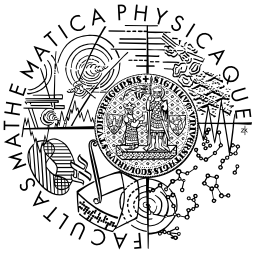


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Maximilian Kulikov

Emulátor zvukových syntezátorů

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. David Klusáček, Ph.D.
Studijní program: Informatika
Studijní obor: IOI

Praha 2022

UNIVERZITA KARLOVA

Matematicko-fyzikální fakulta

Ústav formální a aplikované lingvistiky Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jméno a příjmení: **Maximilian Kulikov**

Studijní program: **Informatika**

Studijní obor: **Obecná informatika**

Děkan fakulty Vám podle zákona č. 111/1998 Sb. určuje tuto bakalářskou práci:

Téma v jazyce práce: **Emulátor zvukových syntezátorů**

Téma práce v anglickém jazyce: **Sound Synthesizer Emulator**

Zásady pro vypracování:

Cílem práce je vytvoření polyfoního konfigurovatelného syntezátoru zvuku s nízkou latencí vhodnou pro profesionální použití. To znamená méně než 10 až 15 ms od stisku klávesy (nebo přijetí příkazu) do rozeznění zvoleného tónu. Průběh práce se bude řídit experimentálními výsledky různých přístupů k řešení, kde se bude latence a další parametry měřit.

Uživatelské rozhraní:

Syntezátor bude rozumět konfiguračnímu jazyku, kterým uživatel nastaví celkovou strukturu syntezátoru. Tento proces by měl být podobný analogovému
zapojení syntezátorů. Konfigurace proběhne těsně po spuštení programu a dále budou vstupem MIDI příkazy, například z připojených MIDI kláves nebo ovládacích pultů.

Platforma:

Program bude vyvíjen pro operační systém Windows. Pro komunikaci se zvukovou kartou bude použito rozhraní ASIO od firmy Steinberg, pro jeho nízkou latenci.

Seznam odborné literatury:

[1] R.W.Hamming. Digital Filters. Prentice-Hall, New Jersey, 1977

[2] Jiří Jan: Číslicová filtrace a restaurace signálů, VUTIUM, 2002

Vedoucí bakalářské práce: **Mgr. Klusáček David, Ph.D.**

Navrhovaní oponenti:
Konzultanti:
Datum zadání bakalářské práce: 24.9.2020

Termín odevzdání bakalářské práce: dle harmonogramu příslušného akademického roku

.....
Vedoucí katedry

.....
29 Děkan9

V Praze dne 7.1.2022

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Děkuji Mgr. Davidovi Klusáčkovi, Ph.D. za veškerou pomoc s mnohými teoretickými i praktickými otázkami v průběhu práce a nasměrování k výsledné podobě.

Název práce: Emulátor zvukových syntezátorů

Autor: Maximilian Kulikov

Ústav: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. David Klusáček, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Nástroj na tvoření emulátorů zvukových syntezátorů. Základem práce je imperativní programovací jazyk Cynth popisující signály tvořící výsledný zvuk. Kód v jazyce Cynth se přeloží do jazyka C pro následující slinkování s programem řídícím GUI a MIDI vstupní ovládání a výstupní monitorování a napojení zvukové karty. Mezikrok s překladem do jazyka C přináší výhodu z využití optimalizací překladače C. Jazyk Cynth je omezený tak, aby za běhu nedocházelo k žádné dynamické alokaci, ale zároveň umožňuje komplexní programování za překladu a práci se staticky alokovanými datovými strukturami určenými k expresivnímu popisu signálů.

Klíčová slova: syntezátor emulátor programovací jazyk

Title: Sound Synthesizer Emulator

Author: Maximilian Kulikov

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. David Klusáček, Ph.D., Institute of Formal and Applied Linguistics

Abstract: A tool for creation of emulators of audio synthesizers. The base of the work is an imperative programming language Cynth that describes signals of the resulting sound. Cynth code is translated into C code for further linkage with a program that controls GUI and MIDI input controls and output monitoring and connection with a sound card. The intermediate step of translation into C allows taking advantage of the C compiler optimizations. The Cynth language is restricted in a way that eliminates any dynamic allocations at run-time while allowing complex compile-time programming and working with statically allocated data structures for expressive description of signals.

Keywords: synthesizer emulator programming language

Obsah

Úvod

Cílem této práce je navrhnout a implementovat nástroj na tvoření emulátorů analogových či digitálních zvukových syntezátorů. Základem je konfigurační jazyk Cynth, který umožní popsat signály a deklarovat vstupy a výstupy. Kód v jazyce Cynth se přeloží do jazyka C pro následující statické slinkování s předkompilovaným řídícím programem. Tento řídící program má na starosti komunikaci se zvukovou kartou a vykreslení GUI a napojení MIDI k ovládání a monitorování.

Práce nejprve uvede do problematiky syntezátorů a *DSP* (digital signal processing) obecně a také do tématu analogových syntezátorů a jejich digitální emulaci. V návaznosti na důležité aspekty analogových i digitálních syntezátorů bude navržen jazyk Cynth. Důležitá rozhodnutí v návrhu budou odůvodněna.

Dále budou uvedeny praktické příklady použití jazyka. Vlastnosti jazyka jsou v práci dále kompletně specifikovány. Nejde však o formální specifikaci jazyka. Krátce bude popsána implementace překladače. Nakonec se zdokumentují provedená měření výkonu výsledných programů.

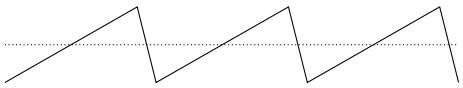
1. Úvod do problematiky

1.1 Analogové syntezátory

Analogové syntezátory využívají napětí v elektrických obvodech k reprezentaci zvukového signálu. Kolísání napětí se ovlivňuje různými komponentami obvodu. Výsledné napětí na vodičích se vyvede do výstupu, např. do reproduktoru, nebo dalších tzv. *modulů*.

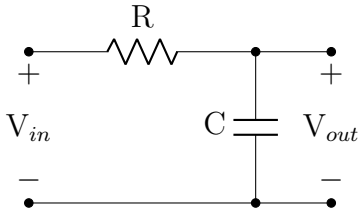
Jednoduchou sinusoidu lze vygenerovat např. jednoduchým *LC oscilátorem* složeným z cívky a kondenzátoru. Takto vygenerované sinusoidy z různých oscilátorů lze sčítat do komplexnějšího, harmonicky bohatějšího signálu. Takový postup se označuje jako *aditivní syntéza*.

Mezi prvními syntezátory 60. a 70. let byla nejběžnější tzv. *subtraktivní syntéza* (Pekonen a Välimäki, 2011). Nejprve se vygeneruje relativně komplexní signál, který je už složený z mnoha různě škálovaných a posunutých sinusoid, aby zahrnoval velké množství frekvencí. Takový signál se vygeneruje např. *relaxačním oscilátorem*, který využívá zapojení kondenzátorů a tranzistorů k prudkému poklesu či nárůstu napětí. Běžně se využívá třeba *čtvercová* nebo *pilová vlna* znázorněná v grafu 1.1. Na takové komplexní signály se následně aplikují *filtry*, které odebírají specifické frekvence ze signálu.



Obrázek 1.1: Pilová vlna

Jednoduchým příkladem analogového filtru je tzv. *RC filtr*, který je v základní podobě složen z rezistoru a kondenzátoru. Obvod takového filtru je zobrazen v diagramu 1.2.



Obrázek 1.2: RC filtr

Dalším zajímavým přístupem k syntéze zvukového signálu je tzv. *frekvenční modulace*. Základní myšlenkou je použití měnících se hodnot signálu k modulaci vlastností jiného signálu. Např. při použití nízkofrekvenčního oscilátoru s nízkou amplitudou k modulaci frekvence signálu lze napodobit tzv. *vibráto* efekt, který lze uslyšet třeba na některých strunných nástrojích. Při zvýšení frekvence modulačního oscilátoru na dostatečně vysoké hodnoty lze ale také dosáhnout velmi

neobvyklého a harmonicky zajímavého zvuku. Přitom taková syntéza nevyžaduje žádné použití filtrů.

FM syntéza se stala velmi populární i v novějších digitálních syntezátorech (Pekonen a Välimäki, 2011). Použití filtrů v digitálních syntezátorech totiž vyžaduje poměrně komplexní výpočty, které mohou být prakticky neproveditelné při práci se signálem v reálném čase na výpočetně slabším hardwaru.

1.2 Digitální syntezátory

Běžně se pro digitální reprezentaci zvukového signálu používá tzv *pulzně kódová modulace*. Signál se tzv. *vzorkuje*. Zaznamenávají se jen pravidelných časových intervalech vybrané výchylky, zvané *vzorky*. Tyto vzorky se také musí *kvantovat*, tedy zaznamenat jen s konečnou přesností. Taková reprezentace je obecně ztrátová. Rekonstrukce reprezentovaného signálu je možná při dodržení omezení, která určuje *Nyquistův–Shannonův vzorkovací teorém*. Vzorkovací frekvence musí být větší než dvojnásobek maximální frekvence vzorkovaného signálu (Zölzer, 2008, 3.1.1 Nyquist Sampling).

Často se předpokládá, že člověk běžně slyší frekvence v rozmezí od 20 Hz do 20 kHz. Přitom různé studie potvrzují, že někteří slyší i vyšší frekvence až do 26 kHz (Ashihara a kol., 2006). Standardně volenou hodnotou vzorkovací frekvence je např. 41,1 kHz, což odpovídá standardnímu CD formátu. Mnozí zvukoví inženýři však preferují vyšší vzorkovací frekvence, např. 96 kHz. Jedním z důvodů jsou praktická omezení při filtrování vysokých frekvencí signálu před jeho vzorkováním. Aby bylo možné signál rekonstruovat po vzorkování, je třeba se předem zbavit frekvencí od určitého prahu, což ale v praxi nelze provést přesně. Vyfiltrování vyšších frekvencí vždy nějakým způsobem ovlivní i ty nižší.

Kromě vzorkovací frekvence je důležitý i rozsah možných hodnot kvantizovaných vzorků. Pro dostatečně kvalitní rekonstrukci signálu se doporučuje použít alespoň 20 bitové vzorky (Burgel a kol., 2001). 20 bitové vzorky mohou být vhodné pro záznam zvukových souborů, ale při zpracování signálu v reálném čase je důležitější volit vzorky, se kterými je schopen procesor jednoduše pracovat, než šetřit místem. Proto bude na moderních procesorech rozumnější zvolit třeba 32 nebo 64 bitové vzorky.

Velkou výhodou digitálních syntezátorů je jejich cena. Digitální syntezátory je dnes v mnohých případech možné jednoduše spustit na osobních počítačích. Tzv. *digital audio workstation* (DAW) programy umožňují spolu s technologiemi jako např. *Virtual Studio Technology* (VST) spojovat dohromady různé pluginy implementující komplexní syntezátory nebo jejich jednoduché moduly (Hummel, 2016).

Charakteristický zvuk analogových subtraktivních syntezátorů je i přes výhody digitálních syntezátorů stále lákavý pro moderní hudebníky i posluchače. V devadesátých letech se objevily první digitální syntezátory, které napodobovaly právě princip subtraktivní syntézy starších analogových syntezátorů (Pekonen a Välimäki, 2011).

1.3 Digitální emulace analogových syntezátorů

Pro filtr zobrazený v diagramu 1.2 lze z Kirchhoffova zákonu o napětích odvodit vztah (1.1) a jeho diskretizací vznikne (1.2). V_{in}, resp. V_{out}, reprezentuje vstupní, resp. výstupní analogový signál. x a y jsou odpovídající diskrétní signály, které jsou vzorkované v pravidelných intervalech Δ_t. Dalšími úpravami lze získat vztah (1.3), který popisuje, jak k výsledku přispívá vstupní signál a navíc *zpětná vazba* z dosavadního výstupního signálu. Z takového popisu již lze odvodit implementaci digitálního filtru emulujícího výše uvedený analogový RC filtr.

$$V_{in}(t) - V_{out}(t) = RC\frac{\mathrm{d}V_{out}}{\mathrm{d}t} \tag{1.1}$$

$$x_i - y_i = RC\frac{y_i - y_{i-1}}{\Delta_t} \tag{1.2}$$

$$y_i = \alpha x_i + (1-\alpha)y_{i-1} \text{ kde } \alpha = \frac{\Delta_t}{RC + \Delta_t} \tag{1.3}$$

Digitální filtr lze obecně vyjádřit lineární kombinací vzorků vstupního a případně i výsledného výstupního signálu. Nerekurzivní filtry lze spočítat jen ze vzorků vstupního signálu, zatímco rekurzivní filtry ve výpočtu využívají i předchozí vzorky výstupního signálu. Filtr s koeficienty $a_0, \ldots, a_p$ a (v případě rekurzivního filtru) zpětnovazebnými koeficienty $a_1, \ldots, a_q$ je dán výrazem (1.4), který vyjadřuje hodnotu n-tého vzorku výstupního signálu y v závislosti na vstupním signálu x:

$$y_n = \frac{1}{a_0} * \left(\sum_{i=0}^{p} b_i x[n-i] - \sum_{i=1}^{q} a_i y[n-1] \right) \tag{1.4}$$

Je také důležité podotknout, že takový filtr, je tzv. *kauzální*. To znamená, že výsledný signál neovlivní vzorky z budoucnosti, tedy vzorky x_m pro $m > n$ z (1.4). Nekauzální filtry lze aplikovat např. na zvukovou nahrávku, ale jejich použití není praktické v syntezátorech, jelikož jakýkoliv zásah do budoucnosti by způsobil zpoždění signálu.

1.4 Výpočty v reálném čase

Tato práce se zaměřuje na syntézu potenciálně nekonečného signálu, který se mění v reakci na uživatelský vstup v reálném čase. Je tedy třeba zaručit dostatečně rychlou reakci na vstup. Při zpracování signálů v reálném čase je kromě rychlosti výpočtů důležité zohlednit, kde může vzniknout zpoždění, jak tomu předejít, a hlavně jak zaručit, že se vše stihne spočítat včas.

Práce se zvukovými signály syntezátoru i pro umělecké potřeby v některých aspektech odpovídá definicím hard nebo firm real-time požadavků. Výsledek nespočítaný včas je prakticky nepoužitelný. Nezávisí na tom lidský život nebo třeba závažné poškození hardwaru, ale chybějící, nebo v nesprávném pořadí sestavené

vzorky obecně nelze v reálném čase rozumně opravit a mohou způsobit lidskému uchu velmi nepříjemné zvuky, které značně ovlivní vnímaný výsledek.

1.4.1 Převod na spektrum

Při offline zpracování konečného signálu je možné pomocí DFT převést signál na odpovídající spektrum, pozměnit frekvenční komponenty jeho vynásobením požadovanou funkcí a výsledek převést zpátky na signál. Výstup programu musí být signál, vzorky kterého se předávají zvukové kartě. V případě zapojení s jinými programy (např. v podobě VST pluginu v DAW softwaru) bude i vstup programu signál. Pro práci s real-time signálem výše popsaným způsobem by tedy bylo třeba vytvořit spektrogram *krátkodobou DFT*, tedy DFT aplikovanou postupně po konečných úsecích signálu, pro zjištění vývoje spekter v čase. Takový spektrogram by pak bylo třeba zpět převést inverzní DFT na signál. To způsobí nezanedbatelné zpoždění.

Přitom je ale možné provádět ekvivalentní operace i na signálech bez mezikroku s převodem na spektrum. Místo násobení spekter se provede konvoluce signálů a naopak (konvoluční věta). Není tedy nutné pracovat se spektry, které by jen zbytečně způsobovaly zpoždění. Proto v návrhu postupu výpočtu výsledného signálu dále nebude zvažována práce se spektry. Všechny operace budou probíhat na signálech.

Jakákoliv práce s frekvencí signálu, ať už jde o převod na spektrum a zpět, či o konvoluci s impulzní odezvou filtru, vyžaduje znalost nějaké souvislé části signálu. Jednotlivé vzorky nestačí. Pro účely běžných filtrů ale postačí zásah do minulosti signálu. Jakýkoliv zásah do budoucnosti signálu by způsobil zpoždění.

1.4.2 Hromadné výpočty

Zpracování souvislých úseků signálů je třeba zvážit nejen kvůli filtrům. Alternativou výpočtům po jednotlivých vzorcích signálu jsou výpočty po tzv. *bufferech*. Výpočet se provede pro daný počet souvislých vzorků a tyto výsledky se předají další komponentě, třeba zvukové kartě, nebo dalšímu softwaru. Hromadný výpočet po bufferech má několik výhod (Welch a kol., 2016).

U jednotlivých výpočtů je v běžných operačních systémech náročné zaručit maximální čas výpočtu. Pokud je ale časové omezení stanoveno pro výpočet celé řady vzorků, neobvykle dlouho trvající výpočet jednoho vzorku může zachránit kratší doba výpočtu ostatních vzorků. Je tak tedy jednodušší zaručit správné načasování vzorků. Stále však nelze plně spoléhat na předpokládanou průměrnou dobu výpočtu a je třeba zohledňovat horní časový odhad. Další výhodou je lokalita zapisovaných dat, což může zlepšit čas přístupu k paměti.

Délka bufferu určuje vzniklé zpoždění. Reakce na vstup, který proběhl během zpracování bufferu, se totiž projeví až po zápisu všech vzorků do bufferu. Kratší buffery nebo výpočty po jednotlivých vzorcích mají tedy naopak výhodu v tom, že způsobují menší zpoždění. Při návrhu řešení tedy bude uvážen kompromis mezi výpočty po bufferech a po vzorcích.

1.4.3 Vnímané zpoždění

Se zpožděným signálem je velmi obtížné udržet správný rytmus při hraní na hudební nástroj, tedy pro umělecké účely je dostatečně nízké zpoždění kriticky důležitým požadavkem. Člověk je dle některých provedených experimentů (Jack a kol., 2018) schopen vnímat negativní vliv zpoždění signálu při hře na hudební nástroj už těsně nad 10 ms. Je třeba také pomyslet na to, že se běžně syntezátory zapojují dohromady, třeba pomocí výše zmíněné technologie VST. V běžícím DAW programu tak může být použito běžně několik desítek jednotlivých virtuálních nástrojů. Běžný syntezátor by tedy rozhodně neměl přesáhnout hranici několika milisekund.

1.4.4 Dynamická alokace

Jak již bylo zmíněno, v běžných operačních systémech, jako Windows a Linux, není možné vynutit real-time požadavky. Konkrétně Windows umožňuje spustit proces v *real-time priority* nebo *high priority* režimu, což se může přiblížit běhu v real-time systému (Microsoft). Komě plánování procesů může výsledky také ovlivnit např. změna frekvence procesoru, či přechod na jiné jádro, ale i stránkování paměti. V běžném osobním počítači je tedy náročné dosáhnout opravdu real-time běhu programu.

V embedded systému s dedikovaným hardwarem to ale může být jednodušší. Běžně se vyrábí digitální syntezátory i v podobě fyzických přístrojů s vlastním hardwarem. Bylo by tedy vhodné, kdyby byl výsledný program použitelný i v embedded prostředí. To by znamenalo především rozumné omezení dynamické alokace paměti. Dynamická alokace může být nevhodná v embedded systémech kvůli omezenější paměti na rozdíl od osobních počítačů, ale i pro real-time aplikace obecně kvůli její časové náročnosti v nejhorším případě.

Dynamická alokace může být dostatečně rychlá v průměrném případě, ale horní hranice její časové náročnosti může být naprosto nepřijatelná pro real-time aplikace. Při dynamické alokaci totiž může dojít k systémovým voláním, která obecně nemusí být deterministická. Při stránkování paměti může dokonce dojít k zápisu na disk. Obvyklým řešení je předalokace horního odhadu potřebné paměti a následné použití vhodných algoritmů dynamické alokace na této paměti (Shen a kol.). To však má nevýhodu v tom, že pesimistický odhad využije více paměti, než je třeba, zatímco optimistický odhad způsobí to, že bude třeba opět alokovat další velký blok paměti.

V tomto projektu přitom není dynamická alokace nutná. V běžných syntezátorech není nutnost za běhu vytvářet nové komponenty. U analogových syntezátorů to ani není možné. Mělo by samozřejmě být možné za běhu připojit další syntezátor, ale k tomu stačí mít staticky alokován prostor pro jejich vzájemnou komunikaci. Alokace komponent takového připojeného syntezátoru může také proběhnout staticky při jeho spuštění. Je tedy rozumnější se dynamické alokaci vyhnout úplně, než vybírat kompromis mezi problematickým časem a nadbytečným využitím paměti.

1.5 Cíle

Cílem práce je navrhnout a implementovat nástroj na tvoření emulátorů analogových či digitálních zvukových syntezátorů. Konkrétně půjde o překladač konfiguračního jazyka, řídící program na zpracování vstupu a komunikaci se zvukovou kartou a utility na sestavení výsledku. Celý projekt i samotný konfigurační jazyk je označován pracovním názvem *Cynth*.

Výsledek by měl být použitelný jako realtime aplikace s dostatečně nízkou latencí a výhodou bude použitelnost v embedded prostředí. Popis syntezátoru by měl být vhodný pro emulaci analogových i digitálních syntezátorů.

1.5.1 Konfugurační jazyk

Konfigurační jazyk Cynth by měl uživateli umožnit jednoduchý popis signálů syntezátoru a vztahů mezi nimi s využitím vstupních dat. Tento jazyk má jen popisovat signály. Jeho účelem není popsat grafické rozhraní, ani napojení hardwaru.

Jazyk je zde sice označen jako konfigurační, ale nejde o ryze konfigurační jazyk typu JSON. V průběhu návrhu se totiž ukázalo, že k praktickému použití je třeba relativně komplexní jazyk. Některé vlsatnosti jazyka se naopak ukázaly být nepříliš nutnými a naopak implementačně zbytečně komplikovanými.

Rozumným řešením by mohla být i knihovna, namísto nového jazyka. Kdyby šlo o knihovnu, byla by omezena cílovým jazykem. Výsledek musí být optimalizovaný na velmi specifické programy, s realtime požadavky a nízkou latencí. Dynamické scriptovací jazyky typu Python, JavaScript apod. by tedy nebyly vhodné. Na druhou stranu více nízkoúrovňový jazyk typu C, C++ apod. by nebyl vhodný pro méně zkušené programátory. Jazyky typu C# a Java by mohly umožnit vhodnou práci s pamětí i jednodušší interface pro uživatele, ale stále zde má uživatel možnost velmi jednoduše provést nějaké nevhodné, zbytečně náročné operace. Rozumným řešením by ale mohl být nějaký kompilovaný čistě funkcionální jazyk jako třeba Haskell, který provádí dost silné optimalizace.

V ideálním případě by byl jazyk jednoduše použitelný i pro uživatele bez předchozích zkušeností s programováním. Po osobní zkušenosti s několika jazyky, které tak byly původně navrženy (např. Lua a PHP) jsem se však rozhodl touto cestou nevydávat. Totiž právě ty vlastnosti, co byly navrženy pro neprogramátory, dodnes často komplikují použitelnost a další vývoj těchto jazyků.

1.5.2 Kompilovaný jazyk

Jazyk by měl být kompilovaný, aby bylo možné kód dostatečně optimalizovat. Implementovat optimalizace samotné by však bylo nad rámec této bakalářské práce. Proto se jazyk Cynth přeloží do jazyka C a následně se využije optimalizací již existujících překladačů. Transpiler Cynthu by pak měl jen zařídit vygenerování dobře optimalizovatelného C kódu.

1.5.3 Jazyk vhodný na popis signálu a komponent syntezátoru

Jazyk by měl relativně jednoduše a elegantně popisovat signály a komponenty syntezátoru. Půjde o syntaxi, ale i o sémantické vlastnosti, které umožní uživateli dostatečně abstraktní popis, který bude možné přeložit do dostatečně efektivního kódu podle ostatních požadavků.

1.5.4 Výpočetní program

Výsledkem transpilace Cynthu a kompilace vygenerovaného C kódu bude tzv. *výpočetní program*. Tento program provádí výpočet dalších vzorků výstupních signálů v pořadí. Dále alokuje globální data syntezátoru, konkrétně ovládací hodnoty, ze kterých čte, a monitorovací hodnoty, do kterých zapisuje. Výpočetní program může být implementován multiplatformně, jelikož nemusí komunikovat přímo s operačním systémem ani s hardwarem. Navíc díky omezení dynamické alokace může být využitelný i v embedded systémech.

1.5.5 Řídící program

Vše ostatní bude vykonávat tzv. *řídící program*. Konkrétně jde o komunikaci se zvukovou kartou, reakce na MIDI signály či grafické ovládání a vykreslení grafického rozhraní. Řídící program se musí sestavit v závislosti na platformě. Součástí práce bude konfigurovatelný řídící program, který podporuje Windows a zvukové karty kompatibilní s technologií ASIO. Jeho konfigurace by měla určit detaily MIDI či grafického ovládání.

Takové rozdělení výpočetní a ovládací části umožňuje jakousi modularitu. Uživatel může napsat vlastní řídící program, který např. bude komunikovat s jinými zvukovými kartami, nebo využije jinou technologii na vykreslení GUI.

Ideálně by měl být řídící program relativně obecný a jednoduše rozšiřitelný, aby uživatel nemusel vytvářet kompletně vlastní řídící program pro svůj systém (např. pro specifický operační systém nebo zvukovou kartu).

1.6 Existující řešení

1.6.1 FAUST

Jak již bylo zmíněno, čistě funkcionální přístup může být vhodný pro tuto aplikaci. Umožňuje totiž kromě elegantního popisu signálů hlavně možnost silných optimalizací, které jsou v mnohých případech schopny vygenerovat dost výkonný kód.

Překladač Faustu transpiluje Faust kód do C++ kódu a následně využívá existující překladač C++. Provádí např. optimalizace smyček vygenerovaného C++ kódu, aby se využilo automatické vektorizace překladače C++, tedy převodu na SIMD instrukce (Orlarey a kol., 2009, Code Generation). Kromě snahy vygenerovat dobře zoptimalizovatelný C++ kód se provádí i optimalizace ještě před generací C++ kódu, které jsou možné jen v čistě funkcionálních jazycích. Také

je zde kladen důraz na staticky alokovanou paměť (Orlarey a kol., 2009, Introduction).

Překladač Cynthu také využívá transpilaci do jiného jazyka. Výhodou překladu do C oproti C++ může být rychlejší následující kompilace do strojového kódu.

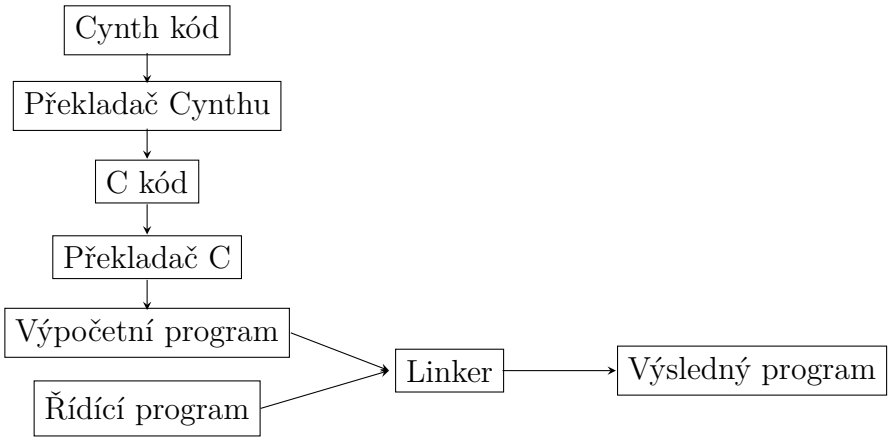
2. Návrh

Jak již bylo uvedeno v sekci 1.5, výsledný program bude rozdělen do dvou částí. Půjde o *výpočetní program* a *řídící program*.

Výpočetní program je výsledkem překladu konfiguračního jazyka Cynth. Při návrhu bude třeba zohlednit vlastnosti jazyka Cynth a také rozhraní výsledného přeloženého programu.

Řídící program představuje jen jednu z možností, jak výpočetní program použít. Je možné, že výsledek bude dost obecný a rozšířitelný na jiné platformy, ale také se může ukázat, že bude pro některé uživatele výhodnější používat jen přeložený výpočetní program ve vlastní aplikaci.

Implementace bude rozdělena na tři komponenty. Půjde o překladač Cynthu, řídící program a rozhraní výpočetního programu. Překladač a řídící program budou implementovány v C++. Překladač se přeloží do kompletního spustitelného souboru, zatímco řídící program se přeloží do statické či dynamické knihovny. Z výpočetního programu bude připraveno jen jeho rozhraní v podobě header souboru. Uživatel pak použije překladač Cynthu spolu s překladačem C k vytvoření implementace výpočetního programu. Výsledek se slinkuje do jednoho spustitelného programu a k němu se případně připojí potřebné dynamické knihovny. Tento postup znázorňuje diagram 2.1



Obrázek 2.1: Sestavení výsledného programu

Překladač nemusí záviset na platformě. Kód by tedy měl být psán ve standardním C++ bez použití knihoven závislých na platformě. Bude vhodné použít např. Bison a Flex (případně nějaké alternativy) pro generaci parseru.

Řídící program naopak bude silně závislý na platformě. Bude muset komunikovat se zvukovou kartou, vykreslovat grafické rozhraní a přijímat MIDI signály.

2.1 Řídící program

Komunikace se zvukovou kartou bude probíhat pomocí knihovny ASIO SDK, která komunikuje přímo se zvukovou kartou. Alternativou by mohlo být využití nějaké multiplatformní knihovny, která by v případě dostupnosti využila právě ASIO. Pro lepší možnost detailně přizpůsobit způsob komunikace s knihovnou bez zbytečného overheadu by ale naopak mohlo být vhodnější použít jen ASIO a případně implementovat další možná rozhraní (např. nativní WASAPI na Windows) ručně. ASIO je proprietární technologie firmy Steinberg pro ovladače zvukových karet. Použití ASIO je bezplatné, ale v případě komerčního šíření programu je třeba dát souhlas s výčtem některých omezení. ASIO bylo zvoleno pro jeho nízkou latenci a rozšířenost ve spojení s technologií VST (Steinberg).

Pro komunikaci s MIDI periferiemi bude možné použít např. JACK, nebo nativní rozhraní Windows Multimedia (WINMM).

Řídící program by měl být do určité míry konfigurovatelný. Jazyk Cynth totiž popíše jen signály syntezátoru a uvede, jeho vstupy a výstupy. Ovládání a monitorování těchto vstupů a výstupů už nespadají do jeho rozsahu. Řídící program by tedy mohl za běhu (při spuštění) podle dané konfigurace vykreslit GUI ovládání a napojit MIDI periferie. U MIDI periferií by bylo ideální, kdyby bylo možné za běhu identifikovat části MIDI nástroje a ty zapojit dle potřeby. Signály kláves (imitující klaviaturu klavíru) jsou jednoznačně dány pro standardní MIDI nástroje. Ostatní ovládací prvky (různá tlačítka a posuvníky) jsou ale očíslována na každém nástroji po svém.

Pro konfiguraci řídícího programu by mohl být vhodný formát YAML. Zahrnuje totiž velmi populární formát JSON a rozšiřuje ho o pohodlnější a potenciálně i vizuálně přehlednější syntaxi. Rozdělení konfigurace řídícího programu (YAML) a výpočetního programu (Cynth) dává smysl kvůli separaci těchto programů, ale pro uživatele by mohlo být pohodlnější uvést popis ovládání spolu s popisem signálů. Možným rozšířením překladače by bylo přečtení speciálních komentářů u deklarací ovládacích prvků obsahujících úryvky YAML konfigurace. Z těchto úryvků by se pak sestavila výsledná konfigurace v případě použití kompletního procesu překladu a sestavení se standardním řídícím programem.

Jednou z variant pro grafické rozhraní by bylo použití webových technologií pro generaci nativní grafiky. Šlo by o popis rozložení ovládacích prvků pomocí HTML, jejich stylistické podoby v CSS a reakce na ovládání v JS. Nevýhodou je potenciální overhead kvůli zpracování dynamických JS hodnot. Grafické rozhraní ale nemusí mít natolik nízkou latenci. Přitom výhodou je možnost načíst dodatečnou konfiguraci v HTML, CSS a JS popisující vizuální stránku výsledného programu. Tyto jazyky jsou totiž dnes velmi populární. Uživatel ovládající základy těchto tří jazyků bude schopen velmi jednoduše popsat kompletně vzhled i chování ovládacích prvků a naopak i nezkušený uživatel nebude mít problém pomocí CSS napsat např. jednoduchý popis barvy pozadí nebo ovládacích prvků.

2.2 Konfigurační jazyk

Tato sekce uvede a odůvodní rozhodnutí provedená při návrhu jazyka. Konkrétní konstrukce jazyka zde budou popsány jen okrajově a neformálně. Většinou půjde o srovnání s jinými jazyky, kterými byl konstrukt inspirován. Princip implementace je promyšlen jen povrchově a je možné, že se při implementaci objeví nutnost některé navržené vlastnosti přizpůsobit praktičtější implementaci nebo vynechat.

2.2.1 Imperativní prvky

Základní struktura a syntaktická podoba programu v Cynthu je navržena takovým způsobem, aby připomínala jazyk C nebo běžné jazyky jím inspirované. Pro uživatele, kteří mají základní zkušenosti s programováním v takových jazycích, by tedy nemělo být problém se zorientovat v základech jazyka Cynth. Navíc imperativní návrh jazyka umožňuje relativně přímočarý překlad do C.

Jazyk bude obsahovat řídící struktury pro sekvenční vykonávání příkazů, pro větvení i pro cykly jako v běžných imperativních jazycích. Konkrétně jde o struktury pojmenované `if`, `when`, `while` a `for`. Jedním rozdílem od C je řídící struktura `when`, která odpovídá struktuře `if` v C bez `else` větve. To umožňuje jednoznačný popis jazyka jednodušší gramatikou bez doplňujících pravidel. Dále struktura `for` odpovídá spíše konstruktu `foreach` např. v jazyce C#, který reprezentuje iteraci přes pole, nebo přes n-tici prvků z několika polí. Operace s poli jsou totiž pro DSP důležité a přitom takový konstrukt je relativně jednoduše přeložitelný do C `for` cyklu.

Syntakticky jsou příkazy odděleny středníky, nikoliv ukončeny. Pro jednodušší přechod od jazyků, kde jde o ukončení středníkem, jsou explicitně povoleny středníky navíc, kde by je programátor očekával v C.

Následující úryvek demonstruje syntaxi některých řídících struktur a minimální potřebné středníky.

```
{
    a;
    b;
    if (x)
        c
    else
        d;
    when (y)
        e
}
```

Deklarace a definice proměnných i přiřazení do nich je také inspirována jazykem C. Rozdílné je jen to, že při deklaraci bez definice je vždy jednoznačně dáno, jaká hodnota se implicitně do proměnné přiřadí. Nedefinované chování by totiž mohlo být nepříznivé pro méně zkušené uživatele. Přiřazení je na rozdíl od C jen příkazem. To zjednodušuje gramatiku vzhledem k ostatním, komplexnějším konstruktům. Princip scopu je také prakticky identicky převzat z C.

```
Int a = 1;
Int b;
a = b;
```

2.2.2 Funkcionální prvky

Pro pohodlný popis signálů byly v jazyce navrženy některé prvky inspirované funkcionálními jazyky. Byl kladen důraz nejen na to, aby byly funkce first-class objekty, ale i na možnost sestavování komplexních výrazů, které nahradí potřebu příkazů. Výrazy ale přitom mohou mít side effecty, jako běžné imperativní jazyky. Tyto side effecty ale mohou proběhnout jen v rámci funkce, nebo na speciálních typech, které jsou k tomu přímo určeny. Proměnné mohou být explicitně označeny jako imutabilní použitím keywordu `const` za typem.

Výrazy jsou podmnožinou příkazů. Přitom řídící struktura *block*, umožňuje sloučit více příkazů do výrazu. Block zahrnuje funkcionalitu, kterou by uživatel očekával od C blocku (složeného příkazu), ale poskytuje i další funkcionalitu při použití v roli výrazu. Jeho hodnotu určuje příkaz vrácení. To při použití blocku jako příkazu vede ke stejné sémantice jako v C, a navíc ji rozšiřuje o ekvivalent konstruktu statement expression z GNU C při použití blocku jako výrazu. Je to vhodné k definici pomocných hodnot pomocí jediného výrazu bez uvádění pomocných proměnných do aktuálního kontextu. I jiné řídící struktury, které jsou v C pouhými příkazy, mohou představovat výrazy. Konkrétně `if` odpovídá ternárnímu operátoru v C a `for` odpovídá konstruktu *map* (případně i *zip*) z funkcionálních jazyků.

Funkce lze definovat příkazem, jako v C, ale i anonymě výrazem. Narozdíl od C, je tělo funkce dáno výrazem. To ale opět zahrnuje možnosti C, jelikož lze jako tělo funkce uvést block. Funkce mohou být vnořeny v jiných funkcích a přitom zachycují proměnné z vnějšího scopu v *closure*. Tyto proměnné se zachycují kopií a ve výsledném *closure* se implicitně stávají imutabilními. Takový mechanismus umožňuje použití některých běžných konstruktů funkcionálního programování, ale přitom nevyžaduje dynamickou alokaci, ani nenutí uživatele spravovat ručně paměť. Mutabilita zachycených zkopírovaných hodnot by mohla být vhodným rozšířením, ale pro jednodušší implementaci zůstanou tyto proměnné prozatím imutabilní. Samotný typ funkce je vždy implicitně imutabilní, tedy do proměnných typu funkce nelze přiřadit.

Následující příklady demonstrují ekvivalentní definice dvou funkcí využívající různé syntaktické konstrukty. Některé z nich jsou definovány anonymně a přiřazeny do proměnných. I anonymní funkce mohou být rekurzivní s použitím keywordu `self`.

```
// Addition:
Int (Int) add1 = Int fn (Int a, Int b) a + b;
Int (Int) add2 = Int fn (Int a, Int b) {
    return a + b;
};
Int add3 (Int a, Int b) a + b;
```

```
add1(1, 2);
// Factorial:
Int (Int) fact1 = Int fn (Int n) if (n == 0) 1 else n * self(n - 1)
Int (Int) fact2 = Int fn (Int n)
    if (n == 0) {
        return 1;
    } else {
        return n * self(n - 1);
    };
Int fact3 (Int n) {
    return if (n == 0)
        1
    else
        n * self(n - 1);
};
fact3(3);
```

Rekurze není v aktuální verzi implementována. Není pro syntezátory nutná.

2.2.3 N-tice

Uživatel může pomocí funkcí definovat spojování různých komponent syntezátoru. Aby ale výsledná komponenta mohla mít naopak více výstupů, je třeba umožnit funkcím vracet více hodnot. Proto bylo nezbytné do jazyka zakomponovat nějaké uživatelem definovatelné datové struktury, konkrétně produkty typů. Nabízí se nominální struktury typu `struct` v C, nebo čistě strukturální n-tice typu `std::pair` v C++. Byla zvolena druhá varianta, aby byl typový systém co nejjednodušší.

Namísto uvedení speciálního typu navíc bylo ale rozhodnuto, že každý typ bude dán *n-ticí* nějakých *elementárních typů*. Tedy i každá hodnota je dána n-ticí *elementárních hodnot*. Zahrnuje to i 1-tici a 0-tici. Konstrukt n-tic je použit i u deklarací proměnných. Je tak možné deklarovat více proměnných jedním příkazem a právě tak se uvádí deklarace parametrů funkce.

Syntakticky jde o uzávorkovaný a čárkami oddělený seznam typů či hodnot. Čárka za poslední položkou je také povolena. Syntakticky lze jako prvek uvést kromě elementárních typů i vnořenou n-tici. Sémanticky se však takové vnořené n-tice rozbalí do výčtu jednotlivých elementárních prvků, a tedy výsledné n-tice jsou vždy ploché.

0-tice je využita ve funkcích bez návratové hodnoty nebo bez parametrů, tedy jde o ekvivalent `void` v C. Taková funkce použije typovou 0-tici jako výstupní typ a deklarační 0-tici jako vstupní parametry. Potom se taková funkce aplikuje na hodnotovou 0-tici argumentů. Pro větší podobu s jazykem C je možné jako výstupní typ funkce uvést klíčové slovo `void`, které je ekvivalentní typové 0-tici. Příkaz vrácení bez hodnoty implicitně vrací 0-tici.

1-tice může být syntakticky vyjádřena jako explicitně uzávorkovaný prvek, ale i neuzávorkované prvky jsou implicitně 1-ticemi. Uzávorkování výrazů se tak právě vyjadřuje explicitním uzávorkováním 1-tic. Navíc lze uzávorkovat i typy,

což sice není s aktuální syntaxí typů nikdy potřeba, ale může to zlepšit čitelnost u komplexnějších typů.

0-tice se syntakticky vyjádří jako prázdné závorky, což je obecně nejednoznačné. Nelze totiž obecně poznat, jde-li o typ, hodnotu či deklaraci. Proto je použití 0-tic v syntaxi omezeno tak, aby byly vždy jednoznačné. Např. 0-tici hodnot je možné použít v rámci volání funkce, ale ne v přiřazení do proměnné.

Nevýhodou takového návrhu je rozdíl od běžných jazyků typu C a obtížné vysvětlení konstruktu. Přitom ale uživatel může bez problémů používat jazyk podobně, jako by používal C, aniž by měl představu, že je ve skutečnosti každý objekt reprezentován n-ticí. Na druhou stranu je výhodou to, že se implementace nemusí zabývat samotnými hodnotami a n-ticemi zvlášť.

Následující příklady demonstrují deklarace a přiřazování n-tic a také definice funkcí bez návratové hodnoty a parametrů.

```
(Int a) = (1);
(Int b) = 1;
Int c = 1;
(Int d, Int e) = (1, 2);
(d, e) = (2, 3);
(Int, Int) f = (1, 2);
f = (2, 3);
(Int g, Int h) = f;
(g, f) = (1, 2, 3);
(g, f) = (f, 3);
() f1 () {}
void f2 () {}
() f3 () { return }
() f3 () { return () }
```

2.2.4 Skaláry

Skalární hodnoty by měly vhodně reprezentovat hlavně čas a amplitudu signálu. Pro popis analogových syntezátorů by mohly stačit jen tyto dvě hodnoty a obě by mohly být reprezentovány floating point hodnotou. Pro práci s diskrétním časem ale budou potřeba i celá čísla.

Čas bude obecně reprezentován celým číslem odpovídajícím počtu vzorků. V případě potřeby výpočtu reálného času postačí vydělení tohoto diskrétního času vzorkovací frekvencí. Bude použit jen jeden celočíselný Cynth typ `Int` a ten bude ekvivalentní některému z celočíselných C typů. Výběr tohoto typu by měl být konfigurovatelný (alespoň za překladu překladače).

Amplituda signálu bude reprezentována floating point hodnotou. Stejně jako pro celočíselné hodnoty, bude i pro floating point hodnoty použit jeden typ `Float` odpovídající některému z floating point C typů.

Tyto dva typy by mohly postačit k popisu syntezátorů. Pro pohodlnost však bude navíc použit typ `Bool` reprezentující pravdivostní hodnoty `true` a `false`. Takové hodnoty mohou reprezentovat např. ovládací prvky se dvěma stavy.

Dalším vhodným typem by mohla být komplexní čísla. V této verzi však nebudou uvedena a pro jejich reprezentaci se bude používat dvojice `(Float, Float)`.

2.2.5 Názvy hodnot a typů

Syntakticky budou všechny názvy typů začínat velkým písmenem na rozdíl od názvů proměnných, které budou začínat malým písmenem. Oba názvy mohou po prvním písmenu obsahovat libovolnou kombinaci písmen, číslic a podtržítek. To tedy odpovídá identifikátorům v C s rozdílem omezení prvního písmene typů. Takové omezení zjednoduší lexikální analýzu, jelikož je ihned jasné jde-li o typ, či hodnotu bez nutnosti porozumění sémantickému kontextu.

Pro opakované použití komplexnějších typů (např. typů funkcí nebo polí) bude vhodné umožnit uživateli definovat vlastní názvy nejen skalárních typů. Půjde jen o čistě strukturální tzv. *aliasy* typů. Tedy nový název nepřidává typu jinou sémantiku, jen umožňuje na něj jinak odkazovat. To odpovídá konstruktu `typedef` v C. Byla však zvolena syntaxe inspirovaná ekvivalentním konstruktem `using` z C++ pro jeho lepší čitelnost. Zvolená syntaxe se podobá syntaxi deklarace proměnných, jen místo typu proměnné obsahuje klíčové slovo `type`. Následující příklad demonstruje typový alias funkce reprezentující signál v diskrétním čase.

```
type Signal = Float (Int);
```

2.2.6 Pole

Práce s poli je poměrně důležitá pro DSP. Pole můžou reprezentovat např. impulzní odezvu filtru, nebo třeba předpočítané tabulky goniometrických funkcí. Bylo by tedy rozumné věnovat při návrhu pozornost pohodlné manipulaci polí v jazyce. Kvůli zmíněným omezením dynamické alokace budou pole vždy statické velikosti a budou obsahovat prvky jednoho typu. Přitom je ale výhodné umožnit předávání pole odkazem, aby nedocházelo ke zbytečným kopiím. K tomu je třeba rozmyslet způsob alokace takovým způsobem, aby nedocházelo k dynamické alokaci a přitom aby uživatel nemohl způsobit situaci, kdy zůstane odkaz na již neexistující pole.

Sémantika předávání polí je proto navržena tak, aby docházelo ke kopii při vrácení pole z funkce, nebo jeho zachycení ve funkci z vnějšího scopu. Alokace polí pak probíhá na úrovni funkce. To znamená, že i pole deklarovaná ve vnořených scopech vytvoří hodnotu, životnost které je rozšířena na scope celé funkce. Je tak možné např. vracet pole z vnořených blocků v rámci funkce. Vhodné by také bylo implementovat nějakou variantu copy-on-write při zachycení polí ve funkcích, tedy zachycovat je referencí a kopírovat až v okamžiku, kdy do nich proběhne zápis, nebo kdy skončí jejich životnost. To by mohlo být komplikované implementačně a ne vždy možné. V případě polí z nejvnějšího (globálního) scopu by taková optimalizace nemusela být problematická.

Pole je možné vytvořit literálem, tedy výrazem, který obsahuje výčet prvků oddělených čárkami a uzavřených v hranatých závorkách. Prvky bude možné udávat po jednom, pomocí speciálních konstruktů vyjadřujících aritmetickou posloupnonst,

nebo použitím prvků jiného pole. Tyto tři typy vyjádření prvků lze libovolně kombinovat v jednom literálu.

Indexace pole je inspirována klasickým subscriptem z C, ale může být navíc rozšířena o výběr souvislého podpole a indexaci pozpátku zápornými čísly. Indexace se vyjádří použitím hranatých závorek za výrazem odkazujícím na pole. Do hranatých závorek je možné uvést výčet indexů, jako by šlo o výše zmíněný literál. Výběr jednoho prvku pole se vyjádří uvedením jednoho indexu. Výběr souvislého podpole se vyjádří uvedením více indexů. Výběr podpole velikosti 1 není možný. Bylo by rozumné zvážit, jsou-li jednoprvková pole vůbec vhodná. Pole velikosti 0 nejsou povoleny obecně. Uvedení prázdného seznamu indexů se využije pro výběr celého pole. Výběr jednotlivých prvků i podpole proběhne kopií. To se týká i výběru celého pole. Tedy uvedení prázdné hranaté závorky označuje kopii všech hodnot do nového pole.

Indexaci je možné použít i na levé straně přiřazení. Na této pozici se obecně nenachází výrazy, ale tzv. *cíle*. Cíl může být tvořen jménem proměnné a případně indexací. Cíle jsou také vyjádřeny jako n-tice. Je tak možné přiřadit do prvků pole i do souvislých podpolí, což zahrnuje i možnost přiřadit do všech prvků pole a přitom neměnit ukazatel pole.

Vyjádření typu pole je také podobné jakyzu C, ale nevyžaduje uvedení názvu mezi typ hodnoty a velikost pole. Celý typ je tedy uveden typem hodnot pole následovaným jeho velikostí v hranatých závorkách (např. `Int [16]`). Typ hodnot bude v aktuální verzi omezen na 1-tice skalárních hodnot. Velikost by mělo být možné uvést i komplexnějšími výrazy. Postačí výrazy vyhodnotitelné za překladu. Klíčovým slovem `const` za typem hodnoty je možné vyjádřit imutabilitu obsažených hodnot, zatímco `const` za celým typem pole (za hranatými závorkami) vyjadřuje imutabilitu odkazu.

```
[1, 2, 3 to 6]; // [1, 2, 3, 4, 5, 6]
[1 to 10 by 2]; // [1, 3, 4, 6, 7, 8]
[3 to 0 by -1]; // [3, 2, 1]
[1, ...[2, 3]]; // [1, 2, 3]
Int [3] a = [1, 2, 3];
a[0] = 2;
a[1, 2] = [4, 5];
a[] = [2, 3, 4];
a = [3, 4, 5];
```

2.2.7 Buffery

Další důležitou datovou strukturou pro DSP je cyklická fronta, neboli cyklický buffer o konstantní velikosti. Cyklický buffer může reprezentovat historii diskrétního signálu. V případě emulace analogových syntezátorů pak mohou tyto buffery odpovídat vodičům v obvodu. Takový buffer proto bude reprezentován vestavěným typem.

Buffer se v syntaxi i v sémantice bude shodovat s polem až na jejich charakteristické rozdíly. Typ položek v bufferu bude omezen jen na 1-tici `Float`. Vyjádření typu bufferu proto místo typu jeho hodnot obsahuje klíčové slovo `buffer` násle-

dované velikostí v hranatých závorkách (např. `buffer [64]`). Pokud bude jazyk rozšířen o buffery jiných typů, bude třeba tento zápis rozšířit.

Indexace bufferů podporuje všechny konstrukty indexace polí. Výběr jednoho prvku dá zvolený prvek, zatímco výběr více prvků dá pole zkopírovaných hodnot. Buffery reprezentují historii signálu, tedy jejich indexace by mohla pro pohodlnost odpovídat diskrétní reprezentaci času (počtu vzorků). Rostoucí indexy by tak měly odpovídat posunu v čase dopředu. Aktuální vzorek bude reprezentován indexem 0. Starší vzorky pak budou na menších indexech.

Zápis do bufferu způsobem explicitního přiřazení do jeho prvků nebude umožněn. Místo toho uživatel k bufferu uvede tzv. *generátor*, tedy funkci reprezentující signál, historii kterého má daný buffer ukládat. Běh výsledného programu se určí právě těmito generátory. Ve smyčce se vypočítají hodnoty nových vzorků všech bufferů a následně se do nich zapíší, a tím přepíší nejstarší vzorek. Odkaz bufferu je vždy implicitně imutabilní. Jeho hodnoty, i přes to, že se mohou měnit při výpočtu nových vzorků z generátorů, jsou z pohledu uživatele také imutabilní.

```
buffer [3] b = sin;
buffer [16] c = fn (Int t) b[0.] + b[1.] + b[2.] / 3.;
```

2.2.8 Iterace

K iteraci slouží konstrukty `for` a `while`. Konstrukt `while` je prakticky identický tomu v C a umožňuje obecnější iteraci podmíněnou danou pravdivostní hodnotou. `for` je určený k iteraci přes pole a buffery. Syntaxe klasické C `for` smyčky chybí. Místo ní se používá omezenější syntaxe, která uvede iterační proměnné a jejích rozsah určí danými poli nebo buffery.

Iteraci v C typu `for (int i = 0; i < n; ++i)` lze zapsat jako iteraci přes pole `for (Int i in [0 to n])`. Speciální prvky konstruktoru pole tak umožňují i takovou syntaxi použít dost pohodlně k iteraci v daném rozmezí. Implementačně by bylo vhodné v některých případech překládat takové smyčky bez alokace pomocných polí. Tělo obou smyček se udává příkazem. Je tak možné použít block, ale i libovolný jiný příkaz. Následující příklad demonstruje několik ekvivalentních smyček.

```
Int i = 0;
Int s = 0;
while (i < 5) {
    s = s + 1;
};
while (i < 5)
    s = s + 1;
for (Int i in [0 to 5])
    s = s + 1;
```

Konstrukt `for` lze také použít jako výraz, pokud se jeho tělo uvede výrazem. Umožňuje tak vyjádřit transformaci daného pole nějakým výrazem.

```
Int [5] a = for (Int x in [0 to 5]) x * 2;
```

2.2.9 Vstup a výstup

Vstup a výstup programu se definuje pomocí *vstupních a výstupních typů*. Klíčové slovo `in`, resp. `out` uvedené za typem vyjádří jeho vstupní, resp. výstupní variantu. V aktuální verzi budou vstupní a výstupní jen skaláry a buffery. Skaláry reprezentují ovládací a monitorovací prvky syntezátoru. Buffery reprezentují vstupní a výstupní signály. V aktuální verzi budou využity jen jeden až dva výstupní signály (pro stereo zvuk), ale další vstupy i výstupy budou užitečné při komplexnějším zapojení více syntezátorů v dalších verzích.

Každá proměnná deklarovaná s takovým typem se alokuje jednou po dobu běhu celého programu. Ve výsledném přeloženém C programu se vygeneruje funkce vracející ukazatel na vstupní či výstupní proměnnou s názvem daným parametrem. Standardní řídící program tak využívá proměnné s názvy odpovídajícími ovládacím a monitorovacím prvkům v konfiguraci řídícího programu. Výstupní buffery budou prozatím dány napevno názvy `out1` a `out2`. Těmto bufferům stačí jeden vzorek, jelikož se následující hodnoty zapisují do zvukové karty po jednom.

Do hodnot vstupních typů bude uživateli zakázán zápis a z hodnot výstupních typů zakázáno čtení. Dále vstupní a výstupní typy musí mít referenční sémantiku. Pro konzistenci budou modelovány podobně jako pole a buffery. Přiřazení do pole změní jeho odkaz, ne hodnotu, proto i přiřazení do výstupní proměnné by nemělo měnit odkazovanou hodnotu. Pole se také předává odkazem, proto předání vstupních typů by nemělo kopírovat odkazovanou hodnotu. Předávání výstupních typů odkazem bude také možné. Pro zápis do výstupních typů a čtení ze vstupních typů se použije již existující syntaxe indexace pole. V tomto případě se však indexace omezí na případ prázdných hranatých závorek. V případě pole jde o kopii celého pole nebo zápis do všech jeho prvků. Stejně tak i u vstupních a výstupních typů půjde o kopii (jediné) odkazované hodnoty nebo její zápis.

2.2.10 Výpočty za překladu

Výpočty za překladu jsou nutné hlavně kvůli možnosti určení velikosti polí a bufferů z komplexnějších výrazů. Navíc může být vhodné předpočítávat pole vzorků různých funkcí. Takové hodnoty se označují jako *kompilační konstanty*.

Pro jednodušší a předvídatelnější sémantiku bylo rozhodnuto, že se za překladu budou vypočítávat jen hodnoty, které nezávisí na potenciálně měnících se hodnotách. To zahrnuje imutabilní typy, buffery a vstupní či výstupní typy. Imutabilní skaláry totiž může měnit uživatel a ty ostatní mohou být měněny implicitně. S takovým omezením platí, že každý výraz spočtený za překladu zůstává neměnným ve stejném scopu. Tedy např. vyjádření typu pole `Int [a + 2]` bude mít stejný význam, dokud se nezastíní proměnná `a` ve vnořeném scopu. Každá kompilační konstanta tedy musí být opravdu konstantou a její hodnota musí být známa za překladu.

2.2.11 Předdefinované hodnoty

Už v prázdném programu může uživatel počítat s některými existujícími předdefinovanými hodnotami. Tyto hodnoty jsou definovány jako obyčejné proměnné v

nejvnějším (globálním) scopu. Ve vnořených scopech je lze zastínit.

Jde konkrétně o relevantní matematické konstanty π a e a goniometricke funkce sin a cos. Další takové hodnoty a funkce mohou být přidány do budoucích verzí dle potřeby.

2.2.12 Konverze typů

Explicitní konverze typů se provede použitím typu, jako by šlo o funkci (např. `Float (1)`). To odpovídá konstruktu *functional cast* v C++. Syntaxe jazyka C nebyla zvolena kvůli potenciálním komplikacím gramatiky a její špatnou čitelností. Díky možnosti uzávorkování typů se lze C syntaxi alespoň přiblížit (např. `(Float) (1)`).

Konverze mezi celými a floating point čísly se mírně liší od konverzí odpovídajících typů v C. Jsou navrženy tak, aby vždy docházelo k zaokrouhlování směrem k nule. Konverze z pravdivostních hodnot na jiné skaláry odpovídají těm v C, tedy `true` odpovídá celočíselné hodnotě `1` a false hodnotě `0`. Hodnota `0` se konvertuje na `false` a všechny ostatní hodnoty na `true`. Při konverzi mezi `Bool` a `Float` dojde nejprve ke konverzi na `Int`.

Konverze polí a bufferů umožňuje změnu velikosti. Tyto konverze nekopírují obsažené hodnoty, ale jen vytvoří nový odkaz, který počítá s jiným počtem prvků. Takové konverze je možné provést bezpečně se statickou pamětí pouze při zmenšování velikosti. Konverze bufferu na pole vytvoří nové pole se zkopírovanými prvky.

Konverze mohou přidat i odebrat imutabilitu skalárních hodnot. V případě polí je možné přidat i odebrat imutabilitu odkazu, ale imutabilitu hodnot lze jen přidat. Jiný návrh konverzí polí by znamenal, že by bylo možné do pole s imutabilními hodnotami zapisovat z jiné proměnné odkazující na stejné pole.

Implicitní konverze nebudou prozatím uváženy. Rozumný návrh implicitních konverzí je obecně těžké provést bez uvedení potenciálních nejasností sémantiky. Avšak rozšíření o implicitní konverze by mohlo být vhodné do budoucích verzí, umožňuje kratší a občas i přehlednější zápis. Mohou v některých případech uživateli usnadnit práci, ale v některých případech naopak i skrýt chyby. (Pradel a Sen, 2015, Introduction).

2.2.13 Matematické a logické operátory

Veškeré unární i binární matematické a logické operátory z C budou v Cynthu použity. Stejně jako v C budou tyto operátory přijímat různé skalární typy. Bez implicitních konverzí však bude uživatel muset sám zařídit, že jsou oba operandy binárních operátorů stejného typu. Typ výsledku těchto operací bude stejný, jako jejich vstup.

Logické operátory negace, konjunkce a disjunkce odpovídají sémanticky těm v C až na to, že přijímají striktně jen hodnoty typu `Bool`. Konjunkce i disjunkce nevyhodnocují jejich druhý operand v případě jednoznačnosti výsledku už z toho prvního.

Kromě matematických operací z C (unární plus, unární mínus, sčítání, odečítání, násobení, dělení a modulo) se využije i operace celočíselného a floating point umocnění. Operace dělení i modulo mají také celočíselnou i floating point sémantiku. Celočíselné dělení na rozdíl od C vždy provádí zaokrouhlení směrem k nule. Modulo pak odpovídá právě takovému dělení. Taková sémantika je vhodnější hlavně kvůli potřebě indexování cyklických bufferů.

Navíc kromě uvedených operací byly zváženy i speciální operace sumace a konvoluce na polích. Po uvedení výrazu `for` bylo rozhodnuto, že k tomu postačí jen funkce sčítající všechny prvky pole. Následující příklady demonstrují, jak lze pomocí konstruktu `for` iterovat různá pole, nebo hodnoty v daných rozmezích a výsledek jednoduše sečíst.

```
sum(for (Int i in [0 to n]) i)
sum(for (Int i in [0 to n]) a[i] * b[n - i])
sum(for (Int e in a, Int f in b[n - 1 to -1]) a[i] * b[n - i])
```

S aktuálně navrženým typovým systémem není možné definovat funkci, která by přijímala libovolně velké pole. Bylo by tedy třeba uvést nějaké vhodné rozšíření nebo implementovat funkci `sum` jako speciální konstrukt. V obou případech by bylo třeba navíc promyslet, jak sumaci ve vhodných případech provádět bez zbytečné alokace pomocných polí. Kromě konstruktu sumace by bylo možné uvést i odpovídající konstrukt produktu. V aktuální verzi žádný z těchto konstruktů nebude implementován ani navržen.

3. Demonstrace

Tato kapitola ukáže doporučený postup překladu a sestavení výsledného programu a příklady Cynth programů.

3.1 Pracovní prostředí

Proces překladu a sestavení výsledného programu vyžaduje použití překladače jazyka C. Žádný překladač jazyka C však není součástí přiloženého projektu. Je tedy na uživateli, aby měl na svém stroji nainstalován nějaký vhodný překladač. Podmínkou je podpora GNU C dialektu. Konkrétně tomu vyhovují např. překladače GCC a Clang. U jiných překladačů nemusí být zaručen správný výsledek.

3.1.1 Sestavení

Zkompilování překladače a dalších částí se provádí pomocí programu GNU make. GNU make na Linuxu a dostupné verze GNU make pro Windows, jako např. GnuWin32, nebo make dostupný na správci balíčků Chocolatey, fungují s přítomnými Makefile soubory správně. Spouštění Makefile souborů (a obecně kompilace) v prostředích jako MinGW, Cygwin a MSYS není doporučeno. Může to způsobit komplikace s použitou GUI knihovnou, nebo s ASIO ovladačem. Překlad byl na Windows otestován s překladačem MSVC. Na Linuxu byl překlad otestován s překladači GCC a Clang.

Program lze přeložit a spustit v omezeném režimu na Linuxu s tím, že místo ovladače zvukových karet ASIO se použije mockup ovladač, který výsledný signál jen vypíše na standardní výstup. Tento mockup ovladač lze použít i na Windows.

Každý Makefile soubor obsahuje cíl `help` který slouží vypsání popisu funkcionality daného Makefile souboru a jeho možné parametry, které mohou např. zvolit použitý překladač. Pro překlad z přiloženého zdrojového kódu postačí jen hlavní Makefile soubor v kořenovém adresáři. Ostatní Makefile soubory jsou použity jen během vývoje.

Spuštění programu make v kořenovém adresáři bez uvedení cílů provede kompletní kompilaci překladače, ovládacího programu a všech pomocných binárních souborů. Výsledné binární soubory se umístí do adresáře `build/`, kde jsou dále rozděleny do složek podle operačního systému a použitého ovladače.

```
make
```

Pro jednoduché použití přeloženého Cynth překladače, C překladače a sestavení s řídícím programem je k dispozici tzv. *workspace*. Jde o adresář s nutnými binárními soubory, příklady Cynth kódu a Makefile souborem k sestavení výsledného Cynth programu. Spuštění programu make v kořenovém adresáři s cílem `workspace` vytvoří workspace adresář v build adresáři.

```
make workspace
```

K určení umístění workspace adresáře lze použít parametr `WORKSPACE`.

```
make workspace WORKSPACE=c:/users/me/my-cynth-workspace
```

3.1.2 Použití

Spuštění programu make z workspace adresáře zkompiluje soubor se zdrojovým Cynth kódem daným parametrem `INPUT`. Výsledný spustitelný soubor se uloží s cestou `out/cynth.exe`. Potřebné dynamické knihovny jsou pak umístěny ve stejném adresáři.

```
make INPUT=examples/basic.cth
```

Specifický C překladač lze určit parametrem `CYNTH_C_COMPILER`.

```
make INPUT=examples/basic.cth CYNTH_C_COMPILER=gcc
```

3.2 Praktické příklady

V této sekci budou uvedeny praktické příklady použití jazyka Cynth se stručným vysvětlením některých konstruktů jazyka. Kompletní spustitelné příklady jsou k nalezení v adresáři `examples/`, nebo uvnitř adresáře *workspace* na stejné cestě.

Příklady v této sekci předpokládají definice následujících typů a funkcí.

```
type Signal     = Float ();
type TimeSignal = Float (Int);

Float realTime (Int const time) Float (time) / srate;

Float cos (Float const x) -sin(x - pi / 2.);

Float abs (Float const x) if (x < 0.) -x else x;
```

Typ `Signal` reprezentuje signál nezávislý na časovým parametru. Takové signály mohou využívat hodnoty z neustále měnících se hodnot v bufferech a tedy nepotřebují informaci o aktuálním čase. `TimeSignal` naopak reprezentuje signály, které ke své definici potřebují časový parametr.

Čas v signálech je dán celočíselnou hodnotou reprezentující pozici aktuálního vzorku. Pro získání hodnoty odpovídajícího času v sekundách lze použít funkci `realTime`. Funkce `sin` je předdefinována v každém programu. Funkci `cos` lze tak pomocí ní definovat jako uživatelem definovanou funkci.

Dále budou příklady předpokládat s následujícími deklaracemi vstupních proměnných.

```
Int  in note;
Int  in press;
Int  in knob1;
Int  in knob2;
```

```
Int  in knob3;
Bool in switch1;
Bool in switch2;
```

V budoucích verzích bude vhodné rozšířit implementaci řídícího programu o YAML konfiguraci vstupních ovládacích prvků, jak bylo zmíněno v návrhu. V aktuální verzi má řídící program předdefinované ovládací prvky a jejich názvy. Tyto prvky jsou ovladatelné klávesnicí, a některé z nich i připojenými MIDI ovladači nebo GUI prvky. Jejich hodnota je v aktuální verzi vždy celočíselná nebo binární (pravdivostní). Pokud uživatel deklaruje vstupní proměnnou s názvem odpovídajícím některému z v řídícím programu definovaných ovládacích prvků, je hodnota této proměnné za běhu nastavována na hodnotu odpovídajícího ovládacího prvku.

Proměnná `note` obsahuje pořadí noty odpovídající stisklé klávese klaviatury MIDI ovladače, nebo kláves písmen a, w, s, e, d, f, t, g, y, h, a u na klávesnici. Navíc předdefinované pole `notes` obsahuje frekvence not chromatické stupnice od 32,7 Hz po 61,74 Hz, tedy $C1$ až $B1$, což umožňuje získat frekvenci noty odpovídající stisklé klávese ve standardním ladění, kde $A4 = 440\,\text{Hz}$.

Proměnná `press` pak obsahuje počet vzorků od posledního stisku klávesy. Tuto proměnnou lze použít na výpočet tzv. *envelope*, neboli *obálky* signálu, která popisuje změnu zvuku v čase po stisku (a také puštění) klávesy. Při ovládání syntezátoru klávesnicí nelze spoléhat na rozumné hodnoty této proměnné při držení klávesy.

Pro výpočet frekvence noty budou příklady níže využívat následující funkce. Funkce `keyOct` využívá hodnoty ovládacího prvku `knob2` k posunu kláves o celé oktávy, tedy zvýšení této hodnoty o jedna odpovídá vynásobení všech použitých not dvěma.

```
Int keyOct () if (knob2[] > 8) 8 else if (knob2[] < 0) 0 else knob2[];
Float keyFreq () notes[(note[] % 12)] * (2. ** Float (keyOct() + 1));
```

3.2.1 Jednoduché signály

Následující příklad demonstruje, jak lze použít definice vnořených funkcí k popisu parametry konfigurovatelných jednoduchých signálů.

```
TimeSignal sawWave (Float const amp, Float const freq)
    Float fn (Int const time)
        amp * (2. * ((freq * realTime(time)) % 1.) - 1.);

TimeSignal sineWave (Float const amp, Float const freq)
    Float fn (Int const time)
        amp * sin(realTime(time) * 2. * pi * freq);

TimeSignal saw1 = sawWave(.6, 220.);
TimeSignal saw2 = Float fn () sawWave(.6, keyFreq());
```

Takový „naivně“ generovaný signál pilové vlny má v praktickém použití nedosta-

tek v tom, že po vzorkování takového signálu se ve výsledném spektru vyskytnou nechtěné frekvence, které v původním spojitém signálu přesahovaly Nyquistovu frekvenci (Välimäki a Huovilainen, 2006).

3.2.2 Signály s pamětí

Pro vygenerování poněkud zajímavějšího signálu, který tolik netrpí výše zmíněným problémem, a navíc více připomíná signál, který by mohl vzniknout v analogovém syntezátoru, lze použít bipolárního modula a jednoduchého filtru.

```
TimeSignal mod (Float freq)
    Float fn (Int time)
        (realTime(time) + freq / srate) % 1.;

TimeSignal bipolar (Float freq, Float phase)
    Float fn (Int time)
        (2. * (((realTime(time) - phase) * freq) % 1.) - 1.) ** 2.;

type DiffMem = buffer [3];

(TimeSignal, Signal (DiffMem)) saw (Float () freq) (
    Float fn (Int t) bipolar(freq(), 0.)(t),
    Signal fn (DiffMem memory) Float fn () {
        Float const freq  = freq();
        Float const scale = srate / (4. * freq * (1. - freq / srate));
        return scale * (memory[0] - memory[-2]) / 2.;
    }
);
```

Poslední funkce `saw` je parametrizovaná funkcí určující základní frekvenci požadovaného signálu a dává dva signály. Jeden z nich reprezentuje hodnoty bipolárního modulo počítadla a druhý popisuje výsledný signál, který závisí na průběžně ukládaných hodnotách.

Následující příklad demonstruje alokaci potřebné paměti a sestavení výsledné komponenty. První signál z funkce výše se použije jako generátor hodnot nového bufferu a v druhém signálu se pak specifikuje tento buffer jako zdroj potřebných hodnot modulo počítadla.

```
Signal sawTone1 = {
    Float freq () notes[(note[] % 12)] * (2. ** Float (keyOct() + 1));
    (TimeSignal sawIn, Signal (DiffMem) sawOut) = saw(freq);
    DiffMem sawMem = DiffMem(sawIn);
    return sawOut(sawMem);
};
```

Při použití dvou počítadel lze vygenerovat např. i čtvercovou vlnu demonstrovanou níže. Výsledná komponenta `squareLfo1` představuje oscilátor o nízké frekvenci ovládané ovládacím prvkem `knob1`. Takový nízkofrekvenční oscilátor je vhodný např. k modulaci amplitudy jiného signálu.

```
(TimeSignal, TimeSignal, Signal (DiffMem, DiffMem)) square (
    Float () freq
) (
    Float fn (Int t) bipolar(freq(), 0.)(t),
    Float fn (Int t) bipolar(freq(), 1. / freq() / 2.)(t),
    Signal fn (DiffMem memory1, DiffMem memory2) Float fn () {
        Float const freq  = freq();
        Float const scale = srate / (4. * freq * (1. - freq / srate));
        Float s1 = scale * (memory1[0] - memory1[-2]) / 2.;
        Float s2 = scale * (memory2[0] - memory2[-2]) / 2.;
        return s1 - s2;
    }
);

Signal squareLfo1 = {
    Float freq () Float(knob1[] + 1) / 4.;
    (
        TimeSignal squareIn1,
        TimeSignal squareIn2,
        Signal (DiffMem, DiffMem) squareOut,
    ) = square(freq);
    DiffMem squareMem1 = DiffMem(squareIn1);
    DiffMem squareMem2 = DiffMem(squareIn2);
    return Float fn () (squareOut(squareMem1, squareMem2)() + 2.) / 3.;
};
```

3.2.3 FIR filtry

Pro implementaci FIR filtrů, tedy nerekurzivních filtrů s konečnou impulzní odezvou, lze jednoduše provést konvoluci signálu s danou impulzní odezvou.

Následující příklady budou vycházet z *brick wall* filtru, který ve výsledném signálu nechává jen daný souvislý úsek frekvencí. Brick wall filtr má nekonečnou impulzní odezvu danou funkcí *sinc*. Pouhé „useknutí“ konečného úseku této odezvy by nebylo vhodnou aproximací brick wall filtru, kvůli nevhodným změnám spektra impulzní odezvy (Smith a kol., 1997, Chapter 16). Proto se navíc impulzní odezva brick wall filtru vynásobí vhodnou *okénkovou funkcí*, např. Blackmanovým oknem, které impulzní odezvu zbaví ostrého přechodu na krajích vybrané konečné části.

```
Float sinc (Int const i, Int const size, Float const cutoff) {
    Int const j = i - size / 2;
    return if (j == 0)
        2. * pi * cutoff
    else
        sin(2. * pi * cutoff * Float(j)) / Float(j);
};

Float blackman (Int const i, Int const size)
    .42 -
```

```
    .5  * cos(2. * pi * Float(i) / Float(size - 1)) +
    .08 * cos(4. * pi * Float(i) / Float(size - 1));
```

Následující příklady budou počítat s pevně danou velikostí impulzní odezvy danou proměnnou `size` definovanou níže. Výpočet impulzní odezvy výše popsaného filtru demonstruje následující úryvek.

```
Int const size = 501;
type Impulse = Float const [size];

Impulse const impulse1 = Impulse ({
    Float const cutoff = .2;
    Impulse const s = Impulse (
        for (Int const i in [0 to size])
            sinc(i, size, cutoff) * blackman(i, size)
    );
    Float sum = 0.;
    for (Float const e in s) sum = sum + e;
    return Impulse (for (Float const e in s) e / sum);
});
```

Tento výpočet lze také provést ve funkci.

```
Impulse impulse (Float const cutoff) {
    Impulse const s = Impulse (
        for (Int const i in [0 to size])
            sinc(i, size, cutoff) * blackman(i, size)
    );
    Float sum = 0.;
    for (Float const e in s) sum = sum + e;
    return Impulse (for (Float const e in s) e / sum);
};

Impulse impulse2 = impulse(.3);
```

Dále úryvek níže definuje funkci pro konvoluci pole s bufferem a její použití. Funkce `conv` očekává buffer, proto je třeba níže zvolený signál `saw2` navíc konvertovat do bufferu. Tím se alokuje nový buffer, jako by byl deklarován proměnnou.

```
type Buff = buffer [size];

Signal conv (Buff input, Impulse const impulse) Float fn () {
    Float result = 0.;
    for (Int const i in [0 to size])
        result = result + input[-i] * impulse[i];
    return result;
};

Signal filtered = conv(Buff (saw2), impulse1);
```

Kromě takto obecného filtru lze implementovat i různé specifické FIR filtry, jako

např. *klouzavý průměr* (níže `avg`), nebo různá zpoždění (níže `delay`). *Hřebenový filtr* (níže `comb`) sčítá signál se svou zpožděnou verzí.

```
Signal avg (buffer [size] input) Float fn () {
    Float result = 0.;
    for (Int const i in [0 to -size])
        result = result + input[i];
    return result / Float (size);
};

Signal delay (Buff b, Int const shift)
    Float fn () b[-shift];

Signal comb (Buff b, Float const amp, Int const shift)
    Float fn () b[0] + amp * b[-shift];
```

4. Specifikace jazyka Cynth

4.1 Lexikální tokeny

Zdrojový kód programu je posloupností znaků, konkrétně velkých a malých písmen latinské abecedy (dále jen *písmen*), *číslic* 1 až 9, *bílých znaků* `\t`, `\n`, `\r`, a *speciálních znaků* `.`, `,`, `;`, `$`, `_`, `=`, `+`, `-`, `*`, `/`, `%`, `!`, `&`, `|`, `(`, `)`, `[`, `]`, `{`, `}`, `<` a `>`.

Zdrojový kód lze jednoznačně rozdělit na posloupnost lexikálních *tokenů*. Tokeny zahrnují *klíčová slova*, *symboly*, *názvy proměnných*, *názvy typů*, *číselné literály*, *komentáře* a *bílé znaky*.

Klíčová slova jsou posloupnostmi písmen. Velká a malá písmena mohou být v klíčových slovech libovolně zaměněna beze změny významu. Následující seznam je výčtem všech klíčových slov. Klíčové slovo `self` je vyhrazeno pro budoucí verze.

```
if      else    when
while   for     in
to      by      self
true    false   return
type    buffer  const
out     fn      void
```

Symboly jsou posloupnostmi speciálních znaků. Následující seznam je jejich kompletním výčtem. Symbol `$` je vyhrazen pro budoucí verze.

```
( ) [ ] { }
+ - * / % **
! && ||
== != >= <= > <
, ; = $ ...
```

Bílé znaky jsou tabulátory (`\t`), mezery ( ) a konce řádků (`\n` a `\r`). V aktuální verzi bílé znaky neovlivňují sémantiku programu.

Komentáře jsou *řádkové* a *víceřádkové*. Řádkové komentáře začínají posloupností `//` nebo znakem `#` a končí prvním dalším výskytem libovolného znaku konce řádku. Víceřádkové komentáře začínají posloupností `/*` a končí prvním dalším výskytem posloupnosti `*/`.

Názvy proměnných a typů jsou posloupnostmi písmen, číslic a podtržítek, které nejsou klíčovými slovy. Navíc názvy proměnných musí začínat malým písmenem, zatímco názvy typů musí začínat velkým písmenem.

Číselné literály jsou celočíselné a floating point. Celočíselné literály jsou dány regulárním výrazem `[0-9]+([eE][+-]?[0-9]+)?`. Jde o posloupnost číslic volitelně následovanou exponentem ve vědecké notaci. Floating point literály navíc obsahují desetinnou tečku a levá, nebo pravá strana může být vypuštěna. Jsou dány regulárním výrazem `([0-9]+\.[0-9]*|\.[0-9]+)([eE][+-]?[0-9]+)?`.

Ostatní znaky a tokeny netvoří validní zdrojový program. V budoucích verzích může být rozšířena definice písmen na znaky mimo latinskou abecedu.

4.2 Syntaktické a sémantické konstrukty

Validní posloupnosti tokenů tvoří syntaktické konstrukty. Validní syntaktické konstrukty reprezentují sémantické konstrukty. Pro daný sémantický konstrukt se jeho odpovídající syntaktický konstrukt označuje jako jeho *syntaxe*. Např. vyjádření typu a syntaxe vyjádření typu nebo výraz a syntaxe výrazu.

Povinně uzávorkovaný seznam je syntaktický konstrukt, který začíná otevřenou kulatou, hranatou, nebo složenou závorkou a končí odpovídající uzavřenou závorkou. Obsahuje nula a více prvků oddělených separátorem, konkrétně čárkou, nebo středníkem. Separátor se navíc může vyskytovat za posledním prvkem před zavřenou závorkou. *Nepovinně uzávorkovaný seznam* navíc nemusí být ohraničen závorkami, pokud obsahuje jen jeden prvek.

Výrazy, *vyjádření typů*, *deklarace*, *iterační deklarace* a *cíle* se obecně označují jako *n-ticové konstrukty*. N-ticové konstrukty jsou n-tice, obsahující tzv. *n-ticové atomy*.

Syntaxe n-tice je seznam prvků oddělených čárkami a uzavřených kulatými závorkami. Jejími prvky jsou vnořené nepovinně uzávorkované syntaxe n-tic. Syntaxe daná prázdným seznamem reprezentuje prázdnou n-tici. Každý prvek seznamu v syntaxi navíc určuje dalších jedna a více prvků jím reprezentované n-tice. Atomický prvek určuje právě jeden n-ticový atom ve výsledné n-tici. Neatomické prvky rekurzivně reprezentují n-tici, prvky které určují další prvky výsledné n-tice (bez vnoření).

4.3 Syntaxe hybridních konstruktů

Jako *hybridní* se označují konstrukty, které mohou představovat výraz i příkaz.

Syntaxe hybridního konstruktu určuje výraz, pokud je použita na místě výrazu. To znamená, že představuje část syntaxe jiného konstruktu, která má být dle specifikace syntaxí výrazu. Jinak syntaxe hybridního konstruktu určuje příkaz.

Hybridními konstrukty jsou konstrukty *block*, podmínka *if* a smyčka *for*.

Např. syntaxe binárního operátoru je složena ze dvou syntaxí výrazů a operátoru mezi nimi. Použití syntaxe bocku na místě operandu určuje výraz blocku. Naopak třeba použití syntaxe blocku jako jednoho z příkazů jiného blocku určuje příkaz blocku.

4.4 Syntaxe výrazů

Syntaktické konstrukty tvořené jediným tokenem jsou tzv. *syntaktické atomy*.

Syntaktické konstrukty, které nejsou jednoznačně ohraničeny zleva, resp. zprava na pevno danou posloupností tokenů se označují jako zleva, resp. zprava *neomezené*.

Např. syntaxe binárních operátorů obsahuje dvě syntaxe výrazů oddělené symbolem a je tedy zleva i zprava neomezená. Třeba k interpretaci syntaxe `1 + 2 * 3`

nestačí informace, že syntaxe sčítání, resp. násobení obsahuje dvě syntaxe výrazů oddělené symboly `+`, resp. `*`, jelikož není jasné, jestli je syntaxe výrazu sčítání součástí výrazu násobení, nebo naopak.

V popisu gramatiky se využívá více úrovní syntaxí výrazů. V této specifikaci se místo toho využije koncept *předností výrazů*. Přednost výrazu je dána celým číslem. Přednost syntaxe výrazu je právě předností výrazu. Ne všechny výrazy mají definovánu přednost. Výrazy, které nemají definovánu přednost (výrazy podmínky, smyčky a definice funkce) se označují jako *asymetrické výrazy*. Dále může mít výraz definovánu *asociativitu* levou nebo pravou. Výrazy, které nemají definovánu asociativitu (syntaktické atomy a oboustranně omezené syntaxe) mají nejvyšší přednost.

Pro každou validní interpretaci syntaxe výrazu platí: Syntaxe výrazu nemůže být nejlevější ani nejpravější podposloupností syntaxe výrazu s vyšší předností. Syntaxe výrazu levé, resp. pravé asociativity nemůže být nejpravější, resp. nejlevější podposloupností výrazu se stejnou předností. Nechť je asymetrická syntaxe x obsažena v syntaxi y a přitom y neobsahuje zprava omezenou syntaxi obsahující x. Potom y nemůže být nejlevější podposloupností žádné syntaxe výrazu. Tabulka níže definuje přednost a asociativitu výrazů. Nejsou zde uvedeny asymetrické konstrukty.

Název	Příklad výrazu	Přednost	Asociativita
název proměnné	`a`	1	
celočíselný literál	`1`	1	
floating point literál	`1.0`	1	
literál pole	`[a]`	1	
block	`{a}`	1	
indexace	`a[b]`	2	levá
negace	`!a`	3	pravá
plus	`+a`	3	pravá
minus	`-a`	3	pravá
umocnění	`a ** b`	5	pravá
násobení	`a * b`	6	levá
dělení	`a / b`	6	levá
modulo	`a % b`	6	levá
sčítání	`a + b`	7	levá
odečítání	`a - b`	7	levá
menší	`a < b`	8	levá
menší nebo rovno	`a <= b`	8	levá
větší	`a > b`	8	levá
větší nebo rovno	`a >= b`	8	levá
rovnost	`a == b`	9	levá
nerovnost	`a != b`	9	levá
konjunkce	`a && b`	10	levá
disjunkce	`a || b`	11	levá

Tabulka 4.1: Přednost a asociativita výrazů

4.5 Běh výpočetního programu

Primárním účelem výpočetního programu je výpočet a zápis nových vzorků deklarovaných *bufferů*. Tyto výpočty probíhají v tzv. *výpočetních krocích*. Při každém takovém výpočtu může navíc dojít ke změně hodnot deklarovaných výstupních proměnných.

Vstupem výpočetního programu jsou řídícím programem průběžně modifikované hodnoty deklarovaných vstupních proměnných, zápis do vstupních bufferů a signály iniciující inicializaci výpočtu a následující výpočetní kroky. Způsob zápisu do vstupních proměnných a bufferů a intervaly iniciace výpočtů specifikuje implementace řídícího programu. Podmínkou je iniciace inicializace výpočtu před prvním výpočetním krokem a zajištění synchronního běhu výpočetních kroků, tedy vyčkání na dokončení předchozího kroku před iniciací dalšího.

Výstupem výpočetního programu jsou jím modifikované hodnoty v deklarovaných výstupních proměnných, což zahrnuje i výstupní buffery. Využití tohoto výstupu specifikuje implementace řídícího programu.

Při iniciaci inicializace výpočtu se nastaví výchozí hodnoty ve všech deklarovaných bufferech a vstupních a výstupních proměnných. Diskrétní čas výpočtu t se nastaví na 0.

Při iniciaci výpočetního kroku se provedou výpočty vzorků v čase $t+1$ pro všechny buffery v nespecifikovaném pořadí. Následně se hodnoty bufferů posunou o jeden index níž, nejstarší hodnota se zapomene a nejnovější se zapíše na pozici 0. Implementačně jde o operaci v konstantním čase. Všechny výpočty hodnot bufferů v čase $t+1$ závisí na hodnotách bufferů v čase nejvýše t a vstupních proměnných.

4.6 Kontext

Kontext je souborem hodnot, typů a názvů. Obsah kontextu se může změnit s každým krokem programu. Hodnoty a typy se mohou *uložit* do kontextu samotné nebo s asociovaným názvem.

Každá hodnota je asociována se svým *typem*, který nelze změnit. V kontextu uloženou hodnotu nelze nahradit hodnotou (ani identickou) s jiným typem. Typ určuje hodnoty, které s ním mohou být asociovány. Některé hodnoty tak mohou spadat pod více typů, ale i přes to musí být asociovány s právě jedním z nich.

Referenční hodnoty, neboli *odkazy*, jednoznačně určují jednu či více v kontextu uložených hodnot. Jinými slovy na tuto hodnotu *odkazují*. Nahrazení odkazu jinou referenční hodnotou (jiným odkazem) neovlivní původně odkazované hodnoty. A naopak změna odkazovaných hodnot neovlivní odkaz. Odkazy jsou vždy *validní*, tedy dokud odkaz existuje, odkazuje na existující uloženou hodnotu.

4.7 Scope

Hodnoty, typy a názvy uložené v kontextu jsou rozděleny do disjunktních podmnožin, tzv. *scopů*.

Výrazy, resp. příkazy mohou při jejich vyhodnocení, resp. vykonání vytvořit nový scope. První, tzv. *nejvnější* scope je vytvořen samotným programem, tedy *nejvnějším blockem*.

Každý další nově vytvořený scope je dceřiným scopem toho předchozího. Po ukončení vyhodnocení, resp. vykonání výrazu, resp. příkazu, který scope vytvořil, musí tento scope zaniknout. To znamená, že v kontextu již nejsou uloženy hodnoty, typy a názvy z tohoto scopu. Jelikož se nikdy nevykonává více příkazů ani nevyhodnocuje více výrazů naráz, scopy jsou v každém kroku programu lineárně uspořádány.

4.8 Inicializace výpočetního programu

Celý program je definován jako block se syntakticky implicitními složenými závorkami. Syntaxe programu je tedy neuzávorkovaný seznam příkazů oddělených středníky.

Tento block se označuje jako *nejvnější* (anglicky *outermost*), neboli *globální* block. Označení globální je zavedeno kvůli podobnosti s C, ale ve skutečnosti není nijak sémantiky odlišný od ostatních bloků, proto je preferován pojem nejvnější.

Při spuštění programu se nejprve provede jeho *inicializace*, tedy vyhodnocení nejvnějšího blocku. Hodnota tohoto blocku reprezentuje výsledek inicializace programu. V aktuální verzi tato hodnota není nijak využita.

4.9 Alokace referenčních hodnot

Hodnoty vstupních a výstupních typů a buffery se *alokují* na úrovni programu. To znamená, že se neukládají do aktuálního scopu, ale do toho nejvnějšího. Odkazy na tyto hodnoty tak zůstávají být validními po dobu běhu celého programu. Sémantika alokace hodnot vstupních a výstupních typů a bufferů uvnitř definic funkcí není v aktuální verzi specifikována.

Hodnoty polí *alokují* na úrovni funkcí. To znamená, že ukládají do scopu odpovídajícího deklaraci funkce, nebo, pokud nejsou deklarovány ve funkci, do nejvnějšího scopu. Odkazy na tyto hodnoty zůstávají být validními po dobu běhu programu funkce.

4.10 Vyhodnocení výrazů a vykonání příkazů

Vyhodnocení výrazu dá jeho *hodnotu*. *Vykonání* příkazu může *vrátit* jeho *návratovou hodnotu*. Každý výraz má hodnotu, ale ne každý příkaz vrací.

Příkazy se *vykonávají* a výrazy se *vyhodnocují* vždy v rámci nějakého kontextu. Side effect vykonání příkazu či vyhodnocení výrazu je libovolná jim způsobená změna v kontextu. Není-li to explicitně uvedeno pro daný příkaz, resp. výraz, nemá jeho vykonání, resp. vyhodnocení žádné side effecty. *Vyhodnocení výrazu* vytvoří *dočasnou hodnotu*, tzv. *hodnotu výrazu*. Dočasné hodnoty nejsou uloženy v kontextu.

Syntaxe výrazu samotného může reprezentovat *příkaz výrazu*. Vykonání takového příkazu je ekvivalentní vyhodnocení daného výrazu. Hodnota výrazu se nijak nevyužije.

4.11 Hodnoty a typy

Výrazy i vyjádření typů jsou n-ticemi. Obsahují n-ticové atomy výrazů a vyjádření typů. Vyhodnocení výrazu dá hodnotu. Vyhodnocení vyjádření typu dá typ. Hodnoty i typy je také n-ticemi. Obsahují n-ticové atomy hodnot a typů.

Počet n-ticových atomů ve výrazu, resp. vyjádření typu nemusí odpovídat počtu atomů v odpovídající hodnotě, resp. odpovídajícím typu. Vyhodnocení n-tice výrazů, resp. vyjádření typů vyhodnotí jednotlivé n-ticové atomy a následně složí výslednou hodnotu, resp. výsledný typ jako n-tici obsahující vyhodnocené hodnoty, resp. typy atomů za sebou.

4.12 Cíle

Cíl reprezentuje odkaz na hodnotu. Syntaxe cíle je tvořena buď názvem proměnné, nebo subscriptem s odkazem daným názvem proměnné. Syntakticky tedy jde o podmnožinu výrazů. Cíl reprezentuje odkaz na hodnotu, která by byla výsledkem vyhodnocení odpovídajícího výrazu indexace. Syntaxe cílu se může vyskytovat jen součástí levé strany syntaxe přiřazení.

4.13 Subscript a výraz indexace

Syntaxe subscriptu obsahuje n-ticový atom pro výraz reprezentující jeho odkaz následovaný syntaxí literálu pole. Prvky tohoto pole reprezentují jednotlivé indexy subscriptu.

Subscript může reprezentovat konstrukt cíle, pokud je jeho odkaz dán názvem proměnné nebo dalším stejně rekurzivně omezeným subscriptem. Jinak reprezentuje *výraz indexace*.

Validní subscript je asociován s jeho *pododkazem* odkazujícím na jednu nebo více hodnot odkázaných jeho odkazem a navíc tomuto pododkazu přiřazuje typ. V případě prázdného subscriptu, tedy subscriptu bez indexů, jsou pododkazem odkázány všechny hodnoty odkázané jeho odkazem. Jinak celočíselná hodnota indexů subscriptu odpovídá pořadí hodnot v odkazu, na které má odkazovat pododkaz. Validní pododkaz subscriptu odkazuje na alespoň jednu asociovanou hodnotu.

Index hodnoty 0 odpovídá prvnímu prvku v poli, poslednímu prvku v bufferu a jedinému prvku v hodnotě vstupního či výstupního typu. V poli i v bufferu index o 1 větší index odpovídá dalšímu prvku v pořadí. V případě pole se tedy používají jen nezáporné indexy a v případě bufferu jen nezáporné.

Subscript pole nebo bufferu s jediným indexem má pododkaz typu odpovídajícímu typu obsaženému v daném poli. Jinak je jeho pododkaz typu pole stejné,

nebo menší velikosti. Subscript hodnoty vstupního nebo výstupního typu vždy odkazuje jen na jednu hodnotu a jeho odkaz je vždy typu obsažené hodnoty.

V aktuální verzi je subscript omezen na nula nebo jeden index.

Vyhodnocení výrazu indexace zkopíruje odkazovanou hodnotu a ta pak tvoří hodnotu výrazu.

4.14 Deklarace, definice, přiřazení a aliasy

Konstrukt deklarace může být součástí jiných konstruktů, jako např. smyčka for, nebo deklarace funkce, ale také tvoří samy o sobě validní příkaz. Syntaxe n-ticového atomu pro deklaraci je tvořena syntaxí vyjádření typu a názvem. Každý n-ticový atom deklarace reprezentuje *proměnnou*. Proměnná má dán název a hodnotu uloženou v kontextu. Vykonání příkazu deklarace vytvoří novou proměnnou s daným názvem a jako její hodnotu do aktuálního scopu uloží *výchozí hodnotu* odpovídající danému typu.

Hodnota výrazu *názvu proměnné* je právě ta v kontextu uložená hodnota asociovaná s daným názvem, která patří do nejnovějšího scopu. Hodnoty v novějších scopech tak mohou tzv. *zastínit* proměnné se stejnými názvy ve starších scopech.

Syntaxe příkazu *definice* je složena ze syntaxe deklarace, symbolu `=` a syntaxe výrazu reprezentující *inicializační hodnotu*.

Vykonání příkazu definice vytvoří nové proměnné ekvivalentně odpovídajícímu příkazu deklarace a navíc nastaví jejich hodnoty podle dané inicializační hodnoty. Jednotlivé n-ticové atomy inicializační hodnoty se nastaví jako hodnoty proměnných postupně zleva doprava. Jednotlivé n-ticové atomy typů a jejich počet na levé straně se musí shodovat s atomy typu inicializační hodnoty na pravé straně.

Syntaxe příkazu *přiřazení* je dána cílem symbolem `=` a *přiřazovaným výrazem*. Vykonání příkazu přiřazení nahradí v kontextu uložené hodnoty odkázané cíli příkazu na hodnoty dané přiřazovanou hodnotu. Jednotlivé n-ticové atomy inicializační hodnoty se nastaví jako hodnoty odkázané cíli postupně zleva doprava. Jednotlivé n-ticové atomy typů a jejich počet na levé straně se musí shodovat s atomy typu inicializační hodnoty na pravé straně.

Do proměnné nelze přiřadit hodnotu jiného typu, což plyne už z toho, že nelze nahrazovat hodnotu v kontextu za hodnotu jiného typu. Proměnnou nelze inicializovat z hodnoty jiného typu. Z toho a z předchozího plyne, že typ hodnoty proměnné bude vždy odpovídat typu uvedenému při deklaraci proměnné.

Syntaxe *aliasu typu* začíná klíčovým slovem *type* a pokračuje názvem typu, symbolem `=` a vyjádřením typu. Vykonání příkazu aliasu typu uloží do aktuálního scopu typ asociovaný s daným názvem typu. Typ v kontextu nelze nahradit jiným typem. Typové aliasy se mohou navzájem zastínit, jak je tomu i u proměnných.

Deklarace proměnné, resp. alias typu je validní pouze pokud daný název již není uložen v aktuálním scopu. Stejný název může být uložen více krát v kontextu, ale jen jednou v daném scopu.

4.15 Jednoduché typy

Typy *bool*, *int* a *float* jsou tzv. *jednoduchými typy*. Ostatní typy jsou typy *složenými*, jelikož jsou složeny z těch jednoduchých. Jednoduché typy nemají speciální syntaxi, ale lze je vyjádřit pomocí předdefinovaných názvů `Bool`, `Int` a `Float`. Tyto názvy typů se sémanticky neliší od jiných typových aliasů a lze je tedy zastínit. Výrazy *literálů* jednoduchých typů reprezentují hodnoty těchto typů.

Typ bool zahrnuje dvě pravdivostní hodnoty, konkrétně *true* a *false*. Odpovídají jim literály `true` a `false`. Výchozí hodnotou je false.

Typ int zahrnuje konečnou posloupnost všech celých čísel mezi minimem a maximem danými implementací překladače výpočetního programu. Celočíselný literál popsaný v sekci Tokeny reprezentuje právě tyto celočíselné hodnoty. Výchozí hodnotou je nula.

Typ float reprezentuje konečnou podmnožinu reálných čísel. Konkrétní rozsah čísel je dán implementací překladače výpočetního programu. Floating point literál popsaný v sekci Tokeny reprezentuje právě tyto floating point hodnoty. Výchozí hodnotou je floating point reprezentace nuly.

4.16 Imutabilní typy

Uvedení klíčového slova `const` za syntaxí typu x tvoří syntaxi imutabilního typu x. Imutabilní typy mohou být součástí jiných typů, které samy o sobě nemusí být imutabilními. Některé typy jsou implicitně imutabilní. Přidání klíčového slova `const` k jejich syntaxi nezmění jejich sémantiku. Typy které nejsou imutabilní jsou `mutabilní`. Pojem mutability se vztahuje i na hodnoty. Hodnota imutabilního typu je vždy imutabilní a naopak.

Např. `Int const` je syntaxí imutabilního intu. `Int [4] const` je syntaxí imutabilního pole čtyř. mutabilních intů. `Int const [4]` je syntaxí pole čtyř imutabilních intů. Typ `Int const` *obsahuje* typ `Int`.

Implicitně imutabilními jsou typy funkcí, vstupní a výstupní typy a typy bufferů.

V kontextu uloženou hodnotu imutabilního typu nelze nahradit jinou hodnotou.

Atomické typy a jejich imutabilní varianty jsou označovány jako *skalární typy*.

Imutabilní typy nemají výchozí hodnotu. Proměnné imutabilního typu nelze deklarovat bez explicitního uvedení jejich hodnoty.

4.17 Pole

Syntaxe typu pole obsahuje syntaxi vyjádření typu následované syntaxí výrazu v hranatých závorkách. Typ pole pak *obsahuje* typ daný v jeho syntaxí. Výraz v jeho syntaxi určuje *velikost* pole, neboli počet v něm *obsažených* prvků. Hodnota pole je odkazem na pevně daný počet prvků.

V aktuální verzi je typ obsažený v poli omezen na 1-tice skalárních typů. Velikost pole musí být dána 1-ticí typu int, kterou lze vyhodnotit za překladu.

Syntaxe literálu pole je seznamem prvků oddělených čárkami a uzávorkovaných hranatými závorkami. Prky mohou být syntaxe výrazů, nebo speciálních syntaktických konstruktů *rozmezí* a *rozbalení*. Syntaxe výrazů reprezentuje jednotlivé prvky výsledného pole. Syntaxe rozmezí obsahuje syntaxi výrazu reprezentujícího první prvek aritmetické posloupnosti, dále klíčové slovo `to` a syntaxi výrazu reprezentujícího horní, nebo dolní závoru této posloupnosti. Navíc může obsahovat dále klíčové slovo `by` následované syntaxí výrazu reprezentujícího diferenci posloupnosti. Syntaxe rozbalení obsahuje symbol `...` následovaný výrazem určujícím pole. Syntaxe rozmezí i rozbalení pak určují jednotlivé prvky výsledného pole v daném pořadí.

Výchozí hodnotou pole je pole zaplněné výchozími hodnotami obsaženého typu.

4.18 Vstupní a výstupní typy

Hodnota *vstupního typu* je odkazem na hodnotu odpovídající *ovládacímu prvku* v řídícím programu. Odkázaná hodnota je z pohledu uživatele imutabilní. Přiřazení do cíle daným subscriptem takové hodnoty je zakázáno. Řídící program ale může tuto hodnotu mutovat.

Hodnota *výstupního typu* je odkazem na hodnotu odpovídající *monitorovacímu prvku* v řídícím programu. Odkázaná hodnota je uživatelem mutabilní, ale nelze ji ve výpočetním programu nijak využít. Indexace takové hodnoty je zakázána.

Vstupní typy mohou obsahovat jen jednoduché typy a buffery. Výchozí hodnoty vstupních a výstupních typů určí řídící program.

4.19 Ekvivalence typů a typové konverze

Na typech jsou definovány dvě ekvivalence - *podobné typy* a *identické typy* (neboli *stejné typy*). Jednoduché typy jsou identické i podobné jen sami se sebou. Imutabilní typy jsou identické pokud obsahují stejné typy. Pole jsou identická pokud mají stejnou velikost a obsahují identické typy. Buffery jsou identické pokud mají stejnou velikost. Vstupní, resp. výstupní typy jsou identické pokud obsahují identické typy. Funkce jsou identické pokud mají identické vstupní a výstupní typy. Všechny mezi sebou identické typy si jsou i podobné. Navíc imutabilní typy jsou podobné jim obsaženým typům. Funkce jsou si podobné, pokud mají podobné vstupní a výstupní typy.

Podobnost typů tedy povoluje neshodu v mutabilitě. Nezahrnuje to ale mutabilitu obsažených typů. Tedy např. Typ `Int [3]` je podobný typu `Int [3] const`, ale ne typu `Int const [3]`.

Při kopii hodnoty lze novou hodnotu asociovat s podobným typem implicitně. Tedy například definice imutabilního intu z mutabilního přidá implicitně imutabilitu k inicializační hodnotě. Jakoukoliv jinou změnu typu hodnoty je (v aktuální verzi) třeba uvést explicitně výrazem *explicitní typové konverze*.

Syntaxe výrazu *explicitní typové konverze* obshuje syntaxi vyjádření typu následované povinně uzávorkovaným výrazem. Vyhodnocení tohoto výrazu provede

typovou konverzi hodnoty dané uzávorkovaným výrazem. Explicitní typová konverze může změnit i hodnotu, jde-li o její kopii.

Při konverzi mezi typy bool a int se hodnota true nahradí hodnotou 1 a hodnota false hodnotou 0. Naopak hodnota 0 se nahradí hodnotou $false$ a ostatní celočíselné hodnoty se nahradí hodnotou 1.

Při konverzi mezi z typu float na typ int dojde k zaokrouhlení floating point hodnoty směrem k nule. Naopak při konverzi z int na float se celočíselná hodnota nahradí jí nejbližší floating point hodnotou.

Při konverzi mezi typy bool a float dojde nejprve ke konverzi na typ int a dále se postupuje dle výše popsané sémantiky.

Konverze polí může zmenšit velikost a přidat imutabilitu obsažené hodnoty. Konverze pole kopíruje odkaz, ale ne obsažené hodnoty.

Konverze bufferu může jen zmenšit velikost.

Funkci lze konvertovat na buffer libovolné velikosti, pokud má funkce výstupní typ float a vstupní typ int, nebo prázdnou n-tici (resp. typově podobné ekvivalenty). Taková konverze má jako jediná side effect. Vytvoří novou hodnotu bufferu a uloží ji do nejvnějšího scopu.

Obecně předání a odebrání imutability typu je možné až na typy obsažené v jiných typech. V případě funkcí lze volně přidávat i odebírat mutabilitu vstupních a výstupních typů.

Ostatní konverze nejsou možné.

4.20 Block

Block je hybridní konstrukt (výraz nebo příkaz) obsahující posloupnost příkazů. Tyto příkazy se označují jako jeho *vnitřní příkazy*. Syntaxe blocku je seznam syntaxí příkazů oddělených středníky a uzavřených složenými závorkami. Prázdný seznam určuje prázdný block bez vnitřních příkazů. Každý prvek v seznamu navíc určuje další vnitřní příkaz blocku.

Vykonání příkazu blocku i vyhodnocení hodnoty blocku vytvoří nový scope a vykoná jeho příkazy v daném pořadí.

Hodnota výrazu blocku i návratová hodnota příkazu blocku je dána návratovou hodnotou jeho prvního vykonaného vnitřního příkazu, který vrací.

Program není validní, pokud žádný takový příkaz ve výrazovém bloku není.

Příkazový block vrací vždy, pokud obsahuje vnitřní příkaz, který vrací vždy. Příkazový block nevrací, pokud neobsahuje žádný vnitřní příkaz, který vrací.

Všechny vnitřní příkazy se musí shodovat v typu návratové hodnoty.

Příkazy po prvním vždy vracejícím příkazu se ignorují.

4.21 Podmínky a smyčky

Konstrukt *when* je příkazem reprezentujícím podmíněné vykonání větve programu. Syntaxe příkazu when začíná klíčovým slovem `when` následovaným povinně uzávorkovanou hodnotou reprezentující podmínku a dále příkaz reprezentující jeho *větev*. Podmínka musí být dána 1-ticí typu bool. Je-li podmínka vyhodnocena jako pravdivá, daný příkaz se vykoná, jinak nevykoná. Větev podmínky vytváří nový scope. Příkaz when vrací, pokud je vykonána a vrací jeho větev.

Konstrukt *while* je také příkazem a reprezentuje opakované vykonání části programu. Syntaxe příkazu while začíná klíčovým slovem `while` následovaným povinně uzávorkovanou hodnotou reprezentující podmínku a dále příkaz určující jeho *tělo*. Podmínka musí být dána 1-ticí typu bool. Dokud je podmínka vyhodnocena jako pravdivá, bude daný příkaz dokola vykonáván ve vlastním vytvořeném scopu. Příkaz while vrací, jakmile vrací jeho tělo. Pokud v žádné iteraci tělo nevrátilo, celý příkaz nevrací.

If a *for* je hybridní konstrukty. Mohou reprezentovat výrazy i příkazy. Jsou výrazy, obsahují-li jejich syntaxe jen výrazy. Jinak jsou příkazy.

Syntaxe if začíná klíčovým slovem `if` a pokračuje povinně uzávorkovaným výrazem podmínky. Dále obsahuje výraz nebo příkaz kladné větve, klíčové slovo `else` a výraz nebo příkaz záporné větve. Je-li podmínka vyhodnocena jako pravdivá, vyhodnotí, resp. vykoná se kladná větev, jinak záporná. Obě větve vytváří vlastní scope. Hodnotou výrazu if je hodnota vykonané větve. Příkaz if vrací, pokud vrací alespoň jedna z jeho větví.

Syntaxe for začíná klíčovým slovem `for` a pokračuje povinně uzávorkovanou iterační deklaraci. Dále obsahuje výraz nebo příkaz určující jeho tělo.

Vykonání příkazu for, resp. vyhodnocení výrazu for opakovaně vykoná, resp. vyhodnotí jeho tělo v iteracích. V každé iteraci se vytvoří scope pro proměnné deklarované iteračními deklaracemi a dále v něm scope pro tělo. Každý n-ticový atom iterační deklarace je syntakticky tvořen deklarací jedné proměnné následované klíčovým slovem `in` a výrazem určující 1-tici typu pole. Každá taková proměnná se definuje na začátku každé iterace s inicializační hodnotou danou dalším prvkem v uvedeném poli v pořadí. Lze tak iterovat přes několik polí naráz.

Hodnotou výrazu for je pole obsahující prvky dané hodnotami všech iterací vyhodnocení těla v daném pořadí. Příkaz for vrací v okamžiku, kdy vykonání jeho těla vrací. Pokud v žádné iteraci tělo nevrátilo, celý příkaz for nevrací.

4.22 Definice a aplikace funkce

Syntaxe typu funkcí je složena ze syntaxe *výstupního typu funkce* následované povinně uzávorkovanou syntaxí *vstupního typu funkce*. Funkce reprezentuje zobrazení, jehož definiční obor, resp. obor hodnot je dán jejím vstupním, resp. výstupním typem. Výraz aplikace funkce je validní jen s argumenty jejího vstupního typu. Typ hodnoty aplikace funkce je dán jejím výstupním typem.

Výraz, resp. příkaz definice funkce při jeho vyhodnocení, resp. vykonání vytvoří

hodnotu typu funkce. Hodnota funkce obsahuje tzv. *nekompletní program* a tzv. *zachycený kontext*, neboli *closure*. Nekompletní program je program, který může obsahovat chybějící části. Doplněním těchto částí se může stát validním programem.

Syntaxe výrazu i příkazu definice funkce obsahuje syntaxi výstupního typu funkce, povinně uzávorkovanou syntaxi deklarace vstupních *parametrů* a syntaxi *těla* funkce. Výraz aplikace funkce obsahuje *argumenty*, tedy hodnoty, které doplní chybějící inicializační hodnoty parametrů. Syntaxe příkazu definice funkce navíc obsahuje název. Vykonání příkazu definice funkce navíc definuje novou proměnnou s hodnotou danou funkcí, kterou reprezentuje.

Podprogram hodnoty funkce je tvořen blockem obsahujícím jako vnořené příkazy definice parametrů s chybějícími inicializačními hodnotami následované příkazem vrácení s tělem funkce jako vraceným výrazem. Následující příklad demonstruje pro funkci výrazu `Int fn (Int x, Int y) { return x + y }` její nekompletní program. Podtržítka označují chybějící argumenty.

```
{
    Int x = _;
    Int y = _;
    return {
        return x + y;
    };
}
```

Program funkce s doplněnými argumenty vždy tvoří výraz. Výraz aplikace funkce má hodnotu ekvivalentní vyhodnocení programu funkce s danými argumenty a kontextem zachyceným při definici funkce.

Doplnění argumentů všech typů kromě polí je ekvivalentní použití výrazů argumentů na místě inicializačních hodnot. Hodnoty argumentů se tedy zkopírují, ale pokud jde o odkaz, pak jím odkazované hodnoty nejsou kopírovány. V případě polí se vytvoří nový odkaz odkazující na zkopírované obsažené hodnoty, které se uloží do scopu funkce. Pro výraz argumentu `a` to odpovídá doplnění výrazu `a[]` jako inicializační hodnoty parametru. Při vrácení funkce se také provede kopie obsažených hodnot. Zkopírované hodnoty se uloží do scopu vnější funkce, resp. nejvnějšího blocku, pokud žádná vnější funkce není vyhodnocována.

Při zachycení kontextu se provede kopie všech hodnot v aktuálním kontextu do nového zachyceného kontextu. Odkazy na nestatické hodnoty v novém kontextu se nahradí odkazy odkazujícími na odpovídající kopie v zachyceném kontextu. Odkazy na statické hodnoty stále odkazují na stejné staticky uložené hodnoty. Každá mutabilní hodnota se v zachyceném kontextu nahradí její imutabilním ekvivalentem.

4.23 Buffery a generátory signálů

Syntaxe typu buffer obsahuje klíčové slovo `buffer` následované syntaxí výrazu v hranatých závorkách. Výraz v jeho syntaxi určuje *velikost* bufferu, neboli počet v

něm *obsažených* prvků. Obsažené prvky jsou vždy typu float. Hodnota bufferu je odkazem na pevně daný počet prvků. Velikost bufferu musí být dána 1-ticí typu int, kterou lze vyhodnotit za překladu.

Buffer velikosti n reprezentuje historii posledních n vzorků nějakého signálu. Tento signál lze určit jeho tzv. *generátorem*. Funkce se stane generátorem, pokud se konvertuje na buffer. V každém hlavním výpočetním kroku programu se pak spočítá hodnota aplikace generátoru na aktuální pozici vzorku, nebo na prázdnou n-tici a tato hodnota se zapíše jako nová hodnota bufferu. Jiný způsob zápisu do bufferu není možný. Přiřazení do cíle daného subscriptem bufferu je zakázán.

Speciálním případem vstupních a výstupních typů jsou vstupní a výstupní buffery. Jde o jediné vstupní a výstupní typy obsahující neskalární hodnoty. Výstupní buffery reprezentují celkový zvukový výstup programu, vstupní reprezentují zvukový vstup.

Aktuální řídící program podporuje jen výstupní buffery.

4.24 Výpočty za překladu

Deklarace proměnné imutabilního typu s inicializační hodnotou známou za překladu vytváří proměnnou s hodnotou známou za překladu. Hodnota výrazu je známa za překladu, neobsahuje-li žádné proměnné, nebo jen proměnné s hodnotou známou za překladu. Příkazy mají návratovou hodnotu známou za překladu, obsahují-li jen výrazy s hodnotami známými za překladu. Hodnoty známé za překladu se označují jako *kompilační konstanty*.

5. Implementace

V této kapitole bude popsán přehled základních postupů implementace.

5.1 Jazyk a nástroje

Překladač Cynthu je napsán v C++20. Využívají se jen obecně přístupné vlastnosti a standardní knihovny standardu C++20. Překlad je otestován na překladačích GCC 10, Clang 10 a 11. Poslední verze MSVC by měla také podporovat vše relevantní z C++20, ale není to otestováno. Části programu překladače jsou generovány nástroji Bison (konkrétně verze 3.7.1) a Flex (verze 2.6.1). K generaci kódu a překladu kompletního programu jsou připraveny Makefily. Makefile pro překlad kompletního programu navíc spoléhá na soubory se závislostmi mezi implementacemi a headery vygenerované pomocí GCC. Překladač Cynthu vygeneruje kód v jazyce C (konkrétně v dialektu gnu17), který je následně potřeba přeložit do spustitelného souboru. Dialekt gnu17 podporuje jak GCC tak Clang.

5.2 Platforma

Program dle původního rozsahu cílil na platformu Windows a zvukové karty podporující technologii ASIO. Jelikož je aktuální verze programu omezená jen na překlad základní emulace bez propojení se zvukovou kartou, tato omezení odpadají. Výsledný C program je multiplatformní až na jedno systémové volání. Kvůli němu je třeba v kompilační konfiguraci `inc/config.hpp` nastavit cílovou platformu, podle čehož se ve výsledném programu vygeneruje `#include` headeru `<windows.h>` nebo `<unistd.h>`. Mezikrok s překladem do C umožňuje využít následující optimalizace překladače C.

5.3 Fáze překladu

Samotný překlad ze Cynthu do C je rozdělen na několik fází: Lexikální analýza rozdělí vstupní text na tokeny. Syntaktická analýza dle vstupních tokenů vytvoří *AST* (Abstract Syntax Tree) reprezentované v C++ strukturách. AST struktury se dále mohou transformovat do pomocných sémantických struktur, nebo rovnou překládat do úryvků C kódu. Sémantické struktury se buď využívají na validaci částí programu za překladu, nebo přímo vyhodnocování částí programu za překladu. Některé sémantické struktury se dále přeloží do úryvků C kódu. Nakonec se všechny vytvořené úryvky C kódu sestaví do výsledného C programu. Tento program je obsažen v jediném `.c` souboru.

Pro další slinkování s řídícím programem se využije header soubor (`.h`) s interface, skrze které by řídící program obdrží informaci o vygenerovaných statických datech (vstupní a výstupní typy a buffery) a následně do nich může zapisovat, nebo z nich číst.

5.4 Struktura implementace překladače

5.4.1 Interface AST a sémantických struktur

Pomocné struktury mají pro výše popsané přechody určený přesný interface. Všechny struktury jsou v C++ implementovány jako obyčejné structy a polymorfismu je docíleno pomocí `std::variant`ů. Každý takový struct implementuje potřebnou funkcionalitu jako metodu. Tyto metody se nevolají přímo, ale přes daný soubor volných funkcí, představující interface různých typů struktur, které se aplikují přes `std::visit`. *Přesněji řečeno je implementována utilita, která umožňuje „lift“ funkcí nad různé typy, mezi něž patří i variant s implementací aplikace funkce visitor patternem.*

Přechod z AST struktury do sémantické struktury označuji jako **resolution**, přechod do sémantické struktury nebo C kódu (v závislosti na možnostech dané AST struktury) jako **processing** a přechod ze sémantických struktur do C kódu jako **translation**. Snažil jsem se držet jednotného přístupu, ale časem se ukázalo, že jsou potřeba stále nové konstrukty. Ve výsledku interface není tak jednoduchý. Kromě uvedených případů se také řeší: Získávání hodnot kompilačních konstant, kontrola shody typů a extrakce volných proměnných. Dále je funkcionalita deklarací a definicí rozdělena mezi struktury příslušných typů. Pseudokódem, jde zhruba o následující interface:

```
processStatement    (Node) -> ()
processExpression   (Node) ->  (Value | TypedExpr)
processArrayElement (Node) -> [(Value | TypedExpr)]

resolveType               (Node) -> Type
resolveDeclaration        (Node) -> Declaration
resolveRangetDeclaration (Node) -> RangeDeclaration
resolveTarget             (Node) -> Target

translateValue (Value) -> TypedExpr
translateType  (Type)  -> Name

valueType (Value)       -> Type
get       (Value) <Out> -> Out

sameTypes  (Type, Type) -> Bool

processDefinition  (Type, ResolvedValue) -> ()
processDeclaration (Type)                -> ()

extractNames     (Node) -> [Name]
extractTypeNames (Node) -> [Name]
```

Tyto funkce se spouštějí rekurzivně od kořene programu (nejvnějšího bloku). Pro kompletní překlad je aktuálně potřeba až 3 průchody AST. Jeden průchod vyřeší základní validaci i překlad zároveň. Další dva průchody jsou potřeba na extrakci volných jmen proměnných a jmen typů. Bylo by ale možné je spojit do jednoho průchodu. Spojit extrakci jmen i překlad by ale mohlo být nevýhodné. Aktuálně

totiž překladač ignoruje mnohé části kódu, které se nikdy nespustí a provede na nich jen extrakci, což lehce urychluje překlad.

5.4.2 Kontext

Dále je velmi podstatnou částí překladových algoritmů tzv. **context** (neboli **kontext**). Kontext je soubor několika struktur, které řídí data sdílená mezi rekurzivními voláními výše uvedených funkcí. **Global context** obsahuje globální úryvky C kódu (globální alokace dat, definice typů a funkcí, includy standardních headerů), globální počítadlo pro unikátní identifikátory jmen a informace o generátorech. Vytvoří se jednou na začátku překladu a používá se po celou dobu běhu překladu. **Function context** obsahuje úryvky C kódu pro alokace na úrovni funkce, a také aktuálně deklarované parametry odpovídající funkce. Vytvoří se jednou při každé deklaraci (nebo compile-time evaluaci) funkce. Po dokončení deklarace či evaluace se odstraní. **Branching context** obsahuje aktuální číslo větve pro komplexnější případy vracení. Vytvoří se jednou při zpracování výrazových blocků a následně se také odstraní. **Lookup context** obsahuje definice proměnných (a také typů), jak jsou deklarované a viditelné z pohledu Cynth kódu, a k nim buď kompilační konstantu, nebo název odpovídající proměnné v C kódu.

Nakonec **main context** spojuje tyto kontexty dohromady, a také obsahuje úryvky C kódu pro lokální příkazy. Main context obsahuje odkazy na global context, function context a branching context a pak přímo obsahuje lookup context. Main context spolu se svým lookup contextem odpovídá scopům v Cynth kódu. Pro nový scope se vytvoří dceřiný main context i s novým, odpovídajícím lookup contextem. Ostatní kontexty jsou tvořeny mimo main context dle potřeby.

Tyto kontexty shromažďují úryvky C kódu roztříděné na různé kategorie (typy, funkce, atd), které se na konci složí do výsledného programu.

6. Evaluace

Měření výkonu výsledných programů bude provedeno pro ověření splnění požadavku dostatečně nízké latence. Jak bylo v úvodní části práce zmíněno, člověk je schopen vnímat negativní vliv zpoždění při hře na nástroj (v tomto případě MIDI ovladač digitálního syntezátoru) už při hodnotách těsně nad 10 ms. Proto by výsledné programy běžných komponent syntezátorů rozhodně neměly mít zpoždění v řádu milisekund. Nebude však určena žádná přesná hranice latence. Výsledky budou tedy spíše orientační.

6.1 Faktory ovlivňující latenci

Při stisku klávesy MIDI ovladače ho zpracuje hardware daného ovladače a vyšle signál, na který musí zareagovat řídící program. Ten ho dál zpracuje a identifikuje a dle jeho významu změní hodnoty alokovaných vstupních proměnných ve výpočetním programu. Výpočetní program na tuto změnu zareaguje při dalším hlavním kroku výpočtu, což odpovídá zpoždění jednoho vzorku.

Řídící program v pravidelných intervalech začíná počítat n vzorků, tedy spouštět hlavní výpočetní kroky pro dané vzorky. Počet vzorků n je dán ovladačem ASIO. Těchto n vzorků tvoří jeden ASIO buffer. Po dokončení výpočtů se výsledné vzorky předají ovladači ASIO, který se postará o jejich zpracování na zvukové kartě. Výpočet následujících n vzorků začne ve chvíli, kdy to ovladač ASIO určí.

Čas od okamžiku, co řídící a ovládací program zareaguje na MIDI signál během výpočtu ASIO bufferu, do předání signálem ovlivněných vzorků zvukové kartě je dán počtem vzorků v ASIO bufferu a vzorkovací frekvencí. Např. při bufferu o 512 prvcích a běžné vzorkovací frekvenci 44 100 Hz se následující buffer začne počítat až po cca 12 ms.

Výslednou latenci tedy hlavně ovlivní velikost ASIO bufferů a vzorkovací frekvence. I kdyby Cynth program stihl výpočet n bufferů během několika nanosekund, stále půjde o celkovou latenci 12 ms a výš.

Použití menších bufferů by mohlo umožnit dosáhnutí menší latence. Na druhou stranu výpočty větších bufferů mohou být v průměru rychlejší. Optimální volba velikosti bufferů tedy silně závisí na platformě.

6.2 Metody měření

Měřit se bude čas potřebný pro výpočet všech vzorků ASIO bufferu. Toto měření se provede na určitém počtu bufferů a z toho se budou analyzovat nejpomalejší, nejrychlejší a průměrné výpočty. Nejpomalejší výpočty dají vhodný horní odhad možností výpočetního programu, zatímco průměrné výpočty dají lepší představu běžných případů při použití dostatečně velkých ASIO bufferů.

Provede se měření na příkladu složeném z běžně používaných komponent syntezátorů. Ve výsledcích se uvede výčet těchto komponent a přehled reálně alokovaných

bufferů, pro které výpočty probíhají.

Jelikož nejde o měření na dedikovaném hardwaru a real-time operačním systému, nelze z výsledků měření odvodit obecný odhad času. Při zátěži operačního systému jinými procesy může výkon značně klesnout v libovolném okamžiku. Proto navíc budou ilustračně provedeny měření s jinými programy v pozadí. Pro simulaci běžné situace bude spuštěn DAW program Ableton Live. Postup takového měření ale není nijak přesně specifikován a půjde o čistě orientační výsledky.

6.3 Výsledky

Veškerá měření byla provedena na vzorkovací frekvenci 44 100 Hz a velikosti bufferu 512 vzorků. Byl zvolen demonstrační Cynth program, který je k nalezení v souboru `examples/benchark/composed.cth`. Měření proběhla na 100 000 bufferech. Každé měření tak trvalo cca 19 minut. Měření proběhla na operačním systému Windows na počítači s procesorem Intel Core i7-10510U o taktovací frekvencí 1,80 GHz a 16 GB operační paměti.

První měření proběhlo při relativně nízké zátěži počítače. Nejdelší výpočet všech vzorků bufferu trval 23,1 ns, průměrný výpočet trval 3,9 ns.

Druhé měření proběhlo při vyšší zátěži počítače. Paralelně byl spuštěn DAW program Ableton Live s rozpracovaným projektem, na kterém byly prováděny drobné úpravy. Nejdelší výpočet všech vzorků bufferu trval 47,4 ns, průměrný výpočet trval 6,8 ns.

6.4 Diskuze

6.4.1 Výsledky měření

Ve druhém měření byl průměrný čas skoro dvakrát delší než v prvním a ten nejhorší čas dva krát delší. Z takto omezeného měření nelze s jistotou dojít k žádným specifickým závěrům, ale lze říct, že nedošlo ke značně většímu zpomalení v nejhorším případě než v tom průměrném. To by mohlo naznačovat, že obecně zátěž systému neovlivní čas výpočtu náhlými záseky, jelikož nejhorší čas výpočtu nevzrostl neúměrně tomu průměrnému.

Pro ladění specifického výsledného programu je měřítko nejhoršího času vhodným ukazatelem použitelnosti v reálné situaci. Pokud program po dostatečně dlouhou dobu poběží s nejhorším výpočtem pod požadovanou hodnotou, pak lze předpokládat, že výpočty potrvají déle jen ve výjimečných případech.

6.4.2 Možné vylepšení měření

Vhodným k měření pro určení použitelnosti výsledného programu v embedded prostředí by bylo provedení následujících akcí před měřením: Zamknutí stránek v paměti, spuštění výpočtu v režimu real-time scheduling prioritě, uzamknutí výpočtu na jedno jádro procesoru a nastavení fixní frekvence procesoru.

Dále by bylo vhodné změřit opravdový čas mezi stiskem klávesy MIDI ovladače a změnou audio výstupu. To by bylo možné provést pomocí zvukové nahrávky jiným zařízením, na které by šlo identifikovat zvuk úhozu klávesy MIDI ovladače a začátek výsledného audio signálu.

Taková měření už ale jsou nad rámec obsahu této práce.

6.4.3 Výhody návrhu

Mnohé aspekty návrhu se opravdu osvědčily být praktickými. Použití blocků jako výrazů se osvědčilo kvůli mnohým deklaracím globálně alokovaných komponent. Použití blocku pro definici komponenty i bez použití funkce tak zabraňuje kontaminaci globálního scopu.

Dále výraz smyčky se ukázal být užitečným pro popis polí. Celkově se navržená sémantika polí zdá být dostatečnou pro popis syntezátorů i přes to, že nebylo implementováno vše z původního návrhu.

Popis signálů pomocí bufferů působí přehledně a vhodně omezuje možné operace se sdílenými daty.

6.4.4 Nedostatky návrhu

Velkým nedostatkem aktuálního návrhu je nemožnost sebeodkazování generátoru na vlastní buffer. V návrhu je zmíněno, že pro definice syntezátorů není rekurze nutná. To je pravda v případě rekurze funkcí, ale možnost rekurzivního odkazu na starší vzorky v definici bufferu je nezbytná k popisu rekurzivních, neboli filtrů s nekonečnou impulzní odezvou (IIR filtrů). Takové filtry jsou důležitou součástí analogových syntezátorů, jako např. ikonický *Moog ladder* filtr.

Základní princip Moog filtru lze digitálně reprezentovat poměrně jednoduchým algoritmem (Välimäki a Huovilainen, 2006). Jaká koliv implementace libovolného IIR filtru ale vyžaduje *feedback*, neboli *zpětnou vazbu*, což v aktuálním návrhu jazyka Cynth nelze provést.

Nešlo by však o příliš velkou změnu. Pokud se lépe promyslí možnost mutability bufferů, mohlo by být možné deklarovat buffer bez definice jeho generátoru. V následujícím přiřazení generátoru by již bylo možné v rámci existující sémantiky jazyka odkázat na takto deklarovaný buffer.

6.4.5 Čas implementace

Největší překážkou k dokončení práce byl špatný odhad náročnosti implementace navrženého jazyka. Implementace samotného jazyka tak zabrala většinu času z celé práce. Lepším přístupem by od začátku mohl být nějaký více inkrementální návrh jazyka, který by byl funkční už v nějaké jednoduché podobě. V případě nedostatku času by tak bylo možné část implementace vynechat na úkor vyladění ostatních aspektů práce.

Závěr

Výsledkem práce je překladač poměrně komplexního jazyka popisujícího zvukové syntezátory. Jazyk se ukázal být použitelným pro popis běžných důležitých konstruktů. Výkon přeložených a sestavených programů byl měřen za různých podmínek. Ukázalo se, že čas provedení výpočtů je vyhovující původním požadavkům.

Výsledek má také mnohé nedostatky. Většinu z nich způsobil špatný odhad náročnosti implementace, což omezilo časové možnosti práce na ostatních aspektech projektu. Ve výsledku ale práce ukazuje, že navržený koncept může v praxi fungovat se splněním uvedených požadavků.

Úspěšně byl implementován překladač jazyka a k němu navíc jednoduchý řídící program demonstrující jeho využití. Dalším možným rozvojem projektu může být dokončení řídícího programu a jeho konfigurace.

Seznam použité literatury

Ashihara, K., Kurakata, K., Mizunami, T. a Matsushita, K. (2006). Hearing threshold for pure tones above 20 khz. *Acoustical science and technology*, **27**(1), 12–19.

Burgel, C.-C., Bartholomaus, R., Fiesel, W., Hilpert, J., Hoelzer, A. a Linzmeier, K. (2001). Beyond cd-quality: Advanced audio coding (aac) for high resolution audio with 24 bit resolution and 96 khz sampling frequency. In *Audio Engineering Society Convention 111*. Audio Engineering Society.

Hummel, Z. (2016). Audio software (vst plugin) development with practical application.

Jack, R. H., Mehrabi, A., Stockman, T. a McPherson, A. (2018). Action-sound latency and the perceived quality of digital musical instruments: Comparing professional percussionists and amateur musicians. *Music Perception: An Interdisciplinary Journal*, **36**(1), 109–128.

Microsoft. Setpriorityclass function (processthreadsapi.h). URL `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setpriorityclass`.

Orlarey, Y., Fober, D. a Letz, S. (2009). Faust: an efficient functional approach to dsp programming.

Pekonen, J. a Välimäki, V. (2011). The brief history of virtual analog synthesis. In *Proc. 6th Forum Acusticum, Aalborg, Denmark*, pages 461–466.

Pradel, M. a Sen, K. (2015). The good, the bad, and the ugly: An empirical study of implicit type conversions in javascript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Shen, J., Hamal, M. a Ganzenmüller, S. Dynamic memory allocation on real-time linux. *Architecture*, **86**, 32.

Smith, S. W. a kol. (1997). The scientist and engineer's guide to digital signal processing.

Steinberg. 3rd-party developers support & sdks. URL `https://www.steinberg.net/developers/`.

Välimäki, V. a Huovilainen, A. (2006). Oscillator and filter algorithms for virtual analog synthesis. *Computer Music Journal*, **30**(2), 19–31.

Welch, T. B., G., W. C. H. a Morrow, M. G. (2016). *Kapitola 6*. CRC Press.

Zölzer, U. (2008). *Digital audio signal processing*. John Wiley & Sons.

Seznam obrázků

Seznam tabulek

A. Přílohy

A.1 Cynth.zip

Zip archiv obsahující kompletní zdrojový kód programu. Části programu, jako vygenerovaný parser a lexer nebo závislosti pro Makefile, jsou přítomny v archivu a není třeba je opět sestavovat. Žádný spustitelný program přítomen není. Výsledný překladač i výsledné přeložené emulace syntezátorů je třeba sestavit pomocí Makefilu.

Archiv je nahrán prostřednictvím SISu. Navíc ale ještě uvedu odkaz na osobní repozitář na GitHubu: `https://github.com/egst/cynth`.