VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MEZIJAZYKOVÝ PŘEKLADAČ C#-JAVASCRIPT PRO DOTVVM

C#-JAVASCRIPT TRANSPILER FOR DOTVVM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MICHAL MRNUŠTÍK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN PLUSKAL**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20958/2017/xmrnus01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Mrnuštík Michal**
Obor: Informační technologie
Téma: **Mezijazykový překladač C#-JavaScript pro DotVVM**
C#-JavaScript Transpiler for DotVVM
Kategorie: Překladače

Pokyny:

1. Seznamte se s jazyky C#, JavaScript a webovým frameworkem DotVVM.
2. Zvažte možnosti použití .NET Compiler Platform (Roslyn) pro zpracování zdrojových souborů po konzultaci s vedoucím.
3. Navrhněte řešení pro kompilaci tříd z jazyka C# do JavaScriptu za použití zvoleného přístupu.
4. Implementujte navržené řešení.
5. Vytvořte demonstrační aplikaci pro výsledné řešení v rámci webového frameworku DotVVM.

Literatura:

- Albahari, J., & Albahari, B. (2015), *C# 6.0 in a Nutshell (6th Edition)*, O'Reilly Media, Inc.
- Strauss, D. (2017). *C# 7 and .NET Core Cookbook*.
- Price, M. (2017). *C# 7 and .NET Core modern cross-platform development : create powerful cross-platform applications using C# 7, .NET Core, and Visual Studio 2017 or Visual Studio Code*.

Pro udělení zápočtu za první semestr je požadováno:

- body 1, 2 a 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Pluskal Jan, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem této práce vyřešit problém překladu serverového kódu napsaného v jazyce C# do klientského kódu jazyka JavaScript, tak aby bylo možné jej spouštět v prohlížeči. A navržené řešení poté implementovat. Následně je cílem integrovat toto řešení do frameworku DotVVM a napsat aplikaci demonstrující této technologie v praxi.

Abstract

Aim of this thesis is to solve issue of transpiling server code written in C# to client-side JavaScript code, so it could be run in the browser. And then implementing the proposed solution. Next goal is to integrate this solution into framework DotVVM and develop an application demonstrating this techonology.

Klíčová slova

ASP.NET, C#, Javascript, Roslyn, překladač, DotVVM, webové aplikace, .NET Framework, .NET Core, Model-View-ViewModel

Keywords

ASP.NET, C#, Javascript, Roslyn, transpiler, DotVVM, web applications, .NET Framework, .NET Core, Model-View-ViewModel

Citace

MRNUŠTÍK, Michal. *Mezijazykový překladač C#-JavaScript pro DotVVM*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

Mezijazykový překladač C#-JavaScript pro DotVVM

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Pluskala. Další informace mi poskytli... Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Mrnuštík
14. května 2018

Poděkování

Tímto bych chtěl poděkovat Honzovi Puskalovi za poskytnutí vedení a podpory při této práci. Také bych chtěl poděkovat všem lidem vyvíjejícím framework DotVVM za jejich pomoc a rady, když jsem byl ztracen. A také bych chtěl poděkovat Terezce Košťálové za to, že se mnou měla pevné nervy a byla mi oporou.

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Kapitola 1

Úvod

Webové aplikace se v poslední době stávají čím dál populárnější platformou k vývoji. Tyto aplikace se většinou skládají ze serverové části a klientské části.

Serverová část je obvykle programem, který běží na nějakém počítači připojeném k internetu. Tato část aplikace většinou provádí komunikaci s databází, do které jsou ukládána data, a pro klientskou část exponuje rozhraní, kterým klientská část aplikace může tato data získávat, modifikovat nebo mazat. Pokud jde o použité technologie, při psaní serverové části si můžeme vybrat mezi řadou jazyků (Python, PHP, C#, Java, ...) a knihoven(Django, Netter, DotVVM, Spring, ...).

Klientská část programu je potom datově reprezentována pomocí HTML, nastylována pomocí CSS (případně SASS nebo LESS) a programována v JavaScriptu (případně TypeScriptu). JavaScript se zde používá převážně pro komunikaci se serverovou částí a úpravou dat na základě této komunikace, případně ještě pro validaci a úpravu HTML stromu.

Velká část tohoto kódu se poté v rámci klientské části aplikace opakuje. Data se serializují, poté se odešlou na server, kde jsou následně deserializována a provádí se s nimi vyžádané operace. Potom se výsledek operace opět serializuje a odesílá klientské části programu, která ho deserializuje a upraví na jeho základě vizuální reprezentaci dat. Právě kvůli tomuto opakovanému, často se objevujícímu, procesu vznikl framework DotVVM, který umožňuje specifikovat datovou část programu pouze v rámci serverové části a její automatické propsání do klientské části. Při komunikaci se serverovou částí se data opět sama doplní do připravených vlastností objektů.

Toto řešení ulehčuje práci vývojářů, ale znamená to, že při každé změně v rámci datové části musí dojít ke komunikaci se serverem. Cílem této práce je vytvořit řešení, které by umožňovalo překlad serverového kódu napsaného v jednom jazyce (C#) do jazyka druhého (JavaScript), který bude spustitelný v klientské části programu.

V rámci podporovaných vlastností by se měly nacházet základní programátorské konstrukce, jako jsou cykly, podmínky, binární operace, deklarace lokálních proměnných, atd. A také by mělo výsledné řešení podporovat alespoň základní knihovní konstrukce, například práce se seznamy (přidání prvku, odebrání prvku, vymazání seznamu,...), výpis do konzole, atd.

Obsah práce

Ve druhé kapitole jsou shrnuty všechny informace o frameworku DotVVM od jeho vnitřního fungování po to, jakým způsobem v něm psát aplikace. Jsou tam též popsány jednotlivé vlastnosti frameworku, jako je syntaktický zápis uživatelského rozhrání, životní cykly viewmodelů a práce s validací a autorizací, Také je tam představeno, jak tento framework pracuje s překladem do JavaScriptu v současnosti a jaké by bylo ideální řešení, kterého bych chtěl touto prací dosáhnout.

Ve třetí kapitole je popsán obecný model kompilátor s jeho fázemi. Následuje popis diagnostických informací, které framework Roslyn implementuje z hlediska jednotlivých fází. Nachází se tam ukázka konstrukce syntaktického stromu, práce se symbolickými a operačními informacemi, práce s ukládáním zkompilovaného projektu na disk a pokročilé analytické nástroje sledující datový a kontrolní tok kódu.

Ve čtvrté kapitole se nachází analýza překladových problémů této práce. Ať už se jedná o problémy jazykové, kde jsou sledovány rozdíly mezi oběma jazyky, nebo o problémy překladové, kde se nachází obecný popis problémů spojených s překladem. Z hlediska překladových problémů je tam popsán problém určení přeložitelnosti a problém práce se vstupním a výstupním jazykem. Z hlediska jazykových problémů je tam popsán hlavně problém sledující rozdílné chování datových typů.

V páté kapitole se nachází stručný popis implementační části této práce. Tato kapitola je rozdělena do dvou sekcí. V první sekci je popsán krok po kroku celý proces překladu včetně implementačních detailů. U každého kroku je popsán jeho vstup a výstup a jeho implementace V druhé sekci se objevuje popis napojení přeložených tříd do frameworku DotVVM.

Kapitola 2

DotVVM

DotVVM je open-souce webový framework založený na platformě ASP.NET umožňující rychlé psaní webových aplikací za použití architektonického návrhového vzoru Model-View-ViewModel (dále už jen MVVM) [3].

V rámci této kapitoly si popíšeme:

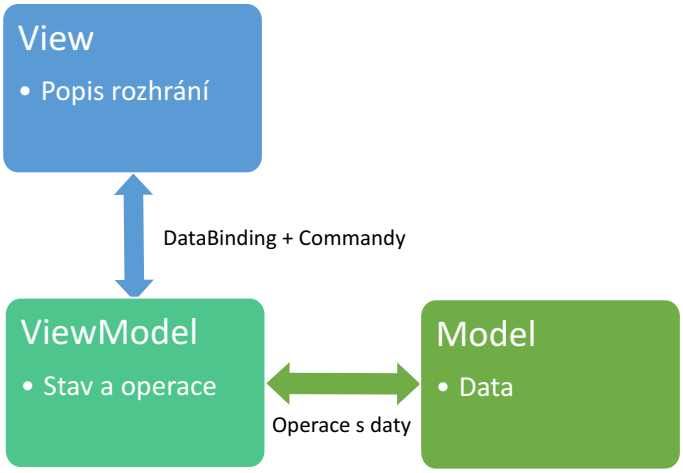
- způsob fungování architektonického návrhového vzoru MVVM;
- jaké technologie DotVVM používá na pozadí;
- jakým způsobem se v DotVVM zapisují View;
- jak se v prostředí DotVVM programuje a jak funguje ViewModel;
- jak pomocí DotVVM vyřešit validaci dat a autorizaci uživatelů;
- jak tento framework pracuje s překlady do JavaScriptu v současnosti a jaké by bylo ideální řešení, kterého bychom chtěli dosáhnout.

2.1 Architektonický vzor Model-View-ViewModel

Při tvorbě aplikací poskytujících uživatelské rozhraní se často setkáváme s přístupem, kdy vývojář tvořící aplikaci umístí veškerý kód definující chování programu přímo do kódu uživatelského rozhraní. Tento přístup není ideální hned z několika důvodů. Za prvé takto dochází k "bobtnání"kódu uživatelského rozhraní, což vede k problému s udržitelností a rozšiřitelností programu. Za druhé to znepříjemňuje práci více vývojářů najednou na stejném rozhraní, protože při každé změně dochází ke konfliktům [4].

Když se na vývoj softwaru podíváme z architektonického hlediska, můžeme každý program rozdělit na tři části:

- uživatelské rozhraní (View), které má na starost vzhled aplikace
- datová vrstva (Model), který říká jaká data chceme zobrazit
- spojení mezi těmito dvěma vrstvami, které definuje chování aplikace (ViewModel)



Obrázek 2.1: Schéma fungování MVVM. View popisuje vzhled uživatelského rozhraní a pomocí bindingů a commandů je napojeno na ViewModel. ViewModel zase popisuje stav aplikace a poskytuje reakce na příkazy, na jejichž základě aktualizuje Model.

2.1.1 View

View definuje strukturu, rozložení a vzhled uživatelského rozhraní programu. Říká, co kde a jakým způsobem se má vykreslit. View se může skládat z několika uživatelských rozhraní vnořených do sebe (např. můžeme definovat menu aplikace, které je pro všechna rozhraní stejné a to mezi nimi sdílet).

Každé View může mít svůj vlastní ViewModel, případně může ViewModel podědit po svém předkovi. View získává data z ViewModelu pomocí bindingu na proměnné ViewModelu. Ve většině MVVM frameworků zároveň můžeme u bindingu nastavit směr, ve kterém proběhne navázání vlastnosti ViewModel. Směr se dá definovat jako obousměrný (z View do ViewModelu a naopak), jednosměrný ve vybraném směru (pouze z View do ViewModel nebo obráceně) případně se dá definovat jako „onetime“ (binding se načte pouze jednou a poté už není napojený na ViewModel).

Zároveň View provádí navázání událostí, které mohou být vyvolané uživatelem (například stisk tlačítka), na metody v rámci ViewModelu, což se provádí pomocí návrhového vzoru zvaného Command.

2.1.2 Model

Model představuje datovou vrstvu programu. Definuje data (například ve formě DTO objektů[1]), která chceme uživateli zobrazit, a zároveň popisuje, jakým způsobem se získávají

[1]Data Transfer Object (zkráceně DTO) je návrhový vzor používaný pro objekty, které jsou určeny pro přenos datových informací mezi jejich zdrojem (databáze, soubory na disku, ...) a příjemcem. Tyto objekty by měly být serializovatelné a neměly by obsahovat žádnou vnitřní logiku. Více o návrhovém vzoru DTO lze nalézt v [5, strana 401]

(například pomocí návrhového vzoru Facade[2]). Zdroj těchto dat může být jakýkoliv, od databáze přes soubor na disku po hodnoty nasbírané senzory v chytré domácnosti.

2.1.3 ViewModel

ViewModel se chová jako prostředník mezi View a Modelem. Jeho odpovědností je implementace aplikační logiky. Aplikační logika je pouze popis způsobu reakce na jednotlivé uživatelské akce (např. po kliknutí na tlačítko „Uložit“ proveď odeslání dat do modelu).

Typický sled chování u MVVM aplikací je následující, ViewModel z Modelu načte data a případně na nich provede nějaké transformace. Data z ViewModelu se do View dostanou pomocí již zmíněných bindingů (v Sekce 2.1.1).

V případě, že uživatel provede nějakou interakci, dochází k vyvolání odpovídajícího Commandu v rámci ViewModelu, jenž na tuto událost zareaguje. V případě, že dojde ke změně vlastností, které jsou navázány do View, provede se jejich aktualizace v rámci View. Pro informování View o změně se většinou využívá vyvolání nějaké události (v rámci frameworku WPF je například pro tuto situaci definovaná událost `PropertyChanged` [7]).

2.2 Realizace MVVM v DotVVM

DotVVM pro realizaci návrhového vzoru MVVM na klientské straně používá Javascriptovou opensource knihovnu Knockout.js (dále jen Knockout). Jedná se o knihovnu vyvíjenou zaměstnancem společnosti Microsoft Stevem Sandersonem, která umožňuje pomocí HTML5 `data-*` atributů definovat bindingy na Javascriptové ViewModely. V případě, že dojde ke změně hodnot v rámci ViewModelu, Knockout aktualizuje Document Object Model (DOM) HTML tak, aby bindingy odpovídaly hodnotám ve ViewModelu [8].

```
<p>First name: <input data-bind="value: firstName" /></p>
<p>Last name: <input data-bind="value: lastName" /></p>
<h2>Hello, <span data-bind="text: fullName"> </span>!</h2>
<script>
    // Here's my data model
    var ViewModel = function(first, last) {
        this.firstName = ko.observable(first);
        this.lastName = ko.observable(last);

        this.fullName = ko.computed(function() {
            return this.firstName() + " " + this.lastName();
        }, this);
    };

    // This makes Knockout get to work
    ko.applyBindings(new ViewModel("Planet", "Earth"));
</script>
```

Ukázka kódu 2.1: Příklad použití Knockout knihovny

Vzhledem k tomu, že v prostředí ASP.NET se s Javascriptem na serverové části nepracuje, jsou ViewModely v DotVVM na serverové části reprezentovány pomocí tříd napsaných

[2]Facade je návrhový vzor definovaný Gang Of Four používaný jako rozhraní pro komunikaci s více různými objekty na jedno místo. Více o návrhovém vzoru Facade se můžete dočíst v [6, strana 192].

v jazyce C#, které se pro přenos mezi serverem a klientem serializují do formátu JSON. Zároveň se na serveru generuje odpovídající HTML kód s bindingy pro Knockout, který se následně odešle klientovi (ukázky kódu pro DotVVM ekvivaletního k ukázce kódu 2.1 najdete v ukázce kódu 2.2 a ukázce kódu 2.3).

```
namespace ProjectName.ViewModels
{
    public class ViewModel
    {
        public string FirstName { get; set; } = "Planet";
        public string LastName { get; set; } = "Earth";
        public string FullName => FirstName + LastName;
    }
}
```

Ukázka kódu 2.2: Zápis ekvivalentního ViewModelu ukázce kódu 2.1 v DotVVM.

```
@viewModel ProjectName.ViewModels.ViewModel, ProjectName

<p>First name: <dot:TextBox Text="{value: FirstName}"/> </p>
<p>Last name: <dot:TextBox Text="{value: LastName}"/> </p>
Hello, {{value: FullName}}!
```

Ukázka kódu 2.3: Zápis ekvivalentního View ukázce kódu 2.1 v DotVVM.

Na to, jak to funguje pod uvnitř a na co všechno lze DotVVM použít, se podíváme v následujících sekcích.

2.2.1 View

Pro definici uživatelských rozhraní poskytuje DotVVM vlastní formát pod názvem DotHTML. Jedná se o formát HTML, který byl upravený tak, aby umožnil napojení na ViewModel, definovat napojení hodnot v rámci View na vlastnosti ViewModelu a zároveň dovnitř umisťovat prvky uživatelského rozhraní [9].

Definice uživatelského rozhraní lze provádět pomocí technologií, na které jsme při tvorbě webových stránek zvyklí (HTML, Cascading Style Sheets). Zároveň však DotVVM přichází se sadou standardních prvků webového uživatelského rozhraní (dále zmiňované jako kontrolky), které zároveň umožňují provést binding na základní vlastnosti těchto prvků. Tyto prvky se následně při odeslání na klienta překládají do svých ekvivalentů v HTML s navázanými bindingy pomocí knihovny Knockout.

Direktivy

Na začátek každého DotHTML souboru se umisťují direktivy popisující speciální vlastnosti daného rozhraní. Každá direktiva začíná znakem zavináče, poté následuje její typ a hodnota dané direktivy.

Pro napojení DotHTML na ViewModel se používá direktiva `@viewModel`, která definuje ViewModel, jenž bude použit v rámci daného View (použití direktivy můžete nalézt v ukázce kódu 2.3).

Syntaxe vkládání prvků uživatelského rozhraní

Prvky uživatelského rozhraní se v rámci DotHTML zapisují stejně, jako každé jiné HTML elementy. Jediným rozdílem je, že používají jméno elementu definované pomocí prefixu a názvu prvku. Elementy definující daný prvek mohou mít stejně jako HTML elementy obsah (viz. Ukázka kódu 2.4). Zároveň však mohou být definovány bez obsahu (viz. Ukázka kódu 2.5).

```
<prefix:JmenoPrvku Atribut1="Hodnota" ...>
    <!-- Telo kontrolky -->
</prefix:JmenoPrvku>
```

Ukázka kódu 2.4: Syntaxe zápisu kontrolky s obsahem.

```
<prefix:JmenoPrvku Atribut1="Hodnota" ... />
```

Ukázka kódu 2.5: Syntaxe zápisu kontrolky bez obsahu.

Syntaxe bindingů

Zápis bindingů ve formátu DotHTML lze provést dvěma způsoby. Tento způsob se liší podle toho, zda je binding umístěn jako hodnota atributu elementu, nebo je umístěn v těle elementu [10].

Pro zápis bindingů jako hodnoty atributu elementu se binding umisťuje mezi složené závorky, je definován pomocí typu bindingu a výrazu popisujícího hodnotu, na kterou se váže (příklad naleznete v ukázce kódu 2.6).

```
<prefix:JmenoPrvku Text="{TYP_BINDINGU: VyrazBindingu}" />
```

Ukázka kódu 2.6: Syntaxe zápisu bindingů jako hodnoty atributu kontrolky.

Zápis bindingu uvnitř těla elementu je podobný zápisu bindingu jako hodnoty atributu. Jediným rozdílem je to, že uvození a ukončení direktivy bindingu je provedeno dvěma složenými závorkami místo jedné (příklad naleznete v ukázce kódu 2.7). Tímto zápisem se dá zapsat pouze binding typu value (popsán v Sekce 2.2.1).

```
<html>
    <body>
        {{value: Title}}
    </body>
</html>
```

Ukázka kódu 2.7: Syntaxe zápisu bindingů uvnitř elementu.

Typy bindingů

Jak už jsem zmínil (viz. Sekce 2.2.1), DotVVM nabízí více druhů bindingů, z nichž každý má svůj specifický kontext použití.

Základní typy v prostředí DotVVM jsou následující:

- Value binding;
- Command binding;

- Static Command binding;
- Resource binding.

Value binding je druh bindingu u nějž, jak už samotný název napovídá, jde o navázání hodnoty ViewModelu do View. Tuto hodnotu můžeme napojit buď na hodnotu atributu nebo ji použít jako čistý text. Při přenosu na klienta se tento binding překládá na normální Knockoutový binding.

DotVVM umožňuje v rámci tohoto bindingu navazovat více než jen pouhou hodnotu vlastnosti ViewModelu. Můžeme nad touto vlastností vytvářet i různé výrazy (ukázku využití této vlastnosti můžete vidět v ukázce kódu 2.8, seznam možných tvarů výrazů naleznete zde [11]).

```
<div Visible="{value: Items.Count == 0}">
    You seem to have selected no items.
</div>
```

Ukázka kódu 2.8: Příklad možnosti použití výrazu ve value bindingu.

Command binding je direktiva, která umožňuje napojení události vyvolané nějakou kontrolkou na metodu ve ViewModelu. Navázaná metoda může mít návratový typ `void`, nebo `Task`[3].

Command binding se na úrovni javascriptu překládá do volání funkce `dotvvm.postBack()` definované v souboru `DotVVM.js`, která provede serializaci aktuálního stavu ViewModelu do JSON formátu, který následně odešle na server, kde se ViewModel deserializuje a vyvolá se navázaná metoda ViewModelu [13].

```
<dot:Button Click="{command: Submit()}" Text="Submit Form" />
```

Ukázka kódu 2.9: Příklad použití command bindingu, který vyvolá metodu `Submit()` ve ViewModelu.

Při každém zavolání Command bindingu se provádí serializace ViewModelu a jeho odeslání na server, což může být zbytečně komplikovaná a zatěžující operace pro vykonání jednoduchého příkazu. Proto DotVVM podporuje typ bindingu pojmenovaný jako **Static Command**.

Tento typ bindingu umožňuje událost v rámci View navázat na zavolání jakékoliv metody v rámci ViewModelu s jakýmikoliv argumenty, která může vracet cokoliv. Pro tuto metodu jsou pouze dvě podmínky: její parametry a návratové hodnota musí být serializovatelné do JSON formátu a metoda musí být označena pomocí atributu `[AllowStaticCommand]`[4]. Static Command zároveň umožňuje přiřazení návratové hodnoty získané z volané metody do vlastnosti ViewModelu [14].

Při vyvolání Static Commandu se na server odesílají pouze parametry definované v rámci bindingu a cesta k metodě, která má být takto zavolána. Ukázku použití Static Com-

[3] `Task` je třída představená ve verzi .NET Frameworku 4.0. Tato třída je použita pro vysokoúrovňovou abstrakci nad konkurentní operací, která může i nemusí běžet na jiném vlákně. Dále se používají pro implementaci asynchronních operací v rámci jazyku C# [12]

[4] Atributy jsou speciální konstrukce jazyku C#, které umožňují přidávat další informace do metadat metod, tříd a vlastností. Atributem se může stát jakákoliv třída, jež dědí z třídy `System.Attribute` a jejíž název končí slovem Attribute. Takto vytvořená třída se následně aplikuje na jednotlivé prvky jazyka skrze její název v hranatých závorkách před deklarací daného prvku (např. `[Obsolete] public class Foo {...}`). Toto označení prvků se poté používá například pro označení, že daná třída je zastaralá a neměla by se při vývoji používat [12].

mandu můžete vidět v ukázce kódu 2.10 a ukázku obsahu requestu při vyvolání normálního Commandu a statického Commandu vykonávajícího stejnou věc můžete vidět na Obrázku 2.2.

```
<dot:Button Text="Something" Click="{staticCommand: SomeProperty = MyClass.
    MyMethod(Name)}" />
```

Ukázka kódu 2.10: Příklad použití static command bindingu, který vyvolá statickou metodu `MyClass.MyMethod(Name)` a výsledek uloží do vlastnosti ViewModelu nazvané `SomeProperty`.

```
Bytes Sent: 2,736 (headers:2,074; body:662)
Bytes Received: 275 (headers:233; body:42)
```

(a)

```
Bytes Sent: 2,546 (headers:2,043; body:503)
Bytes Received: 549 (headers:337; body:212)
```

(b)

Obrázek 2.2: Zde můžete vidět porovnání počtu přenesených dat při provedení příkazu z ukázky kódu 2.10 zachycených pomocí programu Fiddler. V obou případech jsme měli jednoduchý ViewModel obsahující pouze jednu položku typu `string` a oba příkazy provedly pouze konkatenaci tohoto řetězce s jiným řetězcem. V případě (a) byl použit Static Command zatímco v případě (b) jsme použili klasický Command binding. Je vidět, že už při takto malé velikosti ViewModelu se objem přenášených dat výrazně liší, a to hlavně v odpovědích (42 bytů dat při Static Commandu oproti 212 bytům při běžném Command bindingu).

Static Command zároveň umožňuje vyhodnocení jednoduchých výrazů s vlastnostmi ViewModelu pouze na úrovni klienta bez nutnosti komunikace se serverem. Podporované výrazy pro tuto funkcionalitu jsou stejné jako výrazy, které podporuje Value binding. Tyto výrazy jsou při získávání View ze serveru přeloženy do JavaScriptu a při vyvolání události, na kterou jsou navázány, dojde k jejich provedení na straně klienta.

V DotVVM ve verzi 2.0 došlo také k rozšíření View o možnost deklarování služeb pomocí speciální direktivy `@service`, které se následně používají v direktivách Static Commandů. Toto umožňuje rozšíření Static Command bindingu o napojení na Dependency Injection[5]. DotVVM získá z DI kontejneru instanci objektu deklarovaného v již zmíněné direktivě `@service` a na ní následně provede zavolání zmiňované metody. Tím, že je tento objekt získávaný z DI kontejneru, dochází rovnou k jeho naplnění dalšími potřebnými službami (napojení na databázi atd.) [16]. Způsob použití takovéto služby můžete vidět v ukázce kódu 2.11.

[5]Dependency Injection je princip používaný v rámci aplikací pro zajištění vkládání závislostí mezi jednotlivými komponentami aplikace, aniž by na sebe tyto komponenty přímo držely referenci [15].

```
@service menuFacade = ClassLibrary1.MenuFacade

<dot:Repeater DataSource="{value: Menu}" WrapperTagName="ul">
    <li>
        <dot:LinkButton Click="{staticCommand: Children = menuFacade.
            LoadMenu(_this)}" Text="{value: Title}" />
    </li>
    <dot:Repeater DataSource="{value: Children}" WrapperTagName="ul">
        <li>{{value: Title}}</li>
    </dot:Repeater>
</dot:Repeater>
```

Ukázka kódu 2.11: Příklad použití Static Command služeb pro automatické vykreslení dynamického menu použitého pro načítání položek podmenu.

Resource binding je speciální druh bindingu, který umožňuje načítání textů z .resources souborů[6].

2.2.2 ViewModel

Jak už bylo zmíněno v Sekce 2.1.3, ViewModel je vrstva aplikace v návrhovém vzoru MVVM popisující stav a chování aplikace. DotVVM umožňuje použít jako ViewModel jakýkoliv objekt, který je serializovatelný do formátu JSON. To znamená, že objekt je použitelný jako ViewModel pokud obsahuje:

- základní datové typy jazyka C# (`string`, `int`, `bool` `Guid`,...);
- nullovatelné varianty základních datových typů[7];
- struktury datového typu `DateTime` pro reprezentaci času [8];
- výčtové datové typy;
- kolekce objektů splňující tyto podmínky;
- vnořené objekty splňující tyto podmínky;

DotVVM také poskytuje bázovou třídu `DotvvmViewModelBase`, kterou lze použít pro inicializaci ViewModelu. Tato třída poskytuje všem poděděným třídám vlastnost `Context`, skrze níž je možné přistupovat k parametrům z URL adresy, informacím o přihlášených uživatelích, HTTP hlavičkám, informaci o typu požadavku, atd. Dále tato třída poskytuje

[6] .resources soubory jsou v rámci jazyku C# používané pro uložení potenciálně lokalizovatelných dat. Jedná se o binární soubory, které jsou zpravidla vytvořeny při kompilaci aplikace a to ze souborů .resx nástrojem resgen nebo přímo aplikací Visual Studio [12].

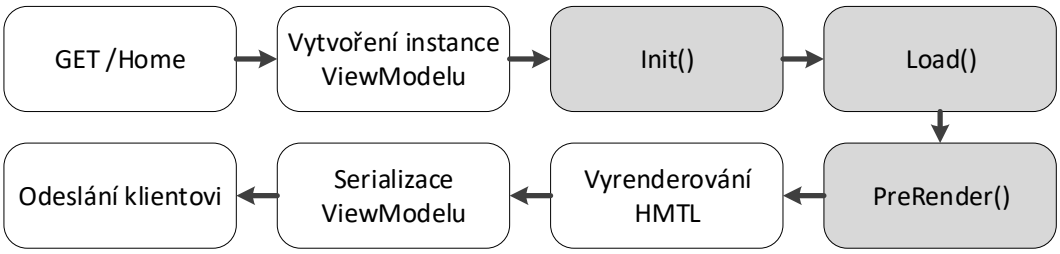
[7] Nullovatelné varianty datových typů jsou speciální vlastností jazyku C#, která byla přidána ve verzi .NET Frameworku 2.0. Jedná se možnost obalit hodnotové datové typy do generické struktury `Nullable<Type>`, díky níž je možné s hodnotovým datovým typem pracovat jako s referenčním datovým typem (přiřadit hodnotu null). Syntakticky se takto obohacený typ značít pomocí otazníku přidaného na konec jména datového typu při deklaraci proměnné (například `int? promena;`) [12].

[8] V současnosti (24. 1. 2018) se ve framewoku DotVVM vyskytuje bug, kdy při přenosu časového záznamu z klienta na server dochází ke konverzi časové informace na zónu UTC, ale při přenosu zpět na klienta se tato časová zóna nemění zpět na časovou zónu na klientském zařízení.

virtuální metody[9] `Init`, `Load` a `PreRender`, které lze použít k incializaci použitého ViewModelu (načtení dat z databáze, získání parametrů z URL, získání informací o přihlášeném uživateli, ...).

ViewModel v DotVVM podléhá dvěma životním cyklům. Inicializační životní cyklus probíhá, když dojde k prvotnímu načtení stránky, a aktualizační životní cyklu nastává, když dojde k vyvolání Commandu z View[10].

Inicializační životní cyklus nastává ve chvíli, kdy klient přistupuje na stránku poprvé, to znamená, že dojde k zavolání metody HTTP metody GET nad adresou dané stránky. DotVVM informaci o dotazu zachytí a pomocí URL identifikuje, které View je touto URL adresováno. Poté dojde k vytvoření instance ViewModelu náležícího danému View (určeno direktivou ViewModel, která byla popsána v Sekce 2.2.1) a na něm provede zavolání již zmíněných metod třídy `DotvvmViewModelBase`. Poté se provede vyrenderování HTML kódu z DotHTML popisujícího dané View, serializaci ViewModelu do formátu JSON a jeho vložení do stránky. Tato stránka je poté odeslána klientovi. Schéma popisující toto chování můžete vidět v Obrázek 2.3.

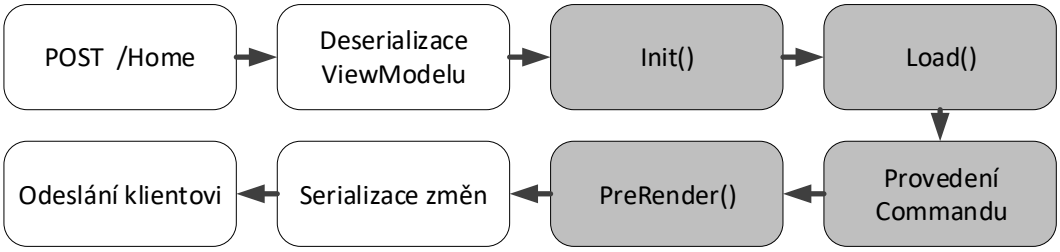


Obrázek 2.3: Diagram chování DotVVM při prvotním načtení stránky. Modře jsou znázorněny vnitřní procesy DotVVM, zatímco šedě jsou znázorněny prvky uživatelského kódu.

Druhý případ nastává ve chvíli, kdy má klient stránku již načtenou a nastala situace vyžadující zavolání Commandu (např. uživatel kliknul na tlačítko odesílající formulář). V tuto chvíli se na klientské straně provede serializace ViewModelu v aktuálním stavu. Následně se zaobalí ViewModel do metadat obsahujících informaci o vyvolaném Commandu a tento celek se odešle HTTP metodou POST na server. Zde se provede zjištění, o které View se jedná a jaká metoda má být zavolána. Provede se deserializace ViewModelu a poté dojde k zavolání metody ViewModelu odpovídající danému Commandu. Poté DotVVM zjistí, které položky ViewModelu změnily svou hodnotu a provede jejich odeslání zpět klientovi. Zde dojde ke zpracování odpovědi a nastavení odpovídajících vlastností na základě přijatých změn. Schéma popisující toto chování můžete vidět v Obrázek 2.4.

[9] Virtuální metoda v jazyce C# je metoda označená klíčovým `virtual`. Toto označení znamená, že potomci dané třídy mohou redefinovat chování dané metody [12].

[10] Tato činnost se v rámci DotVVM také označuje jako Postback [17].



Obrázek 2.4: Diagram chování DotVVM při vyvolání příkazu klientem. Modře jsou znázorněny vnitřní procesy DotVVM, zatímco šedě jsou znázorněny prvky uživatelského kódu.

2.2.3 Validace

Framework DotVVM v rámci funkcionality také nabízí možnost validace dat přicházejících od klienta. Mezi tyto možnosti patří jak jednoduchá kontrola přítomnosti povinných údajů, tak komplexnější kontroly vyjádřené přímo pomocí kódu. Kontrola validity dat probíhá jak v klientské, tak v serverové části DotVVM. Při jednoduchých validačních požadavcích (kontrola přítomnosti povinného údaje, rozsah hodnoty, splnění regulárních výrazů a platnost formátů) probíhá validace přímo v Javascriptové části DotVVM v prohlížeči a při porušení validačních pravidel ani nedochází k zaslání dat na server. V případě komplexnějších validací dochází k odeslání dat na server, ale ještě před provedením požadované metody se provede kontrola validity a pokud jsou data obsažena v modelu nevalidní, metoda se vůbec nezavolá [18].

Jednoduché validační požadavky se definují pomocí atributů umístěných nad vlastnost, jichž se tento požadavek týká. Podporované jsou následující atributy: `[Required]` pro povinné údaje, `[RegularExpression]` pro regulární výrazy, `[Range]` pro rozsahy hodnot a `[DotvvmEnforceClientFormat]` pro nutnost dodržení specifického formátu číselných a časových hodnot. Zároveň je možné pomocí atributů definovat i vlastní validační pravidla implementací rozhraní `IValidationAttribute`, takto definovaná pravidla ovšem neproběhnou v rámci Javascriptové části DotVVM. Způsob použití této možnosti můžete vidět v ukázce kódu 2.12.

```
public class RegisterViewModel
{
    [Required]
    public string Email { get; set; }

    [Required(ErrorMessage = "The password is required!")]
    public string Password { get; set; }

    [Range(4, 110, ErrorMessage = "The age has to be real value.")]
    public object Age;
}
```

Ukázka kódu 2.12: Příklad použití atributů pro validaci hodnot ViewModelu.

Složitá validační pravidla zahrnující více proměnných (např. kontrola, zda datum konce události nebylo nastaveno před její začátek) se dají implementovat přímo v rámci View-

Modelů implementací rozhraní `IValidatableObject`. Způsob použití této možnosti můžete vidět v ukázce kódu 2.13.

```
public class EventViewModel : IValidatableObject
{
    public DateTime BeginDate { get; set; }

    public DateTime EndDate { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext
        validationContext)
    {
        if (BeginDate >= EndDate)
        {
            yield return new ValidationResult(
                "The begin date of the appointment must be lower than the
                    end date.",
                new[] { nameof(BeginDate) } // path of the property
            );
        }
    }
}
```

Ukázka kódu 2.13: Příklad implementace rozhraní `IValidatableObject` využité pro porovnání zda konečné datum nebylo zadané před počáteční datum.

2.2.4 Autorizace

Z hlediska autentizace uživatelů DotVVM nenabízí žádnou funkcionalitu. Informace o tom jaký uživatel je přihlášen, jaké má role případně jaká práva mu náleží získává DotVVM přímo z informací poskytovaných ASP.NET (konkrétně se jedná o vlastnost `User` třídy `HttpContext`).

Pokud jde o autorizaci, zde DotVVM nabízí hned několik možných nastavení, která určují zda je uživatel přihlášen, případně zda má uživatel práva k vykonání určité akce. Toto řešení se opět provádí pomocí atributů.

Konkrétně se jedná o atribut `[Authorize]`. Tento atribut můžeme umístit nad třídu ViewModelu, ke kterému chceme neautorizovanému uživateli zakázat přístup, případně jej můžeme umístit nad metodu Commandu, který neautorizovaný uživatel nesmí vykonat. Dále je možné v rámci tohoto atributu nastavit role, které musí být uživateli přiřazeny, aby byl pro přístup tomuto ViewModelu (nebo Commandu) autorizován [19].

Příklad použití autorizace v rámci ViewModelu můžete vidět v ukázce kódu 2.14 a příklad použití autorizace s definováním uživatelské role na Commandu můžete vidět v ukázce kódu 2.15.

```
[Authorize]
public class TopSecretViewModel
{
}
```

Ukázka kódu 2.14: Příklad použití atributu `[Authorize]` pro zamezení přístupu uživatelů k ViewModelu.

```
public class RegularViewModel
{
    [Authorize(Roles = new [] { "Administrator" })]
    public Task TopSecretMethod() { ... }
}
```

Ukázka kódu 2.15: Příklad použití atributu `[Authorize]` pro zamezení vyvolání příkazu uživatelem, který nemá roli Administrator.

2.3 Překlad do JavaScriptu

V této kapitole si popíšeme současné řešení používané v rámci DotVVM pro překlad výrazů do JavaScriptu a ukážeme, proč dané řešení není ideální. Poté si představíme ideální řešení, které by mělo být řešením této práce.

2.3.1 Současné řešení

DotVVM v současnosti umožňuje překlad do JavaScriptu ve velmi omezené míře a to pouze na úrovni výrazů. Pokud při zápisu DataBindingu použiji výraz, DotVVM provede přeložení tohoto výrazu do JavaScriptu. Zároveň DotVVM umožňuje provést registraci určité metody a poskytnout pro ni implementaci rozhraní `IJavascriptMethodTranslator`, která vrací třídu `JsExpression`, jež reprezentuje daný výraz v Javascriptu.

Současné řešení používá při překladu třídy vycházející z prostředí .NET pro zápis výrazů v jazyce C#, a to konkrétně třídy dědící z abstraktní třídy `System.Linq.Expressions.Expression`, která poskytuje komplexní řešení pro práci s jakýmikoliv výrazy poskytovanými na úrovni jazyka. Po sestavení reprezentace výrazu bindingu pomocí třídy `Expression` se provede překlad pomocí třídy `JavascriptTranslationVisitor` na odpovídající třídy vycházející z `JsExpression`.

Toto řešení není zdaleka ideální. Za prvé toto řešení nemá podporu pro bloky kódu, tím pádem nelze vytvářet složitější konstrukce jazyka (jako cykly, složité podmínky, ...). Za druhé se řešení provádí pouze v případě výrazů použitých v rámci bindingů. Příklad jednoho z přeložitelných výrazů můžete vidět v ukázkách kódu 2.16 a 2.17.

```
<dot:TextBox Text="{value: Items.Count == 0 ? "Prazdny" : "Plny"}" />
```

Ukázka kódu 2.16: Použití bindingu obsahujícího výraz přeložitelný na klientský Javascript.

```
<input type="text" data-bind="value: (Items() && Items().length) == 0 ? "
    Prazdny" : "Plny"">
```

Ukázka kódu 2.17: Přeložený Javascript z ukázky kódu 2.16. Přistupování k poli `Items` jako k funkci je způsobeno použitím Knockoutu jako nástroje pro tvorbu MVVM prostředí.

2.3.2 Ideální řešení

Ideálním řešení celé této situace by bylo, kdyby vývojář mohl označit metody, které by chtěl v rámci jeho viewmodelů mít přeloženy do Javascriptu. Framework by při kompilaci aplikace tyto metody analyzoval z hlediska jejich přeložitelnosti a, pokud by to bylo možné, provedl by jejich přeložení. Ve chvíli přístupu uživatelů k aplikaci by došlo k identifikaci, které

metody byly přeloženy a ty by místo provedení postbacku byly provedeny v Javascriptové části aplikace.

Dále by bylo důležité toto řešení navrhnout tak, aby bylo rozšiřitelné o registrování knihovních vlastností a metod, aby vývojář mohl přidávat k dispozici i překlady metod, jejichž zdrojové kódy nejsou už v době kompilace k dispozici (byly už zkompilovány a aplikací jsou pouze referencovány).

Kapitola 3

Roslyn

Programátoři se na kompilátory často dívají jako na černé skříňky. Na vstup se vloží kód a na výstupu vznikne spustitelný binární soubor. Kompilátor ovšem při procesu kompilace získává spoustu informací o kódu, se kterým právě pracují. Tyto informace jsou ihned zahozeny poté, co je kompilace dokončena [20]. Microsoft se při tvorbě projektu Roslyn zaměřil na to, aby většina informací, které kompilátor v průběhu procesu kompilace získá, byla dostupné každému, kdo by s nimi chtěl pracovat. Tato data se poté dají využít ať už pro statickou analýzu kódu, nebo také v rámci integrovaných vývojových prostředích pro zvýšení pohodlí samotného programování.

V rámci této kapitoly si nejprve popíšeme, do jakých funkčních celků můžeme rozdělit kompilátor a následně se podíváme, co nám v jednotlivých částech nabízí API frameworku Roslyn.

3.1 Struktura kompilátoru

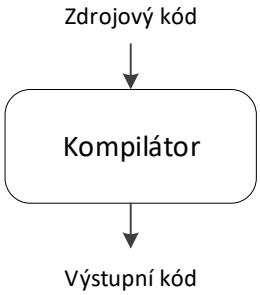
Kompilátor si můžeme velice jednoduše představit jako nástroj, který na vstupu dostane text ve zdrojovém jazyce a na jeho výstupu nám vrátí kód ve výstupním jazyce. Případně pokud námi napsaný kód nebyl správný, kompilátor nám nahlásí, co bylo špatně a kde se to nachází. Dále nám ještě kompilátor může poskytnout varování o úsecích kódu, které jsou v rámci naší implementace nějakým způsobem zavádějící [21]. Schéma obecného kompilátoru můžete vidět na Obrázek 3.1.

Při svém běhu prochází kompilátor několika fázemi, kde každá má jasně definovaný vstup a výstup. Z praktického hlediska mohou být některé části spojeny dohromady. V průběhu kompilace se zároveň mimo tyto fáze nachází tabulka symbolů, která nese veškeré informace o zdrojovém kódu a je k ní přistupováno všemi fázemi [21]. Jednotlivé fáze a jejich vstupy a výstupy můžete vidět na Obrázek 3.2.

3.1.1 Fáze kompilátoru

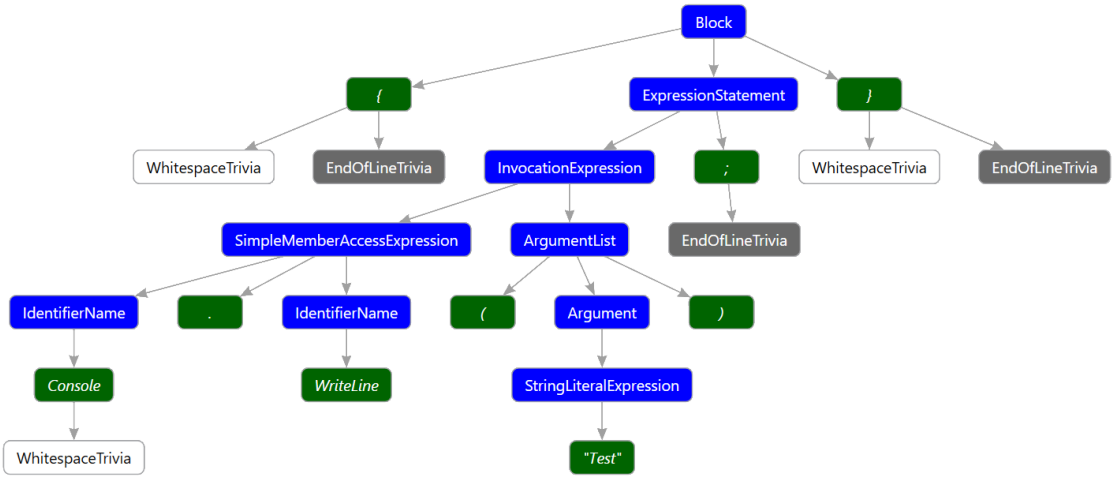
První fází je *lexikální analýza*, ve které dochází k načítání jednotlivých znaků zdrojového textu a jejich seskupení do tokenů, kde každý token je nositelem informace, o jaký lexém vstupního jazyka se jedná. V případě, že vstupní znaky neodpovídají žádnému lexému vstupního jazyka, skončí tato fáze s chybou [22].

Po lexikální analýze přichází na řadu *syntaktická analýza*, která provádí kontrolu, zda vstupní kód splňuje gramatická pravidla vstupního jazyka, na jejich základě sestaví syntaktický strom. V případě, že pořadí tokenů neodpovídá žádnému gramatickému pravidlu



Obrázek 3.1: Schéma obecného kompilátoru. Na vstup přichází kód ve zdrojovém jazyce a na výstupu se nachází výstup ve výstupním (často binárním) jazyce.

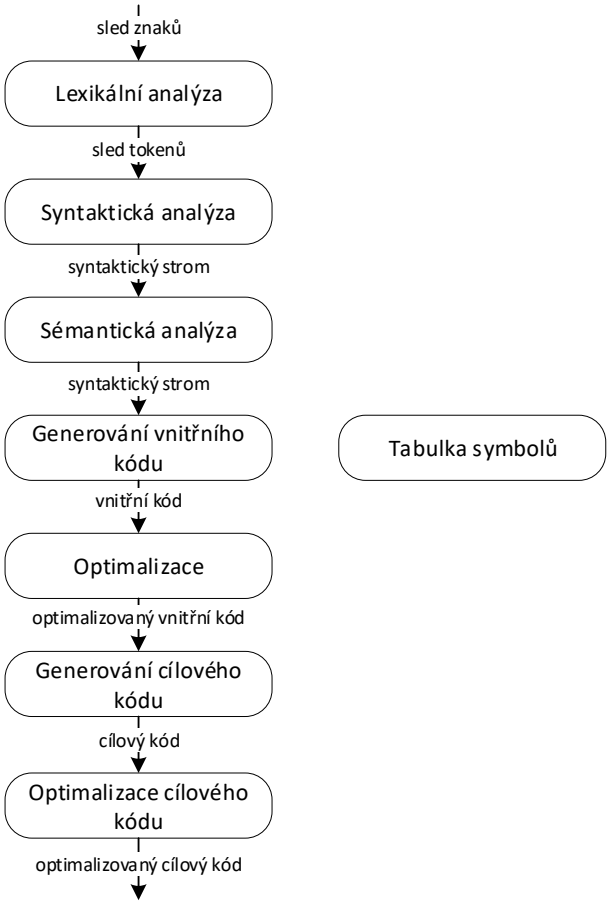
jazyka, skončí tato fáze s chybou. Ukázku toho, jak syntaktický strom vypadá, můžete vidět na Obrázek 3.3 [22].



Obrázek 3.3: Ukázka syntaktického stromu těla metody pomocí nástroje Roslyn Syntax Visualizer.

Po syntaktické analýze dochází ke kontrole *sémantické*, která přijímá vzniklý syntaktický strom a provádí kontrolu, zda je tento strom správně i po sémantické stránce. Mezi tyto kontroly patří například zjištění, zda všechny proměnné používané v kódu jsou řádně deklarovány (pokud to jazyk vyžaduje), zda jsou operandy operací správnými datovými typy, atd. Spolu se sémantickou kontrolou se zde provádí doplnění informací o datových typech do tabulky symbolů [22].

Po skončení sémantické analýzy nastává fáze *generování vnitřního kódu* a jeho následné optimalizování. Tato fáze je závislá na vnitřní implementaci kompilátoru a pro kompilátor je možné tuto fázi úplně vynechat. Po vygenerování dochází na různě složité druhy optimalizace výsledného kódu (od odstranění nepoužitých proměnných po složitější úpravy kódu vedoucí k jejich zrychlení). Optimalizace může znamenat zrychlení provádění výsledného



Obrázek 3.2: Jednotlivé fáze kompilátoru s jejich vstupy a výstupy.

kódu, ale může být provedena i za účelem snížení spotřeby energie nebo snížení objemu výsledného kódu [22].

Poslední fází je *generování a optimalizace výsledného kódu*, kdy po dokončení všech předchozích fází dojde k vytvoření kódu v cílovém jazyce, k jeho případné optimalizaci a následnému uložení tohoto výstupu[1].

3.2 Přístup k informacím pomocí Roslynu

Pokud chceme pomocí frameworku Roslyn přistupovat k informacím získaným během kompilace jednotlivého projektu, musíme tento projekt nejprve načíst do paměti a vyžádat od Roslynu jeho kompilaci. V této sekci se podíváme krok po kroku, jakým způsobem přidat referenci na Roslyn do projektu a poté jej použít k načtení již existujícího projektu a jeho kompilaci.

[1]Výstupem mohou být i dodatečné informace o výstupním produktu kompilace. Například kompilátory prostředí .NET vytváří i soubory pdb obsahující informace umožňující ladění výsledných programů [12].

Roslyn je rozdělený na několik modulů, kde každý z modulů začíná prefixem `Microsoft.CodeAnalysis`. Dvěma nejdůležitějšími z nich jsou `Microsoft.CodeAnalysis.Common`, který obsahuje základní prvky používané pro práci s frameworkem na úrovni kompilátoru (ať už jde o syntaxi, tabulku symbolů nebo práci s binárním výstupem)[2] a `Microsoft.CodeAnalysis.Workspaces` pro práci s projektovým systémem[3]. V případě, že si nejsme jistí, který modul potřebujeme, můžeme provést nainstalování NuGet[4] balíčku `Microsoft.CodeAnalysis`, který obsahuje všechny moduly frameworku Roslyn.

Jakmile máme v projektu přidané balíčky potřebné pro práci s Roslynem, musíme získat object abstraktního typu `Workspace`, který slouží ke shromáždění všech informací o načteném řešení (soubor .sln) včetně všech přidaných projektů a dokumentů v rámci těchto projektů. Objekt typu `Workspace` můžeme získat z jeho potomka `MsBuildWorkspace`, který poskytuje rozhraní k načtení řešení do paměti [20].[5]

```
var workspace = MsBuildWorkspace.Create();
await workspace.LoadSolutionAsync(@"C:\Test\Solution.sln");
return workspace;
```

Ukázka kódu 3.1: Ukázka načtení řešení pomocí Roslynu.

Jakmile máme načtený `Workspace`, můžeme pracovat s řešením jako celkem (= objekt typu `Solution`), případně s jednotlivými projekty uvnitř řešení (= objekty typu `Project`). Všechny tyto objekty mají (až na `Workspace`) neměnný stav[6], to znamená, že v případě zavolání jakékoliv metody působící změnu dochází k vrácení nového objektu daného typu s aplikovanou danou změnou.

Abychom poté mohli přistupovat k jakýmkoliv informacím získaným během kompilace některého z jeho projektů, musíme vyvolat jeho kompilaci a získat objekt typu `Compilation`, který zaobaluje veškeré informace získané během kompilace projektu. Po získání objektu `Compilation` můžeme pracovat se všemi získanými informacemi včetně všech diagnostických informací, které můžeme vidět i v průběhu běžné kompilace prostřednictvím varování a chyb včetně jejich kódů a chybových hlášek [20].

```
var compilation = await project.GetCompilationAsync();
foreach (var diagnostic in compilation.GetDiagnostics())
{
    Console.WriteLine("An error occured:");
    Console.WriteLine(diagnostic.GetMessage());
}
```

Ukázka kódu 3.2: Získání objektu `Compilation` a vypsání všech diagnostických informací získaných při kompilaci.

[2] Následně jsou tu také moduly vytvořené pro práci se specifickými prvky pro jednotlivé jazyky zvlášť `Microsoft.CodeAnalysis.CSharp` a `Microsoft.CodeAnalysis.VisualBasic`

[3] Následně jsou tu také moduly vytvořené pro práci se specifickými prvky pro projektový systém jednotlivých jazyků zvlášť `Microsoft.CodeAnalysis.CSharp.Workspaces` a `Microsoft.CodeAnalysis.VisualBasic.Workspaces`

[4] NuGet je balíčkovací systém používaný v rámci prostředí .NET používaný pro správu přidaných balíčků/knihoven v rámci projektu.

[5] Toto řešení je v době psaní této práce (3. 4. 2018) z důvodu nekompatibility podporováno pouze na operačních systémech Windows.

[6] *angl. immutable objects*

3.3 Syntaktická část

Jak již bylo uvedeno v této kapitole, vstupem kompilátoru je zdrojový kód napsaný v daném programovacím jazyce. Tento text je následně rozdělen na posloupnost tokenů a dochází ke kontrole jeho syntaktické správnosti. Syntaxe je soubor pravidel popisujících tento soubor tokenů z hlediska gramatiky vstupního jazyka. Na základě syntaxe jazyka jsme poté schopni vytvořit abstraktní syntaktický strom vstupního kódu (viz. Obrázek 3.3).

Pomocí Roslynu můžeme na syntaktických stromech provádět tři různé druhy operací. Jednak máme možnost procházet syntaktické struktury, které už byly získány v rámci Compilation, dále můžeme tyto struktury měnit [7] a také můžeme syntaktické struktury čistě tvořit[8].

Pokud nás zajímá přístup k syntaktickým informacím pouze z hlediska jejich analýzy-/čtení, můžeme k nim přistupovat skrze syntaktické stromy (viz. Obrázek 3.3). Syntaktické stromy v Roslynu poskytují reálný obraz původního zdrojového kódu a to od všech gramatických konstrukcí, přes všechny lexikální tokeny po strukury, které z hlediska konečného fungování kódu nenesou žádný význam (komentáře, netisknutelné znaky, ...). Syntaktický strom zároveň umožňuje získat zpětný přepis stromu ve formě řetězce zavoláním metody `ToDisplayString()`.

Pokud chceme syntaktické struktury tvořit a buď s nimi tvořit nové stromy, případně editovat s nimi staré, máme na výběr dvě možnosti. První možnosti je použít statické třídy `SyntaxFactory`, které existují dvě a to jedna ve jmenném prostoru `Microsoft.CodeAnalysis.CSharp` pro C# a druhá ve jmenném prostoru `Microsoft.CodeAnalysis.VisualBasic` pro Visual Basic. Tato třída obsahuje statické metody pro vytvoření každé syntaktické struktury popsané jednotlivými jazyky. Ukázku vytvoření syntaktického stromu pomocí třídy `SyntaxFactory` můžete vidět v ukázce kódu 3.4. Druhou možností je použít třídu `SyntaxGenerator` obsaženou ve jmenném prostoru `Microsoft.CodeAnalysis.Editing`, tato třída má možnost získat generátor syntaktických konstrukcí specifický pro určitý jazyk na základě objektu `Project`, což znamená, že kód vytvořený tímto generátorem se poté tvoří v jazyce, který je používaný právě v onom konkrétním projektu.

```
class C
{
    static void Main()
    {
    }
}
```

Ukázka kódu 3.3: Ukázkový původní kód

[7] Opět se jedná o neměnné objekty, takže při každé změně v rámci struktury vzniká nový objekt struktury obsahující změnu, tyto změny se týkají i nadřazených struktur. To znamená, že pokud změníme identifikátor v rámci deklarace proměnné, vytváří se nová deklarace, nový blok, který deklaraci obsahoval, až po horní uzel synaktického stromu.

[8] Tato schopnost Roslynu se například používá v DotVVM pro kontrolu správnosti bindingů.

```
SyntaxFactory.CompilationUnit()
.WithMembers(
    SyntaxFactory.SingletonList<MemberDeclarationSyntax>(
        SyntaxFactory.ClassDeclaration("C")
        .WithMembers(
            SyntaxFactory.SingletonList<MemberDeclarationSyntax>(
                SyntaxFactory.MethodDeclaration(
                    SyntaxFactory.PredefinedType(
                        SyntaxFactory.Token(SyntaxKind.VoidKeyword)),
                    SyntaxFactory.Identifier("Main"))
                .WithModifiers(
                    SyntaxFactory.TokenList(
                        SyntaxFactory.Token(SyntaxKind.StaticKeyword)))
                .WithBody(
                    SyntaxFactory.Block()))))
.NormalizeWhitespace()
```

Ukázka kódu 3.4: Původní kód z ukázce kódu 3.3 vytvořený pomocí třídy `SyntaxFactory`

3.4 Sémantická část

Jak už jsme si řekli, po syntaktické analýze přichází na řadu sémantická analýza. Po dokončení syntaktické analýzy máme k dispozici gramaticky správný syntaktický strom. Z něho už jsme schopni vyčíst velké množství informací, ale chybí nám informace týkající se datových typů použitých v kódu a jejich navázání na sebe (máme proměnnou určitého datového typu a provádíme do ní přiřazení nějakého výrazu). V sémantické fázi dochází tedy ke kontrole, zda výsledné datové typy prováděných operací odpovídají datovému typu, do kterého přiřazujeme, zároveň dochází k budování symbolických informací o jednotlivých typech používaných v programu. Každý jmenný prostor, datový typ, metoda, vlastnost, událost, parametr nebo lokální proměnná je reprezentovaný pomocí symbolu [20].

Sémantické informace jsou v rámci Roslynu poskytované jak ve formě tabulky symbolů, tak ve formě tzv. operačního stromu.

Symboly jsou reprezentované třídami implementujícími rozhraní `ISymbol`. Z tohoto rozhraní potom ještě dědí několik dalších, které jsou implementovány třídami popisujícími specifický druh symbolu (např. `IMethodSymbol`, `IFieldSymbol`, `INamedTypeSymbol`, ...), které poskytují dodatečné rozhraní potřebné pro dané specifické případy. Symboly jsou také poskytovány i pro datové typy, které se nenacházejí přímo v uživatelském kódu, ale jsou přítomné v externích knihovnách používaných uživatelem. V objektech popisujících jednotlivé symboly poté můžeme najít informace o jejich datovém typu, symbolu, který je obsahuje, jejich bázových symbolech a také informace o konkrétním místě jejich původu (může jít o zdrojový kód nebo o metadata nalinkovaných knihoven).

Pokud chceme přistupovat k symbolům, máme hned několik možností, kde každý má různé případy užití.

Prvním způsobem je přístup ke stromu symbolů přítomných v projektu skrze globální jmenný prostor, který má také svůj symbol. K tomuto symbolu se můžeme dostat skrze objekt `Compilation` a jeho vlastnost `GlobalNamespace`. Poté už můžeme pomocí metody `GetMembers()` přistupovat ke vnitřním symbolům reprezentujícím další jmenné prostory (skrze rozhraní `INamespaceSymbol`), až se dostaneme přímo k typům (rozhraní

`INamedTypeSymbol`) a na koncových uzlech stromu poté můžeme přistupovat k symbolům reprezentujícím prvky uvnitř těchto typů (rozhraní `IMethodSymbol`, `IPropertySymbol`, `IFieldSymbol` a další).

Další způsob, kterým můžeme přistupovat k symbolickým informacím použitým v našem kódu, je použití tzv. sémantického modelu (reprezentován objektem typu `SemanticModel`). Tento model je nositelem sémantických informací na úrovni jednoho syntaktického stromu. Tento model dále umožňuje přistupovat k symbolickým informacím právě na základě syntaktických deklarací, ze kterých tyto symboly vznikly.

```
//Vytvoreni syntaktickeho stromu
SyntaxTree tree = CSharpSyntaxTree.ParseText(@"using System;

class Program
{
    static void Main(string[] args)
    {
    }
}");

//Ziskani syntakticke struktury pro deklaraci metody Main
var methodDeclaration = tree.GetRoot()
    .DescendantNodes()
    .OfType<MethodDeclarationSyntax>()
    .First();

//Vytvoreni kompilace programu
var compilation = CSharpCompilation.Create("HelloWorld")
                    .AddReferences(
                        MetadataReference.CreateFromFile(
                            typeof(object).Assembly.Location))
                    .AddSyntaxTrees(tree);

// Ziskani semantickeho modelu
var semanticModel = compilation.GetSemanticModel(tree);
// a~symbolu popisujiciho deklarovanou metodu
var methodSymbol = semanticModel.GetSymbolInfo(methodDeclaration);
```

Ukázka kódu 3.5: Ukázka získání symbolických informací za pomoci objektu `SemanticModel`.

Dále Roslyn poskytuje operační strom. Jedná se o stromovou strukturu popisující operace zapsané syntaktickým stromem doplněné o sémantické informace, kde každý uzel stromu je typem implementujícím rozhraní `IOperation`. Z tohoto rozhraní opět dědí několik dalších, které jsou implementovány typy popisujícími specifický druh operace (např. `IAssignmentOperation`, `IBinaryOperation`, `IInvocationOperation`, ...). V rámci těchto objektů poté máme k dispozici informace o všech symbolech, které se na dané operaci podílejí, o vnořených operacích, nacházejících se v rámci dané operace, a také dodatečné informace potřebné pro sémantiku dané operace. Například objekt implementující `IVariableDeclaratorOperation` v sobě nese informace o symbolu proměnné, která touto

deklarací vzniká (vlastnost `Symbol`), a operaci popisující inicializaci dané proměnné (vlastnost `Initializer`).

Pro přístup ke stromu operací je potřeba opět získat buď objekt typu `Compilation` nebo objekt typu `SemanticModel`. Dále nám stačí získat syntaktické struktury popisující operace, které nás zajímají.

```
//Vytvoreni syntaktickeho stromu
SyntaxTree tree = CSharpSyntaxTree.ParseText(@"using System;

class Program
{
    static int Main(string[] args)
    {
        return 0;
    }
}");

//Ziskani syntakticke struktury pro deklaraci metody Main
var methodDeclaration = tree.GetRoot()
    .DescendantNodes()
    .OfType<MethodDeclarationSyntax>()
    .First();

//Vytvoreni kompilace programu
var compilation = CSharpCompilation.Create("HelloWorld")
                    .AddReferences(
                        MetadataReference.CreateFromFile(
                            typeof(object).Assembly.Location))
                    .AddSyntaxTrees(tree);

//Ziskani operace popisujici telo deklarovane metody
var methodBodyOperation = compilation.GetOperation(methodDeclaration.Body);
```

Ukázka kódu 3.6: Ukázka získání operačního stromu za pomoci objektu `Compilation`.

Kromě různých možností pro přístup k symbolickým datům Roslyn rovněž nabízí spoustu pomocných tříd pro práci s těmito daty. Pro práci s operacemi například nabízí objekty `OperationWalker` a `OperationVisitor` implementující návrhový vzor Visitor[9], které umožňují jednoduché procházení stromové struktury operací. Pro práci se symboly jsou v nabídce, kromě obdobné varianty visitorů jako u operací, také různé nástroje pro hledání použití daného symbolu, či nástroje pro editaci daného symbolu i na úrovni celého projektu a mnohé další.

[9] Návrhový vzor Visitor (česky též nazývaný návštěvník) je používaný ve chvíli, kdy potřebujeme provádět různé operace nad heterogenní agregovanou datovou strukturou a nechceme tyto operace zanášet přímo do objektů reprezentujících prvky dané struktury. Implementace se provádí většinou tak, že v rozhraní visitoru vytvoříme metody pro každý prvek datové struktury, která jej bude akceptovat jako parametr. V objektech reprezentujících jednotlivé uzly poté pouze přidáme metodu, která zavolá odpovídající metodu na visitoru a předá ji jako parametr aktuální instanci objektu.

3.5 Kompilační část

Poslední vrstvou, kterou nám takto otevřené rozhraní kompilátoru nabízí, je možnost pracovat s výstupem kompilátoru. Na této úrovni tedy můžeme námi vytvořený program zkompilovat a vytvořit jeho binární výstup (soubor .dll pro knihovnu a soubor .exe pro spustitelné programy). Kromě získání binárního výstupu si můžeme od Roslynu také vyžádat vytvoření pdb souborů a vytvoření dokumentace programu na základě komentářů ve formátu XML. Dále je možné s těmito výstupními prvky pracovat na úrovni sledování rozdílů mezi jednotlivými kompilacemi [23].

Získání výstupů kompilace je velice jednoduchý proces, který funguje tak, že nejprve získáme objekt `Compilation` pro náš daný projekt, a poté nad ním pouze zavoláme rozšiřující metodu `Emit` ze třídy `FileSystemExtensions`.

```
var tree = CSharpSyntaxTree.ParseText(@"
using System;
public class C
{
    public static void Main()
    {
        Console.WriteLine(""Hello World!"");
        Console.ReadLine();
    }
}");
var mscorlib =
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
var compilation = CSharpCompilation.Create("MyCompilation",
syntaxTrees: new[] { tree }, references: new[] { mscorlib });
var emitResult = compilation.Emit("output.exe", "output.pdb",
    "output.xml");
```

Ukázka kódu 3.7: Zakompilování projektu a zapsání jeho binárního obrazu(`output.exe`), ladících informací(`output.xml`) a dokumentace ve formátu XML(`output.xml`) na disk.

3.6 Analytická část

Roslyn kromě všech vrstev kompilátoru které nabízí, přidává několik analytických nástrojů, které nám umožňují kód sledovat například z hlediska toku dat či z hlediska toku kódu. Tyto analytické nástroje se poté dají vhodně použít buď z hlediska vytváření nástrojů pro podporu programování (například pomocné nástroje pro refaktoring kódu) nebo z hlediska statické analýzy kódu, kde bychom například mohli sledovat jestli nemáme proměnnou, do které nikdo nezapisuje.

3.6.1 Analýza toku dat

Jedním z těchto analytických nástrojů je nástroj pro analýzu toku dat. Jakmile získáme objekt typu `Compilation` můžeme získávat tyto informace na základě jednotlivých prvků syntaktického stromu, které ohraničují část kódu, jež nás z hlediska této analýzy zajímá. Z tohoto objektu poté stačí získat sémantický model stromu, který chceme sledovat a poté už jen stačí vyvolat tuto analýzu zavoláním metody `AnalyzeDataFlow`. Tato metoda nám

vrátí objekt typu `DataFlowAnalysis`, který v sobě nese informace o všech proměnných, které byly čteny a nacházejí se v rámci daného bloku kódu (vlastnost `ReadInside`) nebo do kterých bylo něco zapsáno a nacházejí se v rámci tohoto bloku (vlastnost `WrittenInside`) nebo které proměnné jsou v rámci bloku deklarovány (vlastnost `VariablesDeclared`).

```
int[] outerArray = new int[10] { 0, 1, 2, 3, 4, 0, 1, 2, 3, 4};
for (int index = 0; index < 10; index++)
{
    int[] innerArray = new int[10] { 0, 1, 2, 3, 4, 0, 1, 2, 3, 4 };
    index = index + 2;
    outerArray[index - 1] = 5;
}
```

Ukázka kódu 3.8: Analyzovaný kód [1].

AlwaysAssigned	index
DataFlowsIn	outerArray
ReadInside	outerArray, index
VariablesDeclared	index, innerArray
WrittenInside	index, innerArray
WrittenOutside	this, outerArray

Tabulka 3.1: Ukázkový výstup analýzy datových toků postavených na základě cyklu použitého v ukázce kódu 3.8. V levém sloupci vidíme vlastnosti třídy `DataFlowAnalysis` a v pravém sloupci jejich hodnoty.

3.6.2 Analýza toku programu

Dalším analytickým nástrojem, který Roslyn nabízí, je nástroj pro analýzu toku programu. Tímto nástrojem je možné sledovat, jakým způsobem se určité části kódu vstupuje a na jakých místech se z ní dá vystoupit. Toto chování poté můžeme sledovat z pozice jednoho bloku kódu, případně v rámci rozmezí mezi jednotlivými výrazy v kódu. Pomocí tohoto nástroje je například možné identifikovat nedosažitelný kód v rámci programu či sestavit graf posloupnosti kódu v rámci programu [24].

Pro získání informací o toku kontroly uvnitř programu je opět potřeba nejprve získat objekt typu `Compilation` a z něj pomocí sémantického modelu sledovaného syntaktického stromu tyto informace získat. Tyto informace jsou reprezentovány třídou `ControlFlowAnalysis`, která v sobě nese informace o všech vstupních a výstupních bodech do sledovaného bloku kódu (vlastnosti `EntryPoints` a `ExitPoints`) ve formě odkazů na prvky syntaktického stromu, které tyto vstupy a výstupy toku řízení ovlivňují.

```
for (int i = 0; i < 10; i++)
{
    if (i == 3)
        continue;
    if (i == 8)
        break;
    if (i == 4)
        return;
```

```
}
```

Ukázka kódu 3.9: Analyzovaný kód [2].

`EntryPoints`	
`ExitPoints`	`continue`, `break`, `return`
`ReturnStatements`	`return`
`EndPointIsReachable`	`true`
`StartPointIsReachable`	`true`

Tabulka 3.2: Ukázkový výstup analýzy řízení toku progaramu postavené na základě cyklu použitého v ukázce kódu 3.9. V levém sloupci vidíme vlastnosti třídy `ControlFlowAnalysis` a v pravém sloupci jejich hodnoty v prvních třech řádcích reprezentované jako prvky syntaktického stromu a v posledních dvou řádcích jako pravdivostní hodnoty.

Kapitola 4

Analýza problému

Při tvorbě překladače z jednoho programovacího jazyka do druhého existuje několik faktorů, které se musí vzít v potaz před návrhem konečného řešení. Tyto problémy poté můžeme rozdělit do dvou kategorií. První kategorii tvoří problémy související s rozdíly mezi jazyky (například v jejich přístupu k základním typům, objektové orientovanosti atd.). Druhou kategorii tvoří problémy související už přímo s procesem překladu (např. způsob získání informací, reprezentace vstupního a výstupního jazyka, analýza přeložitelnosti, atd.).

V této kapitole se nejprve podívám na problémy související se samotným překladem a následně na problémy vzniklé rozdíly mezi jazyky.

4.1 Překladové problémy

Při psaní překladače se musím také zaměřit na řešení problémů souvisejících se samotným překladem a ne se vstupním/výstupním jazykem jako takovým. Budu muset být schopen získat veškeré informace, které potřebuji k překladu. Poté s těmito informacemi patřičným způsobem naložit a vytvořit výstup tak, aby se s ním dalo dále pracovat, pokud to bude potřeba.

4.1.1 Určení přeložitelnosti

Věcí, kterou je nutné určit, je míra přeložitelnosti. Do jaké míry jsme schopni přeložit uživatelský kód tak, aby se choval stejně jako doposud? Zároveň je nutné sledovat vůbec možnost přeložitelnosti. Je tato metoda ještě přeložitelná? Nebo volá tato metoda ještě nějaký kód, který na klientovi nelze spustit? Dá se k této metodě ještě najít ekvivalentní metoda pro toto volání v JavaScriptu? Tento problém bude potřeba adresovat hned na několika místech a to: volání metody napsané uživatelem, volání knihovní metody, přistupování k vlastnosti objektu na úrovni knihovních objektů.

Uživatelský kód

V rámci uživatelských metod bude nutné určit jejich přeložitelnost na úrovni metody a všech metod, které jsou jí volány. Zároveň je potřeba sledovat, zda všechny operace použité v uživatelském kódu jsou vůbec překladačem podporované (volání knihovních metod, asynchronní operace, ...).

Knihovní metody a vlastnosti

Na úrovni kódu přístupného ve standardních knihovnách jazyka C#, případně knihovnách třetích stran, nastává úplně odlišný problém. Při přístupu k těmto vlastnostem a metodám nemáme k dispozici jejich zdrojový kód, takže není možné provádět překlad přímo. Pro řešení tohoto problému bude třeba dát vývojáři možnost, aby si nadefinoval, jakým způsobem budou některé metody a operace přeloženy do jejich ekvivalentu v jazyce JavaScript.

4.1.2 Práce se vstupním jazykem

Při překladu bude nutné získat veškeré informace zapsané v kódu ve vstupním jazyce, které jsou potřebné pro překlad. Tyto informace jsou:

- informace o třídách;
- informace o vlastnostech[1] uvnitř tříd;
- informace o metodách a jejich implementaci.

O třídách dostupných v programu potřebuji zjistit jejich jména, jmenný prostor, ve kterých se nachází, třídy, ze kterých dědí, a seznam vlastností a metod, které se uvnitř třídy nachází. Jméno je potřeba z hlediska překladu pro vytvoření ekvivalentní třídy na klientské úrovni. Jmenný prostor je potřeba pro případ, že dojde k překladu dvou tříd se stejným jménem, aby bylo možné mezi nimi rozlišit. Seznam vlastností tříd a jejích vnitřních metod je potřeba hlavně z důvodu možnosti vytvoření jejich protějšků v klientském prostředí.

Vlastnosti třídy musím identifikovat z hlediska jejich názvů a datových typů. Název je opět důležitý pouze z hlediska dostupnosti a přístupu k dané vlastnosti v rámci klientského kódu. Její datový typ potřebujeme pro kontrolu, zda je daný datový typ primitivní a přeložitelný nebo je objektový, a tím pádem jej bude třeba také přeložit. Zároveň je potřeba sledovat speciální datové typy, jako jsou například pole, která mají jiné specifické vlastnosti pro jejich přístup v rámci výstupního kódu.

U metod nacházejících se uvnitř tříd je třeba sledovat jejich název, parametry a implementaci. Název je znovu potřebný pouze z důvodu vytvoření metody o ekvivalentním názvu v klientském kódu. Parametry jsou důležité kvůli sledování jejich názvu a datového typu, aby opět došlo k tvorbě ekvivalentní verze parametrů na straně klientské. Implementace je poté potřeba k samotnému překladu, aby bylo možné vytvořit ekvivalentní implementaci na straně klienta.

4.1.3 Práce se výstupním jazykem

Pro reprezentaci a práci s výstupním jazykem je důležité soustředit se na to, aby implementace překladače umožňovala v budoucnu provést změnu mezi různými výstupními jazyky bez nutnosti přepisování samotné analytické práce. Všechny konstrukce vstupního jazyka, které budou překladačem podporované, musí mít svoje ekvivalenty v definici struktur výstupního jazyka. To znamená, že bude třeba definovat všechny podporované konstrukce na úrovni vstupního jazyka a provést implementaci prvků syntaktického stromu výstupního jazyka, které jim budou ekvivalentní.

[1] angl. *properties*

4.2 Jazykové problémy

Rozdíly mezi jazyky JavaScript a C# jsou z velké části dány tím, v jakém prostředí se dané jazyky používají. Jazyk JavaScript je primárně určený pro běh za pomoci interpretu (ať už ve webovém prohlížeči, či na počítači pomocí interpretu Node) [25]. C# je jazyk kompilovaný, který funguje na všech platformách, které podporují .NET Framework (případně .NET Core). Právě z rozdílného způsobu používání vyplývá většina rozdílů, které mezi těmito jazyky jsou.

4.2.1 Rozdílné datové typy

Počítačové programy potřebují mít při práci možnost uložit si data do proměnných. Proměnné poté na základě dat, která jsou v nich uložena, mají určitý datový typ, ať už se jedná o číslo nebo text. Datové typy poté můžeme rozdělit do dvou skupin: primitivní datové typy (čísla, pravdivostní hodnoty atd.) a objektové typy. Při překladu mezi jazyky je nutné sledovat právě způsob reprezentace datových typů, aby při překladu bylo dosaženo, že kód, který se ve vstupním jazyce nějak chová, se bude stejně chovat i po překladu při jeho spuštění ve výstupním jazyce.

Primitivní datové typy

Do primitivních datových typů jsou obecně řazeny:

- číselné datové typy;
- pravdivostní hodnoty (pravda, nepravda);
- textové a znakové datové typy.

U těchto datových typů musíme dát pozor na dvě věci: jejich reprezentace z hlediska jejich zápisu (syntaxe) a jejich interní reprezentace za běhu programu. Z hlediska syntaxe je důležité sledovat, zda se literály tohoto datového typu zapisují stejně. U interní reprezentace je potřeba sledovat rozdíly reprezentace datových typů v běhovém prostředí programu, aby nedošlo k problémům jako je přetečení datového formátu, případně dosazení nevalidní hodnoty.

Číselné datové typy jsou v rámci jazyků C# a JavaScript reprezentovány z hlediska syntaxe stejně. Umožňují zápis jak v celých číslech, tak v desetinných číslech i pomocí exponentu. Na úrovni interní reprezentace už nastává problém. Zatímco C# používá pro reprezentaci čísel hned několik datových typů, kde každý má různé druhy užití (celá čísla, desetinná čísla a přirozená čísla) a různé rozsahy hodnot, které je schopen pojmout. JavaScript má pouze jeden datový typ, který se používá pro reprezentaci obecného čísla (ať už se jedná o přirozené, desetinné nebo celé číslo). Tento problém bude potřeba na úrovni překladače sledovat, protože v případě určitých speciálních operací (např. dělení) může dojít k neshodě výsledku mezi C# a JavaScriptem. Tento jev je ilustrován v ukázkách kódu 4.1 a 4.2. Možnost výskytu tohoto jevu bude potřeba v průběhu překladu sledovat a použít ochranné prvky, aby se kód i po přeložení choval identicky.

```
int a = 5;
int b = 4;
int c = a / b;
//c = 1;
```

Ukázka kódu 4.1: Sledování výsledku operace dělení dvou celých čísel v jazyce C#. Běhové prostředí provede výpočet operace a odebere všechna čísla nacházející se za desetinnou čárkou.

```
var a = 5;
var b = 4;
var c = a / b;
//c = 1.25;
```

Ukázka kódu 4.2: Sledování výsledku operace dělení v JavaScriptu, která je syntakticky ekvivalentní s ukázkou 4.1. Výsledek se liší, protože JavaScript umožňuje do svého číselného datového typu uložit jak celá, tak desetinná čísla.

Dále nastává problém při přetečení datového typu v jazyku C#, kdy každý datový typ má svůj daný rozsah, na kterém operuje. Při tzv. přetečení jeho hodnoty dojde k počítání od horní/spodní hranice typu na základě toho, která hranice byla přetečena. Vzhledem k tomu, že JavaScript má pouze jeden datový typ pro čísla, který umožňuje dosazení hodnoty nekonečno, tento jev u něj nikdy nenastane. Bude jej tedy třeba na úrovni překladače emulovat, aby chování jazyků zůstalo stejné.

U pravdivostních datových hodnot nenastává problém ani v jednom ze specifikovaných případů. Syntakticky jsou reprezentovány stejně (jako hodnoty `true` a `false`) a interní reprezentace dodržuje stejné chování jako v případě jazyku C#.

U znakových datových typů nastává problém jejich neexistence v jazyku JavaScript. Tento problém je třeba řešit buď jejich emulací na úrovni jazyka JavaScript nebo jejich zakázáním na úrovni celého překladu.

Textové hodnoty jsou z hlediska syntaxe ekvivalentní k jazyku C# a nedochází zde k žádné kolizi, která by mohla způsobovat problémy. Na úrovni vnitřní reprezentace pak můžeme sledovat rozdíly, které jsou způsobeny nahlížením na tento datový typ. C# k řetězci přistupuje jako k instanci objektu `string`, zatímco JavaScript na něj pohlíží jako na pole 16-bitových číselných hodnot. Z hlediska základního používání textového datového typu to není problém, protože operace jako přiřazení a konkatenace se budou chovat v obou jazycích identicky. Problém nastává při přístupu k jednotlivým vlastnostem a metodám objektu `string` v rámci jazyku C#, kdy se takovéto vlastnosti na jeho JavaScriptovém ekvivalentu nenachází. Na tento problém bude potřeba nahlížet stejně jako na používání knihovních metod a vlastností (viz. Sekce 4.1.1).

Objektové datové typy

Pokud se na jazyky díváme z hlediska jejich objektové orientace, tak C# můžeme zařadit mezi třídní objektově orientované jazyky[2]. JavaScript svůj přístup k objektové orientaci

[2]Třídní objektově orientované jazyky jsou jazyky vytvářející objekty na základě jejich třídy. To znamená, že máme „šablonu“ (třída), která definuje všechny vlastnosti a metody objektu. Každá instance tohoto objektu se poté vytváří na základě této třídy. Třída může dědit od nadřazené třídy a získat nebo upravit všechny její vlastnosti a metody [26].

inspiroval jazykem Self, takže se řadí mezi beztřídní objektově orientované jazyky, konkrétně prototypově založené[3].

Při překladu z třídně orientovaného prostředí do prototypově orientovaného jazyka bude potřeba na úrovni prototypově orientovaného jazyka emulovat třídní prostředí, aby došlo k zachování stejného chování, jaké očekáváme na úrovni vstupního (třídního) jazyka.

[3]Prototypově založené (angl. *prototype-based*) objektově orientované jazyky vycházejí z řešení, při kterém dojde k vytvoření prvotního objektu, kterému se nadefinují všechny metody a vlastnosti. Další objekty stejného typu se následně vytváří pomocí klonování tohoto objektu [25].

Kapitola 5

Implementace

Implementační část této práce se skládá ze dvou částí. První částí bylo naprogramovat aplikaci, která umožňovala překlad viewmodelů ze C# do JavaScriptu, a druhou částí bylo napojení těchto přeložených tříd na úrovni frameworku DotVVM.

5.1 Překladač

Překladač je navržen jako konzolová aplikace, která přijímá jako argumenty pouze cestu k souboru `.sln` a jméno projektu, jehož viewmodely se mají překládat. Aplikace používá framework .NET Core a to z důvodu možnosti jejího použití na všech platformách. Tato aplikace má na starosti následující úkony:

1. Načtení informací o projektu do paměti;
2. Nalezení metod a typů, které mají být přeloženy;
3. Vytvoření ekvivalentní syntaktické struktury odpovídající vstupním částem aplikace;
4. Přidání přeložených JavaScriptových viewmodelů do projektu.

V této sekci jsou tyto části popsány z hlediska implementace a použitých technologií. Dále se v této sekci zmiňuje řešení několika problémů způsobených rozdíly mezi oběma jazyky.

5.1.1 Načtení informací o projektu

Informace o projektu musí být načteny na základě argumentů aplikace, kterými jsou cesta k souboru `.sln` a jméno projektu, jenž se v tomto řešení nachází.

Jak už bylo v této práci zmíněno (viz. Sekce 3.2), pro přístup k informacím pomocí frameworku Roslyn je potřeba nejprve získat objekt typu `Compilation` reprezentující kompilaci právě analyzovaného projektu. Pro toto načtení je nejprve třeba získat objekt typu `Workspace`. Vzhledem k tomu, že objekt `MsBuildWorkspace` není podporovaný na platformě .NET Core, byl jsem nucen použít balíček `Buildalyzer`, který možnost načítání souborů `.sln` poskytuje na všech platformách. Z načteného objektu `Workspace` se poté najde objekt typu `Project`, který už nabízí metodu `GetComilationAsync()`, jež vrací objekt `Compilation`, ze kterého je možné všechna data získat.

5.1.2 Nalezení přeložitelných typů

Ve chvíli, kdy jsou všechny informace nasbírané kompilátorem dostupné přes objekt `Compilation`, je možné začít vyhledávat přeložitelné části kódu, které se v daném projektu nachází. Pro překlad je třeba získat všechny symbolické informace, které o daných typech, metodách a vlastnostech budou potřeba (viz. Sekce 4.1.2).

Z hlediska zpětné kompatibility je potřeba zachovat původní chování frameworku DotVVM. Tím je myšleno, že překládat se budou pouze metody, které programátor označí, všechny ostatní se budou vykonávat na serveru. Toto označení je provedeno atributu `ClientSideMethod` pro metody a `ClientSideConstructor` pro konstruktory. Tímto se zároveň řeší problém přeložitelnosti u uživatelského kódu (viz. Sekce 4.1.1).

Překladači tím pádem stačí najít třídy, které obsahují takto označené metody případně konstruktory a provést nad nimi překlad. Vyhledávání probíhá na úrovni symbolů, které jsou zastoupeny uvnitř zkompilovaného projektu. Vyhledávání má na starosti objekt `MultipleSymbolFinder`, který jako parametr přijímá implementaci rozhraní `ISymbolFilter`, jež představuje predikát popisující vlastnosti hledaných symbolů. Použitou implementaci je možné vidět v ukázce kódu 5.1. `MultipleSymbolFinder` je už jen implementací abstraktní třídy `SymbolVisitor`, která projde všechny symboly přítomné v projektu a vrátí ty z nich, které splňují podmínku obsaženou v předané implementaci `ISymbolFilter`.

```
public class ClientSideMethodFilter : ISymbolFilter
{
    public bool Matches(ISymbol symbol)
    {
        return symbol is IMethodSymbol
            && symbol.HasAttribute<ClientSideMethodAttribute>();
    }
}
```

Ukázka kódu 5.1: Ukázková implementace rozhraní ISymbolFilter, používaná pro dohledání všech metod, které na sobě mají navěšený atribut `ClientSideMethodAttribute`.

Takto nalezené symboly jsou poté uloženy v rámci objektu `TypeRegistry`, který si drží informace o všech přeložitelných a již přeložených typech. Přes nepřeložené symboly se po jejich kompletním vyhledání iteruje a provádí se na nich překlad. Tyto získané symboly zároveň obsahují všechny potřebné informace (viz. Sekce 4.1.2), případně k nim umožňují přístup.

5.1.3 Tvorba výstupní syntaktické struktury

Nyní se překladač nachází ve stavu, kdy má k dispozici všechny informace o typech a metodách, které má za úkol přeložit. Dalším krokem je poté už samotný překlad. V tuto chvíli se na vstupu nachází symbolické informace o jednotlivých prvcích, které mají být přeložené, a na výstupu se očekávají syntaktické stromy popisující jednotlivé přeložené typy.

Pro reprezentaci výstupní syntaxe byla navržena sada rozhraní, která popisuje jednotlivé syntaktické struktury, jež se objevují ve vstupní reprezentaci jazyka (viz. Ukázka kódu 5.2). Kromě jednotlivých rozhraní popisujících syntaktické prvky jazyka bylo navrženo i rozhraní `ISyntaxFactory`, které se používá pro vytváření právě těchto prvků. Cílem takto navržené abstrakce nad syntaxí je velice jednoduché přepnutí z generování jednoho jazyka na druhý,

stačí pouze implementovat jednotlivá rozhraní popisující všechny konstrukce a poté implementovat `ISyntaxFactory`, aby vracela tyto implementace. Tímto se jednoduše vyřešil již zmiňovaný problém záměny jednotlivých jazyků (viz. Sekce 4.1.3).

Současně používaná implementace `ISyntaxFactory` neprovádí překlad přímo do JavaScriptu, místo něj se používá jazyk TypeScript, který má k jazyku C# syntakticky blíže (nabízí třídní systém). Tímto použitím se eliminuje problém překladu mezi prototypovými a třídními objektově orientovanými jazyky (viz. Sekce 4.2.1).

```
public interface IMemberDeclarationSyntax : ISyntaxNode
{
    AccessModifier Modifier { get; }
    IIdentifierSyntax Identifier { get; }
}
```

Ukázka kódu 5.2: Ukázka rozhraní popisujícího deklaraci prvku nacházejícího se uvnitř třídy. Z tohoto rozhraní poté dědí další, které jsou specifické pro daný prvek (např. `IPropertyDeclarationSyntax`).

Z hlediska návrhu už chybí jen způsob, kterým by se nasbírané symbolické informace překládaly na syntaktické struktury výstupního jazyka. Pro tento překlad bylo navrženo generické rozhraní `ITranslator` (viz. ukázce kódu 5.3). Pro každý symbol následně implementujeme toto rozhraní a jako generický parametr mu předávám symbol, který překládá. Tyto jednotlivé překladače poté registrujeme do objektu `TranslatorsEvidence`, který umožňuje právě na základě datového typu symbolu dohledat ekvivalentní překladač a provést přeložení symbolu na ekvivalentní syntaktickou strukturu vytvořenou pomocí `ISyntaxFactory`.

```
public interface ITranslator<in TInput> where TInput: ISymbol
{
    bool CanTranslate(TInput input);
    ISyntaxNode Translate(TInput input);
}
```

Ukázka kódu 5.3: Rozhraní umožňující překlad jednotlivých symbolů. Symbol, který implementující objekt překládá se určuje pomocí typu dosazeného za generický parametr `TInput`.

V případě překládání metod/konstruktorů je důležité provést i překlad jejich těla. Jak již bylo zmíněno (viz. Sekce 3.4), Roslyn pro reprezentaci těla metod nabízí práci s tzv. operačním stromem, který je ekvivalentem syntaktického stromu obohaceného o symbolické informace. Pro překlad těla metody/konstruktoru se nejprve získá operační strom, toto tělo popisující, a poté se předá objektu typu `OperationTranslatingVisitor`. Tento visitor je implementací abstraktní třídy `OperationVisitor`, která provede navštívení všech operací uvnitř stromu a na jejich základě sestaví syntaktický strom vytvořený pomocí `ISyntaxFactory`. Ukázkovou implementaci takového překladu můžete vidět v ukázce kódu 5.4. Zároveň zde provádí kontrolu na správnou práci s číselnými typy, kde se mimo jiné u jakéhokoliv číselného výrazu provádí kontrola, zda na něm daná operace může změnu datového typu (z celého čísla na reálné). Například při překladu `IBinaryOperation`, kde výsledkem má být celé číslo, je přidána konstrukce zajišťující zachování tohoto datového typu, aby operace s ním odpovídali chování v jazyce C# (viz. Sekce 4.2.1).

```
public override ISyntaxNode VisitUnaryOperator(IUnaryOperation operation,
    ISyntaxNode parent)
{
    _logger.LogDebug("Operations", "Translating unary operation.");
    var operand = operation.Operand.Accept(this, parent) as
        IExpressionSyntax;
    var unaryOperator = operation.OperatorKind.ToUnaryOperator();
    return _factory.CreateUnaryOperation(operand, unaryOperator, parent);
}
```

Ukázka kódu 5.4: Jedna z metod objektu `OperationTranslatingVisitor` provádějící překlad unární operace na rozhraní `IUnaryOperationSyntax` za pomoci `ISyntaxFactory` (v kódu je to její instance `_factory`).

V případě, že `OperationTranslatingVisitor` objeví v uživatelském kódu volání metody (`IInvocationOperation`), provede se kontrola, zda má daná metoda na sobě umístěný atribut `ClientSideMethod`. Pokud se zde tento atribut nachází, dojde k překladu volání oné metody, protože tato metoda je/bude přeložena a dá se s ní na klientské straně počítat. Pokud se tento atribut na metodě nenachází, provede se její hledání v registru vestavěných metod. Tento registr umožňuje pomocí objektu identifikujícího metodu (reflexí získaný objekt `MethodInfo`) a implementací rozhraní `IMethodCallTranslator` zaregistrovat překladač pro vestavěnou metodu, u které je v rámci překladu znám její ekvivalent. Za volání takto zaregistrované metody se poté dosadí syntaktický strom vytvořený zaregistrovanou implementací rozhraní `IMethodCallTranslator`. Ekvivalentní chování se poté používá i pro překlad některých vlastností objektů (např. vlastnosti `Count` u objektu `List`).

Po provedení všech těchto úprav se překlad nachází ve stavu (za předpokladu, že všechny kosntrukce použité v klientském kódu byly podporovány), kdy je pro všechny třídy a metody, které měly být přeloženy, k dispozici ekvivaletní syntaktický strom v jazyce daném implementací `ISyntaxFactory` (aktuálně TypeScript).

5.1.4 Uložení JavaScriptových Viewmodelů

V poslední fázi překladu má program k dispozici přeložené syntaktické stromy tříd, které mají být používány na klientské straně. Tyto třídy je potřeba uložit na disk a zkompilovat z jazyku TypeScript do JavaScriptu.

Pro jednoduché procházení syntaktických stromů je v překladači k dispozici rozhraní `INodeVisitor`. Toto rozhraní umožňuje navštívit jednotlivé prvky stromů zvlášť na základě jejich typu (pro každou syntaktickou strukturu má k dispozici metodu, která bude zavolána). Převedení syntaktických stromů do textové podoby je provedeno implementací tohoto rozhraní. Příklad takovéhoto převodu je možné vidět v ukázce kódu 5.5. Takto naformátované struktury se poté uloží na disk.

```
public void VisitWhileStatement(IWhileStatementSyntax whileStatement)
{
    Indent();
    Append("while (");
    whileStatement.Condition.AcceptVisitor(this);
    Append(") ");
    IncreaseIndent();
    whileStatement.Body.AcceptVisitor(this);
    DecreaseIndent();
}
```

Ukázka kódu 5.5: Metoda nacházející se v implementaci `INodeVisitor`, která se používá pro překlad syntaktických struktur do jejich textové podoby. V tomto případě se jedná o zápis cyklu typu `while`.

Po dokončení překladu a uložení jednotlivých přeložených tříd na disk se ještě provede zavolání procesu kompilátoru jazyku TypeScript pro jejich přeložení do JavaScriptu. Všechny tyto třídy se zkompilují do souboru `dotvvm.viewmodels.generated.js`. V tuto chvíli úloha překladače končí.

5.2 Napojení na DotVVM

Jakmile jsou přeloženy všechny třídy, které obsahují metody prováděné na klientské straně je potřeba provést jejich napojení na framework DotVVM, tak aby DotVVM odesílalo při přístupu na stránku soubor `dotvvm.viewmodels.generated.js`, vytvářelo na klientské straně odpovídající instance ViewModelů a provádělo volání odpovídajících metod.

5.2.1 Odesílání souboru `dotvvm.viewmodels.generated.js`

DotVVM nabízí pro připojování zdrojových souborů na stránku rozhraní pro práci se zdroji. Toto rozhraní umožňuje zaregistrovat soubor tak, aby došlo k jeho odeslání při každém přístupu na kteroukoliv stránku. Takže na této úrovni stačilo pouze soubor `dotvvm.viewmodels.generated.js` zaregistrovat do seznamu zdrojů a DotVVM ho automaticky odesílá při každém požadavku. Kód použitý pro dosažení tohoto řešení k dispozici v ukázce kódu 5.6.

```
configuration.Resources.Register("dotvvm.viewmodels.generated",
    new ScriptResource(
        new UrlResourceLocation("~/Scripts/dotvvm.viewmodels.generated.js")
    )
);
```

Ukázka kódu 5.6: Kód použitý pro registraci souboru `dotvvm.viewmodels.generated.js` do seznamu používaných zdrojů.

5.2.2 Vytváření instance Viewmodelů

Pro vytvoření instance třídy na klientské úrovni je třeba mít k dispozici jméno třídy, jejíž instance se má na klientské straně vytvořit. Pro přidání této informace do dat odesílaných klientovi v rámci informací o ViewModelu bylo vyřešeno úpravou třídy

`ViewModelSerializationMap`. Objekt této třídy při prvním dotazu na stránku vytvoří delegáta, který je poté používán při serializaci jejího ViewModelu. Tento delegát přidává do JSONu reprezentujícího ViewModel metadata potřebné pro jeho fungování. Do těchto metadat byly přidány další s klíčem `$class` a hodnotou nastavenou podle jména třídy.

Tato informace je poté v klientské části během deserializace vyhledána a pokud nalezena, dojde k vytvoření instance třídy, která se v rámci Knockoutu a nastaví jako kontext dané aplikace.

5.2.3 Volání odpovídajících metod

Jako poslední část integrace do frameworku DotVVM je třeba upravit Command bindingy (viz. Sekce 2.2.1), které jsou navázané na metody označené atributem `ClientSideMethod` tak, aby místo vyvolání požadavku na serveru zavolaly metodu v JavaScriptovém viewmodelu.

Při renderování dotazu na serveru se provede kontrola, zda navázaná metoda na ViewModelu má atribut `ClientSideMethod`. Pokud ano, provede se nahrazení dotazu na server voláním metody JavaScriptové. V opačném případě se nic neprovádí.

Kapitola 6

Závěr

Cílem této práce bylo vytvořit překladač, který bude překládat jazyk C# do jazyku JavaScript za pomoci frameworku Roslyn. Toto řešení poté mělo za cíl umožnit spouštět přeložený kód přímo v prohlížeči za použití frameworku DotVVM. Poslední částí řešení bylo vytvoření demonstrační aplikace, která bude využívat překladovou technologii.

V textové části této práce se nachází popis webového frameworku DotVVM, a to jak z pohledu uživatelského, tak z pohledu vnitřního fungování. Dále je diskutován popis jednotlivých vlastností frameworku Roslyn, který umožňuje získávat diagnostické informace nasbírané kompilátorem v průběhu překladu.

Implementace se skládá ze tří částí. První částí je samotný překladač, který byl realizován jako konzolová aplikace přijímající zdrojové soubory v jazyce C# a vytvářející odpovídající přeložené soubory v jazyce JavaScript. Druhou částí je napojení přeložených tříd do frameworku DotVVM na klientské úrovni. Třetí částí je implementace ukázkové aplikace, která tuto technologii používá.

V ukázkové aplikacei je možné vidět ukázky jednotlivých podporovaných konstrukcí. Od běžných syntaktických konstrukcí, jako jsou cykly, podmínky, binární operace a mnohé další, po ty složitější konstrukce, jako je tvorba objektů a cykly typu for-each. Dále je tam možné sledovat práci se zaregistrovanými metodami a vlastnostmi objektů (například operace s objektem typu `List`, případně vypisování do konzole,...). Všechny výše zmíněné části demonstrační aplikace fungují. Problém, který se zatím nepovedlo vyřešit, je přetečení číselného datového typu, které způsobí znemožnění jakékoliv další operace s aplikací.

Cíle této práce byly splněny. Navazující cíle této práce, na kterých je možno pracovat, jsou: zasadit se o začlenění těchto změn do oficiálního repozitáře DotVVM, rozšířit funkcionalitu o možnost napojení volání REST API a přidat možnost kompilace pouze změněných souborů.

Slovníček pojmů

Cascading Style Sheets je formát používaný pro popis vzhledu elementů na stránkách napsaných v jazyce HTML. 12

Document Object Model je objektově orientovaná reprezentace formátu XML a odvozených formátů. Poskytuje také rozhrání k úpravě dokumentu. 11

Gang Of Four Přezdívka používaná pro skupinu autorů stojících za knihou Design Patterns: Elements of Reusable Object-Oriented Software. Občas se používá se zapisuje zkráceně jako GoF[27]. 11

Model-View-ViewModel je architektonický vzor používaný při tvorbě programů, který rozděluje aplikaci na View (reprezentující prezentační logiku), ViewModel (reprezentující aplikační logicku) a Model (reprezentující datovou logiku) [4]. 1, 3, 9–11

autentizace je označení pro proces provádějící ověření identity dané osoby. Nejčastěji na základě uživatelského jména a hesla. 19

autorizace je označení pro proces provádějící kontrolu, zda má daná osoba právo provést danou akci. 1, 19

Command Command je způsob komunikace v návrhovém vzoru MVVM mezi View a ViewModelem. Při vyvolání události v rámci View dojde k odeslání zprávy vyvolávající metodu v rámci ViewModelu, tomuto provázání se říká Command [4]. 3, 10, 11, 13–15, 17, 19, 43

DataBinding (zkráceně též binding) Je způsob, jakým se v architektonickém vzoru MVVM ViewModel napojuje na View [4]. 13, 14

Model je implementace datové vrstvy programů popisující způsob uložení dat a způsob přístupu k nim [**?**]. 1, 3, 9–11

View Je vrstva aplikace definovaná v rámci návrhového vzoru MVVM, která popisuje vzhled a strukturu uživatelského rozhraní [4]. 1, 3, 5, 9–12, 14, 15, 17

ViewModel je vrstva aplikace definovaná v rámci návrhového vzoru MVVM, která má na starosti držení informací o stavu programu a specifikuje jeho chování [4]. 1, 3, 5, 9–12, 14–19, 42, 43

Zkratky

CSS Cascading Style Sheets. 12, 45

DOM Document Object Model. 11, 45

GoF Gang Of Four. 11, 45

JSON JavaScript Object Notation. 12, 14, 16, 17

MVVM Model-View-ViewModel. 1, 3, 9–11, 16, 45

Literatura

[1] Learn Roslyn Now: Part 8 Data Flow Analysis. [Online; navštíveno 17. 4. 2018].
URL https://joshvarty.com/2015/02/05/learn-roslyn-now-part-8-data-flow-analysis/

[2] Learn Roslyn Now: Part 9 Control Flow Analysis. [Online; navštíveno 17. 4. 2018].
URL https://joshvarty.com/2015/03/24/learn-roslyn-now-control-flow-analysis/

[3] DotVVM - Introduction. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/introduction/latest

[4] Shukkur, S.: Understanding the basics of MVVM design pattern. [Online; navštíveno 19. 11. 2017].
URL https://blogs.msdn.microsoft.com/msgulfcommunity/2013/03/13/understanding-the-basics-of-mvvm-design-pattern/

[5] Fowler, M.: *Patterns of enterprise application architecture*. Addison-Wesley, 2015.

[6] Gamma, E.; Helm, R.; Johnson, R. E.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2016.

[7] How to: Implement Property Change Notification. [Online; navštíveno 19. 11. 2017].
URL https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/how-to-implement-property-change-notification

[8] Knockout homepage. [Online; navštíveno 19. 11. 2017].
URL http://knockoutjs.com/

[9] DotVVM - Creating the First Page. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/basics-first-page/latest

[10] DotVVM - Binding Syntax. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/basics-binding-syntax/latest

[11] DotVVM - Value Binding. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/basics-value-binding/latest

[12] Albahari, J.; Albahari, B.: *C# 7. 0 in a Nutshell*. Sebastopol: O'Reilly Media, Incorporated, 7 vydání, 2017, ISBN 9781491987650.

[13] DotVVM - Command Binding. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/basics-command-binding/latest

[14] DotVVM - Static Command Binding. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/basics-static-command-binding/latest

[15] Seemann, M.; Block, G.: *Dependency injection in .NET*. Manning, 2012.

[16] The road to DotVVM 2.0: Part 2 - Static Command Services. [Online; navštíveno 16. 1. 2018].
URL https://www.dotvvm.com/blog/45/The-road-to-DotVVM-2-0-Part-2-Static-Command-Services

[17] DotVVM - ViewModels. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/basics-viewmodels/latest

[18] DotVVM - Validation. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/basics-validation/latest

[19] DotVVM - Authentication and Authorization. [Online; navštíveno 19. 11. 2017].
URL https://www.dotvvm.com/docs/tutorials/advanced-authentication-authorization/latest

[20] .NET Compiler Platform ("Roslyn") Overview. [Online; navštíveno 3. 4. 2018].
URL https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview

[21] Grune, D.; Bal, H. E.; Jacobs, C. J. H.; aj.: *Modern compiler design*. New York, Berlin: Springer, druhé vydání, 2012, ISBN 978-1-4614-4698-9.

[22] Aho, A. V.: *Compilers*. Boston: Addison Wesley, druhé vydání, 2007, ISBN 0-321-48681-1.

[23] *Roslyn Cookbook*. Birmingham: Packt Publishing, první vydání, 2017, ISBN 1787286835.

[24] Misek, J.; Zavoral, F.: Control Flow Ambiguous-Type Inter-Procedural Semantic Analysis for Dynamic Language Compilation. *Procedia Computer Science*, ročník 109, č. 1, 2017: s. 955–962, ISSN 1877-0509, doi:10.1016/j.procs.2017.05.452.
URL https://www-sciencedirect-com.ezproxy.lib.vutbr.cz/science/article/pii/S1877050917311341

[25] Flanagan, D.: *JavaScript*. Sebastopol: O'Reilly, fifth span class vydání, 2006, ISBN 0-596-10199-6.

[26] Meyer, B.: *Object-oriented software construction*. Upper Saddle River: Prentice Hall, druhé vydání, 1997, ISBN 0-13-629155-4.

[27] Gang Of Four Wiki. [Online; navštíveno 15. 1. 2018].
URL http://wiki.c2.com/?GangOfFour

Přílohy

Příloha A

Obsah CD

V této příloze se nachází popis obsahu CD, které je přiloženo k originálnímu výtisku této práce. Na přiloženém CD se nachází elektronická práce ve formátu PDF, originál zdrojových souborů LaTeXa zdrojové kódy frameworku DotVVM včetně provedených úprav. Struktura disku z pohledu kořenového adresáře vypadá následovně:

- *src/* - adresář obsahující zdrojové kódy frameworku DotVVM. V podadresáři *DotVVM.TypeScript.Compiler* se nachází projekt překladače a v podadresáři *DotVVM.Samples.Common* se nachází projekt obsahující ukázkové kódy aplikace;
- *docs-src/* - adresář obsahující original zdrojových kódů této práce ve formátu LaTeX;
- *docs-build/xmrnus01.pdf* - dokument ve formátu PDF obsahující bakalářskou práci v takovém stavu, v jakém byla odevzdána do informačního systému.