat s paralelismem, například za pomoci struktur jako je `RwLock` (sekce 2.9.5) nebo `Mutex` (sekce 2.9.4), a budou vracet zámky implementující traity `Deref` nebo `DerefMut` pro dočasný přístup.

6.4.2 Implementace sdíleného úložiště

Kód 50: Implementace sdíleného úložiště - struct `SharedStorageImpl` [Vlastní tvorba]

```
type StorageItem = Arc<RwLock<Option<Box<dyn Any + Send + Sync>>>>;

pub struct SharedStorageImpl {
    inner: Arc<Mutex<HashMap<TypeId, StorageItem>>>,
}
```

Implementace sdíleného úložiště, stejně jako implementace lokálního úložiště, vychází z návrhu pro úložiště typu klíč-hodnota (obrázek 5), ale oproti návrhu k trait objektu `dyn Any` jsou přidány auto traity `Send + Sync` a tento trait objekt je ještě obalen v atomické referenci (`Arc` - sekce 2.9.3) na zámek (`Rwlock` - sekce 2.9.5) obsahující možná data (`Arc<RwLock<Option<Box<dyn Any + Send + Sync>>>>`). Tento obal umožňuje zamykání pro každý datový typ zvlášť, bez nutnosti držení zámku pro celou type mapu. Dále je tato celá upravená type mapa obalena do atomické reference (`Arc` - sekce 2.9.3) a zámku (`Mutex` - sekce 2.9.4), pro souběžný přístup z více vláken.

Implementace dočasných zámků

Kód 51: Implementace sdíleného úložiště - implementace dočasných zámku [Vlastní tvorba]

```
pub struct ReadGuard<T: 'static> {
    pub guard: async_lock::RwLockReadGuardArc<Option<Box<dyn Any + Send +
    ↪  Sync>>>,
    pub _item_type: std::marker::PhantomData<T>,
}

impl<T: 'static> Deref for ReadGuard<T> {
    type Target = T;

```

```
    fn deref(&self) -> &Self::Target {
        let any_ref: &dyn Any = &**self.guard.as_ref().unwrap();
        any_ref.downcast_ref::<T>().unwrap()
    }
}

pub struct WriteGuard<T: 'static> { .. }
impl<T: 'static> Deref for WriteGuard<T> { .. }
impl<T: 'static> DerefMut for WriteGuard<T> { .. }
```

Struktury dočasných zámků obsahují dočasný zámek na data obsažená v type mapě, což jsou trait objekty, a jediné co dělají je, že implementují `Deref` trait, a `DerefMut` trait u zámku pro čtení. V těchto implementacích se získá reference na data v type mapě a pomocí downcast metod (sekce 2.7.3) se tato reference převede referenci konkrétního datové typu, která je poté vrácena.

Implementace základních traitů

Implementace základních traitů `Update` a `Join` je velmi jednoduchá, protože data jsou sdílena mezi instancemi, a tak není nutno nijak řešit aktualizaci a slučování kontextů. Trait `Fork` je také velmi jednoduchý, protože jen klonuje celou strukturu a vrací ji. Při tomto klonování dochází jen k naklonování atomické reference (`Arc` - sekce 2.9.3), což je velmi rychlé a levné.

Implementace traitu SharedStorage

Kód 52: Implementace sdíleného úložiště - implementace traitu `SharedStorage` - metoda `get` [Vlastní tvorba]

```
fn get<T>(&self) -> impl Future<Output = Option<impl Deref<Target = T>>>
    + Send
where
    T: 'static,
{
    let rw_lock = {
        let guard = self.inner.lock().unwrap();
        guard.get(&TypeId::of::<T>()).cloned()
    };

    async move {
        let rw_lock = rw_lock?;
        let rw_lock_guard = rw_lock.read_arc().await;
        if rw_lock_guard.is_none() {
            return None;
        }
        let read_guard = guards::ReadGuard {
            guard: rw_lock_guard,
            _item_type: std::marker::PhantomData,
```

```
            };

            Some(read_guard)
        }
    }
```

Trait `SharedStorage` je implementován, tak že v každé metodě se nejprve zamkne type mapa a získá se hodnota uložená v type mapě pro daný datový typ, tato hodnota se naklonuje a poté je type mapa odemknuta, aby ostatní vlákna k ní měla přístup co nejdříve. Klonování zde je velmi levné, protože se klonuje jen atomická reference (`Arc` - sekce 2.9.3). Z metod je poté vrácena asynchronní úloha, která obsahuje tuto naklonovanou hodnotu, a pokud v type mapě tato hodnota neexistuje, tak asynchronní úloha vrací `None`, ale pokud tato hodnota existuje, tak se získá zámek pro tuto hodnotu a ten je poté obalen do dočasného zámku, který implementuje `Deref` nebo `DerefMut` a je vrácen asynchronní úlohou. V kódu výše je uvedena pouze implementace metody `get`, protože implementace ostatních metod jsou velmi podobné a uvedení celé implementace traitu `SharedStorage` by bylo příliš dlouhé.

7 Implementace popisu uzlů a toků

Pro účely vizualizace a introspekce byly vytvořeny struktury k popisu uzlů a toků. Tyto struktury plně popisují vstupy a výstupy uzlů, u toků také plně popisují, jak jsou uzly propojeny, včetně typu kontextu, popisu poskytnutého uživatelem a externích zdrojů použitých při běhu uzlu.

7.1 Enum Description

Kód 53: Implementace enumu `Description` [Vlastní tvorba]

```rust
pub enum Description {
    Node {
        base: DescriptionBase,
    },
    Flow {
        base: DescriptionBase,
        nodes: Vec<Self>,
        edges: Vec<Edge>,
    },
}
```

Enum `Description` je nejvyšší vrstva, která může reprezentovat popis uzlu či toku. Pro popis uzlu i toku je použit struct `DescriptionBase` a u toků je ještě potřeba znát popisy vnitřních uzlů a jejich propojení. Díky těmto propojením lze poté sestrojit vizualizační graf.

Pro tento enum je také implementováno několik pomocných funkcí a metod, jejichž implementace je zdlouhavá, ale triviální. Funkce `new_node` bere referenci na uzel a vrací vyplněný `Description`. Nápodobně funkce `new_flow` bere referenci na tok, popisy vnitřních uzlů, jejich propojení a vrací vyplněný `Description`. Vnitřně funkce `new_node` a `new_flow` volají funkce z `DescriptionBase` a výstup z těchto funkcí je použít pro vytvoření enumu `Description`. Metody `get_base_ref` a `get_base_mut` jsou pomocné metody, které zjednodušují získání `DescriptionBase`. Poté také existují metody `with_description`, `with_externals` a `modify_name`, což jsou pomocné metody, které jsou zde pro jednodušší úpravu hodnot vnořených atributů.

7.2 Struct DescriptionBase

Kód 54: Implementace structu `DescriptionBase` [Vlastní tvorba]

```rust
pub struct DescriptionBase {
    pub r#type: Type,
    pub input: Type,
    pub output: Type,
    pub error: Type,
    pub context: Type,
    pub description: Option<String>,
    pub externals: Option<Vec<ExternalResource>>,
}

impl DescriptionBase {
    pub fn from<NodeType, Input, Output, Error, Context>() -> Self { .. }

    pub fn from_node<NodeType, Input, Output, Error, Context>(_node:
    ↪  &NodeType) -> Self where NodeType: Node<Input,NodeOutput<Output>,
    ↪  Error, Context> {
        Self::from::<NodeType, Input, Output, Error, Context>()
    }

    pub fn with_description(mut self, description: impl Into<String>) ->
    ↪  Self { .. }
    pub fn with_externals(mut self, externals: Vec<ExternalResource>) ->
    ↪  Self { .. }
}
```

Struct `DescriptionBase` je struktura, která slouží jako základ k popisu typu uzlu nebo toku, vstupu, výstupu, chyby, kontextu, textového popisu a externích zdrojů.

Pro tuto strukturu jsou také implementovány pomocné metody a funkce. Funkce `from` vytváří struct `DescriptionBase` z poskytnutých datových typů. Funkce `from_node` bere referenci na uzel a pomocí funkce `from` vytváří struct `DescriptionBase`. Poté jsou také implementovány pomocné metody `with_description` a `with_externals`, které nastavují atributy této struktury.

7.3 Struct Type

Kód 55: Implementace structu `Type` [Vlastní tvorba]

```rust
pub struct Type {
    pub name: String,
}

impl Type {
    pub fn of<T>() -> Self { Self { name: type_name::<T>().to_owned() } }
    pub fn of_val<T>(_: &T) -> Self { Self::of::<T>() }
```

```
8
9       pub fn get_name_simple(&self) -> String {
10          tynm::TypeName::from(self.name.as_str()).as_str()
11      }
12  }
```

Struct `Type` slouží k popisu datového typu a má jediný atribut, kterým je název daného datového typu. Obsahuje funkce `of` a `of_val`, které využívají vestavěné funkce `std::any::type_name` k získání názvu datového typu. Tato struktura, také obsahuje metodu `get_name_simple`, která vrací zjednodušený název datového typu, například pro datový typ `std::option::Option<std::string::String>` vrací `Option<String>`. Díky této metodě je možno redukovat informační zahlcení a tím docílit vytváření přehlednějších vizualizací.

7.4 Stuct Edge a enum EdgeEnding

Kód 56: Implementace structu `Edge` a enumu `EdgeEnding` [Vlastní tvorba]

```
1   pub struct Edge {
2       pub start: EdgeEnding,
3       pub end: EdgeEnding,
4   }
5
6   pub enum EdgeEnding {
7       ToFlow,
8       ToNode {
9           node_index: usize,
10      },
11  }
```

Struct `Edge` a enum `EdgeEnding` slouží k reprezentaci hran v orientovaném grafu, pro vizualizaci struktury toku a propojení uzlů.

Enum `EdgeEnding` vyjadřuje, k čemu má být jeden z konců hrany připojen. Hodnota `ToFlow` znamená, že konec hrany je připojen k toku a hodnota `ToNode` připojí konec hrany k uzlu, který se nachází na indexu `node_index`. Tento index také slouží pro vybrání popisu uzlu v enumu `Description`.

Pro struct `Edge` jsou také implementovány pomocné funkce `passthrough`, `flow_to_node`, `node_to_flow` a `node_to_node`. Tyto funkce usnadňují vytváření struktury `Edge`, tím že uživatel poskytne na vstupu 0 až 2 parametry, což jsou indexy uzlů, a získá vytvořenou strukturu.

7.5 Struct ExternalResource

Kód 57: Implementace structu `ExternalResource` [Vlastní tvorba]

```rust
pub struct ExternalResource {
    pub r#type: Type,
    pub description: Option<String>,
    pub output: Type,
}

impl ExternalResource {
    pub fn new<ResourceType, Output>() -> Self { .. }
    pub fn with_description(mut self, description: String) -> Self { ..}
}
```

Struct `ExternalResource` slouží k popisu nějakého externího zdroje, který uzel potřebuje ke svému běhu, poskytovaného nějakou službou. Obsahuje atributy pro jméno datového typu služby, textový popis a jméno datového typu externího zdroje. Pro tuto strukturu je také implementovaná funkce `new`, která z přijatých generických typů vytváří struct `ExternalResource`. Dále je také implementována pomocná metoda `with_description`, která nastavuje atribut `description`.

7.6 Privátní funkce remove_generics_from_name

Kód 58: Implementace funkce `remove_generics_from_name` [Vlastní tvorba]

```rust
pub(crate) fn remove_generics_from_name(orig_name: &mut String) {
    let generic_start_idx =
     ↪  orig_name.find('<').unwrap_or(orig_name.len());
    orig_name.truncate(generic_start_idx);
}
```

Funkce `remove_generics_from_name` je pomocná privátní funkce, která odstraňuje vše za a včetně generických parametrů v řetězci. Tohoto je docíleno tak, že se najde první lokace znaku otevírací špičaté závorky (<) v řetězci a originální řetězec je zkrácen na tento nalezený index. Tato funkce je použita při vytváření popisů pro toky. Díky této pomocné funkci a pomocné funkci `Description::modify_name` je úprava jména datového typu toku výrazně zjednodušena.

7.7 Konfigurovatelný D2 formátovač popisů uzlů a toků

Pro vytváření vizualizací uzlů a toků je implementován konfigurovatelný formátovač popisů. Tento formátovač převádí popis uzlu či toku do D2 jazyka[1] pro vytváření grafů. Ukázka vygenerované vizualizace (obrázek 11) se nachází v přílohách.

Kód 59: Implementace structu `D2Describer` [Vlastní tvorba]

```rust
pub struct D2Describer {
    pub simple_type_name: bool,
    pub show_context_in_node: bool,
    pub show_description: bool,
    pub show_externals: bool,
}

impl D2Describer {
    pub fn new() -> Self { .. }
    pub fn modify(&mut self, func: impl FnOnce(&mut Self)) -> &mut Self { .. }
    pub fn format(&self, desc: &Description) -> String { .. }
}
```

Struct `D2Describer` definuje vše potřebné pro převod popisů uzlů a toků do D2 jazyka a obsahuje přepínače pro konfiguraci formátovače. Atribut `simple_type_name` rozhoduje, zda mají mít datové typy zjednodušený název pomocí metody `Type::get_name_simple` a atributy `show_context_in_node`, `show_description` a `show_externals` rozhodují o tom, jestli výsledný graf má obsahovat datový typ kontextu, textový popis a externí zdroje.

Pro tento struct je implementována funkce `new`, pro vytvoření tohoto structu s výchozím nastavením. Dále je implementována metoda `modify` pro jednoduchou úpravu nastavení a metoda `format`, která převádí popis uzlu či toku do D2 jazyka a vrací výsledný řetězec.

[1] https://d2lang.com/

8 Implementace toků

Tato kapitola obsahuje nejen implementace všech toků poskytovaných touto knihovnou, ale i implementace pomocných maker typových aliasů a struktur, díky kterým je implementace umožněna či zjednodušena.

Implementace toků vychází z návrhu pro konstrukci toků, uložení uzlů v toku a definice běhu toků (sekce 4.3, 4.4, 4.5). Každý tok implementuje trait `Node` (sekce 5.1), aby mohl být použit jako uzel v dalším toku, a bere uzly implementující tento trait. Díky tomuto návrhu je možno teoreticky vytvářet vnořené toky s nekonečnou hloubkou. Avšak v realitě by uživatel jistě narazil na nějaký limit, jako je například stack limit, který lze jednoduše vyřešit, nebo nedostatek fyzické paměti.

8.1 Privátní pomocné typové aliasy

Kód 60: Implementace typových aliasů `NodeIOE`, `ChainLink` a `NodeResult` [Vlastní tvorba]

```
type NodeIOE<Input, Output, Error> = (Input, NodeOutput<Output>, Error);
type ChainLink<Head, Tail> = (Head, Tail);
type NodeResult<Output, Error> = Result<NodeOutput<Output>, Error>;
```

Typový alias `NodeIOE` slouží k definici typu reprezentující vstupní, výstupní a chybový datový typ uzlu obsaženého v toku. Je také důležité zmínit, že uvedený výstupní typ je tímto aliasem obalen do enumu `NodeOutput` (sekce 5.2).

Typový alias `ChainLink` reprezentuje jednoduchou vazbu v řetězu a jediné, co dělá, je, že určuje jak má být tato vazba reprezentována. Tento typový alias je použit při vytváření a přístupu k datovým typům elementů v tuple listech (sekce 2.10).

Typový alias `NodeResult` definuje reprezentaci výstupního a chybového typu uzlu. Díky tomuto type aliasu je definice výstupu uzlu v tocích výrazně zjednodušena a možnost nesprávné definice snížena.

8.2 Privátní pomocná struktura enum SoftFailPoll

Kód 61: Implementace struktury enum `SoftFailPoll` [Vlastní tvorba]

```
pub enum SoftFailPoll<T> {
    Pending,
    Ready(T),
    SoftFail,
}
```

Enum `SoftFailPoll` slouží k zjednodušení reprezentace výstupu z asynchronní úlohy. Přidává navíc oproti enumu `Poll` (sekce 2.3.1) možnost vrátit `SoftFail`, který má stejný význam jako v enumu `NodeOutput` (sekce 5.2).

8.3 Privátní pomocný trait ChainDebug

Kód 62: Implementace traitu `ChainDebug` [Vlastní tvorba]

```
pub trait ChainDebug {
    fn fmt_as_list(&self, f: &mut std::fmt::DebugList<'_, '_>);
    fn as_list(&self) -> ChainDebugAsList<&Self> {
        ChainDebugAsList(self)
    }
}

pub struct ChainDebugAsList<T>(T);
impl<T> Debug for ChainDebugAsList<&T> where T: ChainDebug { .. }
```

Trait `ChainDebug` slouží k vytištění všech uzlů v toku. Tohoto cíle je samozřejmě možno dosáhnout pomocí výchozí implementace traitu `Debug` pro tuply, ale vyprodukovaný výsledek není intuitivní a záleží na vnitřní reprezentaci uzlů v toku. Díky tomuto traitu jsou tuple listy tištěny jako heterogenní pole, i když jsou vnitřně reprezentovány jinak.

Například pro heterogenní pole obsahující hodnoty `A` až `C` by výchozí implementace mohla mít výstup `((((), A), B), C)`, ale při použití tratu `ChainDebug` by výstup byl `[A, B, C]`.

Tento trait má výchozí implementaci pro tuple listy, které mají strukturu takovou, že první element je na první pozici a poslední vrstvě `((((), A), B), C)` (kód 21).

8.4 Privátní pomocný trait ChainDescribe

Kód 63: Implementace traitu `ChainDescribe` [Vlastní tvorba]

```
pub trait ChainDescribe<Context, T> {
    const COUNT: usize;

    fn describe(&self, description_acc: &mut Vec<Description>);
}
```

Trait `ChainDescribe` slouží k získání popisů uzlů obsažených v toku. Obsahuje metodu `describe`, která bere referenci na tuple list a měnitelnou referenci na vektor popisů uzlů, do kterého jsou všechny popisy uzlů vloženy. Dále také obsahuje konstantu `COUNT`, která vyjadřuje počet elementů v tuple listu. Pomocí této konstanty lze vektor pre-alokovat a vyhnout se tak zbytečným re-alokacím.

Tento trait stejně jako trait `ChainDebug` má výchozí implementaci pro tuple listy, které mají strukturu takovou, že první element je na první pozici a poslední vrstvě `((((), A), B), C)` (kód 21).

8.5 Privátní pomocná makra

V této kapitole jsou definována a popsána privátní pomocná makra pro tisknutí uzlů v toku, definici toku a definici builderu pro tok.

Tyto makra výrazně usnadňují vytváření toků se stejnou strukturou nebo se stejnými buildery. Umožňují měnit vnitřní definice toků či builderů pro všechny toky využívající tyto makra zároveň a na jednom místě.

8.5.1 Makra impl_debug_for_builder a impl_debug_for_flow

Kód 64: Implementace makra `impl_debug_for_builder` [Vlastní tvorba]

```
macro_rules! impl_debug_for_builder {
    ($flow_name:expr, $builder:tt $(,$param:ident: $bound0:ident
    ↪  $(+$bound:ident)*)*) ⇒ { .. };
}
macro_rules! impl_debug_for_flow {
    ($flow_name:expr, $flow_type:tt $(,$param:ident: $bound0:ident
    ↪  $(+$bound:ident)*)*) ⇒ { .. };
}
```

Makra `impl_debug_for_builder` a `impl_debug_for_flow` slouží pro jednoduchou implementaci `Debug` traitů pro builder a tok. Berou název toku, datový typ, pro který má být `Debug` implementován, a dodatečné nepovinné omezení typů pomocí traitů.

Řetězec pro tisk je vytvořen pomocí metody `debug_struct`, která je definována standardní knihovnou, jejíž parametr je název datového typu struktury. Do této struktury jsou přidány atributy pomocí metody `field`. Hodnota pro atribut „nodes“, který obsahuje uzly toku či builderu, je získána pomocí traitu `ChainDebug` (sekce 8.3).

8.5.2 Makro define_flow

Kód 65: Implementace makra `define_flow` [Vlastní tvorba]

```
1 macro_rules! define_flow {
2     ($flow_name:ident, $builder:ident, $chain_run:ident, |$self:ident|
      ↪  $describe_code:block $(,$param:ident: $bound0:ident
      ↪  $(+$bound:ident)*)* $(,)? $(#[doc = $doc:expr])*) ⇒ { .. };
3 }
```

Makro `define_flow` slouží k usnadnění definice toků, které mají stejnou strukturu. Toto makro bere název datového typu toku, který se má definovat, datový typ builderu pro tento tok, datový typ takzvaného „chainrunu“, který definuje běh toku, definici těla funkce pro metodu `describe` z traitu `Node` (sekce 5.1), dodatečné nepovinné omezení typů pomocí traitů při definici `builder` metody a nepovinnou dokumentaci pro definovaný tok.

Toky definované tímto makrem mají implementovány trait `Debug`, s využitím pomocného makra, a `Clone`. Dále je také vytvořena funkce `builder`, která vrací instanci builderu vytvořenou funkcí `new` implementovanou v builderu. Implementace této funkce může být omezena pomocí dodatečných traitů, tyto traity většinou pochází ze samotné implementace builderu. Pro funkci `builder` je také automaticky vygenerována dokumentace s ukázkou použití.

Toto makro také implementuje trait `Node` (sekce 5.1) pro každý tok, který definuje. Implementace metody `run` jen volá takzvaný „chainrun“, který musí být implementován pro tuple list uzlů v toku, s parametry, které jsou do funkce poslány, a vrací výstupní asynchronní úlohu. Implementace metody `describe`

je velice triviální, protože jediné, co dělá je, že do těla této funkce vloží kód, který byl poskytnut tomuto makru pro přesně tento účel.

8.5.3 Makro define_builder

Kód 66: Implementace makra `define_builder` [Vlastní tvorba]

```
macro_rules! define_builder {
  ($flow_type:ident $(,>$global_param:ident: $global_bound0:ident
  ↪  $(+$global_bound:ident)*)* $(,#$fn_param:ident: $fn_bound0:ident
  ↪  $(+$fn_bound:ident)*)*) ⟹ { .. };
}
```

Makro `define_builder` slouží k usnadnění definice builderu pro toky, které mají stejné požadavky na uzly. Toto makro bere název datového typu toku, pro který se má builder definovat, nepovinné omezení typů pomocí traitů pro definici builder typu a pro definici požadavků na uzly.

Požadavky, které jsou kladeny na uzly vkládané do builderu vytvořeného tímto makrem, zní takto:

1. Datový typ vstupu toku musí být převoditelný na datový typ vstupu vkládaného uzlu.
2. Datový typ výstupu vkládaného uzlu musí být převoditelný na výstupní datový typ toku.
3. Datový typ chyby vkládaného uzlu musí být převoditelný na chybový datový typ toku.

Pro buildery definovány tímto makrem je implementován trait `Debug` s využitím pomocného makra. Dále je také implementován pro builder trait `Default`, který jen volá funkci `new` a vrací její hodnotu. Funkce `new` je také definována pro builder a jediné, co dělá, je, že vrací novou instanci builderu, kde hodnota atributu `nodes` je unit typ.

Definovány jsou také metody pro přidání uzlu a sestavení konečného toku. Metody `add_node` a `build` jsou definovány pro builder splňující základní omezení pro builder, definovány pomocí vstupů s předponou `global_` a značením `>`. Metoda `add_node` navíc vyžaduje po vkládaném uzlu, aby splňoval požadavky na něj

kladeny builderem spolu s dodatečnými požadavky, definovanými pomocí vstupů s předponou `fn_` a značením #.

Kód 67: Implementace makra `define_builder` - implementace metody `add_node` [Vlastní tvorba]

```
impl<..> Builder<
  Input, Output, Error, Context, NodeTypes,
  ChainLink<OtherNodeIOETypes, LastNodeIOETypes>
> where
  $($global_param: $global_bound0 $(+$global_bound)*,)*
{
  pub fn add_node<NodeType, NodeInput, NodeOutput, NodeError>(
    self,
    node: NodeType,
  ) -> Builder<
    Input, Output, Error, Context,
    ChainLink<NodeTypes, NodeType>,
    ChainLink<
      ChainLink<OtherNodeIOETypes, LastNodeIOETypes>,
      NodeIOE<NodeInput, NodeOutput, NodeError>,
    >,
  >
  where
    Input: Into<NodeInput>, NodeOutput: Into<Output>, NodeError:
     ↪  Into<Error>,
    NodeType: Node<NodeInput, NodeOutput<NodeOutput>, NodeError,
     ↪  Context>,
    $($fn_param: $fn_bound0 $(+$fn_bound)*,)*
  {
    Builder { .., nodes: (self.nodes, node) }
  }
}
```

Kód výše ukazuje, jak probíhá práce s typový systémem, kde builder s uzly `NodeTypes` a jejich vstupními, výstupními a chybovými datovými typy je po volání metody `add_node` převeden na builder s přidaným uzlem `NodeType` a jeho vstupním, výstupním a chybovým datovým typem. V kódu není uvedena verze metody `add_node`, která se stará o přidání prvního uzlu do toku a řeší požadavky na první uzel, protože je skoro totožná uvedené verzi a dělá prakticky to samé.

8.5.4 Makro define_flow_and_ioe_conv_builder

Makro `define_flow_and_ioe_conv_builder` je pomocné makro, které kombinuje makra `define_flow` a `define_builder` za účelem vytvoření toku se stejnou strukturou a builderem, ale s jinou funkcí. Na vstupu bere všechny parametry obou maker a vnitřně pomocí těchto parametrů volá tato makra, která

vygenerují definici toku a builderu. Díky tomuto makru je velice jednoduché definovat nový tok, který se liší pouze ve funkcionalitě či dodatečných požadavcích.

8.6 Tok SequentialFlow

Tok `SequentialFlow` spouští uzly postupně ve stejném pořadí, jako při vkládání uzlů do builderu. Funguje jako pipeline, kde výstup předchozího uzlu je použit jako vstup do dalšího uzlu.

Tok vrací zpracovanou vstupní hodnotu, pouze když všechny uzly úspěšně skončí. V tomto případě je výstup z posledního uzlu vrácen jako výstup toku. V ostatních případech výstup tohoto toku záleží na výstupu uzlů a zní takto:

1. Pokud uzel vrátí `NodeOutput::Ok`, tak je výstupní hodnota poslána do dalšího uzlu.
2. Pokud uzel vrátí `NodeOutput::SoftFail`, tak tok končí a vrací soft-fail.
3. Pokud uzel vrátí chybu, tak tok končí a vrací chybu.

8.6.1 Definice toku

Kód 68: Implementace toku `SequentialFlow` - definice toku [Vlastní tvorba]

```
define_flow!(
    SequentialFlow,
    ChainRunSequential,
    |self| { .. },
    Input: Send, Error: Send, Context: Send,
    ..
);
```

Pro definici toku je použito makro `define_flow` a běh toku je obstaráván `ChainRunSequential` traitem. Tělo `describe` metody vytváří acyklický orientovaný graf, kde každý uzel má jednu hranu mířící do uzlu a jednu hranu vycházející z uzlu. Vycházející hrana je vždy, až na poslední uzel, spojena s následujícím uzlem v toku.



Obrázek 6: Vizualizace toku `SequentialFlow` [Vlastní tvorba]

8.6.2 Definice builderu

Kód 69: Implementace toku `SequentialFlow` - definice builderu [Vlastní tvorba]

```
pub struct Builder<Input, Output, Error, Context, NodeTypes = (),
 ↪  NodeIOETypes = ()>
where
    Input: Send, Error: Send, Context: Send,
{
    ..,
    nodes: NodeTypes
}

impl_debug_for_builder!("SequentialFlow", Builder, Input: Send, Error:
 ↪  Send, Context: Send);
```

Definice builderu má stejnou základní strukturu, jako builder vytvořený makrem `define_builder`, ale liší se v požadavcích na uzly vkládané do builderu a v konečném sestavení toku.

Implementace metody `add_node` je rozdělena do dvou případů. Prvním případem je, že se vkládá první uzel do toku, a je nutno zajistit, že vstup do toku lze převést na vstup do prvního uzlu a chybu z uzlu lze převést na chybu toku. Tato verze metody `add_node` je implementována jen pro specifický typ builderu, který má generické parametry `NodeTypes` a `NodeIOETypes` nastaveny na unit typ, což je výchozí typ. Definice výchozích typů je vidět v definici builder structu (kód 69). Tento typ buidleru lze vytvořit jen pomocí funkcí `new` a `default`, a nelze ho získat jiným způsobem. Tímto se zajistí, že tato verze metody `add_node` je volána pouze jednou pro první uzel na novém builderu.

Implementace metody `add_node` pro ostatní uzly, zajišťuje, že výstup předchozího uzlu lze převést na vstup vkládaného uzlu a chybu z vkládaného uzlu lze převést na chybu toku.

Implementace metody `build` zajišťuje, že výstup z posledního uzlu lze převést na výstup z toku. V této metodě není nutno kontrolovat, zda chybu posledního uzlu lze převést na chybu toku, protože tato podmínka je již řešena metodami `add_node`.

Implementace metod `add_node` pro ostatní uzly a `build` se nachází v přílohách (kód 87). Tyto metody lze volat pouze po vložení prvního uzlu do builderu. Tímto se zajistí, že konečný tok není prázdný a umožňuje definovat jiné požadavky na uzly podle jejich pozice v toku.

8.6.3 Privátní trait ChainRunSequential

Kód 70: Implementace toku `SequentialFlow` - definice traitu `ChainRunSequential` [Vlastní tvorba]

```
pub trait ChainRunSequential<Input, Output, Context, T> {
    fn run(&self, input: Input, context: &mut Context) -> impl
     ↪  Future<Output = Output> + Send;
}
```

Trait `ChainRunSequential` obsahuje veškerou logiku pro běh toku a správu asynchronních úloh. Generický typ `T` v definici traitu je použit pro uložení vstupních, výstupních a chybových datových typů uzlů.

Implementace traitu `ChainRunSequential` pro tuple list definuje sekvenční vykonávání jednotlivých uzlů toku. Běh je implementován rekurzivní dekompozicí tuple listu na hlavu a ocas, kde metoda `run` z traitu `Node` je vyvolána nejprve na prvním uzlu v toku. Výstup uzlu je poté použit jako vstup do dalšího uzlu a v případě chyby je běh toku ukončen a chyba je vrácena jako výsledek toku.

Tato implementace zajišťuje, že uzly jsou prováděny ve stejném pořadí, v jakém byly vloženy do builderu, a že výsledek posledního uzlu je vrácen jako výstup celého toku.

8.7 Tok OneOfSequentialFlow

Tok `OneOfSequentialFlow` spouští uzly postupně ve stejném pořadí, jako při vkládání uzlů do builderu. Funguje jako „fallback vodopád“, kde stačí jen, aby jeden uzel úspěšně zpracoval vstup.

Tok vrací zpracovanou vstupní hodnotu, pouze když nějaký z uzlů úspěšně skončí. V tomto případě je výstup z prvního úspěšného uzlu vrácen jako výstup toku. Všechny případy výstupu tohoto toku záleží na výstupu uzlů a zní takto:

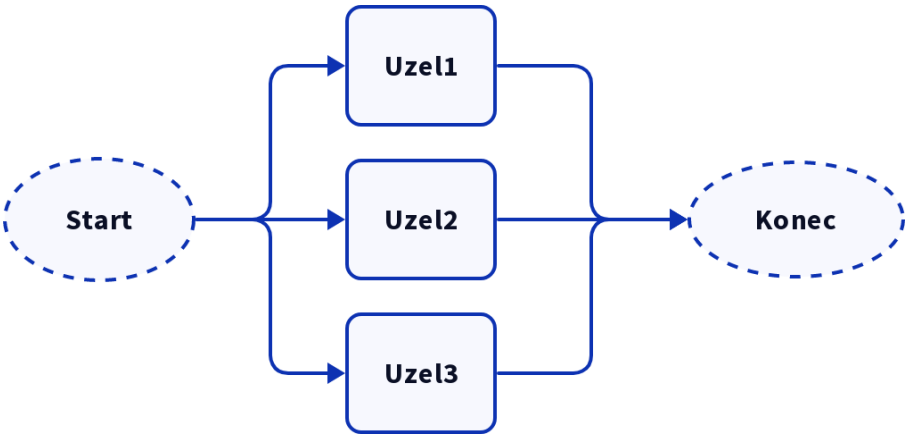
1. Pokud uzel vrátí `NodeOutput::Ok`, tak je výstupní hodnota tohoto uzlu vrácena tokem.
2. Pokud uzel vrátí `NodeOutput::SoftFail`, tak tok zkusí další uzel (větev toku).
3. Pokud uzel vrátí chybu, tak tok končí a vrací chybu.
4. Pokud všechny uzly vrátí `NodeOutput::SoftFail`, tak tok končí a také vrací soft-fail.

8.7.1 Definice toku a builderu

Kód 71: Implementace toku `OneOfSequentialFlow` - definice toku a builderu [Vlastní tvorba]

```
define_flow_and_ioe_conv_builder!(
    OneOfSequentialFlow,
    ChainRunOneOfSequential,
    |self| { .. },
    >Input: Send + Clone,
    >Context: Fork + Update + Send,
    #NodeType: Send + Sync + Clone
    ..
);
```

Pro definici toku a builderu je použito makro `define_flow_and_ioe_conv_builder` a běh toku je obstaráván `ChainRunOneOfSequential` traitem. Tělo `describe` metody vytváří acyklický orientovaný graf, kde každý uzel má jednu hranu z toku mířící do uzlu a jednu hranu vycházející z uzlu a mířící do toku.



Obrázek 7: Vizualizace toku `OneOfSequentialFlow` [Vlastní tvorba]

8.7.2 Privátní trait ChainRunOneOfSequential

Kód 72: Implementace toku `OneOfSequentialFlow` - definice traitu `ChainRunOneOfSequential` [Vlastní tvorba]

```
pub trait ChainRunOneOfSequential<Input, Output, Context, T> {
    fn run(&self, input: Input, context: &mut Context) -> impl
     ↪  Future<Output = Output> + Send;
}
```

Trait `ChainRunOneOfSequential` je skoro totožný v definici traitu `ChainRunSequential` a obsahuje veškerou logiku pro běh toku a správu asynchronních úloh. Generický typ `T` v definici traitu je použit pro uložení vstupních, výstupních a chybových datových typů uzlů.

Implementace traitu `ChainRunOneOfSequential` pro tuple list je také velmi podobná v tom, že definuje sekvenční vykonávání jednotlivých uzlů toku. Běh je implementován rekurzivní dekompozicí tuple listu na hlavu a ocas, kde metoda `run`, z `Node` traitu, je vyvolána nejprve na prvním uzlu v toku. Ale liší se funkcionalitou od `ChainRunSequential` traitu tím, že úspěšný výstup uzlu je použit jako výstup toku a v případě chyby je běh toku ukončen a chyba je vrácena jako výsledek toku. Dále se liší tím, že využívá traity `Fork` (sekce 6.1.1) a `Update` (sekce 6.1.2) ke správě kontextu.

Tato implementace zajišťuje, že uzly (větve toku) jsou spouštěny ve stejném pořadí, v jakém byly vloženy do builderu, a že výsledek toku je první úspěšný nebo chybový výsledek nějaké větve tohoto toku. Dále je také zajištěno, že každá větev toku dostane nezávislou instanci kontextu díky `Fork` traitu. Trait `Update` je poté použit pro aktualizaci originálního kontextu v případě, že nějaký uzel úspěšně skončí.

8.8 Tok ParallelFlow

Tok `ParallelFlow` spouští všechny uzly najednou a vrací výsledek sloučení všech výstupů uzlů takzvaným „joinerem“.

Tok vrací zpracovanou vstupní hodnotu, pouze když všechny uzly úspěšně skončí. V tomto případě je výstup ze všech uzlů zpracován uživatelem definovanou strukturou či funkcí, která se nazývá `Joiner`, a výsledek z této struktury je použit jako výsledek toku. V ostatních případech výstup tohoto toku záleží na výstupu uzlů a zní takto:

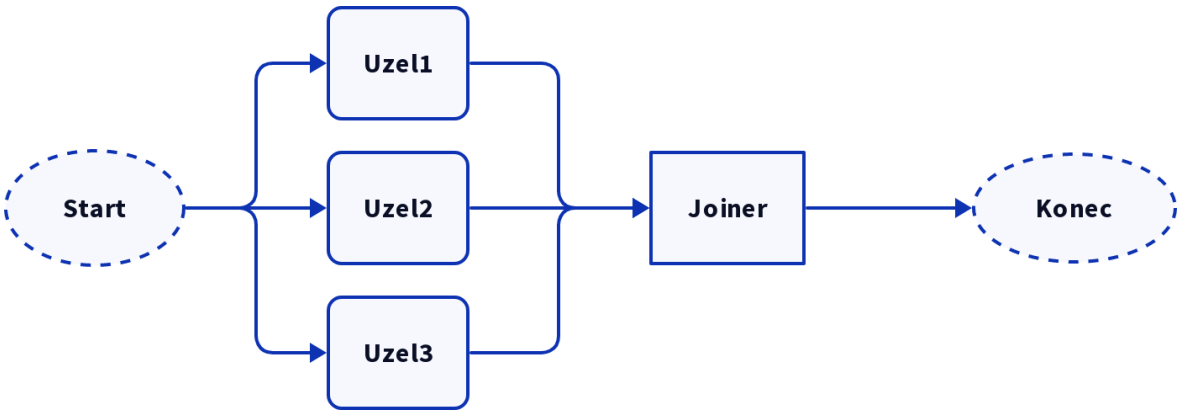
1. Pokud uzel vrátí `NodeOutput::Ok` nebo `NodeOutput::SoftFail`, tak tok čeká dále na dokončení ostatních uzlů (větví toku).
2. Pokud uzel vrátí chybu, tak tok končí a vrací chybu.

8.8.1 Definice toku

Kód 73: Implementace toku `ParallelFlow` - definice toku [Vlastní tvorba]

```
pub struct ParallelFlow<Input, Output, Error, Context, ChainOutput = (),
    Joiner = (), NodeTypes = (), NodeIOETypes = ()> {
    ..
    pub(super) nodes: NodeTypes,
    pub(super) joiner: Joiner,
}
```

Pro definici toku nemohlo být použito makro `define_flow`, protože tento tok navíc obsahuje takzvaný `Joiner`. Běh toku je obstaráván `ChainRunParallel` traitem a tělo `describe` metody vytváří acyklický orientovaný graf, kde každý uzel má jednu hranu vycházející z toku mířící do uzlu a jednu hranu vycházející z uzlu mířící do joineru. Tento graf má ještě jednu hranu vycházející z joineru a mířící do toku.



Obrázek 8: Vizualizace toku `ParallelFlow` [Vlastní tvorba]

8.8.2 Trait Joiner

Kód 74: Implementace toku `ParallelFlow` - trait `Joiner` [Vlastní tvorba]

```
pub trait Joiner<'a, Input, Output, Error, Context>: Send + Sync {
    fn join(
        &self,
        input: Input,
        context: &'a mut Context,
    ) -> impl Future<Output = NodeResult<Output, Error>> + Send;
}
```

Trait `Joiner` obsahuje pouze jednu metodu `join` a jeho účel je definovat, jak má být sloučen výstup ze všech uzlů `ParallelFlow` toku. Díky tomuto traitu je uživatel schopen definovat, jak mají být výstupy uzlů sloučeny. Je také možno s výstupy provést nějakou další operaci díky tomu, že metoda `join` je asynchronní a má velice podobnou definici jako trait `Node`.

Trait `Node` zde nemohl být použit, protože docházelo k problémům s lifetimy. Z tohoto důvodu byl vytvořen trait `Joiner`, který tento problém řeší.

Pro trait `Joiner` byla také vytvořena výchozí implementace pro jakýkoli datový typ implementující trait `Fn`. Díky této implementaci je možno použít jednoduchou asynchronní funkci pro sloučení výstupů uzlů (sekce 2.3.4, kód 5).

8.8.3 Definice builderu

Kód 75: Implementace toku `ParallelFlow` - definice builderu [Vlastní tvorba]

```
pub struct Builder<Input, Output, Error, Context, NodeTypes = (),
 ↪  NodeIOETypes = ()>
where
    Input: Send + Clone, Error: Send, Context: Fork + Join + Send,
{
    ..
    nodes: NodeTypes,
}

impl_debug_for_builder!("ParallelFlow", Builder, Input: Send + Clone,
 ↪  Error: Send, Context: Fork + Join + Send);
```

Definice builderu má stejnou základní strukturu, jako builder vytvořený makrem `define_builder`, ale liší se v požadavcích na uzly vkládané do builderu a v konečném sestavení toku.

Implementace metody `add_node` je rozdělena do dvou případů, metoda pro první uzel a metoda pro ostatní uzly, ale oba tyto případy dělají funkcionálně to samé. Toto rozdělení je nutné pro kompatibilitu s traity `ChainDebug` a `ChainDescribe`, které očekávají specifickou strukturu tuple listu. Dále bylo toto rozdělení ponecháno proto, aby bylo možno implementovat nové traity jen pro jeden typ tuple listu.

Implementace metody `add_node` zajišťuje, že vstup do toku lze převést na vstup vkládaného uzlu a chybu z vkládaného uzlu lze převést na chybu toku.

Implementace metody `build` bere nějakou strukturu implementující trait `Joiner` a zajišťuje, že vstup joineru je stejný s výstupem všech uzlů, a že výstup joineru je stejný jako výstup toku.

Implementace metody `build` se nachází v přílohách (kód 88). Metodu `build` lze volat pouze po vložení prvního uzlu do builderu. Tímto se zajistí, že konečný tok není prázdný.

8.8.4 Privátní trait ChainRunParallel

Kód 76: Implementace toku `ParallelFlow` - definice traitu `ChainRunParallel` [Vlastní tvorba]

```
1 pub trait ChainRunParallel<Input, Output, Context, T> {
2     fn run(&self, input: Input, context: &mut Context) -> impl
      ↪  Future<Output = Output> + Send;
3 }
```

Trait `ChainRunParallel` spravuje privátní traity pro vytváření, spouštění a čekání na asynchroní úlohy a z pohledu uživatele obsahuje veškerou logiku pro běh toku a správu asynchronních úloh. Generický typ `T` v definici traitu je použit pro uložení vstupních, výstupních a chybových datových typů uzlů.

Implementace traitu `ChainRunParallel` pro tuple list definuje souběžné vykonávání uzlů toku. Běh je implementován za pomocí rekurzivní dekompozice tuple listu na hlavu a ocas. Dále jsou definovány a použity traity `ChainSpawn`, sloužící k vytvoření asynchronních úloh, a `ChainPollParallel`, sloužící k čekání na dokončení asynchronních úloh, zpracování a správu jejich výstupů. Definice traitů `ChainSpawn` a `ChainRunParallel` se nachází v přílohách (kód 89).

Tok končí úspěšně, jen když všechny uzly a joiner skončí úspěšně. Při jakékoli chybě, buď z uzlů nebo joineru je tok ukončen a chyba je vrácena.

Tato implementace zajišťuje, že uzly jsou prováděny souběžně a nezávisle na sobě. K docílení této implementace jsou také použity traity `Fork` (sekce 6.1.1) a `Join` (sekce 6.1.3), které zajišťují, že každá větev toku dostane nezávislou instanci kontextu, a že kontexty je na konci toku možno sloučit do originálního kontextu.

8.9 Tok OneOfParallelFlow

Tok `OneOfParallelFlow` spouští všechny uzly najednou a vrací výsledek uzlu, který skončí nejdříve.

Tok vrací zpracovanou vstupní hodnotu, pouze když nějaký z uzlů úspěšně skončí. V tomto případě je výstup z prvního úspěšného uzlu vrácen jako výstup toku. Všechny případy výstupu tohoto toku záleží na výstupu uzlů a zní takto:

1. Pokud uzel vrátí `NodeOutput::Ok`, tak je výstupní hodnota tohoto uzlu vrácena tokem.
2. Pokud uzel vrátí `NodeOutput::SoftFail`, tak tok čeká na dokončení dalšího uzlu (větve toku).
3. Pokud uzel vrátí chybu, tak tok končí a vrací chybu.
4. Pokud všechny uzly vrátí `NodeOutput::SoftFail`, tak tok končí a také vrací soft-fail.

8.9.1 Definice toku a builderu

Kód 77: Implementace toku `OneOfParallelFlow` - definice toku a builderu [Vlastní tvorba]

```
define_flow_and_ioe_conv_builder!(
    OneOfParallelFlow,
    ChainRunOneOfParallel,
    |self| { .. },
    >Input: Send + Clone,
    >Output: Send,
    >Error: Send,
    >Context: Fork + Update + Send,
    #NodeType: Send + Sync + Clone
    ..
);
```

Pro definici toku a builderu je použito makro `define_flow_and_ioe_conv_builder` a běh toku je obstaráván `ChainRunOneOfParallel` traitem. Definice toku a builderu je velmi podobná `OneOfSequentialFlow` toku, ale navíc od toku `OneOfSequentialFlow` tato definice klade dodatečné požadavky na generické parametry `Error` a `Output`. Tělo `describe` metody vytváří stejný acyklický orientovaný graf (obrázek 7), jako tok `OneOfSequentialFlow`, protože tyto toky mají stejnou strukturu, ale jinou funkcionalitu.

8.9.2 Privátní trait ChainRunOneOfParallel

Kód 78: Implementace toku `OneOfParallelFlow` - definice traitu `ChainRunOneOfParallel` [Vlastní tvorba]

```
pub trait ChainRunOneOfParallel<Input, Output, Context, T> {
    fn run(&self, input: Input, context: &mut Context) -> impl
     ↪  Future<Output = Output> + Send;
}
```

Trait `ChainRunOneOfParallel`, stejně jako trait `ChainRunParallel`, spravuje privátní traity pro vytváření, spouštění a čekání na asynchroní úlohy a z pohledu uživatele obsahuje veškerou logiku pro běh toku a správu asynchronních úloh. Generický typ `T` v definici traitu je použit pro uložení vstupních, výstupních a chybových datových typů uzlů.

Implementace traitu `ChainRunOneOfParallel` pro tuple list je také velmi podobná traitu `ChainRunParallel` v tom, že běh je implementován za pomocí rekurzivní dekompozice tuple listu na hlavu a ocas. Používají se traity `ChainSpawn`, sloužící k vytvoření asynchronních úloh, a `ChainPollOneOfParallel`, sloužící k čekání na dokončení asynchronních úloh, zpracování a správu jejich výstupů. Definice podobných traitů pro `ChainRunParallel` se nachází v přílohách (kód 89).

Ale liší se funkcionalitou od `ChainRunParallel` traitu tím, že úspěšný výstup uzlu je použit jako výstup toku a v případě chyby je běh toku ukončen a chyba je vrácena jako výsledek toku. Dále se liší tím, že ke správě kontextu využívá traity `Fork` (sekce 6.1.1), sloužící k poskytnutí nezávislé instance kontextu uzlu, a `Update` (sekce 6.1.2), sloužící k aktualizaci originálního kontextu.

Tato implementace zajišťuje, že uzly jsou prováděny souběžně a nezávisle na sobě, a že výsledek toku je první úspěšný nebo chybový výsledek nějaké větve tohoto toku.

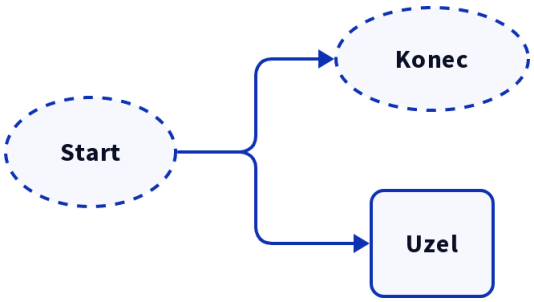
8.10 Tok Detached

Kód 79: Implementace toku `Detached` [Vlastní tvorba]

```
pub struct Detached<Input, Error, Context, NodeType = (), NodeOutput =
    (), NodeError = ()> {
    ..
    node: NodeType,
}
```

Tok `Detached` je obal, který vytvoří a spustí samostatnou asynchronní úlohu na jejíž dokončení nečeká a vrací vstup do tohoto obalu, jako výstup, bez jakékoli úpravy.

Je implementován za pomocí traitu `SpawnAsync` (sekce 6.2.2), pro vytvoření a spuštění asynchronní úlohy, a traitu `Fork` (sekce 6.1.1), pro získání nezávislé instance kontextu. V implementaci není použit žádný trait na synchronizaci kontextu, protože obalený uzel je spuštěn nezávisle na toku ve kterém byl definován. Tělo `describe` metody vytváří acyklický orientovaný graf, kde jedna hrana míří z toku do uzlu a další hrana míří ze začátku toku na konec toku.



Obrázek 9: Vizualizace toku `Detached` [Vlastní tvorba]

Tento tok umožňuje vytvářet a spouštět uzly, které jsou vedlejší a nepotřebné k běhu toku, běžících na pozadí bez toho, aby ovlivnily výsledek či průběh toku.

8.11 Tok FnFlow

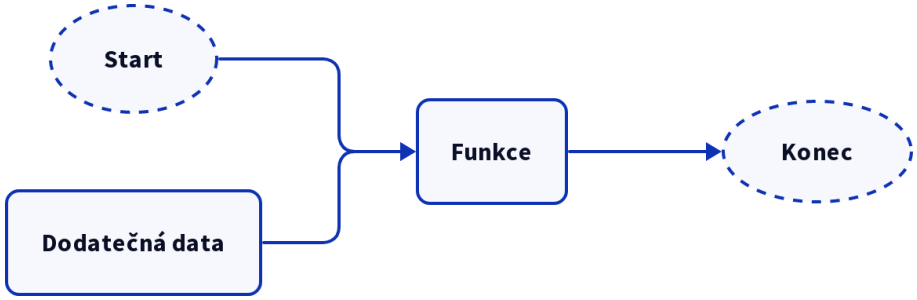
Kód 80: Implementace toku `FnFlow` - definice toku [Vlastní tvorba]

```
pub struct FnFlow<Input, Output, Error, Context, InnerData = (), R = ()>
    {
    ..
    inner_data: InnerData,
    runner_description: Option<Description>,
    runner: R,
}
```

Tok `FnFlow` je obal, který vytváří z nějaké synchronní funkce uzel implementující trait `Node`.

Tento obal přijímá asynchronní funkci či strukturu implementující trait `Runner`, dodatečná data, která mají být funkci poskytnuta spolu se vstupem, a nepovinný popis pro vytvořený uzel.

Implementace tohoto obalu je velice jednoduchá, protože jen volá poskytnutou funkci se vstupem a poskytnutými dodatečnými daty. Tělo `describe` metody vytváří acyklický orientovaný graf, kde dvě hrany míří do funkce, jedna z toku a druhá z dodatečných dat, a jedna hrana míří z funkce do toku.



Obrázek 10: Vizualizace toku `FnFlow` [Vlastní tvorba]

8.11.1 Trait Runner

Kód 81: Implementace toku `FnFlow` - definice traitu `Runner` [Vlastní tvorba]

```
pub trait Runner<'a, Input, Output, Error, Context, InnerData>: Send +
    Sync {
    fn run(
        &self,
        data: InnerData,
        input: Input,
        context: &'a mut Context,
    ) -> impl Future<Output = NodeResult<Output, Error>> + Send;
}
```

Trait `Runner` obsahuje pouze jednu metodu `run` a jeho účel je definovat, jak má být vstup a dodatečná data zpracována do nějakého výstupu.

Trait `Node` zde nemohl být použit kvůli limitaci jazyka. Rust zatím nedovoluje vytvořit výchozí implementaci pro jakýkoli datový typ, kterou by poté uživatel mohl přepsat. Dále také definice traitu `Runner` umožňuje lépe rozlišit, zda implementace je zamýšlena pro definici uzlu, nebo pro použití v `FnNode` toku.

Pro trait `Runner` byla také vytvořena výchozí implementace pro jakýkoli datový typ implementující trait `Fn`. Díky této implementaci je možné použít jakoukoli asynchronní funkci (sekce 2.3.4, kód 5).

9 Testování, dokumentace a dostupnost

Všechny zdrojové kódy implementace, testů a dokumentace knihovny je možné nalézt v jednom repozitáři na autorově GitHubu. Tento repozitář se nachází na adrese https://github.com/HANDZCZ/node-flow/.

V této práci je popsána verze knihovny `0.2.1` s MSRV (**M**inimum **S**upported **R**ust **V**ersion) `1.90.0`. Tato knihovna je také dostupná na oficiálním registru balíčků pro programovací jazyk Rust `crates.io`.[1] Na tomto registru je také možno najít dokumentaci pro jednotlivé verze knihovny a v repozitáři je odkaz na GitHub Pages,[2] který obsahuje aktuální dokumentaci pro větev `main`.

Dokumentace je vytvořena pro každou strukturu, která je uživateli této knihovny dostupná. Pro každou tuto strukturu je popsáno, co dělá a jak se chová, nebo k čemu slouží. Dále také dokumentace pro tyto struktury obsahuje ukázku použití a dodatečné odkazy na další struktury, které by uživateli umožnily lépe pochopit, jak se má daná struktura vytvořit a kde a k čemu je ji možno použít.

Pro struktury této knihovny, ať už privátní či veřejné, jsou vytvořeny unit testy, tam kde je potřeba testovat jejich funkčnost či jiné vlastnosti nebo omezení. Dále také existují testy na kontrolu, jak se toky chovají, když nejsou splněny jejich požadavky, aby práce s touto knihovnou byla co nejintuitivnější a aby chyby generované kompilátorem byly přehledné a dávali uživateli smysl.

Například u toků je testován zvlášť „chainrun“ a celý tok, aby se izolovalo, zda se jedná o chybu v implementaci „chainrun“ do toku či samotné logiky toku. Dalším příkladem jsou úložiště, kde je nutno otestovat nejen celé úložiště na povrchové úrovni, ale i jejich jednotlivé části.

V přílohách (kód 90) je uveden zkráceny běh testů, ve kterém lze vidět, že testy se nacházejí na více místech. Většina testů se nachází přímo u definice struktur, ale komplexnější testy, které například skládají více toků do sebe, se nachází v samostatných souborech. Dále je vidět, že ukázky v dokumentaci jsou také testovány, za pomocí takzvaných „Doc-testů“.

[1] https://crates.io/crates/node-flow
[2] https://handzcz.github.io/node-flow/

10 Experimenty

Pro generování kódů je použit nástroj cargo-show-asm.[1] Tento nástroj je využit pro získání assembly, LLVM a MIR (**M**id-level **I**ntermediate **R**epresentation) kódů.

Všechny zdrojové kódy byly obaleny do takzvaného „polleru," jehož jediným úkonem je volat metodu `Future::poll` (sekce 2.3.2) do té doby, dokud asynchronní úloha neskončí nějakým výsledkem. Tato funkce byla použita namísto asynchronního runtimu (sekce 2.3.3), protože hlavním cílem je zjistit, jaký kód je generován a asynchronní runtime je příliš komplexní pro tyto experimenty. Funkce `poller` také umožňuje izolovat generovaný kód, pouze na ten, který se vztahuje k testovanému případu.

Dále byl využit takzvaný `black_box`,[2] který značí kompilátoru, že má být co nejpesimističtější o tom, co `black_box` je a umí. Díky této funkci je možné zabránit, aby kompilátor odoptimalizoval části kódu.

10.1 Vliv inliningu a monomorfizace na jediný a samotný uzel

V kapitole „Kontext za pomocí generického typu" (sekce 4.6.2) je uveden kód „Příklad použití traitu `Node` s generickým kontextem" (kód 27). Tento kód lze využít skoro beze změny a poslouží k testu, zda inlining (sekce 2.6) a monomorfizace (sekce 2.5) fungují pro tento velmi jednoduchý příklad.

Kód 82: Vygenerovaný assembly kód pro jeden samostatný uzel [Vlastní tvorba]

```
1 node_flow_testing::poller:
2     mov al, 17
3     ret
```

V kódu výše je vidět, že celá asynchronní úloha byla odoptimalizována a nahrazena výsledkem. Nedochází tedy za běhu k žádným výpočtům a výsledná hodnota je přímo vrácena.

[1] https://github.com/pacak/cargo-show-asm
[2] https://doc.rust-lang.org/beta/std/hint/fn.black_box.html

Tento případ není ale moc realistický, protože vstupy jsou známy dopředu a pro kompilátor je hračka celé tělo nahradit výsledkem.

Kód 83: Vygenerovaný assembly kód pro jeden samostatný uzel s `black_boxem` [Vlastní tvorba]

```
node_flow_testing::poller:
    mov byte ptr [rsp - 2], 5
    lea rax, [rsp - 2]
    lea rcx, [rsp - 1]
    mov byte ptr [rsp - 1], 12
    movzx eax, byte ptr [rsp - 2]
    add al, byte ptr [rsp - 1]
    ret
```

Pokud jsou hodnoty `5` a `12` obaleny do `black_boxu`, tak je vidět, že už dochází ke sčítání operandů, ale nedochází k žádnému spouštění asynchronních úloh díky tomu, že definovaný uzel neobsahuje žádný await bod.

Práce se stackem je vedlejší účinek `black_boxu` a při použití argumentů do funkce `poller` místo `black_boxu` je generována pouze jedna instrukce sčítající argumenty.

Kód 84: Pozměněná implementace metody `Node::run` pro jeden samostatný uzel [Vlastní tvorba]

```
fn run(&mut self, input: u8, context: &mut Context) -> impl Future<Output
 ↪  = Result<u8, ()>> + Send {
    let fut: Pin<Box<dyn Future<Output = Result<u8, ()>> + Send>> =
     ↪  Box::pin(async move { Ok(input + context.how_much()) });
    fut
}
```

Byl také proveden jednoduchý test, kde originální kód příkladu byl pozměněn tak, že implementace metody `Node::run` (sekce 5.1) vracela trait objekt `Pin<Box<dyn Future>>`.

Vygenerovaný kód byl mnohem delší a věnoval se hlavně alokaci a dealokaci asynchronní úlohy na heap. Získání výsledku byla jen malá část, ve které ani neproběhl žádný výpočet, protože hodnota `input` a výstup metody `HowMuch::how_much` jsou známy při kompilaci. Podobný fenomén také nastal, když byl místo konkrétního typu kontextu použit trait objekt `Box<dyn HowMuch>`.

10.2 Vliv inliningu a monomorfizace na toky

Pro experimenty s toky byl použit stejný uzel a kontext jako při experimentech s jedním uzlem. V této kapitole je hlavně testováno kolik uzlů, pouze se synchronním tělem metody `Node::run`, je schopen kompilátor odoptimalizovat bez vytváření struktur pro asynchronní úlohy nebo jiných nadbytečných instrukcí.

Tabulka 3: Počet uzlů úspěšně odoptimalizovaných kompilátorem pro každý tok [Vlastní tvorba]

Tok	Počet uzlů
`SequentialFlow`	6
`OneOfSequentialFlow`	59*
`ParallelFlow`	0*
`OneOfParallelFlow`	11

V tabulce výše je vidět, že pro každý tok existuje různý limit počtu uzlů, které kompilátor dokáže odoptimalizovat. U toku `OneOfSequentialFlow` bylo potřeba navýšit tzv. „recursion_limit“ na 301 a u toku `ParallelFlow` jsou generovány dodatečné instrukce vždy, protože alokuje paměť na heap. Bez tohoto omezení by tok `ParallelFlow` dokázal odoptimalizovat aspoň jeden uzel, což bylo možné vidět ve vygenerovaném assembly kódu.

Tok `OneOfSequentialFlow` si vedl velmi dobře, protože kompilátoru jen stačilo dokázat, že první uzel vrátí jakýkoli výsledek kromě `NodeOutput::Softfail` (sekce 5.2). Pro ostatní toky, ale kompilátor musel projít všechny uzly. Optimalizace u toku `OneOfParallelFlow` je velmi překvapivá a autor očekával, že počet odoptimalizováných uzlů bude nižší než pro tok `SequentialFlow`. Protože pro odoptimalizování toku `OneOfParallelFlow` musí kompilátor dokázat, že všechny uzly v toku vrací stejnou hodnotu, nebo zjistit, který uzel skončí jako první a nevrací `NodeOutput::Softfail` (sekce 5.2).

Limity pro tyto toky jsou také nižší, díky tomu, že tato knihovna je asynchronní, což způsobuje, že existují dodatečné hranice a přidaná komplexita způsobena asynchronním zpracováním. Když jsou tyto hranice překročeny, tak se kompilátor dále nesnaží, nebo již nedokáže odoptimalizovat asynchronní kód.

Překročení těchto hranic způsobuje, že je vytvářen dodatečný kód pro asynchronní zpracování. Bohužel bez dalších nových funkcí jazyka, jako jsou variadické argumenty či funkce, specializace, explicitní „tail call“ (koncové rekurze/volání) optimalizace, nebo lépe optimalizovaného kompilátoru tato knihovna už naráží na limitace jazyka.

Právě díky těmto limitacím a různým heuristikám kompilátoru je nejspíše prakticky nemožné zamezit kompilátoru vytvářet asynchronní kód při plně synchronních tělech uzlů. To, že je tato knihovna schopna poskytnout tuto možnost v nějaké kapacitě akorát potvrzuje, že návrh a implementace umožňuje tohoto jevu dosáhnout, a že kompilátor je schopen odoptimalizovat části kódu napříč uzly a await body.

10.2.1 Vliv na skládání toků

Jedním z důvodů, proč tato knihovna nepoužívá trait objekty je, že bez jejich použití by měl být kompilátor schopen sloučit, nebo i odstranit nějaké await body napříč skládanými toky. Tím tak docílit, že výsledný kód bude rychlejší, více optimalizován a nemusí se zabývat zbytečnou dereferencí na konkrétní datové typy.

Pro otestování, zda je kompilátor schopen odoptimalizovat toky složené z dalších toků, byl vytvořen experiment, který skládá 3 `SequentialFlow` toky do sebe. Každý tok obsahuje aspoň dva další jednoduché uzly, kromě jednoho vnitřního toku. Vizualizaci tohoto toku lze nalézt v ukázkách (obrázek 12).

Kód 85: Vygenerovaný assembly kód pro kompozici uzlů [Vlastní tvorba]

```
1 node_flow_testing::poller:
2     mov al, 52
3     ret
```

Jak je možné vidět z kódu výše, tak kompilátor dokázal odoptimalizovat celý složený tok. Tohoto výsledku byl kompilátor schopen docílit, jen díky tomu, že znal všechny konkrétní datové typy obsažené v toku a operandy použité při výpočtech.

V tomto experimentu si lze povšimnout toho, že i když se tok skládá z 8 uzlů a 4 toků, tak je tento celý tok odoptimalizován. Zajímavé na tomto jevu

je, že kdyby se použil jen jeden tok `SequentialFlow`, tak by k odoptimalizování nedošlo. Jak je vidět v tabulce 3, tak limit pro tento tok je 6 uzlů. Dále je potřeba zmínit, že pro tento experiment i experiment pro maximální počet uzlu bylo použito úplně stejné prostředí - stejný uzel, kontext, verze jazyka a nastavení optimalizací kompilátoru. Což nejspíše znamená, že kompilátor byl schopen postupně odoptimalizovat vnitřní toky a postupovat ke vnějšímu toku a ten poté taky odoptimalizovat.

Kód 86: Vygenerovaný assembly kód pro kompozici uzlů s `black_boxem` [Vlastní tvorba]

```
node_flow_testing::poller:
    mov byte ptr [rsp - 2], 5
    lea rax, [rsp - 2]
    lea rcx, [rsp - 1]
    mov byte ptr [rsp - 1], 12
    movzx eax, byte ptr [rsp - 2]
    shl al, 3
    add al, byte ptr [rsp - 1]
    ret
```

Podobně jako u experimentu s jedním uzlem byl použit `black_box`, aby kompilátor vygeneroval kód, jako kdyby hodnoty operandů byly neznámé.

Z assembly kódu výše je vidět, že kompilátor odoptimalizoval veškerý asynchronní kód a nahradil jej instrukcemi `add` a `shl`. Celý tok byl tedy redukován jen na bitový posun sčítané hodnoty a k výsledku byl poté přičten vstup.

10.3 Maximální hloubka toku

V kapitole „Implementace toků“ (sekce 8) bylo vzneseno tvrzení „teoreticky lze vytvářet vnořené toky s nekonečnou hloubkou, avšak v realitě by uživatel jistě narazil na nějaký limit.“ Pro zjištění právě tohoto limitu byl vytvořen experiment, který skládá `SequentialFlow` toky do sebe. Tok `SequentialFlow` byl vybrán, protože při reálném použití je tento tok nejvíce využíván.

Při výchozím nastavení „recursion_limit“, což je 128, bylo dosaženo hloubky 25 a i při této hloubce byl kompilátor schopen odoptimalizovat celý složený tok na jednu `add` instrukci. Po překročení této hloubky již bylo nutno navýšit „recursion_limit“.

Tento limit byl poté navyšován až na hodnotu 8 191 a bylo dosaženo aspoň hloubky 1 296. Při této hloubce byl ještě kompilátor schopný program zkompilovat. Při dalším navýšení hloubky o 128, již přeteče stack kompilátoru. Tento test proběhl na Linux systému a kompilátor měl k dispozici výchozí velikost stacku, což je 8 MiB.

10.4 Maximální šířka toku

Podobně jako proběhl experiment pro zjištění maximální hloubky toku, tak proběhl experiment pro zjištění maximální „šířky“ toku. Šířkou toku je myšleno kolik uzlů je možné do jednoho toku vložit, než je dosaženo nějakého limitu. Pro tento experiment byl také použit tok `SequentialFlow` ze stejných důvodů.

Při výchozím nastavení „recursion_limit“, bylo dosaženo šířky 23. Po překročení této šířky již bylo nutno navýšit „recursion_limit“.

Tento limit byl poté navyšován až na hodnotu 1 644 a bylo dosaženo šířky 327. Při této šířce byl ještě kompilátor schopný program zkompilovat. Při navýšení šířky na 328, již přeteče stack kompilátoru stejně jako u experimentu maximální hloubky. Tento test také proběhl na Linux systému a kompilátor měl k dispozici 8 MiB stacku.

10.5 Maximální kombinovaná hloubka a šířka toku

Experimenty pro maximální hloubku a šířku probíhaly pouze s jedním uzlem, nebo s jedním tokem, ale při reálném použití této knihovny by toky měly různé hloubky a šířky. V tomto experimentu bylo testováno, jaký je limit pro toky složené z dalších toků, které obsahují nějaké uzly či další toky.

Poměry toků, uzlů a hloubky byly odhadnuty tak, aby nejvíce odpovídaly reálné práci s touto knihovnou. Aplikovaním těchto poměrů vznikl tok, který obsahoval 4 673 toků, 23 008 uzlů, dosáhl hloubky 358 a potřeboval „recursion_limit“ nastavený na 2 112. Při přidání dalších 13 toků, 64 uzlů a navýšeni hloubky na 359 kompilace selže při linkovací fázi, protože velikost debug symbolů přesahuje velikost sekce. Po vypnutí generace debug symbolů kompilace skončí úspěšně.

10.6 Shrnutí výsledků

Z experimentů lze zjistit, že implementace statických toků byla úspěšná a kompilátor je schopen odoptimalizovat await body nejen napříč uzly toku, ale i napříč sloučenými toky a jejich uzly.

Bohužel bylo také zjištěno, že tento jev není nekonečný a existují limity, a při jejich překročení je generován kód pro asynchronní zpracování. Tomuto jevu se však nedá vyhnout při práci s asynchronními úlohami a nikdy to nebyl ani cíl této knihovny.

Dále také byly zjištěny praktické limity pro tyto statické toky, při kterých je program úspěšné zkompilován a běží, tak jak má. Tyto limity by měly být dostačující pro jakékoli použití těchto statických toků. Pokud tyto limity, ale nejsou dostačující, tak budou řešeny budoucími verzemi této knihovny.

11 Porovnání s existujícími řešeními

Tato knihovna zatím nemá žádnou konkurenci, ani podobná řešení se stejnými ideály. Existují řešení s podobným nápadem, poskytnutí knihovny pro vytváření toků, ale tyto knihovny využívají pouze trait objekty a sdílené úložiště bez explicitních vstupů a výstupů.

Jednou z těchto knihoven je knihovna `freactor`,[1] která reprezentuje základní asynchronní jednotku práce asynchronní funkcí, která má jediný vstup, což je kontext. Všechna práce s daty probíhá přes kontext, což je jednoduché úložiště s klíčem `String` a nějakou hodnotou typu JSON. Typ toho úložiště je pevně daný, uživatel nemůže použít jiný typ úložiště, a nemůže na toto úložiště klást žádné požadavky. Uzly v tocích jsou uloženy za pomocí trait objektů a dynamic dispatch probíhá vždy.

Díky tomu, že knihovna `freactor` výhradně využívá trait objekty oproti této knihovně, tak může nabízet dynamické vytváření toků za běhu programu. Tohoto rozdílu knihovna využívá velmi dobře, protože nabízí definici celého toku jen za pomocí jednoho JSON dokumentu, který obsahuje název asynchronních funkcí a jak jsou mezi sebou propojeny. Bohužel oproti této knihovně, knihovna `freactor` nemá žádnou dokumentaci, kromě úvodní stránky na repozitáři, a nebylo ji možno dále porovnat s autorovou knihovnou a zjistit, zda podporuje skládání toků.

Další podobnou knihovnou je knihovna `graph-flow`,[2] která reprezentuje základní jednotku strukturou implementující trait `Task`, podobně jako tato knihovna reprezentuje základní jednotku traitem `Node`, obsahuje trait `Task` metodu `run` pro definici běhu. Tato funkce, ale podobně jako předchozí knihovna bere pouze kontext. Dále také trait `Task` obsahuje metodu `id`, která vrací unikátní identifikátor pro uzel v toku.

Kontext v knihovně `graph-flow` je implementován úplně stejně jako v předchozí knihovně a se stejnými limitacemi. Dále díky tomu, že také výhradně

[1] https://crates.io/crates/freactor
[2] https://crates.io/crates/graph-flow

využívá trait objekty, tak je možné vytvářet toky dynamicky. Oproti předchozí knihovně si liší v tom, že umožňuje používat `if/else` logiku pro hrany mezi uzly, toky se vytvářejí přes metody, nemá jednoduché vytváření toků přes JSON, a má dokumentaci.

Autorem vytvořená knihovna v porovnání s knihovnami `freactor` a `graph-flow` nepodporuje vytváření toků dynamicky a za běhu programu, kvůli tomu, že tato verze knihovny se soustředí na implementaci statických toků. Díky tomu je ale schopna, na rozdíl od porovnávaných knihoven, provést kontrolu požadavků na kontext, vstupních, výstupních a chybových datových typů při kompilaci programu. Oproti těmto knihovnám také umožňuje skládání toků a možnost odoptimalizace await bodů. Dále nabízí rozsáhlou dokumentaci a aspoň jednu ukázku použití u každé funkce, metody či struktury. Obsahuje také plno testů, jak pro kontrolu logiky, tak i ergonomiky knihovny, které chybí v porovnávaných knihovnách.

12 Budoucí vývoj

Jak se ukázalo z experimentů, tak definice toků za pomocí static dipspatche (sekce 2.4.1) má své výhody, ale i nevýhody. Mezi tyto nevýhody hlavně patří větší velikost výsledného binárního souboru a delší kompilace. Z tohoto důvodu by se další verze knihovny měly soustředit na implementaci dalších toků, které mají stejnou funkcionalitu, ale používají dynamic dispatch (sekce 2.4.2).

Implementací těchto dynamic dispatch toků se uživateli poskytne možnost vybrat si, zda chce používat dynamic či static dispatch. Pro tvorbu dynamic dispatch uzlů by měl být implementován nějaký obal, který by dovolil použít trait objekty uzlů jako konkrétní uzly. Za pomocí těchto dodatečných implementací by poté bylo možno libovolně míchat static a dynamic dispatch ve stejném toku i při skládání toků.

Dále traity `Join` (sekce 6.1.3) a `Merge` (sekce 6.3.2) používá datový typ `Box<[Self]>`, který je pevně dán a vždy alokuje paměť na heap. Tyto traity by neměly využívat datový typ `Box<T>` (sekce 2.9.1), ale generický typ, který musí implementovat nějaké traity, jako je například trait `Index<Idx>`,[1] který slouží k získání elementu za pomocí nějakého indexu. Použitím generického typu si uživatel knihovny poté může vybrat, zda chce paměť alokovat na heap, nebo na stack a jaká struktura má být použita.

Budoucí verze knihovny by také mohla obsahovat vytváření toků dynamicky za běhu programu, například za pomocí JSON dokumentu. Toto by ale znamenalo přidat tok, který kontroluje veškeré datové typy za běhu programu. Což jde proti ideálům této knihovny, přenesení co nejvíce kontrol a možných chyb na krok kompilace. Z tohoto důvodu by mohla být vytvořena nová knihovna, která bude používat stavební bloky z této knihovny. Pokud by uživatel poté potřeboval vytvořit tok dynamicky, tak by importoval obě knihovny a mohl by je i kombinovat.

[1] https://doc.rust-lang.org/std/ops/trait.Index.html

13 Závěr

Cílem diplomové práce bylo provést analýzu funkčních a nefunkční požadavků, navrhnout možné řešení pro tvorbu a kompozici uzlů a toku, a implementovat jej.

Celé řešení pokud možno nemá využívat dynamic dispatch a trait objekty. Tohoto cíle bylo dosaženo pro toky, uzly a kontextový systém. Pro implementované úložiště však z praktických důvodů byly použity trait objekty. Toto rozhodnutí, ale neomezuje uživatele, protože díky implementaci generického kontextu, uživatel může použít jakoukoli implementaci úložiště a není mu vnucena ta poskytovaná touto knihovnou.

Použitím pouze static dispatche pro běh, definici a skládaní toků a uzlů by měl být kompilátor schopen lépe optimalizovat kód. Tohoto cíle bylo také dosaženo a kompilátor je schopen optimalizovat kód napříč uzly i toky.

Dalším cílem také bylo, aby uzly byly schopné klást požadavky na kontext a tím docílit toho, že je jim vždy poskytnuta nějaká funkcionalita. Tohoto cíle bylo také plně dosaženo a uzly mohou klást požadavky nejen na kontext, ale i na vstup a výstup. Tyto požadavky jsou také kontrolovány za kompilace a je možné předejít zbytečným chybám za běhu programu.

V závěru tedy se autorovi povedlo navrhnout a implementovat plně statickou knihovnu, která dovoluje provést veškeré kontroly, kompatibility mezi vstupem a výstupem uzlů či toků a požadavků na kontext, na krok kompilace programu. Dále také tato knihovna umožňuje kompilátoru lépe optimalizovat kód díky tomu, že kompilátor zná veškeré datové typy použité v toku.

Dále je také probírán budoucí vývoj, kde autor chce implementovat další toky, které tentokrát budou používat dynamic dispatch. Uživatel by byl poté schopen si vybrat, zda chce používat dynamic nebo static dispatch, nebo dokonce i jejich kombinaci.

14 Seznam použité literatury

[1] SCHWARZKOPF, Malte. *The Remarkable Utility of Dataflow Computing* [online]. B.m.: ACM SIGOPS. 7. březen 2020 [vid. 2025-11-20]. Dostupné z: https://www.sigops.org/2020/the-remarkable-utility-of-dataflow-computing/

[2] DENNIS, Jack. *Control Flow and Data Flow: Concepts of Distributed Programming* [online]. B.m.: Springer Berlin Heidelberg. 1986 [vid. 2025-11-20]. Dostupné z: https://doi.org/10.1007/978-3-642-82921-5_8

[3] САМОЙДЮК, Андрій a Оксана ОСТАПЧУК. *Optimizing High-Load Systems with Asynchronous Programming Techniques* [online]. 25. duben 2025 [vid. 2025-11-20]. Dostupné z: https://doi.org/10.31713/MCIT.2024.036

[4] BELSON, Bruce, Jason HOLDSWORTH, Wei XIANG a Bronson PHILIPPA. *A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems* [online]. B.m.: Association for Computing Machinery (ACM). květen 2019 [vid. 2025-11-20]. ISSN 1558-3465. Dostupné z: http://dx.doi.org/10.1145/3319618

[5] VERMA, Pragati. *Unlocking Performance: A Guide to Async Support in Django* [online]. 25. červen 2023 [vid. 2025-11-20]. Dostupné z: https://dev.to/pragativerma18/unlocking-performance-a-guide-to-async-support-in-django-2jdj

[6] THE RUST TEAM. *Enum Poll - The Rust Standard Library documentation* [online]. 19. leden 2026 [vid. 2026-01-29]. Dostupné z: https://doc.rust-lang.org/std/task/enum.Poll.html

[7] THE RUST TEAM. *Asynchronous Programming in Rust* [online]. 2. listopad 2025 [vid. 2026-01-29]. Dostupné z: https://rust-lang.github.io/async-book/index.html

[8] THE RUST TEAM. *Trait Future - The Rust Standard Library documentation* [online]. 19. leden 2026 [vid. 2026-01-29]. Dostupné z: https://doc.rust-lang.org/std/future/trait.Future.html

[9] PAXDIABLO. *What is polymorphism, what is it for, and how is it used?* [online]. B.m.: Stack Overflow. 23. červen 2009 [vid. 2025-11-21]. Dostupné z: https://stackoverflow.com/a/1031385

[10] THE RUST TEAM. *Object-Oriented Programming Features of Rust* [online]. 7. listopad 2024 [vid. 2025-11-21]. Dostupné z: https://doc.rust-lang.org/book/ch18-00-oop.html

[11] OSWALT, Matt. *Polymorphism in Rust* [online]. 22. červen 2021 [vid. 2025-11-21]. Dostupné z: https://oswalt.dev/2021/06/polymorphism-in-rust/

[12] SALEWSKI, S. *Further Trait Features - Rust for C-Programmers* [online]. 2025 [vid. 2025-11-21]. Dostupné z: https://rust-for-c-programmers.com/ch11/11_4_further_trait_features.html

[13] SMITH, Brandon. *Three Kinds of Polymorphism in Rust* [online]. 5. leden 2022 [vid. 2025-11-21]. Dostupné z: https://www.brandons.me/blog/polymorphism-in-rust

[14] THE RUST TEAM. *Monomorphization - Rust Compiler Development Guide* [online]. 28. červenec 2025 [vid. 2025-11-23]. Dostupné z: https://rustc-dev-guide.rust-lang.org/backend/monomorph.html

[15] THE RUST TEAM. *Generic Data Types: Performance of Code Using Generics - The Rust Programming Language* [online]. 7. listopad 2024 [vid. 2025-11-23]. Dostupné z: https://doc.rust-lang.org/book/ch10-01-syntax.html#performance-of-code-using-generics

[16] ALLRED, Robert.🦀 *Rust Tips: What is Monomorphization?* [online]. 31. květen 2024 [vid. 2025-11-23]. Dostupné z: https://medium.com/@lightworld/rust-tips-what-is-monomorphization-49b5840778fe

[17] NETHERCOTE, Nicholas. *Inlining - The Rust Performance Book* [online]. 7. září 2025 [vid. 2025-11-23]. Dostupné z: https://nnethercote.github.io/perf-book/inlining.html

[18] SHAH, Drashti. *Inlining in Rust: Understanding the Compiler's Role* [online]. 10. duben 2023 [vid. 2025-11-23]. Dostupné z: https://drashti-shah.medium.com/inlining-in-rust-understanding-the-compilers-role-e171600614d1

[19] THE RUST TEAM. *Module any - The Rust Standard Library documentation* [online]. 7. listopad 2025 [vid. 2025-11-24]. Dostupné z: https://doc.rust-lang.org/std/any/index.html

[20] THE RUST TEAM. *Struct TypeId - The Rust Standard Library documentation* [online]. 7. listopad 2025 [vid. 2025-11-24]. Dostupné z: https://doc.rust-lang.org/std/any/struct.TypeId.html

[21] THE RUST TEAM. *Trait Any - The Rust Standard Library documentation* [online]. 7. listopad 2025 [vid. 2025-11-24]. Dostupné z: https://doc.rust-lang.org/std/any/trait.Any.html

[22] THE RUST TEAM. *Dyn compatibility - The Rust Reference* [online]. 18. srpen 2025 [vid. 2025-11-28]. Dostupné z: https://doc.rust-lang.org/reference/items/traits.html#dyn-compatibility

[23] THE RUST TEAM. *Smart Pointers - The Rust Programming Language* [online]. 7. listopad 2024 [vid. 2025-11-29]. Dostupné z: https://doc.rust-lang.org/book/ch15-00-smart-pointers.html

[24] TORRES, Roger. *Smart Pointers in Rust: What, why and how?* [online]. 19. červenec 2021 [vid. 2025-11-29]. Dostupné z: https://dev.to/rogertorres/smart-pointers-in-rust-what-why-and-how-oma

[25] CHIGOZIE, Oduah. *What are Smart Pointers in Rust? Explained with Code Examples* [online]. 30. listopad 2024 [vid. 2025-11-29]. Dostupné z: https://www.freecodecamp.org/news/smart-pointers-in-rust-with-code-examples/

[26] OKOH, Ebina. *Understanding Smart Pointers in Rust: A Comprehensive Guide* [online]. 6. listopad 2023 [vid. 2025-11-29]. Dostupné z: https://perelyn-sama.medium.com/understanding-smart-pointers-in-rust-a-comprehensive-guide-cc06eb94a147

[27] THE RUST TEAM. *Module pin - The Rust Standard Library documentation* [online]. 19. leden 2026 [vid. 2026-01-30]. Dostupné z: https://doc.rust-lang.org/std/pin/index.html

15 Seznam obrázků, tabulek a kódů

Seznam obrázků

1 Porovnání asynchronního a synchronního zpracování [5] 3
2 Struktura trait objektu [11] 8
3 Návrh builderu pro toky [Vlastní tvorba] 24
4 Ilustrace použití úložiště [Vlastní tvorba] 26
5 Návrh úložiště pro trait `Node` [Vlastní tvorba] 28
6 Vizualizace toku `SequentialFlow` [Vlastní tvorba] 58
7 Vizualizace toku `OneOfSequentialFlow` [Vlastní tvorba] 61
8 Vizualizace toku `ParallelFlow` [Vlastní tvorba] 63
9 Vizualizace toku `Detached` [Vlastní tvorba] 68
10 Vizualizace toku `FnFlow` [Vlastní tvorba] 69
11 Příklad vytvořené vizualizace pomocí formátovače `D2Describer` [Vlastní tvorba] 91
12 Vizualizace toku pro experiment kompozice toků [Vlastní tvorba] . 94

Seznam tabulek

1 Popis `inline` atributu [17, 18] 10
2 Popis implementováných metod pro varianty traitu `Any` [21] 13
3 Počet uzlů úspěšně odoptimalizovaných kompilátorem pro každý tok [Vlastní tvorba] 74

Seznam kódů

1 Definice enumu `Poll` ve standardní knihovně [6] 3
2 Definice trait `Future` ve standardní knihovně [8] 4
3 Příklad definice bloku vracejícího asynchronní úlohu [Vlastní tvorba] 5
4 Příklad definice asynchronní funkce [Vlastní tvorba] 5
5 Příklad implementace traitu `Future` pro datový typ [Vlastní tvorba] 6
6 Příklad statického výběru [Vlastní tvorba] 7

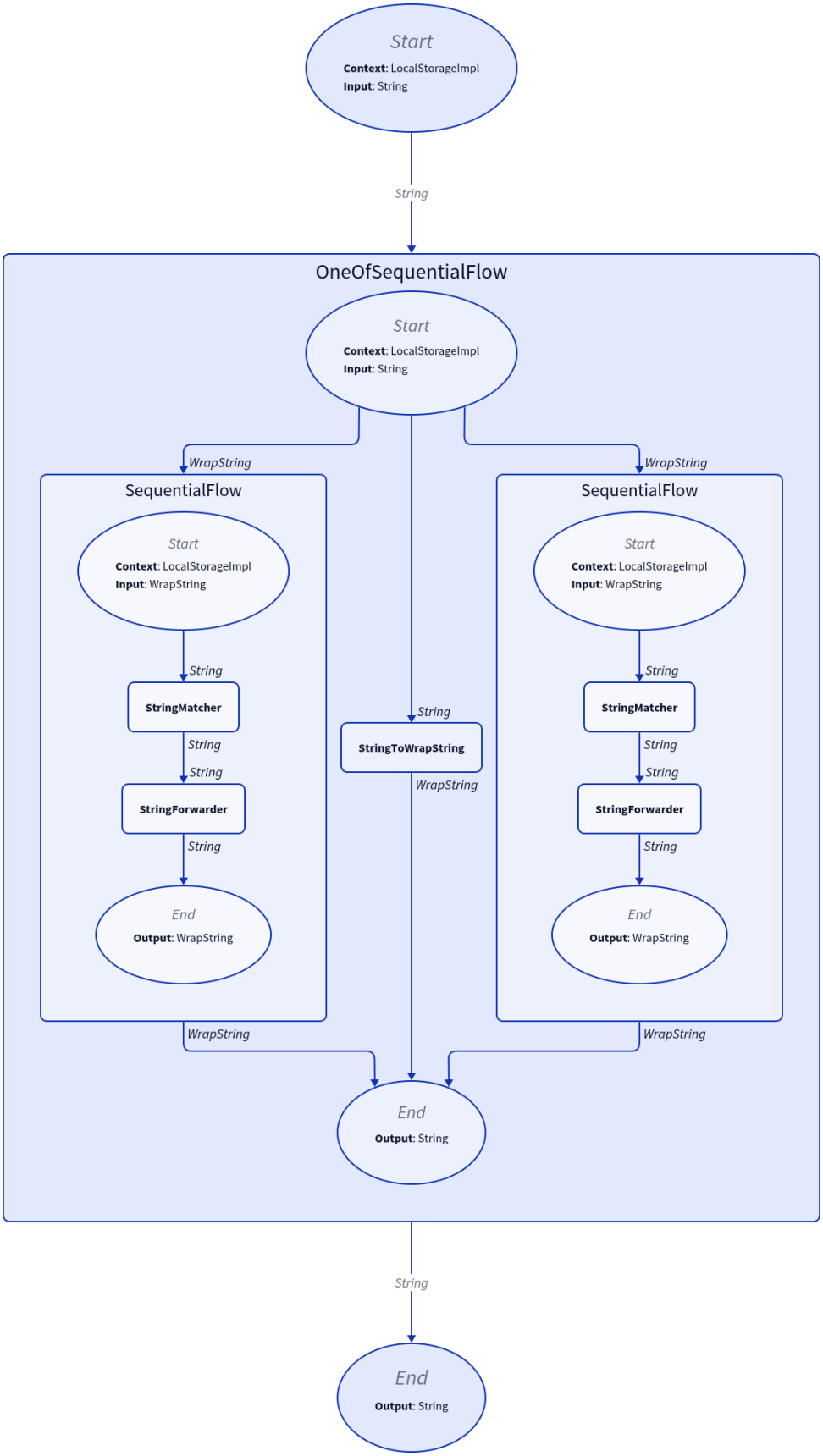
7 Příklad dynamického výběru [Vlastní tvorba] 8
8 Příklad kódu před monomorfizací [15] 9
9 Příklad kódu po monomorfizaci [15] 9
10 Příklad kódu před inliningem [Vlastní tvorba] 11
11 Příklad kódu po inliningu [Vlastní tvorba] 11
12 Příklad použití `TypeId::of<T>()` funkce [20] 12
13 Příklad použití traitu `Any` [21] 12
14 Příklad dyn kompatibilního traitu [Vlastní tvorba] 14
15 Příklad dyn nekompatibilního traitu [Vlastní tvorba] 14
16 Příklad použití smart pointeru `Box<T>` [Vlastní tvorba] 15
17 Příklad použití smart pointeru `Pin<T>` [27] 15
18 Příklad použití smart pointeru `Arc<T>` [Vlastní tvorba] 15
19 Příklad použití smart pointeru `Mutex<T>` [Vlastní tvorba] 16
20 Příklad použití smart pointeru `RwLock<T>` [Vlastní tvorba] 17
21 Tuple listy reprezentované kódem [Vlastní tvorba] 18
22 Příklad vytištění elementů v tuple listu [Vlastní tvorba] 18
23 Návrh traitu `Node` [Vlastní tvorba] 23
24 Návrh traitu `Flow` [Vlastní tvorba] 23
25 Návrh struktury objektu toku [Vlastní tvorba] 25
26 Ukázka návrhu implementace běhu toku pro paralelní tok [Vlastní tvorba] 26
27 Návrh traitu `Node` s úložištěm [Vlastní tvorba] 27
28 Návrh traitu `Node` s generickým kontextem [Vlastní tvorba] 29
29 Příklad použití traitu `Node` s generickým kontextem [Vlastní tvorba] 29
30 Chyba, která nastane při kompilaci, když nejsou splněny požadavaky na kontext [Vlastní tvorba] 30
31 Implementace traitu `Node` [Vlastní tvorba] 31
32 Implementace enumu `NodeOutput` [Vlastní tvorba] 32
33 Implementace traitu `BoxedNode` [Vlastní tvorba] 33
34 Příklad použití makra `impl_node_output` [Vlastní tvorba] 34
35 Implementace makra `node` [Vlastní tvorba] 34
36 Příklad použití makra `impl_node_output` [Vlastní tvorba] 34
37 Implementace traitu `Fork` [Vlastní tvorba] 35
38 Implementace traitu `Update` [Vlastní tvorba] 36

39 Implementace traitu `Join` [Vlastní tvorba] 36
40 Implementace traitu `Task` [Vlastní tvorba] 37
41 Implementace traitu `SpawnAsync` [Vlastní tvorba] 37
42 Implementace traitu `SpawnSync` [Vlastní tvorba] 38
43 Implementace enumu `MergeResult` [Vlastní tvorba] 38
44 Implementace traitu `Merge` [Vlastní tvorba] 39
45 Implementace traitu `LocalStorage` [Vlastní tvorba] 39
46 Implementace lokálního úložiště - struct `LocalStorageImpl` [Vlastní tvorba] 40
47 Implementace lokálního úložiště - trait `StorageItem` [Vlastní tvorba] 40
48 Implementace lokálního úložiště - implementace traitu `LocalStorage` - metoda `get_mut` [Vlastní tvorba] 42
49 Implementace traitu `SharedStorage` [Vlastní tvorba] 43
50 Implementace sdíleného úložiště - struct `SharedStorageImpl` [Vlastní tvorba] 44
51 Implementace sdíleného úložiště - implementace dočasných zámku [Vlastní tvorba] 44
52 Implementace sdíleného úložiště - implementace traitu `SharedStorage` - metoda `get` [Vlastní tvorba] 45
53 Implementace enumu `Description` [Vlastní tvorba] 47
54 Implementace structu `DescriptionBase` [Vlastní tvorba] 48
55 Implementace structu `Type` [Vlastní tvorba] 48
56 Implementace structu `Edge` a enumu `EdgeEnding` [Vlastní tvorba] . 49
57 Implementace structu `ExternalResource` [Vlastní tvorba] 50
58 Implementace funkce `remove_generics_from_name` [Vlastní tvorba] 50
59 Implementace structu `D2Describer` [Vlastní tvorba] 51
60 Implementace typových aliasů `NodeIOE`, `ChainLink` a `NodeResult` [Vlastní tvorba] 52
61 Implementace struktury enum `SoftFailPoll` [Vlastní tvorba] . . . 53
62 Implementace traitu `ChainDebug` [Vlastní tvorba] 53
63 Implementace traitu `ChainDescribe` [Vlastní tvorba] 54
64 Implementace makra `impl_debug_for_builder` [Vlastní tvorba] . 54
65 Implementace makra `define_flow` [Vlastní tvorba] 55
66 Implementace makra `define_builder` [Vlastní tvorba] 56

67 Implementace makra `define_builder` - implementace metody `add_node` [Vlastní tvorba] 57
68 Implementace toku `SequentialFlow` - definice toku [Vlastní tvorba] 58
69 Implementace toku `SequentialFlow` - definice builderu [Vlastní tvorba] 59
70 Implementace toku `SequentialFlow` - definice traitu `ChainRunSequential` [Vlastní tvorba] 60
71 Implementace toku `OneOfSequentialFlow` - definice toku a builderu [Vlastní tvorba] 61
72 Implementace toku `OneOfSequentialFlow` - definice traitu `ChainRunOneOfSequential` [Vlastní tvorba] 62
73 Implementace toku `ParallelFlow` - definice toku [Vlastní tvorba] . 63
74 Implementace toku `ParallelFlow` - trait `Joiner` [Vlastní tvorba] . 64
75 Implementace toku `ParallelFlow` - definice builderu [Vlastní tvorba] 64
76 Implementace toku `ParallelFlow` - definice traitu `ChainRunParallel` [Vlastní tvorba] 65
77 Implementace toku `OneOfParallelFlow` - definice toku a builderu [Vlastní tvorba] 66
78 Implementace toku `OneOfParallelFlow` - definice traitu `ChainRunOneOfParallel` [Vlastní tvorba] 67
79 Implementace toku `Detached` [Vlastní tvorba] 68
80 Implementace toku `FnFlow` - definice toku [Vlastní tvorba] 69
81 Implementace toku `FnFlow` - definice traitu `Runner` [Vlastní tvorba] 69
82 Vygenerovaný assembly kód pro jeden samostatný uzel [Vlastní tvorba] 72
83 Vygenerovaný assembly kód pro jeden samostatný uzel s `black_boxem` [Vlastní tvorba] 73
84 Pozměněná implementace metody `Node::run` pro jeden samostatný uzel [Vlastní tvorba] 73
85 Vygenerovaný assembly kód pro kompozici uzlů [Vlastní tvorba] . 75
86 Vygenerovaný assembly kód pro kompozici uzlů s `black_boxem` [Vlastní tvorba] 76
87 Implementace toku `SequentialFlow` - definice builderu - metoda `add_node`, pro ostatní uzly, a metoda `build` [Vlastní tvorba] 92

88 Implementace toku `ParallelFlow` - definice builderu - metoda `build` [Vlastní tvorba] 92
89 Implementace toku `ParallelFlow` - definice traitů `ChainSpawnParallel` a `ChainPollParallel` [Vlastní tvorba] . . . 93
90 Zkrácený výstup běhu testů [Vlastní tvorba] 93

16 Přílohy



Obrázek 11: Příklad vytvořené vizualizace pomocí formátovače `D2Describer` [Vlastní tvorba]

Kód 87: Implementace toku `SequentialFlow` - definice builderu - metoda `add_node`, pro ostatní uzly, a metoda `build` [Vlastní tvorba]

```
impl<..> Builder<
    Input, Output, Error, Context, NodeTypes,
    ChainLink<OtherNodeIOETypes, NodeIOE<LastNodeInType, LastNodeOutType, LastNodeErrType>>,
> where .. {
    pub fn add_node<NodeType, NodeInput, NodeOutput, NodeError>(self, node: NodeType) -> Builder<
        Input, Output, Error, Context,
        ChainLink<NodeTypes, NodeType>,
        ChainLink<
            ChainLink<OtherNodeIOETypes, NodeIOE<LastNodeInType, LastNodeOutType, LastNodeErrType>>,
            NodeIOE<NodeInput, NodeOutput, NodeError>,
        >,
    > where
        LastNodeOutType: Into<NodeInput>,
        NodeError: Into<Error>,
        NodeType: Node<NodeInput, NodeOutputStruct<NodeOutput>, NodeError, Context>,
        ..
    {
        Builder { .., nodes: (self.nodes, node) }
    }

    pub fn build(self) -> SequentialFlow<
        Input, Output, Error, Context, NodeTypes,
        ChainLink<OtherNodeIOETypes, NodeIOE<LastNodeInType, LastNodeOutType, LastNodeErrType>>,
    > where
        LastNodeOutType: Into<Output>,
    {
        Flow { .., nodes: Arc::new(self.nodes) }
    }
}
```

Kód 88: Implementace toku `ParallelFlow` - definice builderu - metoda `build` [Vlastní tvorba]

```
impl<..> Builder<
    Input, Output, Error, Context, NodeTypes,
    ChainLink<OtherNodeIOETypes, LastNodeIOETypes>,
> where .. {
    pub fn build<J, ChainRunOutput>(self, joiner: J) -> Flow<
        Input, Output, Error, Context, ChainRunOutput, J, NodeTypes,
        ChainLink<OtherNodeIOETypes, LastNodeIOETypes>,
    > where
        for<'a> J: Joiner<'a, ChainRunOutput, Output, Error, Context>,
        NodeTypes: ChainRunParallel<
            Input,
            Result<ChainRunOutput, Error>,
            Context,
            ChainLink<OtherNodeIOETypes, LastNodeIOETypes>,
       >,
    {
        Flow { .., nodes: Arc::new(self.nodes), joiner }
```

```
    }
}
```

Kód 89: Implementace toku `ParallelFlow` - definice traitů `ChainSpawnParallel` a `ChainPollParallel` [Vlastní tvorba]

```rust
pub trait ChainSpawn<Input, Error, Context, HeadOut, T> {
    type ChainOut;
    const NUM_FUTURES: usize;

    fn spawn(
        &self,
        input: Input,
        context: Context,
    ) -> impl ChainPollParallel<Self::ChainOut, Context>;
}

pub trait ChainPollParallel<Output, NodeContext>: Send {
    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        tail_ready: bool,
        context_acc: &mut Vec<NodeContext>,
    ) -> Poll<Output>;
}
```

Kód 90: Zkrácený výstup běhu testů [Vlastní tvorba]

```
╭─« node-flow on  main is 📦 v0.2.0 via 🦀 v1.93.1
╰─» cargo test --all-features
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running unittests src/lib.rs
      ↪  (target/debug/deps/node_flow-200139d2573f6e74)

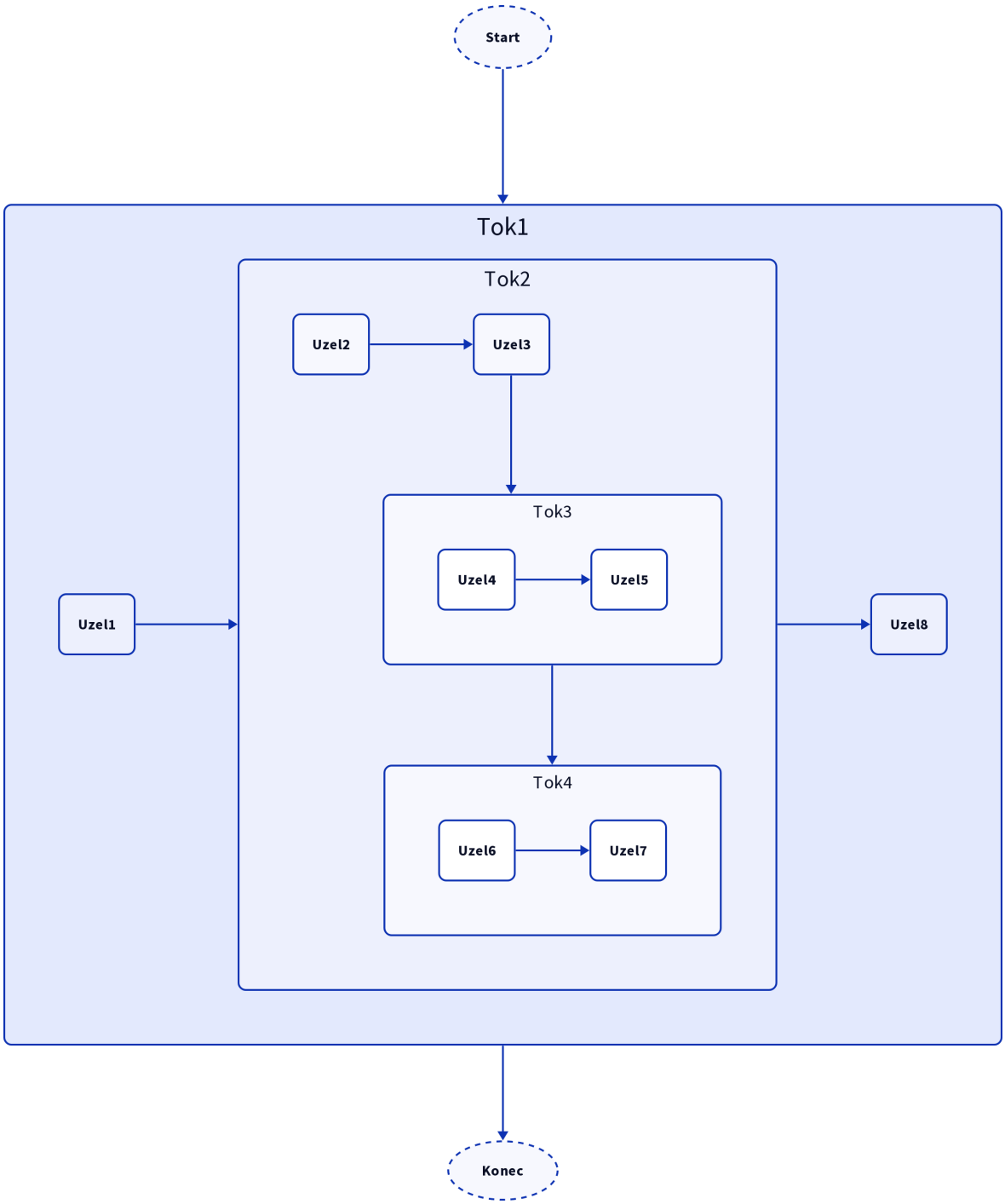
running 22 tests
test flows::parallel_flow::flow::test::test_flow_storage ... ok
..
test flows::one_of_sequential_flow::test::test_flow ... ok
test flows::sequential_flow::test::test_chain ... ok

test result: ok. 22 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
 ↪  out; finished in 0.46s
.
.
.
   Doc-tests node_flow

running 45 tests
test src/context/traits.rs - context::traits::SpawnSync (line 194) ... ok
test src/context/traits.rs - context::traits::Join (line 71) ... ok
..
test src/describe/d2.rs - describe::d2::D2Describer (line 9) ... ok

test result: ok. 45 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
 ↪  out; finished in 0.01s

all doctests ran in 0.83s; merged doctests compilation took 0.82s
```



Obrázek 12: Vizualizace toku pro experiment kompozice toků [Vlastní tvorba]

UNIVERZITA HRADEC KRÁLOVÉ

Fakulta informatiky a managementu

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Jan Najman**
Osobní číslo:	**I2400616**
Studijní program:	**N0613A140042 Aplikovaná informatika**
Téma práce:	**Návrh a implementace asynchronní knihovny pro tvorbu toků dat**
Zadávající katedra:	**Katedra informatiky a kvantitativních metod**

Zásady pro vypracování

Cílem diplomové práce je navrhnout a implementovat asynchronní knihovnu umožňující tvorbu a kompozici datových toků. Práce se zaměřuje na návrh architektury, definici základních typů toků a jejich implementaci.

Osnova:
I. Úvod
A. Kontext a motivace
B. Cíle práce
II. Teoretická část
A. Toky dat
B. Asynchronní programování
D. Polymorfismus
E. Monomorfizace a Inlining
III. Analýza požadavků
A. Funkční požadavky
B. Nefunkční požadavky
IV. Návrh
A. Definice uzlů a toků
B. Typování uzlů v toku
C. Kontextový systém
V. Implementace
A. Základní bloky knihovny
B. Popis realizace toků a kontextového systému
C. Testování
D. Dokumentace
VI. Vyhodnocení a experimenty
A. Vliv polymorfismu, monomorfizace a inliningu
B. Omezení jazyka a možné rozšíření
VII. Závěr
A. Shrnutí výsledků
B. Splnění cílů práce
C. Budoucí vývoj a doporučení
VIII. Seznam použité literatury

IX. Přílohy
A. Diagramy, schémata a kódy
B. Kódový repozitář
C. API reference

Rozsah pracovní zprávy:
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

SCHWARZKOPF, Malte. The Remarkable Utility of Dataflow Computing [online]. B.m.: ACM SIGOPS. 7. březen 2020 [vid. 2025-11-20]. Dostupné z: https://www.sigops.org/2020/the-remarkable-utility-of-dataflow-computing/
DENNIS, Jack. Control Flow and Data Flow: Concepts of Distributed Programming [online]. B.m.: Springer Berlin Heidelberg. 1986 [vid. 2025-11-20]. Dostupné z: https://doi.org/10.1007/978-3-642-82921-5_8
BELSON, Bruce, Jason HOLDSWORTH, Wei XIANG a Bronson PHILIPPA. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems [online]. B.m.: Association for Computing Machinery (ACM). květen 2019 [vid. 2025-11-20]. ISSN 1558-3465. Dostupné z: http://dx.doi.org/10.1145/3319618
THE RUST TEAM. Object-Oriented Programming Features of Rust [online]. 7. listopad 2024 [vid. 2025-11-21]. Dostupné z: https://doc.rust-lang.org/book/ch18-00-oop.html
OSWALT, Matt. Polymorphism in Rust [online]. 22. červen 2021 [vid. 2025-11-21]. Dostupné z: https://oswalt.dev/2021/06/polymorphism-in-rust/
THE RUST TEAM. Monomorphization – Rust Compiler Development Guide [online]. 28. červenec 2025 [vid. 2025-11-23]. Dostupné z: https://rustc-dev-guide.rust-lang.org/backend/monomorph.html
THE RUST TEAM. Generic Data Types: Performance of Code Using Generics – The Rust Programming Language [online]. 7. listopad 2024 [vid. 2025-11-23]. Dostupné z: https://doc.rust-lang.org/book/ch10-01-syntax.html#performance-of-code-using-generics
SHAH, Drashti. Inlining in Rust: Understanding the Compiler's Role [online]. 10. duben 2023 [vid. 2025-11-23]. Dostupné z: https://drashti-shah.medium.com/inlining-in-rust-understanding-the-compilers-role-e171600614d1

Vedoucí diplomové práce: **prof. RNDr. PhDr. Antonín Slabý, CSc.**
Katedra informatiky a kvantitativních metod

Datum zadání diplomové práce: **18. prosince 2024**

L.S.

prof. Ing. Mgr. Petra Marešová, Ph.D., MBA v.r.
děkanka

Mgr. Jana Medková, Ph.D. v.r.
vedoucí katedry

V Hradci Králové dne 18. prosince 2024