Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Szűcs Dániel

2023

Szegedi Tudományegyetem
Informatikai Intézet

Es2panda fordítóprogram fejlesztések

Szakdolgozat

Készítette:
Szűcs Dániel
Mérnökinformatika BSc
szakos hallgató

Belső Témavezető:
Dr. Kiss Ákos
egyetemi adjunktus

Külső Témavezető:
Gál Péter
tudományos segédmunkatárs

Szeged
2023

Feladatkiírás

Az Es2panda fordítóprogram fejlesztése

A jelentkező feladata az Es2panda fordítóprogram funkcionalitásának kibővítése, az ETS nyelvi konformanica növelése érdekében. Az elkészült programnak az alábbi funkciókkal kell rendelkeznie:

- Meg kell oldani a default és named argumentumok kezelését.
- Meg kell oldani az ETS nyelv által előírt változó és típusnév feloldás
- Meg kell oldani a kiemittált kód regiszter spill-fill utasításait és kidolgozni egy regiszter allokálási stratégiát.

Tartalmi összefoglaló

- ***A téma megnevezése:***

Es2panda fordítóprogram fejlesztése, az ETS nyelvi konformancia növelés érdekében.

- ***A megadott feladat megfogalmazása:***

A cél, hogy az Es2panda fordítóprogram funkcionalitását kibővítsem. A Checker komponensnek képesnek kell lennie különböző, az objektum orientált programozási paradigma által leírt szemantikai elemzésekre, továbbá az ETS nyelv specifikus szkóp feloldási szabályainak kezelésére. A Compiler komponensnek képesnek kell lennie a feltételes vezérlési szerkezet és a különböző bináris operátorok bytecode-ra fordítására, valamint a változók regiszter allokálására és spillelésére.

- ***A megoldási mód:***

Először is, a Compiler komponens egyszerűbb részein, a vezérlési szerkezeteken és az operátorokon, majd a Checker komponens OOP mintáinak szemantikai elemzésén dolgoztam. Ezek után a két komponenst párhuzamosan fejlesztettem, az új nyelvi elemek implementációs sajátosságainak megfelelően.

- ***Alkalmazott eszközök, módszerek:***

Az Es2panda fejlesztése Ubuntu 22.04 operációs rendszeren folyik, a C++ nyelv 17-es szabványában. A fejlesztéshez Visual Studio Code fejlesztői környezetet használok. A fordításhoz Clang 14-es fordítót használom, debuggoláshoz a gdb eszközt választottam.

- ***Elért eredmények:***

Implementáltam egy robosztus rendszert a regiszter spillelésre, valamint a regiszterek statikus típusainak követésére, továbbá implementáltam a szkóp feloldást, amely felhasználóbarát fordítási hibaüzeneteket ad a fordítóprogram felhasználójának.

- ***Kulcsszavak:***

fordítóprogram, szemantikai analízis, regiszter spillelés

Tartalomjegyzék

1. Bevezetés

Szakdolgozatom témájául az Es2panda [3] fordítóprogramon végzett fejlesztéseimet szeretném bemutatni. Mivel a fordítóprogramon végzett munkáim rendívül szerteágazóak, így sok különböző problémával találkoztam a fejlesztés során, melyekre a megoldás nem mindig volt triviális a már meglévő szoftver architektúra mellett. A fejlesztés alatt a következők voltak a céljaim:

- Robosztus könnyen érthető kód írása, amely követi a C++ nyelv bevált gyakorlatait. Ezek magukba foglalják az explicit memória kezelése minimalizálását, a const korrektséget, a makró használat minimalizálását, valamint a komponensek szoros összekapcsolódásának elkerülését.
- A futásidei teljesítmény növelése, akár a fordítási idő romlása árán. Ez leginkább a template metaprogramozási gyakorlatokkal értem el, melyek drasztikusan javítottak a kód hívó oldalának egyszerűsítésében.
- Az implementált függvények és típusok interfészeinek egyszerűsítése, hogy minél nehezebb legyen őket rosszul használni.
- A modern fordítóprogramok által alkalmazott technikák kutatása és az Es2pandába való integrálásuk. Ezek magukba foglalják a különböző optimalizációkat és statikus analíziseket.

A következő oldalakban, betekintést adok az Es2panda általam fejlesztett részeibe, azok tervezési folyamataiba, és példákat mutatok a teljesítmény orientált gondolkodásmódra, amely elengedhetetlen egy fordítóprogram esetében. Először is bemutatom az Es2pandát jelenlegi állapotában, majd egy áttekintést nyújtok az általános struktúrája felett. Bemutatom az Es2panda öt nagy komponensét és rajtuk végzett fejlesztéseimet, mindezt a komponensek függőségének sorrendjében, kezdve a belépéstől a kész bytecode-ig. A komponenseket és az Es2panda kódját az általa támogatott ETS nyelvet érintő részekből fogom bemutatni. Az ETS egy menedzselt nyelv, melynek szintaxisa a Typescripthez, működése pedig a Java-hoz hasonlít. Végezetül kitérek a generált bytecode-ra és az érdekesebb implementációs részletekre.

2. Mi az az Es2panda?

Ebben a fejezetben általánosan beszélek az Es2pandáról mint szoftverről.

Az Es2panda [3] egy fejlesztés alatt álló fordítóprogram frontend melyet C++ nyelven fejlesztünk. A program több különböző szkriptnyelvet támogat és párhuzamos fordítással rendelkezik.

2.1. Hogyan fejlesztjük az Es2pandát?

Az Es2panda fejlesztése Linux platformon történik, a projekt fordításához szükséges csomagokat egy bootstrap szkript tudja telepíteni Ubuntu 20.04-es és 22.04-es kiadásokra. A build rendszer a CMake 3.10-es verziója. A build konfigurálható gcc és clang fordítóprogrammal is, a támogatott verzió gcc esetében 11.1, clang esetében pedig 14.0. Az Es2panda 64 bites Unix alapó rendszerekre lefordítható, mind x86-64 mind arm64 architektúrára. A projekt a C++17-es nyelv szabványt használja.

A verzió kezelést biztosító szoftver a git. A forráskód megtalálható a Gitee weboldalon. A különböző toolchain komponensek, mint az Ark runtime [2], az Es2panda, és a runtime nyelvi pluginjai (Ecmascript, ETS) különböző repositorykban érhetők el.
Az Es2panda mind logikailag, mind fájlrendszer szinten szegmentálva van a struktúráját illetően. Ez a szegmentáció segít átlátni az absztrakciókat, kezelni a dependenciákat, elrejteni az implementációs részleteket és optimalizálni a fordítási időt.

2.2. A projekt felépítés

Az Es2panda a *panda::es2panda* névtéren belül helyezkedi el, a kód példákban ezt a névteret nem fogom külön megnevezni. A program a standard C/C++ header módszerrel van strukturálva. Az absztrakt szintaxis fa (AST) minden node-ja a saját header fájljában van, viszont az öt nagy logikai komponens, amelyeket később részletesen kifejtek, javarészt monolitikusan egy-egy header fájlban található. Néhány függvény, amelyek különben implementálhatók lennének egy külön fordítási egységben (*.cpp*), mégis

headerben van implementálva. Ennek az oka a fejlesztés elején a fordítási idő gyorsítása volt, valamint a hívások jobb optimalizálása fordítási egység határokon (*inlining)*.

A projekt tartalmaz több olyan fájlt is, amelyeket a build rendszer fordítás előtt generál. Ezek a fájlok általában olyan kódot tartalmaznak, amelyek repetitívek, vagy könnyen generálhatók egyszerű markup nyelvekből. A projekt a YAML markup nyelvet használja, amit Ruby szkriptek dolgoznak fel, majd az Embedded Ruby (ERB) rendszer alakítja át megfelelő forrás fájllá, általában headerré. Néhány generált fájlra a logikai egységek részletezésénél külön ki fogok térni.

Az Es2panda két ELF állományból áll. Az egyik az *libes2panda-lib.so* programönyvtár, ami a tényleges fordítási logikát tartalmazza, a másik maga az *es2panda* nevű futtatható állomány, ami dinamikusan linekelt az előző programönyvtárhoz. Ez a szeparáció lehetővé teszi, hogy a fordítóprogram a futtatható állomány nélkül is integrálható lehessen más rendszerekbe.

2.3. A logikai struktúra

Ebben a fejezetben röviden bemutatom a logikai egységeket, a mélyebb működésükre és implementációs részleteikre a saját fejezeteikben fogok kitérni, kivéve az első egység esetében. Mivel ennek az egységnek a fejlesztésében nem foglaltam kulcsfontosságú szerepet, így nem szentelek neki külön fejezetet. Az Es2panda fordítóprogram öt nagy logikai elemre van strukturálva. Ezek név szerint a Lexikális Elemző, az Elemző, a Binder, a Szemantikai Elemző és a Kód Generátor. Van egy hatodik elem is, amin az előbb említett logikai elemek dolgoznak, ez az absztrakt szintaxis fa (AST).

2.3.1. A Lexikális Elemző

A Lexikális Elemző a továbbiakban Lexer, felelős a beolvasott forráskód tokenizációjáért. Az Es2panda indulásakor a Lexer az első komponens amelyik inicializálódik. A Lexer szolgáltatja az Elemző számára a tokeneket és a sor, oszlop információt hibaüzenetek esetén. Ennek a komponensnek egy része, a kulcsszavak beolvasása egy YAML markup fájlból generálódik a build során.

A Lexerből a *Lexer::NextToken* függvénnyel lehet az aktuális tokent kikérni, ez előre is mozgatja a lexer fejét a következő tokenre. Lehetőség van az aktuális pozíción lévő tokent követő karakter megvizsgálására, ezt a *Lexer::Lookahead* függvény valósítja meg. Esetenként előfordul, hogy a parsolás során a Lexert vissza kell tekerni. Erre lehetőséget adnak a *Lexer::Save* és a *Lexer::Rewind* függvények.

A generált *keywords.h* nevű fordítási egység valósítja meg a tényleges tokenizációt. A generált fájl függvényei a *Keywords::Scan_** alakúak, ahol a '*' a következő betűt jelöli. Például, ha az első függvény a *Keywords::Scan_i* volt, akkor a következő hívott függvény négy különböző lehet. Lehet a *Keywords::Scan_if* ami ha f-et whitespace követi felvesz egy 'if' *Lexer::Token-t* ami kulcsszó. Hasonlóan lehet a *Keywords::Scan_im és Keywords::Scan_in* mely függvények az 'implements' és 'interface' kulcsszavakat próbálják szkennelni. Ha ezek közül egyik sem teljesül, akkor a negyedik esetben az előállított *Lexer::Token* azonosító (identifier) lesz.

A **Kód Részlet 1** mutatja hogyan lehet egy kulcsszót felvenni a leíró YAML fájlba.

```
keywords:
- name: 'async'
  token: KEYW_ASYNC
  keyword: [as, ets]
  keyword_like: [js, ts]
```

Kód Részlet 1: 'async' kulcsszó YAML rekord.

A *name* mező a kulcsszó konkrét formáját adja meg, amit a Lexer keresni fog a tokenizáció során. A *token* mező a *Lexer::Token-hez* tartozó *Lexer::TokenType* generált enum egyik mezőjének neve lesz. A *keyword* és *keyword_like* mezők megadják, hogy melyik nyelvben hogyan viselkedik ez a kulcsszó. Mivel az Es2panda fordítóprogram több nyelvet is támogat ezért a Lexer nagy része újra használható a nyelvek közös pontjai között. Tehát ahogy a **Kód Részlet 1** mutatja, az *async* kulcsszó az AssemblyScript és ETS nyelvek esetén hard kulcsszó, azaz minden kontextusban kulcsszó, míg Ecmascript és Typescript nyelvek esetén soft kulcsszó, azaz a megfelelő pozícióban kulcsszó más pozícióban pedig azonosító.

2.3.2. Az Elemző

Az Elemző logikai egység a Lexer által előállított tokenekből építi fel az absztrakt szintaxis fát. Mivel az implementációja egy LR(1) -es elemző így ezt alulról felfelé teszi, először a levél node-okat állítja elő majd azok szülő node-jait, így haladva egészen a gyökér node-ig. Az Elemző hívja a Binder megfelelő függvényeit, hogy a létrehozott AST node-ok megfelelő szkópokba legyenek bekötve.

2.3.3. A Binder

A Binder komponens az Elemző komponenssel együtt dolgozva építi fel a szkóp fát. A szkóp gyakorlatilag egy kód blokk, amiben a változók és típusnevek keresik egymást. Ebből a primitív elemből épül fel a szkóp fa adatszerkezet. A szkópok *binder::Variable* pointereket kötnek nevekhez egy *std::unordered_map-ben*. A *binder::Variable* igazából egy szimbólum mivel nem csak változóra mutathat, hanem típus névre is és a megfelelő AST node deklarációjára mutat. Minden egyes szkópban a feloldás úgy történik egyszerűbb esetben, hogy a szkóp fát alulról felfelé bejárjuk, mindig csak a szülőkön iterálva, és megkeressük név alapján a megfelelő *binder::Variable-t*.

2.3.4. A Szemantikai Elemző

A Szemantikai Elemző az Es2panda legkomplexebb logikai egysége. Feladata a kód szemantikai analízise. Ez Ecmascript esetében gyakorlatilag egy üres operáció, minden más támogatott nyelv esetén viszont több feladatot is el kell látnia. Az Es2panda által támogatott nyelvek az Ecmascript kivételével mind rendelkeznek valamilyen típus rendszerrel. A típusok által hordozott információból és a nyelvek által támogatott különböző programozási paradigmák szabályrendszeréből a fordítóprogramnak warning és error üzeneteket kell előállítania, ha valamilyen szemantikai szabály sértett. A Szemantikai Elemző gyakorlatilag majdnem minden AST node-hoz rendel egy *checker::Type* példányt, amelyben leírja az adott típust és különböző típus relációkat tud

felállítani. Például osztályok között inheritencia láncot tud felállítani, vagy primitív típusok közötti konverziókat tud a primitívek mérete alapján engedélyezni vagy letiltani.

2.3.5. A Kód Generátor

A Kód Generátor az absztrakt szintaxis fából, és a Szemantikai Elemző által előállított típus információból generál bytecode-ot. Ez a bytecode statikusan típusos, kivéve Ecmascript esetén, ahol az Ark runtime egy plugin segítségével támogatja a dinamikus bytecode-ot. A Kód Generátor párhuzamosan fut a fordítási egységekben lévő függvényekre. Ezt a működést mutexek nélkül tudja végezni (kivéve a szál scheduling esetében). A bytecode gyakorlatilag egy regiszter gép. Egy regiszter az akkumulátor szerepét tölti be. A Kód Generátor felelős a regiszterek megfelelő kiválasztásáért a regiszter spillelésért. A Kód Generátor egy vectorban tárolja el a kigenerált utasításokat.

2.3.6. Az Emitter

Az Emitter komponens fordítja át az Es2panda belső reprezentációját az Ark runtime reprezentációjára. Az Emitter ezenfelül annotációkkal látja el a bytecode-ot. Ezeknek az információknak egy részét az Ark runtime más részét a debugger használja fel.

2.3.7. Az Absztrakt szintaxis fa

Az Absztrakt Szintaxis Fa továbbiakban AST, az Es2panda legfontosabb adat szerkezete. Megvalósítását tekintve egy osztály hierarchiából áll, melynek gyökere az *ir::AstNode* pure virtual base osztály. Ez az osztály három fontos adattagot deklarál. Az első az *ir::AstNodeType* enum, aminek segítségével egy makrón keresztül *Is##NodeType* és *As##NodeType* típus ellenőrző, és típus konverziós függvények kerülnek kigenerálásra. A második szintén egy enum típusú adattag, az *ir::ModifierFlags*. Ahogy a neve is mutatja ezek a flagek bizonyos modifier kulcsszavak meglétét tárolják egy AST node-on. Ilyenek például a Java-ból ismert és az ETS által is használt láthatóságot szabályozó kulcsszavak a *public, protected, private* vagy más egyéb módosítók például *abstract,*

async, const, constructor, static. A harmadik legfontosabb adattag a szülő node-ra mutató AST node pointer.

Az *ir::AstNode* négy fontos pure virtual függvényt deklarál. Ezek az *Iterate, Dump, Check* és *Compile* függvények. A *Check* és *Compile* függvények külön overloadokkal rendelkeznek Javascript és ETS esetekre.

- Az *ir::AstNode::Iterate* virtuális függvény paraméterében egy callback függvényt vár, amelynek szignatúrája *void(ir::AstNode *)*. Ez a virtuális függvény egy node minden gyerekére meghívja a paraméterben kapott callback függvényt.
- Az *ir::AstNode::Dump* kírja az adott node JSON reprezentációját. A függvényt az előzővel együtt használja a program a teljes AST kiírására. Ez az osztály képes több különböző primitív, vagy tömb szerű adatszerkezetet JSON formátumba alakítani.
- Az *ir::AstNode::Check* függvény visszaadja egy adott node típusát. A típust a Checker komponens állítja elő. Általában a függvény mellékhatása, hogy a node típusát is beállítja, ha a node egy *ir::TypedAstNode*.
- Az *ir::AstNode::Compile* függvényt a Kód Generátor hívja. Ebben a függvényben az AST node elvégzi a lefordításához szükséges lépéseket, majd az eredményt az akkumulátorba teszi.

Az *ir::AstNode-nak* három fontosabb leszármazott osztálya van. Az első az *ir::Statement* ami az utasítások ősosztálya. A második az *ir::TypedAstNode* amely kibővíti a funkcionalitást típus információval. A másodikból örököl a harmadik *ir::Expression*, ami a kifejezések ősosztálya.

3. Az Elemző

Az Elemző komponens állítja elő az AST-t. Az Es2panda által használt elemző egy LR(1) elemző. Ez azt jelenti, hogy az AST-t alulról felfelé építi fel. Az Elemző komponens *parser::AllocNode* függvénye az *ir::AstNode::Iterate* függvény mechanizmusát használja az új node-ok létrehozására. Miután lefut az allokálni kívánt node konstruktora, a *parser::AllocNode* meghívja az *ir::AstNode::Iterate* függvényt egy olyan callbackkel, amely az allokált node összes gyerekének beállítja a szülőjét az új node-ra. Ez gyakorlatilag az Elemző általános mechanizmusa.

3.1. Az Elemző belépési pontja

Az Elemző belépési pontja a *parser::Parser::ParseProgram* függvény. Ez a függvény ETS esetében először is felépíti az ETSGLOBAL nevű osztályt. Erre azért van szükség mert a bytecode nem támogatja a globális változókat és függvényeket, viszont az ETS nyelv igen, erre nyújt megoldást az ETSGLOBAL. Ez az osztály gyakorlatilag egy shadow, amelyre minden globális változó és függvény fel lesz véve, mint statikus property.

Következő lépésben a default import-ok kerülnek parsolásra. Erre azért van szükség, mert az ETS nyelv előírja, hogy minden fordítási egység impliciten importálja az *std.core* csomagot. Ebben a csomagban érhető el a Java-hoz hasonlóan az *std.core.Object* osztály, ami az osztály hierarchia gyökere. Ezek mellett még sok más funkcionalitást is biztosít.

Az utolsó lépésben a *parser::ETSParser::ParseETSGlobalScript* függvény hívódik. Ez a függvény először összegyűjti az egymodulba tartozó fájlokat, majd elindítja a parsolás lényegi részét a *parser::Parser::ParseTopLevelDeclaration* függvényt, amely addig olvassa a Lexer által előállított tokeneket ameddig el nem jut a *lexer::TokenType::EOS-ig*. Ezután allokálódik a AST gyökér node-ja az *ir::ETSScript*.

3.2. Az Elemző komponensen végzett fejlesztéseim

Az Elemző komponensen végzett fejlesztéseim főként a default és named argumentumok parsolása. Ezek az ETS nyelvi elemek a Python-hoz hasonlóan lehetővé teszik, hogy egy függvényt opcionális argumentumokkal ruházzunk fel és az is lehetséges, hogy az argumentumokat a függvény deklaráció argumentum listájától eltérő sorrendben adjuk át, expliciten megnevezve azokat.

3.2.1. A default argumentumok

A default argumentumokkal rendelkező függvények esetében nem csak egy, hanem kettő függvényt is elő kell állítani. Az eredeti függvény mellett a második egy wrapper függvény, aminek feladata, hogy runtime megvizsgálja, hogy ténylegesen mely argumentumok lettek átadva, és a hiányzó argumentumok default incializáló expressionjét meghívja.

Az első feladat létrehozni egy új AST node-ot. Ez az AST node az ir::*ETSParameterExpression* tartalmazza az információt minden függvény, minden argumentumáról. Gyakorlatilag minden *ir::ETSParameterExpression* két részből áll, amit az ETS nyelv nyelvtana diktál. Az egyik egy *ir::Identifier*, a másik pedig egy *ir::Expression*. Ezek közül a második az argumentum default initializere, ami opcionális. A **Kód Részlet 2** mutatja az AST node struktúráját.

```
class ETSParameterExpression : public Expression {
public:
    explicit ETSParameterExpression(Identifier *left,
Expression *right);

private:
    Identifier *name_;
    Expression *initializer_;
};
```

Kód Részlet 2: Az implementált *ir::ETSParameterExpression* struktúrája.

A getter függvényeket a kód részlet tömörségének érdekében kihagytam. Setter függvényekkel a node nem rendelkezik, mivel sem egy argumentum neve, sem default initializere nem változhat.

A következő feladat az új node parsolása. A default argumentum parsolása a meglévő elemző infrastruktúrával egyszerű. Először is egy nevet, azaz *ir::Identifier-t* kell parsolni majd egy típus annotációt. Ha ezen a ponton hiányzik bármelyik is akkor szintaxis hibát dobok. Ezek után opcionálisan parsolni kell egy *ir::Expression-t*, ha az argumentumnak van default initializere. Ezt a lépés sorozatot addig kell ismételni ameddig van még argumentum a függvény deklaráció szignatúrájában.

3.2.2. A proxy függvény

Miután a teljes függvényt beparsoltam, elő kell állítani egy proxy függvényt, amely becsomagolja azokat a függvény hívásokat, amelyek használják a default argumentumok értékét. Egy default argumentum initializerét természetesen csak akkor kell kiértékelni, ha azt az argumentumot nem adják át a függvénynek. Ahhoz, hogy ez eldönthető legyen a függvény argumentumai közé be kellen szúrni még egy flaget amit a compiler állít be, hogy ténylegesen lett-e passzolva argumentum, ezt a flaget megvizsgálni és annak alapján kiértékelni a default initializert. Ez a logika rontana a függvény futás idején abban az esetben, amikor nincs használva default initializer. Ezért van szükség a proxy függvényre, amely pontosan ezt a logikát implementálja.

3.2.2.1. A proxy függvény struktúrája

A proxy függvény argumentum listája megegyezik a sima függvényével kivéve, hogy tartalmazza a fentebb említett flaget. A flag egy 32 bites változó, tehát 32 default argumentumot tud kódolni. Ha több default argumentumra van szükség, akkor egy újabb 32 bites flag változó kerül az argumentum listába. Miután a proxy függvény letesztelte a default argumentumokat és beállította az értékeket, áthív a tényleges overhead nélküli függvénybe. A proxy függvény minden szintetikusan előállított *ir::Identifier-e* mangled, tehát olyan név, amelyet a felhasználó sohasem tud leírni. Ez gyakorlatilag azt jelenti,

hogy minden ilyen névbe egy ’#’ karakter van belefűzve. A compiler csak olyan hívásokat cserél a proxy hívására, ahol szükséges legalább egy default argumentum használata.

3.2.2.2. A proxy függvény generálása

Először is elő kell állítani a proxy függvény forrás kódját. Ehhez a már meglévő függvényből kell generálni egy stringet ami megfelel az ETS nyelv szintaktikájának. Ez a string generálás a függvény nevével kezdődik, ami úgy épül fel, hogy az eredeti függvény neve után egy ’#’ karakterrel összefűzve felkerül a ’default’ szó. Ezután végig iterálok az eredeti függvény paraméterein és minden paraméter nevét hozzáfűzöm a proxy függvény paraméter listájához. Majd a flag paraméterek kerülnek be a paraméter listába. Ezek után a visszatérési típus kerül hozzáfűzésre, ami *Object* lesz. Ezt később kicserélem az eredeti függvény tényleges visszatérési típusára. Ezek után ki kell generálni a stringbe a bitmaszkoló feltételeket, majd a tényleges függvény hívást.

```
std::string proxy_method {};
const std::string proxy_method_name = function->Id()-
>Name().Mutf8() + "#default";
util::Helpers::AppendAll(proxy_method, proxy_method_name, '(');

for (const auto *const it : function->Params()) {
      const auto *const param_ident = it->Name();
    util::Helpers::AppendAll(proxy_method, param_ident-
    >Name().Utf8(), ':', param_ident->TypeAnnotation(), ',');
}
const std::size_t num_flags = (num_default_params / 33) + 1;
for (std::size_t i = 0; i < num_flags; ++i) {
    util::Helpers::AppendAll(proxy_method, "flag#",
    std::to_string(i), ":int");
    if (i != num_flags - 1) {
        proxy_method += ',';
    }
}
proxy_method += "):void{";
```

Kód Részlet 3: Részlet a string összeállítás implementációjából.

A proxy függvény parsolása természetesen nem tud úgy történni, mint egy sima függvény parsolása, mivel nincs jelen a forráskódban. Maga a proxy függvény létrehozásához nincs szükség semmilyen szkóp, vagy típus információra, ezért az egész szintetikus függvény létrehozható elemzési időben. Mivel kézzel felépíteni egy teljes függvény AST-jét komplex fenntarthatatlan kódhoz vezetne, ezért helyette a *parser::InnerSourceParser-t* használom. Ez az osztály gyakorlatilag kicseréli a Lexer komponenst az aktuális Elemző példány alatt, és egy előre összeállított stringre cseréli a forráskódot, ez az előbb összeállított string lesz. A parsolás tehát úgy történik, hogy a proxy függvény ETS kódját előállítom egy *std::string-be* majd az Elemző mintha csak egy sima függvényt látna, beparsolja és a Binder megfelelő függvényeivel létrehozza a szkópokat és beköti a változókat. A **Kód Részlet 4** mutatja a *parser::InnerSourceParser* létrehozását.

```
const InnerSourceParser isp(this);
GetContext().Status() |= ParserStatus::ALLOW_HASH_MARK;
const auto lexer = InitLexer({"<default_methods>.ets",
proxy_method});

auto *const ident = AllocNodeNoSetParent
<ir::Identifier>(util::StringView(proxy_method_name), Allocator());
ParseClassMethodDefinition(ident, method->Modifiers());
GetContext().Status() &= ~ParserStatus::ALLOW_HASH_MARK;
```

Kód Részlet 4: A *parser::InnerSourceParser* létrehozása a proxy függvény implementációjában.

A *parser::InnerSourceParser* paraméterében elkapja az aktuális *parser::ETSParser this* pointerét, ezzel a konstruktorában elmenti a Lexer állapotát. Következő lépésben az Elemző státuszára felkerül a *ParserStatus::ALLOW_HASH_MARK* flag. Erre azért van szükség mert az Elemző alapvetően nem engedi a ‘#’ karaktert használni string literálokon kívül. A *parser::InitLexer* függvény létrehoz egy új lexer-t. A *<default_methods>.ets* egy placeholder név, ha a parsolás során hiba keletkezik, akkor az Elemző erre a dummy fájlra fog hivatkozni. A *proxy_method* a kézzel összeállított ETS kód *std::string* formában. Ezután a *parser::Parser::AllocNodeNoSetParent* függvény allokál egy *ir::Identifier-t* a függvénynek majd ezzel az azonosítóval elindul a parsolás a *parser::Parser::ParseClassMethodDefinition* függvényben. Ez a függvény

gyakorlatilag nem is látja, hogy éppen egy szintetikus függvényt parsol. A függvény végén leveszem a flaget az Elemző státuszáról, majd a *parser::InnerSourceParser* destruktora visszaállítja az elmentett Lexert.

A *parser::InnerSourceParser* használatának legfőbb előnye, hogy nem kellett a kódban ugyan azt a logikát kétszer leírni. Ha kézzel lenne felépítve az *ir::MethodDefinition* AST node, akkor ugyan azt a kódot kellene megismételni, mint egy hagyományos függvény parsolása esetén, kivéve a Lexer mozgatásokat. Mivel egy függvény potenciálisan több száz AST node-ból is állhat, ez a lehetőség komplex, hosszú, fenntarthatatlan és hibára hajlamos kódhoz vezetne.

4. A Binder

A Binder komponens építi fel a szkóp fát, és kezeli a szkópok közötti váltásokat. A Binder nyilván tartja az éppen aktuális szkópot és a globális szkópot egy-egy *binder::Scope* pointer adattag segítségével. A Binder komponens építi fel az AST node-ok teljesen kvalifikált neveit is mivel ehhez a minden névvel rendelkező szkópra szükség van.

4.1. A szkóp fa

A szkóp fa ősosztálya a *binder::Scope*. Hasonlóan az AST ősosztályához a *binder::Scope* is tartalmaz egy szülőre mutató pointert. Ezen kívül a szkóp tratalmaz egy *ir::AstNode* típusú adattagot is, ez általában egy *ir::BlockStatement* ami a szkóp létrejöttét eredményezte. A *binder::Scope* legfontosabb adattagja a *binder::VariableMap* ami igazából csak egy alias a egy *std::unordered_map-re*. Ez a map tartja számon, hogy egy szkópban milyen névhez, milyen változó, vagy típusnév tartozik. Továbbá, a map szemantikájából adódóan megoldja a változó nevek elrejtését is. Alapvetően a mapbe mindig az *std::unordered_map::insert_or_assign* C++17-es függvényt használva szúrunk be, ami kulcs ütközés esetén felülírja az értéket.

A *binder::Scope* feletti egyik legfontosabb absztrakció a *binder::LexicalScope*. Ez az absztrakció teszi lehetségessé Javascript esetén az *eval* ellenőrzését, azonban ETS szempontból nem sok jelentősége van, mivel ETS-ben nincs *eval*. Ennek ellenére a *binder::LexicalScope::Enter* statikus függvényt kell egy szkópba való belépéshez hívni. A belépés azt foglalja magába, hogy a *binder::LexicalScope* kicseréli a Binder szkóp pointerét a *binder::LexicalScope::Enter* által kapott pointerre. Ez azért fontos mert az Elemző amikor egy *ir::Identifier-t* allokál, akkor nem tudja, hogy milyen szkópban áll, ezért mindig a Binder által mutatott aktuális szkóp *binder::VariableMap-jéhez* fűzi hozzá.

4.2. A Binder komponensen végzett fejlesztéseim

A Binder komponensen végzett fejlesztéseim, a szkóp keresés eredmény típusának a *binder::ScopeFindResult-nak* az újra implementálása és az azonosító feloldásnak az újra dolgozása volt.

4.2.1. A szkóp keresés eredmény típus újra implementálása

A szkóp keresés eredmény típusa a *binder::ScopeFindResult* típus egy egyszerű típus, amit a *binder::Scope-on* található kereső függvények adnak vissza. Ez a típus három fontos adattaggal rendelkezik:

- A feloldott név, amit a kereső függvények kaptak.
- Egy *binder::Scope* pointer, ez a szkóp az, amiben a kereső függvény megtalálta a keresett nevet.
- Egy *binder::Variable* pointer, ami a név alapján feloldott szimbólum.

A három változó közül a *binder::Scope* pointer problémásnak bizonyult több kereső függvény esetében is. Mivel a függvényekben előfordulhat, hogy a visszaadott *binder::ScopeFindResult binder::Scope* pointere az maga a *this* pointer, és ezt a pointert a *binder::ScopeFindResult* non-const pointerként tárolta ezért a kereső függvényeket nem lehetett konstanssá tenni. Ez a tény az egyszerű kereső függvények esetében hátrányos, mert a kiadott pointeren keresztül nem változtak a szkópok, viszont ezt a tényt a függvény nem kommunikálta. A típus rendelkezik továbbá még kettő *uint32_t* típusú adattaggal, amik csak Javascript esetben vannak használva.

4.2.1.1. A megvalósítás

A *binder::ScopeFindResult* típust először is átneveztem *binder::ScopeFindResultT-re* és template-sé tettem. Erre azért volt szükség, hogy javítani tudjam a const korrektséget. A template használatával létre lehet hozni olyan eredmény típust is ami const tagfüggvényekben és nem const tagfüggvényekben egyaránt használható. A template-nek egyetlen típus paramétere van a *ScopeT*. Ezt a típus paramétert szerettem volna megszorítani C++20 *requires* klózokkal, viszont a projekt C++17-ben van írva. A

megszorításokat így kénytelen voltam a C++17 által nyújtott *std::enable_if_t* típus template paraméter használatával megoldani. Az *std::enable_if_t* egy olyan template típus, ami kihasználja a SFINAE-t. A SFINAE (Substitution failure is not an error) egy olyan szabály a C++ template-eken. Ez a szabály azt mondja ki, hogy ha egy template példányosítás nem sikerül, akkor a fordító program ne álljon le hibával, hanem próbálkozzon addig ameddig egy helyes példányt elő tud állítani. Ha minden lehetőséget kipróbált csak akkor lesz fordítási hiba. Az *std::enable_if* úgy használja ki ezt a szabályt, hogy ha a feltételében adott fordítási idejű predikátum igaz, akkor a behelyettesítés sikerülni fog, különben nem. Az *std::enable_if_t* predikátuma, hogy a kapott *ScopeT* típus, pointer és *binder::Scope* vagy annak leszármazott típusa.

Egy alternatív implementáció lehetne, ha a megszorítások között a pointer típus nem szerepel, és csak az inheritencia vizsgálat szerepelne. Ez az átdolgozott *binder::Scope::Find* függvények implementációja is tisztább volna.

Végezetül készítettem két *using* deklarációt *binder::ScopeFindResult* és *binder::ConstScopeFindResult* néven, hogy a visszaadott típusok nevei kifejezőbbek legyenek.

4.2.1.2. Az általános kereső függvény megvalósítása az új típusokkal

Az általam implementált általános kereső függvény működése rendkívül egyszerű. Gyakorlatilag a *binder::Scope* szülein iterál felfelé, ameddig meg nem találja a keresett deklarációt a szkópban. Ha nem találja meg akkor a *binder::ScopeFindResultT binder::Variable* pointere *nullptr* lesz. Ennek a függvénynek csinálnom kellett konstans én nem konstans overloadot is, mert esetenként a visszaadott *binder::ScopeFindResultT binder::Scope* pointerén keresztül, mutáció történik. Ennek ellenére nem szerettem volna megismételni az implementációt kétszer úgy, hogy ez a két overload között a különbség csak és kizárólag a visszatérési értékük.

Ennek elkerülésére létrehoztam a *binder::Scope::FindImpl* függvényt. Ez a függvény egy template függvény, amelynek két típus paramétere van. Az első a *ResultT* ezt az előbbiekhez hasonlóan korlátoztam két típusra az *std::enable_if_t* segítségével. Ez a két típus az előbb említett *using* deklaráció által definiált *binder::ScopeFindResultT* template

példányok. A második típus paraméter a *ScopeT* amire ugyan azok a megszorítások vonatkoznak, mint a *binder::ScopeFindResultT* típusparaméterére. A függvény a *binder::Scope* pointert forwarding referenciával veszi át, tehát a típus megtartja minden esetben a *const* és *volatile* dekorátorait, így lehetséges, hogy a *binder::Scope::Find* függvény konstans és nem konstans overloadja is ugyan ezt a függvényt hívják.

A **Kód Részlet 5** mutatja a *binder::Scope::FindImpl* függvény SFINAE template fejlécét.

```
template <typename ResultT, typename ScopeT,
   std::enable_if_t<std::is_same_v<ResultT,
ConstScopeFindResult> || std::is_same_v<ResultT,
ScopeFindResult>, bool> = true,
   std::enable_if_t<std::is_pointer_v<ScopeT>
&& std::is_base_of_v<Scope, std::remove_pointer_t<ScopeT>>,
bool> = true>
static ResultT FindImpl(ScopeT &&scope, const util::StringView
&name, const ResolveBindingOptions options);
```

Kód Részlet 5: Az implementált *binder::Scope::FindImpl* függvény fejléce

Itt az *std::enable_if_t-re* azért van szükség, hogy a függvény első paraméterére ugyan azok a megszorítások vonatkozzanak, mint a *binder::ScopeFindResultT* esetén, valamint, hogy a visszaadott *binder::ScopeFindResultT* az ténylegesen a *using* deklarációk által létrehozott template típus példányok egyike legyen. Ezekkel a megszorításokkal biztonságosan használható a függvény, ugyanis a helytelen template példányok létre sem tudnak jönni, mert fordítási idejű hibához vezetnek.

4.2.2. Az azonosító feloldás módosítása ETS esetben

Az azonosítók feloldása az ETS nyelvben a megszokottól egy kicsit másképpen működik. A szabályok rá a következők:

- Ha az azonosítót a *this* kulcsszó kvalifikálja, akkor a keresett azonosítót az aktuális osztályban és az ősosztályaiban, interfészeiben kell keresni, csak a nem statikus adattagok között.

- Ha az azonosítót a *super* kulcsszó kvalifikálja, akkor a keresett azonosítót az ősosztályokban és az ősinterfészekben kell keresni csak a nem statikus adattagok között.
- Ha az azonosítót az osztály neve kvalifikálja, akkor a keresett azonosítót az aktuális osztályban és az ősosztályaiban, interfészeiben kell keresni, csak a statikus adattagok között.
- Minden más esetben (egy eset maradt, a kvalifikálatlan azonosító), a keresett azonosítót először a lokális szkópban kell keresni kivéve, ha a lokális szkóp osztály szkóp, majd a globális szkópban.

Ezek a szabályok gyakorlatilag azt jelentik, hogy egy azonosítót akkor és csak akkor keresünk egy osztályban, hogyha valamilyen kvalifikálóval rendelkezik. Ez a hagyományos azonosító feloldástól merőben eltér. A hagyományos feloldás esetében elég csak a szkóp fán végig iterálni ameddig meg nem találjuk a keresett azonosítót.

Könnyű belátni ahhoz, hogy megtaláljuk, amit keresünk nem elég önmagában a szkóp fa. Szükség van a Szemantikai Elemző által előállított típus információkra is mivel nem csak szülő szkópokban kell keresni az azonosítót, hanem egy potenciális inheritencia láncot is be kell járni, minden kvalifikált feloldás esetében. A Szemantikai Elemző érintő részeket a komponenst érintő fejezetben fogom részletesen taglalni.

A fenti okokból adódik, hogy ezt a feladatot nem lehet egészében a Binder komponensen belül implementálni, így ebbe a komponensbe csak kettő egyszerűbb új függvényt valósítottam meg. A továbbiakban ezek implementációit mutatom be.

4.2.2.1. A megvalósítás

A megvalósításban két új függvényt implementáltam. Az első a *binder::Scope::FindInGlobal* a másik pedig a *binder::Scope::FindInFunctionScope* függvény. Mind a kettő függvény konstans tagfüggvénye a *binder::Scope-nak*.

4.2.2.2. A globális szkópban kereső függvény

A globális kereső függvény, a globális szkópban keres. Ez azonban nem triviális keresés, mert ETS esetében kettő globális szkóp található. Az első globális szkóp a tényleges globális szkóp, ebbe a szkópba kerülnek a más modulokból importált nevek. Ilyen név például a standard függvénykönyvtárból automatikusan importált *Object*. A második az ETSGLOBAL nevű shadow osztály, amely az összes globális függvényt, típust és változót tartalmazza, az atkuális a fordítási egységből.

A függvény úgy működik, hogy egy while ciklusban feliterál addig a szkópig amelyiknek a szülő szkópja a tényleges globális szkóp, így az ETSGLOBAL szkópban fog állni először. Itt megpróbálja a feloldást végrehajtani, és ha sikerül akkor visszatér a feloldott értékkel. Ha nem sikerül akkor megpróbálja a feloldást végrehajtani a tényleges globális szkópban. Ha ez sem sikerül akkor a visszaadott *binder::ConstScopeFindResult binder::Variable* pointere *nullptr* lesz.

A **Kód Részlet 6** mutatja az implementált függvényt.

```
ConstScopeFindResult Scope::FindInGlobal(const util::StringView
&name, const ResolveBindingOptions options) const {
    const auto *scope_iter = this;
    // One scope below true global is ETSGLOBAL
    while (!scope_iter->Parent()->IsGlobalScope()) {
        scope_iter = scope_iter->Parent();
    }

    auto *res = scope_iter->FindLocal(name, options);
    if (res == nullptr) {
      /* If the variable cannot be found in the scope of the
      local ETSGLOBAL, then we still need to check the true
       global scope which contains all the imported ETSGLOBALs */
        res = scope_iter->Parent()->FindLocal(name, options);
    }

    return {name, scope_iter, 0, 0, res};
}
```

Kód Részlet 6: Az implementált globális szkópban kereső függvény.

4.2.2.3. A függvény szkópban kereső függvény

A függvény szkópban kereső függvény implementációja valamivel egyszerűbb, mint a globális szkópokban kereső függvényé. Itt azonban azt kell figyelembe venni, hogy egy függvényben sok féle szkóp lehet például *binder::LoopScope*. Ezek a szkópok közül csak az osztály és a globális szkópokat kell kiszűrni.

Implementációját tekintve a függvény az aktuális szkóptól kezdeve iterál felfelé a szkópokon ameddig a következő szülő szkóp nem osztály vagy globális szkóp. Itt fontos kiemelni, hogy ennek a szkóp iterációnak kezelnie kell azt az esetet amikor a szülő szkóp *nullptr*, mert előfordulhat, hogy a keresés a tényleges globális szkópból indul. Minden szinten történik egy kísérlet a feloldásra. A feloldás eredménye hasonlóan alakul a globális feloldó függvény eredményéhez. Ebben az esetben is egy *binder::ConstScopeFindResult* lesz az eredmény, aminek *a binder::Variable* pointere *nullptr* ha nem létezik a függvény szkópjában a keresett azonosító.

A **Kód Részlet 7** mutatja az implementált függvényt.

```
ConstScopeFindResult Scope::FindInFunctionScope(const
util::StringView &name, const ResolveBindingOptions options)
const {
    const auto *scope_iter = this;
    while (scope_iter != nullptr && !scope_iter->IsClassScope()
            && !scope_iter->IsGlobalScope()) {
        if (auto *const resolved =
            scope_iter->FindLocal(name, options);
            resolved != nullptr) {
             return {name, scope_iter, 0, 0, resolved};
        }
        scope_iter = scope_iter->Parent();
    }

    return {name, scope_iter, 0, 0, nullptr};
}
```

Kód Részlet 7: Az implementált függvény szkópban kereső függvény.

5. A Szemantikai Elemző

A Szemantikai Elemző komponens végzi a legnehezebb feladatot a fordítóprogramban. Feladata a statikus típus információk előállítása, statikus és szemantika analízisek elvégzése. Ilyen analízisek például az Objektum Orientált programozási paradigmából adódó öröklődési viszonyok ellenőrzése, virtuális függvény hívások feloldása, vagy lambda függvények capture változóinak ellenőrzése, és még sok egyéb fordítási idejű analízis.

5.1. A típus ősosztály

Maga a típus ősosztály a *checker::Type* nem rendelkezik sok adattaggal. Két adattagját érdemes említeni, az egyik egy *checker::TypeFlag* típusú tag, ami nevéből adódóan különböző flageket tárol egy típuson. Ilyen flagek például, hogy az adott típus primitív vagy referencia-e, esetleg szintetikus a Szemantikai Elemző által előállított. A másik fontos adattagja egy *binder::Variable* pointer típusú adattag. Ezzel az adattagal csak olyan típusok rendelkeznek, amelyekhez valamilyen felhasználó által létrehozott típusnév is tartozik. Ilyenek például az osztályok, vagy a típus aliasok.

A típus ősosztály deklarál sok pure virtuális függvényt. Néhány egyszerűbb virtuális függvényei:

- *checker::Type::ToString* a típus string reprezentációja.
- *checker::Type::ToAssemblerType* egy string típust ad vissza, ami a felhasználó által deklarált típus assembler neve lesz a bytecode-ban.
- *checker::Type::ToAssemblerTypeWithRank* csak a tömb típusok használják, a tömb típusnevén túl reprezentálja a méretét is, különben úgy működik, mint az előző függvény

A típus ősosztály komplexebb virtuális függvényei a típusok relációival foglalkoznak, például:

- *checker::Type::Identical* eredménye igaz, ha a két típus megegyezik.

- *checker::Type::AssignmentTarget* megvizsgálja, hogy ha ez a típus egy értékadás bal oldalán áll, akkor a jobb oldalon álló kifejezés típusa kompatibilis-e vele.
- *checker::Type::Cast* megpróbálja a paraméterül kapott típust az aktuálisra castolni
- *checker::Type::IsSubType* megnézi, hogy a paraméterül kapott típus leszármazott típusa-e az aktuális típusnak

5.1.1. Az ETSObjectType típus

A *checker::ETSObjectType* típusról érdemes külön beszélni mert ez a legfontosabb előállított típus. Az ETS nyelv legtöbb típusa referencia típus a Javahoz hasonlóan, így minden típus, ami nem primitív, vagy enum ebből a típusból öröklődik.

A *checker::ETSObjectType* sok különböző adattaggal rendelkezik. *std::vector-ban* tárolja a következő tulajdnoságait:

- A típus argumentumait, ha generikus típusról beszélünk.
- A konstruktor szignatúráit.
- Az általa implementált interfészeket

Ezeken felül rendelkezik egy saját *checker::ETSObjectFlags* típusú flag adattaggal, ami a *checker::TypeFlags* felett még sok más tulajdonságot is beállít. A legfontosabb adattagja a *checker::ETSObjectType::PropertyHolder* nevű típus, ami a háttérben egy hat elemű tömb. A tömb minden eleme egy *std::unordered_map*. Ezek a mapek tárolják az összes propertyt egy osztályon. Ezek a propertyk lehetnek:

- Példány függvények.
- Statikus függvények.
- Példány adattagok.
- Statikus adattagok.
- Példány deklarációk, például inner belső osztály
- Statikus deklarációk., például statikus belső osztály

Ezen mapek nyilvántartása, és bennük az adott kontextusban a megfelelő property megtalálása a *checker::ETSObjectType* legfőbb feladata.

5.2. A check függvény

Az *ir::AstNode::Check* pure virtuális függvény az, amit a Szemantikai Elemző használ. Ezt a függvényt hívja a gyökér node-tól kezdődően lefelé, mélységi bejárást alkalmazva. Előfordulhat azonban, hogy egy node típusa már egyszer létre lett már hozva, azonban újra ráfut a függvényre a vezérlés. Ebből az okból kifolyólag minden node függvénye először le ellenőrzi, hogy van-e már típusa, és ha van egyszerűen csak visszatér azzal.

5.3. A Szemantikai Elemzőn végzett fejlesztéseim

A Szemantikai Elemzőn számos fejlesztést végeztem, ezek közül a nagyobb fejlesztéseket szeretném bemutatni. Ez konkrétan négy új működés implementálását jelentette, az első a beépített típusok típusinformációinak előállítása, a második az immutable adattagok egyszeri inicializálásának ellenőrzése, a harmadik a **4.2.2-es** fejezetben említett azonosító feloldás, a negyedig pedig az absztrakt osztályok és függvényekre vonatozó szemantikai analízis implementálása.

5.3.1. A beépített típusok inicializálása

A beépített típusok típusinformációira mindig szükség van. Ilyen típusok például az *std.core.Object* ami az osztály hierarchia gyökere, vagy a *Console* objektum amely a standard kimenetre tud írni. Ezen típusok előállítása a megfelelő *ir::AstNode::Check* függvényeken keresztül történt, mint minden más node esetében is. Azonban ez a lazy módszer nem működött jól, sok helyen okozott problémát, hogy egy típus már inicializáltnak tekintette a *checker::GlobalTypesHolder-ben* lévő beépített típusokat.

5.3.1.1. A megvalósítás

A megvalósítás a *checker::InitializeBuiltins* nevű függvény implementálása. Ezt a függvényt hívja a *checker::StartChecker* függvény, ami a Szemantikai Elemző belépési pontja. Az implementált függvény első lépése, hogy megvizsgálja, hogy a beépített

típusok inicializálva vannak-e már. Ez egy biztonsági lépés mivel minden fordítási egységnek saját Szemantikai Elemzője van, viszont a beépített típusokat elég egyszer megvizsgálni. A következő lépésben a globális szkópban megkeresem a beépített *std.core.Object* osztályt majd inicializálom. Azért kezdek ezzel a típussal, mivel minden más típusnak be van kötve, mint ős, így garantált, hogy más típusok vizsgálata során ez a típus már előállt. Ezután végig iterálok a globális szkóp minden interfész és osztály típusán és egyesével beinicializálom őket a *checker::GlobalTypesHolder::InitializeBuiltin* függvényével. Ez a függvény megkeresi a beépített típusok között az adott típust, majd a kiszámított típusinformációt elmenti. Előfordulhat, hogy a fejlesztés során a standard függvénykönyvtárba olyan típus kerül be, ami még nem létezik a *checker::GlobalTypesHolder-ben*, ekkor, ha az Es2panda debug módban van buildelve akkor a loggeren keresztül egy figyelmeztetést adok ki, hogy az új típust fel kellene venni a beépített típusok közé.

5.3.2. Immutable adattagok egyszeri inicializálása

Egy osztály immutable adattagjait pontosan egyszer lehet inicializálni. Ezt az inicializációt két féle képpen lehet elvégezni, vagy az adattag deklarációjával egy időben (inline), vagy attól függően, hogy statikus vagy példány adattag az osztály static blokkjában, vagy konstruktorában.

5.3.2.1. A megvalósítás

A megvalósításom erre a problémára a következő. Először is végig iterálok az összes immutable addatagján az osztálynak a *checker::ETSChecker::CheckConstFields* függvényben. Második lépésben, ha az adott adattag statikus, akkor összegyűjtöm az osztályon lévő összes static blokkot, ha példány akkor pedig az összes konstruktort. Példány esetben megnézem, hogy a konstruktor első utasítása az egy másik konstruktorba delegáló hívás-e, mert ha igen akkor itt a változót inicializáltnak lehet tekinteni. Tovább menve végig iterálok a konstruktor vagy static blokk teljes utasítás listáján, és ha találok még egy elemet, ami inicializálná az adattagot, akkor hibát dobok.

A megvalósításnak van egy hibája, hogy nem veszi figyelembe a vezérlési gráfot. Így előfordulhat a kódban olyan hely, ahol az inicializáló kifejezés elérhetetlen ágon van. A fordított eset is előállhat, amikor hibásan dob hibát mert a második inicializáló kifejezés van elérhetetlen ágon. Olyan hiba is előállhat, hogy az inicializáló kifejezés egy ismétléses vezérlési szerkezetben van, ami szintén többszöri inicializáláshoz vezethet, azonban ezt a hibát eltekintve az előző kettőtől lehet orvosolni azzal, hogy a kifejezés szülei között megnézzük, hogy van-e ilyen vezérlési szerkezet. Mivel a Control Flow Graph (CFG) komponens még implementáció alatt áll, így az előbb említett problémákat még nem lehetett javítani.

5.3.3. Az azonosító feloldás módosítása ETS esetben

A **4.2.2-es** fejezetben ismertettem az ETS nyelv azonosító feloldási szabályait, és bemutattam, hogy ehhez milyen fejlesztéseket kellett elvégezzek a Binder komponensben. Most a Szemantikai Elemzőt érintő módosításokról fogok beszélni.

5.3.3.1. A megvalósítás

A megvalósítás során két esetet kellett figyelembe vegyek.

Az első eset amikor egy azonosító önmagában áll, kvalifikáló kifejezés nélkül. Ebben az esetben először is megpróbálom feloldani a változót a függvény szkópjában. Ha ez nem sikerül akkor újra próbálom a feloldást a globális szkópban. Ez után a *checker::ValidateResolvedIdentifier* függvényt hívom. Ez a függvény végzi a feloldott változó validálását. Az első lépésben ellenőrzi a függvény, hogy sikerült-e a feloldás, és ha nem akkor megpróbálom feloldani az aktuális osztály szkópjában. Ezt azért teszem, hogy a felhasználónak jobb fordítási idejű hibaüzenetet tudjak adni. Például a felhasználó egy osztály adattagot szeretett volna elérni, de elfelejtette kirakni a *this* kulcsszót, akkor a hibaüzenet megmutatja neki, hogy a hivatkozott változót milyen kvalifikáló kifejezésen keresztül lehet elérni. A további vizsgálatok a feloldott azonosító által hivatkozott típusra, vagy változóra vonatkoznak, hogy az adott kontextusban helyes-e használni.

A második eset amikor egy kvalifikáló kifejezéssel rendelkező azonosítót kell megvizsgálni. Ebben az esetben először is össze kell állítani a jelenlegi kontextusból, hogy az osztályban adattagot, függvényt vagy deklarációt keresünk és az statikus-e vagy sem. Ennek a műveletnek az eredménye egy *checker::PropertySearchFlags* típusú flag lesz, amivel a *checker::ETSObjectType* property-jei között lehet keresni. Ezután megtörténik a feloldás. Az első esethez hasonlóan itt is validálva van a feloldott property, ezt a *checker::ValidateResolvedProperty* végzi. Ez a függvény megpróbálja a propertyt úgy feloldani, hogy ha eddig statikust keresett akkor most példányt és fordítva. Ez szintén a jobb hibaüzenet érdekében történik. A függvénynek ehhez meg kell fordítani a kereső flaget. A flagben hat különböző bit van, amik a keresett property típusokat kódolják. Ezt a hat bitet kell egymással kicserélni. A **Kód Részlet 8** mutatja a cserélő algoritmust.

```
unsigned int i, j; // positions of bit sequences to swap
unsigned int n;    // number of consecutive bits in each
sequence
unsigned int b;    // bits to swap reside in b
unsigned int r;    // bit-swapped result goes here

// XOR temporary
unsigned int x = ((b >> i) ^ (b >> j)) & ((1U << n) - 1);
r = b ^ ((x << i) | (x << j));
```

Kód Részlet 8: A bit kicserélő algoritmus. [1]

Mivel ez az algoritmus általános, viszont az én esetemben az *i* és *j* pozíciók, valamint az *n* konstans, így a művelet átalakítottam az alábbiak szerint: **Kód Részlet 9.**

```
using Utype = std::underlying_type_t<PropertySearchFlags>;
const auto flags_num = static_cast<Utype>(flags);
const Utype x = (flags_num ^ (flags_num >> 3U)) & 7U;
const auto new_flags = PropertySearchFlags {flags_num ^ (x | (x
<< 3U))};
```

Kód Részlet 9: Az implementált egyszerűsített bit cserélés

Ebben a kódban az *std::underlying_type_t* visszaadja egy enum típus háttér típusát, ez a típus minden esetben valamilyen előjeles vagy előjel nélküli egész típus, ez lesz az *Utype*. Ezután az flag enumot átkonvertálom a háttér típusra. Ezután létrehozom az XOR művelet részeredményt az *x* változóba, majd a *new_flag* változóba kerül a bit cserélés eredménye.

5.3.4. Absztrakt osztályok és függvények szemantikai analízise

Az ETS-ben is mint más objektum orientált nyelvekben létezik az absztrakt osztály fogalma. Egy absztrakt osztály olyan osztály, amelynek van legalább egy absztrakt függvénye. Egy függvény akkor absztrakt, ha nincs implementációja és szerepel rajta az *abstract* kulcsszó. Absztrakt osztályból nem lehet példányt létrehozni, továbbá az absztrakt osztály leszármazottjai kötelesek minden absztrakt függvényének implementációt adni. Az implementációm a következők.

Ahhoz, hogy egy absztrakt függvényt meg tudjunk vizsgálni tudnunk kell a Szemantikai Elemzőben, hogy milyen osztályban állunk éppen. Ennek érdekében létrehoztam egy új flaget a *checker::CheckerStatus::IN_ABSTRACT* névvel. Ez a flag akkor kerül fel amikor létrejön egy új *checker::CheckerContext*.

Első lépésben leteszteltem, hogy az aktuálisan vizsgált függvény absztrakt-e, majd megnézem milyen más módosítók vannak még rajta. A *native*, a *private*, az *override*¸az *open* és a *static* módosítókat nem lehet együtt használni az absztrakt módosítóval. Mivel ezek mind flagek egy *ir::MethodDefinition* AST node-on, ezért az illegális módosítókat bitenkénti vagyolással összekombinálom, majd egy bitenkénti éseléssel megnézem, hogy bármelyik illegális módosító rajta van-e. Ha igen akkor fordítási hiba dobok.

Második lépésben pedig leteszteltem a fentebb említett flaget, hogy absztrakt osztályban áll-e a Szemantikai Elemző. Ha nem, és a függvény mégis absztrakt akkor hibát fordítási hibát dobok. A **Kód Részlet 10** mutatja az implementált analízist.

```
auto not_valid_in_abstract = ir::ModifierFlags::NATIVE |
ir::ModifierFlags::PRIVATE | ir::ModifierFlags::OVERRIDE |
                             ir::ModifierFlags::OPEN |
                             ir::ModifierFlags::STATIC;
if (IsAbstract() && (flags_ & not_valid_in_abstract) != 0U) {
    checker->ThrowTypeError("", Start());
}
if (IsAbstract() &&
    !(checker->HasStatus(checker::CheckerStatus::IN_ABSTRACT))) {
    checker->ThrowTypeError("", Start());
}
```

Kód Részlet 10: Az implementált analízis. A hibaüzenet szövegét a tömörség érdekében kihagytam.

6. A Kód Generátor

A Kód Generátor feladata a típusokkal ellátott AST-ből, bytecode-ot generálni. Ehhez kettő építőelemre van szüksége. Az első a *compiler::IRNode*. Ez az osztály reprezentál a bytecode-ban egy utasítást. A másik fontos építőelem a *compiler::VReg*. Ez az osztály reprezentál egy allokált regisztert. A Kód Generátor az Es2panda egyetlen párhuzamosan futó komponense, minden egyes lefordított függvényhez létrejön egy példánya.

6.1. A Panda Bytecode

Mielőtt rátérhetnék a fent említett két elemre, beszélni kell az Es2panda által generált bytecode-ról. Ez a bytecode alapvetően egy regiszter alapú gép, viszont van egy kitüntetett regisztere az akkumulátor, amit a legtöbb utasítás impliciten használ. A Panda Bytecode típusos, a legtöbb utasítás több különböző típusra is elérhető. Néhány példa a teljesség igénye nélkül:

- *mov v1, v2* egy 32 bites értéket mozgat *v2-ből v1-be*
- *mov.64 v1, v2* egy 64 bites értéket mozgat *v2-ből v1-be*
- *mov.obj v1, v2* egy referenciát mozgat *v2-ből v1-be*

Az említett *mov* és változatai egy elég általános utasítás, mivel az operandusai típusát nem köti szigorúan, csak a méretüket. Az *fmovi* utasítás már szigorúbb, mivel ez egy 32 bites konstans lebegőpontos számot másol a cél regiszterbe.

A Panda Bytecode feltételezést tesz, hogy egy függvény argumentumait kevesebbszer használja, mint a lokális változóit, ezért az alacsonyabb regiszter indexeket a lokális változóknak osztja ki, míg a magasabbakat a paramétereknek. Ez a kiosztás lehetővé tesz egy optimalizációt a bytecode-on mégpedig, hogy a kevésbé használt utasításokat két bájton tudja kódolni, ez a rövid utasítás formátum. Nyolc bit kódolja az operációs kódot, a fennmaradó nyolc bit pedig vagy egy, vagy két regisztert kódol. Ez azt jelenti, hogy egy rövid utasításnak egy regiszteres esetben a maximum regiszter indexe kétszázötvenhat, két regiszteres esetben pedig tizenhat.

A normál formátumú utasítások legalább három bájton kódoltak, és általában két maximum kétszázötvenhatos indexű regisztert tudnak címezni, ez alól kivétel a *mov* utasítás, ami 2^{32} indexig képes címezni mindkét regiszterét.

6.2. Az IRNode

A *compiler::IRNode* osztály létesít absztrakciót a Panda Bytecode utasításai felett. Ez az osztály két pure virtuális függvénnyel ír le egy utasítást. Az első függvény a *compiler::IRNode::Registers* visszaadja az utasítás által használt regiszterek számát, és egy tömbben a regisztereit. A másik a *compiler::IRNode::GetFormats* függvény az utasítás formátumait adja vissza, azaz mekkora indexű regiszter operandusokat képes fogadni.

A *compiler::IRNode* tartalmaz továbbá egy *compiler::IRNode::Transform* függvényt, ami az Es2panda belső reprezentációjáról átalakítja az utasítást a runtime nyújtotta reprezentációra.

6.3. A VReg

A *compiler::VReg* egy vékony absztrakciós réteg a regiszter indexelés felett. Gyakorlatilag csak aritmetikai műveletek valósít meg, és specializálja a *std::hash* osztályt, hogy asszociatív adatszerkezetekben is tárolható legyen.

6.4. A Kód Generátoron végzett fejlesztéseim

A Kód Generátoron végzett fejlesztéseim a regiszter spillelés ETS nyelv esetén, valamint a regiszter allokálás implementálása volt. Ezen felül az előbb említett *compiler::VReg* osztályon a C++ legjobb praktikák jegyében minden függvényt c*onstexpr-ré* tettem. Ez azért lehetséges, mert a *compiler::VReg* gyakorlatilag csak egy *std::uint32_t-t* tartalmaz, így az egész osztály kiértékelhető constexpr kontextusban. Továbbá implementáltam még a feltételes vezérlési szerkezet lefordítását is.

6.4.1. A regiszter allokálás

A regiszter allokálás legtrükkösebb része a Panda Bytecode által megszorított allokálási stratégia betartása. A stratégia, ahogy a **6.1-es** fejezetben is említem a lokális változók regisztereit az alacsonyabb indexeken, míg az argumentumok regiszterei a magasabb indexeken foglalódnak, közvetlen a lokális változók után.

Az ETS nyelv viszont sok olyan konstrukcióval rendelkezik, amihez ideiglenes operációkat végezni. Ahhoz, hogy ezeket elvégezzük az operációk eredményeit regiszterekbe kell menteni. Ahhoz, hogy el tudjuk menteni őket ki kell számolni a következő regiszter indexet. Ehhez tudnunk kell, hogy mennyi lokális regisztert használunk. Nos, ez egy körkörös dependencia.

local n	local 1	local 0	arg 0	arg 1	arg n
65535 - n	65534	65535	65536	65537	65536 + n

Táblázat 1: A regiszter kiosztási stratégia.

Ahogy a **Táblázat 1** mutatja a regiszter kiosztás a következőképpen történik a Kód Generátor futása során. A lokális változók *UINT16_MAX-tól* indulnak és lefelé nőnek, ez lehetőséget ad 65535 darab lokális változóra. Az argumentum regiszterek 65536-tól indulnak és tartanak *UINT32_MAX-ig*. A gyakorlatban sem a lokális, sem az argumentum regiszterek nem érik el az értéktartományaik maximumát.

Szóval a kód generálás során a folyamatosan allokált regiszterek lefelé nőnek, és minden spill-fill utasítás azonnal kigenerálásra kerül. A spill-fill utasítások olyan utasítások, amelyek arra szolgálnak, hogy egy regiszter értéke el legyen mentve, ez a spill, majd egy adott operáció után vissza legyen állítva, ez a fill. Amikor a Kód Generátor terminál a regisztereket újra rendezi. A lokális regiszterek megtükröződnek az *uint16_t* értéktartományban, a paraméter regiszterek pedig redukálódnak *UINT16_MAX* mínusz az összes regiszter számával, így a felhasznált regiszterek teljesítik a Panda Bytecode által előírt allokálási stratégiát.

6.4.2. A regiszter spiller

A regiszter spilleléshez szükség van típus információra a regiszterekről ahhoz, hogy típusos bytecode-ot lehessen emittálni. A regiszterek típusát a *compiler::CodeGen* példány vezeti egy *std::unordered_map-ben*, aminek a kulcsa a *compiler::VReg*, értéke pedig a *compiler::CodeGen::VRegType.*

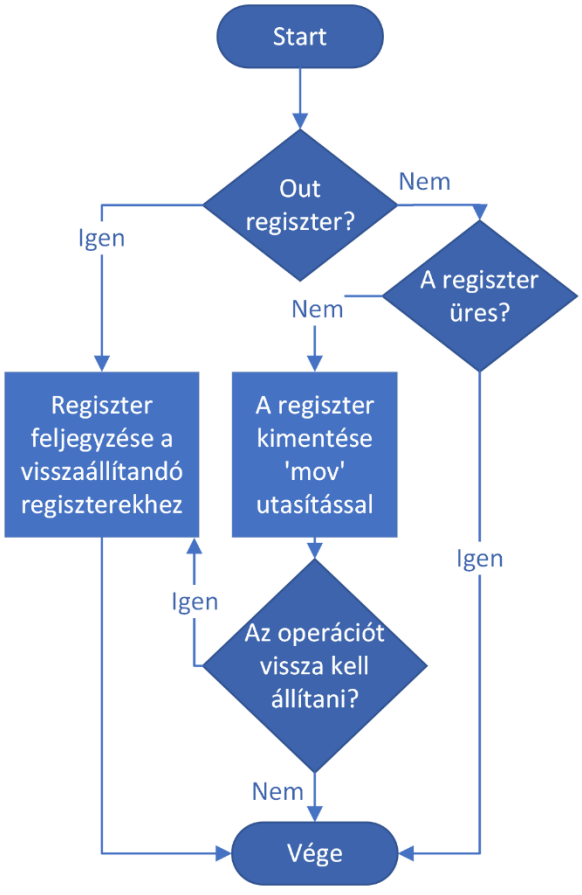
A *compiler::CodeGen::VRegType* reprezentálja a három állapotot, amiben egy regiszter típusa állhat. Egy regiszter típusa lehet *nullptr*, lehet egy konkrét típus, vagy lehet neki jövőbeli típusa. A jövőbeli típus azt jelenti, hogy a regiszter egy *out* regiszter, azaz a következő utasítás eredményként fog értéket kapni. A jövőbeli típus nem lehet *nullptr*, ezt az osztály interfész meg is követeli. A háttérben az osztály az *std::variant* típus biztonságos unió típussal van implementálva. Ezzel gyakorlatilag a *compiler::CodeGen::VRegType* is egy unió típus, ahol az unió elemei a három lehetséges állapot. A **Kód Részlet 12** mutatja a *compiler::CodeGen::VRegType* interfészét.

A típus információk birtokában már meg lehet kezdeni a regiszter spillelést. A *compiler::RegAllocator* minden egyes kiemittált utasítás előtt megnézi, hogy az utasítás regiszterei megfelelnek-e az utasítás formátumainak követelményeihez, ez a *compiler::RegAllocator::CheckRegIndicies* függvényben történik. Ha a regiszterek nem felelnek meg az utasítás formátumainak, akkor a *compiler::RegSpiller* megkezdi a regiszter spillelést. Az ETS nyelv regiszter spillelését a *compiler::StaticRegSpiller* végzi. A spiller működése a következő, először is allokál egy új regisztert a nulla és tizenöt közötti 4 bites értéktartományban, ha ennek az új regiszternek már van típusa, azaz értéke is, akkor átmozgatja a következő üres regiszterbe, ez egy spill utasítás. Ezen mozgatásokat feljegyzi a Spiller egy *std::vector-ba* mivel az új regiszter értékét majd vissza kell állítani. Az *std::vector* elemei *compiler::RegSpiller::SpillInfo* osztály példányok. Ez az osztály két *compiler::VReg* adattaggal rendelkezik. Gyakorlatilag egy spill utasítást reprezentál, hogy melyik regiszterből hová került az érték. Van rajta egy *compiler::RegSpiller::SpillInfo::Reversed* függvény is, ami egy másolatot ad vissza az osztályról, a regiszterek megcserélésével. Ez a függvény a fill utasításkor lesz hasznos.

Ezután a használt regisztert átmozgatja az új regiszterbe, ha van típusa. Ha out regiszterről beszélünk, aminek jelenleg még nincs típusa, akkor a spiller feljegyez egy

mozgatást az *std::vector-ba*, annak ellenére, hogy nem történik tényleges spill utasítás. Erre azért van szükség mert a hívó kód az általa allokált regiszterben várja az értéket, az utasítás viszont csak a [0..15] intervallumon tud dolgozni, ezért az utasítás után rá kell venni a Spillert, hogy mozgassa ki az értéket az out regiszterből a hívó által allokáltba. Ez egy speciális eset és a legtöbb utasítás esetében nincs rá szükség, csak akkor, ha egy utasítás tisztán csak regisztereken dolgozik, 4 bites címezést használ, és az eredményét az akkumulátor helyett egy regiszterbe teszi. Ilyen utasítások az új referencia típus létrehozó *newobj*, és tömböt létrehozó *newarr* utasítások.

A *compiler::RegAllocator* kicseréli a *compiler::IRNode* regisztereit az újonnan allokáltakra. Ezután a Spiller visszafelé bejárja, az *std::vector-ját* és visszamozgatja a regisztereket eredti helyükre, ez a fill utasítás. Az **Ábra 1** mutatja a regiszter spillelés folyamat ábráját.



Ábra 1: A kidolgoztt regiszter spiller folyamatábrája.

6.4.3. A feltételes vezérlési szerkezet lefordítása

A feltételes vezérlési szerkezet lefordításának implementációja hasonlóan történik más nyelvek feltételes vezérlési szerkezetéhez.

Az implementációm során először is megvizsgálom, hogy a vezérlési szerkezet feltétele konstans kiértékelhető-e. Ha az akkor fordítási időben elég ezt a kifejezést kiértékelni, és vezérlési szerkezetnek csak egyik ágát kell lefordítani.

Ha ez az eset nem áll fent akkor először is allokálni kell egy *compiler::Label-t* amire akkor ugrik a vezérlési szerkezet, ha a feltétel hamis. Ezután le kell fordítani a feltételt. Ha a vezérlési szerkezet több ággal is rendelkezik, akkor mindegyikághoz allokálni kell egy további *compiler::Label-t*. Majd az alternatív ágat is le kell fordítani. Ez az *else if* esetében egy rekurzív hívást jelent. A végén frissíteni kell a Kód Generátor utolsó ugrási pontra mutató *compiler::Label* pointerét.

6.5. Az Emitter

Az Emitter komponens gyakorlatilag egy adapter réteg az Es2panda és az Ark runtime Program reprezentációi között, így egy alkomponense a Kód Generátornak. Az Emitter komponens két alkomponensből áll. Az egyik a Record Emitter a másik pedig a Function Emitter.

A Record emitter felelősség a létrehozott osztályok emittálása. Az osztályokat a Rekord emitter a Binder komponensből gyűjti össze.

A Function emitter felelőssége a létrehozott függvények emittálása. A Function Emitter a Kód Generátor után fut egyből, tehát amint egy CodeGen példány lefordított egy függvényt, a Function Emitter átfordítja az Ark runtime reprezentációjára.

6.5.1. Az Emitter komponensen végzett fejlesztéseim

A Panda Bytecode rekordjai és függvényei is elláthatók annotációkkal. Ezek az annotációk lehetnek olyanok, amelyek futásidőben is megmaradnak, de lehetnek olyanok, amelyeket az Ark runtime a bináris beolvasásakor eldob. Az annotációk információiból

később a debugger tud majd dolgozni, esetleg az Ark runtime just-in-time (JIT) fordítója tud végrehajtani optimalizációkat, vagy futás idejű kód reflexiót lehet vele megvalósítani. Az Emitter alkomponensen a generikus osztályok és függvények annotáció emittálását fejlesztettem. A generikus osztályok és függvények típus paraméterei rendelkezhetnek felső korlátokkal. Minden egyes kiemittált rekordhoz tartozik egy annotáció, ha generikus osztály volt, ami felsorolja a típus paramétereit és a felső korlátjaikat, ha van nekik.

Kidolgoztam továbbá egy adapter réteget, ami képes az Es2panda osztály reprezentációján használt flageket átfordítani az Ark runtime flagjeire.

A megvalósítás során az annotációkat generáló függvényben a már kiemittált annotációkat elmentem egy cache-be. Ezt a **Kód Részlet 11** mutatja.

```
void ETSEmitter::GenAnnotationRecord(std::string_view
record_name_view, bool is_runtime, bool is_type) {
    const std::string record_name(record_name_view);
    const auto record_it = Program()->record_table.find(
                                                    record_name);
    if (record_it == Program()->record_table.end()) {
        pandasm::Record record(record_name, EXTENSION);
        record.metadata->SetAttribute(Signatures::EXTERNAL);
        record.metadata->SetAttribute(
           Signatures::ANNOTATION_ATTRIBUTE);
        if (is_runtime && is_type) {
            record.metadata->SetAttributeValue(
           Signatures::ANNOTATION_ATTRIBUTE_TYPE,
           Signatures::RUNTIME_TYPE_ANNOTATION);
        } else if (is_runtime && !is_type) {
            record.metadata->SetAttributeValue(
           Signatures::ANNOTATION_ATTRIBUTE_TYPE,
           Signatures::RUNTIME_ANNOTATION);
        } else if (!is_runtime && is_type) {
            record.metadata->SetAttributeValue(
           Signatures::ANNOTATION_ATTRIBUTE_TYPE,
           Signatures::TYPE_ANNOTATION);
        }
        Program()->record_table.emplace(record.name,
                                        std::move(record));
    }
}
```

Kód Részlet 11: Az implementált annotáció generáló függvény.

7. Összefoglaló

Összességében a fejlesztéseim az Es2pandában hasznosnak bizonyultak. Sikerült olyan rendszereket felállítsak, amelyek könnyen érthetőek és robusztusok. Célom volt a fejlesztések során az érintett kódok const korrektségének javítása, és a modern C++ legjobb gyakorlatok betartása, amely helyenként teljesítmény növekedéshez is vezetett, azonban a nagyobb jelentősége az implementáció során felderített bugok tekintetében van.

Sikeresen implementáltam egy robosztus rendszert az Elemző komponensben a default és named paraméterekhez szükséges proxy függvény létrehozására. Implementáltam az ETS nyelv szabványa által megkövetelt komplex azonosító feloldási szemantikának megfelelő rendszert. Ezen felül implementáltam a Szemantikai elemzőben több statikus analízist is. Továbbá a Kód Generátor komponensben létrehoztam a regiszterek spill-fill utasításainak emittálására egy alrendszert, amely magas absztrakciós szintjének köszönhetően könnyen használható. Létrehoztam egy bytecode annotálási rendszert is a Kód Generátor alkomponensében, az Emitterben.

Természetesen mindig van lehetőség tovább fejlesztésre. A Kód Generátor komponens interfésze például túl gyenge megszorításokat tesz a Javascript és az ETS Kód Generátoraira ezért érdemes volna virtualizálni ezt az interfészt.

Sok helyen lenne érdemes jobban felbontani a Javascript és az ETS nyelvi elemeket kezelő kódok interfészeit a jobb kód fenntarthatóság érdekében. Ez kifejezetten igaz a Binder komponensre és függőségeire. A Binder komponens jelenleg majdnem ugyanúgy működik ETS esetén is mint Ecmascript esetén. Ez nyilvánvalóan nem tesz jót a kódfenntarthatóságnak. Vannak olyan szabályok is mint például az azonosító feloldás, melynek procedúrája teljesen máshogy működik a két nyelv között mégis egy azonos komponensen fut át. Ebben a komponensen rengeteg realizálatlan fejlesztési lehetőség van.

A Szemantikai Elemző komponenst érdemes volna generációkra bontani a jelenlegi mélységi bejárás helyett. Ennek az lenne az előnye, hogy az AST node-okon elvégzett operációk sorrendje jobban definiált lehetne. A felbontás további előnye, hogy a Szemantikai Elemző komponens túl monolitikus, a fenntarthatatlanság határán van, ezért

átláthatóbb folyamatokhoz vezetne egy jobb nomenklatúra és esetleges Checker osztály hierarchia.

Az Elemző komponenst is lehetne tovább fejleszteni, bár messze ez a legstabilabb komponens a projektben, nagy fordítási egységek esetén ez a komponens sebessége határozza meg a teljesítmény maximumot. Ennek javítása érdekében lehetne a komponenst párhuzamosítani.

Irodalomjegyzék

[1] Sean Eron Anderson, Bit Twiddling Hacks, Swapping individual bits with XOR. http://graphics.stanford.edu/~seander/bithacks.html#SwappingBitsXOR
Legutolsó látogatás: 2023. 05. 08.

[2] Ark Compiler Runtime Core. https://gitee.com/openharmony-sig/arkcompiler_runtime_core
Legutolsó látogatás: 2023. 05. 08.

[3] Es2panda. https://gitee.com/openharmony-sig/arkcompiler_ets_frontend
Legutolsó látogatás: 2023. 05. 08.

Nyilatkozat

Alulírott Szűcs Dániel mérnökinformatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, mérnökinformatikus 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 / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

2023.

Szűcs Dániel

Mellékletek

Kód Részletek és Ábrák

```
class VRegType final {
    using Variant = std::variant<const checker::Type *, const
checker::Type *>;

public:
    [[nodiscard]] static constexpr VRegType FromType(const
checker::Type *type) noexcept;
    [[nodiscard]] static constexpr VRegType FromFutureType(const
checker::Type &future_type) noexcept;

    [[nodiscard]] constexpr std::optional<const checker::Type *>
GetType() const noexcept;
    [[nodiscard]] constexpr std::optional<const checker::Type *>
GetFutureType() const noexcept;

private:
    constexpr explicit VRegType(Variant &&types) noexcept;

    Variant types_ {};
};
```

Kód Részlet 12: A *compiler::CodeGen::VRegType* deklarációja.