Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Subštrukturálne typové systémy v praxi

Diplomová práca

2024
Bc. Patrik Grman

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Subštrukturálne typové systémy v praxi

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Richard Ostertág PhD.

Bratislava, 2024
Bc. Patrik Grman

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Patrik Grman
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Subštrukturálne typové systémy v praxi
Substructural type systems in practice

Anotácia: Statická kontrola programov je prvá možnosť automatického odhaľovania chýb. Skoršie odhalenie chyby vedie k rýchlejšej a často lacnejšej náprave. Súčasťou statickej kontroly je sémantická analýza, ktorá zahŕňa aj typovú kontrolu. Niektoré sémantické požiadavky však nie sú kontrolovateľné použitím štandardných štruktúrovaných typových systémov.

Cieľom práce je spraviť prehľad a preskúmať rôzne nástroje ponúkané modernými programovacími jazykmi (lineárne funkcie v jazyku Haskell, ownership v jazyku Rust a move semantics v jazyku C++), z hľadiska realizácie subštrukturálnych typových systémov. Ďalším cieľom práce je demonštrovať aplikáciu týchto techník na príklade zabezpečenia správneho použitia príležitostného slova počas kompilácie.

Vedúci: RNDr. Richard Ostertág, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 26.11.2021

Dátum schválenia: 10.01.2022 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: S úctou ďakujem školiteľovi RNDr. Richardovi Ostertágovi PhD. za jeho cenné vedenie, odborné rady a podporu počas celého procesu písania diplomovej práce. Poďakovanie tiež patrí Mattovi Godboltovi za excelentý nástroj na pozorovanie výsledkov kompilácie v mnohých jazykoch.

Abstrakt

V práci sa zaoberáme typovou kontrolou kompilovaných jazykov, pričom skúmame možnosti typových systémov, v ktorých neplatia všetky štrukturálne pravidlá. Podľa toho sa nazývajú subštrukturálne typové systémy, a umožňujú v zmysle typovej kontroly zachytiť niektoré dodatočné požiadavky na prácu s hodnotami. Naša práca nadväzuje na článok [15], ktorý využitie subštrukturálnych typov ukazoval na príklade príležitostných slov v jazyku Rust. Hlavnými cieľmi práce je lepšie preskúmať subštrukturálne typové systémy, a tiež iné programovacie jazyky (Rust, C++ a Haskell), ktoré poskytujú nástroje umožňujúce realizáciu týchto systémov. Tieto nástroje nie sú vždy zavádzané so zámerom vytvorenie subštrukturálneho systému, ale ukzauje sa, že ich môžu realizovať. Cieľom je teda zistiť, ktoré konkrétne systémy tieto jazyky dokážu realizovať, a pomocou týchto systémov definovať rozhranie pre príležitostné slovo ako príklad prvku, ktorého použitie nám umožnia vhodne kontrolovať. Pre systémové jazyky, akými sú C++ a Rust, tiež skúmame dopad takejto kontroly na výkon programu počas behu aj na kompiláciu pri vývoji.

Kľúčové slová: bezpečné programy, návrh rozhrania, príležitostné slovo, subštrukturálne typy, lineárne typy, Rust, C++, Haskell

Abstract

In this work, we look at the type checking of compiled languages, investigating the abilities of type systems in which not all structural rules hold. Such systems are called substructural type systems and allow an interface to capture some additional requirements for working with values for additional type checking. This work continues from an article that described such an interface using the example of cryptographic nonces in the Rust programming language. The main goals of this work are to better explore substructural type systems and other programming languages (Rust, C++ and Haskell), that provide tools enabling the implementation of these systems. These tools are not always introduced with the intention of creating a substructural system, but they can implement some of them. Thus, the goal is to find out which specific systems these languages can implement and to define an interface for nonces using those systems. The security requirements of nonces make them an element whose use can be checked using substructural types. For system languages, specifically C++ and Rust, we also explore the impact of this additional checking on the runtime performance of programs, as well as the performance of compilers during development.

Keywords: secure coding, interface design, nonce, static checking, substructural types, linear types, Rust, C++, Haskell

Obsah

Úvod **1**

1 Príležitostné slovo **3**
1.1 Definícia . . . 3
1.2 Použitie . . . 3

2 Ďalšie aplikácie subštrukturálnych typových systémov **5**
2.1 Práca so súbormi . . . 5
2.2 Mutovateľné polia . . . 6
2.3 Kontrola korektnosti 3D objektov . . . 7
2.4 Kvantové výpočty . . . 8

3 Subštrukturálne typové systémy **9**
3.1 Štrukturálne pravidlá . . . 9
3.1.1 Výmena/Exchange . . . 9
3.1.2 Oslabenie/Weakening . . . 9
3.1.3 Kontrakcia/Contraction . . . 10
3.2 Typy subštrukturálnych systémov . . . 10
3.2.1 Zoradené/Ordered . . . 11
3.2.2 Lineárne . . . 11
3.2.3 Afínne . . . 11
3.2.4 Relevantné . . . 12

4 Nástroje pre subštrukturálne typy v praxi **13**
4.1 Haskell . . . 13
4.1.1 Relevantná syntax . . . 13
4.1.2 Dostupné nástroje . . . 14
4.1.3 Implementovateľné systémy . . . 15
4.2 Rust . . . 17
4.2.1 Relevantná syntax . . . 17
4.2.2 Dostupné nástroje . . . 17

4.2.3 Implementovateľné systémy 18
4.3 C++ 19
4.3.1 Relevantná syntax 19
4.3.2 Dostupné nástroje 19
4.3.3 Implementovateľné systémy 21

5 Implementácia príležitostného slova **23**
5.1 Haskell 23
5.2 Rust 25
5.3 C++ 27

6 Dopad lineárnych typov na výkon **29**
6.1 Princípy testov 29
6.2 C++ 30
6.3 Rust 53

Záver **61**

Príloha A **67**

Zoznam obrázkov

3.1 Vzťahy medzi subštrukturálymi systémami 11

6.1 Godbolt GCC -o2 32
6.2 Godbolt GCC needs_fresh -o2 33
6.3 Godbolt GCC main -o2 33
6.4 Godbolt GCC needs_fresh -o0 34
6.5 Godbolt GCC needs_fresh -o0 35
6.6 Godbolt Clang -o2 36
6.7 Godbolt Clang needs_fresh -o2 37
6.8 Godbolt Clang main -o2 37
6.9 Godbolt Clang needs_fresh -o0 38
6.10 Godbolt Clang needs_fresh -o0 39
6.11 Godbolt MSVC -o2 40
6.12 Godbolt MSVC needs_fresh -o2 41
6.13 Godbolt MSVC main -o2 41
6.14 Godbolt MSVC needs_fresh -o0 42
6.15 Godbolt MSVC needs_fresh -o0 42
6.16 G++ -o0 45
6.17 G++ -o0 46
6.18 G++ -o2 47
6.19 G++ -o2 47
6.20 Clang -o0 49
6.21 Clang -o0 49
6.22 Clang -o2 50
6.23 Clang -o2 51
6.24 MSVC /Od 52
6.25 MSVC /O2 53
6.26 Godbolt Rust -o2 54
6.27 Godbolt Rust needs_fresh -o2 55
6.28 Godbolt Rust needs_fresh -o2 55
6.29 Godbolt Rust main -o2 55

6.30 Godbolt Rust needs_fresh -o0 56
6.31 Godbolt Rust needs_fresh -o0 56
6.32 Rust -o0 58
6.33 Rust -o0 59
6.34 Rust -o2 60
6.35 Rust -o2 60

Zoznam tabuliek

6.1 G++ -o0 45
6.2 G++ -o2 46
6.3 Clang -o0 48
6.4 Clang -o2 50
6.5 MSVC /Od 51
6.6 MSVC /O2 52
6.7 Rust -o0 58
6.8 Rust -o2 59

Úvod

Statická kontrola pozostáva z lexikálnej, syntaktickej a sémantickej analýzy. Je to prvá úroveň automatickej kontroly, pri ktorej je možné detegovať chyby v programe. Skoršie odhalenie chýb umožňuje ich včasnú nápravu, čo typicky znižuje aj náklady na ich opravu. Pre kompilované jazyky je typická typová kontrola, ktorá je súčasťou sémantickej analýzy a odhaľuje chyby ako napríklad nesprávne argumenty pre funkciu a podobne.

Pre mnohé kryptografické konštrukcie platí, že ich bezpečnosť závisí aj od niektorých sémantických vlastností. Problém s nesprávnym použitím mnohých kryptografických konštrukcií je, že z funkčného hľadiska môžu programy fungovať správne. To znamená, že napríklad pri šifrovanom komunikačnom protokole je komunikácia úspešná, ale nespĺňa bezpečnostné predpoklady použitých konštrukcií, čo útočníkovi umožňuje jednoduché dešifrovanie. Takto ľahko vzniknú bezpečnostné diery, vedúce ku kompromitácii kryptografie a potenciálne celého systému.

Jedným príkladom je prvok príležitostného slova/nonce, ktorého hlavnou vlastnosťou je jednorazové použitie. V praxi sa vyskytol prípad, kde táto vlastnosť bola porušená pri algoritme pre podpisovanie dát [9]. Výsledkom bola kompromitácia súkromného kľúča použitého v PlayStation 3, kde v algoritme generovanie náhodnej hodnoty nahradili konštantou.

Zabezpečenie kontroly podmienok za behu je možné a pre tvorcu knižnice potenciálne priamočiarejšie. Stačí pri každom použití skontrolovať, či pre sú všetky prerekvizity splnené, a v opačnom prípade vyhlásiť chybu. Takáto kontrola je ale zbytočne drahá ak program je korektný, a chyba je odhalená až pri nasadení programu, nie počas prvotného vývoja.

Táto práca nadväzuje na článok [15], ktorý sa zaoberá aplikáciou typovej kontroly na zabezpečenie korektného použitia príležitostného slova, opísaný v jazyku Rust. Využíva na to koncept lineárnych typov, ktoré sú špeciálnym prípadom subštrukturálnych typových systémov. Tieto systémy umožňujú kontrolovať počet a poradie použití dát alebo operácií, vďaka čomu sú vhodné napríklad na vymedzenie určitých použití rozhraní pre prístup k systémovým zdrojom, ako napríklad pamäť, súbory a zámky. S použitím lineárnych typov článok navrhuje rozhranie pre kryptografickú knižnicu, ktorá by programátorovi neumožnila používať príležitostné slovo nesprávne bez toho, aby toto

bolo odhalené pri statickej kontrole programu. Tým bráni zámernému aj neúmyselnému nesprávnemu použitiu, ktoré môže byť inak ťažké odhaliť.

V práci sa bližšie pozrieme na nástroje dostupné v niektorých populárnych jazykoch, ktoré umožňujú využiť lineárne typy, alebo iné príbuzné systémy. Tvorcami dokumentovaná motivácia pre zavedenie týchto nástrojov do jazyka je často iná, ako zavedenie subštrukturálnych typových systémov, ale potenciálne umožňuje realizovať niektoré z nich. Naším cieľom je preskúmať, aké kombinácie vlastností, ktoré tieto typové systémy definujú, dokážeme vytvoriť.

Tiež vo viacerých jazykoch demonštrujeme ekvivalentný príklad definície a použitia príležitostného slova využitím týchto nástrojov, ich výhody a nevýhody pre takéto použitie. Súčasťou tejto demonštrácie je aj analýza dopadu na kompiláciu a beh programu. Prirodzeným cieľom je, aby cena za dodatočnú statickú kontrolu bola zanedbateľná za behu, a nebola veľká počas vývoja.

Hlavným príkladom, ktorý používame pre demonštráciu a motiváciu práce sú príležitostné slová. V úvodnej kapitole 1 sa podrobnejšie pozrieme na ich definíciu a použitia, z ktorých vyplývajú kontrolované požiadavky. Následne v kapitole 2 popíšeme ďalšie potenciálne aplikácie subštrukturálnych systémov. Prezentujeme tu aj jedno reálne implementované rozhranie, a zaujímavý vzťah s kvantovými výpočtami.

V kapitole 3 popíšeme typickú hierarchiu subštrukturálnych systémov a vlastností, ktoré ich identifikujú. Tieto vlastnosti následne využijeme na analýzu zvolených jazykov C++, Rust a Haskell. Kapitola 4 popisuje nástroje, ktoré použijeme na realizáciu subštrukturálnych systémov. Následne identifikujeme, ktorý systém alebo systémy nám jazyk umožňuje realizovať, a limity takejto realizácie.

Na základe týchto nástrojov následne prezentujeme rozhranie pre príležitostné slová v týchto jazykoch. V kapitole 5 sú popísané implementácie, vrátane niektorých detailov správania statickej kontroly pri ich kompilácii. Kapitola 6 sa zaoberá dopadom týchto implementácií na kompiláciu a beh korektného programu. Veľmi pozitívne bolo zistenie, že kompilátory vedia dobre optimalizovať subštrukturálne rozhranie, vďaka tomu je dopad na beh programu minimálny. Generátory kódu, ktoré sme vytvorili a použili pri testovaní, sú v prílohe.

Kapitola 1

Príležitostné slovo

Príležitostné slovo je kryptografický prvok, ktorý sa vyznačuje jednorazovým použitím. Typicky má formu náhodného alebo pseudonáhodného čísla alebo reťazca, ale toto nie je všeobecnou podmienkou.

1.1 Definícia

Príležitostné slovo je ľubovoľné číslo alebo reťazec, použitý práve jeden krát v kryptografickom protokole. Typicky obsahuje buď časovú pečiatku, alebo dostatočný počet náhodných bitov na zabezpečenie zanedbateľnej pravdepodobnosti, že sa niekedy vygeneruje rovnaká hodnota. Niektorí autori považujú pseudonáhodnosť, alebo nepredikovateľnosť, za nutnú vlastnosť. Pre niektoré použitia ale nie je nevyhnutná.

1.2 Použitie

Typickým príkladom použitia príležitostných slov je ochrana komunikačných protokolov proti replay útokom. Pri tomto type útoku využíva útočník zachytenú legitímnu správu, ktorú bez úprav zopakuje. Bez ochrany by takáto správa bola prijatá ako legitímna napriek tomu, že útočník nepozná heslá alebo podobné prvky použité na jej vytvorenie. Preto napríklad pri autentizačných protokoloch výzva na autentizáciu obsahuje príležitostné slovo. Prijaté slovo je následne vhodne kryptograficky použité v odpovedi, aby pre útočníka nebolo možné vymeniť ho za iné. Preposlanie takejto odpovede útočníkom bude vyhodnotené ako neplatná odpoveď, keďže odpovedá na iné príležitostné slovo.

Požiadavkou na jednorazové použitie sa tiež vyznačujú inicializačné vektory pre šifry. Používajú sa v prúdových aj blokových šifrách, kde zopakovanie inicializačného vektora vedie k opakovaniu šifrovacích bitov.

Požiadavku jednorazového použitia tiež vyžaduje súkromný kľúč Lamportovej jed-

norazovej podpisovej schémy. V tomto prípade opätovné použitie kľúča vedie k postupnej kompromitácii, lebo dizajn schémy umožňuje kombinovať podpisy vytvorené rovnakým kľúčom.

Použitie, kde nie je nutná náhodnosť ani nepredikovateľnosť, je deduplikácia správ, napríklad objednávok. V prípade objednávok sa môže stať, že zákazník nedostane potvrdenie o prijatí objednávky. Príležitostné slovo umožňuje odlíšiť zámerne poslanú ďalšiu objednávku s rovnakým obsahom od znovu poslanej objednávky kvôli chýbajúcemu potvrdeniu.

V špecifickej forme sa príležitostné slová tiež vyskytujú v proof-of-work systémoch, napríklad pre kryptomeny. Pri takomto použití je ale príležitostné slovo typicky hľadaná hodnota a požiadavka jednorazového použitia je samostatne kontrolovaná. Systém má dopredu definovanú funkciu, ktorej vstupom je príležitostné slovo, a dopredu definovaný predikát pre výstupnú hodnotu. Definovaná funkcia je typicky niektorá známa kryptografická hešovacia funkcia a predikát je typicky triviálne overiteľný. Požadovaná vlastnosť hľadaného príležitostného slova je, že výsledkom aplikácie funkcie na toto slovo je hodnota spĺňajúca predikát. Používateľ systému, ktorý chce dokázať vykonanú prácu, teda musí nájsť takéto príležitostné slovo.

Kapitola 2

Ďalšie aplikácie subštrukturálnych typových systémov

Okrem príležitostných slov sú najmä lineárne typové systémy vhodné na viacero oblastí, vrátane oblastí mimo bezpečnosti. Napríklad v jazyku Haskell je s ich pomocou implementovaný systém na správu súborov. Ako ďalšie príklady tiež popíšeme rozhranie pre prácu s mutovateľnými poliami v jazykoch s imutabilnými typmi a kontrolu uzavretosti 3D objektov. Popíšeme tiež vzťah lineárnych typov s kvantovými výpočtami.

2.1 Práca so súbormi

Pri práci so súbormi medzi typické problémy patrí garantovanie uzatvorenia otvoreného súboru, a zabránenie použitia už uzatvoreného súboru. Obe podmienky sa dajú zachytiť pomocou lineárneho typového systému.

Takéto rozhranie implementuje knižnica linear-base pre jazyk Haskell, na príklade ktorej demonštrujeme myšlienku. Kľúčové je použitie lineárnych typov, ktoré vynucujú práve jedno použitie, čo umožňuje vynútiť aj uzatvorenie súboru. Otvorený súbor je reprezentovaný pomocou handle. Handle je návratová hodnota pri otváraní súboru, a argumentom pre zatvorenie súboru. Všetky ostatné operácie vyžadujú handle ako argument, a súčasťou návratovej hodnoty je nový handle. Vďaka lineárnemu obmedzeniu teda na jeden otvorený súbor vždy existuje práve 1 handle, ktorý musí byť použitý, buď na uzatvorenie súboru, alebo na vykonanie operácie. Ak program súbor pomocou handle uzavrie, tak ho práve raz použil a už ho nemôže použiť ďalej, vďaka čomu nedokáže použiť už uzatvorený súbor. Ak vykoná nejakú operáciu, napríklad zápis do súboru, pôvodný handle práve raz použil a už ho nemôže použiť, dostane ale nový handle, ktorý znovu musí práve raz použiť. Kvôli tomu nemôže skončiť bez uzavretia súboru, inak mu zostane nepoužitý handle.

Detailne popísaný príklad použitia reálneho rozhrania, ktoré toto implementuje v jazyku Haskell, je v sekcii 4.1.2

2.2 Mutovateľné polia

Haskell je jedným z jazykov, v ktorých sú dátové štruktúry imutabilné. Toto umožňuje rôzne optimalizácie pri kompilácii, ale spôsobuje to aj problémy. Efektívna implementácia niektorých algoritmov vyžaduje mutovateľné dátové štruktúry, ktorých najjednoduchší príklad sú polia. Navyše výsledkom časti algoritmu, ktorá takéto pole používa môže byť pole výsledkov, typicky jedno z polí použitých pri výpočte. V každom prípade treba zabezpečiť korektné ukončenie práce s mutovateľným poľom, konverziou na imutabilné alebo uvoľnením. Syntax tu použitá je vysvetlená v sekcii 4.1.1.

Tento koncept bol prezentovaný na konferencii [11] ešte skôr, ako boli lineárne typy zavedené v hlavnej verzii jazyka Haskell. Prezentácia bola zameraná na dôvod zavedenia a myšlienky implementácie, prezentované na porovnaní medzi štandardným a lineárnym rozhraním pre mutovateľné polia. Pre účely demonštrácie je úlohou konverzia zoznamu dvojíc (index, prvok) bez garancie na poradie indexov na štandardný imutabilný zoznam prvkov na daných indexoch.

Potrebné operácie pre takéto rozhranie sú vytvorenie poľa, čítanie a zápis prvku a konverzia poľa na imutabilné (operácia freeze), keď chce používateľ ukončiť prácu s poľom. Tiež je vhodné mať nejakú operáciu typu forM alebo fold na preiterovanie poľa a aplikáciu operácie.

Kód 2.1: Jednoduché rozhranie

```
1  newMArray     :: Int -> ST s (MArray s a)
2  read          :: MArray s a -> Int -> ST s a
3  write         :: MArray s a -> (Int, a) -> ST s ()
4  unsafeFreeze  :: MArray s a -> ST s (Array a)
5  forM          :: (Monad m) => [a] -> (a -> m ()) -> m ()
6  runST         :: (forall a. ST s a) -> a
7
8  array :: Int -> [(Int, a)] -> Array a
9  array size pairs = runST $ do
10     ma <- newMArray size
11     forM pairs (write ma)
12     return $ unsafeFreeze ma
```

V štandardnom prípade sa práca s mutovateľnými poliami odohráva v špeciálnej monáde, podobne ako práca so súbormi. Keďže práca s poľom, ako napríklad zápis, sa

odohráva v monáde, použijeme tiež operáciu forM, ktorá umožňuje ako jeden príkaz zapísať zoznam prvkov. Operácia runST zabezpečuje vykonanie monády.

Jedným z problémov je, že jazyk nedokáže zabrániť tomu, aby používateľ menil mutovateľné pole po tom, ako prácu s ním ukončil operáciou *unsafeFreeze*. To by mohlo spôsobiť segmentačné chyby a iné nedefinované správanie, ktorému sa Haskell štandardne vyhýba.

Kód 2.2: Lineárne rozhranie

```
1  newMArray :: Int -> (MArray a %1-> Ur b) %1-> b
2  read      :: MArray a %1-> Int -> (MArray a, Ur a)
3  write     :: MArray a %1-> (Int, a) -> MArray a
4  freeze    :: MArray a %1 -> Ur (Array a)
5  foldl     :: (a %1-> b %1-> a) -> a %1-> [b] %1-> a
6
7  array :: Int -> [(Int, a)] -> Array a
8  array size pairs = newMArray size $ \ma ->
9      freeze (foldl write ma pairs)
```

Lineárne rozhranie používa mierne odlišný prístup. Namiesto monády je mutovateľné pole sprístupnené funkcii, od ktorej je požadovaná linearita. Navyše na výstupe požaduje typ *Unrestricted*, čím neumožňuje vrátiť mutabilné pole samotné v ľubovoľnej forme. Výsledok je práve raz použitý keď je rozbalený na výstup, čím získame práve 1 použitie poľa, propagované cez celú funkciu, ktorej bolo sprístupnené. Správne obmedzenie ďalších operácií vyžaduje trochu kreativity, keďže cieľom je mať lineárne obmedzené práve pole samotné, vo všeobecnom prípade prvky poľa nie.

Súčasťou návratovej hodnoty čítania a zápisu musí byť pole, podobne ako v prípade práce so súbormi je súčasťou hodnoty handle. Pre čítanie ale nechceme obmedzovať prečítaný prvok, preto je navyše obalený v type *Unrestricted*. Podobne pre zápis je pole lineárny argument, ale index a prvok sú obyčajným argumentom. Pre prejdenie poľom nám stačí operácia foldl, ktorú treba správne lineárny otypovať.

2.3 Kontrola korektnosti 3D objektov

V grafických a inžinierskych aplikáciách 3D objekty typicky reprezentujú uzavreté telesá. Pre 3D tlač sa tiež označujú ako vodotesné. Často je vhodné mať už v programe preddefinované niektoré základné modely. V tomto prípade je výhodná možnosť kontroly týchto modelov už v čase kompilácie. Táto idea je podrobnejšie opísaná v blogu [13], z ktorého je to vysvetlená základná myšlienka.

Pomocou lineárnych typov sa dá napríklad definovať rozhranie pre definíciu objektov tak, že kontroluje používanie hrán. Každá stena je definovaná postupnosťou

hrán, a hrana patrí práve dvom stenám. Keďže lineárne typy umožňujú kontrolovať práve jedno použitie, rozhranie pre vytvorenie hrany poskytne hranu dvakrát, akoby dva pohľady, z ktorých sa každý použije práve raz. Toto umožňuje odhalenie objektov s chýbajúcimi stenami, priveľa stenami na hrane (napríklad vo forme stien vnútri objektu), a podobných problémov.

Tento prístup ale neodhalí všetky typy defektov, hlavne také, ktoré závisia od konkrétnych vzťahov medzi rozmermi a pozíciami objektov. Presnejšia kontrola by sa dala dosiahnuť použitím transformácií, ktoré garantujú zachovanie uzavretosti.

2.4 Kvantové výpočty

Kvantové výpočty majú prirodzený vzťah s lineárnymi systémami. Jedným zo základných pravidiel v oblasti kvantových výpočtov je veta o zákaze klonovania [17], ktorá vyjadruje nemožnosť vytvorenia kópie všeobecného kvantového stavu bez jeho poškodenia. Toto je priama paralela pravidla o kontrakcii, popísaného v sekcii 3.1.3, resp. jeho neplatnosti v lineárnom systéme.

Podobne veta o nemožnosti zmazania všeobecného stavu [14] je paralelou pravidla o oslabení opísaného v sekcii 3.1.2. V kvantovom výpočte je síce možné qbit nepoužiť, ale keďže nie je možné ho všeobecne zmazať, konečný stav systému bude rôzny od výpočtu, ktorý ho použije.

Kapitola 3

Subštrukturálne typové systémy

Subštrukturálne typové systémy sú rodinou typových systémov, v ktorých chýba alebo je obmedzené použitie niektorých štrukturálnych pravidiel [16, 15]. Môžu byť vhodné napríklad na kontrolu prístupu k systémovým zdrojom, kde môžu zabrániť vzniku problémových stavov.

3.1 Štrukturálne pravidlá

V subštrukturálnych systémoch uvažujeme 3 štrukturálne pravidlá: výmenu, oslabenie a kontrakciu. Všetky majú formu odvodzovacieho pravidla vo formálnej logike, kde z možnosti úspešnej typovej kontroly termu odvodzujú možnosť úspešnej typovej kontroly príbuzného termu. Kontext pre kontrolovateľnosť termu je postupnosť premenných ako predpokladov.

Vyjadrujú vlastnosti štruktúrovaných typových systémov, ktoré sú prítomné vo väčšine programovacích jazykov s typovou kontrolou.

3.1.1 Výmena/Exchange

$$\frac{\Gamma_1, x : \tau_x, y : \tau_y, \Gamma_2 \vdash e : \tau}{\Gamma_1, y : \tau_y, x : \tau_x, \Gamma_2 \vdash e : \tau} \tag{3.1}$$

Pravidlo výmeny vyjadruje, že na typovú kontrolu nemá vplyv poradie premenných v kontexte. Tento fakt je vyjadrený na základe výmeny susedných premenných. Priamym dôsledkom je, že na permutácií všetkých premenných v kontexte nezáleží. Ostatné pravidlá sú formulované tak, že zmeny kontextu sú umožnené len na konci, a použitím pravidla výmeny alebo jeho dôsledku sa dajú aplikovať kdekoľvek.

3.1.2 Oslabenie/Weakening

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau_x \vdash e : \tau} \tag{3.2}$$

Pravidlo oslabenia vyjadruje možnosť pridať nové premenné do kontextu, pričom sa zachová kontrolovateľnosť. Zjavne pôvodný term nemôže používať takto pridané premenné, sú teda nadbytočné. Pre potreby našej analýzy môžeme definovať príbuzné pravidlo, ktorého implementácia v jazyku umožňuje simulovať systém s pravidlom oslabenia.

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau_x \vdash dump(x); e : \tau} \tag{3.3}$$

Používame na to operáciu *dump*, ktorej jedinou úlohou je pre typový systém reprezentovať použitie premennej, aby formálne nebolo potrebné štandardné pravidlo oslabenia.

3.1.3 Kontrakcia/Contraction

$$\frac{\Gamma, x_2 : \tau_x, x_3 : \tau_x \vdash e : \tau}{\Gamma, x_1 : \tau_x \vdash [x_2 \mapsto x_1, x_3 \mapsto x_1]e : \tau} \tag{3.4}$$

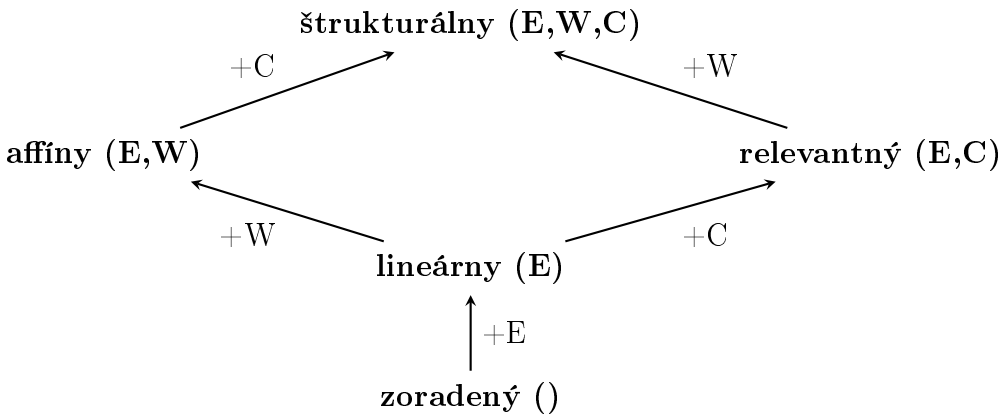
Pravidlo kontrakcie umožňuje odstrániť nadbytočnú, respektíve zdvojenú premennú, kde do termu za obe pôvodné premenné substituuje novú premennú s rovnakou hodnotou, ako mali obe pôvodné. Podobne ako pri oslabení môžeme definovať príbuzné pravidlo, ktoré umožní simulovať systém s pravidlom kontrakcie.

$$\frac{\Gamma, x_2 : \tau_x, x_3 : \tau_x \vdash e : \tau}{\Gamma, x_1 : \tau_x \vdash (x_2, x_3) := duplicate(x_1); e : \tau} \tag{3.5}$$

Používame operáciu *duplicate*, ktorej implementácia musí vhodne vytvoriť kópiu hodnoty.

3.2 Typy subštrukturálnych systémov

Základné typy subštrukturálnych systémov dostaneme výberom podmnožín z množiny opísaných pravidiel [16]. Zvolená podmnožina vyjadruje, ktoré pravidlá platia vo všeobecnom prípade, teda vynechané pravidlá môžu platiť v niektorých prípadoch, ale nie všeobecne. V prípade, že platia všetky 3 pravidlá, typový systém je bežný štruktúrovaný, tiež nazývaný neobmedzený. Typicky užitočné subštrukturálne systémy sú zakreslené v nasledujúcom grafe. E, W, C označujú pravidlá výmeny, oslabenia a kontrakcie. Z grafu tiež vidieť, že ak v systéme niektoré pravidlá platia, tak aspoň pravidlo výmeny platí. Bez pravidla výmeny sú aj ostatné pravidlá značne obmedzené (aspoň bez zmeny formulácie), preto vynechané systémy nie sú považované za veľmi užitočné.



Obr. 3.1: Vzťahy medzi subštrukturálymi systémami

3.2.1 Zoradené/Ordered

V zoradených systémoch neplatí žiadna zo štrukturálnych vlastností. Preto vo všeobecnom prípade každá z premenných musí byť použitá práve raz, a to vo fixnom poradí, v kontexte kde sú definované od poslednej pridanej, ktorá ešte nebola použitá. Modeluje teda správanie zásobníka s premennými, navyše všeobecne nie je možné vrch zásobníka odstrániť bez použitia, ani použiť ho bez odstránenia. V konkrétnom prípade je možné, že sa dá poradie niektorých premenných vymeniť, toto ale tiež nemôže platiť všeobecne, inak by platilo pravidlo výmeny.

3.2.2 Lineárne

V lineárnych systémoch platí len pravidlo výmeny. Vo všeobecnom prípade každá z premenných musí byť použitá práve raz, ale v ľubovoľnom poradí. Toto je veľmi užitočná podmienka pri práci so zdrojmi, keďže umožňuje obmedziť používanie objektov a zjednodušiť tým identifikáciu posledného použitia, zmeny stavov a podobne. Zároveň nijako nebráni ľubovoľnému prelínaniu pri použití viacerých zdrojov, vďaka všeobecnému pravidlu výmeny.

Poskytujú tiež dobrú abstrakciu pre kvantové výpočty ako je opísané v sekcii 2.4, kde je dokázateľne nemožné zduplikovať všeobecný stav a naopak nie je možné všeobecný stav zničiť, čo vytvára veľmi podobné podmienky práve jedného použitia.

3.2.3 Afínne

V afínnych systémoch platí pravidlo výmeny aj pravidlo oslabenia. Každá premenná preto môže byť použitá najviac raz, podobne ako v lineárnych systémoch, ale oslabenie pridáva možnosť premennú nepoužiť. Pre niektoré použitia je táto podmienka postačujúca, a umožňuje väčšiu voľnosť. Napríklad systém nemusí vyžadovať expli-

citné ukončenie práce so zdrojom, a ukončí ho sám keď jediný odkaz naň odstráni z kontextu napríklad pri opustení lexikálnej úrovne.

Potenciálne sa takýto systém dá simulovať v upravenej forme pomocou lineárneho s dodatočnou operáciou zahodenia premennej, ako sme popísali v sekcii 3.1.2. Takto upravený systém stále vyžaduje napríklad explicitné ukončenie práce so zdrojom, dodatočná operácia reprezentuje všeobecné ukončenie na rozdiel od špecifického pre typ zdroja.

3.2.4 Relevantné

V relevantných systémoch platí pravidlo výmeny aj pravidlo kontrakcie. Každá premenná teda musí byť použitá aspoň raz, podobne ako v lineárnych systémoch, pričom kontrakcia pridáva možnosť premennú použiť dva krát. Ako priamy dôsledok tak umožňuje ľubovoľne veľa opakovaných použití premennej.

Podobne ako pri afínnych systémoch sa v upravenej forme dá simulovať lineárnym systémom. Simulácia vyžaduje operáciu zdvojenia ako je opísaná v sekcii 3.1.3.

Kapitola 4

Nástroje pre subštrukturálne typy v praxi

Niektoré programovacie jazyky obsahujú špeciálnu funkcionalitu, ktorá umožňuje implementovať subštrukturálne systémy. Presnejšie povedané, poskytovaná funkcionalita spôsobuje, že ich typový systém nie je štruktúrovaný. V prípade jazyka Haskell je táto funkcionalita priamo nazývaná lineárne typy respektíve lineárne funkcie. Pre jazyky Rust a C++ je zdokumentovaná motivácia efektívna správa pamäte, vo forme konceptu vlastníctva v jazyku Rust a sémantiky presúvania v jazyku C++. Pre každý z týchto jazykov popíšeme správanie príslušnej funkcionality a ukážeme, ktorý subštrukturálny typový systém realizuje.

4.1 Haskell

4.1.1 Relevantná syntax

Definícia funkcie v tejto práci bude vždy pozostávať z dvoch častí, explicitne definujeme typ a následne definujeme implementáciu. Príklad definície:

Kód 4.1: Jednoduchá funkcia

```
1 example :: a -> b %1 -> (a,b)
2 example x y = (x,y)
```

Prvý identifikátor je meno definovanej funkcie. Musí byť rovnaké v obidvoch riadkoch, aby definícia typu a implementácie bola pre tú istú funkciu.

V prvom riadku za :: je explicitne vypísaný typ funkcie. Dve hodnoty v zátvorke označujú usporiadanú dvojicu. Šípka $->$ označuje, že ide o funkciu, v tomto prípade teda o funkciu s prvým argumentom typu a a druhým argumentom typu b, pričom výsledok je typu dvojica s prvkami typu a a b. Prítomnosť %1 pred šípkou indikuje, že pre daný argument platí lineárne obmedzenie, vysvetlené v sekcii 4.1.2.

Druhý riadok definuje správanie funkcie. Pre konkrétne argumenty x a y vytvorí dvojicu v rovnakom poradí. Správanie sa tiež dá definovať použitím anonymnej/lambda funkcie, ekvivalent v tomto tvare je

Kód 4.2: Jednoduchá funkcia 2

```
1 example = \x y -> (x,y)
```

Spätné lomítko značí začiatok anonymnej funkcie, nasledujú premenné, a za šípkou výsledok. V tomto prípade = túto anonymnú funkciu priradí menu funkcie.

V príkladoch v tejto kapitole je ešte použitý typ (), ktorý reprezentuje usporiadanú n-ticu pre n rovné 0, tento typ má jedinú hodnotu (). Ďalej používame typ Ur, ktorého názov je skratka pre unrestricted/neobmedzený, tento typ umožňuje vyňať hodnotu z lineárneho obmedzenia. Zo štandardnej knižnice tiež používame funkciu *const*, ktorá má jediný argument a vracia konštantnú funkciu, ktorá ignoruje vlastný argument a vždy vráti argument *const*, s ktorým bola vytvorená. Napríklad pre volanie *const* () je teda výsledkom funkcia s jedným argumentom, ktorej volanie vždy vráti hodnotu ().

4.1.2 Dostupné nástroje

V jazyku Haskell sú od verzie 9.0.1 implementované takzvané lineárne typy, ktoré rozdeľujú funkcie na všeobecné a lineárne [6]. Ich použitie vyžaduje prepínač XLinearTypes. Funkcia môže byť lineárna od konkrétneho argumentu, pričom podmienka linearity je formálne definovaná ako „Ak výsledok volania funkcie je použitý práve 1 krát, tak daný argument bude použitý práve 1 krát.“ Ak túto podmienku spĺňa, môže byť typovo deklarovaná ako lineárna, ale môže byť označená aj ako všeobecná. Priamym dôsledkom podmienky je rovnaký počet použití argumentu ako počet použití výsledku aj pre viac ako 1 použitie.

Menej formálne táto podmienka vyžaduje, aby premenná bola v tele funkcie spomenutá práve 1 krát v lineárnom kontexte, teda do výstupu funkcie vstupuje len cez lineárnu funkciu (alebo viacero zložených lineárnych funkcií). To tiež prakticky znamená, že lineárne funkcie môžu volať len iné lineárne funkcie, pre všeobecné funkcie podobné obmedzenie neplatí.

Spolu s implementáciou lineárnych funkcií bola tiež vytvorená verzia štandardnej knižnice s vhodne lineárne otypovanými funkciami, a tiež špeciálnou Resource IO monádou na nahradenie štandardnej IO monády. Nová monáda je použitá na zabezpečenie korektnej práce so súbormi, teda aby otvorený súbor bol na konci práce uzatvorený a aby sa z uzatvoreným súborom už nepracovalo. Na toto využíva práve požiadavku linearity funkcií, ktorá vďaka otypovaniu rozhrania stráži použitie súboru.

Toto je najlepšie vysvetliteľné na príklade, ktorý je uvedený v dokumentácii [5]:

Kód 4.3: Resource IO monáda

```
1 linearWriteToFile :: IO ()
2 linearWriteToFile = Linear.run $ Control.do
3     handle1 <- Linear.openFile "/home/user/test.txt" Linear.
        WriteMode
4     handle2 <- Linear.hPutStrLn handle1 (Text.pack "hello␣
        there")
5     () <- Linear.hClose handle2
6     Control.return (Ur ())
```

V prvom riadku definície sú použité 2 funkcie z lineárnej knižnice, *Linear.run* a *Control.do*, ktoré umožňujú samotnú prácu so súborom písať imperatívnym štýlom, v lineárnej IO monáde. Operácia *Linear.openFile* otvorí súbor, v tomto prípade na zápis, pričom dá k dispozícii prvý handle, ktorý pomocou $<-$ priradíme do premennej. Handle následne použijeme na zápis do súboru pomocou *Linear.hPutStrLn*, toto je práve jedno použitie prvého handle, preto dostaneme nový handle, reprezentujúci súbor po zapísaní reťazca. Následne práve raz použijeme nový handle na uzatvorenie súboru. Nakoniec použijeme funkciu *Control.return* na korektné ukončenie práce v monáde a návrat hodnoty z funkcie.

4.1.3 Implementovateľné systémy

Z definície lineárnej funkcie je pomerne zjavné, že výmena poradia premenných je vždy dovolená, keďže podmienka linearity hovorí o práve jednej premennej. To znamená, že pravidlo výmeny platí pre lineárne argumenty, podobne pre všeobecné platí možnosť výmeny štandardne. Dá sa tiež zameniť poradie lineárneho a všeobecného argumentu, triviálne všeobecnými funkciami:

Kód 4.4: Konverzia lineárnych funkcií

```
1 swapLinear1 :: (a %1 -> b -> c) %1 -> (b -> a %1 -> c)
2 swapLinear1 f = \b a -> f a b
3 swapLinear2 :: (a -> b %1 -> c) %1 -> (b %1 -> a -> c)
4 swapLinear2 f = \b a -> f a b
```

Obidve sú typovo korektné podľa kompilátora. Samozrejme, táto vlastnosť je odvoditeľná priamo z definície. Pre *swapLinear*1, vstupná funkcia musí spĺňať podmienku linearity od a, teda po dosadení za prvý argument musí vráti funkciu, ktorá ak bude práve raz použitá, dosadený argument bude použitý práve raz. Túto mení na funkciu, ktorá po dosadení argumentu vracia funkciu, do ktorej keď dosadíme za a a výsledok použijeme práve raz, dosadený argument bude použitý práve raz. Toto sú zjavne ekvivalentné podmienky, pre *swapLinear*2 je situácia analogická.

Pravidlo výmeny teda bude vždy platiť, keďže jazyk nám nedáva možnosť rozlišovať poradie argumentov.

Pravidlo oslabenia vyjadruje možnosť nepoužiť premenné. Toto je zjavne v rozpore s podmienkou linearity, a pri jej korektnej implementácii sa nesmie dať obísť normálnym kódom. V jazyku Haskell však existuje funkcia *coerce*, ktorá umožňuje ľubovoľne meniť typ, a obísť tak typový systém. Je určená pre prípady, v ktorých typový systém nedokáže sám odvodiť potrebný typ [4]. Napríklad v našej demonštrácii príležitostného slova ju používame, aby sme kompilátor presvedčili (respektíve autoritatívne povedali), že funkcia, ktorá príležitostné slovo vypíše, je lineárna. Skutočne ak práve raz vyhodnotíme výstupnú IO monádu, bude práve raz použité vstupné príležitostné slovo, ale kvôli použitiu IO monády potrebujeme použiť *coerce*.

Táto funkcia sa však dá použiť aj v prípade, keď systému chceme klamať, napríklad na vytvorenie možnosti univerzálneho zahadzodenia premennej. Stačí na lineárnu funkciu pretypovať hodnotu *const*(). Týmto nedostaneme štandardné pravidlo oslabenia, ale dostaneme modifikovanú verziu ako bola popísaná v sekcii 3.1.2. Alternatívne môžeme namiesto *const* () použiť *const* (*Ur* ()), aby výsledná hodnota bola vyňatá z lineárneho obmedzenia.

Kód 4.5: Zahodenie premennej cez coerce

```
1 dumpVariable :: a %1 -> Ur ()
2 dumpVariable = Unsafe.coerce (const (Ur ()))
```

Samozrejme, klamanie typovému systému môže mať neželané následky, hlavne v prítomnosti optimalizácií. Umožňuje napríklad zahodiť premennú s otvoreným súborom namiesto korektného uzavretia, a podobné nekorektné programy. Hlavne v prítomnosti optimalizácií môže takéto porušovanie predpokladov typového systému spôsobiť napríklad to, že súbor bude uzatvorený až pri ukončení programu. Ak zahadzovaciu funkciu definujeme len pre typy, pri ktorých takéto problémy nehrozia (nepoužitie príležitostného slova je v poriadku), nedostaneme ani modifikovanú verziu pravidla oslabenia, keďže nie je použiteľná všeobecne.

Teda za cenu potenciálnych problémov dokážeme simulovať systémy s upraveným pravidlom oslabenia, ako sme definovali v sekcii 3.1.2.

Podobne pre pravidlo kontrakcie musí byť nemožné premennú zdvojiť, alebo bez *coerce* definovať zdvojovanie pre modifikovanú verziu pravidla, ako sme definovali v sekcii 3.1.3. Pomocou *coerce* sa takéto zdvojenie definovať dá, ale ako pri zahadzovaní to môže priniesť výrazné problémy kvôli porušovaniu predpokladov. Napríklad pre otvorený súbor premenná reprezentuje aktuálny stav, ktorý sa zmení pri zápise. Ak sme premennú zdvojili, druhá kópia potom reprezentuje už neexistujúci stav a jej použitie bude s veľkou pravdepodobnosťou problematické. Pokus o zápis cez druhú kópiu po uzatvorení prvej s istotou spôsobí neželané správanie.

Kód 4.6: Zdvojenie premennej cez coerce

```
1 dupe :: a -> (a,a)
2 dupe a = (a,a)
3 duplicateVariable :: a %1 -> (a,a)
4 duplicateVariable = Unsafe.coerce dupe
```

Jazyk Haskell teda za normálnych okolností realizuje práve lineárny typový systém, ale zneužitím špeciálnej funkcie sa dá upraviť na ľubovoľný iný systém s modifikovanými pravidlami, za cenu potenciálnych problémov pri použití štandardných knižníc. Ak nepoužijeme lineárne obmedzenie argumentov (respektíve prepínač, ktorý ich povoľuje), realizuje bežný štruktúrovaný systém.

4.2 Rust

4.2.1 Relevantná syntax

Priradenie do premennej sa robí pomocou štandardného operátora =. Názov premennej môže začínať podtržníkom, toto indikuje, že premenná je zámerne nepoužitá v ďalšom kóde.

Zo štandardnej knižnice spomíname typy $Box < T >$ a $Rc < T >$. Oba sú variantou smerníkov, kde $< T >$ reprezentuje typ hodnoty, na ktorý ukazujú. Nepriamo tiež spomíname vlastnosť *Copy*, ktorá indikuje možnosť automatickej výroby kópií hodnôt typu, ktorý ju má.

4.2.2 Dostupné nástroje

V jazyku Rust je implementovaný systém vlastníctva. Ľubovoľnú hodnotu v každom okamihu vlastní práve jedna premenná, hodnotu môže požičať alebo odovzdať novému vlastníkovi. Motivácia pre tento systém je riešenie správy pamäte počas kompilácie. Keďže hodnota má práve 1 vlastníka, pamäť treba uvoľniť práve vtedy, keď tento vlastník opúšťa rámec, v ktorom je definovaný.

Pre prípady, keď je naozaj nutné zdieľať vlastníctvo, Rust implementuje aj explicitné typy, ktoré to umožňujú (napríklad $Rc < T >$). Hodnota má stále jedného reálneho vlastníka, ktorý prakticky vytvára a počíta smerníky, a keď počet klesne na 0 uvoľní pamäť. Vlastník teda neduplikuje hodnotu, iba umožňuje zdieľané vlastníctvo, pričom je z typu premennej viditeľné, že je zdieľaná.

Na duplikáciu hodnoty je nutné buď použiť na to určenú funkciu, špecifickú pre typ hodnoty, alebo typ musí byť označený ako automaticky kopírovateľný. V obidvoch prípadoch je teda rozhodujúcim faktorom typ, ak kopírovanie neumožní, tak nebude možné.

4.2.3 Implementovateľné systémy

Systém vlastníctva nijako neovplyvňuje poradie premenných ako argumentov do funkcie alebo iného bloku. Môže ovplyvňovať povolené poradie použití, keďže po odovzdaní vlastníctva už nemôže použiť premennú ani na požičanie, kým v opačnom poradí smie. Toto však nie je výmena tak, ako ju opisuje pravidlo výmeny. Ľubovoľný realizovaný systém teda bude vždy obsahovať pravidlo výmeny.

Pravidlo oslabenia za štandardných podmienok platí, keďže vlastníctvo hodnoty nás nenúti ju použiť. Platnosť tohto pravidla sa dá zakázať, ak pri kompilácii máme aktívne varovanie pre nepoužité premenné, a varovania sú považované za chyby. Aj v takýchto prípadoch je však triviálne jednoduché obísť potrebu použitia, stačí vlastníctvo odovzdať zámerne nepoužitej premennej, čo vedie k modifikovanej verzii pravidla, ako sme definovali v sekcii 3.1.2.

Kód 4.7: Zahodenie premennej

```
1 _dump = x
```

Ak výrazy tohto typu (alebo iné s rovnakým efektom) dôsledne zakážeme používať, aktívne varovanie nepoužitej premennej stále nebude mať želaný efekt, keďže nevynucuje použitie premennej v každej vetve výpočtu. Kvôli tomu neumožňuje efektívne simulovať systémy s garantovaným použitím premennej.

Pravidlo kontrakcie je pre jazyk Rust najzaujímavejšie. Podľa predpokladu pravidla potrebujeme 2 premenné s rovnakou hodnotou. Kvôli princípu vlastníctva toto nie je možné, ak majú tú istú hodnotu, teda hodnoty sú nutne iba rovnaké, nie totožné. Potom ale vo všeobecnom prípade neplatí pravidlo kontrakcie, keďže funkcia môže vyžadovať vlastníctvo oboch hodnôt, napríklad ak obidve odovzdáva.

Modifikované pravidlo kontrakcie, ako sme definovali v sekcii 3.1.2, tiež neplatí všeobecne, keďže typ nemusí poskytovať možnosť výroby kópie, teda sa nedá vyrobiť univerzálny duplikátor. Kvôli korektnosti správy vlastníctva niektoré typy zo štandardnej knižnice, napríklad $Box < T >$, nie sú (a nesmú byť) kopírovateľné.

Z toho vyplýva, že jazyk Rust štandardne používa afínny typový systém, nie štruktúrovaný. Snaha o simuláciu lineárneho systému je neúspešná, keďže nie je možné korektne zakázať pravidlo oslabenia. Keďže navyše pravidlo kontrakcie platiť nesmie, relevantný typový systém sa pomocou jazyka Rust realizovať nedá.

4.3 C++

4.3.1 Relevantná syntax

V C++ sa mnohé operácie prepisujú na volanie špeciálnych funkcií. Napríklad + sa realizuje pomocou funkcie *operator+*. Pri definícii funkcie je potrebné použiť plný názov, v ostatnom kóde budeme používať krátku formu.

Štandardná premenná sa značí svojím typom, nasledovaným identifikátorom. Rvalue referencia je špeciálny typ premennej, ktorý za typom má napísanú dvojitú referenciu, teda &&.

Typ *void* je špeciálny typ bez hodnôt. Smerník na typ *void* sa používa ako univerzálny typ smerníka, pre odovzdávanie objektov ľubovoľného typu. Typ *auto* reprezentuje ľubovoľný konkrétny typ, podľa potreby odvodený pri kompilácii výrazu. Typová konverzia sa píše ako nový typ v zátvorkách, nasledovaný konvertovanou hodnotou.

Zo štandardnej knižnice spomíname typ $std :: unique\_ptr < T >$, ktorý je smerník na prvok typu T. Spomíname tiež funkciu $std :: move$.

Template je konštrukcia podobná makru, ale s pokročilejšou substitúciou. Umožňuje tiež explicitne obmedziť dosaditeľné hodnoty, napríklad pomocou konceptov. Koncepty môžu definovať rozhranie a vlastnosti, ktoré musí dosadený typ spĺňať.

4.3.2 Dostupné nástroje

V C++ sú od verzie C++11 implementované takzvané rvalue referencie a sémantika presúvania, ktorá ich používa. Sémantika presúvania umožňuje implementovať špecifické správanie funkcií v prípade, keď argument je dočasná hodnota. Dočasné hodnoty sú typicky medzivýsledky výrazu, ktoré nie sú dostupné po ukončení jeho výpočtu. Rvalue referencia je premenná pre dočasnú hodnotu, typicky ako argument funkcie.

Napríklad ak $s1$ a $s2$ sú implementáciou reťazcov so smerníkom na pole znakov, a operátor + reprezentuje konkatenáciu, tak pri výpočte

Kód 4.8: Konkatenácia reťazcov

```
1 auto s = s1 + s2
```

najprv vznikne dočasný objekt, reprezentujúci konkatenáciu $s1$ a $s2$, následne sa použije operátor priradenia, ktorý jeho obsah (pole znakov) skopíruje do nového objektu, a dočasný objekt sa dealokuje. Keďže dočasný objekt nie je mimo výrazu dostupný, dá sa pomocou rvalue referencií preťažiť priradenie tak, že namiesto kopírovania obsahu si ho privlastní (skopíruje len smerník na pole), a ušetrí tým nadbytočnú prácu, potrebnú na výrobu kópie.

Do štandardnej knižnice tiež pribudla funkcia $std :: move$, ktorá zo štandardného objektu vytvorí dočasný pre potreby typového systému. Objektu, ktorý bol jej ar-

gumentom, sa hovorí presunutý, respektíve z premennej, ktorej bol priradený, bolo presunuté. Takáto premenná musí byť vo valídnom stave pre dealokáciu, ale vo všeobecnom prípade ju pred ďalším použitím treba naplniť novým objektom. Štandard garantuje len to, že dealokáciou obsahu premennej nevzniknú problémy. Ľubovoľné iné vlastnosti, ktoré neplatia v každom stave objektu, nemusia platiť. Napríklad vektor, z ktorého bolo presunuté, môže byť prázdny alebo môže obsahovať nedefinované dáta.

Nástroje na statickú analýzu poskytujú možnosť odhaľovať prípady, kde sa ďalej používa premenná, z ktorej bolo presunuté. Napríklad nástroj clang-tidy toto poskytuje ako varovanie bugprone-use-after-move [2].

Od verzie C++20 je poskytnutý systém konceptov [1], ktoré navyše umožňujú priamo definovať rozhranie pre nekopírovateľný typ, tu vložený ako kód 4.9. V pôvodnej demonštrácii [12] je určený pre lineárne typy, aj keď sám nevynucuje použitie, a teda patrí skôr medzi afínne.

Kód 4.9: Koncept pre lineárny typ

```
1  template <typename T, typename U>
2  constexpr bool linear_usable_as =
3  std::is_nothrow_constructible_v<T, U> and
4  std::is_nothrow_assignable_v<T&, U> and
5  std::is_nothrow_convertible_v<U, T>;
6
7  template <typename T, typename U>
8  constexpr bool linear_unusable_as =
9  not std::is_constructible_v<T, U> and
10 not std::is_assignable_v<T&, U> and
11 not std::is_convertible_v<U, T>;
12
13 template <typename T>
14 concept Linear =
15 std::is_nothrow_destructible_v<T> and
16 linear_usable_as<T, T> and
17 linear_usable_as<T, T&&> and
18 linear_unusable_as<T, T&> and
19 linear_unusable_as<T, const T&> and
20 linear_unusable_as<T, const T>;
```

Princíp tohto konceptu je relatívne jednoduchý, pomocou vstavaných predikátov pokrýva predvolené konštruktory a špeciálne funkcie tak, aby zakázal výrobu kópie, a prikázal presúvanie. Tento koncept sa dá použiť napríklad namiesto typu *auto* v prípadoch, keď chceme navyše kontrolovať požiadavku na nekopírovateľný objekt.

4.3.3 Implementovateľné systémy

Štandardný jazyk ani rvalue referencie neovplyvňujú poradie premenných ako argumentov do funkcie alebo iného bloku. Pri aktivovanej kontrole použitia po presunutí môžu ovplyvňovať povolené poradie použití, keďže po presunutí z premennej už nemôže byť použitá bez opätovnej inicializácie. Toto však nie je výmena tak, ako ju opisuje pravidlo výmeny. Ľubovoľný realizovaný systém teda bude vždy obsahovať pravidlo výmeny.

Pravidlo oslabenia za štandardných podmienok platí, keďže existencia premennej nás nenúti ju použiť. Platnosť tohto pravidla sa dá zakázať, ak pri kompilácii máme aktívne varovanie pre nepoužité premenné, a varovania sú považované za chyby. Aj v takýchto prípadoch je však triviálne jednoduché obísť potrebu použitia, stačí použiť explicitnú typovú konverziu na *void* ako príkaz.

Kód 4.10: Zahodenie premennej

```
1 (void)dump;
```

Ak výrazy tohto typu (alebo iné s rovnakým efektom) dôsledne zakážeme používať, aktívne varovanie nepoužitej premennej stále nebude mať želaný efekt, keďže nevynucuje použitie premennej v každej vetve výpočtu. Kvôli tomu neumožňuje efektívne simulovať systémy s garantovaným použitím premennej.

Bez použitia sémantiky presúvania tiež platí pravidlo kontrakcie. Pri jej použití sa ale dá definovať objekty, ktoré nie je možné kopírovať, iba presúvať. Takýto objekt môže patriť najviac jednej premennej, keďže jeho presunutie niekam inam ho odstráni z pôvodnej premennej. Ak je zapnutá kontrola použitia premennej, z ktorej bolo presunuté, neplatí pravidlo kontrakcie, ani jeho modifikovaná verzia, keďže z princípu potrebujeme 2 rovnaké ale nie totožné hodnoty, čo sa nedá nahradiť jednou hodnotou, ak obe presunieme. Zo štandardných typov sa len presúvať dá napríklad $std :: unique\_ptr < T >$, ten navyše garantuje, že po presunutí v premennej zostane prázdny smerník.

Od verzie C++11 so zapnutou kontrolou použitia presunutej premennej je teda dostupný systém afínny. Spolu s kontrolou nepoužitých premenných nedokáže efektívne realizovať lineárny systém. Praktické obmedzenie ale závisí od vlastností implementovaných typov, pre používateľské typy s predvolenými funkciami pre kopírovanie simuluje štrukturálny systém, za predpokladu, že neobsahujú nekopírovateľné typy ako $std :: unique\_ptr < T >$.

Kapitola 5

Implementácia príležitostného slova

Použitím nástrojov opísaných v kapitole 4, ktoré umožňujú používať subštrukturálne typové systémy, demonštrujeme implementáciu príležitostného slova, ktorá v čase kompilácie zabezpečí zamedzenie opakovaného použitia. Pre jednoduchosť bude príležitostné slovo mať tvar celého čísla konečného rozsahu a bude generované jednoduchým pseudonáhodným generátorom. Kryptografická funkcia, požadujúca príležitostné slovo, bude reprezentovaná funkciou obsahujúcou len výpis jeho hodnoty.

5.1 Haskell

Využívame knižnicu linear-base, ktorá je lineárne otypovaným ekvivalentom štandardnej knižnice. Na vstup a výstup obsahuje monádu *System.IO.Linear.IO*, ďalej len ako *Linear.IO*, ktorá je určená na kontrolu prístupu k súborom pomocou lineárnych typov. Obsahuje tiež prvky, ktoré zabezpečujú interakciu so štandardnou nelineárnou knižnicou.

Haskell používa koncept modulov, ktoré združujú príbuzný kód, a umožňujú tvorcovi zverejniť len vybrané spomedzi definovaných prvkov.

Kód 5.1: Lineárny nonce

```
1  data IntNonce = Nonce Int
2
3  newIntNonce' :: Prelude.IO IntNonce
4  newIntNonce' = do
5      gen <- getStdGen
6      let (val::Int, gen') = uniform gen
7      setStdGen gen'
8      return (Nonce val)
9
10 newIntNonce :: System.IO.Linear.IO IntNonce
```

```
11 newIntNonce = System.IO.Linear.fromSystemIO newIntNonce'
12
13 putIntNonce' :: IntNonce -> Prelude.IO ()
14 putIntNonce' (Nonce val) = do
15     putStrLn $ show val
16
17 putIntNonce'' :: IntNonce %1 -> Prelude.IO ()
18 putIntNonce'' = Unsafe.Linear.coerce putIntNonce'
19
20 needs_new_random_every_time :: IntNonce %1 -> System.IO.Linear
       .IO ()
21 needs_new_random_every_time nonce = System.IO.Linear.
       fromSystemIO (putIntNonce'' nonce)
22
23 unwrapIntNonce' :: IntNonce -> Ur Int
24 unwrapIntNonce' (Nonce val) = Ur val
25 unwrapIntNonce :: IntNonce %1 -> Ur Int
26 unwrapIntNonce = Unsafe.Linear.coerce unwrapIntNonce'
```

Dátový typ *IntNonce* pre príležitostné slovo definujeme ako veľmi jednoduchý, obaľujúci hodnotu typu *Int*. Modul nebude zverejňovať funkciu na jeho priame vytvorenie z hodnoty typu Int.

Následne definujeme funkcie na vytváranie hodnôt typu IntNonce. Tu využijeme monádu Linear.IO, ktorá vynúti lineárne obmedzenie pre následné použitie vytvorenej hodnoty. Pseudonáhodný generátor používame zo štandardnej monády *Prelude.IO*. Z kontextu tejto monády získame generátor pomocou *getStdGen*. Ďalšiu hodnotu získavame použitím funkcie *uniform*, ktorej výstupom je vygenerovaná hodnota a nový stav generátora, ktorý treba nastaviť pomocou *setStdGen*, aby ďalšie volanie používalo už tento nový stav.

Najprv definujeme pomocnú funkciu *newIntNonce′*, ktorá funguje v štandardnej monáde *Prelude.IO*. Následne definujeme zverejnenú hodnotu *newIntNonce*, ktorá vznikne aplikovaním konverzie z knižnice linear-base.

Na demonštráciu použitia definujeme funkciu, ktorá príležitostné slovo použije tak, že ho jednoducho vypíše. Ako pri vytváraní, aj tu najprv definujeme Prelude.IO verziu *putIntNonce′*, ktorá obsahuje implementáciu.

Následne definujeme *putIntNonce″*, kde pomocou funkcie coerce autoritatívne hovoríme, že táto funkcia je lineárna. Nakoniec môžeme použitím knižničnej konverziu na Linear.IO monádu definovať zverejnenú verziu, ktorú podľa demonštrovanej požiadavky pomenujeme *needs_new_random_every_time*.

Pre iné použitia bude potrebné, aby používateľ mohol príležitostné slovo rozbaliť a použiť aj sám. Vďaka lineárnemu obmedzeniu na použitie hodnoty ju po rozbalení už používateľ môže použiť len v rozbalenom tvare. Rozbalenú hodnotu mu teda chceme umožniť použiť opakovane, čo docielime jej obalením do typu *Ur*. Tento typ umožňuje vyňať hodnotu z lineárneho obmedzenia. Toto ale používateľovi nedovolí znovu použiť príležitostné slovo, keďže ho nedokáže zabaliť naspäť do typu *IntNonce*. Zverejníme lineárnu verziu *unwrapIntNonce*, ktorú získame použitím coerce na vynútenie lineárneho typu.

Jednoduchý príklad použitia takejto knižnice, kde vytvoríme a následne vypíšeme jedno príležitostné slovo:

Kód 5.2: Lineárny nonce 2

```
1 printNewNonce :: System.IO.Linear.IO (Ur ())
2 printNewNonce = Control.do
3     nonce <- newIntNonce -- vytvor
4     () <- needs_new_random_every_time nonce -- print
5     --() <- needs_new_random_every_time nonce -- print again
6     Control.return (Ur ())
7 main :: Prelude.IO ()
8 main = System.IO.Linear.withLinearIO $ printNewNonce
```

Ak by sme v *printNewNonce* odkomentovali piaty riadok, kde sa znovu použije premenná nonce, dostali by sme kompilačnú chybu, indikujúcu porušenie lineárneho obmedzenia. Môžeme ale pridať vytvorenie ďalšieho príležitostného slova a vypísať ho.

5.2 Rust

Demonštrácia v jazyku Rust pochádza z článku [15], na ktorý práca nadväzuje. Na generovanie náhodných čísel používa populárnu knižnicu rand.

Kód 5.3: Lineárny nonce

```
1 mod nonce {
2     pub struct Nonce {
3         val: u128,
4     }
5
6     impl Nonce {
7         pub fn new() -> Nonce {
8             use rand::prelude::*;
9             Nonce { val: random() }
```

```
10          }
11
12          pub fn get(&self) -> u128 {
13              self.val
14          }
15      }
16  }
17
18  fn need_new_random_u128_every_time(nonce: nonce::Nonce) {
19      let _tmp = nonce.get();
20      println!("Nonce param value: {}", nonce.get());
21      println!("Nonce param value: {}", *nonce);
22  }
```

V jazyku Rust definujeme typ príležitostného slova ako súčasť modulu *nonce*, pričom definujeme aj konštruktor. Typ príležitostného slova *Nonce* definujeme ako jednoduchý, obaľujúci hodnotu typu u128. Obalená hodnota je súkromná, vytvorená v konštrukture.

V konštruktore príležitostného slova náhodne vygenerujeme jeho hodnotu pomocou *rand* :: *prelude* :: *random*(), neumožníme vytváranie príležitostných slov iným spôsobom. Definujeme tiež funkciu *get*, ktorá umožňuje získať obalenú hodnotu.

Keď definujeme knižničnú funkciu, ktorá vyžaduje príležitostné slovo, definujeme príslušný argument tak, aby preberal vlastníctvo od volajúceho. Na demonštráciu implementujeme funkciu *need_new_random_u128_every_time*, ktorá slovo dvakrát vypíše. V kryptografickej knižnici teda samotnú hodnotu môžeme použiť ľubovoľne veľa krát, podľa potreby príslušnej kryptografickej konštrukcie. Volajúci ale nemôže opakovane použiť to isté príležitostné slovo.

Jednoduchý príklad použitia takejto knižnice, kde vytvoríme a následne vypíšeme jedno príležitostné slovo:

Kód 5.4: Lineárny nonce 2

```
1  fn main() {
2      let mut nonce = nonce::Nonce::new(); // create
3      need_new_random_u128_every_time(nonce); // print
4      // nonce = nonce::Nonce::new();
5      // need_new_random_u128_every_time(nonce);
6  }
```

Ak by sme odkomentovali posledný riadok, kde sa znovu použije premenná nonce, dostali by sme kompilačnú chybu, indikujúcu porušenie vlastníctva. Ak odkomentujeme

aj vytvorenie nového príležitostného slova, funkcia korektne vypíše dve (s vysokou pravdepodobnosťou) rôzne príležitostné slová.

5.3 C++

V C++ definujeme príležitostné slovo v mennom priestore, z ktorého zverejníme len povolený spôsob vytvorenia novej hodnoty. Tak, ako ho definujeme, *IntNonce* spĺňa koncept Linear [12], ktorý sme opísali v 4.3.2. Typ obaľuje hodnotu typu *int*32_*t*. Pre ukážku alternatívneho prístupu je hodnota príležitostného slova vytvorené továrenskou funkciou namiesto konštruktora.

Kód 5.5: Lineárny nonce

```
1  class IntNonce {
2  private:
3      int32_t val;
4  public:
5      IntNonce(int32_t value): val(value){};
6
7      IntNonce(const IntNonce& in) = delete;
8      IntNonce(IntNonce&& in) = default;
9      IntNonce& operator=(const IntNonce&) = delete;
10     IntNonce& operator=(IntNonce&&) = default;
11     ~IntNonce(){};
12
13     int32_t getValue() && noexcept {
14         return val;
15     }
16 };
17
18 random_device rd;
19 mt19937 rng(rd());
20 IntNonce new_IntNonce(){
21     int32_t val = rng();
22     IntNonce in(val);
23     return in;
24 }
25
26 void needs_new_random_every_time(IntNonce&& nonce){
27     int32_t val = std::move(nonce).getValue();
```

```
28      cout<<"Nonce: "<<val<<endl;
29      cout<<"That is "<<val<<endl;
30  }
```

Definícia typu je relatívne dlhá, keďže potrebujeme pomocou delete explicitne zakázať niektoré inak dostupné funkcie, ktoré realizujú kopírovanie. Podobne pomocou default povolíme vygenerovanie tých, ktoré chceme, implementáciu tým nechávame na kompilátor.

V definícii metódy *getValue*, ktorá extrahuje hodnotu príležitostného slova, je zaujímavé použitie &&. Znamená, že táto metóda sa dá volať len na dočasnom objekte. Keďže definovaný typ sa nedá kopírovať, efektívne to znamená, že getValue je deštruktívna extrakcia hodnoty. Po jej zavolaní samotný IntNonce bude ako nikam nepriradený dočasný objekt dealokovaný.

Na generovanie náhodných čísel používame generátor *std* :: *mt*19937 zo štandardnej knižnice, inicializovaný pomocou zdroja náhody.

Funkciu *new_IntNonce* zverejníme ako jedinú možnosť vytvoriť hodnotu typu IntNonce. Pre demonštráciu definujeme konzumujúcu funkciu tak, že príležitostné slovo vypíše dva krát. Dôležité je, že funkcia má ako argument rvalue referenciu na príležitostné slovo, teda hodnota musí byť presunutá do jej argumentu. Podobne je tiež nutné použitie *std* :: *move* pri volaní *getValue*, ako sme popísali vyššie. Funkciu *needs_new_random_every_time* tiež zverejníme.

Príklad použitia oboch funkcií môže vyzerať nasledovne:

Kód 5.6: Lineárny nonce 2

```
1  int main(){
2      IntNonce b = new_IntNonce(); // create
3      needs_new_random_every_time(std::move(b)); // print
4      // needs_new_random_every_time(std::move(b));
5      // needs_new_random_every_time(b);
6  }
```

Ak by sme odkomentovali opakované použitie, dostali by sme chybu pri použití statickej analýzy pre použitie po presunutí. Ak by sme sa pokúsili zavolať výpis bez *std* :: *move* ako taký, už pri štandardnej kompilácii dostaneme typovú chybu.

Kapitola 6

Dopad lineárnych typov na výkon

Použitím subštrukturálnych typových systémov získavame na vyjadrovacej sile, najmä v kombinácii s použitím obaľovacích typov nás táto sila potenciálne môže stáť časť výkonu v prevádzke a/alebo počas kompilácie. V tejto kapitole sú popísané experimenty, ktoré sme vykonali za účelom kvantifikovania ceny za získanú silu.

6.1 Princípy testov

Prvý dôležitý test je overenie dopadu rozhrania používajúceho subštrukturálne typy na prevádzku. Keďže všetky populárne kompilátory poskytujú široké schopnosti optimalizácie, ak dokážu pri optimalizácii odstrániť nadbytočné prvky a vygenerovať kód ekvivalentný ako pre program bez využitia subštrukturálnych typov, dopad na výkon počas prevádzky bude nulový. Príklad príležitostného slova je relatívne jednoduchý, takže takéto porovnanie sa dá vykonať pomerne jednoducho tak, že sa pozrieme na výstup kompilácie pre rôzne úrovne optimalizácie (pre lepšiu čitateľnosť zapísané v assembleri, nie ako strojový kód) pomocou niektorého z množstva dostupných nástrojov.

Pre každý testovaný jazyk sme pomocou vlastného generátora vytvorili sadu súborov s rôznym počtom použití rozhrania 6.3. Pri návrhu generátora sme dbali na to, aby každé volanie rozhrania bolo unikátne, čím preventívne bránime zjednoteniu funkcií pri optimalizácii. Vďaka tomu nehrozí, že by sa takáto optimalizácia primiešala do výsledkov, ktoré nás skutočne zaujímajú. Konkrétne volaná funkcia vypisuje hodnotu príležitostného slova plus unikátnej konštanty. Výpis nie je možné pri optimalizácii odstrániť a unikátna konštanta bráni triviálnemu zjednoteniu funkcií. Generátor tiež vytvára funkčne identickú verziu bez použitia subštrukturálnych typov, použitím neobaleného numerického typu. s rovnakou technikou pre unikátnosť volaní. Pripravené boli súbory pre 1, 3, 6, 10, 30, 60, 100, 300, 600, 1000, 3000, 6000, 10 000, 20 000, 40 000, 60 000, 80 000 a 100 000 volaní. Meranie prebiehalo od najmenšieho súboru až do prvého zlyhania.

Po spustení merania boli úplne prvé výsledky značne vysoké, pravdepodobne vplyvom cache a podobných mechanizmov. Preto skutočné merania boli spustené až po krátkom zahrievacom meraní, ktorého výsledky neboli zaznamenávané. Pre každý vstupný súbor boli vykonané dve merania, jedno s vypnutou optimalizáciou a druhé so zapnutou optimalizáciou. Každé meranie bolo opakované desať krát, aby bolo možné sledovať aj rozptyl nameraných hodnôt, a tak určiť finálne hodnoty s väčšou istotou. Hlavnými meranými hodnotami sú reálny čas behu procesu a maximálna veľkosť použitej pamäte.

Vo všetkých prípadoch bol počas merania zbežne sledovaný systémový správca úloh, na odhalenie prípadnej interferencie od iných procesov (ako napr. automatických aktualizácií).

Výsledky meraní budú prezentované vo forme tabuliek a grafov. Pre každý kompilátor a úroveň optimalizácie je daná jedna tabuľka, ktorá pre každú veľkosť vstupu obsahuje priemer a štandardnú odchýlku nameraných hodnôt. Pre dobrú viditeľnosť trendov výsledkov sú tieto dáta zobrazené na grafoch. Vzhľadom na široký rozsah veľkostí vstupov je vodorovná os logaritmická, aby všetky výsledky boli dobre čitateľné. Pre výsledky meraní času je zvislá os tiež logaritmická, meranie pamäte vykazovalo značne menší rozsah, preto os je lineárna. Namerané hodnoty sú reprezentované ako priemer, značený kruhom, a štandardná odchýlka ako čiary nad a pod, resp. cez kruh ak odchýlka je malá. Na grafoch s časom sú jednoduché vstupy reprezentované červenou a lineárne modrou, na grafoch s pamäťou je analogicky oranžová a akvamarínová.

6.2 C++

Pre sledovanie dopadu na čas kompilácie a schopnosť optimalizácie sme použili definíciu rozhrania jednorazového slova, ktorá je popísaná v sekcii 5.3, vrátane podobnej volanej funkcie. Volaná funkcia vyžaduje čerstvú inštanciu, pre testovanie času kompilácie aj optimalizácie je výhodná, keďže výpis nemôže byť odstránený pri optimalizácii. Testovali sme 3 rôzne kompilátory, G++ ako základného reprezentanta, Clang ako známu alternatívu a Microsoft Visual C++ ako oficiálnu alternatívu pre systémy rodiny Windows.

Na pokus s optimalizáciou sme použili nástroj Godbolt [3]. Použité verzie kompilátorov boli: x86-64 gcc vo verzii 13.2, x86-64 clang vo verzii 18.1.0 a x64 msvc verzia v19.38. GCC a Clang používajú na nastavenie úrovne optimalizácie číselné hodnoty, kde 0 znamená vypnutú optimalizáciu a vyššie číslo pridané optimalizácie. Pre nástroj MSVC je vypnutá optimalizácia nastavená ako d (debug), nastavenie 2 optimalizuje rýchlosť kódu, nastavenie 1 naopak optimalizuje veľkosť výsledného súboru [7]. Pre naše účely budeme nastavenie d označovať ako nastavenie 0, analogicky ako majú G++ a Clang.

Pre všetky testované kompilátory sa ukázalo, že od druhej úrovne optimalizácie výsledný strojový kód popisoval len prácu s $int32\_t$. Výsledný súbor teda neobsahoval žiadne volania funkcií, ktoré by korešpondovali s rozhraním, ktorým sme ho obalili. To znamená, že za ochranu pred nevhodným použitím rozhrania potenciálne neplatíme žiadnu cenu počas behu, ak máme dostatočnú úroveň optimalizácie. Na základe tohto výsledku sme tiež použili optimalizáciu úrovne 2 pri meraní času kompilácie.

Prezentované náhľady do strojového kódu sú zľava orezané, keďže niektoré riadky boli kvôli C++ názvom extrémne dlhé a pre ilustráciu stačí ich začiatok. Pre účely prezentácie je použitý kód z generátorov pre 1 volanie rozhrania, s odstránenou číselnou konštantou.

Konkrétne pre GCC obrázok 6.1 ukazuje, že z obaľovacích funkcií sa používa len rozbalenie. Podrobnejší pohľad do strojového kódu 6.2 ukazuje, že z *get* zostala len inštrukcia *mov* a ostatný manažment objektu je úplne preč. Podobnú situáciu vidieť v *main* 6.3, kde zostali len jednoduché inštrukcie a generátor náhodného čísla. Pre porovnanie 6.4, 6.5 ukazujú jasne viditeľné volania obaľovacích funkcií pri vypnutej optimalizácii.

Podobne pre Clang obrázok 6.6 ukazuje, že zostáva len rozbalenie a konštruktor. Aj v tomto prípade podrobnejší pohľad do strojového kódu 6.7 ukazuje, že z *get* zostala len inštrukcia *mov*. Konštruktor sa v tomto prípade stal súčasťou funkcie *main* 6.8, pričom z neho reálne zostalo len volanie náhodného generátora. Ako predtým pre porovnanie 6.9, 6.10 ukazujú jasne viditeľné volania obaľovacích funkcií pri vypnutej optimalizácii.

Pre MSVC je situácia podobná. Na obrázku 6.11 vidieť, že vo výstupe zostali okrem rozbalenia aj konštruktor a deštruktor. Podrobnejší pohľad na volanú funkciu 6.12 ale ukazuje, že nie sú z kódu volané, sú len prítomné inde s súbore. Pohľad na *main* 6.13 dokonca ukazuje, že ani $needs\_fresh$ sa v skutočnosti nezavolá, a vo funkcii zostalo len volanie generátora náhodnéo čísla a výpis. Opäť pre porovnanie 6.14, 6.15 ukazujú jasne viditeľné volania obaľovacích funkcií pri vypnutej optimalizácii.

```cpp
#include <iostream>
#include <type_traits>
#include <memory>
#include <random>

using namespace std;

class IntNonce {
private:
    int32_t val;
public:
    IntNonce(int32_t value): val(value){};
    IntNonce(const IntNonce& in) = delete;
    IntNonce(IntNonce&& in) = default;
    IntNonce& operator=(const IntNonce&) = delete;
    IntNonce& operator=(IntNonce&&) = default;
    ~IntNonce(){};
    int32_t getValue() && noexcept {
        return val;
    }
};

random_device rd;
mt19937 rng(rd());
IntNonce new_IntNonce(){
    int32_t val = rng();
    IntNonce in(val);
    return in;
}

void needs_fresh(IntNonce&& nonce){
    int32_t val = std::move(nonce).getValue();
    cout<<"Nonce: "<<val<<endl;
}

int main(){
    needs_fresh(new_IntNonce());
}
```

Obr. 6.1: C++ kód v GCC s -o2 ponechanými sekciami ofarbenými nástrojom Godbolt

```
3   needs_fresh(IntNonce&&):
4           push    rbp
5           mov     edx, 7
6           mov     esi, OFFSET FLAT:.LC0
7           push    rbx
8           sub     rsp, 8
9           mov     ebx, DWORD PTR [rdi]
10          mov     edi, OFFSET FLAT:std::cout
11          call    std::basic_ostream<char, std::char_traits<char> >&
12          mov     edi, OFFSET FLAT:std::cout
13          mov     esi, ebx
14          call    std::basic_ostream<char, std::char_traits<char> >:
15          mov     rbx, rax
16          mov     rax, QWORD PTR [rax]
```

Obr. 6.2: Zodpovedajúci skompilovaný needs_fresh ofarbený nástrojom Godbolt

```
46  main:
47          sub     rsp, 24
48          mov     rax, QWORD PTR rng[rip+4992]
49          cmp     rax, 623
50          ja      .L21
51  .L19:
52          lea     rdx, [rax+1]
53          mov     rax, QWORD PTR rng[0+rax*8]
54          lea     rdi, [rsp+12]
55          mov     QWORD PTR rng[rip+4992], rdx
56          mov     rdx, rax
57          shr     rdx, 11
58          mov     edx, edx
59          xor     rax, rdx
60          mov     rdx, rax
61          sal     rdx, 7
62          and     edx, 2636928640
63          xor     rax, rdx
64          mov     rdx, rax
65          sal     rdx, 15
66          and     edx, 4022730752
67          xor     rax, rdx
68          mov     rdx, rax
69          shr     rdx, 18
70          xor     rax, rdx
71          mov     DWORD PTR [rsp+12], eax
72          call    needs_fresh(IntNonce&&)
73          xor     eax, eax
74          add     rsp, 24
75          ret
```

Obr. 6.3: Zodpovedajúci skompilovaný main ofarbený nástrojom Godbolt

```
52  needs_fresh(IntNonce&&):
53          push    rbp
54          mov     rbp, rsp
55          sub     rsp, 32
56          mov     QWORD PTR [rbp-24], rdi
57          mov     rax, QWORD PTR [rbp-24]
58          mov     rdi, rax
59          call    std::remove_reference<IntNonce&>::type&& std::move
60          mov     rdi, rax
61          call    IntNonce::getValue() &&
62          mov     DWORD PTR [rbp-4], eax
63          mov     esi, OFFSET FLAT:.LC1
64          mov     edi, OFFSET FLAT:std::cout
65          call    std::basic_ostream<char, std::char_traits<char> >&
66          mov     rdx, rax
67          mov     eax, DWORD PTR [rbp-4]
68          mov     esi, eax
69          mov     rdi, rdx
70          call    std::basic_ostream<char, std::char_traits<char> >:
71          mov     esi, OFFSET FLAT:std::basic_ostream<char, std::cha
72          mov     rdi, rax
73          call    std::basic_ostream<char, std::char_traits<char> >:
74          nop
75          leave
76          ret
```

Obr. 6.4: Skompilovaný needs_fresh pri -o0 ofarbený nástrojom Godbolt

```
77  main:
78          push    rbp
79          mov     rbp, rsp
80          push    rbx
81          sub     rsp, 24
82          lea     rax, [rbp-20]
83          mov     rdi, rax
84          call    new_IntNonce()
85          lea     rax, [rbp-20]
86          mov     rdi, rax
87          call    needs_fresh(IntNonce&&)
88          lea     rax, [rbp-20]
89          mov     rdi, rax
90          call    IntNonce::~IntNonce() [complete object destructor]
91          mov     eax, 0
92          jmp     .L26
93          mov     rbx, rax
94          lea     rax, [rbp-20]
95          mov     rdi, rax
96          call    IntNonce::~IntNonce() [complete object destructor]
97          mov     rax, rbx
98          mov     rdi, rax
99          call    _Unwind_Resume
```

Obr. 6.5: Skompilovaný main pri -o0 ofarbený nástrojom Godbolt

```cpp
#include <iostream>
#include <type_traits>
#include <memory>
#include <random>

using namespace std;

class IntNonce {
private:
    int32_t val;
public:
    IntNonce(int32_t value): val(value){};
    IntNonce(const IntNonce& in) = delete;
    IntNonce(IntNonce&& in) = default;
    IntNonce& operator=(const IntNonce&) = delete;
    IntNonce& operator=(IntNonce&&) = default;
    ~IntNonce(){};
    int32_t getValue() && noexcept {
        return val;
    }
};

random_device rd;
mt19937 rng(rd());
IntNonce new_IntNonce(){
    int32_t val = rng();
    IntNonce in(val);
    return in;
}

void needs_fresh(IntNonce&& nonce){
    int32_t val = std::move(nonce).getValue();
    cout<<"Nonce: "<<val<<endl;
}

int main(){
    needs_fresh(new_IntNonce());
}
```

Obr. 6.6: C++ kód v Clang s -o2 ponechanými sekciami ofarbenými nástrojom Godbolt

```
19  needs_fresh(IntNonce&&):                # @needs_fresh(IntNonce&&)
20          push    r14
21          push    rbx
22          push    rax
23          mov     ebx, dword ptr [rdi]
24          mov     r14, qword ptr [rip + std::cout@GOTPCREL]
25          lea     rsi, [rip + .L.str]
26          mov     edx, 7
27          mov     rdi, r14
28          call    std::basic_ostream<char, std::char_traits<char>
29          mov     rdi, r14
30          mov     esi, ebx
31          call    std::basic_ostream<char, std::char_traits<char>
32          mov     rcx, qword ptr [rax]
```

Obr. 6.7: Zodpovedajúci skompilovaný needs_fresh ofarbený nástrojom Godbolt

```
62  main:                                   # @main
63          push    rax
64          lea     rdi, [rip + rng]
65          call    std::mersenne_twister_engine<unsigned long, 32u
66          mov     dword ptr [rsp + 4], eax
67          lea     rdi, [rsp + 4]
68          call    needs_fresh(IntNonce&&)
69          xor     eax, eax
70          pop     rcx
71          ret
```

Obr. 6.8: Zodpovedajúci skompilovaný main ofarbený nástrojom Godbolt

```
64  needs_fresh(IntNonce&&):                # @needs_fresh(IntNonce&&
65          push    rbp
66          mov     rbp, rsp
67          sub     rsp, 16
68          mov     qword ptr [rbp - 8], rdi
69          mov     rdi, qword ptr [rbp - 8]
70          call    IntNonce::getValue() &&
71          mov     dword ptr [rbp - 12], eax
72          mov     rdi, qword ptr [rip + std::cout@GOTPCREL]
73          lea     rsi, [rip + .L.str]
74          call    std::basic_ostream<char, std::char_traits<char>
75          mov     rdi, rax
76          mov     esi, dword ptr [rbp - 12]
77          call    std::basic_ostream<char, std::char_traits<char>
78          mov     rdi, rax
79          mov     rsi, qword ptr [rip + std::basic_ostream<char,
80          call    std::basic_ostream<char, std::char_traits<char>
81          add     rsp, 16
82          pop     rbp
83          ret
```

Obr. 6.9: Skompilovaný needs_fresh pri -o0 ofarbený nástrojom Godbolt

```
92  main:                                   # @main
93          push    rbp
94          mov     rbp, rsp
95          sub     rsp, 32
96          lea     rdi, [rbp - 4]
97          mov     qword ptr [rbp - 32], rdi       # 8-byte Spill
98          call    new_IntNonce()
99          mov     rdi, qword ptr [rbp - 32]       # 8-byte Reload
100         call    needs_fresh(IntNonce&&)
101         jmp     .LBB12_1
102 .LBB12_1:
103         lea     rdi, [rbp - 4]
104         call    IntNonce::~IntNonce() [base_object_destructor]
105         xor     eax, eax
106         add     rsp, 32
107         pop     rbp
108         ret
109         mov     rcx, rax
110         mov     eax, edx
111         mov     qword ptr [rbp - 16], rcx
112         mov     dword ptr [rbp - 20], eax
113         lea     rdi, [rbp - 4]
114         call    IntNonce::~IntNonce() [base_object_destructor]
115         mov     rdi, qword ptr [rbp - 16]
116         call    _Unwind_Resume@PLT
```

Obr. 6.10: Skompilovaný main pri -o0 ofarbený nástrojom Godbolt

```cpp
#include <iostream>
#include <type_traits>
#include <memory>
#include <random>

using namespace std;

class IntNonce {
private:
    int32_t val;
public:
    IntNonce(int32_t value): val(value){};
    IntNonce(const IntNonce& in) = delete;
    IntNonce(IntNonce&& in) = default;
    IntNonce& operator=(const IntNonce&) = delete;
    IntNonce& operator=(IntNonce&&) = default;
    ~IntNonce(){};
    int32_t getValue() && noexcept {
        return val;
    }
};

random_device rd;
mt19937 rng(rd());
IntNonce new_IntNonce(){
    int32_t val = rng();
    IntNonce in(val);
    return in;
}

void needs_fresh(IntNonce&& nonce){
    int32_t val = std::move(nonce).getValue();
    cout<<"Nonce: "<<val<<endl;
}

int main(){
    needs_fresh(new_IntNonce());
}
```

Obr. 6.11: C++ kód v MSVC s -o2 ponechanými sekciami ofarbenými nástrojom Godbolt

```
1302    void needs_fresh(IntNonce &&) PROC                  ; needs_fresh,
1303    $LN28:
1304            mov     QWORD PTR [rsp+8], rbx
1305            mov     QWORD PTR [rsp+16], rsi
1306            push    rdi
1307            sub     rsp, 48                               ; 00000030H
1308            mov     ebx, DWORD PTR [rcx]
1309            lea     rdx, OFFSET FLAT:`string'
1310            lea     rcx, OFFSET FLAT:std::basic_ostream<char,std::ch
1311            call    std::basic_ostream<char,std::char_traits<char> >
1312            mov     rcx, rax
1313            mov     edx, ebx
1314            call    std::basic_ostream<char,std::char_traits<char> >
1315            mov     rdi, rax
1316            mov     rcx, QWORD PTR [rax]
```

Obr. 6.12: Zodpovedajúci skompilovaný needs_fresh ofarbený nástrojom Godbolt

```
1356    main    PROC                                          ; COMD
1357    $LN34:
1358            mov     QWORD PTR [rsp+8], rbx
1359            mov     QWORD PTR [rsp+16], rsi
1360            push    rdi
1361            sub     rsp, 48                               ; 00000030
1362            lea     rcx, OFFSET FLAT:std::mersenne_twister_engine<
1363            call    unsigned int std::mersenne_twister<unsigned in
1364            mov     ebx, eax
1365            lea     rdx, OFFSET FLAT:`string'
1366            lea     rcx, OFFSET FLAT:std::basic_ostream<char,std::
1367            call    std::basic_ostream<char,std::char_traits<char>
1368            mov     rcx, rax
1369            mov     edx, ebx
1370            call    std::basic_ostream<char,std::char_traits<char>
1371            mov     rdi, rax
1372            mov     rcx, QWORD PTR [rax]
```

Obr. 6.13: Zodpovedajúci skompilovaný main ofarbený nástrojom Godbolt

```
1300  $LN3:
1301          mov     QWORD PTR [rsp+8], rcx
1302          sub     rsp, 56                                 ; 00000038H
1303          mov     rcx, QWORD PTR nonce$[rsp]
1304          call    IntNonce && std::move<IntNonce &>(IntNonce &) ; s
1305          mov     rcx, rax
1306          call    int IntNonce::getValue(void)&&              ; IntN
1307          mov     DWORD PTR val$[rsp], eax
1308          lea     rdx, OFFSET FLAT:$SG39342
1309          lea     rcx, OFFSET FLAT:std::basic_ostream<char,std::cha
1310          call    std::basic_ostream<char,std::char_traits<char> >
1311          mov     edx, DWORD PTR val$[rsp]
1312          mov     rcx, rax
1313          call    std::basic_ostream<char,std::char_traits<char> >
1314          lea     rdx, OFFSET FLAT:std::basic_ostream<char,std::cha
1315          mov     rcx, rax
1316          call    std::basic_ostream<char,std::char_traits<char> >
1317          add     rsp, 56                                 ; 00000038H
1318          ret     0
1319  void needs_fresh(IntNonce &&) ENDP                       ; needs_fresh
```

Obr. 6.14: Skompilovaný needs_fresh pri -o0 ofarbený nástrojom Godbolt

```
1324  main    PROC
1325  $LN4:
1326          sub     rsp, 72                                 ; 00000048H
1327          lea     rcx, QWORD PTR $T1[rsp]
1328          call    IntNonce new_IntNonce(void)             ; new_IntNonce
1329          mov     QWORD PTR tv71[rsp], rax
1330          mov     rax, QWORD PTR tv71[rsp]
1331          mov     QWORD PTR tv70[rsp], rax
1332          mov     rcx, QWORD PTR tv70[rsp]
1333          call    void needs_fresh(IntNonce &&)           ; needs_fresh
1334          npad    1
1335          lea     rcx, QWORD PTR $T1[rsp]
1336          call    IntNonce::~IntNonce(void)                   ; In
1337          xor     eax, eax
1338          add     rsp, 72                                 ; 00000048H
1339          ret     0
1340  main    ENDP
```

Obr. 6.15: Skompilovaný main pri -o0 ofarbený nástrojom Godbolt

Čas kompilácie pomocou G++ a Clang sme merali vo virtuálnom stroji, s operačným systémom Ubuntu verzie 22.04 LTS. Na test boli použité kompilátory z oficiálnych repozitárov, nainštalované boli g++ verzia 11.4.0-1ubuntu1 a clang verzia 14.0.0-1ubuntu1.1. Meranie MSVC bolo vykonané priamo na stroji s operačným systémom Windows 10. Nainštalovaný bol balík pre Visual Studio 2022 z oficiálnej stránky, verzia kompilátora 19.39.33523. Na meranie času a pamäte bol použitý bash s GNU time pod Ubuntu, a MinGW bash s vstavaným príkazom time pod Windows. Na rozdiel od samostatného spustiteľného súboru vstavaný time neposkytuje meranie použitej pamäte, preto toto nie je súčasťou výsledkov pre MSVC. Kompilácia pomocou MSVC vyžaduje najprv inicializáciu prostredia [8], keďže pri úvodnom testovaní inicializačný skript vykazoval stabilný čas, je pre konzistentnosť súčasťou meraného času.

Pri meraní pre G++ a Clang boli všetky zlyhania rovnaké, vo všetkých prípadoch meranie skončilo náhlym ukončením procesu, vrátane procesu bash, v ktorom bolo samotné meranie spustené. Pravdepodobne to bolo spôsobené obmedzenou operačnou pamäťou virtuálneho stroja, posledné merania prekračovali 1GB z 2GB nastavenej pamäte. Konzistentnosť zlyhania je pomerne dobrý znak toho, že vo všetkých prípadoch meranie narazilo na rovnaký limit, teda zvládnuté veľkosti vstupu je možné férovo porovnávať. Merania MSVC úspešne prebehli všetky, s nižšou pozorovanou pamäťou.

V súlade s očakávaniami bola v každom prípade kompilácia kódu s lineárnymi prvkami pomalšia. Hlavným pozitívnym zistením je, že spomalenie je vo väčšine prípadov akceptovateľné, menej ako dvojnásobné aj pre veľké vstupy, a pre rozumne veľké vstupy vo všetkých prípadoch do 25 percent.

Pre G++ pri optimalizácii na úrovni 0 boli do veľkosti vstupu 100 výsledky oboch verzií veľmi podobné, pre väčšie vstupy postupne narastal rozdiel až na 60 percent pre 20 tisíc. Pri 40 tisícoch už zlyhala kompilácia pre lineárny kód, kompilácia jednoduchého kódu zvládla len jednu z plánovaných iterácií, preto nie je zahrnutá v grafoch a v tabuľke nemá danú odchýlku. Podobne aj pamäťová náročnosť bola veľmi podobná do 100 vstupov, potom postupne začal narastať rozdiel až na 80 percent.

Pri optimalizácii na úrovni 2 sme pri prvom meraní mali mierne problémy so stabilitou výsledkov pre veľké súbory s jednoduchým kódom. Pri opakovaní merania vyšli časy o niečo nižšie ako pri prvom meraní, ale trend je zjavne veľmi podobný ako pre optimalizáciu 0. Pre menšie vstupy je spomalenie pomerne konštantné, pre 600 vstupov začína rásť až na takmer dvojnásobné pre 20 tisíc. Pri tejto úrovni skončili oba testy úspešne kompiláciu s 20 tisícovými vstupmi a oba zlyhali pre 40 tisíc.

Pre Clang pri optimalizácii na úrovni 0 sa pre vstupy do 1000 pohyboval rozdiel v čase okolo 10 až 20 percent. Pre väčšie vstupy postupne rozdiel rástol až na 60 percent pri 80 tisícovom vstupe, pre 100 tisíc už zlyhala kompilácia lineárneho vstupu a jednoduchý zvládol len jedno z 10 plánovaných opakovaní. Toto meranie je preto v tabuľke bez danej odchýlky, a nie je zahrnuté v grafoch. Na výsledkoch je tiež vidieť,

že Clang sa s podobnou pamäťou dostal výrazne ďalej ako G++.

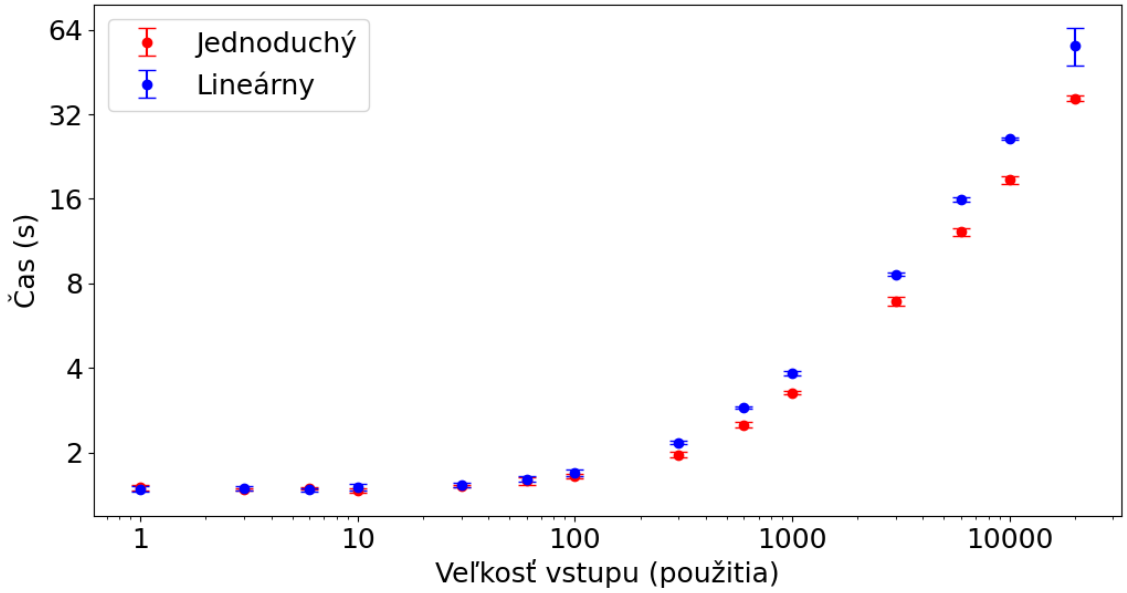
Pri optimalizácii na úrovni 2 bola situácia podobná. Do 300 vstupov bol rozdiel minimálny, pri 3000 sa ešte držal pod 10 percent. Pre väčšie vstupy čas aj rozdiel rástol agresívnejšie ako pri optimalizácii 0, až na 86 percentný rozdiel pri 40 tisícovom vstupe, pri 60 tisíc už lineárny vstup spôsobil zlyhanie. Jednoduchý vstup so 60 tisícami ešte bol zvládnutý so všetkými opakovaniami.

Jediný prípad, v ktorom sme zaznamenali výrazné rozdiely medzi časmi kompilácie, bol kompilátor MSVC pre optimalizácii na úrovni 0. Pre vstupy do 1000 sa rozdiely držali do 5 percent, pre väčšie vstupy však rozdiel pomerne prudko nastúpal takmer na 15 násobok. Pozorovaná záťaž systému bola pre finálne merania okolo 35 percent CPU, do 110 MB pamäte pre jednoduchý kód a pre lineárny až 630MB, tesne pred koncom skokovo 890MB.

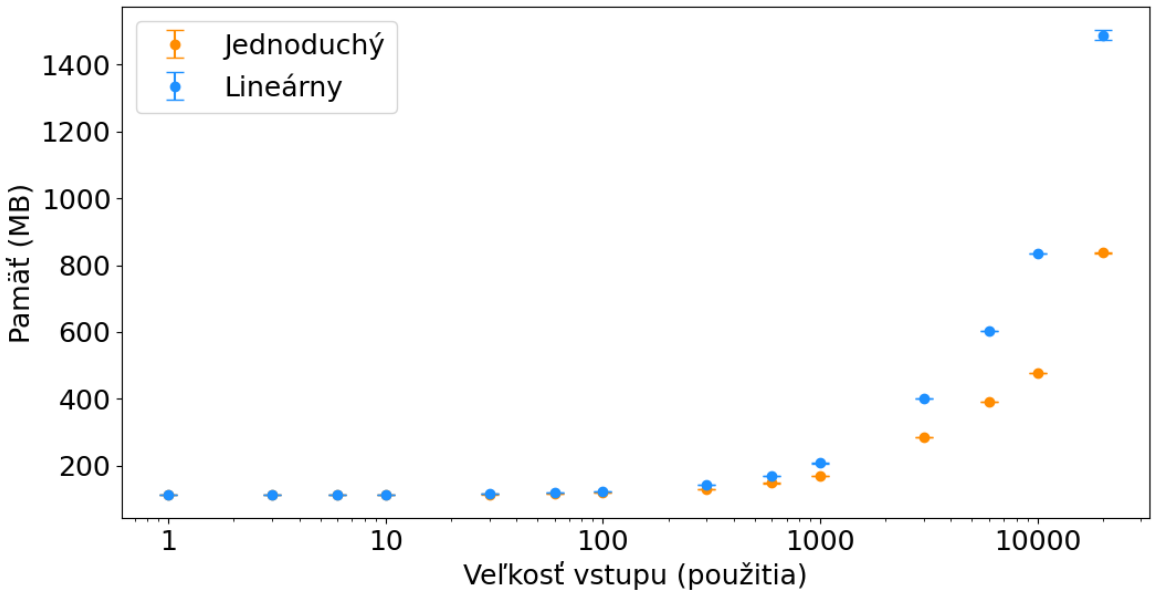
Pri optimalizácii na úrovni 2 boli výsledky podstatne lepšie. Aj časy pre najväčšie vstupy sú veľmi blízko výsledkom pre jednoduchý kód s vypnutou optimalizáciou. Takmer všetky vstupy skončili s rozdielom menším ako 15 percent, nad 20 percent sa dostali len 2 veľkosti a to prekvapivo 1 a 3 tisíc. Jednoduché aj lineárne vstupy vykazujú jasný skok medzi 3 a 6 tisícovými vstupmi, kde prekvapivo poklesne časová náročnosť. Grafy pred aj po skoku vyzerajú podobne ako ostatné merania, s pomalým rastom pre malé súbory a superlineárne pre veľké súbory. Tu je tiež dôležité, že merania jednoduchých a lineárnych kódov prebiehali samostatne, t.j. je veľmi nepravdepodobné, že tento jav je spôsobený napríklad interferenciou iného procesu. To takmer určite znamená, že kompilátor efektívne detegoval možnosť optimalizácie, čím zabránil nárastu času ako pri lineárnom kóde s vypnutou optimalizáciou. Je tiež možné, že kompilátor používa rôzne stratégie podľa veľkosti súboru, čo by vysvetlilo náhly skok pre obe sady vstupov. Pozorovaná záťaž systému bola 200 MB pre jednoduchý kód, pre lineárny 210MB a skokovo 350MB a 100 percent CPU. Aj pamäťová náročnosť pre lineárny kód teda bola výrazne nižšia ako pri vypnutej optimalizácii.

Tabuľka 6.1: Výsledky G++ optimalizácia 0

Veľkosť	Jednoduchý		Lineárny		Čas lin/jed
	čas (s)	pamäť (kB)	čas (s)	pamäť (kB)	
1	1,502 ± 0,039	113226	1,487 ± 0,034	113358	0,990
3	1,482 ± 0,019	113424	1,488 ± 0,027	113612	1,004
6	1,492 ± 0,016	113479	1,477 ± 0,019	113792	0,990
10	1,469 ± 0,028	113751	1,505 ± 0,040	114124	1,025
30	1,522 ± 0,015	114770	1,536 ± 0,024	115880	1,009
60	1,587 ± 0,049	116406	1,609 ± 0,038	118877	1,014
100	1,651 ± 0,031	118852	1,698 ± 0,044	122700	1,028
300	1,965 ± 0,044	129942	2,175 ± 0,030	142223	1,107
600	2,513 ± 0,052	147980	2,901 ± 0,032	169758	1,154
1000	3,264 ± 0,046	170373	3,834 ± 0,056	207612	1,175
3000	6,907 ± 0,234	286657	8,634 ± 0,083	400550	1,250
6000	12,200 ± 0,357	390791	15,937 ± 0,269	602147	1,306
10000	18,688 ± 0,540	476826	26,294 ± 0,258	834197	1,407
20000	36,427 ± 0,891	838111	56,343 ± 8,592	1488330	1,547
40000	97,160	1231432			



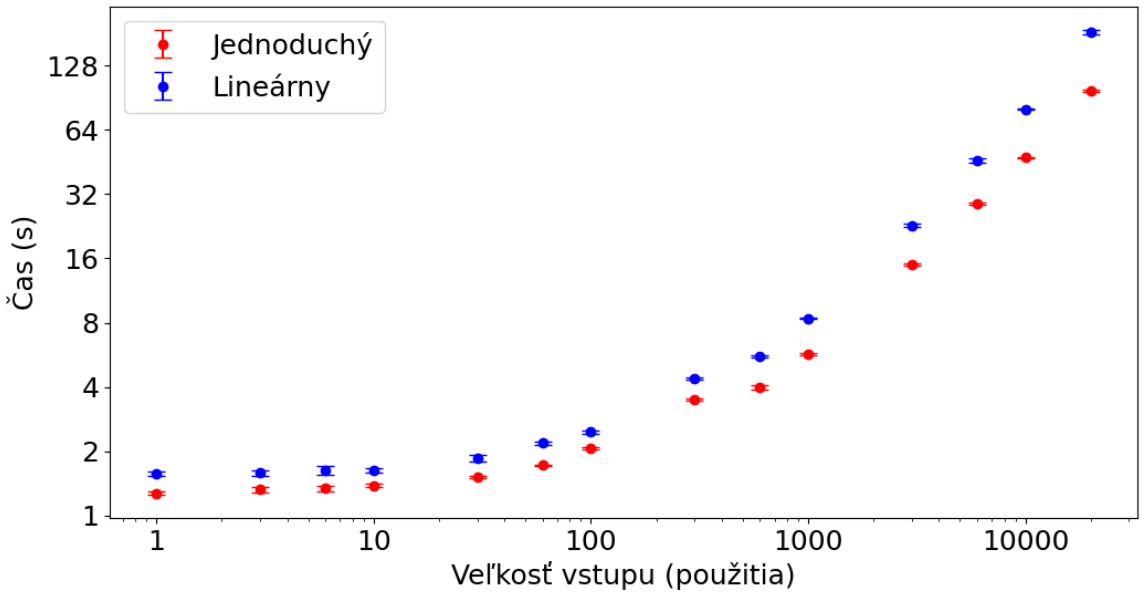
Obr. 6.16: Čas kompilácie pre G++ s optimalizáciou 0



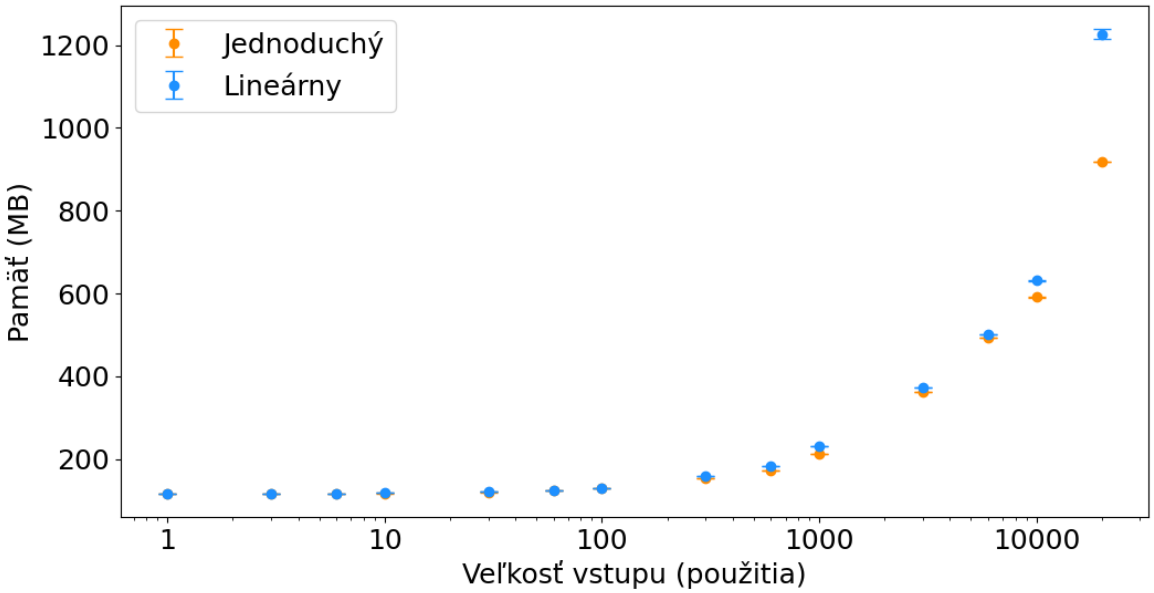
Obr. 6.17: Použitá pamäť kompilácie pre G++ s optimalizáciou 0

Tabuľka 6.2: Výsledky G++ optimalizácia 2

Veľkosť	Jednoduchý		Lineárny		Čas lin/jed
	čas (s)	pamäť (kB)	čas (s)	pamäť (kB)	
1	1,271 ± 0,022	116776	1,568 ± 0,029	116840	1,234
3	1,320 ± 0,043	117084	1,578 ± 0,046	117165	1,195
6	1,335 ± 0,034	117482	1,623 ± 0,072	117574	1,216
10	1,382 ± 0,026	117944	1,626 ± 0,033	118160	1,177
30	1,517 ± 0,021	120616	1,854 ± 0,060	121062	1,222
60	1,719 ± 0,011	124454	2,178 ± 0,035	125535	1,267
100	2,058 ± 0,024	129450	2,464 ± 0,046	131150	1,197
300	3,478 ± 0,050	155268	4,383 ± 0,067	158652	1,260
600	3,975 ± 0,090	173615	5,565 ± 0,062	184207	1,400
1000	5,682 ± 0,064	212044	8,364 ± 0,049	230720	1,472
3000	14,949 ± 0,230	362756	22,884 ± 0,364	372965	1,531
6000	28,779 ± 0,316	492841	45,860 ± 1,097	500991	1,594
10000	47,690 ± 0,302	590923	80,071 ± 0,602	631025	1,679
20000	97,389 ± 1,257	918361	184,513 ± 3,964	1226634	1,895



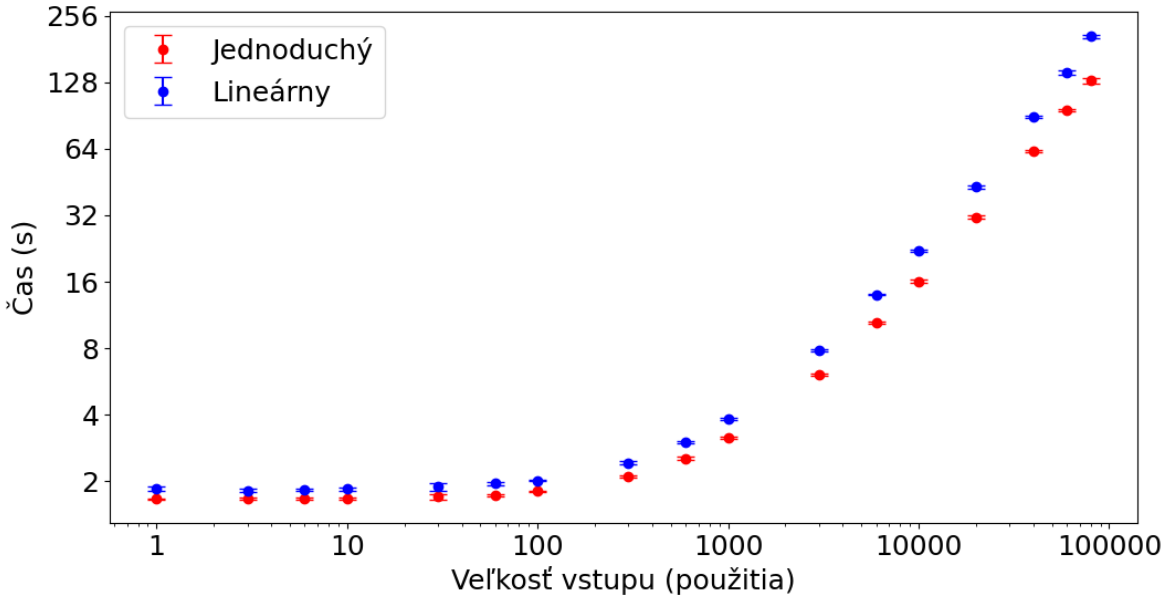
Obr. 6.18: Čas kompilácie pre G++ s optimalizáciou 2



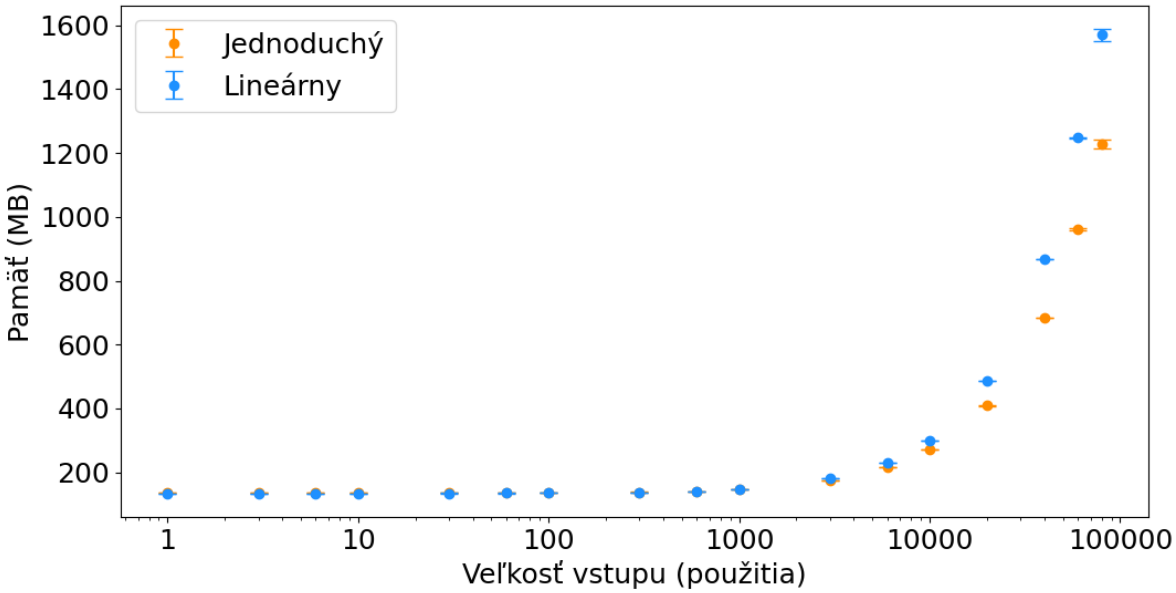
Obr. 6.19: Použitá pamäť kompilácie pre G++ s optimalizáciou 2

Tabuľka 6.3: Výsledky Clang optimalizácia 0

Veľkosť	Jednoduchý		Lineárny		Čas lin/jed
	čas (s)	pamäť (kB)	čas (s)	pamäť (kB)	
1	1,663 ± 0,013	135073	1,853 ± 0,049	133627	1,114
3	1,666 ± 0,018	135231	1,817 ± 0,031	133478	1,091
6	1,673 ± 0,021	135348	1,827 ± 0,025	133567	1,092
10	1,676 ± 0,020	135311	1,839 ± 0,040	133710	1,097
30	1,698 ± 0,044	135740	1,883 ± 0,073	134193	1,109
60	1,732 ± 0,023	136390	1,950 ± 0,026	134841	1,126
100	1,799 ± 0,013	136729	2,013 ± 0,018	135226	1,119
300	2,097 ± 0,020	137983	2,419 ± 0,046	137033	1,154
600	2,534 ± 0,047	140479	3,003 ± 0,033	140788	1,185
1000	3,143 ± 0,045	146082	3,826 ± 0,045	147405	1,217
3000	6,079 ± 0,093	173612	7,862 ± 0,102	180636	1,293
6000	10,471 ± 0,114	215318	13,992 ± 0,118	230518	1,336
10000	16,073 ± 0,217	271327	22,098 ± 0,359	298227	1,375
20000	31,444 ± 0,455	408747	43,144 ± 0,793	485592	1,372
40000	62,432 ± 0,507	684421	88,994 ± 0,973	867947	1,425
60000	95,884 ± 1,463	960304	141,690 ± 3,574	1247706	1,478
80000	130,413 ± 3,382	1227971	206,857 ± 3,096	1570341	1,586
100000	193,840	1405124			



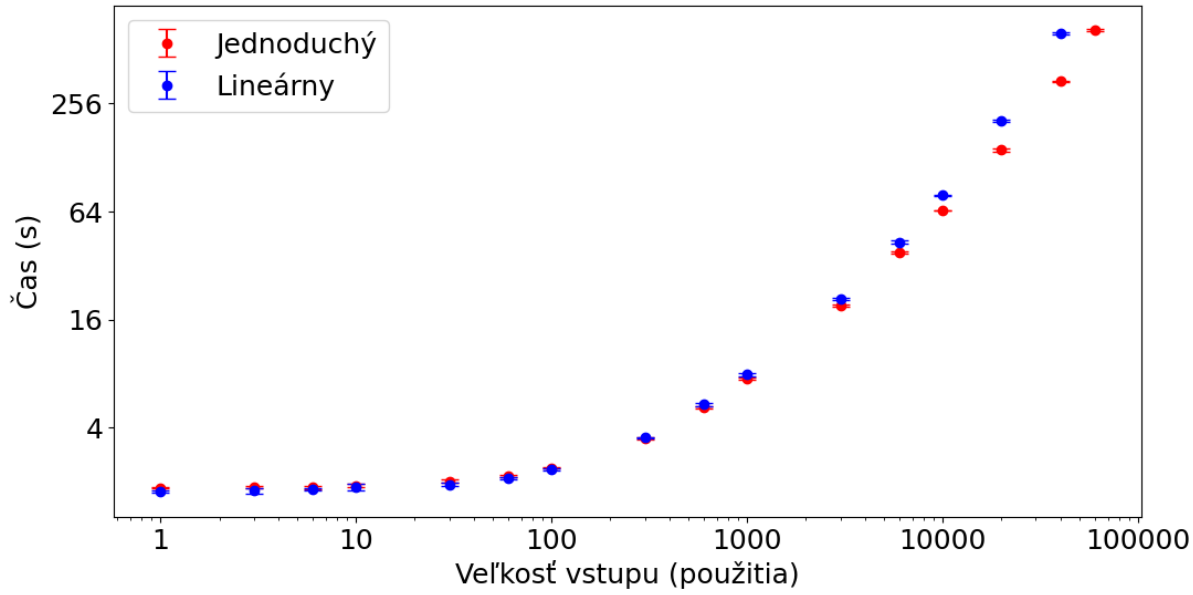
Obr. 6.20: Čas kompilácie pre Clang s optimalizáciou 0



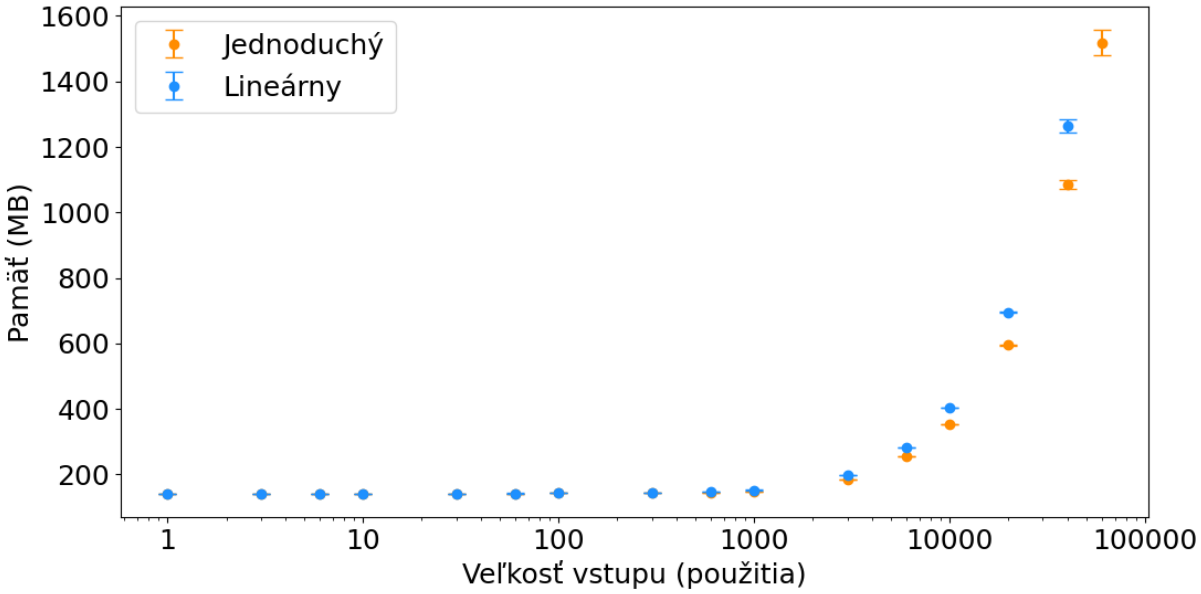
Obr. 6.21: Použitá pamäť kompilácie pre Clang s optimalizáciou 0

Tabuľka 6.4: Výsledky Clang optimalizácia 2

Veľkosť	Jednoduchý		Lineárny		Čas lin/jed
	čas (s)	pamäť (kB)	čas (s)	pamäť (kB)	
1	1,853 ± 0,020	140832	1,767 ± 0,024	140752	0,954
3	1,879 ± 0,025	140721	1,782 ± 0,066	140952	0,948
6	1,856 ± 0,027	140867	1,807 ± 0,026	140902	0,974
10	1,901 ± 0,037	140889	1,876 ± 0,083	141048	0,987
30	2,015 ± 0,037	141084	1,932 ± 0,036	141247	0,959
60	2,159 ± 0,029	141716	2,099 ± 0,027	141802	0,972
100	2,395 ± 0,021	142422	2,342 ± 0,037	142498	0,978
300	3,472 ± 0,045	143546	3,521 ± 0,031	144660	1,014
600	5,175 ± 0,063	143864	5,393 ± 0,137	146914	1,042
1000	7,498 ± 0,114	146696	7,899 ± 0,139	152017	1,053
3000	19,251 ± 0,268	183115	20,857 ± 0,275	197189	1,083
6000	37,793 ± 0,627	254708	43,130 ± 0,833	283397	1,141
10000	64,857 ± 0,357	353672	78,715 ± 0,708	404071	1,214
20000	141,314 ± 3,446	594519	205,113 ± 2,288	695281	1,451
40000	339,426 ± 2,933	1084204	631,687 ± 10,426	1264428	1,861
60000	655,524 ± 10,507	1518305			



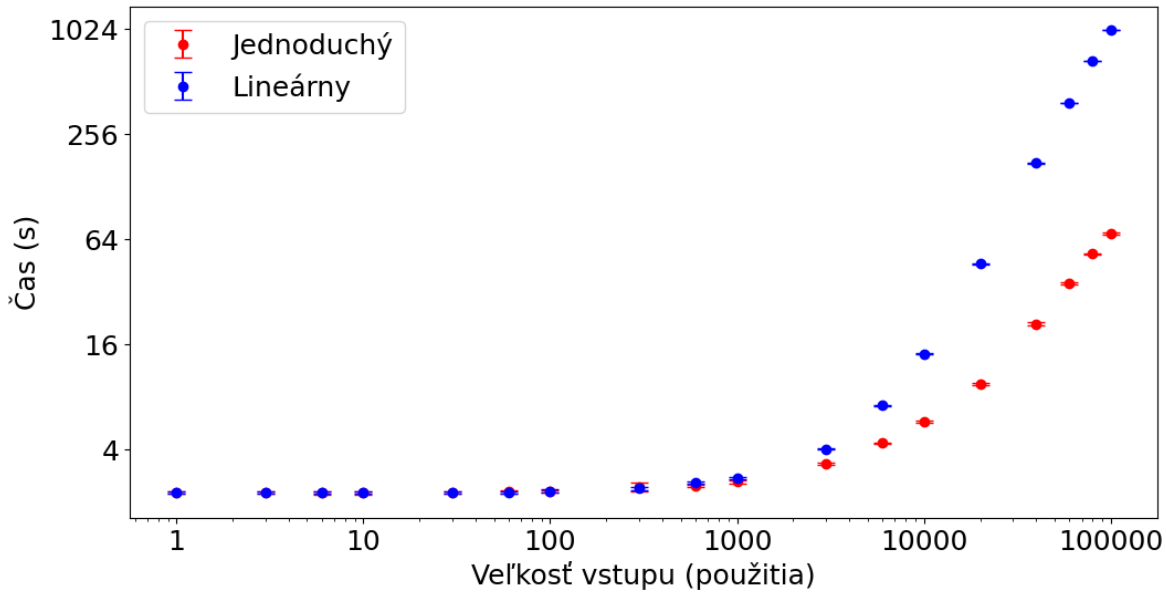
Obr. 6.22: Čas kompilácie pre Clang s optimalizáciou 2



Obr. 6.23: Použitá pamäť kompilácie pre Clang s optimalizáciou 2

Tabuľka 6.5: Výsledky MSVC optimalizácia 0 (d)

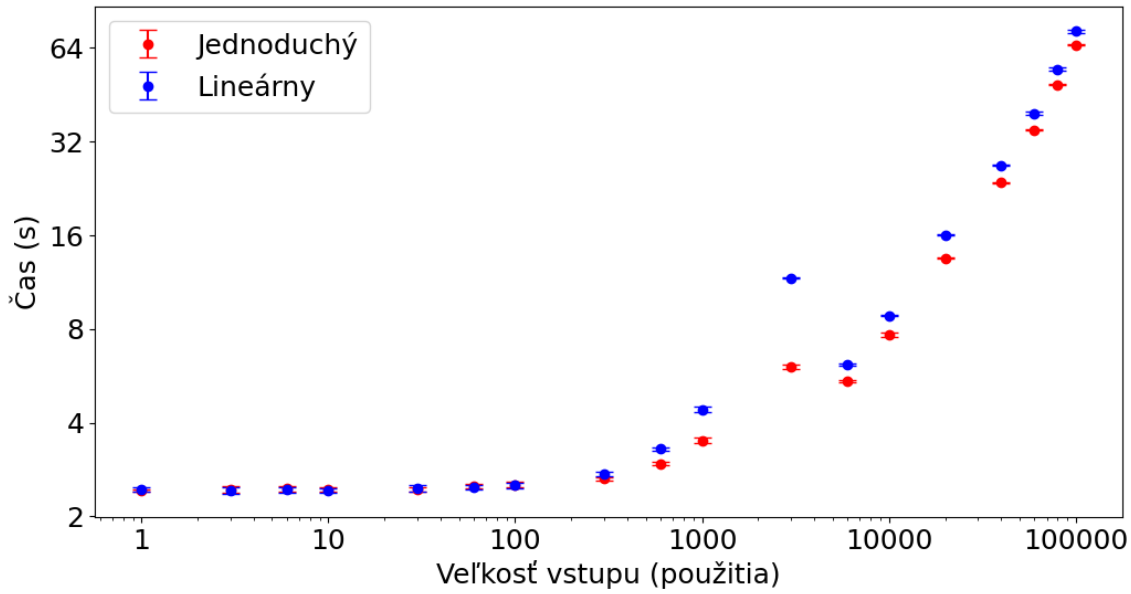
Veľkosť	Jednoduchý čas (s)	Lineárny čas (s)	Čas lin/jed
1	2,268 ± 0,039	2,269 ± 0,037	1,000
3	2,256 ± 0,040	2,277 ± 0,034	1,009
6	2,252 ± 0,053	2,270 ± 0,039	1,008
10	2,251 ± 0,042	2,254 ± 0,036	1,001
30	2,258 ± 0,028	2,255 ± 0,037	0,999
60	2,283 ± 0,045	2,270 ± 0,026	0,994
100	2,307 ± 0,027	2,301 ± 0,048	0,997
300	2,448 ± 0,139	2,390 ± 0,043	0,976
600	2,472 ± 0,040	2,574 ± 0,047	1,041
1000	2,601 ± 0,066	2,742 ± 0,043	1,054
3000	3,303 ± 0,054	4,005 ± 0,029	1,213
6000	4,356 ± 0,038	7,123 ± 0,040	1,635
10000	5,754 ± 0,065	14,059 ± 0,121	2,443
20000	9,470 ± 0,170	46,283 ± 0,276	4,887
40000	20,917 ± 0,411	174,407 ± 0,503	8,338
60000	35,784 ± 0,632	385,181 ± 0,669	10,764
80000	52,698 ± 0,507	671,224 ± 1,821	12,737
100000	69,175 ± 0,935	1007,105 ± 2,426	14,559



Obr. 6.24: Čas kompilácie pre MSVC s optimalizáciou 0 (d)

Tabuľka 6.6: Výsledky MSVC optimalizácia 2

Veľkosť	Jednoduchý čas (s)	Lineárny čas (s)	Čas lin/jed
1	2,425 ± 0,027	2,443 ± 0,045	1,007
3	2,444 ± 0,051	2,423 ± 0,059	0,991
6	2,454 ± 0,044	2,435 ± 0,047	0,992
10	2,448 ± 0,038	2,426 ± 0,044	0,991
30	2,446 ± 0,040	2,465 ± 0,056	1,008
60	2,506 ± 0,036	2,480 ± 0,038	0,990
100	2,531 ± 0,051	2,515 ± 0,045	0,994
300	2,649 ± 0,034	2,737 ± 0,043	1,033
600	2,956 ± 0,044	3,299 ± 0,035	1,116
1000	3,513 ± 0,062	4,408 ± 0,097	1,255
3000	6,041 ± 0,087	11,668 ± 0,053	1,931
6000	5,455 ± 0,052	6,168 ± 0,050	1,131
10000	7,656 ± 0,109	8,848 ± 0,063	1,156
20000	13,506 ± 0,071	16,069 ± 0,071	1,190
40000	23,644 ± 0,107	26,793 ± 0,105	1,133
60000	34,865 ± 0,076	39,439 ± 0,431	1,131
80000	48,745 ± 0,291	54,717 ± 0,562	1,123
100000	65,531 ± 0,227	72,449 ± 0,743	1,106



Obr. 6.25: Čas kompilácie pre MSVC s optimalizáciou 2

6.3 Rust

Pre sledovanie dopadu na čas kompilácie a schopnosť optimalizácie sme použili definíciu rozhrania jednorazového slova, ktorá je popísaná v [15], vrátane podobnej volanej funkcie, vyžadujúcej čerstvú inštanciu, keďže výpis nemôže byť odstránený pri optimalizácii. Alternatíva bez použitia obaľovacej štruktúry je takmer identická, s priamym použitím číselného typu, pričom na zachovanie štýlu je vygenerovanie jednorazového slova definované ako samostatná funkcia, použitá analogicky ako konštruktor štruktúry.

Na pokus s optimalizáciou sme použili nástroj Godbolt [3], s prednastaveným kompilátorom rustc vo verzii 1.76.0. Použitá definícia odkazuje na externú knižnicu rand, ktorá patrí do zoznamu 100 najpoužívanejších knižníc pre Rust. Tieto knižnice sú v nástroji Godbolt podporované, stačí teda otvoriť menu pre správu knižníc a zvoliť ju z ponuky.

Prezentované náhľady do strojového kódu sú zľava orezané, keďže niektoré riadky boli kvôli pomerne dlhé a pre ilustráciu stačí ich začiatok. Pre účely prezentácie je použitý kód z generátorov pre 1 volanie rozhrania, s odstránenou číselnou konštantou.

Od druhej úrovne optimalizácie výsledný strojový kód popisoval takmer výlučne prácu s číselným typom. Z obaľovacích funkcií zostali len základné kostry, konkrétne *new* 6.27, 6.28 obsahuje len trochu práce so zásobníkom a generovanie náhodného čísla (medzi zobrazenými sekciami). Podobne *get* bol inlinovaný aj s celou funkciu, ktorá ho volala, do *main* 6.29, zostali z neho len priamočiare *mov* inštrukcie. To znamená, že za ochranu pred nevhodným použitím rozhrania potenciálne neplatíme prakticky

```
1  extern crate rand;
2
3  pub struct Nonce {
4      val: u128,
5  }
6
7  impl Nonce {
8  pub fn new() -> Nonce {
9      use rand::prelude::*;
10     Nonce {
11         val: random()
12     }
13 }
14
15 pub fn get(&self) -> u128 {
16     self.val
17 }
18 }
19
20 fn needs_fresh(nonce: Nonce) {
21     let tmp = nonce.get();
22     println!("Nonce: {}", tmp);
23 }
24
25 pub fn main() {
26     needs_fresh(Nonce::new());
27 }
```

Obr. 6.26: Rustový kód s -o2 ponechanými sekciami ofarbenými nástrojom Godbolt

žiadnu cenu počas behu, ak máme dostatočnú úroveň optimalizácie. Na základe tohto výsledku sme tiež použili optimalizáciu úrovne 2 pri meraní času kompilácie.

Pre porovnanie 6.30, 6.31 ukazujú jasne viditeľné volania obaľovacích funkcií pri vypnutej optimalizácii.

```
78  example::Nonce::new::hd1dd94632ef96955:
79          push    r15
80          push    r14
81          push    r13
82          push    r12
83          push    rbx
84          call    qword ptr [rip + rand::rngs::thread::thread_rng
```

Obr. 6.27: Zodpovedajúci skompilovaný needs_fresh ofarbený nástrojom Godbolt - hlavička

```
178 .LBB1_32:
179         or      r15, r12
180         mov     rax, r14
181         mov     rdx, r15
182         pop     rbx
183         pop     r12
184         pop     r13
185         pop     r14
186         pop     r15
187         ret
```

Obr. 6.28: Zodpovedajúci skompilovaný needs_fresh ofarbený nástrojom Godbolt - päta

```
238 example::main::h238d10f39e0a48fc:
239         sub     rsp, 88
240         call    qword ptr [rip + example::Nonce::new::hd1dd94632ef
241         mov     qword ptr [rsp + 16], rdx
242         mov     qword ptr [rsp + 8], rax
243         lea     rax, [rsp + 8]
244         mov     qword ptr [rsp + 24], rax
245         mov     rax, qword ptr [rip + core::fmt::num::<impl core::
246         mov     qword ptr [rsp + 32], rax
247         lea     rax, [rip + .L__unnamed_2]
248         mov     qword ptr [rsp + 40], rax
249         mov     qword ptr [rsp + 48], 2
250         mov     qword ptr [rsp + 72], 0
251         lea     rax, [rsp + 24]
252         mov     qword ptr [rsp + 56], rax
253         mov     qword ptr [rsp + 64], 1
254         lea     rdi, [rsp + 40]
255         call    qword ptr [rip + std::io::stdio::_print::h599d580b
256         add     rsp, 88
257         ret
```

Obr. 6.29: Zodpovedajúci skompilovaný main ofarbený nástrojom Godbolt

```
464  example::needs_fresh::hbc246efd918617f9:
465          sub     rsp, 120
466          mov     qword ptr [rsp + 8], rdi
467          mov     qword ptr [rsp + 16], rsi
468          mov     rax, qword ptr [rip + example::Nonce::get::hf1e8
469          lea     rdi, [rsp + 8]
470          call    rax
471          mov     qword ptr [rsp + 32], rdx
472          mov     qword ptr [rsp + 24], rax
473          lea     rax, [rsp + 24]
474          mov     qword ptr [rsp + 104], rax
475          mov     rax, qword ptr [rip + core::fmt::num::<impl core
476          mov     qword ptr [rsp + 112], rax
477          mov     rcx, qword ptr [rsp + 104]
478          mov     rax, qword ptr [rsp + 112]
479          mov     qword ptr [rsp + 88], rcx
480          mov     qword ptr [rsp + 96], rax
481          lea     rdi, [rsp + 40]
482          lea     rsi, [rip + .L__unnamed_13]
483          mov     edx, 2
484          lea     rcx, [rsp + 88]
485          mov     r8d, 1
486          call    core::fmt::Arguments::new_v1::hcaec27582bbd0fcf
487          lea     rdi, [rsp + 40]
488          call    qword ptr [rip + std::io::stdio::_print::h599d58
489          add     rsp, 120
490          ret
```

Obr. 6.30: Skompilovaný needs_fresh pri -o0 ofarbený nástrojom Godbolt

```
492  example::main::h238d10f39e0a48fc:
493          push    rax
494          call    qword ptr [rip + example::Nonce::new::hd1dd9463
495          mov     rdi, rax
496          mov     rsi, rdx
497          call    example::needs_fresh::hbc246efd918617f9
498          pop     rax
499          ret
```

Obr. 6.31: Skompilovaný main pri -o0 ofarbený nástrojom Godbolt

Čas kompilácie sme merali vo virtuálnom stroji s operačným systémom Ubuntu verzie 22.04 LTS. Na test bol použitý kompilátor rustc z oficiálnych repozitárov, nainštalovaná bola verzia 1.75.0. Pri týchto meraniach boli všetky zlyhania rovnaké, vo všetkých prípadoch meranie skončilo náhlym ukončením procesu, pričom bol ukončený aj proces bash, v ktorom bolo samotné meranie spustené. Toto je pomerne dobrý znak toho, že vo všetkých prípadoch meranie narazilo na rovnaký limit, teda zvládnuté veľkosti vstupu je možné férovo porovnávať. Pre všetky merania bola najväčšia úspešná veľkosť vstupu 20000.

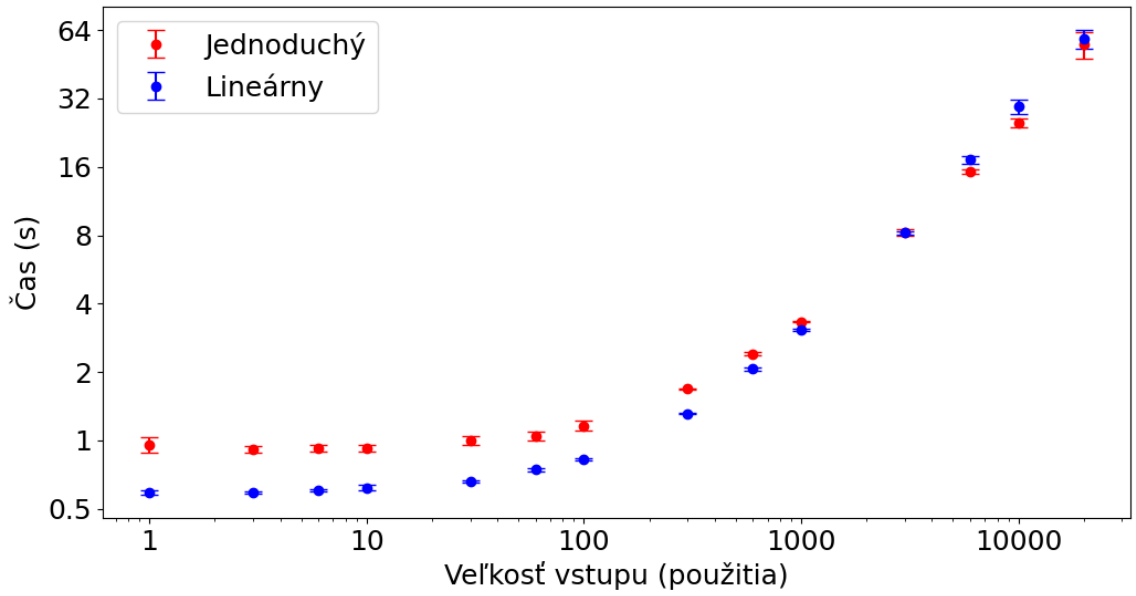
Naše merania zaznamenali veľmi pozitívne výsledky. Pre obe testované úrovne optimalizácie bol pre väčšie vstupy čas kompilácie kódu s lineárnymi prvkami takmer identický ako pre kód bez lineárneho rozhrania. Dokonca v prípade menších vstupov pri optimalizácii na úrovni 0 bola kompilácia lineárneho kódu o niečo rýchlejšia. Pre optimalizáciu úrovne 2 bola kompilácia lineárneho kódu konzistentne o trochu rýchlejšia. Taktiež pamäťové nároky boli pre obe úrovne optimalizácie takmer identické pre obe verzie kódu.

Prejavil sa tiež značný rozdiel v čase medzi úrovňami optimalizácie, ktorá bola pre veľké vstupy viac ako desaťnásobne pomalšia.

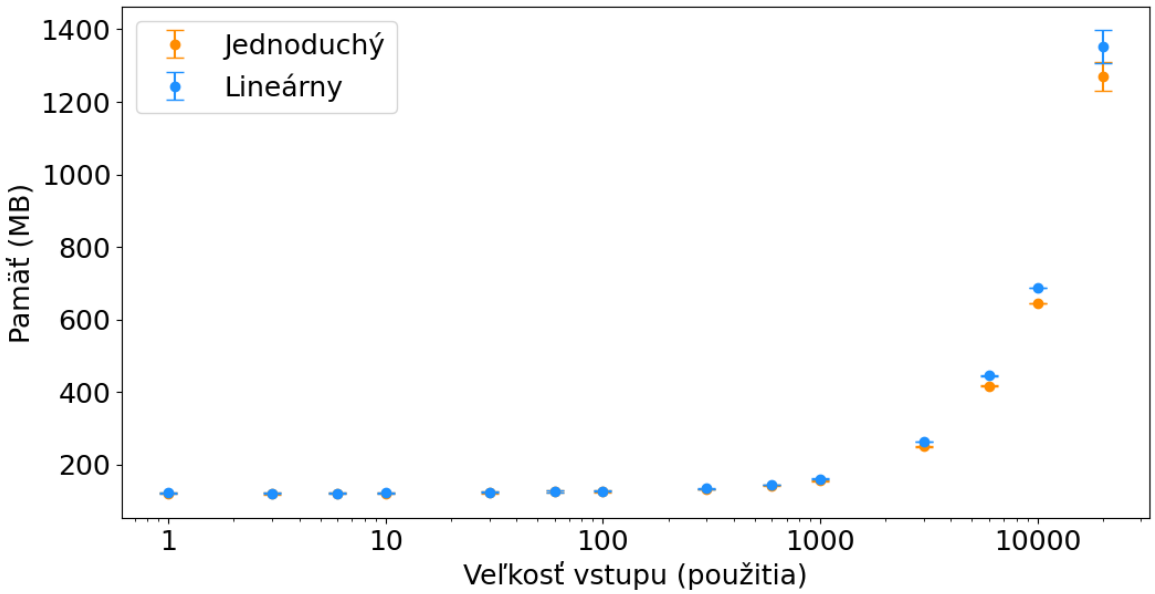
Trend výsledkov je dobre viditeľný na grafoch. Vzhľadom na široký rozsah veľkostí vstupov sú osi logaritmické, aby všetky výsledky boli dobre čitateľné. Grafy pamäťovej náročnosti majú os pamäte lineárnu, keďže pre malé vstupy je pamäť relatívne konštantná a nie je príliš vzdialená veľkým vstupom.

Tabuľka 6.7: Výsledky Rust optimalizácia 0

Veľkosť	Jednoduchý		Lineárny		Čas lin/jed
	čas (s)	pamäť (kB)	čas (s)	pamäť (kB)	
1	0,959 ± 0,072	120186	0,588 ± 0,012	121091	0,613
3	0,914 ± 0,030	119360	0,591 ± 0,007	120600	0,647
6	0,922 ± 0,031	120442	0,604 ± 0,008	120648	0,655
10	0,923 ± 0,031	119980	0,619 ± 0,015	121476	0,671
30	0,997 ± 0,043	123094	0,660 ± 0,008	123386	0,662
60	1,047 ± 0,046	124618	0,742 ± 0,012	125620	0,709
100	1,159 ± 0,059	125434	0,826 ± 0,009	126004	0,713
300	1,684 ± 0,016	131833	1,312 ± 0,007	133773	0,779
600	2,405 ± 0,033	142013	2,061 ± 0,030	144536	0,857
1000	3,335 ± 0,034	155339	3,067 ± 0,027	160028	0,920
3000	8,197 ± 0,282	250059	8,227 ± 0,139	262785	1,004
6000	15,230 ± 0,388	416372	17,176 ± 0,656	444713	1,128
10000	25,048 ± 1,114	644241	29,457 ± 2,100	688148	1,176
20000	55,464 ± 7,373	1270600	58,489 ± 5,677	1353214	1,055



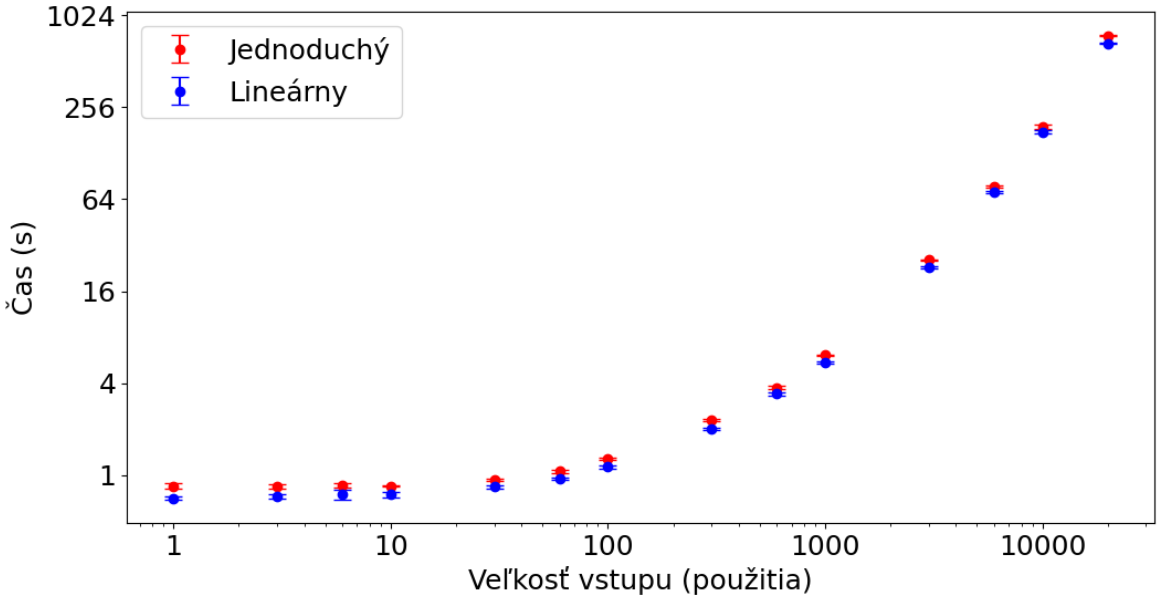
Obr. 6.32: Čas kompilácie pre Rust s optimalizáciou 0



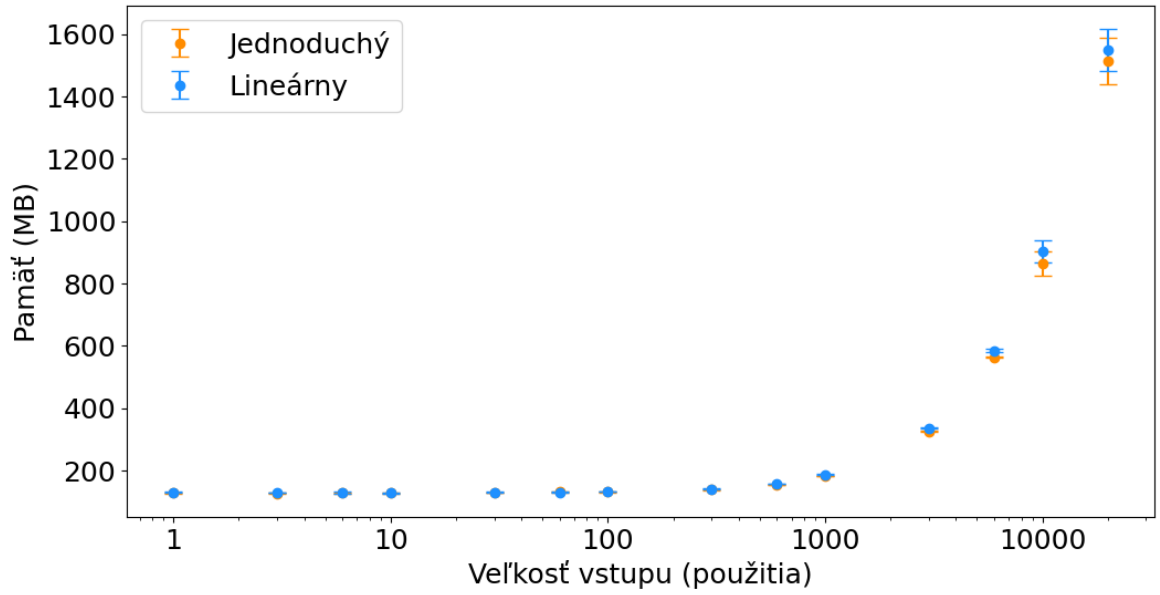
Obr. 6.33: Použitá pamäť kompilácie pre Rust s optimalizáciou 0

Tabuľka 6.8: Výsledky Rust optimalizácia 2

Veľkosť	Jednoduchý		Lineárny		Čas lin/jed
	čas (s)	pamäť (kB)	čas (s)	pamäť (kB)	
1	0,848 ± 0,039	128670	0,705 ± 0,016	130294	0,831
3	0,848 ± 0,028	127254	0,727 ± 0,021	129188	0,857
6	0,853 ± 0,032	128542	0,743 ± 0,051	129294	0,871
10	0,846 ± 0,011	128303	0,743 ± 0,026	128886	0,878
30	0,934 ± 0,016	129160	0,836 ± 0,022	130401	0,895
60	1,061 ± 0,027	131183	0,942 ± 0,019	130590	0,888
100	1,271 ± 0,021	131742	1,132 ± 0,025	132394	0,891
300	2,294 ± 0,051	138477	2,002 ± 0,029	140446	0,873
600	3,745 ± 0,099	154191	3,394 ± 0,093	156918	0,906
1000	6,081 ± 0,099	182157	5,445 ± 0,077	187599	0,895
3000	25,611 ± 0,336	325007	22,893 ± 0,332	335935	0,894
6000	77,803 ± 1,228	562906	71,171 ± 1,493	584777	0,915
10000	190,979 ± 5,364	864199	175,873 ± 4,526	904642	0,921
20000	754,382 ± 7,523	1513896	671,775 ± 8,721	1550046	0,890



Obr. 6.34: Čas kompilácie pre Rust s optimalizáciou 2



Obr. 6.35: Použitá pamäť kompilácie pre Rust s optimalizáciou 2

Záver

V práci sme sa zaoberali subštrukturálnymi systémami a ich praktickými implementáciami. Presnejšie sme identifikovali a demonštrovali implementácie v jazykoch C++, Rust a Haskell. Pre jazyky C++ a Rust sme tiež preskúmali dopad využitia subštrukturálnych elementov na výkon programov a kompilátorov.

Podrobnejšie sme preskúmali požiadavky na príležitostné slová. Ukázali sme rôzne použitia príležitostných slov, vrátane tých, ktoré spĺňajú požiadavky, ale štandardne sa neoznačujú ako príležitostné slová. Tiež sme spomenuli prípady, v ktorých neplatia všetky bežné požiadavky.

Preskúmali sme tiež ďalšie teoretické aj praktické aplikácie subštrukturálnych typových systémov. Z implementovaných systémov sme prezentovali knižnicu linear-base [5] pre jazyk Haskell, ktorá poskytuje nástroje pre prácu so súbormi, pričom využíva lineárne typy. Ukázali sme princíp, na základe ktorého bráni častým problémom pri práci so súbormi, aj praktickú demonštráciu jej použitia. Táto knižnica je zároveň príklad prakticky použiteľnej knižnice, s funkciami potrebnými pre spoluprácu so štandardnou nelineárnou knižnicou a jej postupné nahradenie.

Ako ďalšiu aplikáciu sme prezentovali návrh rozhrania pre mutovateľné polia [11], určeného pre jazyky s imutabilnými štruktúrami. Imutabilné štruktúry poskytujú mnohé výhody, ale efektívna implementácia niektorých algoritmov vyžaduje mutovateľné štruktúry. Preto je dôležité navrhnúť rozhranie pre ich integráciu tak, aby boli čo najviac zachované výhody imutabilný štruktúr. Zabezpečenie izolácie mutovateľných štruktúr pomocou lineárneho systému sme ukázali na prípade polí, ktoré sú najjednoduchším netriviálnym prípadom.

Tretím prezentovaným príkladom bola verifikácia uzavretosti, tiež nazývanej vodotesnosť, pre 3D objekty [13]. Tá je dôležitá napríklad pre 3D tlač, ale aj pre iné typy počítačom riadeného obrábania. V tomto prípade záruky prezentovanej metódy nie sú úplné, ale dokáže pokryť väčšiu množinu prípadov ako prístup bez subštrukturálnych typov. Zároveň je rozhranie pomerne priamočiare, vďaka čomu nekladie väčšie nároky na používateľa.

Ukázali sme tiež podobnosť medzi lineárnymi typovými systémami a kvantovými výpočtami. Vety o zákaze klonovania [17] a nemožnosti zmazania všeobecného kvantového stavu [14] patria medzi základné v oblasti. Lineárne typové systémy poskytujú

ich priamu paralelu v neplatnosti pravidiel o kontrakcii a oslabení.

Podrobnejšie sme popísali základné pravidlá [15], ktorých prítomnosť, respektíve neprítomnosť, určuje základný typ subštrukturálneho systému. Pre pravidlá o kontrakcii a oslabení sme tiež definovali alternatívne verzie, ktoré umožňujú aspoň v upravenej forme simulovať prítomnosť štandardnej verzie v systéme.

Na základe týchto pravidiel sme prezentovali typicky užitočné subštrukturálne systémy [15]. Prítomnosť všetkých troch pravidiel charakterizuje štandardné štruktúrované systémy. Pre každý zo systémov, ktoré vzniknú vynechaním aspoň jedného, sme popísali ich správanie vo všeobecnom prípade.

V zoradených systémoch neplatí žiadne z troch pravidiel. Následkom toho sa správajú ako zásobník na premenné, bez všeobecnej možnosti výmeny poradia, odstránenia vrchného prvku bez použitia alebo použitia vrchného prvku bez odstránenia. Lineárne systémy, v ktorých platí len pravidlo výmeny, umožňujú ľubovoľne meniť poradie použitia, ale podobne v nich platí podmienka práve jedného použitia premennej. Afínne systémy pridávajú k lineárnym pravidlo oslabenia. To umožňuje premennú nepoužiť, stále ale umožňuje vyjadriť podmienku zhora obmedzeného počtu použití. Naopak relevantné systémy pridávajú k lineárnym pravidlo kontrakcie. Tým umožňujú premennú použiť opakovane ľubovoľne veľa krát, ale kvôli neprítomnosti oslabenia vyjadrujú obmedzenie zdola, na nenulový počet použití.

Pre jazyky Haskell, C++ a Rust sme popísali implementované nástroje, ktoré umožňujú realizáciu subštrukturálnych typových systémov.

Jazyk Haskell priamo implementuje lineárne typy vo forme lineárnych funkcií. Funkcia je lineárna od svojho argumentu práve vtedy, ak jedno použitie výsledku spôsobí práve jedno použitie argumentu. Ako naznačuje zvolený názov, tento systém realizuje lineárne typy, čo sme demonštrovali analýzou platnosti pravidiel.

Jazyk Rust implementuje systém vlastníctva, ktorý je určený na zabezpečenie správy pamäte, konkrétne dealokácie. Pomocou analýzy platnosti pravidiel sme demonštrovali, že tento systém realizuje afínne typy, keďže nedokáže vynútiť použitie premennej.

Podobne jazyk C++ implementuje systém sémantiky presúvania, ktorý je tiež zavedený kvôli správe pamäte. V tomto prípade je ale motiváciou obmedzenie nadbytočného kopírovania. Podobne vlastníctvo v jazyku Rust, aj tento systém realizuje afínne typy, pretože takisto nedokáže vynútiť použitie premennej.

Následne sme prezentovali konkrétne implementácie príležitostných slov pomocou týchto nástrojov. Vo všetkých prípadoch je kľúčové, že jazyk umožňuje zverejniť len zvolené funkcie (a nie priamo premenné objektu), a tak vynútiť generovanie hodnoty počas konštrukcie inštancie. Vďaka tomu používateľ nedokáže vytvoriť novú inštanciu so zvolenou hodnotou, alebo inak vytvoriť kópiu. Subštrukturálne typové systémy následne umožnia obmedziť počet použití vygenerovanej inštancie.

Pre systémové jazyky C++ a Rust sme tiež sledovali dopad implementácií na vý-

kon. Jazyk Rust bol testovaný s kompilátorom rustc, pre jazyk C++ sme použili G++, Clang aj Microsoft Visual C++. Vo všetkých prípadoch sme zaznamenali veľmi dobrý výsledok z hľadiska výkonu za behu programu. Pri optimalizácii na dostatočnej úrovni boli všetky kompilátory schopné eliminovať obaľovacie štruktúry a tak dosiahnuť výsledný výkon ako pri priamom použití číselných typov, bez ochrany poskytnutej naším rozhraním.

Vplyv na čas kompilácie oproti verzii bez subštrukturálneho rozhrania bol vo väčšine prípadov tiež akceptovateľný. Pre rozumne malé počty použití rozhrania bol vplyv veľmi malý, jediný výrazný rozdiel bol pri veľkom počte použití rozhrania a vypnutej optimalizácii s kompilátorom MSVC.

Jazyky, ktoré sme opisovali, nie sú jediné, ktoré implementujú subštrukturálne typové systémy. Napríklad Wikipédia [10] má zoznam jedenástich jazykov, ktoré vraj podporujú lineárne alebo afínne typy. Okrem chýbajúcej citácie v tomto zozname tiež chýba jazyk C++, ktorý realizuje afínny typový systém, ako sme ukázali v príslušnej sekcii. Zoznam teda prezentuje deväť ďalších jazykov, a vzhľadom na protipríklad jeho úplnosti pravdepodobne existujú ďalšie.

Z toho vyplýva príležitosť preskúmať ďalšie jazyky, na zistenie ich vzťahu k subštrukturálnym typom. Dôležitou úlohou pre zvýšenie povedomia o nich je tiež implementácia knižníc, ktoré využijú subštrukturálne typy vo svojich rozhraniach. V práci sme demonštrovali, ktoré oblasti sú vhodné pre takéto knižnice, a pre niektoré jazyky aj príslušné techniky.

Tiež je dôležité pokračovať v skúmaní dopadov týchto knižníc na výkon. V práci sme sa zaoberali jednoduchým použitím, ktoré by malo pokrývať väčšinu praktických aplikácií knižnice. Jednou zo zaujímavých tém je hľadanie patologických prípadov, a ak existujú, ich náprava na úrovni kompilátorov. Za patologické prípady považujeme tie, ktoré vykazujú veľký rozdiel medzi výkonom subštrukturálneho a jednoduchého programu, pri relatívne malom programe. Podobne za potenciálne patologické správanie považujeme pozorované správanie kompilátora MSVC pri vypnutej optimalizácii, ktorý pre veľké súbory vykazoval rádový rozdiel vo výkone.

Literatúra

[1] Cpp reference: Constraints and concepts. https://en.cppreference.com/w/cpp/language/constraints. Accessed: 2024-04-09.

[2] Extra Clang tools: bugprone-use-after-move. https://clang.llvm.org/extra/clang-tidy/checks/bugprone/use-after-move.html. Accessed: 2024-04-09.

[3] Godbolt Compiler Explorer. https://godbolt.org/. Accessed: 2024-04-09.

[4] Hackage: base-4.19.1.0 Unsafe.Coerce. https://hackage.haskell.org/package/base-4.19.1.0/docs/Unsafe-Coerce.html. Accessed: 2024-05-08.

[5] Hackage: linear-base-0.4.0 System.IO.Resource.Linear. https://hackage.haskell.org/package/linear-base-0.4.0/docs/System-IO-Resource-Linear.html. Accessed: 2024-04-09.

[6] Haskell documentation: Linear types. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/linear_types.html. Accessed: 2024-04-09.

[7] Microsoft Learn /O options (optimize code). https://learn.microsoft.com/en-us/cpp/build/reference/o-options-optimize-code?view=msvc-170. Accessed: 2024-04-24.

[8] Microsoft Learn Use the Microsoft C++ toolset. https://learn.microsoft.com/en-us/cpp/build/building-on-the-command-line?view=msvc-170. Accessed: 2024-05-09.

[9] PS3 hacked through poor cryptography implementation. https://arstechnica.com/gaming/2010/12/ps3-hacked-through-poor-implementation-of-cryptography/. Accessed: 2024-04-09.

[10] Wikipedia Substructural type system. https://en.wikipedia.org/wiki/Substructural_type_system. Accessed: 2024-05-09.

[11] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order

polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017. Dostupné z https://arxiv.org/pdf/1710.09756.

[12] dr Ivan Čukić. Linear types can save the API. https://cukic.co/files/presentations/2021-07--linear-types-can-save-the-api--ivan-cukic--priml.pdf, 2021. Accessed: 2024-04-09.

[13] Samuel Gélineau. Making non-manifold models unrepresentable. https://www.spiria.com/en/blog/desktop-software/making-non-manifold-models-unrepresentable/, 2015. Accessed: 2024-04-09.

[14] Arun Kumar Pati and Samuel L Braunstein. Impossibility of deleting an unknown quantum state. *Nature*, 404(6774):164–165, 2000.

[15] Richard Ostertág. On the usefulness of linear types for correct nonce use enforcement during compile time, 2023.

[16] David Walker. *Substructural Type Systems*. MIT Press, 2002. in Pierce, Benjamin C. (ed.). Advanced Topics in Types and Programming Languages ISBN 0-262-16228-8.

[17] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, Oct 1982.

Príloha A: generátory testovacieho kódu

Generátory sú písané v čistom jazyku Java verzie 1.8 bez ďalších požiadaviek. Vygenerované súbory vkladajú do priečinka gen, ktorý treba vytvoriť. Po spustení vytvoria dve sady súborov a dva súbory s príkazmi. Súbory base_cislo.cpp resp. base_cislo.rs obsahujú jednoduchý kód a lin_cislo.cpp resp. lin_cislo.rs lineárny. Súbory cmd_base.txt a cmd_lin.txt obsahujú príkazy, ktoré boli použité pre testovací skript.