



Eötvös Loránd Tudományegyetem, Informatikai Kar,
Programozási Nyelvek és Fordítóprogramok Tanszék

Függvények modellezése globális statikus analízis céljából*

TDK Dolgozat
2014

Horváth Gábor
Programtervező Informatikus MSc,
I. évfolyam

Témavezető: **Pataki Norbert**, adjunktus

*A kutatást és fejlesztést az Ericsson Magyarország támogatta.

Tartalomjegyzék

1. Bevezetés	2
2. Statikus analízis	4
2.1. Módszerek a hibakeresésre	5
2.2. Statikus analízis módszerei	6
2.3. Statikus analízis kiterjedése	8
2.4. Modellek a statikus analízishez	10
3. Programozási nyelvek	13
3.1. Elemzőkkel kapcsolatos fogalmak	15
3.2. A C++ programozási nyelv	16
4. Clang és a Static Analyzer	21
4.1. Szimbolikus végrehajtás	21
4.2. Szimbolikus memóriakezelés	23
5. API modellezés	25
5.1. Gyorsan elemezhető modellek	27
5.2. Függvény modell szintetizációja	29
6. Implementáció	32
7. Továbbfejlesztési lehetőségek	36
8. Összefoglalás	37

1. fejezet

Bevezetés

A szoftverek költségének egyre nagyobb részét teszi ki a meglévő kódok javítása, karbantartása, bővítése. Minél több folyamatot sikerül automatizálni a szoftverfejlesztéssel kapcsolatban, annál nagyobb produktivitást lehet elérni. Az egyre hatékonyabb optimalizációk lehetővé teszik, hogy absztraktabb szinten dolgozhasanak a programozók. Az egyre hatékonyabb diagnosztikai eszközök számos hibát megtalálnak már azelőtt, hogy a programot lefuttatták volna, ezáltal rengeteg hibakereséssel eltöltött órát spórolhatnak meg a fejlesztőknek.

A *statikus analízis* egy olyan folyamat, mely a forráskódból próbál meg levonni következtetéseket, anélkül, hogy a kód futtatásra kerülne. Számos célra alkalmaznak statikus analízist alkalmazó eszközöket. A legfontosabb ilyen eszközök a fordítóprogramok, amik statikus analízis segítségével szolgáltatnak a felhasználónak különböző diagnosztikai üzeneteket, hibaüzeneteket, valamint az analízisek eredményeit felhasználva hajtják végre az optimalizációkat.

Sok esetben az algoritmusok, amelyek a statikus analízist hivatottak megvalósítani, nagyon költségesek. Mivel a fordítási idő a fejlesztésnek jelentős részét teszi ki, ezért a gyors fordítás érdekében az ilyen költséges algoritmusokat igénylő analíziseket külön eszközök szokták végezni. Már 1979-ben megjelent az első ilyen eszköz, amit *Lint*-nek neveztek.

Biztonságkritikus alkalmazások esetén gyakran használnak statikus analízist bizonyos típusú hibák kiszűrésére, valamint arra, hogy megállapítsák, az adott kód megfelel-e egy adott szabványnak. A biztonságkritikus alkalmazásokhoz használt szabványok gyakran azért tiltják meg bizonyos nyelvi elemek használatát, mivel azok hatását a jelenlegi statikus analízis eszközök nem tudják jól elemezni.

Általában statikusan típusos nyelveknél a típusrendszer arra hivatott, hogy számos hibalehetőséget megakadályozzon. Egy nagy kifejezőerejű statikus típusrendszer segítségével rengeteg szemantikus hiba kiszűrhető fordítási időben.

Azonban számos olyan hiba létezik, amit a jelenleg az iparban legtöbbször használt nyelvekben nem lehet kiszűrni a típusrendszer segítségével, viszont jól működő statikus ellenőrző programok fel tudják deríteni ezeket a hibákat.

Egyik jellemzője a statikus analízisnek a kiterjedtsége: mekkora egységet tud egyszerre vizsgálni a forráskódból. Bizonyos eszközök egyszerre egy forrásfájlt képesek csupán vizsgálni, ekkor nem tudnak olyan következtetéseket levonni, amikhez a szükséges információ egy része másik forrásfájlban található. Ennél fejlettebb eszközök már egy teljes fordítási foegységet vizsgálni tudnak. Sajnos ez is nagyon jelentős gyakorlati korlátot jelent arra nézve, hogy milyen típusú hibákat talál meg egy ilyen eszköz. Amennyiben az analízis során másik fordítási egységben található információk is elérhetőek, akkor minden információ rendelkezésére áll a forráskóddal kapcsolatban. Ez azonban még nem jelenti azt, hogy minden információ rendelkezésre állna az alkalmazással kapcsolatban, amit elemzünk. A végső cél a globális analízis, mikor az eszköz nem csupán a forráskódról rendelkezik minden információval, de a kód által felhasznált keretrendszerekről és könyvtárakról is.

A dolgozatom témája egy módszer, amellyel függvényeket lehet modellezni a globális analízis megvalósításának az elérése érdekében. Ki fogok térni arra, hogy egy függvény modellje alatt mit értünk, hogyan lehet egy ilyen modellt elkészíteni, és milyen függvényeket van értelme modellezni. Szó lesz modellek függvénydefiníciókból való szintetizálásáról is. A módszert egy *Google Summer of Code* [22] projekt keretén belül valósítottam meg. A megvalósítás bekerült a Clang fordítóba, ami egy Google és Apple által is támogatott nyílt forráskódú fordító C, C++ és Objective-C nyelvekhez [9]. Az itt bemutatott megoldás az első olyan public domain eszköz a fentebb említett nyelvekhez, amely képes hatékonyan támogatni a különböző fordítási egységekben lévő alprogramok invariánsainak feltérképezését, valamint alkalmas ezek megsértésének a detektálására.

A dolgozatban bemutatott módszer mellett annak Clang-es implementációját is tárgyalom, de tetszőleges interprocedurális analízist és szimbolikus végrehajtást támogató statikus analízis keretrendszerbe beilleszthető a technika. Bemutatok továbbá egy lehetőséget, amely segítségével függvények elő- és utófeltételeit lehet ellenőrizni szimbolikus végrehajtás közben, külön modellező vagy specifikációs nyelv használata nélkül.

2. fejezet

Statikus analízis

A statikus program analízist számos területen használják. A fordítóprogramokon túl a lint, refaktoráló, kódmetrikákat és tesztlefedettséget mérő és kódvizualizációs eszközök fejlesztésénél is előkerül. A sokszínűsége és széleskörű felhasználása mutatja a módszer erejét. Sajnálatos módon azonban számos elméleti korlátot ismerünk a statikus analízissel megoldható problémákra. A legfontosabb ilyen talán a „halting problem” (megállási probléma) néven ismert probléma: készítsünk olyan programot, aminek a bemenete egy program és annak egy lehetséges bemenete, és döntse el, hogy a bemenetként kapott program az adott hozzá tartozó bemeneten terminálna-e. Ez a probléma algoritmikusan eldönthetetlen [11]. A „halting problem”-re való visszavezetéssel könnyen belátható, hogy nem lehetséges olyan statikus ellenőrző szoftvert készíteni, ami minden lehetséges futási idejű hibát megtalál egy programban. Ennek ellenére léteznek olyan formális módszerek, ahol a program helyességével kapcsolatos logikai szabályokat statikus eszközök segítségével ellenőriznek. A gyakorlatban tehát jól használható eszközökről van szó.

Az újonnan megjelenő programozási nyelvek tervezésénél egy fontos szempont az, hogy a program szintaxisa mennyire alkalmas statikus analízisre. A Rust [13] névre keresztelt programozási nyelv például egy lépéssel tovább megy, a könyvtárak saját, a fordító statikus analízisét kiegészítő, beépülő modulokkal támogathatják meg a fordítási folyamatot. Ezt kihasználva létezik olyan Rust könyvtár, ami fordítás közben a string literálokban szereplő SQL lekérdezéseken végez szintaktikai ellenőrzést [14]. Ezáltal fordítási hibát ad, ha a programban lévő egyik SQL lekérdezésben szintaktikai hiba van. Ezzel a felépítéssel a Rust könyvtárak íróinak lehetősége van arra, hogy ott, ahol a típusrendszer nem elég erős ahhoz, hogy egy, a könyvtár használatával kapcsolatos szabályt betartassanak a felhasználóval, saját szabályokkal egészítsék ki a fordítási folyamat statikus ellenőrzését. Azokban a programozási nyelvekben, ahol létezik a szekvencia mint vezérlési szerkezet, kevés ren-

	Létezik hiba	Nem létezik hiba
Van hiba jelzés	True Positive	False Positive
Nincs hiba jelzés	False Negative	True Negative

2.1. ábra. Statikus analízis eredményeinek a besorolásai

delkezik olyan kifejező típusrendszerrel, hogy megköttést lehessen adni egy könyvtár függvényeinek a meghívási sorrendjére. Egy fordítási fázisba beépülő ellenőrzéssel egy Rust könyvtár könnyedén adhat hozzá ilyen jellegű szabályokat is.

A dolgozat folyamán a statikus analízisről elsősorban mint hibakereső módszerről lesz szó. Sajnos nem tökéletesek a statikus analízisre felhasznált eszközök. Ebből kifolyólag néha vétenek hibát. A hibát a típusától függően, külön besorolással illetjük. A besorolások a (2.1) táblázatban láthatók. Amennyiben egy hibát nem talál meg az eszköz, azt hívjuk False Negative-nak, ha pedig ad egy találatot, ami valójában nem hiba, azt False Positive-nak. Általában a False Positive-ok számának minimalizálása fontos, mivel ha a fejlesztő, aki javítaná a hibákat, a generált figyelmeztetések között elvértve talál valódi hibát, akkor nem fogja használni az eszközt. Biztonságkritikus alkalmazások esetében viszont a False Negative hibák minimalizálása a cél a minél megbízhatóbb működés érdekében.

2.1. Módszerek a hibakeresésre

Egy algoritmus helyességének a bebizonyítására speciális eszközök állnak rendelkezésre. Az algoritmusok helyessége nem feltétlen egyértelmű fogalom, nehéz formálisan leírni, hogy mikor számít egy algoritmus helyesnek. A gyakorlatban sokszor bizonyos tulajdonságokat akarunk belátni egy algoritmusra, például azt a tulajdonságot egy stabil rendezésre, hogy az ekvivalens elemek relatív sorrendjét megtartja.

Egy program *állapotterén* a program összes lehetséges állapotának a halmazát értjük, ahol a program egy állapota felírható a programban szereplő változók értékeiből képzett rendezett n -esként.

Kódrészletek (például alprogramok vagy adatszerkezetek műveleteinek [12]) specifikálásához szokták az elő- és utófeltételes specifikációt [4] alkalmazni. Egy statikus analízist végző eszköz a működése közben hasonló eszközöket használhat, mint amiket az algoritmusok tulajdonságainak a bizonyításakor szokás használni. Sokszor az algoritmusok bemenete egy bő halmaz, viszont nem várjuk el, hogy minden bemenetre helyesen működjön az adott algoritmus. Az *előfeltétel* segítségével azt

fogalmazhatjuk meg, hogy milyen tulajdonságoknak kell teljesülnie egy bemenetre ahhoz, hogy jól viselkedjen egy adott alprogram. Ha egy eszköz felismeri, hogy az egyik alprogram előfeltétele megsérül, akkor azt hibaként jelezheti. Ehhez azonban szükség van az előfeltételek feltérképezésére. Bizonyos esetekben gyenge előfeltételek automatikusan generálhatóak, más esetekben a programozónak van lehetősége, hogy megadja ezeket az alprogramokhoz.

Az *utófeltételek* segítségével írhatjuk le, hogy egy alprogram végrehajtása után milyen tulajdonságoknak kell teljesülnie a program állapotterére, valamint milyen összefüggés van az alprogram végrehajtása előtti és utáni állapotter között. Amennyiben egy alprogram végrehajtása után nem teljesül az utófeltétel, az azt jelenti, hogy az adott alprogram hibásan lett implementálva. Ha egy eszköz ismeri egy alprogram utófeltételét, akkor lehetősége van annak a teljesülését ellenőrizni. Általános esetben az utófeltétel ellenőrzése is eldönthetetlen, de a gyakorlatban számos esetben kivitelezhető.

Az *invariáns* tulajdonságok azok az állítások, amelyek bizonyos műveletsorozatok előtt és után is teljesülnek, pl. a *ciklusinvariáns* egy olyan állítás, aminek a ciklus minden iterációja végén igaznak kell lennie. A *típusinvariáns* pedig egy olyan állítás, aminek igaznak kell lennie minden a típushoz tartozó értékeken végzett művelet végrehajtása előtt és után. A típusinvariáns ellenőrzése úgy szokott történni, hogy az elő- és az utófeltételhez is konjunkcióval hozzákapcsolják az invariánst. Objektumorientált rendszerekben az enkapszuláció elősegítése végett vannak a típusoknak publikus és privát műveletei. A privát műveletek hívása előtt és után, mivel azok nem elérhetőek a kívülág számára, nem szokás megkövetelni az invariáns tulajdonság teljesülését. Alprogramokra az invariánsok megadása ezért sokszor az előfeltétel és az utófeltétel kiegészítésével történik.

2.2. Statikus analízis módszerei

Számos módszer létezik a statikus analízisre. Ezek közül a legfejlettebb megközelítés, amikor a forráskódot egy kanonikus szöveges formára hozzuk. Ekkor eltávolításra kerülnek a fölösleges fehér karakterek, valamint eltüntetésre kerülnek a szintaktikai cukrok. Ezután az így kapott reprezentáció gyakran tokenizálásra kerül. A tokenizált kódra pedig reguláris kifejezéseket lehet illeszteni. Ennek a módszernek az előnye az egyszerűsége, valamint hatékonysága. Hátránya azonban, hogy a forráskódban megjelenő implicit konstrukciókat (például implicit konverziók) nehéz kezelni. Emellett a típusinformációk sem állnak rendelkezésre az analízis során kényelmesen elérhető formában. A reguláris kifejezésekkel megírt szabályokat gyors

```

bool a, b, c, d;
/*...*/
if(a || b) { /*...*/ }

if(c || d) { /*...*/ }

```

2.2. ábra. Exponenciális futási idő szemléltetése

san és hatékonyan lehet ellenőrizni, azonban a szabályok környezetfüggő osztályát nem lehet leírni reguláris kifejezések segítségével. A CppCheck nevű eszköz ezt a megközelítést alkalmazza [8].

Fejlettebb módszer, ha felépítünk egy absztrakt szintaxisfát, amin elvégezzük a szemantikus analízist is. Ez nem egyszerű feladat, ezért gyakran egy fordítót használunk fel erre a célra. Ekkor a szintaxisfa bejárása közben lehet következtetéseket levonni. Itt a szabály íróinak számára rendelkezésére áll az összes implicit konstrukció, mivel a szintaxisfa több információt hordoz explicit módon a program szemantikájáról, mint egy szöveges reprezentáció. Amennyiben a szemantikus analízist is elvégeztük a szintaxisfán, a szabály írásakor a típusinformációk is könnyen elérhetőek, így még több információ áll rendelkezésére a szabályok íróinak. Ennek a módszernek a hátránya, hogy nehéz olyan szabályokat írni, amelyek az adatok áramlásával és transzformációjával, vagy a program végrehajtási útvonalaival kapcsolatosak.

Amennyiben például bizonyos függvények hívási sorrendjére is akarunk megkötést tenni, akkor „path sensitive” analízist érdemes végrehajtani a kódon, ahol a program vezérlésének a szerkezetét jellemző gráf bejárása közben vonunk le következtetéseket. A módszer egyik legnagyobb hátránya, hogy legrosszabb esetben exponenciális futási idejű. A (2.2) kódrészlet szerint, az első elágazásnál két útvonalat kell ellenőrizni. Azt amikor belépünk az első elágazás törzsébe és azt is amikor nem. A második elágazással azonban az ellenőrizendő útvonalak száma már négy, ugyanis a második elágazás mind a két esetét meg kell vizsgálni abban az esetben is mikor az első elágazás feltétele igaz volt és abban is, amikor nem. Sokszor a változók lehetséges értékeire is kíváncsiak vagyunk, ekkor szimbolikus végrehajtást végzünk. Ez a program végrehajtásának egyfajta szimulációja, ahol a változókat mint szimbolikus értékeket tartjuk számon. Például egy egész számot tartalmazó változóhoz hozzárendelésre kerülnek azok az intervallumok, amik tartalmazzák az összes lehetséges értéket, amit a változó felvehet a program adott pontján. A program végrehajtásának a szimulálása közben minden változóra, ami életben

van egy adott pontján a programnak, rendelkezünk az adott változóra érvényes megkötésekkel is (pl. $x \neq 0$). Ezzel a módszerrel vizsgált forráskód esetén még akkor is találhatunk hibát egy adott kódrészletben, ha annak a tesztlefedettsége teljes, hiszen a teljes lefedettség azt jelenti, hogy minden sora (vagy végrehajtási ága) az adott kódrészletnek végre lesz hajtva a tesztek során, viszont nem jelent semmit arra nézve, hogy a teljes állapotteret lefedtük-e az eseteinkkel.

Ebből kifolyólag egy olyan szimbolikus végrehajtást végző rendszer, ami egy adott típusú hibát ki tud szűrni, nagyobb hatékonysággal tudja ennek a hibának az előfordulásait megtalálni, mint amit tesztek írásával el lehet érni. Ugyanakkor, mivel nem lehetséges minden hibatípusra ellenőrzőszoftvert írni, a tesztelést nem váltja ki a statikus analízis. Ez az oka annak, hogy sok esetben egy kódbázist teszteléssel és statikus analízissel egyaránt vizsgálnak.

Számos egyéb módszer létezik, viszont a dolgozatban ismertetett eredmények megértéséhez nincsen szükség ezek tárgyalására.

2.3. Statikus analízis kiterjedése

Egyik jellemzője a statikus analízisnek a kiterjedtsége: mekkora egységet tud egyszerre vizsgálni a forráskódból úgy, hogy az adott egységről lehetőleg minden információ rendelkezésre álljon. Egy adott változó használatáról például akkor tud jól érvelni egy eszköz, ha ismeri a változó típusát, amihez viszont ismertnek kell lennie a változó deklarációjának a helyének. Amennyiben ez ismeretlen, az analízis semmit sem tud mondani a változó előfordulásáról.

Nagyon ritkán van értelme beszélni egy utasításról vagy egy kifejezésről mint egységről, mivel ezek önmagukban ritkán hordoznak elég információt ahhoz, hogy jelentősebb hibákat meg lehessen találni statikus módszerekkel. A legkisebb egység aminek a vizsgálatának jelentős gyakorlati haszna van, az a függvény vagy metódus. Egy függvény könnyen elemezhető, hiszen az állapottere is viszonylag könnyen meghatározható (amennyiben nem hivatkozik globális változókra) a lokális változói és paraméterei által. Ezek az információk a függvény kódjában mind jelen vannak. További könnyebbség, hogy a függvény működésére vonatkozó összes információ, ha a meghívott függvények szemantikájára nem vagyunk kíváncsiak, megtalálható egyetlen forrásfájlban. A globális változók ugyanakkor okozhatnak gondokat. Ezek a problémák viszonylag egyszerűen feloldhatóak azzal, ha egyszerre nem egy fájlt, hanem egy teljes fordítási egységet adunk bemenetként az analízist végző eszköznek. Azáltal, hogy egyetlen függvény elemzése történik meg egyszerre, és minden függvény vizsgálata izoláltan történik, már számos lehetséges hiba kiszűrhető:

például bizonyos erőforrások felszabadításának a hiánya (többek között a memória szivárgás). Abban az esetben, ha a foglalásért és a felszabadításért külön függvény felel, már nem lehetséges megtalálni a hibát ezzel a módszerrel.

Az ilyen problémák megtalálására interprocedurális analízist szokás alkalmazni. Ebben az esetben egy függvény vizsgálatakor, ha egy függvényhívással találkozunk, akkor azokkal a szimbolikus értékekkel, amikkel a hívott függvény aktuális paraméterei rendelkeznek elemzésre kell kerülnie a hívott függvénynek. Ez azt jelenti, hogy adott f függvény minden hívásakor, ahol a paraméterei olyan szimbolikus értékekkel rendelkeznek, amivel f még nem került kielemezésre, akkor f -et újra ki kell elemezni. Ez sajnos igen költséges lehet, viszont az elemzés bizonyos fázisait, mint például a program végrehajtási gráfjának a felépítését elég egyszer végrehajtani. Bizonyos eszközök fordított irányból próbálják megközelíteni a problémát: minden függvényhez megpróbálják előállítani azt a szimbolikus értékhalmozatot, amivel az adott függvényt meghívva problémákba ütközhetünk, és a függvényhívás helyén csupán a formális paraméterek gondot okozó szimbolikus értékeinek a halmazát hasonlítják össze az aktuális paraméterek szimbolikus értékeivel. Ez a megközelítés hatékonyabb, viszont sokszor lényegesen bonyolultabb implementációt eredményezhet. Az első módszer könnyen implementálható olyan módon, hogy a hívott függvény törzse kifejtésre kerül a hívás helyén, majd a kifejtés után úgy elemezzük tovább a kódot, mintha nem történt volna függvényhívás. Ezt hívjuk *inline-olásnak*. Az egyik problémája az inline-olásnak az, hogy például egy rekurzív függvény esetében gyakori, hogy a szimbolikus értékek alapján akár végtelenszer kellene kifejtetni a függvényt önmagába. Emiatt az ilyen módon működő eszközök gyakran heurisztikák alapján vágásokat végeznek, bizonyos mélységű függvénykifejtési-fa létrejötte után nem végeznek több kifejtést.

Sajnos egy függvény nem inline-olható minden esetben. Elképzelhető, hogy a függvény törzse egy másik fordítási egységben található. Ezen a ponton az analízis eszköznek nincs információja arról, hogy melyik fordítási egység is tartalmazza ezt pontosan, ezért mindegyiket végig kellene néznie, ehhez pedig mindegyiket elemeznie kellene. Mivel az iparban előforduló kódbázisok méreténél irreális elvárás lenne, hogy minden fordítási egység eszköz által felépített reprezentációja egyszerre beleférjen a memóriába, ezért mindig lesz olyan függvény, aminek a törzse nem áll rendelkezésre. Mivel a találgatás, hogy melyik fordítási egység tartalmazhatja a kívánt függvényt nagyon sok ideig tartana, ezért a legtöbb eszköz nem támogatja a fordítási egységek közötti analízist. Sajnálatos módon, ha egy függvénynek nem érhető el a definíciója, az számos problémát okoz a szimbolikus végrehajtás során. Ha egy ilyen függvény paraméterül kap egy referenciát egy olyan változóra, amire számos megkötést is-

merünk már, a függvény hívása után kénytelenek vagyunk érvényteleníteni ezeket a megkötéseket, hiszen a meghívott függvény törzse bármit csinálhatott az adott változó értékével. Valamennyivel javítani lehet a helyzeten, ha rendelkezünk egy listával, amit egy másik eszköz állított elő, ami tartalmazza, hogy melyik függvény melyik fordítási egységben található. Ekkor bizonyos vágási heurisztikák alapján maximum néhány fordítási egységet a memóriában tartva növelhetjük az analízis pontosságát.

Sajnos abban az esetben sem állna rendelkezésünkre minden függvény definíciója, ha végtelen sok memória állna rendelkezésünkre. Az ipari alkalmazásokban gyakori, hogy más fejlesztők által írt könyvtárakat használnak fel. Ezek a könyvtárak sokszor bináris formában kerülnek terjesztésre, tehát a forráskód, ami a könyvtárban lévő függvények törzseit tartalmazza nem elérhető. Amennyiben valamilyen módon bizonyos információ a felhasznált könyvtárakban lévő függvényekről is rendelkezésre áll, akkor beszélünk globális analízisről.

Egy lehetséges megoldás bizonyos függvénydefiníciók elérhetővé tételére egy *unity build* létrehozása. A *unity build* során a teljes kódbázis egyetlen fordítási egységként van kezelve. Sajnos ezáltal nagyban megnő a fordításhoz használandó memória mértéke. A fordítási egységeket általában lehet párhuzamosan, egymástól függetlenül fordítani. Ez a *unity build* esetében, abból adódóan, hogy csupán egyetlen fordítási egység létezik, nem lehetséges, így a párhuzamosságból is vesz a felhasználó. További problémát jelent, hogy nem egyszerű feladat egy ipari kódbázis esetén egy ilyen környezet létrehozása. A naív megközelítés az lenne, hogy egy forrásfájlba importáljuk be az összes többi, azonban sok nyelvben ez névütközésekhez vezethet, amiket fel kell oldani. Ezen felül a kód szemantikája is megváltozhat egy *unity build* során [10].

2.4. Modellek a statikus analízishez

Ahogy az előbbiekben is láttuk, az analízis pontosságán sokat lehet javítani azzal, ha a tényleges elemzés előtt információkat gyűjtünk össze a forráskódról, amit később az analízis során felhasználunk. Ezt szokás többmenetes analízisnek hívni. Ennek a legkézenfekvőbb módja, mikor kigyűjtjük, hogy mely függvények mely fordítási egységekben találhatóak. *Modellnek* hívjuk azt a struktúrált információegyüttest amit a végső analízis számára elérhetővé teszünk az előző menetekben. Minél több információt tárolunk el a modellben annál pontosabb eredményeket kaphatunk, azonban annál nagyobb lesz a modell mérete, és potenciálisan annál nagyobb lesz az analízis számítás- és memóriaigénye is, mivel egyre több lesz a feldolgozandó in-

```

int f[n]() { return 0; }
int f[n-1]() { return f[n](); }
int f[n-2]() { return f[n-1]() + f[n](); }
// ...
int f[1]() { return f[2]() + f[3]() + ... + f[n](); }

int buggy() { return 5/f[1](); }

```

2.3. ábra. Fixpont iteráció erejének szemléltetése

formáció. Az ilyen értelemben vett modelleket többek között azért vezették be, mert az összes fordítási egység precíz fordítók által is használt reprezentációja nem fér el egyszerre a memóriában. Ebből következik, hogy egy jól használható modell, ami a teljes analizált kódbázist írja le, szükségképpen kevesebb szemantikus információt tartalmaz, mint ami az adott kódbázisban el van kódolva. Unity build esetén is lehet létjogosultsága a modelleknek, mivel nem feltétlen rendelkezik a felhasználó akkora számítási kapacitással, amivel tetszőlegesen mély függvényhívási láncok ki-elemezhetőek. Egy mély hívási láncban vágási heurisztikákat végezhet az elemző, bizonyos mélység után az implementáció helyett használhatja a modellben található információkat.

Sajnos többmenetes analízissel sem lehet minden fordítási egységek között átívelő hibát megtalálni. A fő oka ennek, hogy csupán véges számú fordítási egység fér el egyszerre a memóriában. Vegyük a következő $n + 1$ függvényből álló példát (2.3), ahol a forráskódon belül a hasonló nevű függvényeket a szögletes zárójel segítségével indexelem. Az indexelés csupán a függvények neveinek az elkülönítésére szolgál, nincsen kitüntetett szemantikus jelentése. A példakód szemantikája a C nyelv szemantikájával megegyező. Az alábbi kódrészletben az f_n visszatérési értéke mindig 0. Hasonlóképpen az $f_{(n-1)}$ is a konstans 0 függvény. Mivel az $f_{(n-2)}$ az $f_{(n-1)}$ és a f_n visszatérési értékének az összegét adja vissza, ezért az $f_{(n-2)}$ visszatérési értéke is 0. Hasonló módon az f_i -t úgy definiáljuk, hogy a visszatérési értéke legyen $\sum_{k=i+1}^n f_k()$. Teljes indukcióval belátható, hogy f_i visszatérési értéke is 0 lesz.

Észrevehető, hogy a fenti forráskód végrehajtása során 0-val való osztás történik. Tegyük fel, hogy egyszerre egy fordítási egység reprezentációja fér el a memóriában és minden fenti függvény külön fordítási egységben van. A hiba ekkor $n + 1$ fordítási egységen ível át. Mivel a modellt előállító meneteknek nincs különösebb információja arról, hogy milyen sorrendben érdemes feldolgozni a forrásfájlokat, ezért előfordulhat, hogy f_1 -től f_n felé haladva olvassa be a fordítási egységeket. Az első

menetben, mikor f_1 -et elemezzük, a visszatérési értékéről semmit nem tudunk mondani, hiszen nem ismerjük a meghívott függvények implementációját. Az első menet végén tudjuk, hogy f_n az 0-t fog visszaadni. A második menet végén már tudjuk, hogy az $f_{(n-1)}$ visszatérési értéke is 0 lesz, hiszen ekkor már ismerjük az f_n visszatérési értékét. A harmadik menetben hasonló módon meg lehet határozni $f_{(n-2)}$ értékét. Az $n + 1$ -edik menetben pedig megtalálhatjuk a hibát, hiszen ekkor már ismerni fogjuk az f_1 visszatérési értékét. Ez a példa jól mutatja, hogy nem tudunk olyan k korlátot mondani a menetek számára, amire ne lenne konstruálható egy olyan program, ami tartalmaz egy olyan hibát, ami nem található meg k menet után.

Egy általánosabb megoldás a fixpont iteráció, amikor nem adunk felső korlátot a menetek számára, hanem mindaddig ismételtetjük a meneteket, ameddig a modellen változást tapasztalunk, azaz a modell nincs fixpontban. Sajnos a gyakorlatban egy-egy menet futási ideje is nagyon hosszúra tud nyúlni, ezért ezt a módszert csak kisebb programok vagy modulok esetén érdemes használni. A legtöbb piacon megtalálható eszköz egy- vagy kétmenetes, és már ezekkel is számos programozói hibát fel tudnak fedni a forráskódban.

A modell építéskor felmerül az a kérdés is, miszerint mit tekintünk a forráskód belépési pontjának? Honnan kezdjen el futni a szimbolikus végrehajtás? Egy alkalmazás esetében ez egyértelmű lehet, azonban semmi garancia nincs arra, hogy a belépési pontot tartalmazó forrásfájltól kezdődik el a modell építése, mint ahogy arra sincs garancia, hogy minden olyan függvényhez, amit elemezni kezdünk, létezik a már elemzett függvényeken keresztüli hívási lánc a belépési ponttól. Másik probléma, hogy arra sincs garancia, hogy létezik egyértelmű belépési pont, hiszen lehetséges, hogy egy könyvtár forráskódja kerül elemzésre. Ezen esetek miatt bevett szokás minden globálisan elérhető függvényt belépési pontnak tekinteni, és mindegyikből elindítani a szimbolikus végrehajtást. Ennek a megközelítésnek a hátránya, hogy lehetetlenné teszi a nem használt függvények kiszűrését szimbolikus végrehajtás segítségével, valamint sok esetben egy adott függvény azért került többször elemzésre, mert több belépési pontból is elérhető egy hívási láncon keresztül és ezeken a láncokon eltérő szimbolikus értékekkel kerül meghívásra. False Positive eredményeket is kaphatunk azáltal, hogy egy függvényt belépési pontnak tekintünk. Lehetséges, hogy az adott függvényt csak olyan helyről hívják, ahol teljesül minden előfeltétele. Ha belépési pontként kezeljük, akkor nem mondhatunk semmit az előfeltételek teljesüléséről. Egy másik stratégia: véletlenszerűen választunk belépési pontokat azokból a függvényekből amit még nem érintett a szimbolikus végrehajtás, mindaddig ameddig létezik elemzetlen függvény.

3. fejezet

Programozási nyelvek

A programozók ideje értékes, ezért érdemes azt kreatív feladatok megoldására fordítani repetatív és automatizálható feladatok végzése helyett. Sok automatizálható feladat merül fel egy szoftver fejlesztése során. Lehetnek ezek refaktorálási feladatok, kódgenerálási lépések vagy akár bizonyos típusú hibák keresése. Ezeket a feladatokat a produktivitás érdekében sokszor külön alkalmazások végzik el.

Még régen, a programozók elsősorban egy editort használtak a fejlesztésre, addig manapság számos eszköz segítheti a munkájukat. Éppen ezért egy nyelv tervezésekor egyre fontosabb szempont az, hogy a szintaxis mennyire alkalmas statikus analízisre. Az újonnan megjelenő nyelvek fordítói, mint például a Go, Rust és D, gyakran alkalmasak dokumentációs megjegyzések exportálására, tesztlefedettség mérésre és automatikus kódmigrációra. Az automatikus kódmigráció lehetővé teszi azt, hogy visszafelé nem kompatibilis változtatásokat vigyenek végbe a nyelven anélkül, hogy problémát jelentene a meglévő kódbázisok átírása.

Azáltal, hogy a nyelv szintaxisa könnyen elemezhető, gyorsabban jelennek meg az adott nyelvhez refaktoráló eszközök, hamarabb készítenek hozzá jól használható fejlesztői környezetet. Emellett több hiba válik fordítási időben felfedezhetővé azáltal, hogy hatékonyabb elemzőszoftvereket lehet készíteni ezekhez a nyelvekhez. Mivel a fordítóknak ilyenkor az elemzési fázisuk gyors, több idő marad a generálandó kód optimalizációjára is.

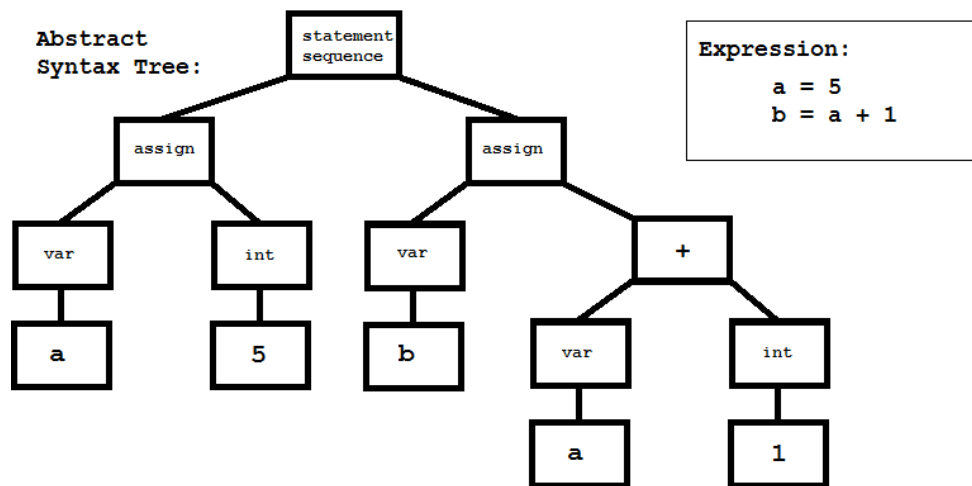
Néhány biztonságkritikus alkalmazás esetében a nyelv szintaxisát szinte teljes egészében alárendelik a statikus ellenőrző programnak, ezekben az esetekben az a cél, hogy az elemző számára minél könnyebben érthető legyen a kód, hogy az verifikálhassa. Másodlagos cél az emberek számára az olvashatóság. Ilyen nyelv az ADA-n alapuló SPARK is, ahol nyelvi eszközök állnak rendelkezésre az elő- és utófeltételek, valamint az invariánsok megadására. Az így megadott szerződéseket fordítási időben tudja verifikálni a fordító [25]. Más esetekben, amikor a nyelv szintá-

xisa már rögzített és utólag nem lehet elemzést elősegítő módosításokat végezni rajta, annotációk segítségével teheti a programozó a kódot elemzőbarátabbá. Minél több szemantikus információt tartalmaz egy kódrészlet könnyen feldolgozható formában, annál könnyebb dolga van az elemzőeszköznek. Jó példa erre a C++14 szabvány által bevezetett `[[deprecated]]` attribútum, amivel függvényeket lehet megjelölni, amiknek a használata már nem javasolt. Ezen annotáció alapján a fordító új fajta figyelmeztetéseket tud generálni.

Fontos még, hogy az adott nyelv hogyan valósítja meg a modularizációt. Abban az esetben, ha egy modult több modul is importál, a fordítási modellettől függ, hogy szükség van-e az importált modul többszöri elemzésére vagy sem. Amennyiben az importált modulokat minden importálás helyén elemezni kell, az jelentős teljesítményromlást jelent.

Minél több egy nyelvben a dinamikus kötés, annál nehezebb a statikus analízis, ugyanis annál kevesebb információ áll rendelkezésre fordítási időben. Általában nem lehet egyértelműen meghatározni futási információk nélkül, hogy egy dinamikus kötés hova fog kötni, ezért minden szóbjövő esetet számításba kell venni. Sokszor nem is elérhető a kötött kódrészlet az elemzőszoftver számára, hiszen lehetséges, hogy egy könyvtárat elemzünk, viszont a dinamikus kötés egy felhasználó által írt alprogramhoz fog kötni.

A szintaxison túl fontos szempont a nyelv elemezhetőségének szempontjából a típusrendszer is. Általában a statikusan típusozott nyelvek forráskódjában lévő szemantikus információ rengeteg segítséget nyújt a statikus elemzőszoftverek számára. Dinamikusan típusozott nyelvek forráskódjában kevesebb szemantikus információ található, ráadásul ezen nyelvek esetén gyakoribb a dinamikus kötés is. Minél erősebb a típusrendszer, annál több és összetettebb szemantikus információt lehet benne ábrázolni. A függőtípusos nyelvek esetén, mint amilyen például az Agda, a típusok részletes specifikációként tudnak működni, és amennyiben a fordító nem tudja belátni az implementációról, hogy az megfelel a specifikációnak, a programozó fordítási idejű hibát fog kapni [1]. Így, egy kellően kifejező típusrendszer esetén a fordító típusellenőrzője (ami valójában statikus ellenőrzéseket végez) tételbizonyító rendszerként is tud működni, ahol a típusok tekinthetők tételeknek és az implementációk bizonyításoknak. Ezt a jelenséget írja le a Curry–Howard izomorfizmus [20].



3.1. ábra. Absztrakt szintaxisfa

3.1. Elemzőkkel kapcsolatos fogalmak

Nehéz egy adott nyelvet elemezhetőség szempontjából vizsgálni anélkül, hogy az elemzőszoftverek implementációjával kapcsolatos alapfogalmakkal tisztában lennénk. Ennek a problémának a feloldása érdekében bevezetnek néhány fogalmat.

Az *absztrakt szintaxisfa* (abstract syntax tree, AST) egy adott nyelven írt forráskód ábrázolásmódja (3.1 ábra). A fában minden csúcs (node) megfeleltethető egy konstrukciónak a forráskódban. Ezek a konstrukciók nem feltétlen jelennek meg a kódban explicit módon, erre jó példa az implicit konverzió. Az absztrakt jelző arra utal, hogy a reprezentációban nem található meg a szintaxis minden részlete, például egy kifejezés esetén a műveletek elvégzési sorrendjét a zárójelek is meghatározhatják, amik viszont nem jelennek meg a szintaxisfában csúcsként, mivel ez az információ a fa felépítésében van kódolva. Az absztrakt szintaxisfát a továbbiakban gyakran csupán szintaxisfaként fogom említeni. A gyakorlatban a fordítók sokszor nem szintaxisfát, hanem szintaxis gráfot hoznak létre teljesítménynövelési céllal. Ennek az az oka, hogy néhány él behúzásával olyan információk válnak elérhetővé az adott csúcsból, amit egyébként csak a szintaxisfa újbóli bejárásával lehetne megszerezni. Ettől függetlenül a fastruktúra megmarad, ugyanis a gráf bejárásakor az ilyen célból behúzott élek a legtöbb esetben figyelmen kívül lesznek hagyva.

A *szemantikus elemzés* során a fordító leellenőrzi, hogy a forráskód a szintaktikus szabályokon túl megfelel-e a nyelv típusrendszerének [2]. Gyakran a szemantikus elemzés részét képezi a *szimbólumtábla* felépítése, ami a felhasználó által bevezetett azonosítókra tartalmaz információkat. A szimbólumtábla része, hogy milyen szerepet tölt be egy azonosító, például típust vagy változót jelöl-e, változó esetén tartal-

```

S (x);
T * y;
func((R) * z);

```

3.2. ábra. Környezetfüggő C nyelvtan

mazza az azonosító típusát. Sokszor a szimbólumtáblában tároljuk le a hátókörről vonatkozó információkat is [2].

A *Control Flow Graph* (CFG) egy, a szintaxisfától különböző reprezentációja a forráskódnak. A gráf csúcsai az úgynevezett *Basic Block*-ok, amik olyan kódtöredékek, amik mindig szekvenciálisan lesznek végrehajtva. A gráf irányított élei a *Basic Block*-ok közötti ugróutasításoknak felelnek meg. Ezáltal a CFG-ben egy út a programon belül egy lehetséges végrehajtási útvonalnak fog megfelelni. Ez a reprezentáció különösen hasznos statikus analízishez és kódoptimalizációhoz.

3.2. A C++ programozási nyelv

A C++ [15] az egyik legelterjedtebb programozási nyelv az iparban. A népszerűségének az egyik oka, hogy alkalmas hatékony programok írására. Amellett, hogy a programozó számára nagyfokú hozzáférést enged a hardverhez, biztosít magasszintű absztrakciós lehetőségeket is. Annak viszont, hogy a programozónak alacsony szinten is lehetősége van programozni, hátránya is van: több a hibalehetőség. Ebből kifolyólag nagy C++ kódbázisok esetén még fontosabb a statikus analízis használata, mint egyéb magasabb szintű nyelvek esetén [3, 16].

Sajnos a C++ nem tartozik a könnyen elemezhető nyelvek közé. Ennek egyik oka az, hogy a nyelv magas kompatibilitását a C-vel meg akarták őrizni. Már a C nyelv nyelvtana sem környezetfüggetlen, ezzel megnehezíti a nyelv elemzését. A környezetfüggőséget jól szemlélteti a fenti kódrészlet (3.2). Amennyiben az első sorban *S* egy típus, akkor a sor az *x* változót definiálja. Ha *S* nem típus, akkor az első sor egy függvényhívás ahol *x* egy paraméter. A második sorban ha *T* egy típus, akkor a sor egy *T*-re mutató mutatót definiál *y* névvel. Ha *T* nem típus, hanem változó, akkor a második sor egy szorzás, aminek az eredményét figyelmen kívül hagyjuk. A harmadik sorban ha *R* egy típus, akkor *z* dereferálásra kerül, majd átkonvertáljuk *R* típusra. Ha *R* változó, akkor a szorzat eredménye kerül átadásra a *func* függvénynek. Ezek a példák mutatják, hogy szimbólumtábla nélkül nem lehet egyértelműen felépíteni a szintaxisfát, tehát a szemantikus és a szintaktikus elemzést nem lehet szétválasztani. Amennyiben az egyik szimbólum szerepét nem sikerül jól meghatározni a kódelemző

```
typedef int AA;
AA x = 5;
int main()
{
    AA AA = 1;
    int BB = AA * 2;
}
```

3.3. ábra. Környezetfüggő szimbólumtábla a C nyelvben

eszköznek, abban az esetben már az épített szintaxisfa is hibás lehet. Sajnos még a szimbólumtábla felépítése és olvasása sem triviális. Ahogy a (3.3) kódrészletben is látszik, az `AA` név a program különböző hatóköreiben különböző jelentéssel bír. Globális hatókörben a deklaráció után mint típus van jelen, még a `main` lokális hatókörében mint típus és változónév.

A C++ nyelv esetében még bonyolultabbá válik a forráskódok elemzése. A *template*-ek bevezetésével például olyan összetetté vált a nyelv nyelvtana, hogy be kellett vezetni olyan nyelvi elemeket, amellyel a többértelműséget a programozó explicit módon oldja fel. Ahogy a (3.4) kódrészletben látszik, a fordító nem tudja eldönteni a `common_type` sablonról, hogy a `type` az egy statikus adattag vagy egy típus, ugyanis ez az információ függ a P1 és P2 template paramétereiről. Mivel a lenti függvénysablon értelmezésekor a fordító még nem tudhatja milyen típus kerül majd P1 és P2 paraméterek helyére, a programozónak explicit módon kellett a `typename` kulcsszó `common_type` elé írásával feloldania a többértelműséget. Egy template nem hordoz elég információt a teljes szemantikus analízishez, például a (3.4) kódrészletben a fordító a típusparaméterek hiányában nem tudja eldönteni, hogy a függvényparaméterekre definiált-e a megfelelő relációs operátor. Éppen ezért, ha C++ kódot szeretnénk statikusan elemezni, akkor érdemes a template-ek kifejtését elvégezni az elemzés előtt. Sajnos a kifejtések elvégzése jelentős számítási kapacitást vesz igénybe ezáltal lassítva a fordítók és elemzőeszközök működését.

Az újabb C nyelvi szabványokban már az előfeldolgozást sem lehet végrehajtani egy jól felépített szimbólumtábla nélkül. Egy jó példa erre a gcc kiterjesztésként jelen lévő `__FUNCTION__` makró, ami minden esetben annak a függvénynek a nevére fejődik ki, amelyikben a makró le lett írva. Ahhoz, hogy az előfeldolgozó tudja, hogy melyik függvényben tart a feldolgozás, azon túl, hogy a függvény nevének szerepelnie kell már a szimbólumtáblában, arról is információval kell rendelkeznie, hogy pontosan milyen szintaktikus egységen belül van leírva a makró. A C99 [23]

```

template <typename P1, typename P2>
typename std::common_type<P1, P2>::type
max (P1 param1, P2 param2) {
    return param1 < param2 ? param2 : param1;
}

```

3.4. ábra. Környezetfüggés a C++-ban

```

String EvaluateSalaryAndReturnName( Employee e )
{
    if( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout << e.First() << " " << e.Last()
            << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}

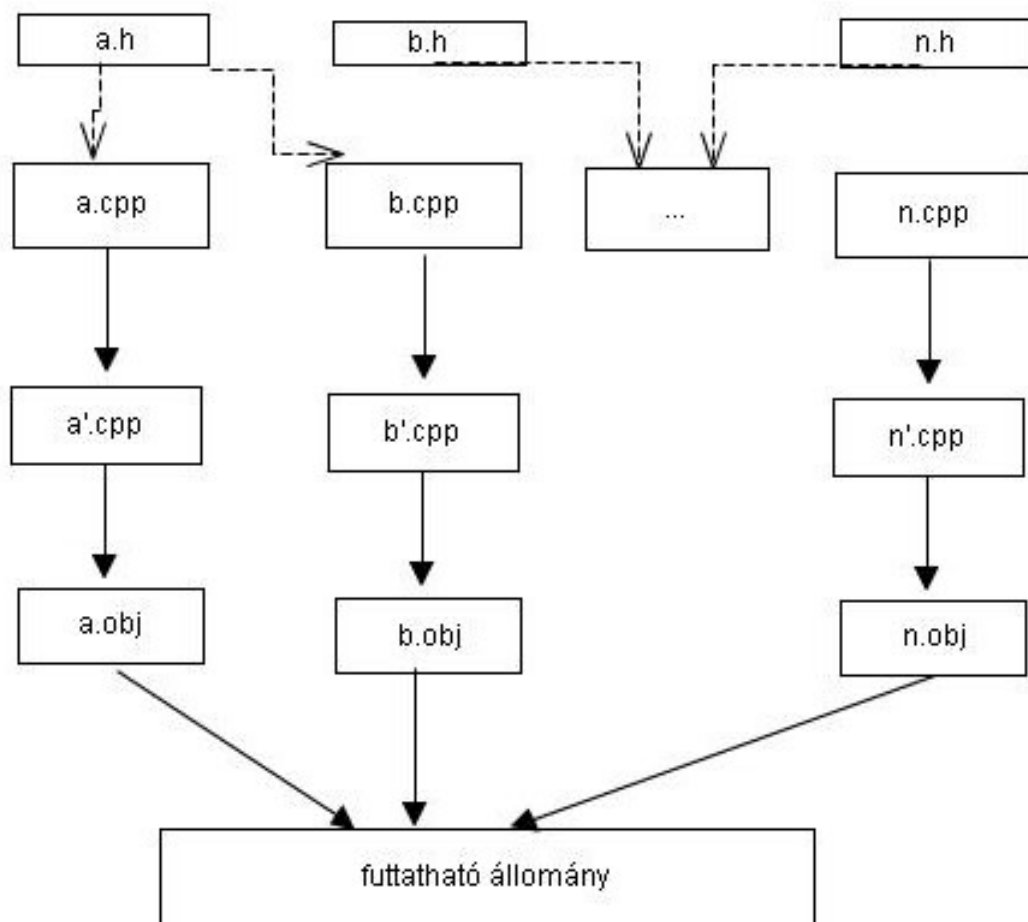
```

3.5. ábra. Exponenciális robbanás a végrehajtási útvonalak számában

szabvány éppen ezért vezette be az előre definiált azonosító fogalmát és egyetlen ilyen azonosítóként a `__func__`-ot, amit így nem kell a preprocesszornak feldolgozni.

A szimbolikus végrehajtás egyik legnagyobb ellensége a kivételkezelés, ami a C++-ban egy igen gyakran használt módszer. A (3.5) ábrán lévő példakód esetén, amennyiben nincsenek engedélyezve kivételek, elég három végrehajtási útvonalat végignézni. Abban az esetben, ha engedélyezve vannak, akkor már huszonháromra rúg az ellenőrizendő végrehajtási útvonalak száma. A nagy szám abból adódik, hogy C++-ban nem csupán a metódusok dobhatnak kivételt, hanem minden művelet, ami memória allokációval jár, valamint a túlterhelt operátorok is. Tovább ront a helyzeten, hogy a keletkezett kivétel elkapásakor a lehetséges catch ágak számával arányosan tovább nő a vizsgálandó esetek száma.

Sajnos a C++ nyelv fordítási modellje sem előnyös statikus analízis szempontjából. Egy tipikus fordítási folyamat látható a (3.6) ábrán. Egy fordítási egységnek nevezünk egy olyan egységet, amit a fordító egyben dolgoz fel. A C++ esetében egy fordítási egység az a fordítandó C++ fájl és az általa importált (include-olt) header állományok tranzitív lezártját jelenti. A probléma forrása, hogy általában egy header állomány több fordítási egységnek is a része, ezért az ilyen állományok



3.6. ábra. A C++ fordítási modellje

tartalma több alkalommal is fordításra kerül. Ez jelentős többletmunkát jelent a statikus ellenőrző eszközök részére is. Bizonyos esetekben lehetősége adódhat az eszköznek egy header állomány elemzése utáni belső reprezentációját szerializálni egy másik állományba és ezt a szerializált állományt használni a továbbiakban a header állomány helyett. Ezzel sok többletmunka megspórolható, viszont a megvalósítás nem egyszerű feladat. Egyrészt különböző fordítási egységekben különböző paraméterekkel kerülhet egy header fordításra, ami befolyásolhatja a headerben található kód szemantikáját. Továbbá a headerek tartalmazhatnak template definíciókat, amik más és más fordítási egységek esetén más és más paraméterekkel kerülnek kifejtésre.

Az előbbiekben leírt okok nem könnyítik meg a statikus elemzőszoftverek fejlesztését. A modern C++ kódbázisokban gyakori a template metaprogramok használata, amelyeket a fordító úgy hajt végre, hogy a nyelv típusrendszere által meghatározott szabályrendszer alapján elvégzi a template példányosításokat. Egy a programot pontosan ábrázoló nyelvi elemző összeállítása olyan nehézségű feladat, hogy gyakran a statikus elemzőszoftverek a fordítók által generált szintaxisfát várják bemenetként. A szintaxisfa továbbra is tükrözi a nyelv összetettségét, de a többértelműségek már fel vannak oldva, valamint mivel a szintaxisfa építésének a feltétele a szimbólumtábla építése, azért az is rendelkezésre áll. A másik nagy előny, hogy a template-ek példányosítása már megtörtént, így ezek az eszközök gyakran a példányosított kódot elemzik a példányosítandó template helyett, mivel a példányosítás helyén több szemantikus információ áll rendelkezésre a típusparaméterek ismerete következtében. A szintaxisfa azonban nem csak a példányosításokat, hanem az eredeti template-eket is tartalmazza, így egy ilyen eszköz akár a metaprogramokkal kapcsolatos hibákat is detektálhatja.

4. fejezet

Clang és a Static Analyzer

A *Clang* egy nyílt forráskódú fordító, ami az *LLVM* projektre épül [9]. Ez egy gyorsan fejlődő projekt, amit számos nagy cég támogat, többek között a *Google* és az *Apple*. Emellett egyre népszerűbbé is válik a fordítók körében, például a Microsoft által fejlesztett Visual Studio 2015 a Clang-et is szállítani fogja, mint alternatív fordítót a Microsoft saját fordítója mellett [24]. A Clang fordító modulárisan épül fel, ezáltal nem csupán egy fordító, hanem egy teljes keretrendszer a C/C++/Objective-C nyelveken működő fejlesztői eszközök készítéséhez. Az API-ja jól dokumentált és a széleskörű felhasználóbázisából adódóan tesztelt és kiforrott. A korábbi tapasztalataim szerint egy kényelmes eszköznek tekinthetjük a Clang-et [5, 6, 7].

A Clang fordító része a *Static Analyzer*, ami a fordító által generált szintaxisfát és CFG-t felhasználva képes szimbolikus végrehajtást végezni a kódbázison. Mivel a szimbolikus végrehajtás tipikusan jelentősen hosszabb ideig tart, mint a fordítási procedúra, azért a Static Analyzer futása nem része a fordítási folyamatnak, a fordítót speciális paraméterezéssel kell meghívni, ha a fejlesztő futtatni akarja a Static Analyzert.

4.1. Szimbolikus végrehajtás

A Static Analyzer két nagy komponensből áll. Az egyik komponens a mag, ami a szimbolikus végrehajtást végzi, a másik nagy komponenst pedig a *checker*-ek alkotják. A checker-ek tartalmazzák azokat a szabályokat, amiknek a program meg kell, hogy feleljen. Ezek passzív résztvevőként vannak jelen a szimbolikus végrehajtás futása során, tehát a végrehajtás módjára nem lehetnek hatással. A checker-ek olyan eseményekre iratkoznak fel, mint például egy függvény meghívása, egy változó megsemmisülése (hatókörből való kilépéskor) vagy értékadás. A mag szimbolikus végrehajtás közben eltárolja a feltételekben lévő megkötéseket a *Const*-

raint Manager-ben. Megpróbálja bejárni az összes útvonalat, és a szimbolikus értékek segítségével ábrázolja a program állapotterét. Amikor a végrehajtás közben olyan eseményhez ér, amire van feliratkozó, akkor meghívja a feliratkozott checker-ek megfelelő függvényeit. Ekkor a checker-ek a program szimbolikus állapotteréhez és a Constraint Manager-hez hozzáférve megpróbálják megállapítani, hogy történhet-e hiba a program ezen pontján, vagy többletinformációt tárolnak el a program állapotterében a későbbi vizsgálatok megkönnyítésének a céljából.

A mag által számontartott állapotter három részből áll. Az *Environment* tartalmazza a megfeleltetést a forráskódban lévő kifejezések és az állapotterben lévő szimbolikus értékek között. A *Store* tartalmazza a megfeleltetést a szimbolikus memóriarégiók és a szimbolikus értékek között. A szimbolikus memóriakezelésről a későbbiekben részletesebben lesz szó. A *GenericDataMap* segítségével pedig a checker-ek tárolhatják el saját állapotaikat és rendelkeznek azokat szimbolikus értékekhez.

A szimbolikus végrehajtsához nem elég egy CFG. Ha a program végrehajtást szimulálni akarjuk, nem elég az az információ, hogy hol tart éppen a program végrehajtása, szükség van arra az információra is, hogy milyen állapotban van éppen a program. Ez utóbbi információt nem tartalmazza a CFG, ezért a Static Analyzer egy úgynevezett *Exploded Graph*-ot épít fel [18]. A gráf felépítéséhez felhasználja a CFG-t. A programok állapottere nagyon sok komponensű lehet, ezért az Exploded Graph nagyon nagyra tud nőni már kis függvények esetében is. Az így kapott gráf bejárása akár végtelen hosszú ideig is tarthat, hiszen végtelen ciklusok vagy rekurziók is lehetnek a forráskódban. Az analízis terminálása érdekében a gráf bejárásakor vágásokat végez a mag. A vágások következtében elfajuló esetekben előfordulhat, hogy a program bizonyos részeire nem futott le az analízis. Ennek ellenére bizonyos heurisztikák segítségével hasznos eredményt lehet elérni.

A Static Analyzer végrehajtási modellje támogatja az interprocedurális elemzést. Ennek eléréséhez a függvény hívásának a helyére a mag be fogja helyettesíteni a hívott függvényt, amennyiben annak a definíciója elérhető az éppen elemzett fordítási egységben. A behelyettesítés hátránya, hogy az *Exploded Graph* felépítésre kerül a függvény minden meghívásának a helyén, hiszen a program állapottere potenciálisan különböző minden híváskor. Ahogy a (4.1) ábrán látható kód szemlélteti, ha az f függvény másik fordítási egységben található és referencia szerint várja a paraméterét, akkor bár x -ről az első h hívásnál tudtuk, hogy negatív, de a második h hívásnál már elvesztettük ezt az információt, mivel f tetszőleges transzformációt végezhetett az értéken. Ilyen esetben azt mondjuk, hogy az x invalidálódik. Hasonlóan ha egy osztály egy példányán meghívunk egy olyan metódust, ami nem

```

void h(int x) { /* ... */ }
void f(int &x);

void g(int x) {
    if (x < 0) {
        h(x);
        f(x);
        h(x);
    }
}

```

4.1. ábra. Információvesztés

`const`, és az adott metódus implementációja másik fordítási egységben található, mint amit elemzünk, akkor a példányhoz tartozó összes tagváltozó invalidálódni fog. Jelenleg az ilyesfajta információvesztés a legnagyobb technikai korlátja a checker-eknek.

A dolgozat írásakor egyszerre csupán egy fordítási egység elemzését támogatja a Clang.

4.2. Szimbolikus memóriakezelés

A C és C++ nyelvek hierarchikus memóriamoddellel rendelkeznek. A nyelvben van rekord adatszerkezet, melynek a tagjai maguk is lehetnek rekordok. Amennyiben egy fordítási egységen kívüli függvényhívás miatt egy rekordhoz tartozó szimbolikus érték invalidálódik, abban az esetben minden olyan szimbolikus értéket is invalidálni kell, amelyek ennek a rekordnak közvetett vagy közvetlen módon a tagjai voltak. Tovább bonyolítják a helyzetet a mutatók, ugyanis amennyiben ennek a rekordnak van olyan tagja, ami egy mutató, akkor a mutató által mutatott szimbolikus értéket is invalidálni kell, hiszen a hívott függvény potenciálisan ehhez is hozzáférhetett. A Static Analyzer szimbolikus memóriakezelés segítségével oldja meg ezeket a problémákat [17].

A mutatók maguk is szimbolikus értékek, és a szimbolikus érték az lehet szimbolikus memóriarégió is. Ezek a régiók hierarchikusan épülnek fel. Amennyiben egy szülőrégiót invalidálni kell, abban az esetben minden gyereket is. Egy rekordhoz, valamint minden tagjához tartozik külön szimbolikus memóriarégió. Azokhoz a tagokhoz, amik nem rekordok, szimbolikus érték is tartozik. Amennyiben az adott tag egy

mutató, akkor annak a szimbolikus értéke valójában egy szimbolikus memóriarégió. A hierarchikus memóriamodell nem csak az invalidálásnál fontos. Például amennyiben egy rekordot felülírunk értékadás operátorral, abban az esetben nem csupán a rekordot, hanem annak minden gyerekrégiójához tartozó szimbolikus értéket is frissíteni kell.

Mivel minden checker a saját maga által számontartott információkon túl azt a tudást használja fel, amiket a Static Analyzer mag komponense tesz elérhetővé számára, ezért minden javítás, ami a mag komponenst érinti, potenciálisan növeli a checkerek által megtalálható hibák számát.

5. fejezet

API modellezés

Az előző fejezetben bemutatott információvesztés, amit a fordítási egységen kívüli függvények hívása okozhat, sok esetben ellehetetlenítheti a statikus analízist. A gyakran használt, de jelentős információvesztést okozó függvények törzsét a Clang Static Analyzer a *BodyFarm* osztály egy példányában tárolja. Azáltal, hogy a gyakran használt API-k (Application Programming Interface) bizonyos függvényeinek a törzsei bekerülnek ebbe az osztályba, a Static Analyzer tudja használni a *BodyFarm* által szolgáltatott függvénytörzset a behelyettesítéskor, ezáltal elkerüli az információvesztés problémáját. A dolgozat hátralévő részében az ilyen módon behelyettesítésre kerülő kódrészletek összességét fogom modellnek hívni. (Nem lehet zárt módon megfogalmazni, hogy pontosan mi tartozhat egy modellhez.)

Az a probléma a jelenlegi *BodyFarm* implementációval, hogy a függvények forráskódja helyett a függvények szintaxisfáját tartalmazza. Egy új függvény felvétele a *BodyFarm*-ba azzal jár, hogy a fejlesztőnek kézzel fel kell építenie a függvénynek megfelelő szintaxisfát a megfelelő segédosztályok segítségével. Az 5.1 ábrán látható függvény szintaxisfájának a felépítését egy 70 soros függvény végzi. Hiába csupán három sor az eltárolt függvény, ahhoz, hogy a Static Analyzer megta-

```
void dispatch_once(dispatch_once_t *predicate ,
                   dispatch_block_t block) {
    if (!*predicate) {
        *predicate = 1;
        block();
    }
}
```

5.1. ábra. *BodyFarm* példa

nulja behelyettesíteni, több mint 70 sorra volt szükség. Ebből is látszik, hogy ez a módszer nem skálázódik jól. Amellett, hogy sok munkával jár új függvények felvétele a BodyFarm-ba, sok fordítóspecifikus ismeret is szükséges hozzá, például a Clang szintaxisfájának az ismerete. További probléma, hogy a szintaxisfa szerkezete verzióról verzióra változik, ezért ezeket a függvénytörzseket karbantartani is jelentős feladat.

Amellett, hogy a BodyFarm fejlesztés és karbantartás szempontjából előnytelen, további problémát jelent az, hogy az így létrehozott függvénytörzsekhez nem tartoznak megfelelő forráskódbeli pozíciók. Így abban az esetben, ha egy így behelyettesített függvényben talál hibát a Static Analyzer, nem lesz képes azt a felhasználó felé jelenteni, hiszen nem rendelkezik a forráskóddal sem, ami alapján a hibaüzenetet legenerálhatná.

Egy alternatíva lehetne, hogy a BodyFarm csupán forráskódot tárol, és azt lusta módon fordítja le akkor, amikor a függvénytörzset be kellene helyettesíteni a hívás helyére. Ezzel az a probléma, hogy az 5.1 ábrán látható kód önmagában nem értelmes, szükség van az elemzéséhez a megfelelő típusoknak a definíciójára. Ezeknek a típusoknak a definícióit csak úgy lehet elérni, ha a megfelelő header állományokat include-oljuk. A probléma forrása, hogy a behelyettesítés előtt ez rengeteg header állomány megnyitását és elemzését eredményezheti. Nem egyszerű meghatározni a header állományok helyét a fájlrendszerben, valamint a fordításhoz szükséges paramétereket sem. Amennyiben készen van az elemzés, akkor az elemzett függvény törzsében lévő típusokat még meg kell feleltetni valamilyen módon a hívás helyén jelen lévő típusokkal. A sok fájlművelet és a header állományok redundáns elemzése, valamint a típusok közti megfeleltetés nagy teljesítményromlást okozna az amúgy sem túl gyors szimbolikus végrehajtásban.

A megfelelő header állományok útvonalának meghatározásában az okoz gondot, hogy az is legális, ha a felhasználó például keresési útvonalnak a `/A/B/C` útvonalat adja meg, és `#include "header.h"` szerepel a forrásfájlban, és az is legális, ha `/A/B` útvonal szerepel a keresési útvonalak között és `#include "C/header.h"` szerepel a fájlokban. Számos esetben nehéz úgy megadni a behelyettesítendő kódot, hogy mindig igazodjon a felhasználó által követett konvenciókhoz. A fordítási paraméterek meghatározásával az a probléma, hogy a header állományokat nem elég azokkal az opciókkal fordítani, amikkel az éppen elemzett fordítási egységet fordítottuk, mivel az állományokban található kód szemantikájára hatással lehetnek azok a makró definíciók, amik a header állományok include-olása előtt a forráskódban találhatóak. Így ezeket is számításba kell venni a fordítási parancs összeállításakor.

5.1. Gyorsan elemezhető modellek

Az én megoldásom a fenti problémákra a gyorsan elemezhető modellek bevezetése. Egy gyorsan elemezhető modell gyakorlatilag szabványos C/C++ kód, ami az 5.1 ábrán is látható. Mint láttuk, önmagában véve a fenti kód elemezhetetlen. A gyorsan elemezhető modellek esetében a modell akkor válik elemezhetővé, amikor behelyettesítésre kerül. Abban az esetben, ha egy függvény nem szerepel az adott fordítási egységben, de tartozik hozzá modell, akkor a fenti kód kerül behelyettesítésre egy trükk segítségével. A trükk abban rejlik, hogy a modellt nem önmagában próbálja meg a fordító elemezni, hanem a meghívás helyén jelen lévő típuskörnyezetet használja fel a célra. Ebből kifolyólag nem kell redundáns módon header állományokat elemezni, nem kell azoknak a helyét megkeresni, valamint nem kell két külön típuskörnyezet között megfeleltetést létesíteni. Így egy ilyen módon megírt és használt modell elemzése várhatóan nagyon kevés időt vesz igénybe. A gyorsan elemezhető elnevezés arra utal, hogy nem szükséges a header állományok feldolgozásával időt tölteni, aminek nagyon jelentős hatása van az elemzés idejére [10].

Mivel a szintaxisfát a fordító építi fel, ezért nincs szükség annak a kézzel való felépítésére és karbantartására, mint a hagyományos *BodyFarm*-ba égetett függvények esetében. Külön jó, hogy az elemzést egy fordítási egységen belül elég egyszer elvégezni, és az így kapott szintaxisfát pedig memoizáció segítségével el lehet raktározni, hogy a további hívások helyére elemzés nélkül egyből behelyettesíthető legyen. A generált szintaxisfához tartozik forrásszöveg is, így amikor egy modellezett függvényben talál hibát a Static Analyzer, a modell forrásszövegét fel tudja használni a hibajelentés generálásához.

A módszer hátránya, hogy csupán azokat a típusokat lehet a modellként megírt függvényben felhasználni, amik garantáltan jelen vannak abban a típuskörnyezetben ahol a függvény meghívásra kerül. Ez azt jelenti, hogy nem minden esetben egyezhet meg a függvény modellje az implementációval. Azonban nem is cél, hogy az implementációt modellként lehessen használni, hiszen ez esetben visszajutnánk ahhoz a problémához, hogy legtöbb esetben nem férne el minden függvény elemzésre alkalmas reprezentációja egyszerre a memóriában.

Egy példa modell látható az 5.2 ábrán. Ez egy olyan `sqrt` függvény modellje, ami helyben végez el egy gyökvonást ahelyett, hogy egy új értékkel térne vissza. Bár a gyakorlatban nem így szokott működni a gyökvonást végző függvény, szemléltetési célokra megfelel. Függvényen belül a C és C++ nyelvekben lehet függvényt deklarálni. A modell két függvény deklarálásával kezdődik. Amennyiben az `x` az kisebb mint 0, nem lehet helyben elvégezni a gyökvonást, hiszen az eredmény komplex

```

void sqrt(double& x) {
    void __math_sqrt(double& x);
    void __clang_warn(const char* c);

    if (x < 0) __clang_warn("Invalid_sqrt");
    else __math_sqrt(x);
}

```

5.2. ábra. Függvény modellek

```

int strcmp(char * str1, char * str2) {
    int __strcmp(char * str1, char * str2);
    int x = __strcmp(str1, str2);
    if (x <= 1 && x >= -1) return x;
    return 0;
}

```

5.3. ábra. Utófeltétel modellezés

lenne, `x` típusa pedig `double`. A függvénynek tehát az az előfeltétele, hogy az `x` nemnegatív. Az első függvénydeklarációnak a szignatúrája ugyanaz mint a modellezett függvényé. A második egy meghatározott nevű függvény, aminek a meghívása speciális jelentéssel bír a Clang számára. A függvény törzse ellenőrzi az előfeltétel teljesülését. Amennyiben nem teljesül, meghívja a speciális jelentésű függvényt, ami egy hibajelentés generálását fogja eredményezni a felhasználó számára. Amennyiben teljesül az előfeltétel, az ugyanolyan szignatúrájú függvény fog meghívódni, aminek nincsen implementációja, így a paraméter invalidálódását fogja eredményezni. Amennyiben nem invalidálnánk a paramétert, a Static Analyzer megtartaná a paraméter szimbolikus értékére vonatkozó megkötéseket.

Az 5.3 ábrán látható `strcmp` függvénymodellben az `__strcmp` segédfüggvény segítségével invalidáluk a paraméter által mutatott memóriarégiókat. A modellfüggvény szerepe itt az, hogy míg egy modelleztelen `strcmp` hívás esetén nincs semmilyen információja a szimbolikus végrehajtást végző motornak a visszatérési értékről, addig egy modellezett hívás esetén a modell kifejtése után ki tudja következtetni, hogy a visszaadott érték csak -1, 0 vagy 1 lehet. Ennek az információnak a segítségével pontosabban lehet elemezni a kódot.

Vegyük észre, hogy a modell valójában teljesen szabványos C++ kód. Alapvetően ez az ajánlott minta a modellek implementálásakor. Deklaráljuk az annotációs és

a paraméter invalidáló függvényeinket. Ellenőrizzük az előfeltételek teljesülését, ha nem teljesül meghívjuk az annotációs függvényeket, ha teljesül, akkor invalidáltatjuk azokat a paramétereket, amelyekre nézve nem tudunk a modellben semmit mondani. Amennyiben tudhatjuk a modellben, hogy bizonyos paramétereket hogyan kell transzformálni, ezt a transzformációt is elvégezhetjük, majd a transzformáció után ellenőrizhetjük a transzformált paraméterekre vonatkozó utófeltételek teljesülését. Összefoglalva a modell felépítésének a lépéseit:

- Írjunk meg egy – a modellezendő függvénnyel azonos szignatúrájú és nevű – függvényt az alábbi implementációval:
- Deklaráljuk a Clang annotációs függvényét, valamint egy – a modellezett függvénnyel megegyező szignatúrájú, de eltérő nevű – függvényt
- Ellenőrizzük az ismert előfeltételeket
- Abban az ágba, ahol nem teljesülnek az előfeltételek, hívjuk meg az annotációs függvényeket a hibajelentés kikényszerítéséhez
- Hívjuk meg az azonos szignatúrájú, de eltérő nevű függvényt, hogy invalidáljuk a megfelelő paramétereket
- A kimeneti paraméterekre és visszatérési értékekre tegyünk megszorításokat, ezáltal információt szolgáltatva az elemzőmotor számára (ebben a lépésben felhasználhatóak a paraméterek értékei is)

A fejezetben bemutatott módszernek az előnye, hogy a függvények elő- és utófeltételeinek az ellenőrzését ugyanazon a nyelven írjuk le, amelyikben a függvény, amit modellezni akarunk, íródott. Ezáltal – az Eiffel nyelvhez hasonlóan – a modell elkészítőjének nincs szüksége specifikációs vagy modellező nyelvek megtanulására, valamint nincs szüksége fordítóspecifikus ismeretekre sem. Bárki elkészítheti a saját könyvtárához a megfelelő modelleket, amik segítségével a Clang Static Analyzer a kódban ellenőrizheti ezen elő- és utófeltételek teljesülését. Ezek a modellfájlok a könyvtárral együtt terjeszthetők. Mivel a modellek nem kell, hogy tartalmazzanak bármilyen információt az implementáció részleteiről, ezért zárt forráskódú könyvtárak mellett is terjeszthetők. A modellekben lévő elő- és utófeltételek dokumentációs célt is szolgálhatnak.

5.2. Függvény modell szintetizációja

Jelenleg a modellek csak kézzel állíthatóak elő, így a dolgozat eredményeinek nincs kvantitatív dimenziója. A jövőben előnyös lehet az implementáció alapján

generálni ezeket. Bár teljesen precíz előfeltétel generálása hamar algoritmikusan eldönthetetlen problémára vezet, de a gyakorlatban számos fontos és gyakran megsértett előfeltételt ki lehet következtetni egy függvény implementációjából.

Jó példa erre, hogy amennyiben egy változóval osztunk, akkor az adott változó értéke nem lehet 0. Amennyiben a változó egy paraméter, akkor hozzá lehet venni az előfeltételhez, hogy az adott paraméter nem 0. Amennyiben nem függvényparaméterről van szó, visszafelé lejátszott szimbolikus végrehajtás segítségével néhány esetben meg lehet határozni, hogy a paraméterekre nézve ez milyen feltételeket jelent. Mivel nincs tökéletes komputeralgebra rendszer, ezért általános esetben ez a probléma megoldhatatlan. További gondot jelenthet, ha egy paraméter értéke függ egy globális változótól, mivel a globális változókról való érveléshez szinte a teljes forráskód teljes elemzése szükséges.

Sokkal gyakorlatiasabb probléma, hogy amennyiben egy paraméterként kapott pointer dereferálásra kerül, akkor az előfeltételhez hozzá kell venni, hogy az adott pointer nem null pointer. Hasonlóan, amennyiben egy paraméterrel egy puffert indexelünk, akkor az előfeltételhez hozzá kell venni, hogy az adott paraméternek a puffér méreténél kisebbnek és nemnegatívnak kell lennie.

Egy fokkal bonyolultabb a paraméterek közti általánosabb függés elemzése. Ilyen például, ha egy adott iterátorpárról kell eldönteni, hogy azonos konténerhez tartoznak-e.

Biztonságkritikus alkalmazásoknál fontos a *taint analízis*. Ekkor minden külső forrásból származó érték megjelölésre kerül, és mindaddig jelölve marad, amíg meg nem lettek vizsgálva, hogy a megfelelő értékkészletbe tartoznak-e. Az előfeltétel része lehet, hogy mely paramétereknek kell ellenőrzöttnek lenniük, az utófeltételnek pedig része lehet, hogy mely paramétereket kell megjelölni, hogy később vizsgálatra szorulnak.

Mivel minden checker más szempontból vizsgálja a kódot, ezért lehetővé kell tenni a checker-ek számára, hogy egy függvénytörzs alapján ők maguk is meghatározhassanak feltételeket, amik belekerülnek a generált előfeltételbe. Ez a modellgeneráló lépés egy a tényleges analízis előtt végrehajtott menetben valósítható meg.

Egy függvény előfeltétele függhet az általa meghívott függvények előfeltételeitől. Amennyiben fontos, hogy minél részletesebb előfeltétel-rendszert kapjunk meg, akkor egy adott függvény előfeltételének a meghatározásakor felhasználhatjuk az általa hívott függvények modelljeit. Ez nem feltétlen fogja egyből megadni a kívánt eredményt, fixpont iteráció segítségével lehet a modellt finomítani, ameddig van változás.

```
int getIndirectHalf(int * ptr) {
    return *ptr / 2;
}
```

5.4. ábra. Egyszerű modell

```
void foobar(){
    int x = new int(5);
    int p = 10 / getIndirectHalf(x);
    int q = getIndirectHalf(NULL);
    delete x;
}
```

5.5. ábra. Modellek szemléltetése

A C++ nyelv a jövőben lehetséges, hogy támogatni fogja a szerződés alapú programozást is [21]. Egy függvény modelljének a szintetizációjára használt eszközök alkalmasak lehetnek arra is, hogy bizonyos alacsony szintű szerződéseket statikusan generáljunk.

A modellek hasznosságának a szemléltetésére nézzük meg az 5.5 kódrészletet. Abban az esetben, ha a `getIndirectHalf` függvény nincs modellezve, akkor a Static Analyzer fog adni egy hibajelzést a `p` inicializálására, mivel egy olyan számmal osztunk, amit nem ellenőriztünk, hogy ne legyen 0. Ez False Pozitive, hiszen ebben az esetben soha nem fogunk 0-val osztani. A `q` inicializálására viszont nem fog hibajelzést adni, mivel nem ismeri a `getIndirectHalf` definícióját, így nem tudja, hogy a paraméterként átadott nullpointer dereferálásra fog kerülni. Jelen esetben a függvény egyszerűsége miatt a `getIndirectHalf` modellje egyezzen meg az implementációval (5.4). Abban az esetben, hogy a modellt elérhetővé tesszük, a Static Analyzer már nem fog hibajelzést adni a `p` inicializálására, mert a definíció és az elérhető értékek alapján ki tudja következtetni, hogy ebben az esetben a visszatérési érték nem lesz 0. A `q` inicializálására viszont fog hibajelzést adni, mivel a modellt ismerve látja, hogy egy nullpointer fog dereferálásra kerülni. Ez a példa mutatja, hogy modell segítségével mind False Pozitive hibát lehetséges elnyomni, mind False Negative hibát megtalálni. Ez jól szemlélteti a módszer erejét.

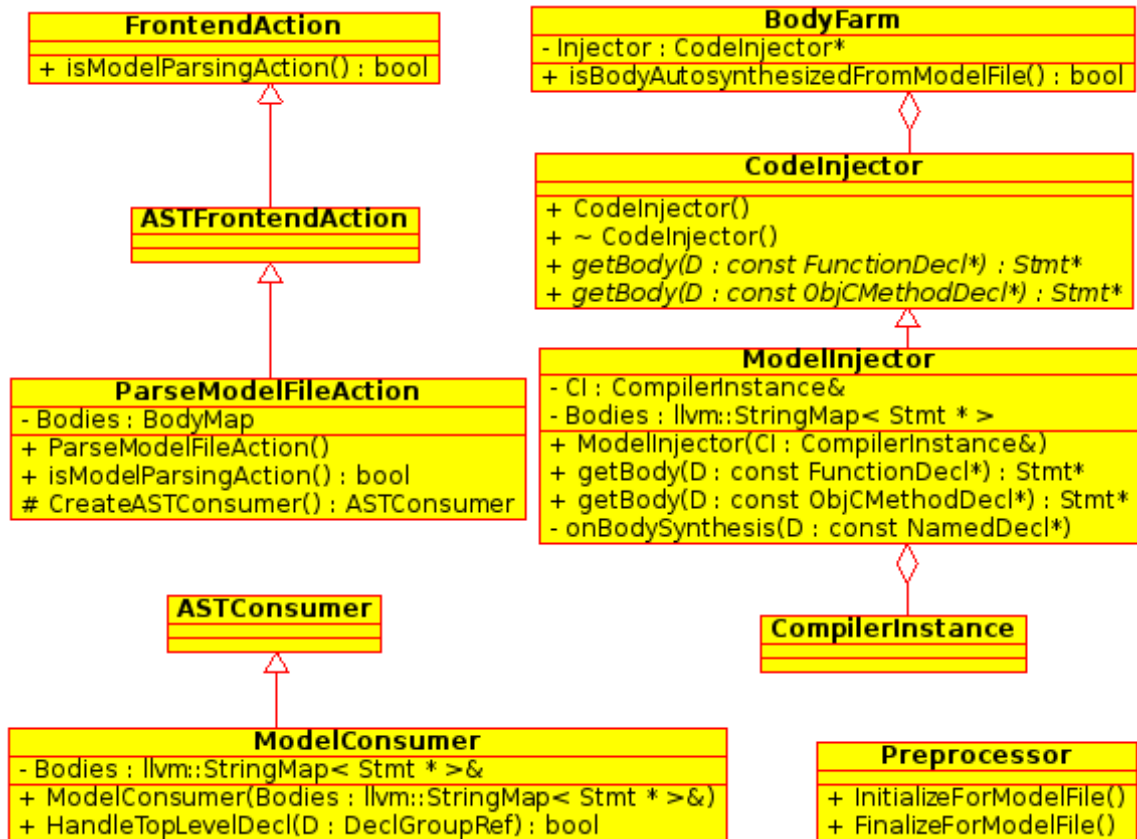
6. fejezet

Implementáció

Az API-k modellezéséért felelős kódot a Clang nyílt forráskódú fordítóban implementáltam egy GSoC (Google Summer of Code) projekt keretén belül. Az a projekt, amire GSoC résztvevőt kerestek, arról szólt volna, hogy bizonyos C és C++ standard könyvtárbeli függvényhez készítsünk a régi BodyFarm osztályba beégetett modell implementációkat. Az ötleten felbuzdulva javasoltam egy általánosabb megközelítést, amit a mentorom el is fogadott. Az én ötletem volt a modellek forráskódként való ábrázolása, a gyorsan elemezhető modellek bevezetése, valamint az elő- és utófeltételek C/C++-ban történő leírása is. A dolgozat írásakor még nem jelent meg a Clang 3.6-os verziója, de a projekt fejlesztési ágában már elérhető az implementáció [19]. Ez mutatja, hogy a projektet a Clang fejlesztői is sikeresnek könyvelték el.

A fordítót klasszikusan egy frontend és egy backend részre szokás felbontani. A frontend foglalkozik a forráskód elemzésével és a belső reprezentáció felépítésével, a backend pedig a kódgenerálással. A Clang fordítóban minden elemzéssel járó lépés egy `FrontendAction`. Ebből kifolyólag a modelleket tartalmazó fájlok (későbbiekben modellfájlok) elemzéséért is egy általam írt `FrontendAction` felel. Ennek a neve `ParseModelFileAction`. A `FrontendAction`-ök egyik feladata, hogy létrehozzanak egy `ASTConsumer`-t, ami feldolgozza az elemző által generált szintaxisfát. A `ParseModelFileAction` egy `ModelConsumer`-t fog létrehozni a modellek feldolgozására. Ez az osztály elemez egy modellfájlt, és a fájlban talált függvényekhez tartozó definíciókat eltárolja. Az implementáció osztálydiagramja a (6.1.) ábrán látható. Az osztálydiagramon csak azok a metódusok és mezők kerülnek megjelenítésre, amelyeket én adtam hozzá a Clang-hez.

Amikor a kód statikus analízis céljából kerül elemzésre, minden függvénydeklarációhoz tartozik egy `AnalysisDeclContext` példány. Ez kezeli a függvénydeklarációhoz tartozó elemeket, mint például a CFG-t vagy a



6.1. ábra. Implementáció: UML osztálydiagram

függvénytörzset. A BodyFarm is itt fejt ki a hatását, ugyanis amennyiben egy függvénydeklarációhoz nem létezik törzs az adott fordítási egységben, akkor a BodyFarm-tól próbálja meg elkérni a törzset az AnalysisDeclContext. Az én implementációmban, ha a BodyFarm nem talál a függvényhez beégetett függvénytörzset, akkor meghívja a CodeInjector osztály getBody metódusát a megfelelő deklarációval. A CodeInjector osztály egy általam bevezetett fogalom, ami segítségével külső forrásból érkező kódot lehet a Static Analyzer számára elérhetővé tenni. A CodeInjector altípusa a ModelInjector osztály, ami modellfájlokban található függvénytörzskeket szolgáltat a BodyFarm-nak. Amennyiben az analíziskor a modellezést bekapcsolja a felhasználó, a BodyFarm számára elérhetővé válik egy ModelInjector példány. A BodyFarm-nak azt is számon kell tartania, hogy egy függvénytörzs honnan származik. Egy beégetett AST-vel rendelkező függvénytörzshöz ugyanis nem tartoznak forráskódbeli pozíciók, így nem tud olyan hibákat jelenteni, amikhez szükség lenne ezekre a pozíciókra. Abban az esetben azonban, hogyha a törzs egy modellfájlból származik, akkor tartozik a törzshöz forráskód, ami a hibajelentés generálásánál felhasználható.

A modellfájlok elemzése lustán történik. Akkor kerül betöltésre egy modellfájl, mikor szükség van rá, és ha egy modellfájl tartalma betöltésre került, akkor a

fordítási egységben ha újra találkozik az analyzer egy megfelelő függvényhívással, akkor a már betöltött függvénytörzs kerül újra felhasználásra. A modellfájl betöltéséhez azonban szükség van az eredeti fájl szimbólumtáblájára. A Clang fordítóban a preprocessorhoz tartoznak lexerek és a szimbólumtábla is. Ebből kifolyólag az eredeti fordítási egységhez tartozó preprocessor kerül újrafelhasználásra. A típusinformációkat a Clang külön egy `ASTContext` példányban tárolja, így ez is újrafelhasználásra kerül. Egy törzs nélküli modellezett függvény első meghívásakor a `ParseModelFileAction` egy példányához kerül a végrehajtás egy újonnan létrehozott `CompilerInstance` segítségével. A `CompilerInstance` egy összefoglaló osztály, ami egy fordításhoz tartozó információkat és műveleteket foglalja magában. Az újonnan létrehozott `CompilerInstance` használja fel az eredeti fordítási egység fordításáért felelős `Preprocessor` és `ASTContext` objektumokat. Az elemzés kezdetekor azonban mind a `Preprocessor` és az `ASTContext` objektumokból új példány jönne létre, viszont amennyiben az elemzés tárgya egy modellfájl, ez az inicializációs lépés elmarad, hiszen ekkor a `CompilerInstance` már rendelkezik a megfelelő objektumokkal. Az elemzés befejeztével azt is biztosítani kell, hogy a `CompilerInstance` ne szabadítsa fel ezeket az objektumokat, hiszen ezekre még szükség lehet a későbbiekben, akár egy másik modellfájl betöltésekor.

Alapvetően a `Preprocessor` osztály megtervezésekor nem volt szempont az, hogy ugyanazon példánnyal több fordítási egység is elemezhető legyen. Ebből kifolyólag számos módosítást kellett végezni az osztályon. Azt a fájlt, amire a fordító meg lett hívva, a Clang `MainFile`-nek nevezi. Egy fordítási egység elemzésekor egyszerre csak egy `MainFile` létezhet. Az elemzés viszont úgy kezdődik, hogy a `Preprocessor` elkezd elemezni a `MainFile`-t. Ebből kifolyólag átmenetileg a modellfájlt kell megjelölni `MainFile`-ként. Az elemzés kezdetén a beépített típusokat leíró kódrészlet egy belső pufferben kerül eltárolásra, valamint a fordító által támogatott *pragma* direktívák kezelői létrejönnek. A probléma forrása, hogy a `Preprocessor` ekkor már rendelkezik mind a megfelelő pragmakezelőkkel, mind pedig a beépített típusokat tartalmazó pufferrel. A problémák elkerülése érdekében ezért ezek az információk a modellfájl elemzése előtt eldobásra kerülnek a `Preprocessor`-ból. Az újbóli inicializáció megakadályozása sokkal nagyobb módosítást igényelt volna szerte a kódbázisban. Ezenfelül szempont volt, hogy a fordítással kapcsolatos kódrészletekre minimális hatással legyenek a módosításaim. Egy új fordítási egység elemzésekor eldobásra kerül minden régi fordítási egységhez tartozó információ a `SourceManager`-ből. Ez modellfájlok elemzésénél nem kívánatos, ezért ezekben az esetekben a `SourceManager` megőrzésre kerül.

A hívott függvény és a modellfájl között a függvény neve hozza létre a megfeleltetést a jelenlegi implementációban. A jövőben külön konfigurációs fájlban lehet majd megadni, hogy egy adott modellfájl milyen függvények modelljeit tartalmazza. Egy modellfájlban belül az azonos nevű függvények között a mangled név segítségével kerül kiválasztásra a megfelelő implementáció. A felhasználónak lehetősége van megadni útvonalakat, ahol a fordító a fájlokat keresi.

A kódbázisba való bekerülésnek a feltétele volt, hogy az implementációhoz tesztek is tartozzanak.

7. fejezet

Továbbfejlesztési lehetőségek

A modellek generálása jelenleg nem megoldott, minden modellt kézzel kell megírnia a fejlesztőknek. A jövőben tervezem, hogy bevezessek néhány sémát, ami alapján a modelleknek legalább egy része automatikusan generálható. A modellek generálása egy külön menetben történhet a tényleges analízis előtt. Ebben a menetben a checker-ek is részt vehetnek, hogy kialakítsák a függvények előkövetelményeit.

Az analízis hatékonysága növelhető lenne azáltal, hogyha nem csupán a C/C++ kód, hanem az AST és a CFG is tárolásra kerülne a modellfájlban. Ezáltal a CFG többszöri generálása elkerülhető lenne.

A régebbi, elavult módszert a BodyFarm-ban, ami a beégetett AST-k használatát jelenti, teljes egészében le lehetne váltani modellből szintetizálás módszerével. Ezáltal többszáz sor kód kiváltható, és könnyebbé válna az AST szerkezetét érintő változtatások bevezetése is. Ezirányban az implementáción a Summer of Code projekttől függetlenül dolgozom.

A modellekben lévő információt jelenleg csak a Static Analyzer használja ki. Lehetséges, hogy a jövőben az optimalizáló számára is érdemes lenne ezeket az információkat elérhetővé tenni, ezáltal gyorsabb generált kódhoz juthatnánk.

A dolgozatban leírt technikákat, melyeket modellek fejlesztéséhez lehet felhasználni, a jövőben tervezem angol nyelven dokumentálni, és a Clang dokumentációi között elérhetővé tenni.

Sajnos az implementáció jelen állapotában még nem robosztus eléggé ahhoz, hogy ipari alkalmazásokon kerüljön tesztelésre. A hibák javítás alatt állnak, és amint elég robosztussá válik a megoldás, ki fogom mérni, hogy mekkora sebességváltozást jelent a modellek használata, valamint milyen hatása van a talált hibák mennyiségére és jellegére. Az eddigi eredmények viszont biztatóak.

8. fejezet

Összefoglalás

A dolgozatban bemutatam az általam fejlesztett bővítményt a nyílt forráskódú Clang C/C++ fordító keretrendszerhez, mely mára a kódbázis része lett. Az én megoldásom public domain eszközök között az első, ami képes hatékonyan támogatni a különböző fordítási egységben lévő alprogramok invariánsainak feltérképezését, illetve megsértésének detektálását C/C++ nyelvek esetében. A megoldást egy Google Summer of Code projekt keretein belül készítettem el. Részletesen bemutatam a motivációt a projekt mögött, az elméletet, amin a megoldás alapul, valamint a megvalósítást magát.

Összefoglaltam számos, a statikus analízis megértéséhez szükséges ismereket, rávilágítottam néhány, a programozási nyelvek elemezhetőségét befolyásoló tényezőre. Megmutattam, hogy egy k -menetes analízis, ahol a k egy rögzített szám, kisebb erővel bír, mint a fixpont iteráció.

Bemutattam továbbá egy lehetőséget, amely segítségével függvények elő- és utófeltételeit lehet ellenőrizni szimbolikus végrehajtás közben, külön modellező vagy specifikációs nyelv használata nélkül. Ezzel adtam egy irányvonalat, hogy hogyan érdemes tetszőleges függvények modelljét úgy elkészíteni, hogy az a lehető legtöbb információt tartalmazza a szimbolikus végrehajtást végző motor számára, úgy, hogy a modell elemzésre alkalmas reprezentációja potenciálisan kisebb memóriaigényű mint az implementációé.

Irodalomjegyzék

- [1] Bove, A., Dybjer, P., Norell, U.: *A Brief Overview of Agda — A Functional Language with Dependent Types*, in Proc. of Theorem Proving in Higher Order Logics 2009, Lecture Notes in Computer Science vol. **5674** (2009), pp. 73–78.
- [2] Csörnyei, Z.: *Fordítóprogramok*, Typotex kiadó, 2006.
- [3] Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: *Columbus - Reverse Engineering Tool and Schema for C++*, in Proc. of International Conference on Software Maintenance 2002, pp. 172–181.
- [4] Hoare, C. A. R.: *An axiomatic basis of computer basis*, Comm. ACM, vol. **12**, (1969), pp. 576–583.
- [5] Horváth G.: *Mintaillesztési módszerek vizsgálata absztrakt szintaxisfákon*, TDK Dolgozat, 2012.
- [6] Horváth G.: *Szemantikus keresés C++ kódbázisban*, BSc Szakdolgozat, 2014.
- [7] Horváth G., Norbert P.: *Clang Matchers for Verified Usage of the C++ Standard Template Library*, Annales Mathematicae et Informaticae (submitted)
- [8] Joshi, A., Tewari, A., Kumar, V., Bordoloi, D.: *Integrating Static Analysis Tools for Improving Operating System Security*, International Journal of Computer Science and Mobile Computing, Vol. **3(4)**, (2014), pp. 1251–1258.
- [9] Lattner, C.: *LLVM and Clang: Next Generation Compiler Technology*, The BSD Conference, 2008.
- [10] Mihalicza J.: *How #includes Affect Build Time in Large Systems*, in Proc. of 8th International Conference on Applied Informatics (ICAI 2010), Volume II., pp. 343–350.
- [11] Papadimitrou, C. H.: *Számítási bonyolultság*, Novadat, 1999.
- [12] Pataki, N.: *Generatív programok helyessége*, PhD Dolgozat, 2013.

- [13] „The Rust programming language”, <http://www.rust-lang.org/>, (2014. november 1.)
- [14] „GitHub: Support macros for Rust-Postgres”, <https://github.com/sfackler/rust-postgres-macros>, (2014. november 9.)
- [15] Stroustrup, B.: *The C++ Programming Language*
Addison-Wesley Publishing Company, fourth edition, 2013.
- [16] Telea, A., Voinea, L.: *An Interactive Reverse Engineering Environment for Large-Scale C++ Code*, in Proc. of the 4th ACM Symposium on Software Visualization (SoftVis '08), 2008, pp. 67–76.
- [17] Xu, Z., Kremenek, T., Zhang J.: *A Memory Model for Static Analysis of C. Programs*, in Proc. of ISoLA'10 4th international conference on Leveraging applications of formal methods, verification, and validation – Volume Part I pp. 535–548.
- [18] Reps, T., Horwitz, S., Sagiv, M.: *Precise Interprocedural Dataflow Analysis via Graph Reachability* in Proc. of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995, pp. 49–61
- [19] Kontribúció clang kódbázishoz, <http://reviews.llvm.org/rL216550> (2014. november 15.)
- [20] Sørensen, M., Urzyczyn, P.: *Lectures on the Curry-Howard Isomorphism*, Studies in Logic and the Foundations of Mathematics **149**, Elsevier, (2006)
- [21] Garcia, D.: *Exploring the design space of contract specifications for C++*, Open Standards, Document N4110, 2014
- [22] Google Summer of Code, <https://developers.google.com/open-source/soc/?csw=1> (2014. november 22.)
- [23] ISO/IEC 9899:TC3, <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf> (2014. november 22.)
- [24] Visual Studio 2015 Preview Announcement, <http://blogs.msdn.com/b/vcblog/archive/2014/11/12/visual-studio-2015-preview-now-available.aspx> (2014. november 22.)
- [25] The SPARK Programming language, <http://www.spark-2014.org/> (2014. november 23.)