FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Překlad QuakeC do Rustu
Student:	Martin Taibr
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Seznamte se s jazyky QuakeC a Rust. Analyzujte a navrhněte vhodný přístup pro překlad jazyka QuakeC do Rust s podporou preprocessingových maker. Při překladu optimalizujte čitelnost - omezte globální proměnné, zachovejte datové typy, rozdělte kód do modulů a další pomocí konstrukcí dostupných v jazyce Rust. Navržený překladač implementujte. Při implementaci vycházejte z projektu c2rust a gramatiky QuakeC dostupné v projektu jcompile. Překladač testujte na projektech v jazyce QuakeC jako je Xonotic.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 13. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Překlad QuakeC do Rustu

Martin Taibr

Katedra teoretické informatiky
Vedoucí práce: Ing. Radomír Polách

16. května 2019

Poděkování

Rád bych poděkoval svému vedoucímu, Ing. Radomíru Poláchovi, za ochutu vést téma, které jsem si vymyslel. Zároveň bych chtěl poděkovat své rodině za veškerou podporu během psaní práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2019 ……

Odkaz na tuto práci

Taibr, Martin. *Překlad QuakeC do Rustu*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se zaměřuje na nalezení vhodného přístupu k překladu jazyka QuakeC do jazyka Rust a implementaci prototypu takového překladače. Překladač se snaží zachovat čitelnost kódu a podporuje překlad některých maker preprocesoru jazyka C (která využívá QuakeC) do deklarativních maker v jazyce Rust.

Klíčová slova preprocesorová makra, deklarativní makra, konverze maker, QuakeC, Rust, transpilace

Abstract

This thesis focuses on finding an appropriate approach to translating the QuakeC language to the Rust language and implementing a prototype of such a compiler. The compiler tries to maintain the code's readability and supports translating some C preprocessor macros (used by QuakeC) into declarative macros in Rust.

Keywords preprocessor macros, declarative macros, macro conversion, QuakeC, Rust, transpilation

Obsah

Seznam tabulek

Seznam výpisů kódu

Úvod

Neustále vznikají nové programovací jazyky, které slibují (ať už oprávněně nebo ne) vyšší produktivitu programátorů, menší počet chyb nebo například vyšší rychlost programů. Existující jazyky se také vyvíjí, ale často jsou omezené zpětnou kompatibilitou. Návrháři programovacích jazyků se proto musí rozhodnout jestli je vhodnější vylepšovat existující jazyk a nebo vytvořit nový – ne vždy je totiž možné jazyky zlepšovat přidáváním nového syntaxu a nových konstrukcí, někdy je potřeba měnit nebo odebírat. Pokud zvolí vytvoření nového, nabízí se otázka, co udělat se softwarem v tom starém. Často se jedná o stovky tisíc, miliony nebo více rádků kódu a není ekonomické je překládat do jiného jazyka ručně. Jako možné řešení se nabízí vytvořit program, který kód přeloží automaticky, což nemusí být snadné, ale i tak může vyžadovat řádově menší úsilí.

Jedním příkladem tohoto přístupu je projekt C2Rust, který umožňuje přeložit C do Rustu. Rust je relativně nový programovací jazyk, přesto si jeho autoři kladou vysoké cíle – snaží se nahradit C a C++. Tyto dva jazyky umožňují psát velmi efektivní kód, to však má svou cenu. Pokud programátor udělá chybu, často je velmi těžké ji objevit během kompilace nebo testováním. Některé chyby se projeví pouze na určitých platformách, za použití specifického kompilátoru nebo pokud je progrm zkompilovaný s optimalizacemi. Tento jev je známý jako nedefinované chování a mnoho programátorů ho bere jako malou cenu, kterou platí za rychlé programy, a to přesto, že mnoho takových chyb je bezpečnostní povahy a umožňují útočníkovi převzít kontrolu nad programem nebo celým systémem. Rust má za cíl tento trend změnit a umožit psát stejně rychlý kód bez nedefinovaného chování. Projekt C2Rust pak umožňuje využít některých výhod Rustu v projektech, které původně vznikly v C.

Jiným příkladem staršího jazyka ve kterém je napsané velké množství kódu je QuakeC. Když v roce 1996 vyšla hra Quake, byla revoluční v několika ohledech. Na první pohled zaujala především svou grafikou a skutečným trojrozměrným prostředím, ale většinu her své doby (a dle mého názoru i dnešní)

překonala také svou modifikovatelností. Bylo možné nejen měnit herní modely a vytvářet nové mapy, ale především v sobě Quake obsahoval skriptovací jazyk QuakeC, který umožnil téměř kompletně měnit herní logiku. Navím byl tento kód spouštěn bezpečně uvnitř Quaku bez přístupu ke zbytku systému, takže se hráči nemuseli bát stahovat modifikace aniž by důvěřovali jejich autorům.

Poté, co byl zdrojový kód Quake uvolněn pod open source licensí vzniklo mnoho projektů na něm založených, některé aktivní až do dnešní doby. Jedním z nich je Xonotic, na kterém se občas ve volném čase podílím. Bez QuakeC by Xonotic nikdy nevznikl, ale poslední dobou brzdí další rozvoj – není navržený na tvorbu velkých projektů, neobsahuje některé konstrukce, které jsou nutné pro psaní efektivního kódu, a v neposlední řadě odrazuje ostatní programátory, kteří chtějí na Xonoticu pracovat.

Moje bakalářská práce spočívá v nalezení způsobu, jak přeložit kód Xonoticu psaný v jazyce QuakeC do Rustu, což by zjednodušilo další vývoj a pomohlo získat nové programátory. Zároveň by se některé mnou navržené postupy mohly uplatnit v projektu C2Rust.

V práci nejprve rozebírám jazyk QuakeC a preprocesor jazyka C, který využívají projekty napsané v QuakeC. Poté popisuji jazyk Rust. U některých konstrukcí porovnávám podobnosti a rozdíly mezi QuakeC a Rustem. V další kapitole uvádím existující projekty s podobným cílem jako má práce, ze kterých vycházím. V předposlední kapitole popisuji průběh návrhu. V poslední kapitole rozebírám prototyp překladače, který jsem během psaní práce implementoval. Závěr shrnuje míru splnění cílů.

Kapitola 1

Cíl práce

Cílem práce je navrhnout a implementovat překladač z jazyka QuakeC do jazyka Rust s podporou preprocesorových maker. Bude se zaměřovat na čitelnost kódu, například zachováním datových typů, rozdělením kódu do modulů a omezením globálních proměnných. Zároveň bude zachovávat části kódu, které neovlivňují jeho funkci, ale zjednodušují porozumění programátorem, jako jsou komentáře a prázdné řádky oddělující bloky kódu.

Práce bude vycházet z podobných projektů, jako je C2Rust, který převádí C do Rustu a gramatiky QuakeC dostupné v projektu Jcompile. Testování bude probíhat na projektech v jazyce QuakeC, jako je Xonotic.

Kapitola 2

QuakeC

Programovací jazyk QuakeC nemá oficiání specifikaci. Tato kapitola tedy vychází z neoficiálních zdrojů vytvořených komunitou kolem hry Quake a jejích modifikací:

- QuakeC Manual [1] a jeho rozšířená verze QuakeC Reference Manual [2]: Oba dokumenty popisují verzi QuakeC, která byla používaná v původní hře Quake, a nezmiňují konstrukce podporované novějšími kompilátory.
- Introduction to QuakeC [3]: Toto je wiki stránka na GitLabu Xonoticu, která navazuje na původní verzi [4] na stránkách komunity hry Nexuiz[1]. Tato stránka popisuje variantu jazyka QuakeC, kterou používá Xonotic, ale nepopisuje ji celou. Zároveň varuje před některými chybami v kompilátoru, které už jsou opravené.
- Kompilátor GMQCC [5]: V některých případech se mi nepodařilo nikde najít správný a kompletní popis očekávaného chování kódu, skutečný výsledek jsem tedy zjistil experimentálně podle chování kompilátoru GMQCC, který používá Xonotic[2].

QuakeC je staticky typovaný imperativní jazyk inspirovaný jazykem C. Byl navržel Johnem Carmackem při vývoji hry Quake, aby umožnil její skriptování a modifikace bez nutnosti kompilovat kód zvlášť pro různé platformy. QuakeC je kompilováno do bytekódu (*bytecode*), který je spouštěn interpretovaně v herním jádře (*game engine*). Tento interpreter je někdy zván QCVM. Xonotic používá jádro DarkPlaces [6].

Původní QuakeC bylo velmi omezené, např. nebylo možné definovat vlastní datové typy, funkce mohly mít maximálně 8 parametrů, nebylo možné použít

[1] Předchůdce Xonoticu.

[2] Jako příklad za všechny bych rád zmínil syntax definic a deklarací funkcí. Případ, kdy jsou kombinované funkce vyššího řádu a odkazy na prvky entit, není nikde popsaný a vysvětlení, které mi dal jeden z autorů GMQCC na IRC, se ukázalo být mylné, když jsem zjistil skutečné chování s pomocí kompilátoru.

volání funkce jako argument, lokální proměnné vyžadovaly klíčové slovo `local` apod [2]. Postupem času se objevovaly lepší a lepší kompilátory, které některá omezení odstraňovaly.

Dnes se pro hry založené na Quaku používají hlavně kompilátory FTEQCC a GMQCC. Xonotic nějakou dobu používal FTEQCC a později přešel na GMQCC [7]. Od té doby se FTEQCC vyvíjel rychleji a dnes podporuje více konstrukcí než GMQCC – např. struktury (*struct*), třídy (*class*) a ukazatele (*pointer*) [8]. Zbytek této práce se zabývá variantou GMQCC. Samotný kompilátor GMQCC ve výchozím nastavení z důvodu kompatibility se starším kódem používá jinou variantu a tuto je nutné si explicitně vyžádat pomocí přepínače `-std=gmqcc`.

2.1 Struktura kódu Xonoticu

Xonotic je rozdělen do několika repozitářů [9]. V QuakeC je napsaná většina herní logiky. Lze ji nalézt v repozitáři `xonotic-data.pk3dir` [7] spolu s částí herních dat. Je rozdělená do tří hlavních částí – `menu`, `client` a `server`. Sdílený kód je ve složce `common`. Xonotic jako celek lze zkompilovat skriptem `all` v hlavním repozitáři [10].

Samotná kompilace QuakeC má několik kroků. QuakeC používá hlavičkové (*header*) a implementační (*implementation*) soubory, obvykle s příponami `.qh` a `.qc`. Podobně jako v C do sebe implementační soubory pomocí preprocesorové direktivy `#include` kopírují kód hlavičkových. V Xonoticu navíc ale existují skriptem generované soubory, které do sebe kopírují kód z implementačních souborů. Nepoužívá se tedy oddělená kompilace, jako je typické v programech v jazyce C. Na místo ní je před samotnou kompilací spouštěn preprocesor, který podle direktiv `#include` nakopíruje kód do tří souborů – `menu.txt`, `client.txt` a `server.txt`. V těch jsou skriptem změněny některé řádky a výsledek se zapíše do odpovídajících souborů s příponou `.qc`. Teprve s těmi pracuje kompilátor GMQCC.

2.2 Datové typy

QuakeC má tyto datové typy [3]:

- `float`: 32bitové číslo s plovoucí desetinnou čárkou (*floating point*)
- `vector`: trojice `float`ů užitečná zejména při práci se 3D souřadnicemi
- `string`: odkaz na neměnný (*immutable*) řetězec ukončený znakem bytem s hodnotou 0 (*null-terminated string*)
- `entity`: odkaz na objekt, který má jako prvky (*fields*) další proměnné
- odkaz na prvek entity (*field reference*)

- funkce
- pole (*array*)

V kódu se ještě vyskytuje klíčové slovo `void`. Používá se na místě návratového typu u funkcí, které nevrací žádnou hodnotu. V kódu Quaku a Xonoticu jsou deklarované dvě proměnné tohoto typu, které nemají žádný význam za běhu, jsou to pouze značky pro kompilátor.

2.2.1 Float

Pro aritmetické a logické operace se `float` chová stejně jako v jazyce C. Při bitových operacích dojde k převodu na `int`, pak se provede bitová operace a její výsledek se převede zpět na `float` [6]. Pro výpis se používají formátovací řetězce `%d` a `%f` [7].

Float může obsahovat speciální hodnoty jako `NaN` (*not a number*) nebo kladné a záporné nekonečno. GMQCC obsahje chybu, která způsobuje, že `NaN` se v Xonoticu někdy rovná sám sobě (reprodukovatelné například pomocí `float x = 0/0; LOG_INFOF("%f %d\n", x, x == x);`). Toto chování se mi ale nepodařilo reprodukovat na izolovaných testovacích příkladech ani za použití stejných parametrů jako používá Xonotic.

2.2.2 Vektor

S vektory lze pracovat podobně jako se strukturami známými z jazyka C. K jednotlivým prvkům lze přistupovat pomocí tečky, např `my_vec.x`. Ve skutečnosti je to možné pouze s novějšími kompilátory jako GMQCC, v původním QuakeC se používal tvar `my_vec_x`. Jednotlivé prvky vektoru se tedy chovaly jako by existovaly tři samostatné proměnné s názvem, který začíná stejně jako název vektoru a má příponu podle osy X, Y nebo Z. Změnou těchto proměnných se měnil obsah vektoru.

Kód Xonoticu používá obě možnosti. Novější kód k prvkům přistupuje pomocí tečky, starší pomocí podtržítka.

Protože vektory jsou velmi často používaný datový typ, mají vlastní literál ve formě tří čísel oddělených mezerami a ohraničených jednoduchými uvozovkami. Například `'1.5 0 0'` je vektor, kde prvek `x` má hodnotu 1.5 a zbývající dva mají hodnotu 0. Pro výpis vektorů lze použít ve formátovacích funkcích řetězec `%v`.

2.2.3 Entita

Entita je jediný uživatelsky definovaný typ v QuakeC, tím se podobají strukturám známým z jazyka C. Liší se tím, že entita je jedinný datový typ – všechny entity v celém projektu mají stejné prvky. Jedná se tedy o strukturu,

která má velké množství prvků, v praxi v Xonoticu většina z nich na konkrétní instanci není využitá.

Entita není definována na jednom konkrétním místě. Prvky, které má mít lze definovat kdekoliv v projektu mimo funkce, tedy v globálním prostoru (*global scope*), pomocí tečky, názvu datového typu, názvu prvku a středníku. Např. `.float score;` definuje prvek typu `float` s názven `score`. Každá entita ve hře potom má prvek, kam lze uložit skóre, ať už se jedná o hráče, krabici s municí nebo tlačítko v menu. Prvky mohou být i typu entita, tedy entity mohou odkazovat na sebe navzájem.

K prvkům entit se přistupuje stejně jako k prvkům struktur v C, tedy pomocí názvu entity, tečky a názvu prvku, např. `player.score` přistupuje ke skóre entity, která je uložená v proměnné `player`.

Každá entita má index podle toho, kde je uložená v QCVM. QuakeC nemá hodnotu `null` známou z mnoha jiných programovacích jazyků. Místo ní používá hodnotu zvanou `world` a v Xonoticu proměnnou se stejným názvem, která tuto hodnotu obsahuje. Hodnota `world` má index 0 a používá se na podobných místech, jako jiné jazyky používají `null`. V podmínkách se `world` vyhodnotí na nepravdivou hodnotu. Také se používá jako speciální návratová hodnota určitých funkcí, např. funkce pro vyhledávání entit vrací `world`, pokud nenajdou žádnou odpovídající entitu. Z entity `world` je možné číst, některé její prvky popisují stav herního světa. Do prvků je možné zapisovat pouze z funkce `spawnfunc_worldspawn`. Chování při zápisu z jiných funkcí se liší mezi jednotlivými QCVM. Ve virtuálním stroji, který je součástí projektu GMQCC a používá se pro jeho testování, dojde k ukončení programu. Ve virtuálním stroji, který používá Xonotic, zápis uspěje, ale zobrazí se varování [9].

V kódu Xonoticu se vyskytuje identifikátor `NULL`, jedná se o alias na `world` vytvořený pomocí preprocesoru.

2.2.4 Odkazy na prvky entit

Odkazy na prvky fungují podobně jako klíče do slovníku (*dictionary*), pokud entita je chápána jako slovník, nebo offsety do struktury, pokud je entita chápána jako struktura. Nejsou to ukazatele – není možné odkazovat na proměnné ani na části vektoru, pouze prvky entit.

Deklarují se pomocí klíčového slova[3] `var`, tečky, typu odkazovaného prvku, názvu odkazu a středníku, např. `var .float resource;`. Pokud je tato deklarace uvnitř funkce, lze vynechat `var`. Protože není možné deklarovat prvky entit uvnitř funkcí, musí se jednat o deklaraci proměnné a nedochází k nejednoznačnosti. Pro deklaraci parametru funkce se naopak `var` nesmí použít.

[3] V některých popisech QuakeC je var zmiňováno jako klíčové slovo. To však není úplně přesné, GMQCC ho akceptuje například jako název parametru funkce. Takovýto parametr je pak možné na některých místech normálně používat, na jiných dojde k chybě při kompilaci. Podobně se chovají i jiná „klíčová“ slova.

```
.float health;
.float energy;

void regenerate_resource(entity player, .float res, float dt) {
    player.(res) += 0.5 * dt;
}

void main() {
    entity player = spawn();
    regenerate_resource(player, health, TICK_LENGTH);
    regenerate_resource(player, energy, TICK_LENGTH);
}
```

Výpis kódu 1: Použití odkazů na prvky

```
// běžná deklarace
vector(vector v) normalize;

// deklarace zabudované funkce
vector(vector v) normalize = #9;

// definice
vector(vector v) normalize {
    return v / vlen(v);
}
```

Výpis kódu 2: Deklarace a definice

Výpis 1 demonstruje, jak odkazy lze využít. Použití závorek kolem názvu odkazu je pouze konvence používaná v Xonoticu, fungovalo by i `player.res`.

2.2.5 Funkce

Podobně jako v C mohou funkce být deklarovány nebo definovány. Deklarace určují pouze parametry a návratový typ, definice parametry, návratový typ a tělo funkce. Funkce musí být ve zdrojovém kódu deklarována před použitím. Funkce nemusí být definovány pouze v QuakeC ale i v herním jádře. Ty se nazývají zabudované funkce (*builtin functions* / *builtins*) a jsou pak v QuakeC pouze deklarovány pomocí svého indexu.

Existují tedy tři varianty (ukázky ve výpisu 2):

- U deklarací běžných funkcí za signaturou (*signature*) následuje středník.
- U deklarací zabudovaných funkcí se před středníkem vyskytuje ještě rovná se, mřížka a index funkce v herním jádře.

- U definic po signatuře následuje volitelně rovná se a poté tělo funkce ve složených závorkách. Znak rovná se se běžně nepoužívá a v kódu Xonoticu se vyskytuje jen vzácně.

Syntax signatury samotné v ukázkách se mírně liší od C – parametry jsou před názvem funkce. QuakeC ve skutečnosti podporuje obě varianty (alespoň u funkcí pracujících se základními typy) — parametry v závorkách mohou následovat ihned po návratovém typu nebo až po názvu funkce. První varianta je častější ve starším kódu, protože jen ta byla podporována některými staršími kompilátory [2]. Druhá varianta je běžnější v novějším kódu.

Situace se komplikuje pokud funkce má jako parametr nebo návratový typ funkci nebo odkaz na prvek. Zvláště pokud deklarace začíná tečkou není na první pohled zřejmé, jestli se skutečně jedná o signaturu funkce nebo deklaraci odkazu na prvek typu funkce. Nenašel jsem zdroj, který by tyto situace vysvětloval dostatečně podrobně, zbytek této podsekce tedy vychází z pokusů s kompilátorem GMQCC [5].

Význam deklarace zavisí na tom, jestli je v lokálním nebo globálním prosturu. V globálním prostoru se může jednat o deklaraci proměnné, prvku entity a nebo funkce. V lokálním prostoru nelze definovat ani deklarovat funkce ani deklarovat prvky entity, může se tedy jednat pouze o deklaraci lokální proměnné. Zároveň před deklarace v globálním prosturu je možné přidat modifikátor `var`, které způsobí, že se deklarace bude vyhodnocovat jako v lokálním prostoru.

Tabulka 2.1 obsahuje ukázky některých globálních deklarací (nejen funkcí), které jsem zkoušel. Jak je z ní vidět, pokud za názvem deklarované položky následují parametry, vždy se jedná o funkci, i pokud deklarace začíná tečkou. Pouze pokud za názvem následuje středník, rozhoduje se podle začátku. Tečka v takovém případě deklaruje prvek entity, ať už základního typu nebo typu funkce. Pokud deklarace nezačíná tečkou, ale za názvem základního typu následují závorky, jedná se opět o deklaraci funkce. Tento základní typ je její návratová hodnota a obsah závorek parametry.

Je možné vytvořit složitější deklarace, GMQCC však obsahuje chyby. Například `void a()()` je funkce bez parametrů, která vrací funkci bez parametrů a bez návratové hodnoty. GMQCC však dovolí z `a` vrátit i funkci, která vrací například `float`. Domnívám se, že toto chování není úmyslné a v Xonoticu jsem takovéto deklarace nenašel, dále se jimi tedy nezabývám.

Pro zajímavost ještě uvedu, že `float f(float)();` deklaruje funkci, která vrací funkci s parametrem `float` a návratovou hodnotou `float`. Závorky blíže k `f` tedy popisují parametry vrácené funkce a ty vzdálenější popisují parametry `f`. Volání takovéto funkce vypadá například takto: `f()(42);`.

Tabulka 2.1: Význam globálních deklarací

Deklarace	Význam `a`
`.float a;`	Prvek entity typu `float`
`float(float x1) a;` `float a(float x1);`	Funkce s parametrem `float` vracející `float`
`.float a(float x1);`	Funkce s parametrem `float` vracející odkaz na prvek typu `float`
`.float(float x1) a;`	Prvek entity typu funkce, která má parametr `float` a vrací `float`
`.float(float x1) a(float x2);`	Funkce s parametrem `float` vracející odkaz na prvek, který je funkce s parametrem `float` vracející `float`

2.2.6 Pole

Pole jsou dostupná pouze v novějších kompilátorech jako GMQCC. Deklarace a použití jsou podobné jako v jazyce C, avšas jsou pomalejší. Absence ukazatelů v QuakeC není daná pouze jazykem, QCVM a bytekód pro ně nemá instukce[4] [6]. GMQCC je tedy simuluje tím, že vytvoří mnoho proměnných a pro přístup k nim funkce, které se rozhodují, ke které z nich přistoupit binárním vyhledáváním podle indexu za běhu [3].

[4]Existují i implementace QCVM, které je mají, ta používaná Xonoticem k nim ale nepatří.

Kapitola 3

Preprocesor

Tato kapitola shrnuje fungování preprocesoru používaného v jazycích C, C++ a QuakeC. Vychází z dokumentace GCC [11], která ho popisuje. Xonotic používá jako preprocesor GCC [7], ale lze použít i Clang.

Preprocesor je nástroj, který slouží k předzpracování (*preprocessing*) zdrojového kódu v jazycích jako C, C++ a QuakeC, které probíhá před samotnou kompilací. Může být implementován jako samostatný program nebo součást kompilátoru.

Samotnému předzpracování předchází několik kroků:

1. Rozdělení kódu na řádky.

2. Nahrazení trigrafů. To jsou trojice symbolů, které mají význam jiného znaku. Xonotic je nepoužívá.

3. Spojení pokračovaných řádků (*continued lines*). Pokud řádek končí zpětným lomítkem, je spojen s následujícím. Toho Xonotic často využívá v definicích maker. Ty by normálně musely být na jednom řádku, avšak pomocí pokračovaných řádků je možné zvýšit čitelnost a rozdělit ho vizuálně v souboru na několik řádků. Po tomto kroku je preprocesor zpracovává jako jeden.

4. Odstranění komentářů. Ty jsou nahrazeny mezerami, aby nedošlo omylem ke spojení tokenů kolem komentáře. Tento krok lze v preprocesorech GCC [12] a Clang [13] potlačit, například pomocí přepínače `-CC`.

Po těchto krocích následuje tokenizace, tedy rozdělení textu na tokeny jako jsou identifikátory, operátory a podobně. Tento proud tokenů lze přímo předat parseru, pokud ale obsahuje preprocesorové direktivy nebo makra, je nutné je nejdříve zpracovat. Direktivy musejí být na začátku řádku a nejdůležitejší z nich jsou jednoho z následujících typů:

Inkluze Pomocí direktivy `#include` je možné do zdrojového kódu nakopírovat obsah jiného souboru. To je užitečné zejména při oddělené kompilaci, kdy hlavičkové soubory obsahují deklarace funkcí. Ty je pak možné inkludovat v mnoha souborech, pokud tyto funkce chceme použít.

Definice maker Direktiva `#define` vytváří definice maker. Ta mohou být buď objektová (*object-like*), tedy bez parametrů, nebo funkční (*function-like*), tedy s parametry v závorkách. Funkční makro může mít nula a více parametrů.

Podmíněná kompilace Pomocí podmíněné kompilace umožňuje preprocesor ze zdrojového kódu některé řádky odstranit před samotnou kompilací. Lze tak vytvořit několik verzí programu, které sdílejí část kódu a v části se liší.

Kontrola řádků (*line control*) Pokud před kompilací dochází k reorganizaci kódu do jiných souborů, lze tímto mechanismem informovat kompilátor o původní poloze řádků a tím získal přesnější zprávy o chybách.

Diagnostika Tyto direktivy umožňují při kompilaci generovat vlastní varování a chyby.

Po zpracování direktiv a expanzi maker je proud tokenů předán parseru a nebo může být vypsán do souboru, například pokud je v GCC použit parametr `-E` [12].

3.1 Makra

Některá makra jsou předdefinovaná preprocesorem, jiná pomocí direktivy `#define`. Za `#define` následuje název makra a volitelně názvy parametrů v závorkách. Zbytek řádku jsou tokeny, na které se makro expanduje. Expanzí se rozumí nahrazení makra těmito tokeny. Pokud obsahují identifikátor, který se shoduje z názvem jednoho z parametrů, je při expanzi nahrazen tokeny, které byly předámy na místě tohoto parametru. Definici lze zrušit direktivou `#undef`, takové makro se na následujících řádcích již neexpanduje. Potom lze definovat jiné makro se stejným názvem.

Kromě běžných expanzí umožňují makra ještě převod na konstantní řetězce, tedy stringizaci (*stringizing*) někdy také zvanou stringifikaci (*stringification*), a spojování tokenů, tedy konkatenaci (*concatenation*). Stringifikace se provádí vložením mřížky (`#`) před použití příslušného parametru. Konkatenace se provádí vložením dvou mřížek (`##`) mezi dva tokeny. Je možné konkatenaci řetězit.

Kapitola 4

Rust

V této kapitole shrnuji základní rysy jazyka Rust, zvláště ty, které se dotýkají mé práce. Mnohem detailnější popis lze najít v

- oficiální knize o Rustu, The Rust Programming Language [14] (také známé jako TRPL nebo The Book),
- v oficiálních ukázkách kódu s vysvětleními, Rust by Example [15]
- nebo v neoficiálním úvodu do Rustu, A Gentle Introduction To Rust [16].

Rust je silně a staticky [16] typovaný programovací jazyk s inferencí [15, 16] založenou na typovém systému Hindley-Milner [17]. Kombinuje prvky z imperativních, funkcionálních a objektově orientovaných paradigmat [14]. Zaměřuje se na správnost kódu, tvorbu abstrakcí, které nesnižují efektivitu, a paralelismus [14].

Nejprve byl vyvíjen Graydonem Hoarem jako osobní projekt, později za podpory organizace Mozilla Foundation [18]. Dnes je vyvíjen komunitou, které kolem něho vznikla [19]. Verze 1.0 byla vydaná v roce 2015 [19].

Syntax Rustu je podobný jazykům C a C++. Má podobné podmnínky a smyčky, avšak kulaté závorky jsou volitelné, složené naopak povinné. Nepodporuje `goto`. Místo příkazu `switch` má Rust `match`, který nepodporuje propadávání mezi větvemi, ale umožňuje používat složitější podmínky a *pattern matching*.

Mnohé příkazy v Rustu zároveň fungují jako výrazy. Rust nemá ternární operátor, na jeho místě lze použít běžný `if`. Také nekonečná smyčka `loop` (ze které je možné vrátit hodnotu pomocí příkazu `break`), `match` a běžné bloky `{}` jsou výrazy. Pomocí bloku je tedy možné nahradit operátor čárka z C a QuakeC nebo i *pre-increment* a *post-increment*, které Rust nemá.

4.1 Datové typy

Mezi primitivní datové typy patří celá čísla (*integer*) se znaménkem (*signed*) i bez (*unsigned*) a desetinná s plovoucí desetinnou čárkou (*foating point*). Název typu vždy obsahuje počet bitů, tedy např. `i32` je 32bitové celé číslo se znaménkem, `f64` je 64bitové desetinné číslo, `u8` je 8bitové celé číslo bez znaménka. Pro znaky se používá typ `char`[5], který má 4 byty pro podporu Unicode [20]. Řetězce jsou vždy v kódování UTF-8, tedy neobsahují pole těchto znaků, ale poskytují iterátory pro přístup k nim.

Nedílnou součástí typového systému Rustu jsou generické typy a obdoba rozhraní (*interfaces*) zvaná *traits*[6]. Traity se ve skutečnosti od interfaců liší v několika klíčových bodech (například je možné je implementovat pro již existující datové typy) a připomínají typové třídy (*typeclasses*) známé z Haskellu, kterými jsou inspirované [21].

Rust podporuje dva hlavní druhy uživatelsky definovaných typů, struktury (*structures* / *structs*) a enumerace (*enumerations* / *enums*). Struktury jsou podobné jako v C++, obsahují několik prvků (*fields*) s daty. Enumerace definují typ, který může nabývat některé z vyjmenovaných hodnot. Tyto hodnoty se nazývají varianty (*variants*). Enumarace může mít libovolný počet variant včetně žádné. Narozdíl od C a většiny z něho odvozených jazyků, v Rustu každá varianta navíc může obsahovat data. Různé varianty mohou mít data různých typů. Struktury i enumerace mohou mít metody.

4.1.1 Reference a ukazatele

Podobně jako C++ obsahuje Rust reference (*references*) a ukazatele (*pointers*). Reference v Rustu vždy odkazují na validní objekt (tedy neobsahují `NULL` ani adresu již uvolněného objektu) a podléhají následujícím pravidlům. Na konkrétní objekt může existovat buď:

- libovolné množství sdílených neboli neměnných (*shared* / *immutable*) referencí,
- nebo jedinná exkluzivní neboli měnitelná (*exclusive* / *mutable*) reference.

To zaručuje, že pokud jedna část kódu má neměnnou referenci, jiná část kódu nemůže po to dobu odkazovaný objekt měnit, což je užitečné například při práci s vlákny. Vytvoření reference na objekt se nazývá vypůjčení (*borrow*) a část kompilátoru, která dodržení těchto pravidel kontroluje se nazývá *borrow checker*. Ten zároveň kontroluje životnost (*lifetime*) všech objektů a referencí na ně a neumožní dealokovat objekt, dokud na něj existují reference.

[5] Jak vysvětluje dokumentace, to, co si člověk představí pod pojmem znak, se nemusí nutně shodovat s tím, co tento datový typ skutečně obsahuje. Budu však dále používat pojem znak jako český překlad slova char.

[6] Nepodařilo se mi vymyslet nebo najít vhodný překlad. Nejblíže mi připadají slova rys, příznak nebo vlastnost.

Tyto podmínky jsou často příliš omezující, Rust tedy podporuje i ukazatele. Ty fungují podobně jako v C a C++, mohou tedy obsahovat `NULL` a nemají žádná speciální omezení, na co mohou ukazovat. dereferencování takových ukazatelů však umožňuje vyvolat nedefinované chování. V Rustu je proto lze dereferencovat pouze uvnitř `unsafe` bloků (více v sekci 4.2).

Kompromisem mezi referencemi a ukazateli jsou chytré ukazatele (*smart pointers*). Ty nemají omezení jako reference, ale nevyžadují `unsafe` jako běžné ukazatele. To je možné díky tomu, že při jejich použití neprovádí některé kontroly staticky kompilátor, ale budou provedeny dynamicky až za běhu program.

4.2 Unsafe

Jedním z cílů Rustu je zamezit chybám typickým pro nízkoúrovňové jazyky aniž by programátorovi odebral kompletní kontrolu nad pamětí. Pro netriviální programy by bylo velmi těžké dokázat jejich správnost a zároveň povolit libovolné operace s pamětí. Rust proto využívá statickou analýzu (vyžadující dodržení určitých omezujících pravidel), zároveň ale umožňuje provést i operace umožňující porušit záruky, které Rust běžně poskytuje, ty však dovoluje provést pouze uvnitř bloků označených pomocí klíčového slova `unsafe`. Tím se snaží omezit množství kódu, které může vyvolat nedefinované chování a které je tím pádem nutné pečlivě kontrolovat. V kódu, který není označený `unsafe` k němu nemůže dojít, pokud `unsafe` kód neobsahuje chyby.

Jedním z principů programování v Rustu je tvorba bezpečných abstrakcí. Často není možné napsat efektivní kód bez využití `unsafe`. Cílem Rustu není se `unsafe` kódu kompletně vyhnout, ale umožnit ho zapouzdřit do bezpečných rozhraní. Pokud není možné toto pravidlo pro nějakou funkci dodržet, je nutné ji také označit `unsafe`, její volání potom bude vyžadovat `unsafe` blok.

Chytré ukazatele a kolekce (*collections*) ve standardní knihovně Rustu typicky ve své implementaci obsahují `unsafe`, ale navenek poskytují rozhraní které pro běžné užití `unsafe` nevyžaduje.

4.3 Makra

Rust podporuje metaprogramování, tedy psaní kódu, který píše další kód, v podobě maker. Poskytují nejen způsob, jak se vyhnout psaní podobných bloků kódu, ale i například jak vytvářet doménově specifické jazyky (*domain specific language / DSL*) nebo jak přidávat objektům metody na základě atributů.

Protože některá použití maker by mohla připomínat volání funkcí, odlišují se tím, že po názvu makra následuje vykřičník. Z toho je zřejmé, že například `println!("Hello, {}!", name);` je makro, ne funkce. Tento příklad zároveň ilustruje jedno z častých použití maker a to simulace funkcí s proměnným

počtem parametrů (*variadické funkce / variadic functions*), které Rust nativně nepodporuje.

Makra se dělí na dva typy — deklarativní a procedurální.

4.3.1 Deklarativní makra

Podobně jako preprocesorová makra pracují deklarativní makra s proudem tokenů. Zde však nedochází k nahrazování libovolných tokenů, ale proud (struktura `TokenStream`) je uspořádán do podstromů (struktura `TokenTree`) [22]. Podstromy mohou být jednotlivé tokeny, další proudy typu `TokenStream` nebo celé podstromy AST — abstraktního syntaktického stromu (*abstract syntax tree*). Pokud například makro přijímá jako vstup podstrom typu výraz, bude i na výstupu ve formě podstromu. Nehrozí tedy například, že by po expanzi mohl vzniknout výraz s jinými přednostni v závislosti na tom, co je kolem expandovaného makra, jako je tomu u preprocesoru [11], a není nutné parametry a celé makro obalovat závorkami. Na druhou stranu tím vznikají určitá omezení — není možné například vytvářet makra, která nemají správně vnořené závorky.

Druhý zásadní rozdíl oproti preprocesorovým makrům je, že deklarativní makra jsou tzv. hygienická (*hygienic*) [23]. To znamená, že identifikátory uvnitř makra nemohou omylem kolidovat s identifikátory vně a naopak. Například proměnné deklarované uvnitř makra nejsou vidět mimo toto makro a opačně. Výjimkou jsou položky (*items*), jako například funkce.

První část definice makra obsahuje popis, jakou strukturu má mít vstupní proud. Jeho jednotlivé části je možné přiřadit do tzv. metaproměnných. Druhá část je tělo makra – kód na který se má expandovat. Tělo může obsahuvat nové tokeny a podstromy, odkazovat se na metaproměnné nebo volat další makra.

Definice makra může být rozdělena na více řádků. Zároveň jedno deklarativní makro může obsahovat více větví, z nichž každá může přijímat podstrom s jinou strukturou a může se expandovat na jiný kód.

Protože struktura první části každé větve připomíná strukturu přijímaného kódu, deklarativní makra jsou někdy také nazývána makra příkladem (*macros by example*).

4.3.2 Procedurální makra

Procedurální makra jsou funkce v jazyce Rust, které berou kód ve formě proudu tokenů jako vstup (popřípadě dva proudy, pokud se jedná o makro, které definuje atribut), provádějí na něm libovolné operace a produkují kód opět ve formě proudu tokenů jako výstup. K převodu QuakeC je nepoužívám, zbytek této práce se tedy zabývá deklarativními.

4.4 Podmíněná kompilace

Při programování je často nutné vytvořit několik verzí programu nebo knihovny, které jsou skoro stejné, ale na vybraných místech se liší, například obsahují různé implementace jedné funkce podle platformy. Takovým verzím se někdy říká konfigurace.

Většina součástí programu je přítomná ve všech konfiguracích, pokud si programátor přeje, aby byl kus kódu přítomný jen v dané konfiguraci, může k tomu použít atribut `cfg` nebo makro `cfg!`.

Forma s atributem se hodí na vynechání celých modulů, funkcí, struktur, bloků a podobně, tedy položek, které mohou mít atributy. Formu s makrem je možné použít tam, kde je potřeba výraz, například v podmínkách.

Podmíněná kompilace v Rustu tedy nepracuje s řádky, jako je tomu v C, ale s podstromy AST, podobně jako je tomu u maker.

Kapitola 5

Současný stav řešení problému

Aktuálně neexistuje kompilátor schopný přeložit QuakeC do Rustu. Existují tři projekty podobné mému, ze kterých vycházím – Jcompile [24], Corrode [25] a C2Rust [26].

5.1 Jcompile

Tato sekce vychází ze zdrojového kódu Jcompile na GitHubu [24] a z osobní komunikace s autorem na IRC.

Projekt Jcompile je nedokončený pokus překládat QuakeC do C++ nebo jazyka Lua. Je napsaný Andrew „TimePath“ Hardakerem[7] – jedním z programátorů, kteří dříve pracovali na Xonoticu. Projekt je napsaný ve starší verzi jazyka Kotlin a využívá verze knihoven, které již nejsou dostupné. Po menších úpravách se mi podařilo ho zkompilovat. Pro syntaktickou analýzu využívá generátor parserů Antlr4 a gramatiku pro QuakeC založenou na gramatice pro jazyk C pro porovnání dostupné v projektu Antlr [27]. Parser není dokončený a z toho důvodu frontend Jcompile není schopný zpracovat aktuální verzi kódu Xonoticu.

Jcompile není navržený na generování čitelného kódu. Pracuje s výstupem preprocesoru, ztrácí tedy informace a makrech a podmíněném překladu. Dále nezachovává komentáře ani bílé znaky. Vzhledem k tomu, že mé cíle jsou čitelnost a zachování maker, autor mi doporučil nepokračovat v Jcompile, ale založit nový a samostatný projekt a Jcompile používat pouze jako referenci.

5.2 Corrode

Následující sekce vychází z repozitáře projektu Corrode na GitHubu [25] a porovnání C2Rust a Corrode [28] na blogu autora Corrode, Jameyho Sharpa.

[7] Autor chce být citován včetně přezdívky, pod kterou je známý na IRC a GitHubu.

Corrode je první projekt s cílem překládat C do Rustu. Zvládá přeložit velkou část syntaxe jazyka C, ale pracuje na výstupu preprocesoru [29], takže nepřekládá makra. Není již dále vyvíjen. Autor dospěl k závěru, že další vývoj je brzděn rozhodnutími, která udělal v počáteční fázi vývoje, a začal zvažovat Corrode od základu přepsat. V té době se zároveň objevil projekt C2Rust, který se poučil z chyb Corrode a podle Sharpa je vhodnou náhradou za Corrode.

5.3 C2Rust

Tato sekce je založena na informacích dostupných na oficiálních stránkách projektu C2Rust [26], na GitHubu C2Rust [30], na blogu firmy Galois [31] a na manuálu C2Rust [32].

Project C2Rust je vyvíjen společnostmi Galois a Immunant. Návrh je inspirovaný projektem Corrode [28] a pro překlad některých konstrukcí jazyka C jako `switch` a `goto` převzal algoritmus *Relooper* z projektu Emscripten.

Jeden z cílů C2Rust je zachovávat komentáře, podpora pro ně je experimentální [33]. Když jsem ji vyzkoušel (online na stránkách C2Rust i nejnovější verzi z GitHubu offline), v generovaném kódu byly zachovány pouze komentáře nad deklaracemi funkcí a v tělech některých funkcí před prvním příkazem. C2Rust nezachoval komentáře za poslední funkcí v souboru ani uvnitř funkcí mezi příkazy. Rozvěž nezachovává prázdné řádky, to jsem ověřil jednak experimentálně a zároveň jsem je nanešel nikde zmíněné v dokumentaci ani kódu.

Většina C2Rust, včetně algoritmu *Relooper*, je napsaná v Rustu. Pro parsování jazyka C používá projekt Clang. Tento kód je napsaný v jazyce C++. Data z Clangu, jako je AST jazyka C, předává do zbytku překladače serializovaná ve formátu CBOR.

V době, kdy jsem na své práci začal pracovat, C2Rust nepodporoval překlad maker. V průběhu mé práce autoři začali hledat způsoby pro překlad jednoduchých maker, jako jsou konstanty [34]. Negenerují místo nich deklarativní makra, ale konstanty v jazyce Rust.

Kód, který C2Rust generuje obsahuje značné množství `unsafe`, ať už se jedná o bloky nebo celé funkce. C2Rust k jejich odstranění obsahuje refaktorovací nástroj, který ale nefunguje plně automaticky – vyžaduje, aby uživatel označil uzly v AST, na kterých mají být změny provedeny.

Kapitola 6

Návrh překladače

V této kapitole popisuji a zdůvodňuji rozhodnutí, která jsem v průběhu návrhu udělal[8].

Důležitou součástí mé práce je podpora maker, tedy jejich převod do ekvivalentu v jazyce Rust. Jak vyplývá z porovnání preprocesorových a deklarativních maker v 4.3.1, ne všechna preprocesorová makra je možná přeložit. Často se jedná o případy, kdy v AST makro nevytvoří kompletní podstrom nebo naopak, kdy vytvoří části jednoho či více podstromů mezi nimiž jsou uzly, které z makra nepochází.

Příkladem první možnosti je dvojice maker `#define MACRO_BEGIN if(1){` a `#define MACRO_END }else voidfunc()`, která jsou v Xonoticu často využívána na začátku a konci maker, která se expandují na více příkazů [7]. Tím se předejde chybám, které by vznikly, kdyby takové makro bylo použito například v podmínce, která nemá složené závorky. Překladač, který by takováto makra byl schopný nahradit deklarativními makry v Rustu (nebo je odstranit úplně) by musel být schopný rozpoznat, že patří k sobě a dohromady s příkazy mezi nimi tvoří kompletní podstrom.

Příkladem druhého případu je použití makra `#define ADD(X, Y) X + Y` ve výrazu jako `ADD(2, 3) * 4`. Protože násobení má přednost před sčítáním, po expanzi díky přednostem z makra `ADD` nevznikne ucelený podstrom. V tomto případě by se pravděpodobně jednalo o chybu, tedy chování, které autor kódu nezamýšlel. Není však vyloučeno, že v Xonoticu (nebo jiných projektech, které by můj překladač měl zpracovat) existují makra, kde je toto chování žádoucí.

Při běžném překladu QuakeC expanze maker probíhá před samotnou kompilací. Při tom se ztrácí informace nutné k jejich obnovení. Mým prvním cílem tedy bylo tyto informace získat a zachovat. Zvažoval jsem dva možné přístupy – parsování maker a zapamatování pozic expanzí:

[8] A z nichž nemalá část se postupem času ukázala být špatná.

Parsování maker První možností bylo vytvořit parser, který zároveň zpracovává preprocesorové direktivy a jazyk QuakeC v nich obsažený. Pro makra, která sama o sobě tvoří validní syntax QuakeC by se musel rozhndnout, jaký podstrom tvoří, tedy například jestli se jedná o celou funkci, několik příkazů nebo výraz, a musel by se vypořádat se změnou předností po expanzi. Makra, která netvoří validní syntax by musel buď přeskočit nebo, pokud v sobě obsahují volání dalších maker, ta expandovat. Tento přístup se mi zdál příliš složitý a pravděpodobně by vedl k reimplementaci velké části preprocesoru, dál jsem ho tedy nerozvíjel.

Zapamatování pozic expanzí Druhá možnost je nechat makra expandovat jako při normální kompilaci, ale u každého tokenu si zapamatovat, ze ktetého makra pochází a, protože makra mohou být vnořená, kterými prošel. Při překládání do Rustu je možné v AST (ať už jazyka QuakeC nebo Rustu) najít podstromy, které přesně odpovídají expanzi jednoho makra nebo jednoho parametru makra a ty nahradit příslušným makrem nebo parametrem. Tento přístup jsem se rozhodl implementovat.

První návrh překladače, který jsem nazval *qc2rust*, měl fungovat v několika krocích:

1. zpracování preprocesorových direktiv a maker se zachováním informací o pozicích tokenů,
2. parsování QuakeC,
3. samotný překlad QuakeC do Rustu,
4. překlad preprocesorových maker do deklarativních,
5. vypisování kódu Rustu.

Protože jsem doufal, že můj postup pro překlad maker by se mohl později uplatnit v C2Rust, snažil jsem se volit podobné technologie. Jako hlavní jazyk pro implementaci jsem zvolil Rust. Doufal jsem také, že bude pro překlad konstrukcí, které nemají v Rustu přesný ekvivalent, jako `switch` s propadáváním a `goto`, možné použít algoritmus *Relooper* ze C2Rust. Zároveň mi to umožnilo použít knihovnu libsyntax [22] pro vypisování Rustu stejně jako C2Rust [30]. Zbývalo vymyslet řešení pro zpracování maker a parsování.

6.1 Preprocesor

Existuje již mnoho implementací preprocesoru jazyka C, přišlo mi tedy rozumné použít některý nich a nepsat vlastní. Zvážil jsem několik možností

a hodnotil jestli poskytují informace o expanzích a direktivách pro podmíněnou kompilaci[9], jestli je možné je volat jako knihovnu z jiného programu a jestli jsou udržované.

Zamítl jsem například Mcpp [35], Jcpp [36], Warp [37] a Wave [38], protože nebyly udržované (u Jcpp jsem později zjistil, že není aktualizovaná pouze webová stránka a údržba pokračuje na GitHubu), nenašel jsem vhodné rozhraní poskytující pozice expanzí maker nebo mi přisly špatně dokumentované.

Dále jsem zvažoval získat tato data z GCC, které obsahuje mnoho přepínačů ovládajících preprocesor [12]. Nejnadějněji vypadal `-fdebug-cpp`, nenašel jsem ale nikde popis formátu výpisu a když jsem zkoušel jak se mění podle změn ve vstupním souboru, nepřipadalo mi, že obsahuje informace, které potřebuji. Například se výpis nezvětšoval pokud jsem přidával nepřímá volání maker ani neobsahoval názvy maker.

Implementace preprocesoru, která určitě poskytuje informace o jednotlivých krocích expanzí je v IDE Eclipse [39]. Nenašel jsem však o ní další informace a než jsem začal procházet její kód, rozhodl jsem se vyzkoušet Clang.

Projekt Clang je překladač pro jazyky C, C++ a Objective-C [40], zároveň poskytuje knihovny LibClang [41] a LibTooling [42], které umožňují psaní nástrojů pracujících se zdrojovým kódem v těchto jazycích.

Rozhraní poskytované knihovnou LibClang je a jazyce C, existují k němu bindingy v dalších jazycích a je stabilní s ohledem na zpětnou kompatibilitu. Oproti tomu LibTooling nabízí rozhraní v jazyce C++, které je méně stabilní, ale poskytuje bohatší funkcionalitu.

Jazyk QuakeC je dostatečně odlišný od C i C++, aby nebylo možné použít existující parsery pro tyto jazyky, mně však stačilo pouze získat informace z lexeru a preprocesoru. Tokeny, které používá QuakeC se mírně liší od těch z C, lexer Clangu je však schopný je zpracovat. Například operátor mocnění `**` vrátí jako dvě samostatné hvězdičky, což není správně, ale je triviální opravit později v parseru. Literály pro vektory vrací jako literály pro `char`, ale jejich obsah zachová, je tedy opět jednoduché je zpracovat později správně.

Protože pro LibClang existují bindingy v jazyce Rust jako clang-sys [43] a clang-rs [44], vyzkoušel jsem tuto knihovnu jako první. Poskytuje informace o pozicích tokenů včetně původního souboru, je tedy možné odlišit kód z hlavního souboru a kód, který pochází z direktiv `#include`. Stejně tak pomocí ní lze najít tokeny, které jsou komentáře, a lze je tedy zachovat při překladu. Neposkytuje informace o jednotlivých krocích expanze, o názvech parametrů maker ani o podmíněné kompilaci.

Rozhraní, které poskytuje LibTooling je mnohem rozsáhlejší oproti LibClang. Hledal jsem zmínky o makrech nebo preprocesoru, zaměřil jsem se na třídy `FrontendAction`, `Preprocessor` a `PPCallbacks`. Jako příklad jejich po-

[9] Značně jsem podcenil množství času, které tato práce zabere, a přesto, že podpora podmíněné kompilace není součástí zadání, myslel jsem, že ji stihnu implementovat a nechtěl jsem navrhnout řešení, které by v tomto směru bylo později těžké rozšířit.

užití jsem našel program pp-trace [45]. Ten, podobně jako velké části rozhraní LibTooling, je navržen pro práci s validním kódem v jazycích, které Clang podporuje, a očekává, že vstup bude možné parsovat a vytvořit z něho AST. Poté, co jsem se naučil, jak s LibTooling pracovat, jsem pp-trace upravil, aby používal pouze preprocesor, ne parser.

Po dalších úpravách jsem z LibTooling získal data o direktivách `#include`, komentářích, názvy parametrů maker a hodnoty argumentů do nich předaných, stringifikaci a částečně informace o přesném průběhu expanzí vnořených maker. Nenašel jsem způsob, jak zjistit přesné rozsahy při vnořených expanzích, tedy případy, kdy se makro expanduje na další makro nebo dostane makro jako argument. Navíc jsem stále hledal způsob jak zachovat všechny tokeny, včetně těch, které jsou smazané podmíněnou kompilací. Z LibToling jsem získal pouze informace o nalezených direktivách podmíněné kompilace a hodnotách podmínek, ne však tokeny, které byly na vynechaných řádcích. Domníval jsem se, že by bylo možné získat všechny tokeny tím, že bych preprocesor spouštěl opakovaně a po každém průchodu bych ze vstupu odstranil podmínky, které byly nepravdivé. Tím bych postupně získal všechny tokeny, všetně hodnot podmínek, na kterých jejich zpracování závisí.

Touto dobou jsem komunikoval s autorem Jcompile, který ve svém projektu dříve také chtěl podporovat makra, zvažoval však naprosto jiný přístup, ale nikdy neměl čas ho vyzkoušet. Navrhoval použít komentáře k ohraničení maker a jejich parametrů, kód zpracovat preprocesorem zachovávajícím komentáře a během parsování porovnat pozice komentářů ve výsledku a z nich odvodit průběh expanzí.

Vyzkoušel jsem ručně ohraničit makra komentáři na jednoduchých příkladech a protože se zdálo, že tato metoda mi poskytně více informací než jsem do té doby získal z LibTooling, rozhodl jsem se jeho nápad vyzkoušet a implementovat (více v kapitole 7).

6.2 Parser

Ze začátku jsem myslel, že komentáře a prázdné řádky by měly být součástí AST, aby bylo možné je v Rustu generovat na co nejpřesnější pozici. Když jsem zvažoval možnost použít existující parser, bral jsem tedy v úvahu nutnost jeho rozšíření. Komentáře ale mohou být v kódu téměř kdekoliv, tím by velmi komplikovaly návrh parseru a AST. Vzhledem k tomu, že napříč parsováním stejně musím zachovat přesné pozice původních tokenů kvůli překladu maker, později jsem se rozhodl použít stejný přístup ke komentářům. Nejsou součástí AST ale jsou uloženy se svými pozicemi v seznamu zvlášť. Při vypisování generovaného kódu stačí procházet tento seznam, porovnávat pozice aktuálního uzlu AST a aktuálního komentáře a vypsat ho ve správný okamžik. Tento přístup používa libsyntax [22].

Ve stejnou dobu, kdy jsem hledal vhodný způsob, jak zachovat informace

z preprocesoru, jsem zároveň hledal způsob, jak QuakeC parsovat. Jediný plně funkční parser byl v samotném kompilátoru GMQCC, ten však nezachovával pozice komentářů a jeho chybové zprávy často obsahovaly špatná čísla řádků nebo sloupců, domníval jsem se tedy, že nezachovává správně ani pozice parsovatelných tokenů. Bylo by tedy nutné ho opravit a rozšířit.

Druhá možnost po GMQCC byla použít parser z jcompile napsaný v Kotlinu nebo alespoň samotnou gramatiku, která používá generátor parserů Antlr4. Ten však nepodporuje generování parserů v jazyce Rust [46], musel bych tedy data předávat do Rustu serializovaná. Samotná gramatika není dokončená, nevěděl jsem, kolik na ní zbývá práce, a autor Jcompile mi doporučoval v jeho projektu nepokračovat, protože zachování informací o komentářích a pozicích tokenů by dalo příliš práce. Navíc uzly v AST Jcompile hodně využívají dědičnost, kterou Rust nepodporuje. V Rustu je typické pro ně využít datový typ `enum`, jak to dělá například libsyntax [22], takže by mi přibyla práce s převodem.

Jak jsem později zjistil, Antlr4 umožňuje komentáře a informace o pozicích tokenů zachovat relativně jednoduše. Potom jsem ale narazil na problém s parsováním deklarací `typedef`, který se mi v Antlr4 nepodařilo vyřešit.

Třetí možnost byla napsat vlastní parser. Protože bych to mohl udělat v Rustu, nebylo by nutné předávat data mezi několika jazyky.[10] GMQCC používá ručně napsaný rekurzivně sestupný parser [5], QuakeC je tedy možné parsovat touto metodou, myslel jsem ale, že ušetřím čas použitím knihovny pro generování či kombinování parserů. Měl jsem dobré zkušenosti s balíčkem nom [47], který však podporoval pouze parsování textu nebo binárních formátů. Protože jsem touto dobou jako lexer a preprocesor chtěl stále použít Clang, potřeboval jsem parser, který přijímá proud tokenů, tedy obecných datových struktur, do kterých nevidí. Z tohoto důvodu jsem vyloučil i jiné známé knihovny. Nakonec jsem se rozhodl použít relativně neznámý baliček plex [48], což je generátor lexerů a parserů. Splňoval i mé další požadavky jako zachování informací o pozicích tokenů a umožňoval jednoduše budovat AST během parsování.

Později se projevily nedostatky rozhraní Plexu. Například, protože QuakeC obsahuje `typedef` podobný jako C, potřeboval jsem předávat názvy typů zpět do lexeru (v tuto dobu už jsem přestal používal Clang a přešel k označování maker komentáři), aby generoval správný token podle toho, jestli se jedná o běžný identifikátor nebo název typu. V Plexu to není možné bez globálních proměnných, což do budoucna ztěžuje paralelizaci parsování. Hlavní nevýhoda plexu je však doba kompilace parseru. Ta se bohužel začala projevovat postupně (ke konci dosáhla dvou až tří minut při každé změně) a když

[10] Vytvoření vlastního parseru jsem také považoval za dobrý způsob, jak se z kompletní gramatikou QuakeC seznámit a nepředpokládal jsem, že mi zabere mnoho času. Toto bylo jedno z výše zmíněných špatnách rozhodnutí, složitost gramatiky se projevila ve chvíli, kdy jsem se dostal k deklaracím funkcí.

jsem začal zvažovat přechod na jinou knihovnu, byl parser skoro hotový, takže jsem zůstal u Plexu.[11]

[11]Historie repozitáře qc2rust často obsahuje dvě paralelní větve. Obvykle jsem měl dvě kopie projektu a pracoval na té, která se zrovna nekompilovala.

Kapitola 7

Implementace a testování

Tato kapitola popisuje výsledný návrh překladače *qc2rust*, některé detaily implementace, algoritmus použitý k překladu maker, omezení vycházející z implementace a postup testování.

Konečný prototyp překladače zpracovává QuakeC v následujících krocích:

1. lexikální analýza,
2. parsování preprocesorových direktiv, označení direktiv a maker, výpis do souboru
3. expanze maker existujícím preprocesorem jako GCC nebo Clang,
4. lexikální analýza předzpracovaného zdrojového kódu,
5. parsování (syntaktická analýza) QuakeC do AST,
6. samotný překlad QuakeC do Rustu, při němž probíhá i překlad maker,
7. vypisování kódu Rustu knihovnou libsyntax.

7.1 Lexer

Lexikální analyzátor využívá knihovnu Plex, je definovaný pomocí regulárních výrazů. Ke každému tokenu přiřadí offset, kde ve vstupu začíná a končí. Této dvojici říkám rozsah (struktura `Span`). Před samotným rozdělením vstupu na tokeny je v něm nutné spojit pokračované řádky. Offsety pro jednoduchost implementace ukazují do textu po spojení řádků.

Používám stejný lexer v krocích 1 a 4, který pouze vrací mírně odlišné tokeny podle parametru. Preprocesor nepotřebuje klíčová slova, komplikovala by rozpoznání maker a jejich parametrů. Parser naopak potřebuje klíčová slova, navíc předává lexeru názvy typů, ten je potom vrací jako token odlišný od běžného identifikátoru pro zjednodušení parseru.

7.2 Preprocesor

Modul `preprocessor` obsahuje jednoduchý ručně napsaný parser, který přijímá proud tokenů z lexeru a hledá v něm direktivy `#define`. Z těch zjistí název makra a parametrů a jestli je variadické. Na začátek a konec makra přidá komentáře, stejně tak označí všechny výskyty parametrů v těle makra a jestli se jedná o stringifikaci.

Pokud makro obsahuje konkatenaci, neoznačí ho vůbec, protože komentáře mezi parametrem a operátorem `##` by jí zabránily a způsobily by chybu. Myslel jsem, že toto je jediný problém této metody, více o omezeních v sekci 7.6.

Výsledný proud tokenů převede zpět na text. Ten je pak uložen do dočasného souboru. Expanzi provádí Clang, je možné použít i GCC, ale přidá na začátek souboru vlastní komentáře, které by bylo nutné odstranit. Důležité jsou parametry `-E`, aby došlo pouze k předzpracování a ne celé kompilaci, `-xc`, aby kompilátor se vstupním souborem nakládal jako s jazykem C a `-CC`, aby byly zachovány komentáře. Je výhodné ještě použít `-P`, aby preprocesor negeneroval další značky, které *qc2rust* nepoužívá a musel by je odstraňovat.

7.3 Parser

Předzpracovaný výstup z Clangu opět lexer převede na proud tokenů. Komentáře a další bílé znaky jsou přesměrovány a uloženy pro pozdější použití, do parseru předávám pouze tokeny, které se projeví v AST. Plex umožňuje napříč celým parserem sledovat rozsahy [48]. Pro každý neterminál je jeho rozsah složen z rozsahů terminálů a neterminálů, které ho tvoří. Neukládám tedy pozice jednotlivých tokenů, jak jsem původně plánoval, ale pro každý uzel v AST znám rozsah vstupu, ze kterého vznikl, což pro překlad maker i umístění komentářů stačí.

Parser je schopný zpracovat celý Xonotic tak, jak je v commitu `b5c36c20` v repozitáři `xonotic-data.pk3dir` [7]. Ten se liší od hlavní větve (`master`) tím, že jsem přejmenoval parametry dvou funkcí z `var` na `var_`, abych nemusel podporovat klíčová slova jako identifikátory na určitých místech.[12]

7.4 Překladač

Překladač se nachází v modulu `translator`. Před samotnou konverzí mezi jazyky musí zjistit rozsahy expanze maker. Projde seznam komentářů a najde v něm všechny značky. Protože tyto komentáře také zabírají ve zdrojovém kódu určité množství místa a mohou být do sebe i vnořené, nekorespondují jejich pozice přesně s pozicemi podstromů, které ohraničují. Překladač pro

[12] Druhá změna je nahrazení bloku podmíněného překladu obsahujícího kód v jazyce Perl komentářem. V době kdy jsem myslel, že budu podporovat i podmíněný překlad mi to zjednodušovalo lexer, teď to není potřeba.

každou značku tedy nejprve najde nejbližší token z podstromu, který ohraničuje. Pozice těchto tokenů uspořádá a uloží do seznamu, který potom paralelně prochází při průchodu AST.

Překladač prochází AST jazyka QuakeC pouze jednou a během toho generuje AST jazyka Rust s použitím knihovny libsyntax. Tu nepoužívá přímo, ale skrz rozhraní poskytované jedním z podprojektů C2Rust. Tím se snižuje množství repetitivního kódu a do budoucna zjednodušuje výměna kódu mezi C2Rust a *qc2rust*.

Pro jeden výraz z QuakeC vždy generuje jeden výraz v Rustu. Pro jeden příkaz obvykle generuje také jeden příkaz, někdy však více (například rozděluje deklarace více proměnných) a někdy žádný (například pro nadbytečné středníky v QuakeC). Z deklarací podporuje pouze jednoduché funkce, proměnné uvnitř funkcí a `typedef`. Překladač si během průchodu stromu ukládá data pro pozdější využití do struktury `Context`. Většina funkcí pro generování AST Rustu jsem metody na této struktuře.

Zpracování výrazů se provádí metodou `gen_expr`. Jako parametr přijímá výraz v AST QuakeC a vrací výraz v AST Rustu. Protože výrazy často obsahují další výrazy, volá sama sebe rekurzivně. Jak je vysvětlení níže, jedná se o nepřímou rekurzi přes metody `gen_normal_expr` a `gen_expr_rec` (která navíc volá rekurzivně sama sebe, tentokrát přímo).

Při zpracování výrazů *qc2rust* zároveň provádí konverzi preprocesorových maker na deklarativní makra v Rustu. Kdykoliv je `gen_expr` zavolána, vyhledá seznam značek přesně odpovídající podstromu výrazu, který dostala jako parametr a předá ho do `gen_expr_rec`. Pokud je seznam prázdný, tento podstrom nepochází z makra, `gen_expr_rec` tedy vygeneruje normální výraz zavoláním `gen_normal_expr` (která pro podvýrazy volá `gen_expr`) a vrátí ho volající funkci (`gen_expr`). Pokud existují odpovídající značky, znamená to, že tento podstrom je buď expandované makro nebo parametr makra.

Pokud se jedná o expandované makro, `gen_expr_rec` také vygeneruje ze svého parametru výraz, ale nevrací ho – uloží ho do struktury `Context` pro pozdější použití jako tělo deklarativního makra. Místo něho vrátí výraz typu volání makra.

Pokud se jedná o podstrom vzniklý z parametru makra, algoritmus se právě nachází uvnitř těla makra a generuje tedy jeho definici. Funkce tedy vrátí referenci metaproměnné. Předtím ovšem opět vygeneruje normální výraz, který nevrací, ale ukládá – při návratu z rekurze bude použit jako argument aktuálního makra.

Takto by bylo možné podporovat jednoduchá makra, která berou výrazy jako parametry a expandují se na výraz. Makra však mohou být vnořená a to dvěma způsoby: volání makra jako argument dalšího makra nebo volání dalšího makra v těle. Oba způsoby je pochopitelně možné kombinovat. Toto je důvod, proč je `gen_expr_rec` rekurzivní a volá sama sebe.

První případ je jednodušší. Výpis 3 obsahuje vstupní kód a označkovaný expandovaný výsledek, pro čitelnost jsem přidal odsazení. Jak je vidět, značky

```
#define ID1(X) X
#define ID2(Y) Y
void main() {
    print(ID1(ID2(555)));
}

// =========

/*qc2rust::define::ID1::X*/

/*qc2rust::define::ID2::Y*/

void main() {
    print(
        /*qc2rust::macro_begin::ID1*/
            /*qc2rust::param_begin::ID1::X*/
                /*qc2rust::macro_begin::ID2*/
                    /*qc2rust::param_begin::ID2::Y*/
                        555
                    /*qc2rust::param_end::ID2::Y*/
                /*qc2rust::macro_end::ID2*/
            /*qc2rust::param_end::ID1::X*/
        /*qc2rust::macro_end::ID1*/
    );
}
```

Výpis kódu 3: Makro jako argument makra

tvoří závorkovou strukturu. Jak jsem zmínil výše, nepoužívají se skutečné pozice značek, ale příslušných tokenů. Všechny značky tedy mají v tomto případě efektivně stejnou pozici, vhodným seřazením je však zachováno pořadí a budou navštívené od vnějších k vnitřním.

Závorková struktura je zpracována rekurzivně. Při zpracování tohoto výrazu `gen_expr` nalezne všechny čtyři značky (značka je pár začátek–konec) a s nimi zavolá metodu `gen_expr_rec`. Zpracování probíhá jak je popsáno výše, jen je potřeba upřesnit co se myslí uložením.

V případě značky, která odpovídá expanzi makra, je jeho název uložen na zásobník. Při dalším zanořování do rekurze (tj. existují další značky) k názvu jsou přiřazeny výrazy odpovídající argumentům. Po vynoření z rekurze tedy `gen_expr_rec` vygeneruje volání makra s těmito argumenty.

V případě značky, která odpovídá argumentu, je tedy tento argument přiřazen odpovídajícímu názvu makra. tím se dostáváme k případu, kdy makro v těle volá další makro. Výpis 4 obsahuje ukázku. Jak je vidět, značky pocháze-

```
#define ID1(X) X
#define ID2(Y) ID1(Y)
void main() {
    print(ID2(555));
}

// =========

/*qc2rust::define::ID1::X*/

/*qc2rust::define::ID2::Y*/

void main() {
    print(
        /*qc2rust::macro_begin::ID2*/
            /*qc2rust::macro_begin::ID1*/
                /*qc2rust::param_begin::ID1::X*/
                    /*qc2rust::param_begin::ID2::Y*/
                        555
                    /*qc2rust::param_end::ID2::Y*/
                /*qc2rust::param_end::ID1::X*/
            /*qc2rust::macro_end::ID1*/
        /*qc2rust::macro_end::ID2*/
    );
}
```

Výpis kódu 4: Makro v těle makra

jící z různých maker se prolínají. Nestačí tedy argument přiřadit názvu makra na vrcholu zásobníku. Proto značky `param_{begin,end}` obsahují nejen název parametru, ale i název makra. Z tohoto důvodu musí metoda `gen_expr_rec` zásobník projít, porovnávat název makra na zásobníku a na parametru a přiřadit hodnotu argumentu pouze pokud se názvy shodují.

Tyto ukázky používají pro jednoduchost makra, která se expandují na svůj argument bez jakýchkoliv změn. Stejný postup funguje i na makra, která mají více parametrů nebo některé parametry používají vícekrát.

Po průchodu stromem má překladač hotový vygenerovaný AST Rustu a ve struktuře `Context` má uložené vygenerované výrazy odpovídající tělům maker (v místě, kde původní makro obsahovalo parametr obsahují referenci metaproměnné). Pro každý z těchto výrazů vytvoří definici makra se správným počtem parametrů a přidá ji na začátek AST Rustu. Tím jsou makra dostupná v celém souboru.

```
#define CAT(X) A##X
#define CAT2(X) CAT(/*tag*/X/*tag*/)
CAT2(42)
```

Výpis kódu 5: Nepřímá konkatenace

7.5 Výpis

K výpisu používám stejně jako C2Rust knihovnu libsyntax. Té předám AST Rustu a seznam komentářů (komentáře použité k označení maker jsou odstraněny). Většina uzlů AST a každý komentář má s sebou pozici, na kterou patří. Libsyntax prochází AST a pokud má vypsat uzel, který následuje po ještě nevypsaném komentáři, vypíše před něj tento komentář [22]. Projekt C2Rust využívá stejné rozhraní libsyntax, ale protože nezachovává přesné pozice všech uzlů, zvolili obrácený přístup. Přidání komentáře do jejich úložiště (`CommentStore`) vrací pozici následující za tímto komentářem. Ta je použita při generování uzlu následujícím po tomto komentáři.

7.6 Omezení

Při návrhu jsem se domníval, že bude pomocí komentářů možné označit a poté nechat expandovat všechna makra. Nebyl jsem si jistý pouze podporou konkatenace. Poté, co jsem tuto metodu implementoval, jsem objevil několik problémů, které se mi nepodařilo vyřešit.

7.6.1 Konkatenace

Makra konkatenující své parametry *qc2rust* neoznačuje. Problém s nimi jsem objevil relativně brzy při testování. Při použití pouze `-E` se expandují správně, pokud je ale zároveň použito `-CC`, které je potřeba aby se značky a komentáře zachovaly, tak se berou jako běžné tokeny a narušují konkatenaci.

Přišel jsem tedy o možnost podporovat překlad konkatenace, ale domníval jsem se, že všechna ostatní makra bude možné označit. Do takového makra však může přijít argument již označený a tím také způsobit chybu. Výpis 5 obsahuje příklad takovéto situace (skutečné komentáře používané ke značkování mají v sobě více informací). Expanzí vznikne `A##/*tag*/X/*tag*/` a preprocesor v GCC i Clangu zahlásí chybu.[13]

Řešení je najít všechna makra, která své parametry tranzitivně předávají do maker s konatenací a zakázat jejich značkování. Názvy takových maker však

[13] Značkování maker jsem pochopitelně testoval od začátku na celém Xonoticu, ne jen malých příkladech, které jsem si vymyslel. Úspěšné označení jsem kontroloval tím, že se výsledný kód správně zkompiloval a šel spustit. Neuvědomil jsem si však, že Xonotic při kompilaci nepoužívá parametr -CC, takže jeden z kroků kompilace Xonoticu je odstranění komentářů. Chyba se tak neprojevila a já se domníval, že značkování funguje.

```
#define EXPAND_MACRO(MAC, param) MAC(param)
#define INDIRECT(MAC, p) EXPAND_MACRO(/*tag*/MAC/*tag*/, p)
#define backtrace(param) some_fn(param)
INDIRECT(backtrace, "log indirect");
```

Výpis kódu 6: Název funkčního makra jako parametr

mohou také vzniknout konkatenací, je tedy těžké najít je všechna v obecném případě bez provedenní expanze.

Protože jsem se chtěl vyhnout reimplementaci velkých částí preprocesoru jako je algoritmus expanze, vymyslel jsem jednodušší způsob. Chybové hlašky z kompilátorů obsahují názvy maker v nichž došlo a pokus konkatenovat komentář. Lze je ve výpisu najít například pomocí nástroje `grep`, zakázat jejich značkování v modulu `preprocessor` a zpracovat vstup znovu. Protože makro, které konkatenuje může vzniklé tokeny předávat do dalších maker, která také konkatenují, je tento postup nutné několikrát opakovat a zakazovat značkování dalších maker, dokud jsou ve výpisu chyby. Makra, která tyto chyby tranzitivně způsobí tedy nelze pomocí metody značkování přeložit do Rustu.

7.6.2 Makra jako argument

Druhý problém se značkováním maker pomocí komentářů je případ, kdy makro přijímá název jiného makra jako argument a používá ho jako volání funkčního makra. Ukázka inspirovaná kódem Xonoticu je ve výpisu 6.

Opět je lehké detekovat případy, kdy se jedná pouze o přímé volání, stačí neoznačovat paramatr, za kterým následuje závorka (zde parametr `MAC` v `EXPAND_MACRO`). Obecný případ je však mnohem těžší detekovat, protože nedojde k chybě při předzpracování, ale až při samotné kompilaci. Ta se navíc může projevit mnoha různými způsoby. V předzpracovaném kódu zůstane název makra, což typicky způsobí, že takový symbol není nalezen. Je však možné, že taková funkce bude existovat (například pokud by byla skryta tímto makrem) a pak může v obecném případě dojít téměř k libovolné chybě.

Tento problém by bylo možné částečně vyřešit v překladači GCC pomocí přepnače `-traditional`, ten ale způsobí, že nelze přeložit jiná makra, která Xonotic potřebuje. Navíc jedna značka je zachována a druhá (mezi názvem makra a závorkou) je odstraněna, vznikne neukončené označení, které by bylo nutné detekovat.

7.6.3 Další omezení

Protože jsem se v práci zaměřil na podporu maker, hodně času jsem strávil hledáním k tomu potřebných nástrojů, a překladač aktuálně podporuje pouze velmi malou část QuakeC. Pro překlad konstrukcí, které nemají v Rustu ekvivalent jako `switch` s propadáváním a `goto` jsem původně plánoval použít

algoritmus *Relooper* ze C2Rust. Teprve později jsem zjistil, že *Relooper* je v C2Rust hluboce provázán s překladem příkazů (*statements*) a závisí na několika strukturách z modulu `c_ast`. Jeho úprava pro použití v *qc2rust* by vyžadovala příliš práce.[14]

QuakeC umožňuje použít typ `float` v podmínkách podobně jako C, Rust nikoliv. Při převodu je tedy nutné vygenerovat dodatečné konverze. *Qc2rust* nesleduje typy výrazů a přidává konverze i kde to není potřeba, což značně snižuje čitelnost. Podpora jných typů je ještě omezenější.

Nestihl jsem kód rozdělit do modulů ani odstranit globální proměnné[15], jak jsem navrhoval v zadání.

Samotná podpora maker také není úplná, kromě výše zmíněných omezení není implementována stringifikace a změna definice makra pomocí direktivy `#undef` a opětovného `#define`. Pro změnu definice by bylo nutné uložit do značky nejen název makra, ale i jeho pozici. Rust navíc nemá ekvivalent `#undef`, aby se předešlo kolizím, bylo by jedno z nich nutné přejmenovat.

Dále *qc2rust* pro jednoduchost generuje definice maker v globálním prosturu. V QuakeC mohou být definována kdekoliv mezi libovolnýni tokeny (kamkoliv lze vložil nový řádek), v Rustu pouze na místech, kde mohou být deklarace. Bylo by vhodné makra generovat co nejblíže k jejich definici v QuakeC.

Makra v QuakeC navíc mohou referencovat proměnné. V Rustu je nutné, aby proměnná byla v místě deklarace makra viditelná. Nestačí tedy generovat deklaraci co nejblíže k původní definici v QuakeC, ale některá makra může být nutné přesunout mezi deklarace proměnných, které používá, a jejich použití.

V případech, kdy strom expanzí maker neodpovídá abstraktnímu syntaktickému stromu, tedy jedná se o makro nepřeložitelné do Rustu, může dojít k několika situacím. V ideálním případě *qc2rust* vygeneruje podstrom AST Rustu obsahující expandované makro. Jsou však i situace, kdy se tento podstrom nachází uvnitř definice jiného makra. Pokud toto nepřeložitelné makro navíc přijímá parametry, může se pokaždé expandovat na jiný podstrom. Tyto podstromy pak není možné použít jako definici makra – ta může být jen jedna. *Qc2rust* se tuto situaci někdy schopný detekovat a zobrazit varování, někdy ne.

Protože *qc2rust* je spíše prototyp demostrující proveditelnost převodu maker a QuakeC do Rustu a není připraven na skutečné použití, nevytvářím k němu uživatelskou příručku. Přijímá jako parametr název souboru ze zdro-

[14] Xonotic obsahuje odhadem pomocí nástroje grep kolem 50 goto. Nepodařilo se mi zjistit počet příkazů switch, které využívají propadávání, ty by však bylo možné přeložit do příkazu match v Rustu, jen by některé větve obsahovaly duplikovaný kód.

[15] K tomu jsem chtěl využít kód ze C2Rust, který je součástí algoritmu *Relooper*, ukázalo se však, že je se zbytkem tohoto projektu příliš provázaný. C2Rust však obsahuje refaktorovací nástroj, který převod globálních proměnných na lokální umožňuje i na již vygenerovaném kódu. Pokud by fungoval tak, jak je v ukázkách v manuálu, nemělo by smysl reimplementovat stejnou funkčnost v *qc2rust*.

jovým kódem QuakeC. V něm označí makra, nechá je expandovat preprocesorem z Clangu, výstup přeloží do Rustu, nechá ho zformátovat pomocí `rustfmt`, zkompiluje ho pomocí `rustc` a spustí. Dočasné soubory i generovaný kód zůstávají ve složce `tmp`. Zároveň v `main.rs` obsahuje funkce pro jednoduché testování jednotlivých částí *qc2rust*.

7.7 Testování

Části překladače, které jsou schopné zpracovat celý Xonotic jsem testoval na zdrojovém kódu Xonoticu. Vzhledem k výše zmíněným problémům s označováním maker a dalším omezením konverze jsem nemohl testovat překlad maker na celém Xonoticu. Vybral jsem tedy pouze některá podporovaná makra ze Xonoticu a jimi se inspiroval při tvorbě jednotkových testů (*unit tests*). Vzhledem k malému množství konstrukcí, které překladač QuakeC podporuje, jsem ho také testoval pouze na malých příkladech. Vypisování Rustu používá knihovnu libsyntax, která je součástí kompilátoru Rustu, `rustc`, neměla by tedy mít potíže s libovolným zdrojovým kódem a jeho testování jsem nevěnoval zvýšenou pozornost.[16]

[16] Jak jsem zjistil, zpracuje správně i případ, kdy QuakeC používá identifikátor, který je v Rustu klíčové slovo – převede ha na tzv. surový (*raw*) identifikátor. Pro čitelnost by však možná bylo lepší nahradit takový název jiným, například na konec přidat podtržítko, bylo by však nutné zamezit kolizím.

Závěr

Cílem práce bylo navrhnout překladač z jazyka QuakeC do jazyka Rust, který zároveň převádí preprocesorová makra do deklarativních v Rustu a zachovává čitelnost kódu. Popsal jsem jazyky QuakeC a Rust a preprocesor jazyka C, který QuakeC používá. Zdokumentoval jsem některé nejasné části syntaxe QuakeC. Některé konstrukce obou jazyků jsem porovnal a zvážil možnosti převodu mezi nimi.

Problém převodu maker do té doby ještě nikdo neřešil, primárně jsem se tedy zaměřil na ten. Lexer a parser zpracují celý Xonotic, zbytek překladače bohužel z časových důvodů podporuje pouze výrazy, některé příkazy a jednoduché funkce.

Zadání zmiňuje zachování datových typů, rozdělení do modulů a omezení globálních proměnných. Uživatelsky definované typy (pomocí `typedef`) *qc2rust* zachovává, na zbylé dvě položky nezbyl čas, avšak na odstranění globálních proměnných by mělo být možné použít existující refaktorovací nástroje.

Narozdíl od předchozích pokusů překládat QuakeC tato práce poskytuje dobrý základ pro generování čitelného kódu. Podařilo se mi zachovávat komentáře lépe, než to dělá C2Rust při překladu C.

Překlad maker se dá se rozdělit na dvě fáze – zjištění průběhu expanzí preprocesoru a zpětné složení expandovaného kódu do deklarativních maker. Zkoušel jsem k tomu potřebné informace získat dvěma způsoby – použitím projektu Clang a označováním pomocí komentářů. Od Clangu jsem upustil, protože se mi dlouho nedařilo získat všechny nutné informace, zvláště pro převod funkčních maker, a měl jsem pocit, že jsem ve slepé uličce. Také jsem se zbytečně zabýval možností podpory podmíněné kompilace, na kterou nezbyl čas. Přístup k překladu maker pomocí komentářů částečně dospěl k cíli, ale nedokáže podporovat všechny případy. V průběhu práce v C2Rust přibyla podpora objektových maker používající Clang. Další pokusy s převodem maker by tedy měly znovu zvážit použití Clangu, popřípadě jeho rozšíření.

Podařilo se mi vytvořit algoritmus pro převod objektových i funkčních

maker s parametry včetně vnořených maker. Aktuálně zvládá vytvořit deklarativní makra pracující s výrazy a mohl by se uplatnit v C2Rust.

Bibliografie

[1] Olivier Montanuy, Francesco Ferrara. QuakeC Manual [online]. URL: http://www.cataboligne.org/extra/qcmanual.html (citováno 24. 04. 2019).

[2] David „DarkGrue“ Hesprich, Olivier Montanuy, Francesco Ferrara. QuakeC Reference Manual [online]. 4. února 1998. URL: http://pages.cs.wisc.edu/~jeremyp/quake/quakec/quakec.pdf (citováno 24. 04. 2019).

[3] Team Xonotic. Introduction to QuakeC [online]. 2017. URL: https://gitlab.com/xonotic/xonotic/wikis/Introduction-to-QuakeC (citováno 24. 04. 2019).

[4] QuakeC [online]. 2009. URL: http://ouns.nexuizninjaz.com/dev_quakec.html (citováno 24. 04. 2019).

[5] Dale „graphitemaster“ Weiler, Wolfgang „Blub\w“ Bumiller. GMQCC [software]. 2019. URL: https://github.com/graphitemaster/gmqcc (citováno 24. 04. 2019).

[6] Id Software, Inc., Forest Hale, aj. DarkPlaces [software]. 2019. URL: https://gitlab.com/xonotic/darkplaces (citováno 07. 05. 2019).

[7] Team Xonotic. Xonotic assets and gamecode [software]. 2019. URL: https://gitlab.com/xonotic/xonotic-data.pk3dir/ (citováno 07. 05. 2019).

[8] FTEQCC [software]. 2019. URL: http://triptohell.info/moodles/fteqcc/ (citováno 24. 04. 2019).

[9] Team Xonotic. Xonotic [software]. 2019. URL: https://gitlab.com/xonotic (citováno 07. 05. 2019).

[10] Team Xonotic. Repository Access and Compiling [online]. 2019. URL: https://gitlab.com/xonotic/xonotic/wikis/Repository_Access (citováno 08. 05. 2019).

[11] Free Software Foundation, Inc. The C Preprocessor [online], 2019. URL: https://gcc.gnu.org/onlinedocs/cpp/index.html (citováno 24. 04. 2019).

[12] Free Software Foundation, Inc. gcc(1) - Linux man page. 2018. URL: https://linux.die.net/man/1/gcc (citováno 09. 05. 2019).

[13] The Clang Team. Clang [software]. URL: https://clang.llvm.org/ (citováno 09. 05. 2019).

[14] Steve Klabnik, Carol Nichols. The Rust Programming Language [online]. 2019. URL: https://doc.rust-lang.org/book/ (citováno 25. 04. 2019).

[15] The Rust Project Developers. Rust by Example [online]. 2019. URL: https://doc.rust-lang.org/rust-by-example/index.html (citováno 25. 04. 2019).

[16] Steve Donovan. A Gentle Introduction To Rust [online]. 2018. URL: https://stevedonovan.github.io/rust-gentle-intro/readme.html (citováno 25. 04. 2019).

[17] Niko Matsakis. Unification in Chalk, part 1 [online]. 25. března 2017. URL: http://smallcultfollowing.com/babysteps/blog/2017/03/25/unification-in-chalk-part-1/#conclusion (citováno 25. 04. 2019).

[18] Graydon Hoare. Rust Prehistory Repo - November 2016 [online]. 2016. URL: https://github.com/graydon/rust-prehistory (citováno 25. 04. 2019).

[19] The Rust Core Team. Announcing Rust 1.0 [online]. 15. kvetna 2015. URL: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html (citováno 25. 04. 2019).

[20] The Rust Project Developers. The Rust Standard Library [online]. 2019. URL: https://doc.rust-lang.org/std/index.html (citováno 25. 04. 2019).

[21] The Rust Project Developers. The Rust Reference [online]. 2019. URL: https://doc.rust-lang.org/reference/influences.html (citováno 08. 05. 2019).

[22] The Rust Project Developers. Crate syntax [online]. 2019. URL: https://doc.rust-lang.org/nightly/nightly-rustc/syntax/index.html (citováno 10. 05. 2019).

[23] Steve Klabnik, Carol Nichols. The Rust Programming Language [First Edition] [online]. 2018. URL: https://doc.rust-lang.org/1.30.0/book/first-edition/macros.html (citováno 25. 04. 2019).

[24] Andrew „TimePath“ Hardaker. jcompile [software]. 2016. URL: https://github.com/TimePath/jcompile (citováno 21. 04. 2019).

[25] Jamey Sharp. Corrode: Automatic semantics-preserving translation from C to Rust [software]. 2017. URL: https://github.com/jameysharp/corrode (citováno 21. 04. 2019).

[26] Galois, Immunant. C2Rust [software]. 2018. URL: https://c2rust.com/ (citováno 21. 04. 2019).

[27] Sam Harwell. C 2011 grammar built from the C11 Spec [online]. 2013. URL: https://github.com/antlr/grammars-v4/blob/master/c/C.g4 (citováno 22. 04. 2019).

[28] Jamey Sharp. C2rust vs corrode [online]. 30. června 2018. URL: https://jamey.thesharps.us/2018/06/30/c2rust-vs-corrode/ (citováno 21. 04. 2019).

[29] Jamey Sharp. corrode/src/Language/Rust/Corrode/C.md [software]. 2017. URL: https://github.com/jameysharp/corrode/blob/master/src/Language/Rust/Corrode/C.md#top-level-translation (citováno 21. 04. 2019).

[30] Immunant, Galois. C2Rust [software]. 2019. URL: https://github.com/immunant/c2rust (citováno 22. 04. 2019).

[31] Eric Mertens, Galois. C2rust [online]. 14. srpna 2018. URL: https://galois.com/blog/2018/08/c2rust/ (citováno 22. 04. 2019).

[32] Immunant, Galois. C2Rust Manual [online]. 2019. URL: https://c2rust.com/manual/ (citováno 22. 04. 2019).

[33] Immunant, Galois. C2Rust: Known Limitations of Translation [online]. 2019. URL: https://github.com/immunant/c2rust/blob/master/docs/known-limitations.md (citováno 22. 04. 2019).

[34] Immunant, Galois. C2Rust: Support translation of C macros to Rust [online]. 2019. URL: https://github.com/immunant/c2rust/issues/16 (citováno 22. 04. 2019).

[35] Kiyoshi Matsui. mcpp – a portable C preprocessor [software]. 2008. URL: http://mcpp.sourceforge.net/ (citováno 11. 05. 2019).

[36] Shevek. JCPP - A Java C Preprocessor [software]. 2018. URL: https://www.anarres.org/projects/jcpp/ (citováno 11. 05. 2019).

[37] Walter Bright, aj. warp: Facebook's C and C++ Preprocessor [software]. 2015. URL: https://github.com/facebookarchive/warp (citováno 11. 05. 2019).

[38] Hartmut Kaiser. Wave V2.3 [software]. 2008. URL: https://www.boost.org/doc/libs/1_70_0/libs/wave/ (citováno 11. 05. 2019).

[39] Purnank Harjivanbhai Ghumalia, Meera Purnank Ghumalia. 2.1.7. Macro Expansion [online]. 2017. URL: http://eclipsebook.in/c-cpp-development/reading-code/macro-expansions/ (citováno 11. 05. 2019).

[40] Clang / LLVM Team. clang(1) - Linux man page. 2018. URL: https://linux.die.net/man/1/clang (citováno 11. 05. 2019).

[41] The Clang Team. libclang: C Interface to Clang [online]. 2019. URL: https://clang.llvm.org/doxygen/group__CINDEX.html (citováno 12. 05. 2019).

[42] The Clang Team. clang Namespace Reference [online]. 2019. URL: https://clang.llvm.org/doxygen/namespaceclang.html (citováno 11. 05. 2019).

[43] Kyle Mayes. clang-sys [software]. 2019. URL: https://crates.io/crates/clang-sys (citováno 11. 05. 2019).

[44] Kyle Mayes. clang-rs [software]. 2019. URL: https://crates.io/crates/clang (citováno 11. 05. 2019).

[45] The Clang Team. pp-trace User's Manual [software]. 2019. URL: https://clang.llvm.org/extra/pp-trace.html (citováno 11. 05. 2019).

[46] Terence Parr, aj. ANTLR v4 [software]. 2019. URL: https://github.com/antlr/antlr4 (citováno 11. 05. 2019).

[47] Geoffroy Couprie. nom [software]. 2019. URL: https://crates.io/crates/nom (citováno 13. 05. 2019).

[48] Geoffry Song. plex [software]. 2019. URL: https://crates.io/crates/plex (citováno 13. 05. 2019).

Příloha A

Seznam použitých zkratek

AST Abstract Syntax Tree

CBOR Concise Binary Object Representation

FTEQCC ForeThought Entertainment QuakeC Compiler

GCC GNU Compiler Collection

GMQCC GraphiteMaster's QuakeC Compiler

IDE Integrated Development Environment

IRC Internet Relay Chat

QCVM QuakeC Virtual Machine

Příloha **B**

Obsah přiloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `src`
  - `qc2rust` ..... zdrojové kódy implementace
  - `c2rust` ..... zdrojové kódy závislosti
- `text` ..... text práce
  - `pdf` ..... text práce ve formátu PDF
  - `latex` ..... zdrojové kódy práce ve formátu LaTeX