VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLADAČ PODMNOŽINY JAZYKA PYTHON

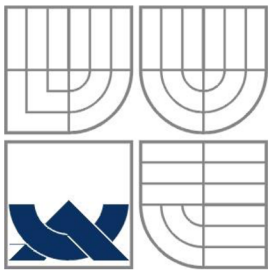
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

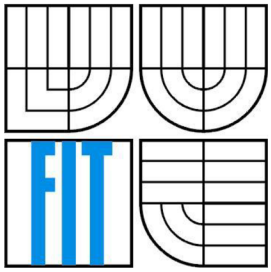
Bc. Radek Falhar

BRNO 2014

ORIGINÁLNÍ ZADÁNÍ



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLADAČ PODMNOŽINY JAZYKA PYTHON

A COMPILER OF LANGUAGE PYTHON SUBSET

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Radek Falhar

VEDOUCÍ PRÁCE
SUPERVISOR

Kolář Dušan, doc. Dr. Ing.

BRNO 2014

Abstrakt

Python je dynamicky typovaný, interpretovaný programovací jazyk. Díky dynamickému typovému systému je tedy obtížné jej zkompilovat do statického zdrojového kódu. Tedy kódu, kde je přesně dáno, jaké typy existují a jaká je jejich struktůra. Existuje několik způsobů jak tohoto dosáhnout a jedním z primárních je typová inference. Tento přístup se snaží určit struktura typů ze zdrojového kódu. V případě jazyka Python je však tento přístup obtížný, protože výsledný typový systém je velice komplexní a jazyk samotný není k typové inferenci navržen.

V této práci jsem se zaměřil na identifikaci podmnožiny tohoto jazyka, aby byla možná typová inference při zachování co nejpřirozenějšího použití jazyka. Následně jsem implementoval překladač, který tuto podmnožinu přeloží do staticky typovaného jazyka, který pak lze přeložit do nativního kódu.

Abstract

Python is dynamically typed interpreted programming language. Thanks to its dynamic type system, it is difficult to compile it into statically typed source code. The kind of source code, where it is exactly specified what types exist and what their structure is. Multiple approaches exist how to achieve this and one of the primary ones is type inference. This approach is attempting to infer the type structure from the source code. In case of Python language, this approach is difficult, because resulting type system is quite complex and language itself is not designed for type inference.

In this work, I have focused on identifying subset of this language, so that type inference is possible while keeping the natural way the language is used. Then I implemented a compiler, which will compile this subset into statically typed language, which can be translated into native code.

Klíčová slova

Python, překladač, typová inference, C++, Hindley-Millner

Keywords

Python, compiler, type inferrence, C++, Hindley-Millner

Citace

Radek Falhar: Překladač podmnožiny jazyka Python, diplomový projekt, Brno, FIT VUT v Brně, rok

Překladač podmnožiny jazyka Python

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Dušana Koláře, doc. Dr. Ing.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Bc. Radek Falhar
26.5.2014

Obsah

1 Úvod

Python je multiparadigmatický, dynamicky typovaný jazyk s jednoduchou syntaxí a sémantikou. Od svého vzniku si získal velké množství příznivců a to díky své vyjadřovací síle i přes jednoduchou syntaxi. Hlavní výhodou je zejména dynamický typový systém, který zjednodušuje tvorbu kódu, protože programátor nemusí do kódu zapisovat typy proměnných a atributů. Dynamický typový systém také dává možnost metaprogramování, kdy kód může za běhu měnit strukturu datových typů a z toho vytvářet novou funkčnost. Tento dynamický systém má ale i své nevýhody.

První je nemožnost před spuštěním programu zajistit typovou korektnost programu. Tedy že kód je schopen pracovat s typy, které jsou mu předávány. Řešení tohoto problému vyžaduje zvýšené množství práce programátora, např. tvorbou automatizovaných testů. Druhým problémem je rychlost provádění kódu. Díky tomu, že struktura dat není známá před spuštěním programu, je nemožné provádět většinu optimalizací, které lze provádět nad staticky typovaným kódem. Výsledkem je obecně pomalejší běh programů v jazyce Python, než je jejich ekvivalent ve staticky typovaném jazyku.

Možností, jak tyto problémy obejít je pokusit se otypovat kód, aby bylo možné provádět optimalizace a typovou kontrolu. První možností je rozšířit syntaxi jazyka o deklaraci typů proměnných a atributů. Problémem tohoto řešení je další práce programátora, zhoršení čitelnosti kódu a ztráta možnosti použít standardní interpret jazyka. Velkým problémem je také to, že takto vždy není možné otypovat celý program a to buď kvůli jeho složitosti, nebo rozsáhlosti. Je tedy nutné, aby takovýto překladač podporoval i dynamické typování a přechody ze statického do dynamického kódu.

Druhou možností je použít typovou inferenci, tedy odvození konkrétních typů ze zdrojového kódu. To nevyžaduje další práci od programátora a kód zůstává kompatibilní s originálními interprety. Sémantika jazyka ale musí být omezena, aby bylo možné typovou inferenci provést. Komplikací také je, že jazyk nebyl pro typovou inferenci navržen. Proto je výsledný statický typový systém a odvozovací algoritmus velice komplexní. I přesto existuje několik implementací, které se snaží odvozovat typy nad konkrétní, předem definovanou podmnožinou jazyka. Tyto implementace však mohou mít v některých oblastech jazyka problémy.

V rámci tohoto projektu navrhl a implementoval překladač a typovou inferenci jazyka Python 3.3. Tento překladač vykazuje vlastnosti, které ostatní překladače nemají. Výstupem překladače je přeložitelný program v jazyce C++. V rámci projektu jsem implementoval základní knihovnu, aby bylo možné jazyk použít k základnímu programování.

V druhé kapitole je detailněji popsán jazyk Python a existující překladače a algoritmy pro typovou inferenci. Ve třetí kapitole je popsán výběr přeložitelné podmnožiny jazyka a návrh typového systému a typové inference. Ve čtvrté kapitole je popsán převod otypovaných konstrukcí jazyka Python na konstrukce jazyka C++. V páté kapitole jsou popsány detaily implementace odvozování typů a překladu. V poslední kapitole je zhodnocení funkčnosti a výkonu překladače.

2 Existující algoritmy a implementace

2.1 Jazyk Python

Jazyk Python [1] je obecně použitelný[1], vysokoúrovňový programovací a skriptovací jazyk. První použitelná verze tohoto jazyka byla představena v roce 1991 Guido van Rossumem. Od té doby se jazyk a jeho implementace vyvíjí a v současné době je nejnovější verze 3.3. Filozofií tohoto jazyka je vysoká čitelnost kódu a vysoká vyjadřovací schopnost na malém počtu řádků. Python je multiparadigmatický jazyk. Podporuje imperativní a objektové programování a má také funkcionální rysy.

2.1.1 Syntaxe jazyka

Syntaxe jazyka Python je jednoduchá a řídí se heslem „Měl by existovat jen jeden zřejmý způsob jak to udělat.“. Ta je podobná jiným imperativním jazykům. Přiřazení do proměnné je operátorem =, volání funkcí je *jmenoFunkce(parametry)*, atd. Hlavním rozdílem oproti ostatním imperativním jazykům je způsob oddělení bloků kódu. Python místo tokenů pro začátek a konec bloku využívá odsazení kódu bílými znaky. Kód, který je odsazený stejným počtem bílých znaků, je chápán jako jeden blok. Toto dává kódu v jazyce python minimalistický vzhled a zajišťuje, že funkční kód bude vždy správně odsazen.

2.1.2 Typový systém

Typový systém jazyka Python je dynamický a celý stojí na principu tzv. duck typing**.** Tedy myšlence, kde kód nevyžaduje u hodnoty proměnné nebo parametru funkce konkrétní typ, ale stačí mu, když typ hodnoty podporuje použité metody[2] a atributy. To dává jazyku vysokou úroveň polymorfismu známou u dynamicky typovaných jazyků. Jazyk poskytuje několik základních typů: `string`, `int`, `float`, `boolean`, `list`, `tuple`, `dict`, `set`, `bytearray`, `bytes` a `complex`. Některé z nich jsou jednoduché, jako například `int` a `float`, které reprezentují čísla. Jiné jsou komplexnější, jako například `tuple`, který reprezentuje heterogenní n-tici hodnot.

Uživatel může definovat vlastní typy pomocí tříd, které podporují atributy, třídní metody a vícenásobnou dědičnost. Třídy však neomezují, jaké atributy a metody může daná instance obsahovat. Za běhu programu lze atributy a metody konkrétních instancí přidávat nebo měnit. Python také podporuje *magické metody* [2], které mohou měnit chování třídy na sémantické úrovni. Například metoda *__getattr__* ovlivňuje, jaká data se vrátí při přístupu k atributu a je tedy možné měnit typ atributu v závislosti na stavu instance.

Funkcionální vlastnosti jazyka Python jsou primárně v možnosti ukládat funkce do proměnných, atributů, předávat je jako parametry nebo jako návratové hodnoty funkcí. Funkcionální chování je v jazyce integrální, protože metody u instancí jsou ve skutečnosti uloženy jako funkce uložené v atributech a při volání metody se čtou stejně jako hodnotové atributy. Jazyk Python také umožňuje definovat nové funkce v definici jiné funkce. Toto umožňuje využití uzávěrů funkcí, které dávají nově

[1] Obecně použitelný ve velkém množství domén na rozdíl od doménově specifických jazyků. Ne nutně ve všech.
[2] Metodou je myšlena funkce, která je vázaná na instanci třídy

definované funkci možnost přístupu k lokálním proměnným uzavírané funkce i po jejím dokončení a to i více vloženým funkcím. Každé volání funkce obsahující vložené funkce pak vytváří nový uzávěr s novými proměnnými. Tímto lze simulovat zapouzdření podobně jako v objektovém programování.

I přesto, že je typový systém jazyka Python jednoduchý, jeho dynamičnost a možnosti měnit strukturu dat a tím i význam příkazů jazyka, prakticky znemožňují jakoukoliv statickou analýzu a tedy i kompilaci.

2.2 Existující překladače

Pro plnohodnotné provozování jazyka Python je nutná jeho interpretace. Tou se v této práci nebudu zabývat. Existuje ale několik implementací, které se snaží jazyk překládat. Překlad lze v tomto kontextu rozdělit na dva druhy: překlad instrukcí a překlad typového systému.

Překlad instrukcí je v jazyku Python relativně přímočarý. Hlavně díky tomu, že jazyk neobsahuje žádné konstrukce, které by mu umožnily za běhu měnit existující kód, pouze jej rozšiřovat.

Překlad typového systému je však mnohem obtížnější a to díky jeho dynamičnosti. Toto lze vyřešit dvěma způsoby. Prvním je přidat do syntaxe jazyka konstrukce, které programátorovi umožní určit typy proměnných a atributů. Druhým je použití typové inference, která se bude snažit určit typy z jejich použití.

2.2.1 Nuitka

Nuitka [3] je jednoduchý překladač, který překládá pouze instrukce a typový systém nechává dynamický. Jeho jedinou výhodou je tedy odstranění nutnosti provádět syntaktickou analýzu a interpretaci zdrojového kódu. Nedostatky typového systému, tedy nemožnost optimalizací a kontroly typů neřeší.

2.2.2 Cython

Cython [4] je překladač, který je primárně určen k provázání kódu v jazyce Pythnon s kódem v nativních jazycích, tedy hlavně kódem v C. V případě nativního kódu se v jazyce Python překládají jen instrukce a typový systém nechává dynamický. Neomezuje tedy možnosti jazyka samotného. Přidává ale novou syntaxi, které programátorovi umožňují určit typy proměnných, atributů tříd a parametrů funkcí. Z takto "otypovaného" kódu pak překladač generuje kód, který pracuje nad statickými typy. Toho lze využít pro optimalizace, typovou bezpečnost, nebo právě integraci dynamického a statického typového sytému. Hlavním přínosem překladače Cython je však schopnost generovat zapouzdření nad tímto staticky typovaným kódem, které umožňují volat dynamický kód ze statického a obráceně. I použití tohoto překladače není přímočaré pro překlad do spustitelné podoby, ale je uzpůsobeno faktu, že překladač má hlavně sloužit jako generátor knihoven, které zapouzdřují práci se staticky typovanými knihovnami.

2.2.3 RPython

Restricted Python (RPython) [5] je součástí projektu PyPy a jedná se o nejvyspělejší překladač jazyka Python. Cílem projektu PyPy je vytvořit „interpret jazyka Python napsaný v jazyce Python", se zaměřením na vysokou modularitu. Ve skutečnosti je PyPy napsán v jazyce RPython. Ten je podmnožinou jazyka Python takovou, že je možné nad ní provést typovou inferenci a kompilaci do

staticky typovaného kódu. V současné době podporuje výstup do jazyka C a platforem JVM a .NET/CLI. Podporovaná podmnožina jazyka má však velice omezenou dynamičnost a není tedy moc dobře použitelný jako překladač existujícího kódu, který tyto dynamické vlastnosti jazyka využívá. Pokud se však kód upraví tak, že jej lze zkompilovat, může jeho rychlost dosahovat rychlosti nativních aplikací. Tedy několikanásobně lepší než je nejrychlejší interpret.

Problémem překladače je omezený duck typing. To se projevuje tak, že typy parametrů funkcí se inferují z jejich prvního použití. Následující kód ilustruje toto omezení:

```
def add(x, y):
     return x+y

print(add(1, 2))  #validni
print(add('a', 'b'))  #nevalidni
```

Zajímavostí implementace překladače RPythonu je jeho podpora metaprogramování. Překlad totiž neprobíhá ze zdrojového kódu, ale zdrojový kód je nejdříve interpretován a překlad samotný probíhá z interní reprezentace objektů a bytekódu v interpretu. Díky tomu dojde k provedení metakódu, který se provádí při spuštění programu. Tyto interní objekty a kód v nich obsažený však déle nemohou používat dynamický systém, protože nad ním by nešla provést typová inference. Interpret zde tedy funguje jako syntaktický analyzátor a preprocesor zároveň. Tento způsob tedy nepodporuje metaprogramování za běhu programu.

2.2.4 Shed Skin

ShedSkin [6] je velice podobný překladači RPython. Využívá podobný algoritmus inference typů a má společný problém, který způsobuje omezený duck typing. Hlavním rozdílem je to, že výsledný kód je v jazyce C++, kde třídy jsou převedeny na třídy. Výstup tedy není převáděn na neobjektový kód.

2.3 Typová inference

Typová inference je mechanismus, kdy jsou konkrétní typy proměnných a atributů odvozovány z jejich použití v kódu. Historie typové inference začíná kolem roku 1958 s vytvořením typového lambda kalkulu. Od té doby vzniklo několik různých algoritmů lišících se ve způsobu zpracování zdrojového kódu, způsobu jakým odvozují typy a jaký je jejich výsledek.

2.3.1 Hindley-Milner

Hindley-Milner [7, 8] je klasický typový systém pro lambda kalkulus poprvé navržený J. R. Hindleym a později znovuobjeven R. Milnerem. Sytém jasně separuje dvě části: popis samotných typů a systém pro dedukci typů z kódu. Popis typů zahrnuje výrazy $x : \sigma$ znamenající že x je typu σ, $\tau \rightarrow \tau'$ značící typ funkce s vstupním typem τ a výstupním typem τ'. Výraz $let\ x = e_1\ in\ e_2$ znamená substituci e_1 za x v e_2. Typy jsou rozděleny do dvou skupin: monotypy a polytypy. Pokud je typ definován pomocí kvantifikátoru, např. jako $\forall \alpha . \sigma$, jedná se o polytyp. Důležitou součástí definice typů je relace $\sigma \sqsubseteq \sigma'$, která znamená, že typ σ je obecnější než σ'. Obecně je možné říct že σ je polytyp, pak σ' může být jeho monotyp.

Deduktivní systém je postaven jako formální systém s typovými pravidly, viz box napravo. Tato pravidla formálně popisují celý systém typového odvození.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \to \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

Pravidlo [Var](variable access) znamená, že pokud se v množině příkazů Γ vyskytuje proměnná x typu σ, pak je možné odvodit, že x je typu σ.

[App](application) umožňuje odvození typu výstupu funkce e_0 pokud je použit jako argument výraz e_1.

[Abs](abstraction). Toto pravidlo říká: pokud z předpokladu, že x je typu τ můžeme odvodit, že e je typu τ', pak můžeme odvodit, že abstrakce e podle x je typu $\tau \to \tau'$. Toto je důležité při odvozování typů funkcí z jejich těla. x je parametr funkce a e je tělo funkce. Pokud z předpokladu, že parametry mají typ τ odvodíme, že výsledek funkce bude τ', pak typ celé funkce je $\tau \to \tau'$.

[Let](variable declaration) pokud můžeme odvodit, že e_0 je typu σ a zároveň z předpokladu, že x je také typu σ můžeme odvodit, že e_1 je typu τ. Pak můžeme odvodit, že výraz vzniklý substitucí e_0 za x do e_1 je typu τ.

Tato čtyři pravidla formálně popisují intuitivní vnímání typového odvození z výrazů vycházejících z lambda kalkulu, jako je vytváření anonymních funkcí, aplikace funkcí nebo substituce výrazů. Není tedy žádným překvapením, že tato pravidla jsou velice podobná pravidlům popisujícím validní výrazy v lambda kalkulu

Pravidla [Inst](instantiation) a [Gen](generalization) reprezentují specializaci a zobecňování typů. První pravidlo říká, že pokud výraz e nabývá typu σ', který je obecnější vůči libovolnému typu σ, pak e pak může nabývat také typu σ. Toto se může vyskytovat, když typ e je polytyp, pak e může také nabývat libovolného monotypu, který je specifičtější verzí daného polytypu. Druhé pravidlo je opačný proces, kdy výraz e, který nabývá jednoduššího typu σ, vytvoříme složitější polytyp kvantifikováním s α v případě, že α není volná v aktuálním kontextu.

Je dokázáno, že tento deduktivní systém vždy odvozuje nejobecnější možný typ[3]. Z tohoto deduktivního systému pak vycházejí algoritmy, které jsou nejčastěji postaveny na principu unifikace. Jedním z prvních byl Robinsonův algoritmus. Algoritmy postavené na tomto systému jsou primárně využívány ve funkcionálních jazycích, které také vycházejí z lambda kalkulu. Nejrozšířenějším reprezentantem je jazyk Hakell a jazyky z rodiny ML. Zásadní výhodou těchto algoritmů je schopen pracovat na úrovni jednotlivých funkcí a tím odvozovat typ definice funkce. Toto je použito například u jazyka Haskell. Během překladu tedy nepotřebuje znát celý zdrojový kód aplikace. Z toho vyplývá také jeho obecně nižší časová a paměťová náročnost Je tedy použitelný i na velké programy s velkým množstvím zdrojového kódu.

[3] Obecností typu se myslí, jak moc typ omezuje možné operace, které s typem lze provádět. Čím větší je omezení, tím konkrétnější je.

Problém je však jeho použití v kontextu jazyka Python. Hindley-Milner je navržen pro funkcionální kód a jeho implementace pro imperativní a objektový jazyk je problematická. Dalším problémem je, že nepodporuje datový polymorfismus, který se v jazyku Python vyskytuje jako dědičnost tříd. Byla navržena rozšíření, která tyto problémy řeší [9], ale ty často velice komplikují a zesložiťují implementaci.

2.3.2 Palsberg a Schwartzbach

Palsberg a Schwartzbach navrhli algoritmus [10] pro jazyk podobající se jazyku SMALLTALK. Algoritmus vytvoří síť omezení, kde každý uzel sítě reprezentuje množinu typů, kterých může nabývat výraz v programu. Na začátku jsou všechny uzly prázdné až na konstanty a literály. Hrany v této síti jsou omezení a reprezentují příkazy programu. Typy se šíří podél těchto hran až do okamžiku, kdy dojde k ustálení množin typů v uzlech. Tento proces simuluje tok dat za běhu programu a tím omezuje možné typy výrazů. Pokud je tedy uzel x propojen hranou s uzlem y, pak nový typ v uzlu x se přidá i do uzlu y. A to samé se provede u uzlů propojených s y. Toto probíhá, dokud nejsou prozkoumány všechny uzly sítě, čímž dojde k jejímu ustálení.

Omezení jsou přidána na základě dvou příkazů ze zdrojového kódu: přiřazení a volání funkcí. Pro přiřazení `x = y` je vytvořeno omezení z uzlu reprezentující výraz `y` do uzlu reprezentující výraz `x`. Toto zajistí, že typy, kterých může nabývat výraz y, jsou podmnožinou typů výrazu x. Omezení pro funkce jsou vytvořena jako hrany mezi výrazy parametrů při volání a formálními argumenty těla funkce. Pokud je tedy volána funkce `f(2, q)`, s definicí `f(x, y)`. Pak algoritmus vytvoří hranu z uzlu konstanty `2` do uzlu argumentu `x` a hranu z uzlu proměnné `q` do uzlu argumentu `y` těla funkce `f`.

Tento algoritmus se však ukázal jako prakticky nepoužitelný, protože přesnost odvozování závisela na duplikaci těl tříd a funkcí a na kompresi hierarchie tříd do ne-hierarchické struktury. To způsobilo obrovské navýšení množství kódu, které bylo nutné analyzovat a zároveň způsobilo, že vznikalo množství kódu, které není ani nutné analyzovat pro získání dostatečné znalosti o použití typů.

2.3.3 Cartesian Product Algoritmus

Tento algoritmus poprvé zveřejnil Ole Agesen v roce 1995 [11] a jeho primární inspirací je právě práce Palsberga a Schwartzbacha. Zásadním rozdílem tohoto algoritmu oproti původnímu je způsob, jakým se zpracovává volání funkcí s odlišnými argumenty. Základní argument popsaný výše sjednocuje typy z různých volání funkce a tím snižuje přesnost odvozených typů. Cartesian Product Algoritmus tento problém řeší vytvořením kopií těla funkce zvané šablony. V místě volání funkce se pak vytvoří kartézský součin množin typů jednotlivých argumentů, čímž vznikne množina seznamů, které obsahují pouze jednoduché typy. Každý tento seznam se pak propojí s jednou šablonou volané funkce a výstupní hodnoty těchto šablon jsou pak propojeny s výsledkem volání této funkce.

Na rozdíl od algoritmů založených na Hindley-Milner systému ale není možné, aby tento algoritmus pracoval pouze na kousku zdrojového kódu. Vždy je nutné znát celý zdrojový kód v době překladu, protože celý algoritmus pracuje „shora dolů“, tedy od počátku programu přes jednodušší funkce až po ty atomické. To má negativní dopad na rychlost a množství využité paměti, která může způsobit problémy při překladu větších programů. Je ale dostatečně efektivní, aby byl použitelný. Zásadní výhodou tohoto algoritmu je, že odvozuje nejkonkrétnější typy. Ty je pak možné přímočaře převést do výsledného generovaného statického kódu.

Tento algoritmus je použit u většiny překladačů jazyka Python, které provádějí odvozování typů. Důvodem je, že algoritmus byl původně navržen pro objektový jazyk a jeho použití je jednodušší než v případě algoritmů, které jsou navrženy pro funkcionální jazyky. Druhým důvodem je fakt, že algoritmus odvozuje nejkonkrétnější typy, díky čemuž je převod do cílového jazyka, který nepodporuje polymorfní typy, přímočařejší.

3 Návrh typového systému

Hlavním problémem implementace překladače dynamického jazyka, jako je Python, je výběr podporované podmnožiny a s tím spojený návrh typového systému. S tím také úzce souvisí i inferenční algoritmus, který s tímto typovým systémem musí být kompatibilní.

V této kapitole se budu zabývat výběrem podmnožiny jazyka. A to takové, aby byla použitelná jako programovací nebo skriptovací jazyk a aby zároveň nad ní bylo možné provádět typovou inferenci.

3.1 Výběr podmnožiny jazyka

Při výběru podmnožiny jazyka jsem se zaměřil primárně na jednoduchou implementaci jak typového systému a inferenci, ale i při překladu samotném.

3.1.1 Základní typy

Jazyk Python poskytuje několik základních typů, které vznikají zapsáním konstanty nebo použitím konverzní funkce. Ty, které jsem vybral do implementace:

- **int** – celá, znaménková čísla
- **float** – desetinná čísla
- **string** – Znaky a neměnné řetězce znaků.
- **boolean** – Booleovské hodnoty.
- **list** – Seznam hodnot. V Pythonu může jeden seznam uchovávat hodnoty rozdílných typů, to by však komplikovalo typovou inferenci. List je tedy omezen na uchovávání pouze jednoho typu hodnot. Jedná se tedy o homogenní strukturu.
- **tuple** – Primárně používané jako seznam heterogenních hodnot. I přesto, že v jazyce Python se jeho použití kryje se seznamem, je jeho význam ve statickém typování mnohem bližší heterogenní struktuře.

Základní typy, které nebudou implementovány:

- **dictionary** – Seznam dvojic klíč-hodnota. Je možno realizovat podporovanými funkcemi.
- **complex** – Struktura pro komplexní čísla. Málo používaná mimo matematické aplikace.
- **bytes, bytearray** – Struktury pro uchovávání bloku bytů. Jejich použití je také dost specifické a nejsou nutné pro základní použití.
- **set, frozenset** – Matematické množiny.

3.1.2 Výrazy

Jednou ze dvou částí kódu jazyka Python jsou výrazy (expression). Ty se skládají z konstantních hodnot, čtení proměnných, volání funkcí, aritmetických, logických a porovnávacích operátorů, indexování a list slices, lambda výrazů a list-comprehension [12] výrazů. Výsledkem výrazu je hodnota, kterou lze uložit do proměnné, atributu instance třídy nebo položky v seznamu, tuple nebo dictionary. Výrazy se vyskytují jako součásti příkazů, kdy různé příkazy mohou mít více výrazů jako jejich parametry a jak se s výsledkem výrazu pracuje dále, záleží na konkrétním příkazu. Výrazy taky mohou být na místě příkazu samotného, ale pak se výsledná hodnota výrazu zahazuje. Toto je nejčastěji při volání funkce, od které neočekáváme, že vrací data.

V rámci překladače jsou tedy implementovány části, které umožní dostatečný rozsah výrazů. Mezi tyto části patří:

- **Výrazy konstantních hodnot**
- **Aritmetické operace** – Tyto operace jsou implementovány pouze pro číselné typy. Sčítání je implementováno jako konkatenace řetězců a seznamů.
- **Porovnávací operace(==, <, >, in)** – Operace, jejichž výsledkem je booleovská hodnota. Test na rovnost a nerovnost jsou implementovány pro všechny typy. Operace menší než a větší než jsou implementovány pouze pro číselné typy a řetězce.
- **Logické operace** – Operace booleovské logiky.
- **Čtení proměnných, indexů a atributů tříd**
- **Volání funkcí a metod tříd** – Počítá se pouze se základním voláním. Není zahrnut např. jmenované nebo nepovinné parametry.
- **Výrazy pro vytvoření seznamu a tuple**

Výrazy, které nejsou implementovány:

- **List slices** - jsou nahraditelné implementovanými vlastnostmi
- **List comprehensions** – velice složité na implementaci a jsou nahraditelné implementovanými vlastnostmi
- **Implicitní konverze na bool** - V jazyce Python lze jakýkoliv výraz implicitně převést na booleovskou hodnotu. Toto není podporováno. Pořád je ale možné použít explicitní *bool()*

3.1.3 Příkazy

Každý blok programu v jazyce Python se skládá ze sekvence příkazů. Příkazy samotné jsou komplexní a skládají se z bloků jiných příkazů a výrazů.

Přiřazení hodnoty pomocí = do proměnné, atributu třídy nebo indexu v seznamu je nejzákladnější operace a je implementována v celé své šíři. Tedy i „rozšířené“ operátory jako +=, -=, etc..

Větvení pomocí příkazu **if-then-else** je základní prvek pro řízení průběhu programu. Díky neexistenci podpory pro implicitní konverzi na boolean je však použitelnost mírně omezena.

Cyklus **for** je také základní, ale komplexní příkaz. V jazyce Python neexistuje cyklus od-do-krok, ale každý **for** cyklus operuje nad iterátorem. Jedná se tedy o obecnou variantu a pro specifickou indexovou variantu se používá funkce *range(from, to, step)*, která vytváří takovýto iterátor. Kvůli optimalizaci se tedy implementace **for** cyklu rozdělila na dvě části: první je varianta, kdy v argumentu je použita pouze funkce *range*. Druhá varianta je verze, kdy se jako parametr vyskytuje obecný výraz. V tom případě se prochází tímto výrazem jako seznamem.

Cyklus **while** má podobné omezení na svou podmínku jako je **if-then-else**. Výraz v podmínce tedy vždy musí nabývat booleovské hodnoty.

Spolu s cykly **for** a **while** jsou také implementovány příkazy **break** a **continue**. Zajímavostí v syntaxi jazyka Python je použití příkazu **else** u cyklů. Ten se vykoná, když cykly doběhnou do konce a nejsou ukončeny příkazem **break**. Tato funkce však není v navrhovaném překladači podporována, protože se jedná spíše o syntaktický cukr a je možné ji vyjádřit již implementovanými prostředky.

Příkaz **with** není podporován, protože je spojen primárně s kontrolou zdrojů, výjimkami a speciálním chováním tříd. Tyto funkce nejsou pro použití požadovány.

Výjimky s příkazy **try** a **raise** nejsou v návrhu. Důvodem je komplexnost jejich implementace a možné problémy s převodem do cílového jazyka.

Posledním implementovaným příkazem je mazání hodnot ze seznamů a mazání lokálních proměnných pomocí příkazu **del**. V překladači je implementován pouze pro mazání hodnot ze seznamů podle jejich indexu.

3.1.4 Funkce a třídy

Specifikace jazyka Python definuje funkce jako možnost strukturování kódu. Podpora definice nových funkcí a jejich volání v navrhovaném překladači je zřejmá. Každá funkce je definovaná svým jménem a signaturou, což je seznam typů, které funkce přijímá jako argumenty a typ hodnoty kterou vrací. Otázkou jsou detaily týkající se funkcionálního chování jazyka Python. To se opírá hlavně o možnost ukládat ukazatele na funkce do proměnných a atributů a definice lokálních funkcí s uzávěry. Ani jedna z těchto vlastností není v návrhu překladače zohledněna. Návrh však možnost implementaci těchto funkci neznemožňuje.

Další otázkou je rekurze funkcí. Tu lze rozdělit na rekurzi jedné funkce a komplexní rekurzi více funkcí, které se volají navzájem. V návrhu je zahrnuta podpora pouze jednoduché rekurze s tím, že překladač by měl jít rozšířit i o komplexnější rekurzi.

Třídy jsou způsob, jak může programátor definovat vlastní datové typy a také zajistit pozdní vazbu a rozšiřitelnost pomocí jejich dědičnosti. Hlavním problémem pro překlad je dynamický přístup k atributům tříd. Jakákoliv část kódu může přidat nový atribut nebo změnit typ existujícího atributu. Toto výrazně komplikuje celý proces typové inference, a proto je v navrhovaném překladači zavedeno omezení, že všechny atributy musí být definovány v metodách třídy samotné. A to nejlépe včetně typů těchto atributů. Pokud není možno inferovat typ atributu v rámci definice třídy, je nutné, aby tyto typy byly odvoditelné z použití instance této třídy a konstantní po dobu jejího použití. V rámci této implementace budou podporovány pouze instanční atributy a nikoliv statické.

Součástí tříd jsou také jejich metody, tedy funkce napojené na instance dané třídy. Tyto metody se rozeznají tak, že jejich první parametr je podle konvence pojmenován *self*. V rámci metod se pracuje s atributy třídy, díky čemuž lze odvozovat typy těchto atributů. Omezení na metody jsou stejná jako na jednoduché funkce. K tomu je navíc omezení, že metoda nemůže být „generická“, tedy musí mít všechny typy definovány nebo jsou generické v rámci celé třídy. Toto je omezeno způsobem, jakým se generuje výsledný kód. Python také podporuje speciální metody, které jsou volány jazykem samotným, tyto metody vždy začínají a končí dvěma podtržítky. V rámci implementovaného překladače je možné použít pouze metodu *__init__*, která je obdobná konstruktoru v C++.

Další vlastností tříd je jejich dědičnost. Python podporuje jak jednoduchou tak vícenásobnou dědičnost. V navrhovaném překladači se však s dědičností nepočítá. Důvodem je složitost její implementace a navýšení komplexnosti jak inference tříd samotných, tak i jejich použití.

Třídy také mohou definovat vlastní operátory a operace, které nejsou volány jako metody. Překladač s touto vlastností nepočítá a nepovoluje ji.

3.1.5 Knihovní funkce a třídy

Python poskytuje taky rozsáhlou základní knihovnu funkcí a tříd, které pak programátor může využít. Aby byl překladač použitelný, musí být i část této knihovny implementována. Omezení na rozsah této

knihovny nezávisí moc na typovém systému, ale spíše na tom, co je implementovatelné v jazyce C++ jako součást základní knihovny překladače.

V překladači se tedy počítá s těmito základními funkcemi a třídami:

- Funkce **print** a **input** pro vstup a výstup do konzole
- Funkce **str**, **int**, **float**, **bool**, **ord** a **chr** pro konverzi základních typů
- Funkce **open** a třída **File** s metodami pro základní čtení a zápis do souborů

3.2 Definice typů

Typy v navrhovaném překladači se dají rozdělit na dvě základní varianty: konkrétní typy a generické typy. Obě varianty jsou však samy o sobě komplexní struktury a i mezi sebou mohou mít vazby. Typ pak může být složen z více různých konkrétních a generických typů propojených parametry konkrétních typů nebo omezeními generických typů. Tato propojení pak způsobí, že typ může mít podobu grafu. Obecná struktura typů a jejich vztahů je v Diagram 1 - Obecný popis struktury typů (str. 47)

3.2.1 Konkrétní typy

Konkrétní typ reprezentuje nějaký základní typ nebo třídu. Tyto typy definují konkrétní vlastnosti a schopnosti daného typu a ve výsledném kódu je možné je generovat pomocí jejich jména. Jednoduché konkrétní typy jsou **int**, **float**, **string** a **boolean**. Konkrétní typy ale mohou být parametrizovány jiným typem a tím říkáme, že tento konkrétní typ je parametrický. To proto, že tyto typy vyžadují dodatečnou informaci o tom, s jakými jinými typy jsou schopny pracovat. V rámci kódu pak může být jeden konkrétní parametrický typ vícekrát, pokaždé s jiným parametrem. Nejjednodušší ukázkou parametrického typu je základní typ **list**. U něj musí být typový parametr, který určuje, jaký typ položek se v seznamu nachází. Parametrů může být více než jeden a mohou nabývat jakéhokoliv typu, tedy jak konkrétní, tak i generický. Parametrické typy jsou blízké šablonám v C++ nebo generickým typům v jazycích Java nebo C#. Složitějšími konkrétními typy jsou **list**, **tuple** a **function**. **List** má jeden parametr. **Tuple** má 0 až n parametrů v závislosti na tom jakého řádu je. **Function** má 0 až n parametrů, která reprezentují parametry funkce a návratovou hodnotu funkce.

3.2.2 Generické typy

Generický typ reprezentuje neznámý konkrétní typ. Nereprezentuje tedy nějaký konkrétní typ, ale definuje, jaké vlastnosti se od tohoto typu očekávají. Generické typy jsou použity primárně ve dvou kontextech. První jsou situace, kdy nejsme přímo schopni zjistit konkrétní typ hodnoty, ale je možné jej zjistit z jejího použití. Toto je např. u argumentů funkcí. Druhá situace nastane, když není možné konkrétní typ odvodit v rámci funkce nebo třídy. V tom okamžiku se generický typ vyskytne jako typový parametr konkrétního typu funkce. Při použití funkce jsou pak generické typy nahrazeny za konkrétní tak, aby typování bylo korektní.

Součástí generických typů jsou generická omezení. Ta jsou podobná typovým třídám v jazyce Haskell [13]. Tato omezení definují, jaké vlastnosti se od konkrétního typu očekávají. Příkladem takovéhoto omezení je např. existence operátoru sčítání u typu nebo existence metody s konkrétní signaturou. Omezení mohou tedy být parametrizovány dalšími typy. Omezení se pak přidávají při

použití typu. Všechna možná omezení jsou součástí implementace překladače a uživatel nemůže nijak definovat vlastní. Mezi implementovaná omezení patří:

- Existence operace sčítání
- Je typ iteratovatelný (např. **list** nebo **string**)
- Existence metody s konkrétní signaturou
- Je možné číst index typu (**list**, **string** nebo **tuple**)
- Typ je referenční (pouze třídy jsou referenční typy)
- Existence atributu s daným jménem a typem

3.2.3 Kompozice typů

Ve výsledku mezi jednotlivými typy mohou vznikat složité vazby a to díky možnosti jejich parametrizace. Hlavně definice funkcí a tříd mohou být grafy skládající se z několika provázaných typů, které mohou být jak konkrétní, tak generické.

Příkladem takovéhoto typu je typ, který je odvozen z funkce:

```
def func(param):
    return param[0]+param[1]
```

Typ této funkce ze zobrazen graficky v Diagram 2 - Popis typu konkrétní funkce (str. 47)

3.3 Unifikace typů

Unifikace je proces, při kterém se dva rozdílné typy nahradí typem novým, vůči kterému lze oba typy substituovat. Nahrazení probíhá ve všech místech, kde jsou původní typy použity. Tedy jak v případě proměnných, tak u typů, které se vyskytují jako parametry jiných typů.

3.3.1 Substituovatelný typ

Nalezení substituovatelného typu pak závisí na variantách vstupních typů. Pro dvě varianty jsou možné 3 kombinace: konkrétní x konkrétní, konkrétní x generický, generický x generický. V případě že není možné nalézt takto substituovatelný typ, je to příznak toho, že se někde nachází typová chyba a překlad je ukončen.

Unifikace dvou generických typů vytvoří nový generický typ, který bude kombinovat omezení obou vstupních typů. Problémem je nutnost unifikace parametrů těchto omezení v případě, že se jedná o stejné omezení. Například pokud vstupní typy vyžadují metodu s jedním jménem ale různou signaturou, tak se algoritmus pokusí tyto signatury unifikovat.

Unifikace konkrétního a generického typu má význam kontroly, zda lze konkrétní typ dosadit na místo generického. Konkrétní typ tedy musí mít vlastnosti, které jsou definovány omezeními generického typu. Pokud konkrétní typ tato omezení nesplňuje, pak se jedná o chybu unifikace. Při unifikaci může také dojít ke změně parametrů konkrétního typu právě kvůli unifikaci s parametry omezení generického typu.

Tyto dva případy unifikace interagují způsobem podobným asociativní operaci. Příkladem jsou dvě situace, kde máme tři typy A, B a C. A a B jsou generické typy a C je typ konkrétní. První situace nastane, pokud unifikujeme typy A a B a pak jeden z těchto typů unifikujeme s typem C. Druhá je situace, kdy unifikujeme typ A s typem C a typ B s typem C. V obou situacích doje k tomu, že se typ

C kontroluje vůči omezením definovanými typy A a B. V první situaci ale nejdříve dojde k tomu, že se omezení typů A a B sjednotí do jednoho typu a ten je pak unifikován s typem C. Kdežto v druhé situaci se kontroluje zda typ C odpovídá omezením typu A a pak zda odpovídá omezením typu B. V obou situacích však dojde k vytvoření jednoho typu unifikací A, B a C.

Unifikace dvou konkrétních typů je složena ze dvou částí. První je nalezení možného typu, který je obecnější k oběma vstupním typům. Tady vstupuje především dědičnost a objektový model. Druhou částí je unifikace parametrů u parametrických typů. Celý algoritmus se zesložiťuje, protože je možné, že v rámci hierarchie objektů mohou typové parametry přibývat nebo ubývat.

Celá unifikace je obecně rekurzivní proces, protože dochází k unifikaci parametrů typů nebo parametrů generických omezení. Unifikace končí v případě, kdy se snažíme unifikovat již unifikované typy nebo jestliže dojde k chybě unifikace.

3.3.2 Tranzitivita unifikace

Důležitou vlastností unifikace je její tranzitivní chování při nahrazení výskytu typů. Při unifikaci dvou typů se musí nahradit všechny jejich výskyty a to jak v rámci kódu, tak jako parametry jiných typů. Následující ukázka pseudokódu demonstruje tento problém:

```
1. a = list<T0>()
2. b = a
3. a.push(A())
```

1. Proměnná `a` obsahuje seznam, který je parametrizován generickým typem `T0`.
2. Do proměnné `b` se přiřadí reference na a. `a` a `b` mají stejný typ `list<T0>`.
3. Do seznamu je vložená nová položka typu `A`. Přes volanou metodu se unifikují typy `A` a `T0`. Výsledný typ proměnné `a` je `list<A>`, ale tento typ musí mít i proměnná `b` i přesto, že v tomto příkazu nijak nefiguruje. Důvod je ten, že `a` a `b` mají stejný typ a při nahrazení typu u `a` se musí typ nahradit i u proměnné `b`.

Algoritmus si tedy musí pamatovat všechna místa, kde je typ použit a při unifikaci tyto výskyty nahradit. A díky tomu, že tyto výskyty mohou být jako parametry u jiných typů, tak vzniká graf typů a jejich výskytů. V kapitole o implementaci je popsáno, jak je toto implementováno detailněji.

3.4 Inference typů z konstrukcí jazyka

Inference typů ze zdrojového kódu je přímočará a jednoprůchodová. Algoritmus postupně čte příkazy v bloku a postupně konkretizuje typy výrazů a proměnných použitím unifikace. Inference je rozdělena na inferenci výrazů a inferenci příkazů. Inferencí výrazu je typ, jakého tento výraz nabývá. Inferencí příkazu je primárně inference typů pro vnitřní logiku příkazu samotného. Nejčastěji inference probíhá tak, že se vytvoří nové typy, které reprezentují typy, s jakými příkaz pracuje, a tyto typy se pak unifikují s typy získanými z parametrů příkazu. Díky tranzitivitě unifikačního algoritmu se tyto typy postupně konkretizují a pak je lze využít pro generování zdrojového kódu.

3.4.1 Výrazy

Následující seznam popisuje, jak se odvozují typy pro jednotlivé konstrukce výrazů:

- **Konstanty** – Vytvoří typ podle typu konstanty.

- **Nová instance seznamu** – V případě, že výraz nemá iniciální výrazy, se vytvoří typ list s generickým typem jako jeho parametrem. Pokud má iniciální výrazy, jsou typy těchto hodnot unifikovány a výsledný typ se nastaví jako parametr typu list.
- **Nová instance tuple** – Vytvoří nový typ tuple s parametry získanými z typů iniciálních výrazů.
- **Aritmetické operátory** – Vytvoří generický typ s omezením požadovaného operátoru a ten pak unifikuje s typem pravého a levého výrazu.
- **Booleovské operátory** – Vytvoří booleovský typ a unifikuje s typy pravého a levého výrazu.
- **Porovnávací operátory** – Unifikuje typ levého a pravého výrazu. Typem výrazu je booleovský typ.
- **Čtení proměnné** – Získá typ proměnné z lokálního kontextu a vrátí jej. Pokud proměnná neexistuje, jedná se o chybu.
- **Čtení indexu** – Vytvoří dva generické typy. První přiřadí druhému jako součást omezení na existenci indexu. Omezení se také nastaví, zda je index konstantní hodnota. Typ výrazu indexu se unifikuje s typem int. Druhý generický typ unifikuje s výrazem, nad kterým se index nachází. Typem výrazu je druhý vytvořený generický typ.
- **Čtení atributu** – Vytvoří dva generické typy. První přiřadí druhému jako omezení na existenci atributu včetně jména tributu. Typ vnitřního výrazu unifikuje s druhým generickým typem. Typ výrazu je první generický typ.
- **Volání funkce** – Získá typ definice funkce podle jejího jména. Unifikuje parametry definice s typy výrazů parametrů volání. Vrací typ, který má definice jako návratový.
- **Volání metody** – Vytvoří generický typ s omezením na existenci metody s typy, které jsou získány z výrazů parametrů volání. Vytvoří druhý generický typ, který se nastaví tomuto omezení jako návratový typ. Typem tohoto výrazu je druhý vytvořený typ.

Jedním z problémů odvozování typů výrazů jsou číselné typy. Tedy interakce celočíselných hodnot a hodnot s plovoucí desetinnou čárkou. V jazyce Python jsou tyto hodnoty reprezentovány typy `int` a `float`. V jazyce Python je implicitní konverze typu `int` na typ `float`, naopak však toto neplatí, protože by mohlo dojít ke ztrátě přesnosti. Zároveň však existují operace, které vyžadují, aby byl použit celočíselný typ. Příkladem je přístup k indexu v poli nebo řetězci, kde hodnota indexu musí být celočíselná. Překladač tedy musí zaručit, aby bylo možné jednoduše kombinovat typy `int` a `float` ve výrazech, ale zároveň aby byl schopen zabezpečit typovou korektnost v případě, kde je vyžadován konkrétně typ `int`. K tomu je v typovém systému přidán nový typ: `numeric`. Typy `int` a `float` pak nabývají významu „typ musí být celočíselný“ a „typ musí desetinný“ a typ `numeric` má význam „je jedno jestli je typ desetinný nebo celočíselný“. Unifikace mezi těmito typy je pak dána tak, že typ `int` a `numeric` se unifikují na `int`, `float` a `numeric` se unifikují na `float` a `float` a `int` nelze unifikovat. Typ `numeric` pak nahrazuje typ `int` tam, kde není vyžadováno, aby byla hodnota celočíselná, jako je například výraz konstanty celého čísla. A typ `int` zůstává tam, kde je vyžadována celočíselná hodnota, například tedy u výše zmíněného indexu. V generovaném kódu je však typ `numeric` reprezentován celočíselným typem, protože pokud nedošlo k unifikaci s typem `float`, je celočíselný typ dostačující.

Další vlastností, kterou je nutné řešit speciálně je konstanta None. Ta má význam „žádná hodnota“, tedy podobně jako má C++, Java nebo C# hodnotu NULL. Problémem je, že tuto konstantu

lze přiřadit pouze referenčnímu typu a nikoliv hodnotovému typu. Zároveň není možné z této konstanty samotné určit, jakého konkrétního typu musí hodnota nabývat. Typ této konstanty je tedy generický typ s omezením, aby konkrétní typ byl referenční typ. V případě, že se v kódu vyskytuje přiřazení hodnotového typu a konstanty None do jedné proměnné, dojde k chybě typování, protože None lze přiřadit pouze do referenčního typu. Toto by bylo možné rozšířit přidáním `nullable` rozšíření pro hodnotové typy. Toto rozšíření by se podobalo Nullable typu v jazyce C#. Problémem tohoto rozšíření by byla komplexita provádění matematických operací nad těmito typy, protože každá taková proměnná může navíc obsahovat konstantu „nedefinováno“, se kterou se v matematických operacích musí počítat.

3.4.2 Přiřazení

Přiřazení je základní příkaz. Lze rozlišit tři základní varianty přiřazení: přiřazení do proměnné, přiřazení do indexu třídy a přiřazení do atributu třídy. Ve všech třech případech se nejdříve zjistí typ výrazu, který se přiřazuje. V případě přiřazení do proměnné se nejdříve zjistí, zda proměnná s daným jménem existuje. Pokud neexistuje, je vytvořena s typem přiřazovaného výrazu. Pokud existuje, je její typ unifikován s typem výrazu. V případě indexu a atributu se v obou případech zjistí typ podvýrazu, nad kterým se index nebo atribut volá. Pro index se pak tento typ unifikuje s typem list, který má jako parametr nastavený typ přiřazovaného výrazu. Pro atribut se tento typ unifikuje s generickým typem, který má omezení na existenci atributu, jehož typ je typ výrazu.

Variantou přiřazení je rozšířené přiřazení (eg. +=, -=, *=, aj..). Tedy přiřazení kombinující čtení, binární operátor, a přiřazení. V případě překladače lze toto jednoduše rozložit na jednotlivé elementárnější příkazy a výrazy a provádět typovou inferenci nad takto zjednodušenou strukturou.

3.4.3 Řízení toku kódu

Mezi příkazy pro řízení toku kódu patří **if-then-else**, **while** a **for-range** a **for-expr**. Jejich inference je přímočará. V případě if-then-else a while se výraz v podmínce unifikuje s typem **bool**. V případě příkazu **for** existují dvě varianty. První, která iteruje přes iterátor vrácený funkcí *range()* a druhý iterující přes obecný výraz. Pro verzi s *range* je inference jednoduchá. Parametry funkce *range* a typ iterované hodnoty se unifikují s typem **int**. Pro **for** nad výrazem se vytvoří typ list s generickým parametrem. List se pak unifikuje s typem výrazu a generický typ se unifikuje s typem iterované hodnoty. V obou případech se také musí brát v potaz, že vnitřní hodnota cyklu je přístupná i mimo tento cyklus a je tedy nutné jí brát jako normální proměnnou. V případě její předchozí existence je tedy nutné typ této hodnoty unifikovat s typem předem inferovaným.

V případě všech těchto příkazů se pak provede inference typů nad všemi příkazy uvnitř bloků, které k těmto příkazům patří.

3.4.4 Řetěz přístupu ke struktuře

Řetěz přístupu ke struktuře vzniká, když se přistupuje k lokální proměnné, atributu složitějšího typu nebo přístupu k položce v seznamu. Příkladem takového řetězu je výraz `var.data[10].value`, který se skládá z přístupu k lokální proměnné `var`, jejímu atributu `data`, položce v seznamu s indexem `10` a atributu `value` této položky.

Tento přístup může sloužit buď ke čtení, k zápisu nebo volání metody. Díky unifikačnímu algoritmu je však možné tyto rozdílné operace provádět stejným způsobem. Kořen tohoto řetězu je

vždy lokální proměnná. U ní však nemusíme vědět konkrétní typ a typ tedy bude generický. Řetěz se skládá z různých prvků: přístup k proměnné, přístup k atributu třídy a přístup k indexu třídy. Každý z těchto prvků vrací typ, který tento prvek reprezentuje a všechny kromě kořene mají i svého potomka.

Řetěz má vždy kořen, což je přístup k proměnné. Pokud proměnná neexistuje, je vytvořen nový generický typ a vrácen. Pokud proměnná už existuje, je vrácen její typ. Pokud je v řetězu pouze kořen, jedná se o přímý přístup do proměnné.

Přístup k atributu staví na omezení generického typu, které definuje, že daný typ má atribut se specifickým jménem a typem. Je vytvořen nový generický typ, který bude reprezentovat návratovou hodnotu atributu. Druhý generický typ je pak vytvořen jako reprezentant typu, který bude mít tento atribut a má tedy omezení na existenci atributu s požadovaným jménem a typem prvního generického typu. Druhý generický typ je pak unifikován s typem, který vrátí potomek v řetězu. Tento prvek pak vátí první generický typ.

Přístup k indexu třídy je podobný. Nejdříve se unifikuje typ výrazu indexu s typem **int**, aby se zaručilo, že parametr bude celočíselný. Pak se vytvoří generický typ, který reprezentuje typ hodnot uložených v seznamu. Tímto typem se pak parametrizuje nový typ list, se kterým se unifikuje typ potomka. Typem tohoto výrazu je typ reprezentující hodnoty.

3.4.5 Definice funkce

Inference typů z funkce zahrnuje inferenci typů lokálních proměnných a typu signatury funkce. Výhodou jazyka Python jsou jednoduchá pravidla pro rozsah platnosti proměnných. Proměnné jsou platné v celém těle funkce a není možné tento rozsah omezit, jako je tomu například v jazyku C++. Pokud se tedy vytvoří nová proměnná v bloku příkazu **if**, pak je tato proměnná viditelná i po tomto bloku. Pro udržení proměnných a jejich typů v rozsahu funkce je vytvořena instance kontextu, který uchovává tuto informaci.

Inference signatury funkce zahrnuje inferenci typů parametrů funkce. Ty jsou inicializovány jako proměnné s generickými typy. Při jejich výskytu ve výrazech v těle funkce pak dochází k jejich unifikaci a konkretizaci. Typ návratové hodnoty funkce se registruje jako speciální hodnota na kontextu těla funkce. Při prvním příkazu **return** je nastaven typ výrazu jako návratový typ. Při každém dalším příkazu **return** se pak unifikuje typ vráceného výrazu a předchozí typ.

V případě, že na konci odvozování typů pro celou funkci dojde k tomu, že některé proměnné jsou generické typy, tak jsou tyto typy nastaveny jako generické parametry funkce a jejich konkrétní typy jsou pak odvozovány z použití této funkce. To ale pouze v případě, že tyto generické typy se vyskytují v některém z parametrů nebo v návratovém typu. V opačném případě není možné odvodit konkrétní typy a jedná se o chybu. Toto ale neplatí pro hlavní metodu, kde musí být známy všechny konkrétní typy. V případě že nejsou všechny typy konkrétní, pak program neobsahuje dostatek informací pro odvození staticky typovaného programu a vzniká chyba inference typů.

Následující příklad popisuje možnou situaci, kde toto může nastat**:** Mějme třídy `A` a `B` takové, že `B` dědí z `A` a `B` je parametrizovaná typem `T`. Třída `B` je tedy v typové inferenci reprezentovaná typem `B`. Pak mějme funkci, která může vrátit buď `A` nebo `B`.

```
def function(value):
     if value:
          return A() # typ A
```

```
else:
    return B() # typ B<T>
```

Návratový typ se pak musí unifikovat z typů `A` a `B<T>`, a výsledný návratový typ pak je `A`.
Problémem ale je, že typ `T`, který je potřebný pro vytvoření instance třídy `B` není možné dále nijak konkretizovat, protože typ `A`, který je návratovou hodnotou funkce o tomto typu nic neví. Tato situace se musí detekovat a je zahlášena chyba.
Druhým případem, kdy podobná situace může nastat je tato funkce:

```
def func():
    lst = []
```

I přesto, že se jedná o validní kód v jazyce Python, z hlediska typů nedává moc smysl. Problémem je, že se vytváří nový seznam, ale dále se s ním nepracuje a není tedy možno zjistit, jaký typ hodnot bude v seznamu uložen.

Výsledkem odvození typu funkce je typ **function** reprezentující celou funkci včetně jejich parametrů a její návratové hodnoty. Mezi jednotlivými parametry pak mohou být vazby jako v jediném konkrétním typu. Různé parametry mohou mít stejný typ, nebo mohou být součástí parametrů nebo generických omezení. Tento komplexní typ je pak přiřazen ke jménu funkce do seznamu funkcí.

Volání funkce nejdříve načte typ funkce ze seznamu aktuálně dostupných funkcí. Pokud funkce s daným jménem neexistuje, je vyvolána chyba Typ funkce obsahuje parametry, které jsou unifikovány s typy výrazů parametrů volání. Typem výrazu tohoto volání je návratový typ, který je parametrem typu funkce. Všechny tyto typy jsou pak uchovávány ve výrazu volání funkce pro pozdější použití v generování kódu.

Vlastností funkcí je také jejich rekurze. Detekce jednoduché rekurze je jednoduchá, protože víme, jakou funkci zrovna odvozujeme a víme, jaká funkce se volá. Pokud se tyto funkce rovnají, vzniká rekurze. Zjištění typů funkce pak je jednoduchá unifikace výrazů parametrů ve volání funkce a proměnných parametrů. Výsledkem výrazu volání této funkce je pak návratová hodnota zpracovávané funkce. Unifikační algoritmus pak propojí typy parametru funkce a její návratové hodnoty. Vzájemná rekurze více funkcí by byla možná v případě, že by bylo možné zajistit, že tyto funkce se odvozují v rámci jednoho kontextu. V tom případě by se unifikovaly typy signatur funkcí a jejich volání. Podobný systém je využit u odvození vzájemné rekurze metod u tříd.

S typem funkcí je spojen také typ `void`. Ten se může vyskytnout pouze v návratové hodnotě funkce a reprezentuje „žádnou hodnotu“. Výhodou toho to typu je, že v unifikačním algoritmu pak lze detekovat situaci, kdy se v kódu vyskytuje situace, kdy se provádí operace s návratovou hodnotou funkce, která nemá návratovou hodnotu. Další výhodnou vlastností tohoto typu je, že v C++ lze typ void přiřadit parametru šablony a výsledný kód je validní, pokud je substituce tohoto typu za šablonu funkce validní. Tento typ ale přináší komplikace, protože je možné tento typ přiřadit proměnné nebo parametru funkce, což ale způsobí nevalidní kód. Proto existuje generické omezení, které znemožňuje unifikaci takto omezeného generického typu s typem `void`. Toto omezení je iniciálně nastaveno všem generickým typům, ale je odebráno u případů, kde tato unifikace nezpůsobí nevalidní kód, jako například návratové hodnoty funkcí.

3.4.6 Definice třídy

Cílem odvození definice typu třídy je získání typové informace pro atributy a metody spolu s jejich jmény. Díky omezení na nemožnost změny struktury mimo definici třídy je toto odvození omezené na kód v metodách třídy. Zde je využito vlastnosti jazyka Python, který v metodách třídy vyžaduje explicitně definovat vstupní parametr, který bude reprezentovat aktuální instanci nad kterou je metoda volána. Tento parametr musí být první a musí se jmenovat **self**. Díky tomu je typová inference jednoduše schopna určit, které atributy a funkce patří instanci třídy. Odvozování atributů je tedy nutné zakomponovat do algoritmu odvozování typů proměnných, aby šlo používat např. řetěz přístupu ke struktuře. Toho lze dosáhnout speciálním typem, který bude přiřazen parametru **self**. Tento typ reprezentuje aktuální třídu a unifikuje pouze s generickými typy. Při unifikaci tohoto typu se pak omezení těchto generických typů na existenci atributů a metod přidávají nebo unifikují jako aktuální atributy a metody.

Podobně jako u funkcí může nastat situace, kdy algoritmus odvodí z použití nejkonkrétnější typ jako generický. V tomto případě se postupuje stejně jako u funkcí. Nejdříve je nutné zajistit, že každý takovýto typ je možné odvodit z použití třídy. Typ se tedy musí vyskytovat jako atribut, parametr, nebo návratová hodnota metody. Pokud tomu tak není, jedná se o chybu.

Výsledkem odvození definice třídy je vznik nového konkrétního typu s definovanými atributy a metodami. Tento typ může být parametrický a každý parametr reprezentuje jeden specifický generický typ. Definice třídy se pak přidá do globálního kontextu funkce, která má parametry ekvivalentní konstruktoru třídy a jejíž návratová hodnota je typ třídy.

3.4.7 Živost proměnných

Dynamický typový systém je použitelný i v jiných případech, než je duck typing. Jedním z nich je používání proměnných se stejným identifikátorem ale s jiným typem v rámci jedné metody. Toto je vhodné hlavně v rámci skriptů, které se nesnaží o vysokou kvalitu kódu. Může tedy nastat situace, kdy se algoritmus snaží unifikovat typy v případech, kdy to není nutné. Jednou možností jak tento problém vyřešit je aplikace algoritmů pro výpočet živosti proměnných [14]. Výstupem tohoto algoritmu je informace ve kterých místech se přiřazením do proměnné v každém případě ztratí původní data. V těchto místech je pak překladač schopen další následující výskyt proměnné nahradit novou proměnnou a význam programu zůstane identický.

Tento algoritmus ale nenahrazuje unifikaci. V kódu pořád můžou nastat situace, kdy je unifikace nutná. Např. když se do proměnné přiřazuje v jedné větvi **if** příkazu. V tomto případě se po tomto příkazu může vyskytovat jak původní hodnota nebo nová hodnota. Musí tedy dojít k unifikaci typů těchto hodnot. Díky tomuto chování nelze předpokládat, že přiřazení do proměnné vždy způsobí vytvoření nové proměnné a nebude vyžadovat unifikaci typů.

Konkrétní ukázky: První případ je ukázka, kdy je možné znovupoužít jednu proměnou:

```
var = 10;
# pouziti promene var
var = "text" # kompletne nahrazuje puvodni hodnotu
```

Druhý případ je situace, kdy je nutné provést unifikaci:

```
var = A()
if something:
```

```
    var = B()
# v tento okamzik neni znamo, jestli var je typu A nebo B a je tedy
nutne unifikovat
```

Další možností je definice různých typů jedné proměnné ve dvou větvích příkazu `if`, která taky vyžaduje unifikaci. V tomto případě se vytvoří nová proměnná, která se nahradí za výskyty původní proměnné v obou větvích a typ této proměnné je unifikací typů hodnot v obou větvích.

3.4.8 Ztráta specifičtějšího typu

Jedním z problémů, které mohou unifikací nastat, je ztráta specifičtějšího typu, který je vyžadován pro přístup k atributu nebo metodě třídy. Předpokládejme, že máme proměnnou s nějakým specifickým typem a na této proměnné se volá metoda. Poté nastane unifikace této proměnné tak, že výsledný typ je obecnější, který ale tuto metodu nemá definovanou. Algoritmus musí zajistit, že se tento kód detekuje a je zobrazena chyba.

Příklad: Mějme typy `A` a `B`, kde `B` dědí z `A` a `B` definuje metodu `delej()`. Pak mějme takovýto kód:

```
var = B()
var.delej()
var = A()
```

Při volání metody `delej()` je typ proměnné `var` odvozen jako `B` a toto volání je tedy validní. Dále ale dojde k unifikaci s typem `A`, čímž dojde k zobecnění typu proměnné `var` na tento typ. Tento typ ale nemá tuto metodu definovanou a její volání tedy není validní. Výsledkem je tedy ztráta specifického typu.

Prvním řešením je druhý průchod po typové inferenci. V tomto průchodu by se kontrolovalo, zda metody volané na proměnných a atributech jsou na těchto typech definovány. Pokud metoda nebo atribut na typu neexistují, je vyvolána chyba.

Druhým možným řešením je rozšíření konkrétních typů o informaci typu třídy, která je nejvyšší možná v hierarchii tříd. Typ této třídy se nastaví, pokud se na typu volá metoda nebo přistupuje k atributu, která je definována na třídě a nikoliv na jejím předku. Pokud by mělo dojít k unifikaci s typem třídy, která je výše než tato povolená třída, dojde k chybě inference. Pokud použijeme příklad výše, pak při volání metody `delej()` by se typu proměnné var nastavilo toto omezení na typ `B`, protože to je typ třídy, na kterém se tato metoda vyskytuje v hierarchii jako první. Pak by se při unifikaci s typem třídy `A` zjistilo, že typ proměnné var nemůže být výše než `B` a tím dojde k chybě inference.

3.5 Shrnutí

V této kapitole bylo popsáno, jaké vlastnosti jazyka bude navrhovaný překladač podporovat. Dále byl popsán statický typový systém a algoritmus, který pomocí tohoto typového systému odvodí z kódu statické typy. Tento algoritmus je v rámci překladače nejproblematičtější, protože vybrané vlastnosti jazyka ovlivňují složitost celého algoritmu a zároveň algoritmus definuje vlastnosti, které bude mít vybraná podmnožina jazyka.

4 Převod konstrukcí jazyka

Provedením algoritmu pro inferenci typů se získá abstraktní syntaktický strom, který v sobě nese informaci o typech výrazů a příkazů. Po té je dalším krokem převod tohoto abstraktního syntaktického stromu na kód v cílovém jazyce, který je možné zkompilovat do strojového kódu. V případě navrhovaného překladače je cílový jazyk C++.

Důvodů pro výběr jazyka C++ je několik. Jedná se o jeden z nejrozšířenějších staticky typovaných jazyků a překladače pro něj jsou kvalitní a rozšířené. Konstrukce jazyka Python se dobře mapují na konstrukce jazyka C++. Jedná se primárně o funkce a třídy, ale např. i výjimky. A nejdůležitějším faktorem pro výběr tohoto jazyka je podpora šablon, které umožní jednoduchou implementaci generických operací, metod a tříd. Součástí překladače je také vytvoření základní knihovny, která bude implementovat funkčnost takovou, aby byla stejná jako v jazyku Python. Tato knihovna se pak bude používat ve vygenerovaném kódu. Všechny typy a funkce v této knihovně jsou ve jmenném prostoru *base*, aby nedocházelo ke kolizi jmen. Výsledný kód tedy bude obsahovat velké množství funkcí a tříd, které budou využívat šablony.

Druhou variantou je místo generovaní šablon generovat konkrétní funkce přímo. Odpadlo by tak využití některých komplexnějších vlastností šablon v C++. Ale pak naopak by zase došlo k výraznému zkomplikování generování kódu, hlavně v případě parametrických funkcí a typů v modulech. A zároveň by se tvorba konkrétních implementací funkcí a tříd přesunula z překladače C++ do implementovaného překladače, čímž by se výrazně zkomplikoval návrh a implementace generování cílového kódu.

V této kapitole je popsán převod otypovaných konstrukcí jazyka Python na konstrukce jazyka C++. Popis je postupný od jednodušších konstrukcí ke složitějším.

4.1 Výrazy

Mezi základní konstrukce jazyka Python patří výrazy, které zahrnují vytváření atomických a složitých hodnot, přístup k atributům a indexům, binární, unární, logické a porovnávací operátory a volání funkcí a metod.

4.1.1 Hodnoty

Převod výrazů atomických typů **int**, **float** a **boolean** nevyžaduje žádné speciální operace, protože jejich reprezentace je v obou jazycích stejná. Generují se tedy přímo konstanty hodnot.

Typ **String** reprezentuje znak nebo řetězec a chová se jako atomický typ[4] i přesto že je to typ referenční. A to proto, že sémantika jazyka neumožňuje změnit znaky uvnitř řetězce. Pro reprezentaci tohoto typu existuje v základní knihovně třída *string*, která tento typ reprezentuje. Tato třída pak implementuje metody, které odpovídají metodám v jazyce Python. Generovaný kód nového řetězce je tedy vytvoření instance třídy *string* s řetězcovou hodnotou v konstruktoru.

```
a = new base::string("abc") // vytvoření instance nového řetězce
```

[4] Tedy podobně jako typy int, float a boolean

Typ **List** je seznam hodnot, kde lze hodnoty přidávat nebo odebírat. Podobně jako u typu **string** je v základní knihovně třída *list* reprezentující seznam. Tento typ je v případě inferenčního algoritmu parametrizován typem položky, která je v seznamu uložena. Implementace této třídy v knihovně je tedy šablonová, kde šablona reprezentuje typ položek. Tato třída dále implementuje operace, které jsou ekvivalentní operacím seznamu v jazyku Python. Konkrétně se jedná o operace pro přístup přes index, vložení a odebrání položky, délku seznamu a spojování dvou seznamů operátorem sčítání. Vytvoření instance této třídy je problematické pouze v případě, kdy je seznam inicializován s položkami. Problémem je fakt, že jazyk Python umožňuje vytvoření seznamu s položkami jako součást výrazu. V C++ je možné takto inicializovat pouze pole s přiřazením. Není tak ale možné vytvořit jinou strukturu v rámci složitějšího výrazu. Je tedy nutné nejdříve vytvořit dočasnou lokální proměnnou pole, které se inicializuje položkami tohoto seznamu. Toto pole se pak předá jako parametr do konstruktoru třídy reprezentující seznam. Následující kód reprezentuje, jak se převede vytvoření nového seznamu:
Kód v jazyce Python:

```
lst = [10, 20, 30]
```

Kód v C++:

```
int[] __temp1__ = {10, 20, 30};
list<int>* lst = new list<int>(__temp1__, 3);
```

Typ **Tuple** je podobný typu list jen s tím rozdílem, že se jeho délka nemění. A podobně jako **string** není možné hodnoty v něm měnit. Jako komplexní typ má v základní knihovně třídu *tuple*, která zaznamenává, kolik položek daný hodnota tuple má a uchovává seznam hodnot, které lze inicializovat a číst. Hlavním problémem při inicializaci hodnoty tuple je to, že v jazyce C++ nelze jednoduše reprezentovat konstrukci třídy včetně jejich hodnot. Inicializace tedy vyžaduje několik kroků.

1. Je vytvořená nová instance třídy `base::tuple` a přiřazena do dočasné proměnné.
2. Pro každý parametr je na této instanci volána metoda `setValue<type>(index, value)`
    a. `type` je typ parametrů získaný z typové inference
    b. `index` je kolikátý parametr se nastavuje
    c. `value` je výraz, kterého má parametr nabývat
3. Dočasná proměnná se pak vrátí jako kód výrazu.

Ukázka generování hodnoty:

```
tpl = 1, 'abc', False
```

Se převede na:

```
base::tuple* __temp__ = new base::tuple(3);
__temp__->setValue<int>(0, 1);
__temp__->setValue<base::string*>(1, new base::string("abc"));
__temp__->setValue<bool>(2, false);
tpl = __temp__;
```

4.1.2 Operátory

Python podporuje aritmetické operátory (`+, -, *, /, **`), logické operátory (`and, or, not`) a porovnávací operátory(`==, !=, <, >, in, is None`). Přímočarou možností jak výrazy těchto

operátorů převést na kód v jazyce C++ je jejich přímý převod na zdrojový kód reprezentující daný operátor. Toto je vhodné řešení pro výrazy s atomickými typy, ale v případě, že by operátor byl definován pro třídu, bylo by nutné přetížit tento operátor pro danou třídu. Problémem ale je, že všechny hodnoty, jejichž typ je třída, jsou v kódu reprezentovány ukazateli a to znamená, že nad nimi nelze použít přetěžování operátorů. Operátory jsou tedy implementovány tak, aby bylo možné je použít jak nad atomickými hodnotami, tak nad třídami, které jsou uložené v proměnné s ukazatelem. K tomu je využito vlastnosti šablon, která umožňuje specializací určit, že šablona se aplikuje na ukazatel šablonového typu. Omezením této specializace je ale to, že je aplikovatelná pouze na typy a nikoliv na funkce. Výsledný kód pro operátor sčítání může vypadat například takto:

```
base::_add<int>::invoke(1, 1)
base::_add<base::list<int>*>::invoke(l1, l2) // l1 a l2 jsou seznamy
```

`base` je jmenný prostor základní knihovny, `_add` je struktura s šablonou a invoke je metoda v této šabloně. K této struktuře jsou vytvořeny specializace, které zavolají správný operátor podle zadaného typu. Podobně jsou pak implementovány ostatní operátory.

4.1.3 Čtení indexu třídy

Při přístupu k indexu třídy je podobný problém jako s operátory. V tomto případě se ale musí brát v potaz rozdílnost v přístupu k prvkům v typech **list**, **string** a **tuple**. Konkrétně se jedná o hodnotu vrácenou přístupem k indexu. Ty jsou pro tyto typy rozdílné. V případě typu list je to typ definovaný v šabloně, v případě typu string je to typ string a pro typ tuple záleží typ hodnoty na jejím indexu. V tomto případě tedy musí být typ hodnoty zapsán do kódu z typového odvození. Ukázky kódu při čtení indexu:

```
lst[1] # lst je list
str[2] # str je string
tpl[0] # tpl je tuple
```

Vygenerovaný kód:

```
base::subscript<base::list<int>*, int>::read(lst, 1)
base::subscript<base::string*, base::string*>::read(str, 2)
base::subscript<base::tuple*, float>::read(tpl, 0)
```

Podobně jako u operátorů je `subscript` struktura a ta má dva šablonové parametry. Prvním je typ, nad kterým se čtení provádí a druhý je typ návratové hodnoty. Parametry metody `read` jsou instance třídy ze které se čte a index hodnoty. Stejně jako u operátorů jsou pak ke struktuře `subscript` implementovány specializace, které provádějí samotné získání hodnoty podle typu třídy a čtené hodnoty.

4.1.4 Volání funkcí

Volání funkcí je na rozdíl od předchozích případů přímočaré a převádí se na kód v C++ takřka jedna ku jedné. Je nutné si ale dát pozor na generické funkce. V jejich případě se do volání musí přidat parametry šablon, které tyto funkce vyžadují.

Ukázka volání generické funkce `pickElement`:

```
pickElement<int, base::list<int>*>(lst)
```

4.1.5 Přístup k atributům a volání metod

Přístup k atributům tříd a volání metod je převedeno přímo na konstrukce C++. Díky tomu, že proměnné všech třídních typů jsou ukazatele, je nutné použít operátor `->` .

4.2 Příkazy

Každý blok kódu v jazyce Python se skládá z posloupnosti příkazů. Příkazy na rozdíl od výrazů nevracejí žádné hodnoty, ale provádějí nějakou akci. Mezi základní příkazy patří přiřazení a konstrukce pro řízení toku kódu jako if, for, while, break a continue.

4.2.1 Přiřazení

Jak bylo řečeno u typového odvození, příkaz přiřazení má tři různé varianty: přiřazení do proměnné, přiřazení do atributu a přiřazení do indexu pole. Přiřazení do proměnné nevyžaduje žádné speciální volání a je tedy v C++ reprezentováno operátorem = s tím, že na levé straně je jméno proměnné a na pravé straně je přiřazovaný výraz. Stejně je to s přiřazením do atributu. Jen se před jméno tohoto atributu vygeneruje kód výrazu pro získání instance třídy, ke které tento atribut patří a pro propojení se použije operátor `->`. Kód pro získání instance třídy je výraz pro získání hodnoty a je tak i interpretován.

Přiřazení do indexu však vyžaduje speciální kód. A to díky tomu, že typy, nad kterými toto přiřazení může probíhat, jsou všechny třídy a není možné u nich nijak přetížit operátor přístupu přes index, aby byl význam `list[index]=value` v jazyce C++ proveditelný. Proto je generovaný kód volání metody na typu list, která nastaví hodnotu na daném indexu a není vůbec použit operátor přiřazení.

Ukázky kódu jednotlivých typů přiřazení:

```
var = value;
class.attribute = value;
lst1->setValue(2, 10);
```

4.2.2 Řízení toku kódu

Konstrukce **if**, **while** se dají převést přímo na ekvivalentní konstrukce v C, protože podmnožina překladače podporuje pouze booleovské hodnoty místo implicitního převodu jakéhokoliv typu na booleovskou hodnotu. Příkazy **break** a **continue** jsou také ekvivalentní s jejich protějšky v C++.

Složitější je ale převod příkazu **for**, protože se může převést jinak v závislosti na typu výrazu nad kterým se cyklí. První možností je, že ve výrazu bude pouze funkce `range(...)`, která se používá pro iteraci přes posloupnost čísel. I přesto, že na první pohled vypadá tato funkce jednoduše, tak se v ní skrývají dva problémy, které je nutné brát v potaz. Prvním je možnost, kdy proměnná, která je v parametru funkce `range` je změněna uvnitř cyklu. Pokud by se proměnná použila přímo v příkazu `for` v C++, tak její změna uvnitř cyklu způsobí změnu chodu cyklu. To samé platí i pro iterační hodnotu. Přiřazením do ní v rámci cyklu nesmí ovlivnit chod celého cyklu. Toto lze řešit jednoduše tak, že se pro všechny parametry funkce `range` a iterační hodnotu vytvoří lokální, dočasné proměnné.

Druhým problémem je fakt, že posloupnost čísel může být jak rostoucí, tak klesající a to v závislosti na třetím parametru funkce `range`. Podle toho se musí měnit ukončovací podmínka mezi > a < , tak aby došlo ke správnému ukončení.

Ukázka převodu jednoduchého cyklu:

```
for i in range(2, a):
    print(i)
```

Se převede na:

```
int __from1__ = 2;
int __to1__ = a;
int __step1__ = 1;
bool __dir1__ = 1 > 0;
for(int __index1__ = __from1__; __dir1__ ? __index1__ <
__to1__:__index1__ > __to1__; __index1__+= __step1__){
i = __index1__;
    base::_print<int>::invoke(i);
}
```

Obecná varianta nastává, když se iteruje přes obecný výraz. Na rozdíl od verze s `range` není v tomto případě žádný podobný problém. Výraz musíme uložit do lokální proměnné, abychom přes něj mohli iterovat, a interní iterační proměnná musí být také lokální, protože reprezentuje index nikoliv hodnotu.

Ukázka převodu iterace přes obecný výraz:

```
for v in get_items():
     print(v)
```

Tento kód se pak převede na kód:

```
list<int>* __temp1__ = get_items();
for(int i = 0; i < __temp1__->count(); i++){
     int v = __temp1__->get(i);
     print<int>(v);
}
```

4.3 Definice

Definice funkcí a uživatelských typů může být problematická, protože vyžaduje generování do dvou souborů. Deklarace do hlavičkového souboru a definice do zdrojového souboru. Ale díky tomu, že překladač prozatím nepodporuje vytváření vlastních modulů, není toto rozdělení potřeba. Kód se tedy generuje přímo bez rozdělení na deklaraci a definici. Ale všechny potřebné informace pro generování jsou dostupné a není tedy problém tuto funkčnost doimplementovat.

4.3.1 Definice funkcí a bloky příkazů

Definice nové funkce se skládá z návratového typu, jména, parametrů funkce, parametrů šablon a těla funkce. Jméno je známo přímo. Návratový typ a typy parametrů se zjistí z typového odvození.

Parametry šablon reprezentují všechny generické typy, které se ve funkci vyskytují. A jejich textová reprezentace je stejná napříč celou funkcí.

Tělo funkce se skládá z bloku příkazů. Každý příkaz se transformuje samostatně. Specifickou částí bloků je jejich vazba na rozsah platnosti proměnných. Všechny lokální proměnné mají rozsah platnosti v rámci funkce. Na začátku těla funkce jsou tedy deklarovány všechny proměnné s jejich typy a jsou inicializovány na nulové hodnoty.

Ukázka kódu definice generické funkce:

```
def func(lst, i):
    x = i + 1
    return lst[x]
```

Převedený kód v C++:

```
template <typename T2, typename T1>
T2 func(T1 lst, int i){
    int x;
    x = base::_add<int>::invoke(i, 1);
    return base::subscript<T1, T2>::read(lst, x);
}
```

4.3.2 Třídy

Třídy v jazyce Python se mapují přímo na třídy v Jazyce C++. Z typového odvození se získají informace o atributech a jejich typech a případné parametry šablon třídy. Pokud má třída přiřazen parametr jako generický typ, je převeden na šablonu třídy. Atributy se převádějí na veřejné proměnné a metody se převádějí podobně jako funkce. Parametr `self` ve výrazech v těle metod se převádí na `this`. Jednou ze složitějších částí generování kódu tříd je přetěžování virtuálních metod, které jsou definovány na základním objektu `base_object`. I přesto, že třída ve zdrojovém kódu tyto metody nepřetěžuje, je nutné, aby výsledný kód v C++ tyto metody přetěžoval a vhodně implementoval, aby bylo jejich chování vhodné dané třídě. K tomu slouží to, že každá vygenerovaná třída musí dědit z `base::base_object` a jeho abstraktní a virtuální metody metody jsou vhodně přetíženy.
Ukázka převodu třídy a jejího použití:

```
class Trida:
      def init(self, data):
            self.data = data

      def update_data(self, newData):
            self.data = self.data + newData
            return self.data

var1 = Trida(10)
print(var1.update_data(5))
var2 = Trida('ab')
print(var2.update_data('cd'))
```

se po odvození typů převede na:

```
// definice tridy
template <typename T1>
class Trida : public base::base_object {
public:
      base::string* toString(){
            return new base::string(
            "<__main__.Trida object at 0x"+__getAddress__()+">");
      }
      T1 data;
      Trida(T1 data){
            this->data = data;
      }

      T1 update_data(T1 newData){
            this->data =
            base::_add<T1>::invoke((this->data), newData);
            return (this->data);
      }
};

Trida<int>* var1;
Trida<base::string*>* var2;
var1 = new Trida<int>(10);
base::_print<int>::invoke(var1->update_data(5));
var2 = new Trida<base::string*>((new base::string("ab")));
base::_print<base::string*>
::invoke(var2->update_data((new base::string("cd"))));
```

Jednou z vlastností jazyka Python je to, že všechny komplexní typy jako jsou list a programátorem definované třídy jsou referenční typy. Při přiřazení do proměnné se přiřazuje pouze reference. Podobného chování se v C++ dá dosáhnout tak, že se se všemi komplexními typy manipuluje jako s ukazateli. Pak ale vyvstává problém správy paměti. Detailně je tento problém popsán v kapitole o implementaci.

4.4 Struktura generovaného kódu

Při pohledu na vygenerovaný kód může vyvstat otázka: Proč se generuje volání šablonových metod, když typy jsou známé a operace samotná je jednoduchá. Toto je nejvíce vidět u výrazů, které jsou jednoduché, jako například operace sčítání, která se generuje jako volání metody `base::_add<int>::invoke(1, 1)` místo přímého generování operátoru +. Překladač je možné o toto chování, kdy překladač generuje kód přímo, místo toho, aby generoval volání metody, v případě, že typy a operace jsou známy, rozšířit. Otázkou ale je, co lze tímto rozšířením získat. V případě takto generovaného kódu by byl výsledný kód čitelnější pro programátora, protože hlavně výrazy by

vypadaly velice podobně jako ve zdrojovém kódu a nebyly by reprezentovány zanořenými funkcemi, které samy o sobě jsou složitě vypadající kusy kódu.

Druhým důvodem, proč by bylo vhodné takovýto kód generovat je rychlost aplikace přeložené z vygenerovaného kódu. Volání funkcí má jisté množství režie, které může být nezanedbatelné, pokud je těchto volání hodně a tělo funkce samotné je velice jednoduché. Například v případě zmíněné funkce `_add` je její tělo pouze příkaz `return x + y;`. Při návrhu překladače a generovaného kódu jsem však předpokládal, že moderní překladače jazyka C++ jsou schopny tento kód optimalizovat. Tento předpoklad je dán tím, že vygenerovaný kód neobsahuje žádné výrazy, které by nebyly známy v době překladu. To dává překladači možnost jak vytvořit konkrétní funkce z funkcí šablonových, tak provést jejich vložení do místa volání[5]. Kvůli tomu je většina těchto knihovních funkcí označena jako `inline`, což dává překladači jazyka C++ možnost tato složitě vypadající volání optimalizovat na jednoduchý binární kód v aplikaci. Je ale diskutabilní, zda generováním konkrétních operací, místo volání metod v základní knihovně bude dosaženo nějakého znatelného navýšení výkonu, hlavně proto, protože moderní překladače jazyka C++ jsou dost kvalitní, aby byly tento kód schopny optimalizovat bez pomoci manuální optimalizace.

Zásadní komplikací v případě implementace tohoto rozšíření by byla duplikace kódu a funkcionality v překladači. Existující šablonové funkce musí být pořád v základní knihovně a musí být generovány překladačem pro případy, kdy nejsou typy známy a místo nich se vyskytují šablonové typy. Tyto šablonové funkce v základní knihovně mají vždy několik variant, které jsou implementovány jako specializace šablon. Překladač jazyka C++ na základě těchto specializací, podle zadaného typu generuje různý kód. Stejná funkčnost by musela být i v implementovaném překladači a to jak funkčnost výběru konkrétního kusu kódu, tak kód samotný. Toto způsobí duplicitu, která může být problematická při další údržbě a rozšiřování překladače. A zároveň se tím zvýší množství případů, které se musí testovat.

Je tedy otázkou, zda implementace tohoto chování do překladače by měla dostatečné výhody, aby se odůvodnily komplikace spojené s horší udržovatelností implementace generování cílového kódu.

4.5 Shrnutí

Převod otypovaného abstraktního syntaktického stromu na kód v jazyce C++ je relativně přímočarý proces, protože velké množství konstrukcí jazyka Python je ekvivalentních s konstrukcemi jazyka C++. Zásadním rozdílem kódu v jazyce Python a vygenerovaným cílovým kódem v jazyce C++ je použití šablon pro definici a volání funkcí a tříd. Díky tomu je výrazné množství komplikací způsobené generováním funkcí a tříd, které operují nad různými typy přesunuto do překladače jazyka C++. Zároveň je značné množství funkcionality cílového kódu přesunuto do funkcí a tříd implementovaných v základní knihovně a není tedy nutné jejich kód generovat.

[5] Function inlining

5 Implementace překladače

V této kapitole popíši konkrétní způsob implementace celého překladače, důvody výběru konkrétních algoritmů a struktur a problémů, se kterými jsem se při implementaci setkal.

Jako každý překladač se je implementace rozdělena do několika částí: lexikální a syntaktická analýza, typová inference a kontrola sémantiky, generování kódu.

5.1 Syntaktická analýza a strom

Hlavním problémem syntaktické analýzy kódu v jazyce Python je význam odsazení pro definici bloků. Toto nelze zapsat bezkontextovou gramatikou, které se používá pro popis syntaxe jazyka. Tento problém lze vyřešit v lexikálním analyzátoru. Ten si pamatuje počet bílých znaků na každém řádku. Pokud je počet bílých znaků na následujícím řádku větší, je generován token INDENT. Pokud je počet bílých znaků menší, je generován token UNDENT. Následně se generuje abstraktní syntaktický strom kódu pomocí jazykem-definované bezkontextové gramatiky. Lexikální analýzu a tvorbu abstraktního syntaktického strumu pak lze zjednodušit použitím knihoven LEX/FLEX nebo YACC/Bison.

V případě implementovaného překladače však není nutné se tímto zabývat, protože Python v základní knihovně poskytuje syntaktický analyzátor jazyka, který generuje syntaktický strom zdrojového kódu [15]. Tento syntaktický strom se pak převede na podobný strom, který se skládá z tříd, nad kterými se pak provádí odvození typů a generování kódu. Tyto třídy jsou rozděleny na dvě skupiny: výrazy a příkazy a obecně kopírují strukturu tříd v originálním syntaktickém stromu.

5.2 Typový systém a unifikační algoritmus

Typový systém a unifikace typů je jádrem celého překladače. Typový systém musí mít primárně tyto vlastnosti:

- Reprezentace všech podporovaných typů
- uchovávání množiny typů v rámci bloku kódu nebo funkce
- podpora pro konkrétní a generické typy
- podpora pro uživatelem definované typy
- unifikace typů a tranzitivní chování unifikace

Na diagramu Diagram 4 - Struktura tříd implementace typového systému a odvozování(str. 48) je celkový pohled na strukturu implementace typového systému a součástí, které slouží pro unifikaci a tedy i odvozování typů. Hlavními třídami v implementaci jsou `TypeDefinition`, `TypeInstance` a `Context`. `TypeDefinition` reprezentuje definici typu, která ale není unifikovatelná. K unifikaci slouží `TypeInstance`, která kopíruje strukturu typů v `TypeDefinition` a rozšiřuje jejich vlastnosti, aby nad nimi unifikace byla proveditelná.

5.2.1 Rozsah platnosti

Rozsah platnosti typů a proměnných je reprezentován instancí třídy `Context` a tato třída uchovává seznam všech typových instancí, které se reprezentovaném rozsahu platnosti vyskytují. Nelze vytvořit typovou instanci, aniž by měla přiřazený kontext a naopak aby tato typová instance byla kontextu známa. Proto se všechny typové instance vytvářejí metodou

`Context.createTypeInstance(definition)`, která zajistí, že tyto vazby jsou správně inicializovány. `Context` také uchovává všechny proměnné a funkce, které lze v tomto bloku využívat.

Specializací této třídy je třída `FunctionContext`, která reprezentuje blok kódu v rámci definice nové funkce. Ta přidává atribut reprezentující typ návratové hodnoty a funkcionalitu unifikace tohoto typu v případě, že se v těle funkce vyskytuje více příkazů `return`. Také zajišťuje, aby se správně odvozovalo rekurzivní volání. V případě, že se volá funkce, která má stejný identifikátor jako právě analyzovaná funkce, pak je to právě tato třída, která zajistí, aby došlo k unifikaci tohoto volání s typy parametrů a návratového typu právě analyzované funkce.

5.2.2 Typové definice

Struktury kolem typové definice jsou vytvořeny tak, aby pomocí nich bylo možné reprezentovat všechny podporované typy. A to nejenom ty základní, ale i parametrické a definované uživatelem. Typové definice jsou rozděleny do dvou skupin, které jsou reprezentovány třídami `ConcreteType` a `GenericType` a jejich podtřídami. Tyto třídy tedy reprezentují typy tak, jak byly popsány v návrhu. Tato reprezentace je často ve formě grafu, kde jednotlivé instance tříd jsou uzly grafu a hrany jsou parametrické vazby mezi nimi. K tomu je využito referenční rovnosti instancí jazyka Python. Toho je primárně využito při převodu složitých typových definic na jejich typové instance, jak je popsáno v další kapitole. Typové definice i přesto, že jsou v implementaci vyjádřeny třídami, nemají vlastní chování a slouží hlavně jako struktury popisující možné typy.

5.2.3 Typové instance a jejich unifikace

Typové definice nemůžou být použity pro unifikaci typů, protože u nich není možné provádět tranzitivní unifikaci. K samotnému odvozování a unifikaci typů slouží třídy `TypeInstance`. Ta nereprezentuje typ, ale reprezentuje místo, kde se může vyskytovat typ. Kód, který provádí odvozování typů ze syntaktického stromu, tedy pracuje primárně s typovými instancemi a nikoliv s definicemi. Typové instance zároveň kopírují strukturu definic a to tak, že každá instance může mít parametry, které jsou také instancemi. To je vyžadováno, aby unifikace byla schopna unifikovat i parametry typů a nejenom typy samotné. Druhý atribut typové instance, který umožňuje unifikaci je množina `unifies`, která obsahuje typové instance, které byly s touto instancí unifikovány. Při vytvoření nové typové instance tato množina obsahuje pouze instanci sebe sama.

Jak je napsáno v návrhu, unifikace dvou typových instancí je rekurzivní proces. Prvním krokem je ukončovací podmínka. Pokud už typové instance byly unifikovány, je unifikace ukončena. To jestli byly typové instance unifikovány, se pozná jednoduše tak, že jejich množiny `unifies` jsou stejné. Druhým krokem je vytvoření nové množiny `unifies` a to tak, že se sjednotí množiny `unifies` obou instancí. Tato nová množina pak reprezentuje všechny typové instance, kterých se unifikace týká a u kterých je nutné změnit jejich definice, parametry nebo omezení. Tato množina se pak nastaví všem instancím v novém `unifies`. `Unifies` pak prakticky reprezentuje relaci ekvivalence mezi všemi typovými instancemi, které byly unifikovány, protože pak pro všechny takovéto instance existuje vazba každé na každou. Na diagramu Diagram 6 - Vizualizace unifikace typových instancí(str. 49) je znázorněn vliv množiny `unifies` na typové instance při unifikaci.

Následně probíhá unifikace odlišně podle toho, jestli se unifikují dva konkrétní typy, dva generické typy nebo generický a konkrétní typ. V každém případě jsou ale změny prováděny na všech

instancích třídy `TypeInstance` v množině `unifies`. To zaručí, že při čtení informací o typu z instance jakékoliv třídy `TypeInstance` jsou stejné pro všechny instance, které byly dříve unifikovány.

Unifikace generického a konkrétního typu je implementačně nejvíce přímočará. Významově se jedná o dvě části: potvrzení, jestli omezení definované generickým typem jsou podporovány konkrétním typem a unifikace parametrů omezení s parametry konkrétního typu v závislosti na typu omezení. Omezení je definováno abstraktní třídou `GenericConstraint`, která má dvě metody: `ensureIsApplicable` a `applyTo`. První metoda musí způsobit chybu unifikace, pokud konkrétní typ toto omezení nepodporuje. Druhá metoda zajistí, že se správně unifikují parametry. Konkrétní typy omezení pak implementují tyto metody podle potřeby tohoto omezení. U některých jednodušších omezení nemusí implementace applyTo dělat nic. Ukázkou implementace složitějšího omezení je omezení na čtení indexu. Ten se totiž podle návrhu chová jinak pro seznam, pro řetězec a pro tuple. Toto omezení má jeden parametr typové instance, která reprezentuje typ hodnoty který index vrací a jeden atribut `indexConstant`, který obsahuje hodnotu indexu a to v případě že je index volán s konstantou typu `int`, pokud ne, je tento atribut prázdný. Metoda `ensureIsApplicable` je implementována tak, že platí pro případ kdy definice konkrétního typu je `list`, `string` a nebo `tuple`, ale v tom případě musí být atribut `indexConstant` definovaný. Metoda `applyTo` pak postupuje rozdílně pro jednotlivé definice. V případě typu `list` se parametr omezení unifikuje s typovou instancí, která reprezentuje parametr seznamu, tím dojde k tomu, že typ návratové hodnoty čtení indexu je typ uložený v seznamu. V případě typu `string` se unifikuje s konkrétním typem `string`, protože takto je definováno čtení indexu na tomto typu. Pro typ tuple je to složitější, protože typ vrácené hodnoty je závislý na hodnotě indexu. K tomu slouží právě atribut `indexConstant`. Pro `tuple` tedy `applyTo` unifikuje parametr omezení a parametr typu `tuple` daný hodnotou tohoto atributu. V závislosti na hodnotě tohoto indexu tedy může být návratový typ čtení indexu jiný. Rozšíření tohoto omezení by nastalo, pokud by překladač podporoval typ `dictionary`. V tomto případě by muselo omezení mít i parametr reprezentující typ indexu a tento parametr by se unifikoval s typem klíče, pokud typ, na který se omezení aplikuje, byl `dictionary` a jinak s typem `int`.

Unifikace dvou generických typů má podle návrhu význam spojení omezení obou typů. Toto spojení je stejné, jako kdyby se oba generické typy unifikovaly s budoucím konkrétním typem. Výsledkem unifikace dvou generických typů je nový generický typ, jehož omezení jsou sjednocením omezení těchto typů. Hlavním implementačním problémem je tedy sjednocení těchto omezení. Omezení, které se vyskytují pouze v jednom, nebo v druhém vstupním typu jsou jednoduše zkopírovány do výstupního typu, protože není nic, co by je mohlo jinak ovlivnit. Jinak je to ale pro omezení, která jsou si rovna. Dvě omezení jsou si rovna, pokud mají stejný typ a stejné atributy. Například `AddConstraint` je roven pouze na základě typu, kdežto `MethodConstraint` je roven jinému `MethodConstraint` pouze pokud reprezentují stejnou metodu a tedy jsou si rovny atributem `name`. Pro takováto omezení se ve výstupním typu vytvoří ekvivalentní omezení se stejným typem a atributy, ale jehož parametry jsou unifikací parametrů dvou rovných omezení. To zajistí, že při budoucí unifikaci s konkrétním typem je toto omezení reprezentativní oběma omezeními, ze kterých bylo spojeno.

Unifikace dvou konkrétních typů je, díky tomu, že implementovaný překladač nepodporuje dědičnost, přímočará. Prvním krokem je ověření, zda jsou oba typy ekvivalentní. Typy jsou ekvivalentní, pokud jsou jejich třídy stejné a pokud jsou jejich argumenty stejné. Toto je důležité

například pro typ `tuple`, kde `tuple` s dvěma položkami není stejný typ jako `tuple` s třemi položkami. Pokud typy nejsou stejné, je možné, aby existoval typ, jehož definice je substituovatelná za oba konkrétní typy. Pro implementovaný překladač to platí pouze pro vztahy mezi typy `numeric`, `int` a `float`, tak jak bylo řečeno v návrhu. Následně, pokud je typ parametrický, jsou unifikovány všechny parametry obou vstupních typových instancí.

V případě, že by překladač podporoval dědičnost, bylo by množství komplikací spojené s její implementací soustředěno právě v unifikaci konkrétních typů. Nalezení společného předka v hierarchii není komplikované. Problém ale nastává v typových parametrech tříd a faktu, že potomci tříd můžou specializovat, přidávat a odebírat typové parametry svých předků. Musí se tedy brát v potaz i unifikace a změna množiny typových parametrů napříč hierarchií mezi dvěma vstupními definicemi tříd a definicí společného předka. A také to, že tyto vztahy je nutné uložit jako typové definice.

Ukázkou těchto problémů je kód těchto tříd:

```
class Parent:
    def __init__(self, a, b):
        self.attA = a
        self.AttB = b

class Child(Parent):
    def __init__(self, a, b, c):
        self.attC = c
        a = c + 1
        b = b + b
        return super().__init__(a, b)
```

Třída `Parent` má dva atributy: `attA` a `attB`. Typy těchto atributů jsou `generic`. To znamená. že tyto typy budou určovat jako typové parametry třídy `Parent`. Třída `Child`, která z dědí ze třídy `Parent`, však tyto typy mění. Přidává nový atribut `attC`, který je generického typu a tedy přidává nový typový parametr třídě `Child`. Také se v těle této třídy provádí unifikace typu atributu `attA` s typem `Numeric`, což je konkrétní typ. Díky tomu nastane situace, kdy z původně generického typu v rodičovské třídě stane typ konkrétní v potomkovi. To způsobí, že typový parametr co byl v rodiči neexistuje v potomkovi. Tato situace také musí být kódována v definicích typu tříd. U atributu attB dochází k něčemu podobnému, ale nejedná se o konkretizaci generického typu, ale o přidání omezení ke generickému typu. V tomto případě se jedná o přidání omezení na existenci operace sčítání. Díky těmto možným změnám dochází ke komplikaci v reprezentaci typů tříd, jejich atributů a metod.

5.2.4 Převod typové instance na definici a naopak

Jedním z problémů implementace odvozovacího algoritmu je propojení typů z definice funkce nebo třídy s jejich použitím. Odvozením definice funkce je typ, který zahrnuje typy parametrů a návratové hodnoty funkce a jejich vztahy, které odpovídají vnitřnímu fungovaní funkce. Voláním funkce se tyto typy unifikují s typy, které mají hodnoty předávané jako parametry. Tato unifikace pak způsobí konkretizaci těchto typů kontextu volání funkce. Rozdílná volání pak můžou mít rozdílné konkrétní typy. Pokud by se pro každé volání funkce použily stejné typové instance, které vznikly z definice, došlo by k tomu, že by se různá volání unifikovala vůči sobě, což je nežádoucí. Pro každé volání je teda

nutné vytvořit kopii celého grafu tvořeného typovými instancemi typů funkce. Zároveň je vhodné, aby tyto kopie byly minimální a neobsahovaly redundantní instance. Podobná situace je i u definice a použití tříd. Proto je implementován převod grafu typových instancí na graf definice typů a zpět.

Tato funkčnost je implementována třídou `TypeInstanceBuilder`. Vstupem procesu převodu jsou typové instance reprezentující parametry a návratovou hodnotu funkce a výstupem je typ `FunctionDefnition`, který reprezentuje stejný typ jako tyto definice. Proces je implementován rekurzivně. Pokud převáděná typová instance je unifikována s typovou instancí, která už byla dříve zpracována, pak je vrácena definice pro tuto instanci. Jinak se vytvoří definice nová. Nová definice je totožná s definicí uloženou v typové instanci. Pokud je tato definice parametrická, je zabalena do instance třídy `ParametricWrapper` a parametry této instance jsou nastaveny na definice získané rekurzivním voláním převodní funkce na parametry v typové instanci. Pokud je definice generická, jsou do nová definice přidány i omezení uložené v typové instanci, jsou zkopírovány jejich atributy a parametry převedeny na definice. Po vytvoření nové instance je tato instance přidána do seznamu zpracovaných instancí. Vytvoření grafu typových instancí z definice je stejný postup.

Důvod, proč jsou výše zmíněné převodní procesy reprezentovány vlastní třídou, je fakt, že v rámci celého procesu je nutné uchovávat seznam zpracovaných typových instancí. A z pohledu objektového návrhu je lepší mít tento seznam jako atribut třídy než si předávat referenci na seznam jako parametr všech metod, které se v těchto procesech vyskytují.

Výstupem výše zmíněných procesů je také množina generických parametrů, které se pak použijí pro generování signatury metody. Ty jsou získány jako typové instance ze seznamu převedených instancí, které jsou generické. Aby tyto generické signatury byl stejné jak pro definici, tak pro volání funkce, je kód strukturován tak, aby pořadí procházení typových parametrů bylo stejné jak při vytváření definice z typové instance, tak vytvoření typové instance z definice.

Součástí tvorby definice funkce je i kontrola na problém popsaný v kapitole 3.4.5 . Předpokladem k řešení tohoto problému je znalost všech typových instancí obsažených v odvozovaném bloku, což je zajištěno třídou `Context`. Dále je potřeba znalost všech typový instancí, které jsou obsaženy v signatuře funkce, což je obsaženo v třídě `TypeInstanceBuilder` po dokončení odvození definice typu funkce nebo třídy. Rozdílem těchto dvou množin lze zjistit, jaké typové instance jsou použity pouze v těle funkce a nefigurují jako typový parametr. Pokud mezi těmito typovými instancemi existuje nějaká s generickou definicí, znamená to, že tato typová instance nemůže být dále unifikována s možným konkrétním typem, což by mělo nastat v případě použití typu definice v kódu a není tedy možné vygenerovat zdrojový kód tohoto typu. Podobná kontrola se provádí i v případě hlavního těla kompilovaného souboru, ale díky tomu, že toto tělo nemá žádnou signaturu, jsou takto kontrolovány všechny typové instance v rámci jeho kontextu. Z toho vyplývá fakt, že hlavní tělo překládaného souboru nemůže obsahovat žádný generický parametr a všechny typy musí být konkrétní. Pokud by toto tělo obsahovalo generický typ, znamená to, že celý program není typově konkrétní a není možné z něj vygenerovat zdrojový kód.

5.3 Odvození a generování tříd

V odvození atributů a metod tříd hraje hlavní roli parametr `self`. Všechny instanční metody třídy musí mít tento parametr jako první a reprezentuje aktuální instanci objektu. Na rozdíl od jazyků jako je C++ nebo Java ale v jazyce Python neexistuje implicitní parametr aktuální instance. Všechen přístup k instanci musí probíhat přes parametr `self`. Toto výrazně zjednodušuje implementaci typového

odvození, protože není nutné brát v potaz fakt, že přístup k proměnné by mohl znamenat přístup buď k atributu aktuální instance, nebo ke globální proměnné. Je zde však jiný problém: Tato proměnná nemá pevně daný typ, protože typ, který reprezentuje, ještě nebyl plně definován, protože právě dochází k jejímu typování. To vyžaduje speciální struktury a chování unifikace. Na začátku odvození třídy se tedy vytvoří speciální třída `ClassTypeInstance` s definicí typu `ClassDefinitionType`. Tato typová instance se pak přiřadí jako typ parametru `self` u všech metod. Důvod proč existuje speciální podtřída typové instance, je v tom, že tato typová instance uchovává všechny atributy, metody a nedefinované metody, které byly zjištěny přístupem k parametru `self`. Tyto uložené informace se pak použijí pro vytvoření pevné definice třídy na konci odvození jejího typu. `ClassDefinitionType` zároveň značí typ třídy, která prochází definicí. Konečná definice třídy má typ `ClassType`.
Mějme kód zápisu do atributu třídy:

```
    def func(self):
        self.a = 1
```

V tomto případě dochází k unifikaci generického typu s omezením na přístup k atributu se jménem a `a` typem `int` a typu třídy ve které je tato funkce definována. V případě, že by byla unifikace prováděna s již definovanou třídou, došlo by pouze ke kontrole, že třída má daný atribut a unifikaci typu v omezení a v třídě od tohoto atributu. Avšak v případě unifikace s nedefinovanou třídou není jasné, jestli tento atribut existuje. Je to tedy podobné chování, jak je přiřazení do proměnné. Pokud tedy atribut s daným jménem neexistuje, je v typové instanci vytvořen atribut nový.

Při dokončení odvození metody je její signatura, tedy její jméno, a typové instance návratového typu a argumentů uložena do typové instance třídy. Problém ale může nastat s tímto kódem:

```
class test:
    def fun1(self, a):
        return self.fun2(a)

    def fun2(self, a):
        return a
```

Toto je validní kód, ale naivní překlad selže, protože se volá funkce, která ještě nebyla definována. Toto je řešeno tak, že v případě že se k takovému volání dojde, je signatura této funkce uložena jako nedefinována metoda. Pokud pak dojde k definici takto dříve volané metody, je odstraněna z nedefinovaných funkcí a jejich signatury jsou unifikovány. Zároveň nastává podobný problém jako u generických omezení a to ten, že jedna nedefinovaná funkce může být volána vícekrát. V tomto případě jsou jejich signatury unifikovány. Nakonec, při převodu na definici se zkontroluje, jestli je seznam těchto metod prázdný. Pokud není, je to indikace toho, že byla volána metoda, která nebyla definována a je vyvolána chyba.

Stejně jako `TypeInstance` má svůj `TypeInstanceBuilder`, tak i `ClassTypeInstance` má svůj `ClassTypeInstanceBuilder`, který dědí z `TypeInstanceBuilder`. Viz Diagram 5 - Struktura typů okolo InstanceBuilder(str. 48). `ClassTypeInstanceBuilder` je použit na převod `ClassTypeInstance` na `TypeDefinition`, který se pak používá jako parametr pro vytvoření typových instancí při vytvoření nové instance třídy voláním konstruktoru. Samotný převod vytvoří definice typů pro jednotlivé atributy

a signatury metod a určí typové parametry třídy. K vytvoření samotných definic se používá funkcionality zděděné z `TypeInstanceBuilder`.

K této funkcionalitě je však nutné zahrnout jeden specifický případ:

```
class test5:
    def __init__(self):
        self.recurse = self

    def returnSelf(self, val):
        return self
```

Tento kód vytváří atribut a metodu, jejichž typy obsahují definici typu, ve kterém se nachází. To je však problém, protože podle výše uvedené logiky se musí zjistit definice atributů a metod před tím, než dojde k vytvoření definice třídy a naopak, k vytvoření definice třídy je nutné znát definice atributů a metod. K rozbití tohoto cyklu se nejdříve vytvoří instance objektu definice třídy, provede se odvození atributů a metod a pak se objektu definice nastaví potřebné informace, aby vznikla plnohodnotná definice třídy. Odvození atributů ale musí mít speciální podmínku, která rozšiřuje chování zjištění již vytvořené definice. V případě že se zpracovává instance, která je unifikována s typovou instancí zpracovávané třídy, je vrácena ona nekompletní definice. Toto je důvod, proč se definice musí vytvořit před převodem atributů.

`ClassTypeInstanceBuilder` je, podobně jako předek `TypeInstanceBuilder` použit i opačným směrem. Tedy pro vytváření typových instancí z definic typů. Avšak na rozdíl od svého předka vytváří typové instance pro atributy a metody. Důvodem k tomuto je fakt, že definice typů atributů a jejich parametry mohou obsahovat definici generického typu a pro něj už existují typové instance jako parametry v typové instanci, nad kterou se atribut nebo metoda volá. Toto je, podobně jako u převodu opačným směrem, implementováno jako rozšíření zjištění existence již vytvořené typové instance. Pokud se zpracovávaná definice rovná definici některého z parametrů, je vrácen korespondující parametr z typové instance volané třídy.

S třídami souvisí i jejich inicializace při vytvoření nové instance. K tomu je v jazyku Python určena metoda `__init__`, která reprezentuje metodu podobnou konstruktoru v jazyce C++, ale s mírně odlišnou sémantikou. Tato metoda se volá vždy při vytvoření nové instance a její atributy definují parametry, se kterými se musí vytvářet nová instance. Na rozdíl od ostatních funkcí však není generována v cílovém kódu, protože C++ má jiný způsob zápisu konstruktoru. Z této funkce se využívají její argumenty. Z těchto argumentů se pak vytvoří definice funkce v globálním jmenném prostoru, která vrací definici třídy, kterou konstruktor vytváří. Tímto je zajištěno, že se může vytvořit instance této třídy voláním jejího jména, jako volání globální funkce.

Podobně jako u metod i třídy mohou být parametrické. To je reprezentováno jak v jejich definicích, tak ve výsledném kódu, kdy takovéto třídy mají šablony. Typy těchto parametrů jsou stejně jako u funkcí odvozeny ze zpracovaných typových instancí při vytvoření definice třídy v `ClassTypeInstanceBuilder`. S parametrickými typy souvisí i omezení, že generické parametry mohou mít pouze třídy a nikoliv metody. Tohle je hlavně díky tomu, že by pak nebylo možné generovat kód, který využívá šablony, protože při volání metody na třídě by nebyla známa informace, zda je metoda parametrická, nebo ne, anebo kolik má parametrů. Toto by šlo vyřešit, kdyby se nepoužívaly šablony, ale generoval se nešablonový kód, ale to je mimo rozsah této práce.

Jak bylo na začátku kapitoly řečeno, s odvozením atributů a metod u třídy souvisí omezení u generických typů na existenci atributu a omezení na existenci metody. Tato omezení však provádějí

pouze kontrolu a v případě že atribut nebo metoda neexistují, pak dojde k chybě. Pro odvození atributů třídy z přístupu k nim, v metodách této třídy, je však nutné, aby tato omezení zaznamenala, jaké atributy a metody volají. Proto je v obou případech přidána funkčnost, která toto zařídí. Pokud, při aplikování omezení je má kontrolovaná typová instance definici `ClassDefinitionType`, pak dochází právě k rozšiřování definice třídy. Místo kontroly na existenci atributu nebo třídy se tedy zavolá metoda na typové instanci `ClassTypeInstance`, která přidá nebo unifikuje atribut nebo metodu, které dané omezení reprezentuje.
Mějme ale tento kód:

```
def func2(self):
    self2 = self
    self2.func1()
    print(self2.a)
```

Na první řádku dojde k unifikaci lokální proměnné na definici právě zpracovávaného typu, ale typová instance bude v tomto případě obyčejná `TypeInstance` a nikoliv `ClassTypeInstance`, na které se nachází metody pro přidání atributu nebo funkce. To ale lze jednoduše vyřešit tak, že `ClassTypeInstance` najdeme v seznamu `unifies` typové instance. Toto si můžeme dovolit, protože `ClassTypeInstance` je vždy jen jedna v aktuálním kontextu. Nemusí tedy docházet k synchronizaci přidávaných atributů nebo metod mezi všemi typovými instancemi, jak je tomu například u parametrů.

5.4 Implementace základní knihovny

Součástí překladače je také základní knihovna, která se kompiluje spolu s vygenerovaným kódem v C++. Návrh a implementace této knihovny se řídí možnostmi jazyka C++ a jeho funkčnost v kombinaci s vygenerovaným kódem musí být ekvivalentní funkčnosti jazyka Python.

Hlavní součásti knihovny je třída `base_class`, ze které dědí všechny třídy, které reprezentují třídy v jazyce Python. A to jak ty v základní knihovně, tak ty vygenerované překladačem. Tato třída neposkytuje množství základní funkčnosti, ale spíše funguje jako rozhraní pro volání operací, které jsou dostupné na každém objektu, jako je porovnávání nebo převod na textovou reprezentaci.

V knihovně se také nachází implementace operátorů a základních funkcí, které nejsou vázány na konkrétní objekt. Některé operátory ale vyžadují speciální vlastnost jejich použití: musí jít provést jak nad atomickými typem, tak nad třídou, která je referenční a tudíž uložená jako ukazatel. Proto jsou tyto operace definovány pomocí šablon a je využito specializace šablon k implementaci operací nad konkrétními typy. Zároveň je využito vlastnosti C++, kdy specializace šablony umožňuje definovat, že šablona se má provést pro typ s ukazatelem. Tato specializace ale funguje pouze pro struktury a třídy a nikoliv pro funkce. Proto jsou všechny operátory definovány jako součást struktury, která má definovanou šablonu a samotná metoda je nešablonová. Ukázka implementace takovéhoto operátoru:

```
template<typename T>
struct _comparison
{
      static bool equals(T left, T right)
      {
            return left == right;
```

```
	}
}

template<typename T>
struct _comparison<T*>
{
	static bool equals(T* left, T* right)
	{
		if (left == NULL && right == NULL)
			return true;
		if (left == NULL || right == NULL)
			return false;
		base_object* bl = (base_object*)left;
		base_object* br = (base_object*)right;
		return bl->equals(br);
	}
}
```

Tato ukázka reprezentuje operaci porovnávání. První část přidává operaci pro porovnávání dvou hodnot. Druhá část specializuje tuto operaci pro ukazatelové typy. Pro atomické typy jako je `int`, `float` nebo `boolean` se zavolá první varianta. Pro třídy, které jsou všechny reprezentovány, jako ukazatele se zavolá druhá varianta. Ve specializaci je také zřejmé využití třídy `base_object` pro implementaci samotné operace. Tuto implementaci je možné použít, protože všechny třídy a tedy všechny proměnné s ukazateli dědí z této třídy. Zároveň není nutné se starat o fakt, že by se jako parametry předaly rozdílné typy, protože typové odvození zajistí, že typy parametrů jsou ekvivalentní.

Funkčnost, kterou základní knihovna poskytuje je také převod hodnot na jejich textovou reprezentaci. Je zde však komplikace, která je reprezentována následujícím kódem:

```
print([1])
print(['1'])
print(1)
print('1')

# vytiskne

[1]
['1']
1
1
```

Když se tiskne řetězec přímo, nejsou u něj vidět uvozovky. Pokud se však řetězec vyskytuje jako část složitějšího typu jako je `list` nebo `tuple`, pak se zobrazuje s uvozovkami. Toto chování je způsobeno, tím, že každá třída jazyka Python má dvě metody pro její textovou reprezentaci: `__str__` a `__repr__`. Metoda `__str__` vrací textovou reprezentaci, která je vhodná pro zobrazení, kdežto metoda `__repr__` vrací interní reprezentaci. V případě typu `string` pak první metoda vrací pouze text a druhá vrací text v uvozovkách. Typ `list` pak interně volá metodu `__repr__` pro zjištění textové reprezentace hodnot v něm uložených. Volání obou metod je podobné ukázce porovnávání

výše. Funkce `_str<T>::invoke(T)` a `_repr<T>::invoke(T)` v základní knihovně jsou implementovány přímo pro atomické typy a mají šablonové specializace pro ukazatelové typy, tedy třídy, kdy dojde pouze k volání metody na `base_object`. Tuto metodu pak třídy implementují tak, aby jejich návratová hodnota byla ekvivalentní hodnotě, kterou vrací jazyk Python.

Implementace tříd `string` a `list` je přímočará. Pro interní reprezentaci je využito tříd `std::string` a `std::vector` ze standardní knihovny jazyka C++. Tyto třídy pak poskytují metody, které mají stejné jména a signatury jako ty v knihovně jazyka Python. Důvod, proč nejsou třídy ze základní knihovny využity přímo, je kvůli zjednodušení generování kódu. Všechna volání metod na třídách ve vygenerovaném kódu musí být stejného formátu, aby se zajistilo, že volání nad šablonovým typem bude validní jak v případě knihovních typů, jako je `string` a `list`, tak nad uživatelskými typy. Bylo by možné generovat přímý přístup k interní reprezentaci, ale to je stejný problém, který je popsaný v kapitole 4.4 .

Implementace třídy `tuple` je však mírně komplikovanější. Jako první verzi implementace bylo použito návrhu, kdy existovalo několik takovýchto tříd, které se lišily primárně šablonami. Existovaly tedy třídy `tuple2<T1, T2>`, `tuple3<T1, T2, T3>`, `tuple4<T1, T2, T3, T4>`, atd..., kde jednotlivé hodnoty byly uloženy jako atributy s typy `T1`, `T2`, atd… Tento návrh má však několik problémů. Prvním je omezení na maximální počet prvků v `tuple`, který je způsobený tím, že počet různých tříd v základní knihovně by byl omezený. To by nebylo tak problematické, protože pokud by se měl `tuple` využívat jako heterogenní struktura, pak návrh, kde je tuple využit pro uložení více jak 5 hodnot není správný. Lepším návrhem by bylo vytvoření samostatné třídy pro uložení hodnot. Druhým a větším problémem je však přístup k jednotlivým položkám. Díky tomu, že jednotlivé hodnoty jsou uloženy v samostatných atributech, není možné k nim přistupovat pomocí indexu. K přístupu k těmto položkám se pak využívaly metody `getItem1()`, `getItem2()`, atd… Přístup k hodnotě struktury přes index je ale nutný i pro typy `string` a `list` a tyto metody pak musely být implementovány i pro tyto třídy. Tento návrh pak navyšoval množství metod v celé základní knihovně a tím zhoršoval její udržovatelnost. Proto jsem tento návrh změnil na ten, který je v konečné verzi překladače.

Tento návrh je zobrazen na Diagram 3 - Popis struktury typu tuple v základní knihovně(str. 47) a využívá kombinace dědičnosti a šablon k tomu, aby uložil rozdílné typy hodnot do jednoho vektoru. Tento návrh nemá omezení na velikost a k hodnotám lze přistupovat přes číselný index bez nutnosti definovat novou metodu pro každý index. Naopak ale vyžaduje, aby se typ, jaký je obsažen na daném místě vektoru, zadal jako šablonový parametr při volání metody `getValue<T>`. Toto však není problém, protože tuto informaci zjistíme z odvození typů ve výrazu pro přístupu k indexu. Nevýhodou tohoto návrhu je pak možné zpomalení čtení hodnot, protože při čtení je hodnota uložena ve třídě, ke které se přistupuje nepřímo přes ukazatel a nikoliv jako atribut třídy samotné, jako tomu bylo u původního návrhu.

S přístupem k položkám přes index souvisí v základní knihovně také abstraktní třída `subscript_base<T>`. Samotná operace pro získání hodnoty je reprezentována funkcí `subscript<TParam, TResult>::read(int index)`. Šablona této operace vychází z toho, že tato operace musí fungovat na třech typech: `list`, `string`, `tuple` a ve všech případech je typ návratové hodnoty jiný. V případě typu `list` je typ hodnoty uložen v šabloně třídy `list`, pro typ `string` to je to typ `string`, protože tak je to definováno v jazyku Python. A u typu `tuple` typ hodnoty na hodnotě indexu a typ je nutné získat z typového odvození. Tato funkce pak má šablonovou

specializaci pro typ `tuple*`, kdy je volána přímo metoda `getValue`. Tato funkce má také šablonovou specializaci pro ukazatelové typy, kdy je hodnota reprezentující strukturu převedena na třídu `subscript_base<TResult>` a přes ní je získaná hodnota. Z výše popsaných typů hodnot je zřejmé, že třída `string` dědí z `subscript_base<string*>` a třída `list<T>` dědí z `subscript_base<T>`. Hlavní výhodou tohoto návrhu je jeho použitelnost i pro třídy generované překladačem. Tam stačí, aby třída ve zdrojovém kódu, která implementuje metodu `__getitem__`, byla převedena na třídu, která implementuje třídu `subscript_base`, a její metody.

Součástí základní knihovny je taky implementace správy paměti ve formě algoritmu pro garbage collector. Konkrétně se jedná o Boehm–Demers–Weiser garbage collector [16]. Implementace tohoto GC do C++ kódu nevyžaduje zásadní změny. Jedinou změnou je to, že všechny třídy musí dědit z tříd `gc` nebo `gc_cleanup`. Hlavní rozdíl mezi těmito třídami je fakt, že pro třídy, které dědí z `gc` není při odstranění z paměti volán destruktor. Nemůže tedy dojít u uvolnění paměti zabrané staticky alokovanými třídami, jako jsou `std::vector` a `std::string`, které jsou použity ve třídách `list` a `string`. `gc_cleanup` tento problém nemá. Zároveň, díky tomu, že všechny třídy dědí z `base_object`, je dostačující aby pouze tato třída dědila z `gc_cleanup`, čímž dojde k tomu, že všechny instance tříd v běžícím programu budou pod správou algoritmu pro garbage collector.

5.5 Shrnutí

Návrh a implementace překladače byla od začátku velice organický proces. I přesto, že jsem měl navrhnuto, jaké části jazyka bude překladač podporovat, je prakticky nemožné dopředu určit všechny způsoby, jak budou tyto části mezi sebou interagovat a jaké komplikace mohou vzniknout. Několikrát se mi tak stalo, že jsem přišel na specifický případ, se kterým aktuální návrh nepočítal a musel jsem tedy návrh změnit a znovu implementovat. Díky tomu jsem ale získal implementaci, která logicky odráží algoritmus typového odvození a je rozšiřitelná do budoucna.

6 Porovnání s existujícími překladači

Poslední kapitola se zabývá zhodnocením výsledného překladače a jeho srovnání s existujícími překladači. Prvním srovnáním je podpora jazyka, kde se zaměřím hlavně na podporu konstrukcí, které jsem si v na začátku práce vytyčil a které ostatní překladače nepodporují. Druhým srovnáním je spotřeba zdrojů počítače testovacích programů. Zde jde hlavně o rychlost vykonávání a spotřebu paměti. Pak je proveden rozbor výsledků a možné optimalizace, které by tyto výsledky zlepšily.

6.1 Podpora jazyka

V prvním případě je nutné zdůraznit, že návrh a implementace tohoto překladače se nesoustředila na maximální podporu jazyka, ale na podporu takových vlastností jazyka, které ostatní překladače nepodporují. Obecně řečeno podporuje překladač množinu, která hrubě odpovídá jazyku C. Jsou podporovány:

- základní datové typy a operace s nimi
- konstrukce pro řízení toku kódu `if`, `for` a `while`
- strukturovaní kódu pomocí funkcí
- homogení struktura ve formě typu `list`
- heterogení struktury ve formě tříd a typu tuple
- nepřímý přístup k datovým typům díky referenčnosti tříd
- základní vstup a výstup do souborů.

Hlavní vlastností, která je rozdílná od ostatních překladačů, je podpora pro typový polymorfismus. Ten lze vyjádřit jednoduchými kousky kódu:

```
def add(a, b):
    return a+b

add(1, 2)
add('a', 'b')
```

Druhý příklad:

```
class cls:
    def __init__(self, val):
        self.val = val

cls(1)
cls('a')
```

V prvním případě musí funkce fungovat nad jakýmkoliv typem, který podporuje operaci sčítání. V druhém případě může třída uchovávat hodnotu jakéhokoliv typu. Překladač RPython si s žádným z těchto případů neporadí, protože konkrétní typy funkcí a tříd odvozuje z jejich prvního použití. Překladač Cython s tímto kódem nemá problém, ale v jeho případě se jedná o fakt, že datové typy nekompiluje a pořád jsou dynamické. Překladač v této práci je však tento kód schopen validně otypovat a zkompilovat. Zároveň je díky statičnosti dosaženo lepšího výkonu než u programů přeložených překladačem Cython, který typy nekompiluje.

Další vlastností, která nevyplývá z podporovaných vlastností popsaných v kapitole 3.1, a která nijak nevyplývá z dynamického typování, je podpora rekurzivních datových typů. V dynamicky typovaném jazyku není nutné nijak speciálně identifikovat tyto typy, ale ve statickém typování je nutné určit, jestli je typ rekurzivní. V typové inferenci toto lze určit buď z typu samotného, nebo z jeho použití. A díky tomu, že v implementovaném překladači musí být všechny vlastnosti typu známy z jeho definice, je zřejmé, že půjde využít pouze první možnost. Zároveň je ale nutné použít speciální konstrukci, ze které typová inference určí, že se jedná o rekurzivní typ. Tato konstrukce vypadá následovně:

```
class list: # linked list
    def __init__(self, next, val):
        self.next = self
        self.next = next
        self.val = val

    def print(self):
        print(self.val)
        if self.next != None:
            self.next.print()

lst = list(list(list(None, 1), 2), 3)
lst.print()
```

Vyznačené řádky způsobí, že inferenční algoritmus odvodí dané atributy jako stejný typ, jako je definice samotné třídy. Všechny tři překladače byly schopny tento kód zkompilovat. V implementovaném překladači je toto možné díky speciální unifikaci typu reprezentujícího právě definovanou třídu a díky schopnosti volat funkce na třídě rekurzivně.

Výhodou kompilace je schopnost odhalit typové a logické chyby v programu. Implementovaný překladač má schopnost odhalit několik takovýchto chyb. První je problém popsaný v kapitole 3.4.5. To zajistí, že typování je v programu vždy kompletní. Druhý případ vyplývá z použití typu `void` pro návratový typ funkce, která nevrací žádnou hodnotu a generickému omezení na unifikaci s tímto typem. Toto znemožňuje kompilaci tohoto kódu:

```
def proc(x):
    print(x)

var = proc(1)
```

I přesto, že v jazyce Python toto nevadí, protože takováto funkce vrací hodnotu `None`. Ale pro programátora je možné, že se jedná o chybu kódu, protože je zřejmé, že v proměnné `var` bude po provedení přiřazení vždy hodnota `None`.

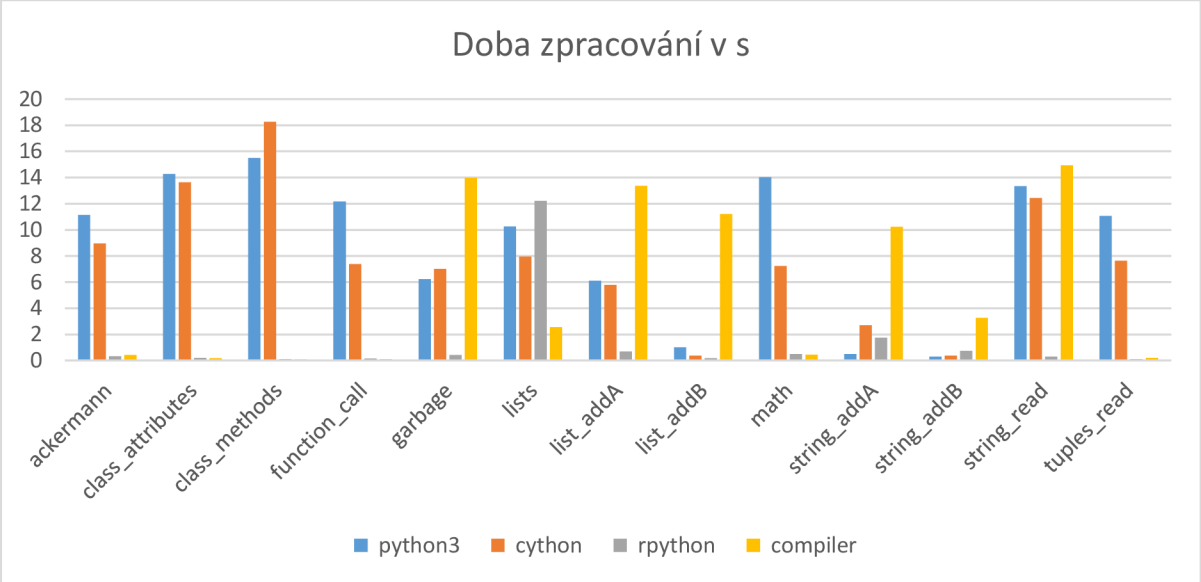
6.2 Spotřeba zdrojů

Výhodou statického typování oproti typování dynamickému je schopnost překladače optimalizovat rychlost provádění kódu a spotřebu paměti. V rámci testovaní, jsem tedy vytvořil sadu syntetických

testů, které testují výkon a spotřebu paměti. Tyto testy jsou obecně krátké kousky kódu, které se opakují v cyklu dostatečně krát, aby byl vidět případný rozdíl ve výkonu a spotřebě paměti mezi jednotlivými překladači. Tyto testy jsou:

- **ackermann** – testování rekurzivního volání funkce
- **class_attributes** – čtení a zápis do atributů třídy
- **class_methods** – volání metody třídy
- **garbage** – vytváření struktury objektů
- **lists** – vkládání a přístup k prvkům v seznamu
- **list_addA** – sčítání seznamů
- **list_addB** – sčítání seznamů, použit operátor +=
- **math** – matematické operátory
- **function_call** – volání funkce
- **string_addA** – sčítání/konkatenace řetězců
- **string_addB** – sčítání/konkatenace řetězcí, použit operátor +=
- **string_read** – čtení znaků řetězce
- **tuples_read** – čtení prvků tuple

6.2.1 Rychlost provádění

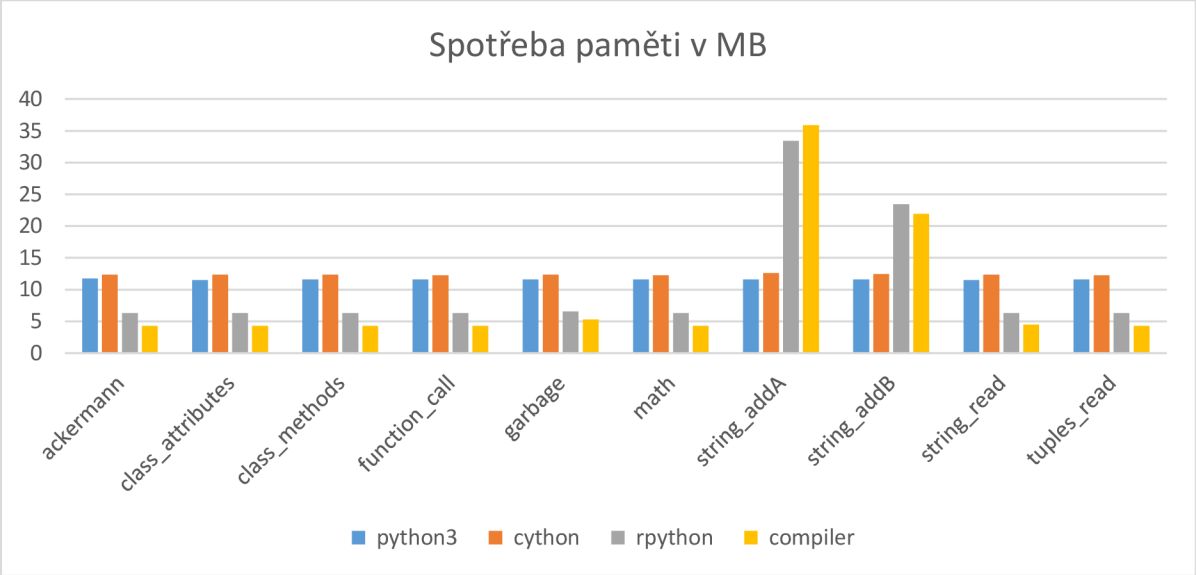


Z tohoto grafu lze vyčíst několik faktů o výhodě statického typového systému oproti dynamickému. Rychlost kódu, generovaného překladačem s typovou inferencí, jako je překladač RPython a implementovaný překladač, je řádově vyšší než dynamicky typovaný kód prováděný interpretem jazyka nebo generovaný překladačem Cython. Je zde ale zásadní problém a tím je správa pamětí. Je zřejmé, že pouze můj překladač používá pro všechnu správu paměti garbage collector. To je vidět v testech *garbage*, *list_add* a *string_add*. V jejichž případě dochází k vytváření velkého množství nových objektů. V případě testů string_add je ale interpret Python3 lepší než překladač RPython. To je nejspíše díky speciální optimalizaci, která využívá immutability typu string a při konkatenaci řetězců jednoduše nový řetězec odkazuje na oba předchozí, místo aby docházelo k jejich kopírování do nové paměti.

Zajímavostí je také srovnání výkonu mezi testy *list_addA* a *list_addB*. V případě varianty A je použito výrazu `x = x + y`, kdežto v případě varianty B je použito příkazu `x += y`. Interpret Python3 a ostatní překladače jsou schopni případ varianty B interpretovat jako přidání položek seznamu `y` do seznamu `x` a nedochází tak k vytvoření nové instance seznamu a kopírování dvou původních do tohoto seznamu, jako je tomu u varianty A. Ale v případě implementovaného překladače k takovéto optimalizaci nedochází, protože operátor += je rozvinut do plného tvaru příkazu přiřazení a výrazu pro součet a je tedy ekvivalentní s variantou A. Toto chování by šlo implementovat i v případě překladače této práce, vyžadovalo by to ale zkomplikování základní knihovny.

Jak bylo řečeno v kapitole 4.4, tak překladač jazyka C++ je schopen dostatečně optimalizovat vygenerovaný kód, který obsahuje velké množství volání šablonových funkcí v místech, kde by bylo možné vygenerovat jednoduchou operaci. Nejlépe to jde vidět v případě testu *math*, který je právě situací, kde se takovýto kód vyskytuje. A z výsledků lze poznat, že volání šablonových funkcí nijak nezpomalilo rychlost provádění přeloženého testu.

6.2.2 Spotřeba paměti



Na grafu nejsou vidět testy *list* a *list_add*. V jejich případě garbage collector zabral všechnu dostupnou pamět počítače.

Z grafu spotřeby paměti je možné vyčíst, že v případě, kdy se neoperuje s velkým množstvím instancí objektů, je spotřeba paměti menší než v případě standardního interpretu jazyka. Avšak pokud dochází k vytváření nových objektů, dojde k situaci, která je stejná pro všechny programy s garbage collectorem a to taková, že GC zabere velké množství paměti a tu začne uvolňovat, až když stroji paměť dochází.

Optimalizace správy paměti však nebyla zaměřením této práce. Pokud by bylo nutné správu paměti optimalizovat, bylo by nutné využít ve vygenerovaném kódu a základní knihovně počítání referencí a garbage collector by zůstal pouze pro detekci cyklických referencí. Podobný způsob využívají i ostatní interprety a překladače. Další možností optimalizace je zlepšení práce s pamětí u řetězců. Díky jejich neměnnosti je možné docílit toho, že různé operace nad řetězci nebudou vytvářet nové instance řetězců a tím i zabírat paměť, ale využije se existujících instancí a referencí na ně.

Rychlost překladu je také jednou z možných metrik, kterou jsem ale nesledoval, protože nebyly k dispozici dostatečně velké zdrojové kódy, které by implementovaný překladač podporoval. Teoreticky by ale bylo možné říct, že překladače z rodiny Hindley-Milner budou rychlejší a budou vyžadovat méně paměti, protože nevyžadují, aby byl celý zdrojový kód, včetně modulů nahrán v paměti pro odvozování. Pro hrubé porovnání, překlad každého z testovaných případů trval překladači RPython přes 100 sekund, kdežto překlad implementovaným překladačem trval řádově sekundy.

6.3 Shrnutí

V porovnání s existujícími překladači se mě v rámci této práce podařilo implementovat překladač, který podporuje typový polymorfismus, který ostatní překladače nepodporují.

Výkon dopadl podle očekávání. Staticky typovaný kód je možné optimalizovat více než dynamicky typovaný a výkon tedy bude vyšší. Problémy ale nastávají u správy paměti, která musí být komplexní a obsahovat řadu optimalizací, aby se vyrovnala existujícím implementacím.

7 Závěr

V rámci této práce jsem navrhnul a implementoval překladač podmnožiny jazyka Python do jazyka C++. Vybraná podmnožina je použitelná pro elementární programování a zahrnuje základní datové typy: *int*, *float*, *boolean*, *string*, *list*, *tuple* a operace s nimi, jako jsou matematické operátory a porovnávací operátory. Základní příkazy pro větvení kódu *if*, *while* a *for*. Umožňuje strukturování kódu do funkcí, včetně funkcí rekurzivních; tvorbu uživatelských typů ve formě tříd s atributy a metodami, ale bez dědičnosti; třídami a metodami pro základní vstup a výstup do konzole a do souborů; a jednoduchou základní knihovnou. Hlavní výhodou oproti existujícím překladačům s typovým odvozením je podpora pro parametrické metody a třídy a schopnost jejich použití na různých typech, pokud typy odpovídají omezením parametrů odvozeným z funkce nebo třídy. Vybraná podmnožina pak omezuje dynamické chování jazyka a znemožňuje např. změnu atributů tříd mimo jejich definici. Překladač je pak navržen, aby bylo možné ho rozšířit o další typy, omezení typů a vlastnosti jazyka Python.

Součástí překladače je algoritmus typového odvozování s podobnými vlastnostmi jako Hindley-Milner, na rozdíl od existujících překladačů, které jsou založeny na Cartesian Product algoritmu. Algoritmus odvozuje typy z výrazů a příkazů v rámci bloků funkcí a tříd. Z těchto bloků odvozuje nejobecnější typy, které mohou být parametrické, protože funkce nemusí nutně obsahovat dostatek informací k tomu, aby bylo možné odvodit konkrétní typy. Tyto parametry mohou mít omezení, které omezují množinu možných konkrétních typů, které lze za konkrétní parametr dosadit. Konstrukce jazyka Python se pak převedou na ekvivalentní konstrukce v jazyce C++ rozšířené o odvozené typy. Parametrické typy, funkce a třídy jsou reprezentovány jako šablony. Součástí překladače je také základní knihovna, která obsahuje implementace základních tříd a funkcí, jejichž chování je ekvivalentní jejich protějškům v jazyce Python a zároveň jsou navrženy tak, aby byly použitelné z vygenerovaného kódu překladače. Pro správu paměti je využit Boehm garbage collector, který je přímo napojen na jazyk C++ a nevyžaduje žádné složitější zásahy do struktury vygenerovaného programu nebo základní knihovny.

Výsledný přeložený program je řádově rychlejší než jeho interpretace ve standardním interpretu, ale pouze v případě, že nedochází k vytváření nových instancí objektů. V tom případě se projeví mnohem lepší optimalizace správy paměti interpretu oproti naivnímu použití Boehm GC a přeložený program se výrazně zpomalí a naroste jeho spotřeba paměti.

7.1 Rozšíření podporované podmnožiny

Finální verze implementovaného překladače nepodporuje celý jazyk Python. Množství funkcionality chybí, hlavně z důvodů jejich přílišné složitosti nebo faktu, že je možné je implementovat v rámci podporované podmnožiny jazyka. Překladač a hlavně algoritmus pro typovou inferenci jsou ale navrženy tak, že je možné chybějící vlastnosti jazyka doplnit bez větších zásahů do již implementovaných vlastností.

Podpora datového typu `dictionary` a výrazů `slices` není ve finální verzi překladače implementována, ale návrh s nimi počítá. Implementace by vyžadovala rozšíření omezení na čtení indexu o typovou instanci reprezentující typ klíče pro `dictionary` a nové omezení na index slice pro `slices`. Spolu s tím by bylo nutné rozšířit základní knihovnu.

Podpora dědičnosti tříd by vyžadovala velké rozšíření typového odvozování a generování kódu. Je zde primárně problém změny generických parametrů mezi třídami v hierarchii, kdy potomek může přidávat, odebírat nebo konkretizovat generické parametry svého předka. Toto je problematické jak v odvozování definice tříd, tak unifikace jejich typů při použití. Problémem je také vícenásobná dědičnost, kde může být problém rozdílného řešení problému diamantu v jazycích C++ a Python.

Jazyk Python je také jazykem funkcionálním. Podpora pro funkcionální konstrukce jako ukládání ukazatelů funkcí a metod a uzávěry funkcí není problematická pro odvozování typů, ale je mnohem větší problém při generování cílového kódu v jazyce C++. Problémem, který by vzniknul v současné verzi generování funkcí, by byl převod funkcí na jejich ukazatel v případě přiřazení do proměnné nebo atributu. Možným řešením by bylo generovat pro každou funkci a metodu vlastní třídu a funkce ukládat jako lokální proměnné a atributy tříd, tak jak to dělá samotný jazyk Python. Tím by se vyřešil i problém s uzávěry lokálních funkcí, protože proměnné v uzávěrech by se uložily jako atributy třídy reprezentující danou metodu.

Důležitou součástí jazyka Python, který implementovaný překladač nepodporuje, jsou moduly. Na rozdíl od překladačů využívajících Cartesian Product algoritmus, kde jsou moduly a knihovny importovány přímo jako jejich zdrojový kód a zahrnuty do typového odvození, je možné s nimi v implementovaném překladači pracovat jako se samostatně kompilovanými jednotkami. Z každého modulu by vznikly hlavičkový a zdrojový soubor C++ a soubor s metadaty. Tento soubor by obsahoval typování všech metod a tříd v daném modulu. Toto typování by bylo serializací tříd definic jednotlivých typů. Generování rozdílného hlavičkového a zdrojového souboru není problém, protože jsou známy jak definice, tak deklarace funkcí a tříd. Tímto by bylo dosaženo i mnohem vyšší rychlosti překladu než u ostatních překladačů, protože není nutné všechny moduly udržovat v paměti najednou, ale je možné je zpracovávat postupně. A také díky tomu, že jeden modul stačí přeložit pouze jednou.

Současná verze překladače využívá naivní způsob správy paměti pomocí garbage collectoru. Správu paměti by bylo možné optimalizovat pomocí systému počítání referencí a detektoru cyklů. Tím by se zamezilo zbytečné práci garbage collectoru, která je hlavním problémem výkonu přeložených aplikací. Toto rozšíření nevyžaduje změnu algoritmu pro typové odvozování a je spíše rozšířením generování C++ kódu.

8 Diagramy

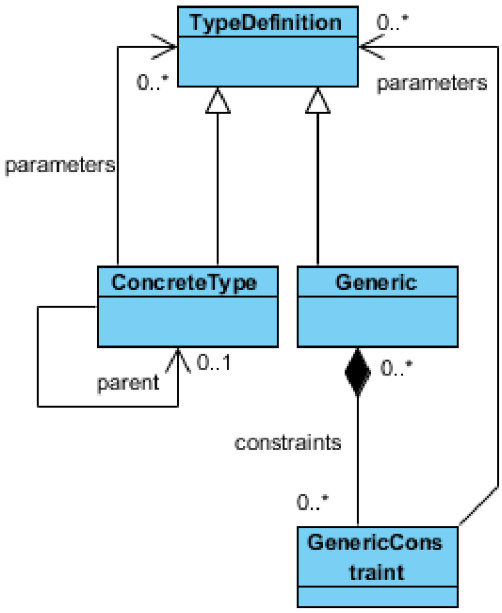


Diagram 1 - Obecný popis struktury typů

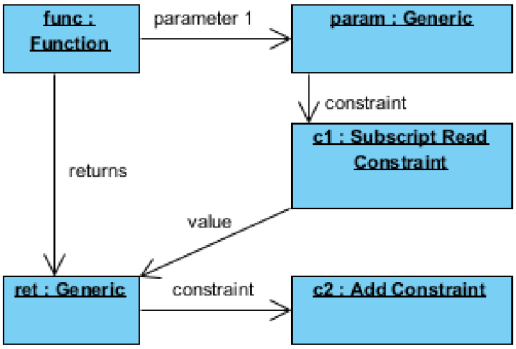


Diagram 2 - Popis typu konkrétní funkce

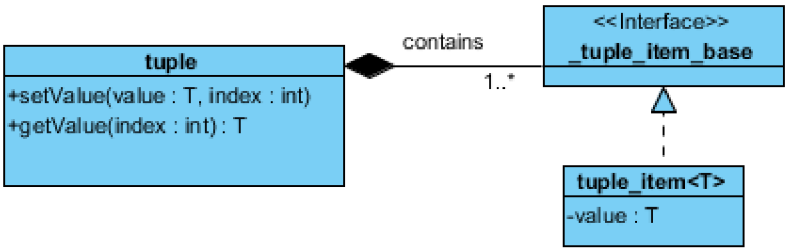


Diagram 3 - Popis struktury typu tuple v základní knihovně

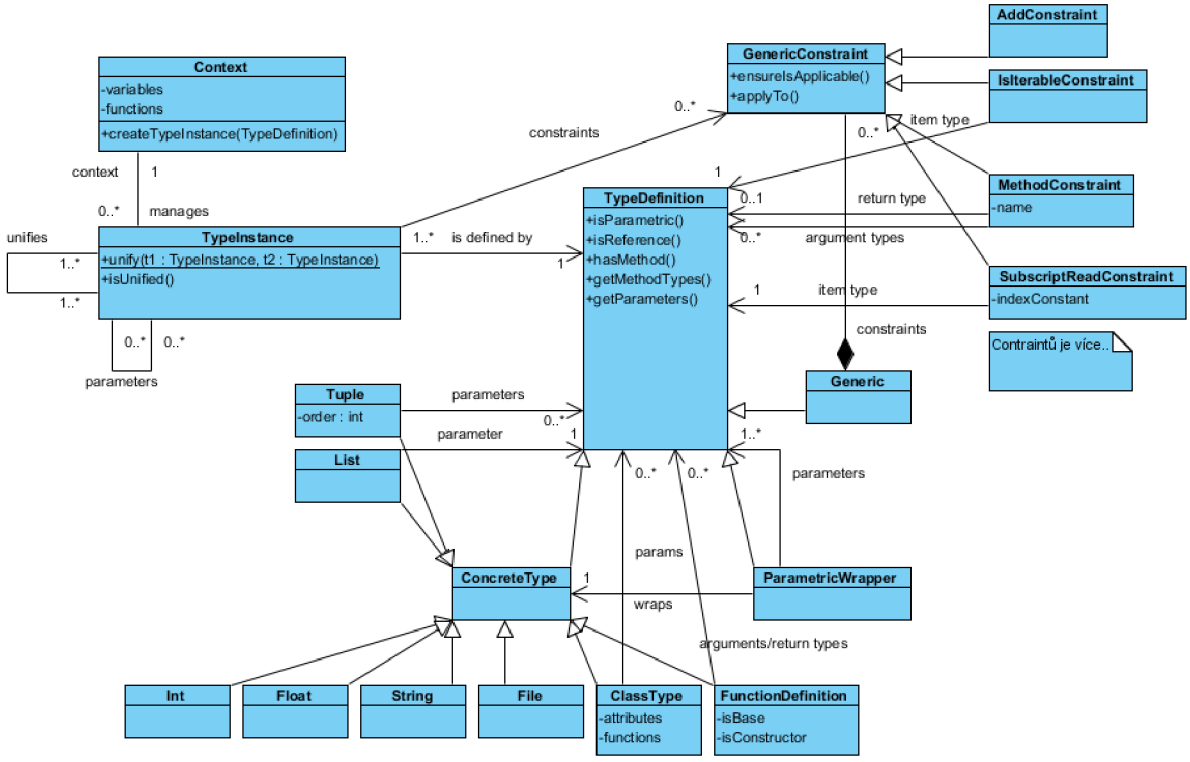


Diagram 4 - Struktura tříd implementace typového systému a odvozování

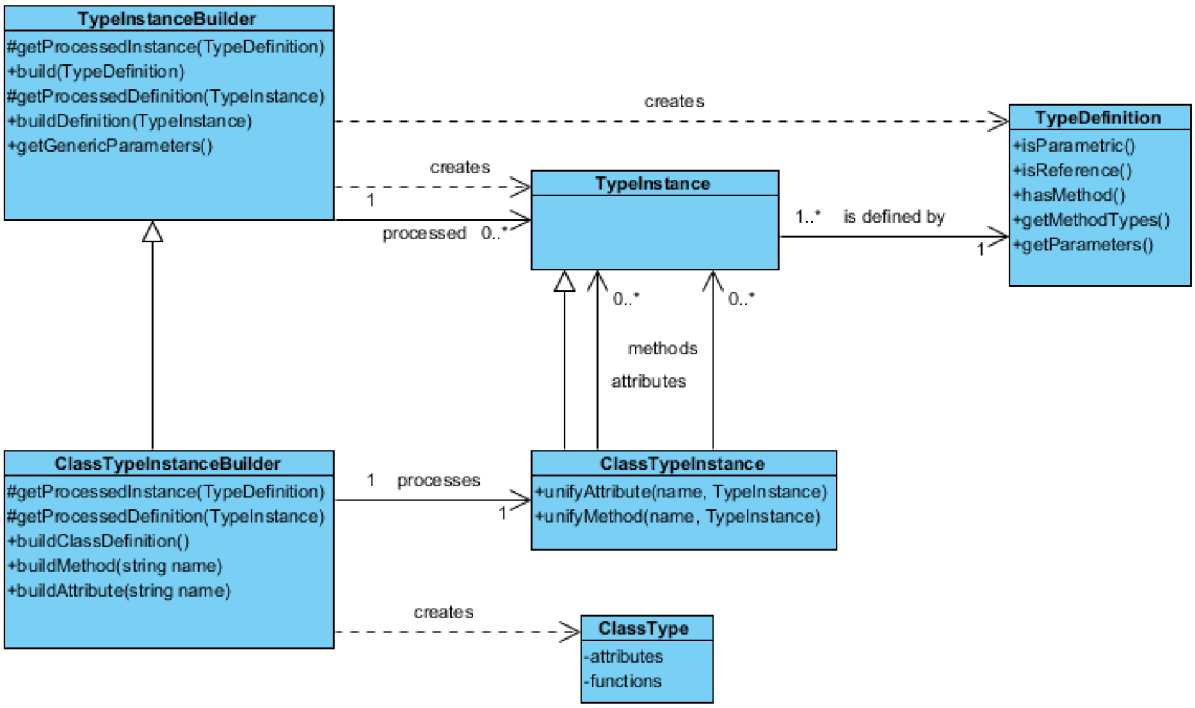


Diagram 5 - Struktura typů okolo InstanceBuilder

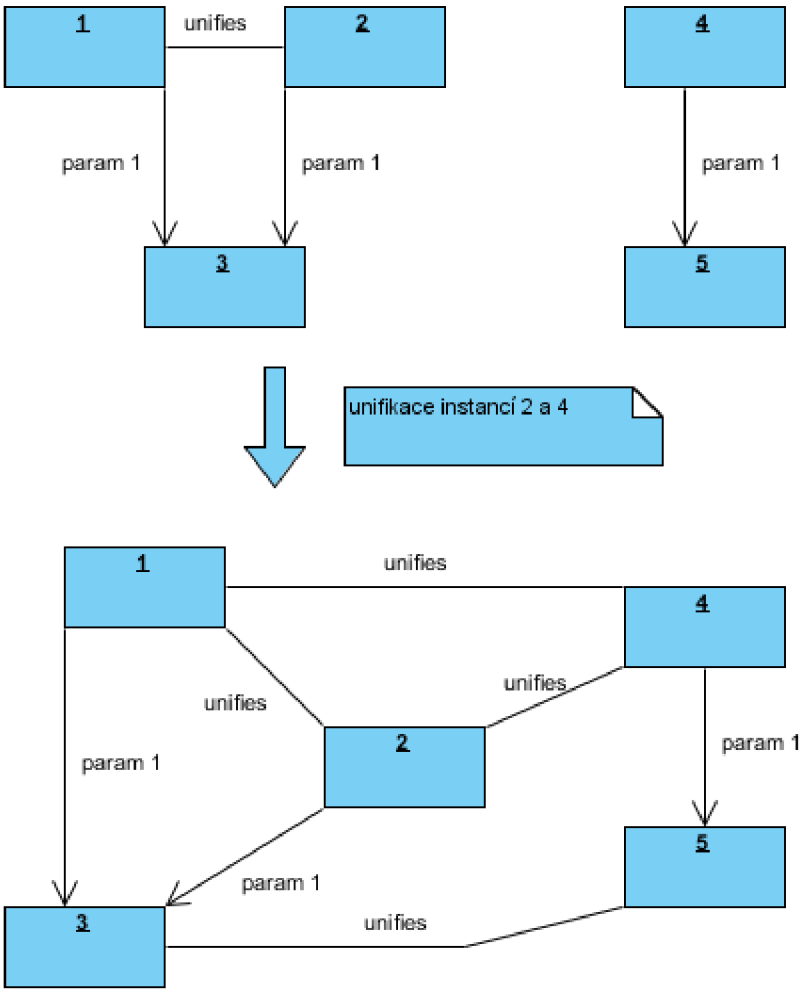


Diagram 6 - Vizualizace unifikace typových instancí

Ukázka unifikace typových instancí. Všechny objekty na tomto diagramu jsou různé typové instance s různými vazbami. Před unifikací jsou unifikovány pouze instance 1 a 2. Pak následuje unifikace instancí 2 a 4. Při tom dojde k tomu, že se sjednotí množiny `unifies` obou instancí. A všem instancím v této množině se zpětně tato množina nastaví na `unifies`. Tím dojde k uzavření ekvivalenční třídy relace `unifies` mezi těmito instancemi. Zároveň, díky tomu, že jsou typové instance parametrické, jsou unifikovány i jejich parametry 3 a 5.

9 Reference

1. Domovská stránka jazyka Python [online]. [cit. 2014-05-24]. Dostupné z: http://www.python.org/
2. Dokumentace jazyka Python: Special method names. [online]. [cit. 2014-05-24]. Dostupné z: http://docs.python.org/2/reference/datamodel.html#specialnames
3. Domovská stránka překladače Nuitka [online]. Dostupné také z: http://nuitka.net/
4. Domovská stránka překladače Cython [online]. Dostupné také z: http://cython.org/
5. Dokumentace interpretu PyPy: Jazyk RPython. [online]. Dostupné také z: http://doc.pypy.org/en/latest/coding-guide.html#id1
6. Domovska stranka překladače ShedSkin [online]. Dostupné také z: https://code.google.com/p/shedskin/
7. HINDLEY, Roger. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*. 1969, č. 146.
8. MILNER, Robin. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*. 1978, č. 17.
9. JONES, Mark P. From Hindley-Milner Types to First-Class Structures. In: *In Proceedings of the Haskell Workshop*. 1995, s. 115-36.
10. PALSBERG, Jens a Michael I. SCHWARTZBACH. Object-Oriented Type Inference. In: *ACM Conference on Object-Oriented Programming*. 1991, s. 146-61.
11. AGESEN, Ole. The Cartesian Product Algorithm, Simple and Precise Type Inference of Parametric Polymorphism. In: *European Conference*. Denmark: ECOOP, 1995.
12. Dokumentace jazyka Python : List Comprehensions [online]. Dostupné také z: https://docs.python.org/3.3/tutorial/datastructures.html#list-comprehensions
13. Haskell official site. *Haskell Type Classes* [online]. 2000. Dostupné také z: http://www.haskell.org/tutorial/classes.html
14. KILDALL, Gary A. A unified approach to global program optimization. In: *In Conference Record of the ACM Symposium on Principles of Programming Languages*. New York: ACM, 1973, s. 194-206.
15. Dokumentace jazyka Python: Abstract Syntax Trees. [online]. Dostupné také z: http://docs.python.org/3.3/library/ast.html
16. Domovská stránka knihovny Boehm GC [online]. Dostupné také z: http://www.hboehm.info/gc/

Seznam příloh

Součástí práce je CD s aktuální verzí překladače, která zvládá typové odvození a překlad vlastností jazyka probrané v tomto dokumentu. Návod k použití je v souboru readme nebo příkazem `--help`. Hlavní spouštěcím souborem je `source/main.py`. Součástí překladače jsou i příklady ve složce `cases`, které reprezentují podporovanou podmnožinu jazyka. Ve složce `error_cases` jsou kódy, které jsou podporovány, ale způsobí chybu překladu. Ve složce `performance` jsou testovací příklady použité pro testování výkonu a spotřeby paměti.

Spuštěním souboru `run_tests.sh` se spustí porovnání výstupů příkladů v `cases` mezi implementovaným překladačem a interpretem `python3`.

Spuštěním souboru run_performance.py v interpretu jazyka Python se spustí testování výkonu příkladů ve složce `performance`. K tomuto je potřeba instalace překladače cython a zdrojové kódy projektu PyPy a správné nastavení cesty k překladači `rpython` v těchto zdrojových kódech