Szegedi Tudományegyetem
Informatikai Intézet

Ipari JavaScript elemző kiegészítése TypeScript támogatással

Industrial JavaScript analyzer enhancement with TypeScript support

Szakdolgozat

Készítette:

Pozsgai Alex

programtervező informatikus BSc szakos hallgató

Témavezető:

Dr.Antal Gábor

egyetemi docens

Szeged
2023

Feladatkiírás

Manapság kezd a piacon egyre több kódelemező megjelenni, többek között JavaScript programozási nyelvre is. Jelenlegi projekt is egy JavaScript Analyzer. A cél az, hogy a projekt több legyen, mint a piacon a többi kódelemező, ezért bővíteni kell TypeScript támogatással.

A hallgató feladata ennek a programnak (JSAN) a fejlesztése úgy, hogy tudjon TypeScript fileokat és projekteket is kellő pontossággal elemezni. Ezután pedig a meglévő programot optimalizálni, hogy kevesebb erőforrást vegyen igénybe a futtatása, és hogy gyorsabban fusson le.

A megoldási módszerek a hallgató kreativitására vannak bízva.

Tartalmi összefoglaló

A téma megnevezése:

JavaScript Analyzer kiegészítése TypeScript supporttal, JSAN optimalizálása.

A megadott feladat megfogalmazása:

A feladat során el kell érni azt, hogy a JavaScript Analyzer Tool tudjon TypeScript fileokra lefutni, és azokat nagy pontossággal elemezni. Emellett a JSAN működését optimalizálni.

A megoldási mód:

A megoldás során kell változtatni a JavaScriptSchémán, és a JSAN-ban az AstTransformer.js fileban.

Alkalmazott eszközök, módszerek:

A megoldáshoz Visual Studio Code IDE-t, a JavaScriptSchema szerkesztéséhez Visual paradigm-t használta. A projekt lebuildeléséhez és ellenőrzéséhez Visual Studio 2017 programot használtam.

Elért eredmények:

JSAN képes TypeScript fileokat nagy pontossággal elemezni, nagyobb projektre lényegesen gyorsabban fut le, kevesebb erőforrást igényel, mint előtte.

Kulcsszavak:

JavaScriptAnalyzer, TypeScriptAnalyzer, JavaScript, TypeScript, Optimalizálás, Visual Studio Code, Visual Paradigm, Vpp, C++

Tartalomjegyzék

Bevezetés

Szakdolgozatomban a JavaScript Analyzer Tool továbbfejlesztése a cél. A továbbfejlesztés arról szól, hogy a javascript mellett előtérbe jön a typescript is. Ezáltal a typescript fileokat és projekteket is elemezni kell. Ez azt eredményezi, hogy a Javascript Analyzer Toolt gyökerestül át kell írni, annak érdekében, hogy JavaScriptet és TypeScriptet is tudjon egyaránt elemezni.

A JavaScript Analyzer Tool az egy nagyobb projektnek az egyik alprojektje. Ezt a nagyobb projektet Analyzer-JavaScriptnek hívják. Az Analyzer-Javascriptnek több alprojektje is van, amiben sok minden egymásra épül. A szakdolgozatomban a következő alprojektekben fogok változtatni: JSAN, és az erre épülő JSAN2Lim. Emellett még megtalálhatóak a következő alprojektek is: ESLintRunner, ESLint2Graph, LIM2Metrics, LIM2Patterns, DuplicatedCodeFinder és a ChangeTracker. Azért kell a JSAN2Limben is változtatni, mivel ha gyökerestül megváltoztatom a JSAN-t, akkor a JSAN2Lim rossz eredményeket fog visszaadni a typescript fileok elemzése közben.

A projekten amin fejlesztettem, azon többen is fejlesztettek egyszerre, ezáltal voltak átfedések, oszthatatlan részek. A projekt legtöbb része egymásra épül, ezért kiemelten fontos volt a csapatmunka egyes részeknél, hogy a projekt eredményesen és hatékonyan haladjon. Az eredményes csapatmunka magában foglalja az eredmények és az előrehaladás folyamatos kommunikálását, ami azt jelenti, hogy mindketten megértjük, hogy milyen célkitűzések vannak a projektben, és rendszeresen jelentést adunk egymásnak az elvégzett munkáról és az elért eredményekről.

Fejlesztés során úgynevezett JavaScriptSchema.vpp file szerkesztése is csoportos munka volt, hiszen minden erre épült, ez volt az alapja mindennek. Sok időt vett igénybe a szerkesztés. Nulláról kellett kellett újraírni az egész vpp filet. Sajnos dokumentáció nem készült az előző filehoz, ezért nekünk kellett kitalálni, hogy mi mit csinált, hiszen akik ezt

írták, ők már nem foglalkoztak ezzel. A szerkesztéshez szükséges volt a Visual Paradigm alkalmazást használni. Ezzel egyikőnk sem találkozott még, szóval először ezt kellett tanulmányozni, megérteni. Ezután értelmezni kellett a meglévő schemát, hiszen eddig az jól működött, csak nem lehetett könnyen bővíteni. Mérlegeltük a két opciót, ahol vagy megpróbáljuk bővíteni a jelenlegi schémát, vagy nulláról elkezdjük újraírni. Végül az újraírás mellett döntöttünk, hiszen ezt láttuk gyorsabb és könnyebb megoldásnak. Egy könnyen bővíthető, dokumentál schema volt az elképzelés. Ketten fejlesztettük le végül ezt a schemát.

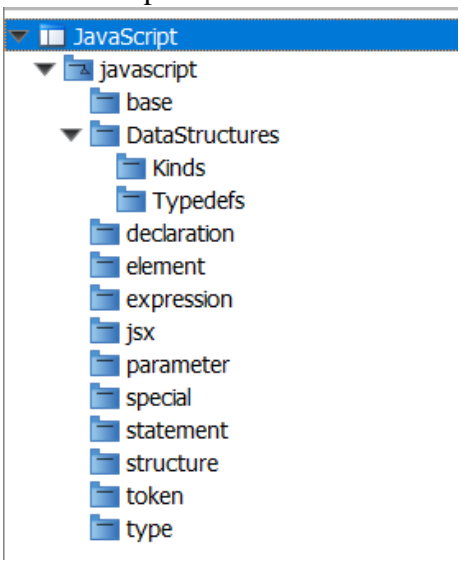
1. fejezet

JavaScriptSchema átírása

1.1. A JavaScriptSchemáról

A JavaScriptSchéma egy UML Diagramhoz hasonlító schéma. Jelen esetben a Visual Paradigm alkalmazással szerkeszthető. A felépítése a következőképpen alakul:

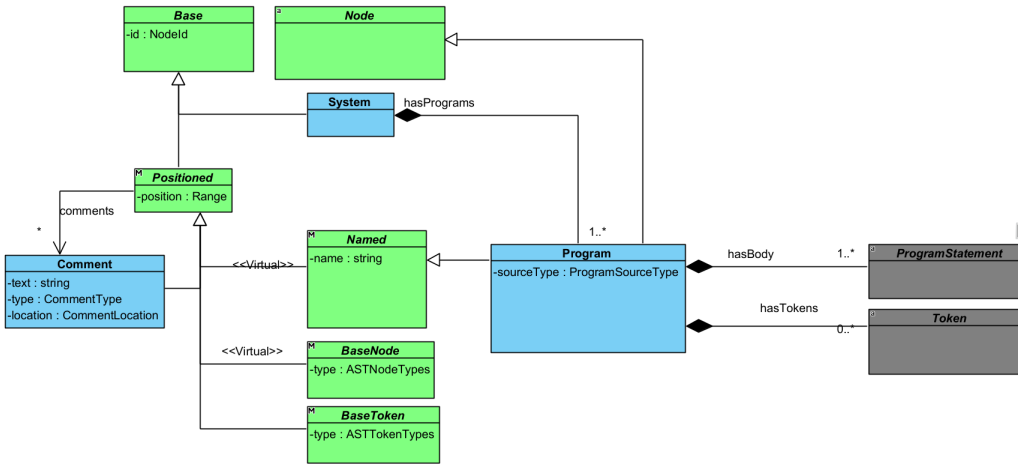
1.1. ábra. JavaScriptSchema struktúrális felépítése



Az 1.1 ábrán a következő struktúra figyelhető meg: ProjektNév-Model-Packagek. A projekt neve a JavaScript, ezen belül található egy model, amit javascript-nek hívnak. A modellen belül találhatóak meg a packagek. A Packagek nem véletlenül így lettek elnevezve. Az alábbi hivatkozáson található meg, hogy milyen logika alapján neveztük el a packageket: [1] Nem feltétlenül muszáj több packaget létrehozni, ez csupán az átlátható-

ság és a könnyen bővíthetőség céljából lett így megvalósítva. Követtük a typescript-eslint githubon lévő projekt struktúrális logikáját, több helyen is eltértünk tőle, mivel a mi projektünk másabb, illetve speciális megoldásokat igényelt egyes helyeken. A következő oldalakon bemutatom a packagek felépítését néhány egyszerűbb példán szemléltetve. A packagek a következőképpen épülnek fel:

1.2. ábra. A base package felépítése



Az 1.2 ábrán különböző osztályok láthatóak. A Base osztály mindennek az alapja, ebből öröklődik minden más. Látható, hogy egy id attribútummal rendelkezik, aminek a típusa NodeId. A zöld háttérszínű osztályok az absztrakciót jelentik. Működésben nincs jelentősége, csak a schéma szerkeszthetősége és olvashatósága miatt van így jelezve. A szürke háttérszínű osztályok pedig csupán annyit jelentenek, hogy más packageben lettek definiálva. A kék a default, normális osztályt jelölik. A Positioned és a System származik le a Baseből, értelemszerűen a system az maga a program lesz. A Positioned az azért absztrakt, mivel majd ebből fognak leszármazni a kisebb osztályok, mint például az expression, statement és a többi. A Positioned osztálynak van egy attribútuma, a poisition, ami egy Range típus. Emellett még tartozhatnak hozzá kommentek is. A komment egyaránt leszármazik a positioned-ből, ezzel garantálva azt, hogy a kommentnek van pozíciója. Látható, hogy a kommentnek van egy text, type és location attribútuma. A CommentType és a CommentLocation a DataStructures-ben van definiálva. Emellett a Named, BaseNode és a BaseToken származik le a Positionedből. A Named osztály azt jelenti, hogy egy nodenak van-e name attribútuma vagy sem. A BaseNodeból és a BaseTokenből fog nagyon sok minden leszármazni aminek van typeja. Végül, megtalálható a Program osztály,

aminek van name attribútuma, mivel Named-ből származik le, és van Systeme. Az 1..* jelenti azt, hogy legalább 1 Systeme van, de lehet több is. A hasPrograms-nak majd máshol lesz jelentősége, a mi esetünkben majd a javascriptben. Tetszőlegesen el lehet nevezni, de konzisztencia miatt, minden attribútum ami osztály (és nem a DataStructuresben azon belül is a Kindsban van definiálva, hanem van neki egy osztály, mint pl Comment) azt a hasOsztály névvel fogjuk ellátni. A ProgramSourceType is a Kinds packageben van definiálva, ami csupán azt mondja meg, hogy az adott program vagy script vagy module típusú. A base package nem követi a typescript-eslint githubon lévő projekt base mappáját, ez teljesen egyedi, átememeltük az egészet apróbb módosítással az előző projekt verzióból.

A base packagen kívül bemutatom a declaration packaget, hogy lássuk milyen logikát követtünk a megírás során. Ezen belül is az ExportAllDeclaration bemutatása:

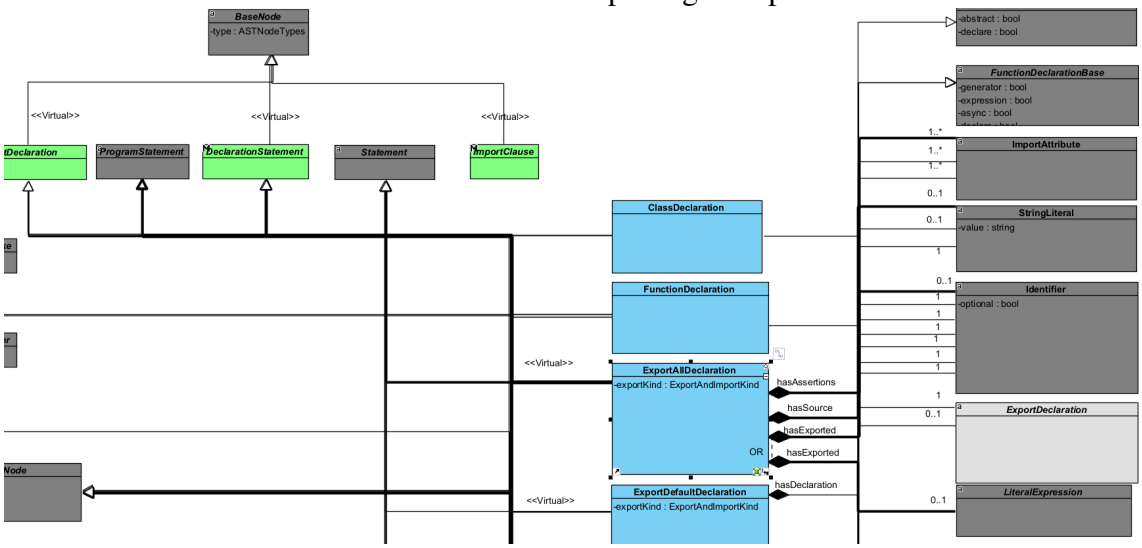
```
1 export interface ExportAllDeclaration extends BaseNode {
2       type: AST_NODE_TYPES.ExportAllDeclaration;
3       assertions: ImportAttribute[];
4       exported: Identifier | null;
5       exportKind: ExportKind;
6       source: StringLiteral;
7 }
```

Kódrészlet 1.1. ExportAllDeclaration typescriptes megvalósítása

Az 1.1 kódrészlet így lett megvalósítva a JavaScriptSchémában:

Az 1.3 ábrán látható az, hogy minden a BaseNodeból származik. Az 1.1 kódrészletben ez az extends BaseNode. Itt maga a declaration osztály az a DeclarationStatement névre hallgat, csupán azért, mert követtük a typescriptes kódot. A főbb osztályok a BaseNode alatt találhatóak. Jobb oldalt a sötétszürke és a világosszürke osztályok azok az attribútumok. Ebben az esetben az ExportAllDeclaration osztály leimplementálását mutatom be a JavaScriptSchemában. Az ExportAllDeclaration öröklődik a Statement, DeclarationStatement, Node és a ProgramStatementből. Ezeket az öröklődéseket ugyanúgy a typescript-eslint githubról néztük, unions mappában érhetőek el. Ezáltal megkapja az összes szülőnek a tulajdonságait. A DeclarationStatement öröklődik a BaseNodeból, ami azt jelenti, hogy a BaseNode tulajdonságait is megkapja az ExportAllDeclaration. A BaseNode ugye meg származik a Positionedből, ez látható az 1.2 ábrán. Emiatt az

1.3. ábra. A declaration package felépítése

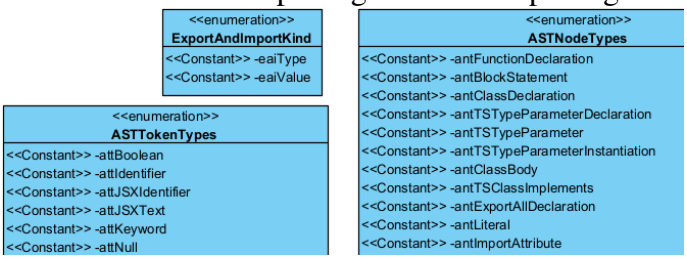


ExportAllDeclaration-nek lesz pozíciója, kommentje és NodeId-ja is. Assertions attribútum típusa az ImportAttribute, látható, hogy egy tömböt vár, ezért 1..* a multiplicityje. Exported attribútumnál egy Identifier típusú attribútumot vár, itt mi kiegészítettük még egy LiteralExpressiönnel is, ami maga a Literal. A vagyolást egy OR-al jeleztük a JavaScriptSchemában. Mivel lehet null-is, ezért 0..1 a multiplicity, szóval vagy 0 vagy 1. Az ExportKind az a Kindsban található meg, így szimplán csak arra hivatkozunk, mint ExportAndImportKind. Végül a source attribútuma egy StringLiteral, nálunk is így szerepel. Természetesen ami sötétszürkével van jelölve, az máshol létre van hozva és vannak neki attribútumai. A világosszürke annyit jelöl, hogy ebben a packageben lett létrehozva az osztály, de láthatóság szempontból többször szerepel, mivel lehet attribútum is. Így lett minden egyes osztály felépítve, természetesen Az 1.3 ábra csak egy részlete a declaration packagenek, ennél jóval nagyobb. Vannak úgynevezett gyűjtő osztályok is, mint pl a Statement, ezekre azért van szükség, mert majd később ha le lett generálva minden, akkor tudjunk majd vizsgálni különböző nodeokra, mint például az isStatement.

Sokat említettem a DataStructurest. Hadd mutassam be egy példán keresztül, hogy ez hogy néz ki:

A DataStructures packageben található 2 package, ez az 1.1 ábrán látható. Ebből a Kinds packaget mutatom be. Az 1.4 ábrán látható egy kis szelet a Kinds packageből. Enumok találhatóak ebben a packageben. Például ha az ExportAndImportKind-ot adjuk meg típusnak az egyik attribútumnak, akkor az lehet vagy eaiType vagy eaiValue. Minden

1.4. ábra. A DataStructures packageben a Kind package felépítése



constant előtt van 3 karakter, ez azért szükséges, mert később ezekkel még foglalkozni fogunk.

A JavaScriptSchema magában még semmit sem csinál. Először ki kell exportálni az egész projectet xml formátumba. Ezután az xml filet átkonvertáljuk egy asg filera. Ez úgy történik, hogy a JavaScript Analyzer projektnek van egy kisebb alprojektje, amit UmlToAsg-nek hívnak. Ez a java kód átkonvertálja az xml fileban lévő adatot egy asg fileba. Jobban nem térnék ki az UmlToAsg projektre, mivel nem szerkesztettük. Az UmlToAsg projektet a SchemaGenerator generálta le. A SchemaGenerator c++ nyelven megírt program. Több mindent is generál c, c++, vagy java nyelven. Ez a projekt is a JavaScript Analyzer alprojektei közé tartozik. A legenerált asg file a következőképp néz ki: A file elején a következő található meg:

```
1 NAME = javascript;
2
3 APIVERSION = 0.3.1;
4 BINARYVERSION = 0.3.1;
5 CSIHEADERTEXT = JavaScriptLanguage;
```

Kódrészlet 1.2. Asg file első sorai

A verziókat kézzel tudjuk átírni abban a fileban ami generálja ezt, ez egy c++ file, a SchemaGeneratorban található meg. Ezután a Kinds mappa tartalmát írja bele a következőképpen:

```
1 KIND ASTNodeTypes (ant) {
2       FunctionDeclaration;
3       BlockStatement;
4       ClassDeclaration;}
```

Kódrészlet 1.3. Asg file kind

Az a 3 karakter amit minden constant elé tettünk azt kitette paraméterbe és csak az utáni stringet írta át. Ugyanígy van a többi kindnál is. Ha az összes kindot beleírta, akkor kezdi írni sorban a többi packaget. Az 1.1 alapján megy sorba. Példának a declaration packaget mutatom be, azon belül is az ExportAllDeclaration-t.

```
1  SCOPE declaration {
2
3        NODE DeclarationStatement : virtual base::BaseNode [ABSTRACT] {
4        }
5
6        NODE ExportAllDeclaration : DeclarationStatement, statement::
             Statement, virtual statement::ProgramStatement, special::Node
              {
7              ATTR ExportAndImportKind exportKind;
8              EDGE TREE 1 hasExported (expression::Identifier |
                   expression::LiteralExpression);
9              EDGE TREE 1 hasSource (structure::StringLiteral);
10             EDGE TREE * hasAssertions (special::ImportAttribute);
11       }
12 }
```

Kódrészlet 1.4. Asg file ExportAllDeclaration

Látható, hogy a Packaget SCOPE-nak értelmezi, és ezen belül NODE-ok találhatóak. A Nodeoknak ATTR és EDGE TREE van. A vagyolás is látható. Az öröklődik egy kettőspont után, felsorolás szerűen írta át, ha valami másból származik, akkor packageNev::osztalyNev szerint. Az kapott ATTR jelölést ami a Kinds-ban megtalálható vagy egy szimpla típus (mint pl string, int). Minden mást EDGE TREE-nek nevezett el. Az Edge tree utáni szám vagy csillag az a multiplicitást jelenti. Ha 1es, akkor a multiplicitás 0..1, ha *, akkor vagy 0..* vagy 1..* a multiplicitás. Az 1.3as ábrán látható, hogy mit hogyan írt át.

1.2. JavaScriptSchema átírása

A JavaScript Analyzer a JavaScriptSchemára épül. Ha valamit meg akarunk változtatni gyökerestül a JavaScript Analyzerbe, akkor a schémát is változtatni kell. Azt elérni,

hogy javascript mellett még typescriptes kódokat is elemezzen a JavaScript Analyzer, ahhoz gyökerestül meg kellett változtatni a JavaScript Analyzert. Az előző JavaScriptSchema (ami csak javascriptet elemzett) az a javascript hivatalos oldala alapján készült. Ami ábrák fentebb megtalálhatóak, azok már az átírt schémából vannak. Több opció is volt, hogy most vagy legyen átírva a schéma, vagy legyen egy külön typescriptre is írva. Végül rájöttünk, hogy ha van egy typescriptes schémánk, az képes javascriptet ugyanúgy elemezni ha megfelelően van lefejlesztve. Így egy schémát fejlesztettünk le. Az előző schéma átláthatatlan volt, ezért úgy döntöttünk, hogy majdnem a nulláról újraírjuk. Egyedül a base packaget emeltük át a régiből, BaseNode-al és BaseTokennel kibővítve (Az 1.2 ábrán látható). Emellett el kellett dönteni, hogy milyen struktúrát kövessünk, ami jól átlatható és később könnyebben bővíthető. Végül a typescript-eslint official github alapján haladtunk. Annyi változtatással, hogy ami nekik a base mappában volt, mi arra létrehoztunk egy külön structure packaget. Mindenhez készítettünk dokumentációt, jól érthetően leírtuk, hogy mit miért kellett csinálni. Mivel minden is egymásra épül a typescriptes schémában, ezért nem tudtuk tesztelni minden egyes package után, hogy működik-e vagy sem. Mint például, látható az 1.4 kódrészleten, hogy az ExportAllDeclaration mennyi mindenből származik le vagy mennyire sok attribútuma van. Az ExportAllDeclaration egyik attribútuma, az assertions típusa az ImportAttribute. Ahhoz, hogy jól észrevegye az ExportAllDeclarationt az Analyzer, ahhoz jól le kellett fejleszteni az ImportAttributet.

```
1 export interface ImportAttribute extends BaseNode {
2       type: AST_NODE_TYPES.ImportAttribute;
3       key: Identifier | Literal;
4       value: Literal;
5 }
```

Kódrészlet 1.5. ImportAttribute

Ahhoz, hogy az ImportAttributet letudjuk fejleszteni a schémában, ahhoz le kell fejleszteni az Identifiert, és a Literalt. Kicsit összetett, hogy mi minden épül egymásra. Ebből adódik a következő alfejezet mondandója is.

1.3. Nehézségek, problémák

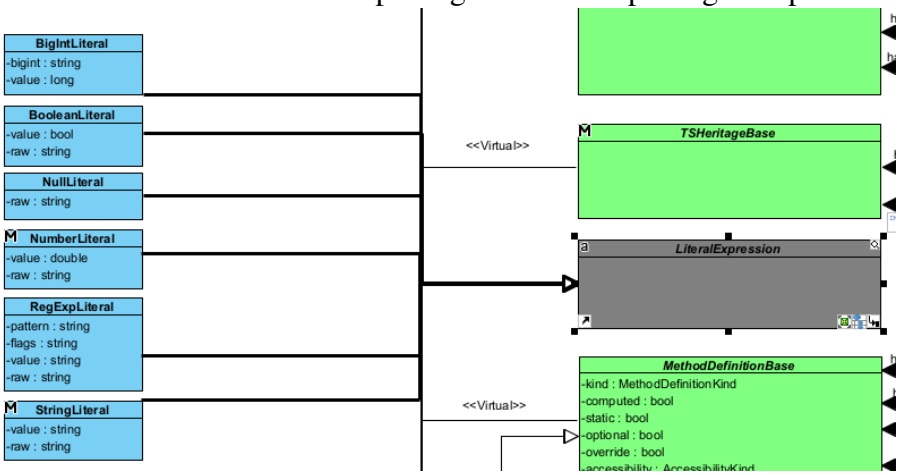
A schéma átírása során több nehézségbe is ütköztünk. A legelső nehézség az a Visual Paradigm program korrekt használata volt. Egy kisebb időbe tellett rájönni arra, hogy egy adott classnak hogyan kell megváltoztatni a háttérszínét, packagek közötti mozgást, hogyan kell egy adott classra hivatkozni ami másik packageben volt. Ezután ami szerintem a legnagyobb nehézség volt, az az, hogy nem volt dokumentáció az előző schémához. Mindent nekünk kellett kitalálni, hogy mit miért csináltak az előző fejlesztők. Akik ezt a schémát fejlesztették le, ők már nem foglalkoztak ezzel, és más nem nagyon mélyedt bele ebbe az egészbe azóta. A következő nagyobb nehézség, az az volt, hogy összehozzunk egy olyan schémát ami működőképes, és az analyzer ezt tudja is használni. Előző alfejezetben említettem, hogy egy adott classt (mint például ExportAllDeclaration) lefejlesszünk, ahhoz sok minden mást is le kell fejleszteni, amihez meg még több minden kellett. Ebből adódóan tesztelni nagyon nem tudtunk, mivel az egésznek működnie kellett ahhoz, hogy az analyzer egyáltalán lefusson. Mivel mi majdnem nulláról írtuk újra, ez volt a hátrány. Amikor elérkeztünk ahhoz az állapothoz, hogy mindenre rámondjuk azt, hogy működik, akkor jött egy nagyobb probléma. Valahol Segmentation Fault-ot kapott a program, és nem nagyon tudtuk ezt debugolni. Több sejtésünk is volt, hogy mi lehet a baj. Emiatt az egész schémát át kellett nézni alaposan, hogy hol vétettünk hibákat. Sok helyen voltak pontatlanságok, rossz osztályból származtattunk le, rossz típus volt megadva attribútumnak. Ezeket mind kijavítottuk, de most már más hibát kaptunk. A JavaScript Analyzerben kiderítettük, hogy pontosan hol szállt el a program. A Literal volt a hiba, mivel ezt nem a github alapján írtuk meg, hanem egyedi ötlettel. Előző schémában is máshogy volt megoldva.

Az 1.5 ábrán látható, hogy a LiteralExpressionből (Ami a Literal) több literal is öröklődik.

```
1 export interface LiteralBase extends BaseNode {
2   type: AST_NODE_TYPES.Literal;
3   raw: string;
4   value: RegExp | bigint | boolean | number | string | null;
5 }
```

Kódrészlet 1.6. Literal

1.5. ábra. A DataStructures packageben a Kind package felépítése



Próbáltuk úgy megoldani, hogy 4 vagyolással 4 különböző literál tartalmazza, de az analyzerben máshogy kezeltük le a literalt, mint nodeot. Ezért a tartalmazás helyett inkább örököltettünk a literalból, így megoldva ezt a problémát. Még az is probléma volt, hogy volt egy LiteralBaseünk, amiből származott ez az 5 literal. A LiteralBase származott a LiteralExpressionből, de valamiért Segmentation fault lett a vége ha így próbáltuk megoldani. Ezért a LiteralBase-t kivettük, és LiteralExpressionből származik minden literal. Végül még az volt a nehézség, hogy kódrészletet keressünk az analyzernek, amin le tudjuk tesztelni, hogy az analyzer jó outputot ad-e.

Mivel nem nagyon volt typescriptes háttértudásunk, először a typescriptet kellett átnézni, hogy mit hogyan tudunk megvalósítani. Emellett még a javascriptes outputokat is át kellett nézni, mivel a mi célunk a fejlesztés volt. A jelenlegi javascriptes projektekre ugyanolyan eredményt adott, sőt néhány helyen jobbat is, mert az előző schémában is voltak pontatlanságok. A typescriptes projektek nagy részét tudja elemezni az analyzer, de közel sem tökéletes, mivel egyfolytában kellene fejleszteni a schémát, mivel a typescript nem annyira régi.

2. fejezet

JavaScriptAddon

2.1. JavaScriptAddonról

A JavaScriptAddon az egy node kiterjesztésű file, amit a SchemaGenerator generál. Minden egyes nodenak külön generál nodeWrapperheader és egy nodeWrapperCC filet. A SchemaGenerator által generált Factory.cc, Factory.h és az összes nodeWrapper.cc és a hozzátartozó nodeWrapper.h fileok összemergelése eredményezi a javascriptAddon.node filet. A SchemaGenerator a JavaScriptSchemából és az ebből generált asgből generálja le a javascriptAddon.node filet. A SchemaGeneratornak több generálása is definiálva van, hiszen nem csak erre használják. Nekünk a NodeAddonGenerator.c fileban és a hozzátartozó header fileban van minden. A main.c-be ezt beincludeolja, és ennek hívja meg egyes metódusait, ami által elkészül a javascriptAddon. A következő pár oldalon bemutatnám, hogy a generálás hogyan zajlik. A SchemaGeneratornak rengeteg kapcsoloja van, köztük a generateNodeAddon is. Ezekre a kapcsolókra vizsgál egyesével, egy nagy if-elseben.

```
1 else if(!strcmp(argv[i], "-genNodeAddon")){
2       options.generateNodeAddon = true;
3 }
```

Kódrészlet 2.1. SchemaGenerator kapcsoló vizsgálás

A kód elején történik meg ez, ha meg van adva paraméternek a genNodeAddon string, akkor az options.generateNodeAddon-t igazra állítja. A default értéke false. Ezután jóval lentebb miután rengeteg mindent legenerált ami kell alapból is a normális működéshez, vizsgál egyet az options.generateNodeAddon-ra. Ez látható a 2.2 kódrészleten is. Else

ága nincs, szóval semmi nem történik ha nincs megadva a genNodeAddon string a paramétereknél.

```
1  if(options.generateNodeAddon ){
2        debugMessage(0, "Generating Node.JS Addon sources\n");
3        if (createAndEnterDirectory(SOURCE_NODE_ADDON_DIR_NAME)) {
4        generatePackageJson();
5        generateBindingGyp();
6        generateAddonCC();
7        generateFactoryWrapper();
8
9        if (createAndEnterDirectory("inc")) {
10       generateWrapperHeaders();
11       leaveDirectory();}
12       if(createAndEnterDirectory("src")){
13       generateWrapperSources();
14       leaveDirectory();}
15
16       leaveDirectory();}
17 }
```

Kódrészlet 2.2. SchemaGenerator javascriptAddon generálás

A createAndEnterDirectory metódus annyit csinál, hogy létrehoz egy mappát és chdirrel belelép. Ha az adott mappa már létezik, akkor csak szimplán belelép. A SOURCE NODE ADDON DIR NAME változó jelen esetben addon értéket fog kapni, hiszen a javascriptAddon dolgai ebbe fognak generálódni. Miután létrehozta és/vagy belelépett az addon mappába először legenerálja a package.json filet a projekthez. A package.json fileban beállítja a projektnevét, verziószámát, dependencyket, scripteket és a végén a gypfile kapcsolónak egy true-t beállít.

```
1  fprintf(f, "    \"rebuild\": \"node-gyp configure && node-gyp rebuild -
       j 8\",\n");
2  fprintf(f, "    \"install\": \"node-gyp configure && node-gyp build -j
       8\"\n");
```

Kódrészlet 2.3. NodeAddonGenerator package.json scripts

A 2.3 kódrészleten látható, hogy majd gyp segítségével fog történni a generálás.

Ezután legenerálja a binding.gyp filet. Ez a file fog felelni azért, hogy a javascriptAddonhoz sikeresen generálódjanak le a wrapperek és azok headerjei. Ha ez megtörtént, utána fogja legenerálni az addonCC-t. Az addon.cc fileban beimportálja a Factory.h headert, amiben több metódus is megtalálható. ,

```
1 if (!traversalDescendantBFT(rootNode, generateWrapIncludes, false)) {
2       debugMessage(0, " failed\n");
3       fclose(f);
4       return false;
5 }
```

Kódrészlet 2.4. Addon.cc Wrapperek includolása

A 2.4 kódrészletben a traversalDescendantBFT metódust egy másik projektből includoltuk. Ez egy bejárás, ami az összes nodera lefut, és az összes nodera meghív egy metódust, jelen esetben a generateWrapIncludes metódust. Ez a generateWrapIncludes a nodeAddonGenerator.c fileban van leimplementálva a következőképp: ,

```
1 if(node->type.abstract){
2       return true;
3 }
4 fprintf(f, "#include \"");
5 fprintf(f, "inc/%sWrapper.h\"\n", node->name );
6 return true;
```

Kódrészlet 2.5. generateWrapIncludes leimplementálása

Először megvizsgálja, hogy az adott node abstract-e, ha nem akkor tovább megy, és az addon.cc-be includolja az adott nodeWrappernek a headerjét. A node->name lehet például FunctionDeclaration, ClassDeclaration, amik megtalálhatóak az asg fileban.

Ezután ezt a bejárást mégegyszer végrehajtja, de most a wrapperIniteket generálja az addon.cc fileba. Annyi változtatással, hogy picit mást ír az addon.cc fileba.

```
1 fprintf(f, "  columbus::%s::asg::addon::%sWrapper::Init(env, exports);\
    n", schemaName, node->name);
```

Kódrészlet 2.6. generateWrapInit leimplementálása

Ha ezek megtörténtek, akkor az addon.cc filet legeneráltuk sikeresen, és így benne találhatóak a wrapperek headerjének includolása és a wrapperek Initjei.

Ezután következik egy generateFactoryWrapper hívás. A generateFactoryWrapperben 2 metódus hivás található, a generateFactoryWrapperHeader és generateFactoryWrapperSource.

A generateFactoryWrapperHeader a Factory.h filet generálta le a következőképp: Először beincludolja az összes nodeWrappernek a headerjét, és utána létrehoz egy Factory osztályt, publikus metódusait, ami Init és Destructor, illetve a private metódusait, ami a destructor, New, SaveAST, LoadAST, CLear, getRoot és az összes nodeWrapper create metódusa. Erre azért van szükség, mivel majd a JSAN-ban ezeket a wrappereket fogjuk létrehozni és szerkeszteni.

A generateFactoryWrapperSource pedig a Factory.cc filet generálja le. Ebben a fileban pedig meg van az összes metódus valósítva, ami a headerjében található. Emellett a Factory Initjének a props változója a következőképp alakul:

```
1 napi_property_descriptor props [] = {
2       DECLARE_NAPI_METHOD("getRoot", getRoot),
3       DECLARE_NAPI_METHOD("createCommentWrapper", createCommentWrapper)
4       ...}
```

Kódrészlet 2.7. Factory.cc file

DeclareNapiMethodokat hoz létre, az összes nodeWrappernek, ezek majd a JSAN-ban lesznek használatosak.

Utolsó lépésként, először az inc mappát majd az src mappát generálja le a SchemaGenerator. Az inc mappában találhatóak a header fileok egyes wrappereknek, az src mappában maguk a wrapperek vannak megvalósítva. Minden nodenak hoz létre wrapper filet, és ezekben valósít meg néhány metódust. Vegyük most egy példának az ExportAllDeclarationt. Létrehoz egy ExportAllDeclarationWrapper.h és egy ExportAllDeclaration.cc filet. A header fileban létrehoz egy ExportAllDeclarationWrapper osztályt, ami származik a BaseWrapperből. A BaseWrapper az egy alap Wrapper osztály, amit bővíteni fogunk. Három publikos metódust generál, Init, Destructor és a NewInstancet. Ezen felül több private metódust is.

```
1 napi_property_descriptor props [] = {
2       static napi_value setPosition(...);
3       static napi_value addAssertions(...);
4       static napi_value setType(...);
```

```
5 ...}
```

Kódrészlet 2.8. ExportAllDeclaration.h file

Látható a 2.8 kódrészleten, hogy létrehozott az ExportAllDeclarationWrappernek egy setPosition, addAssertions, setType és még több metódust. Ezek az ExportAllDeclaration attribútumai, amit a JavaScriptSchemában beállítottunk. Az 1.3 látható, hogy az ExportAllDeclaration hasAssertions attribútumának a multiplicitása 1..*, ezt átkonvertálta addAssertionsre. Ahol 0..* a multiplicitás, azt átkonvertálta setAttribute-ra, pl setSource vagy setExportedre. Minden nodeWrappernek van olyan metódusa, hogy setPath, setPosition, setType, addComments, hiszen minden a BaseNode vagy a BaseTokenből öröklődik le.

Az ExportAllDeclaration.cc fileban ugyanaz történik, mint ami volt a Factory.cc fileban. Declare napi methodokat generál, a setExported, setSource, setType és a többi metódushoz. Ezután az összes metódust megvalósítja.

2.2. JavaScriptAddon változások

A JavaScriptSchema változtatások után, a NodeAddonGeneratort nem módosítottuk. Ennek ellenére a javascriptAddon.node mérete megduplázódott, csupán azért, mert a schéma ekkora bővítésen esett át. Sokkal több nodeWrapper lett, mivel bejöttek a typescript általi dolgok is a javascript mellett. Természetesen a javascriptes dolgok megmaradtak, így javascriptet ugyanolyan jól tud elemezni, sőt néhány esetben jobban is, mert voltak figyelmetlenségek előző verzióban.

Mivel a javascriptAddon mérete nőtt, ezáltal a sebesség csökkent. Ezt majd egy másik fejezetben jobban kifejtem, célunk az volt, hogy először tudjon typescriptes fileokat és projekteket elemezni a jsan.

3. fejezet

JSAN2Lim átírása

3.1. JSAN2Limről

3.2. Bővítések

3.3. Miket nem detektált

4. fejezet

AST Binder Optimalizálás

4.1. A binderről

Az AST binder referenciákat bindol. Ezek a referenciák a JSAN2Lim programhoz kellenek. Binder néven van definiálva az astTransformer.js fileban. Kétszer van használva, ezért is kulcsfontosságú, hogy gyors legyen. Egyszer VariableUsages referenciákat bindol és egyszer ACG referenciákat bindol.

A binder 4 argumentumot vár:

- Egy stringet, ami lehet vagy VU (ami VariableUsages-t rövidíti) vagy ACG (egy javascript callgraph).
- Egy Abstract Syntax Tree-t (AST), ami egyedien van felépítve.
- Egy tömböt, amiben JSON objektumok találhatóak, amik a linkeket tartalmazzák.
- Végül még egy stringet, ami lehet addCalls, vagy setRefersTo. Az AddCalls és a SetRefersTo a javascriptAddonban található meg és onnan hívódik meg. Akkor addCalls ha ACG referenciákat bindolunk és akkor setRefersTo ha VU referenciákat.

A linkeket tartalmó tömb egyik JSON objektuma a következőképp néz ki:

```
1 source: {
2       label: IdentifierNeve,
3       file: AbsPath,
```

```
4       start: { row: <int>, column: <int>},
5       end: { row: <int>, column: <int>},
6       range: { start: <int>, end: <int>},
7       node: [Object]
8   },
9   target: {...}
```

Kódrészlet 4.1. Binder JSON argumentuma

A 4.1 kódrészletben egy JSON elemét láthatjuk a tömbnek. Ilyen JSON elemekből épül fel a tömb. A source és a target felépítése ugyanaz, csupán másak az értékek bennük. A kódrészletben csak a sourcet fejtettem ki kicsit bővebben. A label az egy Identifier vagy PrivateIdentifier nevét fogja jelölni. A file egy abszolút útvonalat, ami az Identifiert tartalmazó filet jelöli. A start az egy object, amiben külön van sor és oszlop meghatározva. Ez az Identifier első karakter pozíciójának a sor és oszlop értékei. Az end ugyanaz, mint a start, csak itt az utolsó karakter pozíciójának a sor és oszlop értékei. A range is egy object, a start ennél a kezdő karakter hanyadik karakter volt a kódban, és az end pedig az identifier utolsó karakterének a karakterszáma. A node is egy object ami maga az Identifier vagy PrivateIdentifier.

4.2. Lassúság okai

A binder nagyobb projektekre lassan fut le. Ennek több oka is van:

- Minden hívásnál meghívja a javascriptAddon.nodeot, ami már magában nagy file. TypeScript támogatás után kétszeres lett a mérete, ezáltal sokkal lassabb lett.
- Mivel nagyobb a projekt, ezért sokkal több a függvényhívás is az elemzett kódban, emellett sok a változószám is.
- Az AST nagyon nagy, és ezt bejárjuk többször is bindolás alatt. Emiatt nagyon lassú a program nagy projektek esetén.
- A JSONokat tartalmazó tömb is nagyon nagy lesz, de annyira ez nem lassítja le a programot.

4.3. Optimalizálás

Először ki kellett derítenem, pontosan melyik funkció mennyi memóriát vesz igénybe és milyen gyorsan fut le. Erre találtam egy javasript profilert, ami kiírta minden függvén hívásnál a lefoglalt memóriát és a memória használatot. Ez nekem nem volt a legjobb, mivel nagyon egybe van ágyazva minden, és eléggé mélyre mentek a functionok. Ezután próbáltam picit egyszerűbbet, a binder kódját 3 részre osztani, $console.time()$ és $console.timeEnd()$ használatával. Ez a $console.time()$ parancs kiírja nekem ms-ben, hogy mennyi idő telt el amíg eljutott a $console.timeEnd()$ig. Így megkaptam, hogy melyik rész nagyjából mennyi idő alatt fut le, könnyebb volt szűkíteni, hogy melyik function lesz a bajos. Végül kiderült, hogy a következő sor lassítja be nagyon a programot:

```
1 globals.getWrapperOfNode(resolveNode(astSet, sourceFile, element.source
    .range.start, element.source.range.end, true));
```

Kódrészlet 4.2. Problémás function

Ezen belül is a resolveNode function volt a probléma. A resolveNode kapott egy ast-t (amit a binder kapott meg paraméterbe), egy filenevet, kezdő- és végPozíciót, illetve egy true vagy false értéket, ami attól függ, hogy sourcenodeot vagy targetnodeot keresünk. Az ast-n forEach-el végigmentünk, amin belül az astNodeon walkoltunk addig amíg nem találtuk meg a számunkra megfelelő nodeot. Rosszabb esetben a legvégén volt a node, mivel már a végén voltunk a bindolásnak, és ebből is látható volt, hogy ez kellően sok időbe is kerülhet, ha rengeteg nodeunk van. Kisebb projekt esetében ez nem baj, mivel az ast abban az esetben nem annyira nagy. Ahol már van 100 vagy 200 ezer változó és vagy függvényhívás, ott már nagyobb a baj, mivel ezt rengetegszer meg kell ismételni. Ezután ötleteltem, hogy hogyan tudnám jobbra átírni a resolveNodeot functiont, vagy egy másik logika alapján megközelíteni a problémát. Végül eszembe jutott egy sokkal könnyebb megoldás erre, ami nem igényli a sokszori bejárását az astnek.

Először is, létrehoztam egy indexAST nevű functiont, ami a következőket hajtja végre:

```
1 let indexAST = function (ast) {
2       ast.forEach(astNode =>{
3             globals.setActualFile(astNode.filename)
4             walk(astNode, {
5                   enter: function(node){
```

```
6                   globals.setIndexed(astNode.filename, node.range[0],
                        node.range[1], node)}})})
7       return globals.indexedAST}
```

Kódrészlet 4.3. indexAST function

A 4.3 kódrészletben látható, hogy egy ast-t várunk paraméterben. Ezt a bindertől fogja kapni, ez az egyénilég létrehozott ast. A functionben forEach-el végig megyünk az ast elemein, amik az astNodeok. Itt beállítjuk a $globals.setActualFile()$ függvénnyel az aktuális filenevet. Ezután az adott astNode-on elkezdünk walkolni. Csak az enter metódusát írtam meg. Meghívunk egy függvényt ami a globalsban lett definiálva, setIndexed a neve. Ennek a függvénynek megadjuk a filevenet, a node rangenek a kezdő- és a végparaméterét (tehát ahol kezdődik az adott node és hol végződik, karakterpontosan), és magát a nodeot.

```
1  const setIndexed = function(filename, range_start, range_end, node){
2       let actualfilename = getFilePathAlt(filename)
3       if (indexedAST[actualfilename + "-" + range_start + "-" +
           range_end] !== undefined && indexedAST[actualfilename + "-" +
            range_start + "-" + range_end] !== node){
4             return
5       }
6       indexedAST[actualfilename + "-" + range_start + "-" + range_end]
           = node
7  }
```

Kódrészlet 4.4. setIndexed function

A setIndexed először végez egy vizsgálatot arra, hogy az adott node be van már indexelve. Ezt úgy teszi meg, hogy az indexedAST tömbben keres egy indexre (Ez az index a következőképp néz ki: fileNev-StartPosition-Endposition), és még vizsgál is arra, hogy ha van ilyen index, akkor ezen van emár olyan node. Ha van akkor nem állít be semmit, csak returnol, mivel már be van indexelve az adott node. Ha nincs, akkor beállítja az indexedAST tömbnek az adott indexre az adott nodeot.

Ezáltal az adott astn csupán csak egyszer megyünk végig foreach-el és egyszer walkolunk, ekkor egy tömbbe beindelexünk minden egyes létező nodeot ami kellhet nekünk. Ha ezzel megvoltunk akkor utána kezdődhet a binder része. A 4.2 látható, hogy először megkerestük a nodeot index alapján és az-

tán hívtuk meg rá a $getWrapperOfNode()$ függvényt. Ezt is megváltoztattam, írtam rá egy getIndexed függvényt. A $getIndexed()$ függvény megvizsgálja, hogy source vagy targetNodeot keresünk. Ha targetNodeot, akkor returnoljuk a $getWrapperOfNode(indexedAST[filename - StartPosition - Endposition])$-t. Ha targetNodeot keresünk, akkor walkolunk ismét, de nem az ast-ben, hanem már a beindexelt tömbben. Ez lényegesen gyorsabb, mint a resolveNodeos megoldás, mert ott minden egyes esetben az ast-n walkoltunk, itt meg csak egy beindexelt elemén a tömbnek. Ha megkaptuk a sourceNodeot akkor returnoljuk a $getWrapperOfNode(result)$t.

4.4. Eredmény

TODO: Ez lehet nem kell ide, majd összefoglalóban írni erről.

5. fejezet

Regteszt frissítés

jobb lett, mükszi hogy van tesztelve, output és performance vizsgálat TS tesztek hozzáadása / eredmények átnézése Specifikusabb tesztek hozzáadása (kapcsolókra tesztelünk –> JSAN / ESlintrunner 1 mappa / 1 kapcsoló) Python beli nehézségek tesztátírás során

5.1. Röviden a regtesztről

5.2. Tesztekről

5.3. Tesztek kibővítése

6. fejezet

Összefoglaló

- javult a program - VPP up do datere hozása - JSAN további optimalizálása

6.1. Program javulása

6.2. Tervek

6.2.1. JavaScriptSchema naprakészre hozása

6.2.2. JSAN további optimalizálása

Nyilatkozat

Alulírott Pozsgai Alex programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2023. május 3. .....
aláírás

Köszönetnyílvánítás

Irodalomjegyzék

[1] Typescript-eslint github repository. https://github.com/typescript-eslint/typescript-eslint/tree/main/packages/ast-spec/src.