Eötvös Loránd Tudományegyetem
Informatikai Kar

Programming Languages and Implementation Summer School - 2019

PLISS - 2019

Témavezető:
Tóth Melinda
egyetemi docens

Szerző:
Ciuciu-Kiss Jenifer Tabita
Programtervező informatikus Bsc

Budapest, 2019
A projekt az Európai Unió támogatásával, az Európai Szociális Alap
társfinanszírozásával valósul meg (EFOP-3.6.3-VEKOP-16-2017-00002).

Szakmai tanulmány

Contents

Chapter 1

Bevezetés

A PLISS (Programming Languages Implementation Summer School) nevú nyári iskola 2019-ben május 20.-24. között, Olaszországban, azon belül egy Bertinoro nevű városkában került megszervezésre. Ennek a szári iskolának nyári iskolának az a célja, hogy a tehetséges fiatalok számára betekintést nyújtson a programozási nyelvek világába. Több különböző témában, elismert szakemberek tartanak előadásokat, lehetőség van kérdezni, illetve a közösen töltött idő alatt tanácsot kérni, kapcsolatokat építeni.

A 2019-es évben a következő témák merültek fel:

- Nyelvek biztonságossá tétele - Cristina Cifuentes: A felhők világában a támadható területek mérete sokkal nagyobb, mint egy helyi hálózat esetében. A jelenleg használt nyelvek csak korlátozott támogatást nyújtanak a biztonsághoz, olyan biztonsági rések merülnek fel, mint például a webhelyen keresztüli szkriptelés, stb. A fontos adatokkal kapcsolatban felmerülő biztonsági rések a legsérülékenyebbek a webes világban az elmúlt 5 évben. A különböző nyelvi megvalósítások miatt, a mikroszolgáltatások felé mutató tendenciával párhuzamosan egyre növekszenek ezek a biztonsági rések, amelyeket meg kell oldani. Az előadás a probléma áttekintéséről szól, bemutat néhány olyan megoldást, ami gyakori alkalmazással rendelkezik. Bemutatja hogyan kezdjük el az első lépéseket egy olyan nyelv felé, amely a mai általánosan használt nyelvek biztonsági koncepcióinak hiányára irányul, webes és felhőalkalmazások esetén egyaránt.

- A szemantikával kezdve - Sylvan Clebsch: A programozási nyelv egy felhasználói felület egy absztrakt nyelv felé. Egy új nyelv létrehozása egy meglévő absztrakció

felé egy hasznos képesség (pl. a Scala a JVM-re, az F# a CLR-re, stb.). Néha azonban nem elég egy meglévő absztrakcióra nem elég, előfordul, hogy szükség van új absztrakcióra is, például az egyéni hardverek kezelésére, hogy új megközelítést nyújtson a párhuzamossághoz, vagy új futási időt biztosítson a forrásnyelv számára. Szó lesz egy kis lépcsős operatív szemantikáról és egy kis futási idő megtervezésről Pony segítégével és a Pony absztrakt gépre vonatkozó hipotetikus kiterjesztésről.

- Polyhedratikus fordítás fordító konstrukciós tervezési mintaként - Albert Cohen: ...a nagy teljesítményű számítástechnikai élmény kihasználása a következő generációs gépi tanulási gyorsítók programozásához.

- Konfigurációs nyelvek tervezése és elemzése - Arjun Guha: Vizsgálni fogjuk hogy, a rendszerkonfigurációs nyelvek (pl- Puppet és Chef) milyen probléma megoldására lettek kitalálva. Megnézzük az erősségeiket és a gyengeségeiket, megvizsgáljuk a konfigurációs nyelvek ellenőrzési és programjavító eszközeinek tervezését és végrehajtását. Beszélünk arról, hogy hogyan lehet mechanikusan leegyszerűsíteni egy nagy valós világbeli konfigurációs nyelvet egy egzszerűbb nyelvre, amely elemzésre alkalmas. Megvitatjuk a kapcsolatot az elemzés tervezése és összetettsége között, és megnézzük, hogyan lehet a problémáinkat lekérdezni egy SMT megoldáshoz.

- Egy kis V8 és WebAssembly - Ben L. Titzer: Ez az előadás V8 JavaScript és WebAssmebly megközelítésének részleteit fogja részletesen tárgyalni. Megismerjük, hogy hogyan alakult a WebAssembly fejlődése az évek során, mekkora hatékonyságot sikerült elérni. Kezdetben minden sokkal lassabb volt. Azonban a technológia fejlődésével, már nem volt elfogadható, hogy perceket várjunk egyetlen weboldal betöltésére, igz változtatni kellett. Eleinte kisebb fejlődéseket sikerült csak elérni, azonban ahogy egyre több ember kezdett el egyre komolyabban foglalkozni a témával, egyre komolyabb előrehaladások következtek. Ma már nagyon gyors az egész webes világ, igy a WebAssembly is, azonban ez még mindig nem elég, még mindig rengeteg terv van a jövőre nézve, hiszen ez egy olyan terület, ami soha nem lehet elég gzors.

- Milyen spektrum jelenti a nyelv implementációt - Ben L. Titzer: A nyelv implementáció egy olyan dolog, aminek a méretét nehéz felbecsülni, amikor elkezdünk egy új nyelvet implementálni, még fogalmunk sem lesz arról, hogy mennyire fogja

majd elnyerni a tetszését az embereknek, mennyire lesz felkapott, stb. Gondoljunk arra, hogy először a nyelv akkora mint az ujjunk. Aztán képzeljük el, hogy mi állunk valahol, és nézzük az újjunkat. Majd képzeljük el, hogy a Grand kanyonban nézzük az újjunkat. Majd gondoljunk bele, hogy van egy sokkal nagyobb önmagunk, aki éppen az újját nézi. amiben benne van a Grand kanyon, amiben benne vagyunk mi az ujjunkat nézve, amiben szintén van egy Grand kanyon...

- Reszponzív fordítók - Nicholas Matsakis: Számos fordítókról szóló tankönyv "kötegelési folyamatként" kezeli a fordítást, ahol a fordító egy bemeneti fájlt vár, végrehajt egy meghatározott lépéssorozatot, és végül egy objektumkódot állít elő. A felhasználók azonban egyre inkább elvárják, az olyan IDE-kkel való integrációt, mint a VSCode, ami más struktúrát igényel. Ezen túlmenően számos nyelven rekurzív konstrukciók vannak, ahol a helyes feldolgozási sorrendet nehéz statikusan megállapítani. Megvitatunk néhány dolgot a Rust csapat munkájából a fordító szerkezetátalakításával kapcsolatban, hogyan támogatja a Rust az inkrementális összeállítást és az IDE integrációt.

- Dolgok amiket tanultam - Nicholas Matsakis: Megtudjuk, hogyan alakult a Rust fejlődése, milyen nehézségek merültek fel a tervezések közben. A Rust-ról fontos tudni, hogy a kezdetek óta örzi néhány tulajdonságát, és azért, hogy a korábban megirt kódok az újabb verziókkal is forduljanak. Ezt sajnos nem mindig sikerült biztositani, de a fejlesztők törekedtek arra, hogy minnél kevesebbet kelljen változtatni az újabb verziók használata esetén, és arra, hogy a nyelv egyre hatékonyabb legyen. Megtudjuk, hogy az előadó mit tanult mindebből a fejlődésből. A legfontosabb dolog az, hogy a fejlesztők kommunikáljanak a felhasználókkal, hiszen az a cél, hogy azok számára optimalizálják nyelvet, akik az iparban használják, hiszen az egész fejlesztés miattuk zajlik. Ehhez lehet különböző vitákat tartani, előzetes fejlesztési tervet publikálni, vagy bármilyen kreativ dologgal megvalósitható ez a kommunició, a lényeg az, hogy legyen.

- Ellenőrzött repülés közbeni párhuzamos szemétgyűjtés (on-the-fly garbage concurrent garbage collection) keresése a modern processzorok esetében - Tony Hosking: Bemutatom a párhuzamos szemétgyűjtés alapelveit, beleértve a repülés közbeni szemétgyűjtést, a helyességre összpontosítva az alkalmazásprogram egyidejű mutá-

ciója miatt. Ezután megvitatjuk ezeket a problémákat és a megoldásokat az algoritmus biztonságát megőrző invariánsok tekintetében egy absztrakt szemétgyűjtő algoritmussal jellemezve. Megismerjük a konkrét szemétgyűjtő algoritmust, melynek célja, hogy hatékonyan működjön a gyenge memória többszintű hardveren, és informálisan beszéljük meg a helyességét. Megnézzük, hogyan tudtuk formálisan és mechanikusan ellenőrizni az algoritmus helyességét a gyenge memória használatára az Isabelle bizonyító asszisztens segítségével.

- Statikus programelemzés - Anders Moller: A statikus programelemzés a számítógépes programok viselkedésével kapcsolatos érvelés művészete, anélkül, hogy ténylegesen futtatná őket. Ez nemcsak a hatékony kódok előállítására szolgáló fordítók optimalizálásában hasznos, hanem az automatikus hibakereséshez és más, a programokat segítő eszközökhöz is. Az előadás bemutatja a statikus programelemzés alapvető elveit és algoritmusait. Korlátozáson alapuló megközelítést alkalmaz, ahol a megfelelő kényszerrendszerek koncepcionálisan elosztják az elemzést egy front-end-re, amely a programkódból és egy back-end-ből származó korlátozásokat generál. Ezek megoldják az elemzési eredmények előállításához szükséges korlátokat, ha az idő lehetővé teszi.

- Elmélet és gyakorlat, nyelvek fejlesztése az iparban - Joe Pamer: Az ipar által alkalmazott nyelv- és fordítótervezési megközelítés nagyon mértékben eltérhet a tudományos szakirodalomban dokumentált technikától. A futásidő teljesítménye és az újrafelhasználhatóság gyakran hátrébb helyezkedik el, mint a méretezhetőség és a marketing. Ez a proritásbeli különbség olyan tervezési vagy végrehajtási döntéseket eredményezhet, amelyek intuitívak lehetnek, de szükségesek, hogy a meredek elfogadási görbék előtt maradjanak, amelyet a tapasztalatok mutatnak. Az elmúlt néhány év tapasztalata alapján szeretnék néhány dolgot jobban megvilágítani ezzel kapcsolatban, és talán néhány új dolgot is mondani. Az előadásban a nyelvfejlesztés elmélete és gyakorlata közötti különbségek lesznek felsorolva, amelyeket a nyelfejlesztési iparban tapasztaltam. Meg fogjuk ismerni a VM-ektől a back-end kódgenerátorokon, a csekkereken, a standard könyvtárakon át a teljes nyelvi jellemzőket és felépítésüket. Eközben sok anekdota fel fog majd merülni a mai népszerű fejlesztőeszközök mögötti kellemes meglepetésekről és sajnálatos hibákról. Elmondom, hogyan látom a következő évtizedben a nyelvfejlesztés jövőjét.

- A soron következő perzisztencia apokalipszise - Mario Wolczko: A DRAM dominanciája hamarabb véget érhet, mint gondolná. Az új memória technológiák megnövekedett sűrűséget, költségcsökkentést ígérnek. A programozással és a programozási nyelvek tervezésével kapcsolatos következmények mélyrehatóak lesznek. Ebben az előadásban ismertetem azokat a lehetőségeket és kihívásokat, amelyek nem a Volatile RAM-ból származnak, valamint a megoldandó komoly problémák sokaságát.

Chapter 2

Előadások

A következő fejezetben részletesen bemutatok pár számomra hasznos, a nyári iskolán érintett témát.

2.1 Ben Titzer - Egy kis V8 és WebAssembly

Ben L. Titzer WebAssembly-vel dolgozik a Google-nél a müncheni csapatban.

Röviden összefoglalva, a WebAssembly alacsony szintű bájtkód. Implicit célja, hogy gyors legyen a verifikáció és a fordítás. Explicit célja, hogy gyosan interpretálható legyen. Jellemzői a statikus típusok, az argumentum számlálás, a direkt és indirekt hívások, és az hogy nem lehet operátorokat túlterhelni. Egy kód egysége a mudule, ez állhat globálisokból, inicializációkból, függvényekből. A module importálható és exportálható más modulokba.

Példa egy WebAssembly module-ra:

- Bináris alak
- Típus deklaráció
- Importok (típusok, függvények, globálisok, memória, táblák)
- Táblák, memóriák

```
header: 8 magic bytes
types: TypeDecl[]
imports: ImportDecl[]
funcdecl: FuncDecl[]
tables: TableDecl[]
memories: MemoryDecl[]
globals: GlobalVar[]
exports: ExportDecl[]
code: FunctionBody[]
data: Data[]
```

- Globális változók
- Exportok
- Függvény definíciók (bájtkódban)

Példa WebAssembly bájtkódra:

- Függvény típusa
- Stack
- Strukturált control flow
- Egy nagy memóriaegység
- Alacsony szintű memória műveletek
- Alacsony szintű aritmetika

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

A WASM motor működése a következő: Be kell tölteni és validálni kell a WASM bájtkódot, allokálni kell az adat struktúrákat. Le kell fordítani, vagy interpretálni kell a WASM bájtkódot. Integrálni kell a JavaScript API-t. Meg kell valósítani a memória kezelést, a hibakezelést, és a szemétgyűjtést.

A WebAssembly-nek köszönhetően várhatóan rövid lesz a fordítási idő, illetve nem lesznek komoly akadályok futás közben. Csúcs teljesítmény, amely közelít a natív kódhoz (kb. 20 %). Minden "felnőtt" motor implementációja használja a saját JIT-ját. (A Just In Time vagy JIT eredetileg egy olyan gyártásszervezési technológia, amely kellő mennyiségű, adott időben való szükséges termék előállítását biztosítja. Itt azt jelenti, a WebAssembly fejlődése során minden implementáció megtartotta az eredeti szervezési technológiáját, ez nem változott meg az optimalizáció során.) Néhány implementáció:

- V8: TurbaFan AOT teljes modulos fordító

- Spider Monkey: Ion AOT alapú teljes fordító + Ion JIT háttérrel
- JSC: B3 szemléltető példaként szolgál, teljes modul
- Edge: lusta fordító, belső bájtkód fordítására és dinamikus sorosításra Charkra-val

A lassú, szüneteket tartalmazó fordítás nem elfogadható. A WebAssemblynek gyorsnak kell lennie, a több másodperces fordítási idő nem elég.

V8 TurboFan fordító a WASM-ot kb. 1-1.3MB/s alatt fordítja le. Fordítás menete a következő: felépíti a teljes csúcshalmazt, fellcimkézi a gráfot. Lefordítja a felcimkézett gráfot alacsony szintű bájtkódra. Regiszter allokáció. Kódgenerálás.

Egy 10 MB méretű modul 8 másodpernyi fordítást igényel. Ez nagyon lassú, ennél sokkal jobbra van szükség.

Nem lehetne egyszerűen csak gyorsítani a fordítást?

A válasz az, hogy lehet, méghozzá Traplf használatával elérhető, hogy kisebb gráfokat generáljunk, ezáltal minden tovóbbi lépés gyorsítható. Ezzel 30%-os gyorsulás érhető el. Ez jó kezdet, de ennél még mindig gyorsabb fordításra van szükség.

Nem lehetne egyszerűen csak párhuzamosítani a fordítást?

De igen, ezt a TurbóFan fordító valósította meg 2016 májusában. A fordításnak nem minden lépése párhuzamosítható a V8 implementációban.

Ezzel a megoldással 5-6-szoros gyorsulás érhető el. Így az a 10MB méretű fájl, ami eddig 8 másodperc alatt fordult le, mostmár 1,5 másodperc alatt fordul.

Nem lehetne aszinkrinozálni a fordítást?

A WebAssembly JavaScript nyújt olyan szolgáltatásokat, amelyekkel aszinkronizálni lehet a fordítást: WebAssembly.compile(bytes). Az API 2016-ban lett implementálva. Az aszinkron iplmenetáció 2017 áprilisában lett implementálva. Sok trükköls feltételt hozzáadtak 2017 novemberében.

Így a fordítási idő 1,5 másodpercről 0,15 + max 0,003 másodpercre csökkent. Nem lehetne folyamatizálni a fordtást?

Az alapötlet az, hogy már letöltés közben el lehetne kezdeni a fordítást. Ezt oldja meg a WebAssembly.cimpileStreaming(), amely 2017-ben lett megvalósítva és 2017 de-

cemberében terjedt el. Egyszerre 8 szálat is lehet párhuzamosan működtetni 50MBit/s sebességgel.

Tehát arra a kérdésre, hogy nem lehetne-e egyszerűen csak gyorsítani a fordítási időn, az a válasz, hogy de, különböző módszerekkel, egyre gyorsabb fordítást lehet elérni. A kérdés csak az, hogy ez így elég-e? Ekkor jött az az ötlet, hogy nem a már meglévő fordítókon kéne gyorsítani, hanem lehetne írni egy teljesen új fordítót. Így készült el a Liftoff.

Már a prototípus is megközelítőleg 10-20 szoros gyorsulást ért el.

A Liftoff fordító felépítése, jellemzői:

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

- Gyors, folyamatosan forduló fordító
- A bájtkós templateket használ
- Dekódoló, verifikáló
- "On-the-fly" regiszter allokáció, tehát kódgenerálás közben zajlik a regiszter allokáció
- Hordozható interfész a dekodoló logika, az áramlás szabályozása és az alacsony szintű kódgeneráció között.
- Exportok
- Függvény definíciók (bájtkódban)

Tehát a Liftoff hatalmas előrelépést jelentett a webes világban. Sikerült 5x-ére gyorsítani az indulási időt, és 50%-al kisebb teljseitményt igényel.

A fordítási idő és az ehhez szükséges erőforrás kapcsolata fontos, több különböző stratégián alapuló implementációt is kidolgoztak, attól függően, hogy mire szeretnénk optimalizálni. Ezek a stratégia implementációk a következők: a Liftoff AOT TurboFan-ra épülve, Liftoff AOT dinamikus sorosítással, Liftoff lusta fordítás dinamikus sorosítással, Liftoff háttér fordító dinamikus sorosítással.

A WebAssembly fejlődése:

- Pázhuzamosítás - 2016 (kész)
- Aszinkronizáció - 2016 (kész)
- Folyamatosítás - 2017 (kész)
- Várható értéken alapuló sorosítás - 2018 (kész)
- Két-soros JIT stratégia - 2018 (kész)
- Fordítást segítő szekció - Prototípus folyamatban

- Ingyenes használatlan kód kihagyás - Folyamatban
- Futtatható JIT stratégia implementáció, amelyet önálló alkalmazásként indítanak el - Fejlesztés kezdése folyamatban

Tehát annak ellenére, hogy a WabAssembly fejlesztésében nagyon szép sikereket értek el, ez koránt sem jelenti azt, hogy a fejlesztést abba lehet hagyni, mindig szükség lesz gyorsabb és gyorsabb fordításra.

2.2 Nichiolas Matsakis - Dolgok amiket tanultam (TIL)

Először is fontos tisztázni, hogy mi az a TIL. Az angol Today I Learned, vagyis Ma Tanultam kifejezésből ered, ami arra utal, hogy minden nap tanul valami újat az ember, és fontos tudni erről, gondolkodni ezen. Valójában kifejezőbb lenne azt mondani, hogy Things I Learned, dolgok amiket tanultam, hiszen az amit tegnap tanultam, az még ma is releváns és holnap is releváns marad a tegnap meg a ma elsajátított tudásom. A TIL rövidítés azonban mindkét esetben működik, tehát ha azt használjuk, akkor kifejezhetjük mindkettőt.

A első dolog amit érdemes megtanulni, hogy amikor egy konferencián egy előadáson meghallgatjuk a bevezetést, akkor csak annyit látunk, hogy milyen témában, milyen módon, gondolatmenet alapján indúlt el az illető, azonban azt sosem mondják el, hogy néhány hónapja, amikor elkezdett foglalkozni a témával, akkor még fogalma sem volt arról, hogy amit kitalált, az működni fog-e. Senki nem mondja el, hogy milyen nehéz volt eljutni oda, ahol most tart, de fontos, hogy tudjuk hogy a dolgok nem úgy működnek, hogy az első ötlet egyből tökéletes lesz, és minden úgy fog működni, ahogy először kitaláltuk. A lényeg az, hogy nem kell feladni az első kudarc után, hanem folytatni kell, és akkor előbb vagy útóbb megtaláljuk a helyes megoldást.

A Rust története:

A minimális mag tulajdonságai, hogy tudja kezelni a tulajdonjogokat és a hitelfelvételt valamint ezek jellemzőit; nulla költségű absztrakciókra képes; nincs futási idő szemétgyűjtés, nem ismeri a szálak fogalmát, nem tudja kezelni őket. A kiterjesztett könyvtárak alapvetően tartalmazzák a vektorokat, hasmappákat és referenciaszámláló pointereket. Magasabb szinten ismerik a szervereket, adatbázisokat, work-stealing stratégiát futási időben. A work-stealing stratégia egy felcímkézési stratégia a többszálas programokhoz.

Ez azonban nem volt mindig így, kezdetben Erlanghoz hasonló munkák voltak, futási időben hajtódtak végre a feladatok, szükség volt a szemétgyűjtésre. A tulajdonjogok egyfajta teljesítmény optimalizásálként voltak hozzáadva. Ez azt jelentette, hogy érvénytelen volt, könnyen összeomlott.

Ezt követően bevezetésre kerültek a régió típusok a rugalmasság megszerzése érdekében, de megmaradt egy ocalm/C szerű mutációs modell. Átléptünk az öröklődésre, ahol már a következőket tudtuk használni: T volt a tulajdonos, & T volt a megosztott, de mutálhatatlan, & mut T, egyedi és mutálható.

Néhány további verzió után megjelent a generikus programozás, voltak osztályok és interfészek, illetve a szálak és a futási idő megszűnt.

A rengeteg változtatás és verzió ellenére voltak olyan tulajdonságok, amelyek végig igazak voltak a Rustra. Ezek a következők: kompromisszummentes hatékonyság, biztonság és helyesség, kíváncsiságot és komoly kutatást hangsúlyozó CoC és kultúra. Fejlesztés közben mindvégig a legjobb megoldás megtalálása volt a cél, nem pedig valamiféle verseny megnyerése.

A kutatás egyfajta spirálszerű folyamat volt. Az első lépés az, hogy létrehozzunk valami újat, a második a megfigyelés, mérések, meg kellett pontosan határozni, milyen és mekkora fejlődést értünk el ezzel az új dologgal, majd ha már pontos eredményeket tudtunk felmutatni, akkor jöhet az a része, amikor csak profitálni kell. Azonban itt nincs vége, mindez megint kezdődik előről, hiszen nem lehet a végtelenségig ugyanabból a dologból profitálni, szükség van új eredményekre.

Az a dolog, ami működhet az eredményes kutatás eléréséhez a következő: találni kell egy olyan kutatási területet, ami tényleg érdekel, függetlenül attól, hogy most még nincs egy olyan konkrét probléma, amit megoldanál. Használd a terület jelenleg létező tudását, oldj meg feladatokat ezzel, és eközben rá fogsz jönni, hogy mi az, ami hasznos lenne, ha létezne, megkönnyítené a munkát. Ha megvan mit kéne csinálni, akkor már neki is lehet fogni a konkrét probléma megoldásának. Ha ez kész, kezdje előlről onnan, hogy használja a már meglévő képességeket. A témaváltás nem érdemes az első megoldott probléma után.

A fejlesztés úgy zajlott, 6 hetente adták ki a stabil eredményeket, illetve minden este adtak ki aznapi eredményeket (többnyire). Az Rust 1.0 2014 környékén jelent meg, ami csak éjszaka volt használható. Ez egyértelműen nem volt elég, az embereket nem érdekli,

hogy mennyi munkát fektettél a fejlesztésbe, és hogy milyen akadályokba ütköztél, csak az, hogy mi az, ami működik, és mi az ami nem.

Jövőbeli kompatibilitási megfigyelések: A türelmi időt a kód javítására nem kell elfogadni. Az apróbb hibákat a kódban elnyomják a függőségek.

A fordító elfogadja és lefordítja mind a 2015-ös, mind a 2018-as Rust kódot, mindkettőt ugyanarra a belső megjelenítésre fordítja. Tehát hasonlóan működik, mint a C++ és Java fordítók.

A szemantikus verziók a következőek voltak:

- Version 1.0
- Version 1.1 -> új képességekkel, de az 1.0 verzióbeli kódok továbbra is működtek
- Version 1.1.1 -> csak hibajavításokat tartalmaz, de mindenképpen érdemes ezt használni pont emiatt
- Version 2.0 -> a 1.x verziót használók valószínűleg meg kell változtassanak néhány dolgot, hogy továbbra is működjenek

A koherencia miatt be kellett tartani a következő szabályokat:

- Minden típusra legfeljebb egy implementáció. Ezt viszonylag könnyű volt egy rekeszben megvalósítani, de minden esetben ellenőrizni kell
- Az implementációnak definiálnia kell vagy a saját típusát, vagy annak a tulajdonságnak a típusát, amelyre vonatkozik.
- "Árva szabály" a Haskellhez hasonlóan

A duplikáció vizsgálat nem egyszerű. Egyértelmű, hogy a következő esetben ütközés lép fel:

```
impl Hash for u32 { ... }
impl Hash for u32 { ... }
```

De a következő esetben már problémákba ütközünk:

```
trait AsU32 {
  fn as_u32(&self) - &u32;
}

impl Hash for u32 { ... }
impl<T: AsU32> Hash for T { ... }
//   ^^^^^^^^
```

Ha a kötvetkező sor lézetik,

```
impl AsU32 for u32 { ... }
```

akkor a következő két sor már duplikációt fog okozni.

```
impl Hash for u32 { .. }
impl<T: AsU32> Hash for T { .. }
```

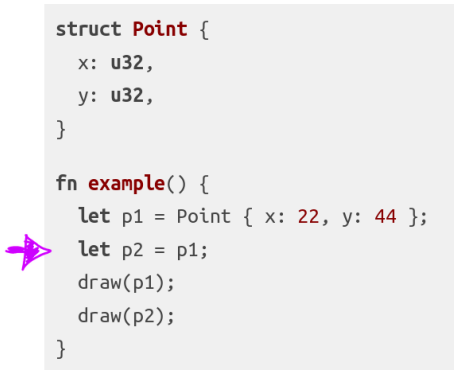
A kérdés az, hogy ezt a problémát hogyan kéne megoldani. A megoldási elv az, hogy pontosítani kell a duplikáció fogalmát, és az operátor szemantikát, hogy azok jól definiálva legyenek, és több felhaszálói esetet fedjenek le. Egy magas szintű összegzés a változásokról a következő:

- Teljes támogatás a több paraméterrel rendelkező osztályok számára, beleértve a funkcionális függőségek egyszerűsített változatát, amely a jövőben társult típusokká alakulhat át.
- Általános explicit és saját típus támogatáson kívül & self és & mut self stb., tehát a self: Rc<Self> saját típusdeklaráció lehetővé válik

- A koherencia szabályainak kibővítése a rekúrzív működéshez és az árvák pontosabb megkülönböztetéséhez
- Felülvizsgálni a vtable algoritmust, hogy fokozatosabb legyen
- Felülvizsgálni az algoritmusmegoldó módszert a vtable megoldás részeként

A koherencia még mindig probléma, a lézető szabályok további problémákat okoznak, trükkös, kiegyensúlyozó megoldásra van szükség.

Egy másik változás a korábbi verzióhoz képest:



```
struct Point {
  x: u32,
  y: u32,
}

fn example() {
  let p1 = Point { x: 22, y: 44 };
  let p2 = p1;
  draw(p1);
  draw(p2);
}
```

A korábbi Rust verzióban a p2 egy másolat lenne a p1-ről, de ekkor fellép az a probléma, hogy ha hozzáírunk egy z: Box<u32> sort, akkor az elveti a változásokat. A későbbi verzióban explicit meg kell írni a másolást a Point típusra.

A Rust fordító kezdetben nem fordította át a számítógépes forráskódot egy sokkal szigorúbb szintaktikus formára. Azért nem, mert felmerült az a félelem hogy a hibaüzenetek sérülni fognak. Az eredmény az lett, hogy nagyon sok hiba merült fel.

Az újrakódolást datalogként valósítja meg. Egy jó C++ prolog megoldót használtak az Oracle-től. Írtak egy kis méretű Rust könyvtárat a datalog hatékony megoldásához. A tulajdonságok megvalósítását újra kellett kódolni lambda prologként.

Később felmerült több más kompatibilitási probléma is a korábbi verziókkal kapcsolatban, mint például a nyelvtani kétértelműség és a ciklis detekció.

Egészen a közelmúltig vita volt arról, hogy a ? hogyan legyen kiterjesztve, de erre sikült egy egyszerű megoldást találni, valószínűleg eddig rossz irányba kerestünk, ezért nem vettük észre.

Felmerül a kérdés, hogy mennyi információra van szükség ahhoz, hogy magabiztosan megértsük, hogy mit csinál egy adott kódsor, és az, hogy milyen nehéz összegyűjteni ezt az információt.

A három tengelyes elv a következő:

- Alkalmazhatóság: Mit lehet kihagyni? Van-e valamilyen alternatív megoldás?
- Teljesítmény: Mennyibe kerül az adott működés?
- Kontextus függőség: Mennyire van szükség a kontextusra?

Ez a három fontos tengely. Minnél nagyobb az egyik, annál kisebb a többi.

Most nézzük meg néhány példán keresztül, hogy hogyan alkalmazható a három tengelyes elv.

A **?** egy szintaktikus cukorka. Az alkalmazhatósága kicsi, van alternatív megoldás. A teljesítményigénye magas, radikálisan megváltoztatja a vezérlőáramlást (return), a kontextusfüggősége közepes, konzultálnia kell a visszatérési típus meghatározásához.

Típus következtetés esetében az alkalmazhatóság közepes, mindig intrafunkció, a teljesítmény magas, radikálisan megváltoztatja a vezérlőáramlást (return), a kontextusfüggősége változó, többnyire intrafunkció, de az aláírások számítanak.

A hibakezelés Rustban több részből épül fel. Az hibás kódokról kapunk információt, megjelenik a hiba, és az, hogy mi váltotta ki. A kivételeket nehezebb kezelni, nem mindig lehet tudni, hogy mi váltotta ki őket. Rustban a lekérdezésükhöz a ? operátort lehet használni, úgy mint Swiftben a try kulcsszót.

Két objektum kapcsolata feszültég szempontjából lehet: magas szintű kód, vagy nagy feszültségű kód.

Nulla költségű absztrakcióhoz szükség van olyan technikák összességére, amelyek segítenek megoldani a fentebbi kapcsolatok problémáját nulla költséggel.

A Rust képes szálak és megosztott memória használatára, adatverseny nélkül. A technika az, hogy amikor feszültésget talál, akkor meg kell határozza pontosan az igényeit, gyakran előfordul, hogy ezek az igények egyszerűsíthetőek, de ezt időbe telik kideríteni.

Néhény felbontás nem része a nyelvnek, de ezekre vannak alternatívák: egyezmények, linkek, dokumentáció, vagy jó hibaüzenetek.

A pluralizmus az egyik legfontosabb dolog amelyre a Rust képes. A pluralizmus egy olyan állapot vagy rendszer amelyben két vagy több állapot, csoport, elv,sorrás, stb. egyszerre áll fent.

Nem igazán van új kritérium egy ideje, de fontos tudni, hogy az alapító csoportból szinte senki nem vett részt a nyilvános vitákban a Rustról. Miért jó az alapítócsoport tagjainak, hogy kihagyják ezt a vitát, és megtartják az ötleteiket, indokaikat a végső döntéssel együtt saját maguknak?

A válasz az, hogy ez nem jó nekik, ezt nem helyesen teszik. Fontos, hogy eljárjunk találkozókra, nyilvános vitákra ahhoz, hogy megtudjuk mire van szükségük a felhasználóknak. Fontos, hogy alapos összefoglaló dokumentációt készítsünk a feltárásokról. Akár előzetes döntést is közzé lehet tenni, hogy megtudjuk milyen reakciókat vált ki az emberekből a végső döntés feltárása előtt. Az összefoglaló dokumentációban gyüjtsük össze az újdonságokat, az érvekkel és a kompromisszumokkal együtt.

Fontos, hogy kommunikálj a haladásról és a tervekről. A Rust fejleszői nem a legjobbak ebben, gyakran figyelmen kívül hagyják a felhasználók igényeit, és csak a saját véleményük alapján hoznak döntéseket, de próbálnak változtatni ezen.

2.3 Statikus program analízis - Anders Moller

Először beszéljünk a típushibákról. Ha ésszerű korlátozásokat vezetünk be a műveletekre, akkor csökkenteni lehet a típushibákat. Ezek a korlátokzások a következőek:

- Az aritmetikai operátorok csak egészekre használhatóak.
- Összehasonlításokat csak az azonos típusú értékekre alkalmazhatunk
- A ki- és bemenet csak egész típusú lehet.
- A kikötéseknek egészeknek kell leniük
- Csak a függvényeket lehet meghívni
- A * operátor csak a pointerekre vonatkoznak
- Mezőt keresni csak rekordokon lehet.

Ha megszegjük ezeket a korlátozásokat, akkor ezek még a futás előtt megjelennek.

Előfordulhat-e, hogy mégsem vesszük észre fordítás közben a típushibákat, és a megkötések mellett is kapunk futási idejű típushibát?

Ez egy érdekes kérdés, ezek alapján nem lehet azonnal eldönteni.

Használjunk konzervatív megközelítést. Egy programozási nyelv típusozható, ha eleget tesz néhány típuskorlátozásnak. Ezek a korlátozások szisztematikusan származtathatóak a szintaxisfából. Ha egy nyelv tipusozható, akkor nem jelenhetnek meg futási idejű típushibák. Ennek következtében előfordulhat, hogy egy program úgy sem fordul le, hogy valójában nincs benne típushiba.

A következőkben meg fogjuk nézni, a Damas-Hindley-Milner típus következtetési technikát, amelyet például a következő nyelvekre alkalmaztak: ML, OCaml, Haskell, stb.

Az a lényeg, hogy a típusellenörzőt okosabbá kell tenni. Ez nem könnyű feladat, ezért is jelenik meg nagyon sok publikáció ebben a témában.

Például az egy szándékos pragmatikus döntés volt, hogy Java-ban bevezették a kovariáns tömböket.

Amit a típusokról tudni kell:

- A típus leírja egy változó lehetséges értékhalmazát.
- A típusok a nyelv nyelvtana alapján generált feltételeknek eleget kell tegyenek.
-

Mi most egy AST-bőn generálunk típuskorlátozásokat. Minden korlátozás egy egyenlőségen alapul, tehát egy egységesítési algoritmussal el lehet dönteni mindenről, hogy eleget tesz e a típuskorlátozásoknak. (Szükség lehet a polyorzfizmus és az altípusok ismeretére.)

A korlátgenerálás folyamata:

I:	$[\![I]\!] = \texttt{int}$
E_1 *op* E_2:	$[\![E_1]\!] = [\![E_2]\!] = [\![E_1 \text{ op } E_2]\!] = \texttt{int}$
$E_1 == E_2$:	$[\![E_1]\!] = [\![E_2]\!] \wedge [\![E_1 == E_2]\!] = \texttt{int}$
`input`:	$[\![\texttt{input}]\!] = \texttt{int}$
$X = E$:	$[\![X]\!] = [\![E]\!]$
`output` E:	$[\![E]\!] = \texttt{int}$
`if` (E) $\{S\}$:	$[\![E]\!] = \texttt{int}$
`if` (E) $\{S_1\}$ `else` $\{S_2\}$:	$[\![E]\!] = \texttt{int}$
`while` (E) $\{S\}$:	$[\![E]\!] = \texttt{int}$

$X(X_1, \ldots, X_n)\{ \ldots \texttt{return}\ E; \}$:

$$[\![X]\!] = ([\![X_1]\!], \ldots, [\![X_n]\!]) \to [\![E]\!]$$

$E(E_1, \ldots, E_n)$:

$$[\![E]\!] = ([\![E_1]\!], \ldots, [\![E_n]\!]) \to [\![E(E_1, \ldots, E_n)]\!]$$

`alloc` E:	$[\![\texttt{alloc}\ E]\!] = \&[\![E]\!]$
$\&X$:	$[\![\&X]\!] = \&[\![X]\!]$
`null`:	$[\![\texttt{null}]\!] = \&\alpha$ (each α is a fresh type variable)
$*E$:	$[\![E]\!] = \&[\![*E]\!]$
$*X = E$:	$[\![X]\!] = \&[\![E]\!]$

Ez az alapötlet, de az nem közvetlenül vezethető át a mi típusrendszerünkre.

Legyen a $f_1, f_2, \ldots, f_m$ az a halmaz, amely a lehetséges mezőneveket tartalmazza.

$\{X_1 : E_1, \ldots, X_n : E_n\}$: $[\![\{X_1 : E_1, \ldots, X_n : E_n\}]\!] = \{f_1 : \gamma_1, \ldots, f_m : \gamma_m\}$

where $\gamma_i = \begin{cases} [\![E_1]\!] \text{ if } f_i = X_j \text{ for some } j \\ \alpha_i \quad \text{otherwise} \end{cases}$

$E.X$: $[\![E]\!] = \{f_1 : \gamma_1, \ldots, f_m : \gamma_m\}$

where $\gamma_i = \begin{cases} [\![E.X]\!] \text{ if } f_i = X \\ \alpha_i \quad \text{otherwise} \end{cases}$

Az egységesítéssel kapcsolatban sok probléma merült fel, például az egyenlőségek esetében.

Nézzük a következő példát:

$k(X, b, Y) = k(f(Y, Z), Z, d(Z))$

Erről így nem tudjuk eldönteni, hogy egyenlőek-e. A megoldás egy hozzárendelés, minden változót rendeljünk hozzá egy zárt alakhoz. Így a fenti példában mindkét oldal egyenlővé válik.

$X = f(d(b), b)\ Y = d(b)\ Z = b$

$c(t_1, \ldots, t_k) = c(t'_1, \ldots, t'_k) \rightarrow t_i = t'_i \forall i$

Az általánosítási hibák:

- Konstruktor hiba: $d(X) = e(X)$
- Aritási hiba: $a = a(X)$

A lineáris egységesítési algoritmus Paterson és Wegman definiálta 1978-ban. Ez $O(n)$ idejű, és megtalálja a legáltalánosabb egységesítéseket. Ez használható egyfajta back-end -ként a típusellenőrzéshez, de csak véges esetekben.

Tehát az implicit korlát a kifejezések egyenlőségére a következő:

A reguláris kifejezések lehetnek:

- Végtelen, de ismétlődő kifejezések:
  - $e(e(e(\ldots)))$
  - $d(a, d(a, d(a, \ldots)))$

- $f(f(f(\ldots)), f(\ldots), f(f(\ldots)), \ldots)$

- Csak végtelen de sokféle különböző eset

Példa nem regulásir kifejezésre: $f(a, f(d(a)), f(d(d(a)), f(d(d(d(a))), \ldots)))$

A reguláris egységesítésre vonatkozó következő állítást Huet mondta ki 1976-ban. Az egységesítési probléma megoldható $O(n * A(n))$ műveletigénnyel, abban az esetben, ha únió keresési algoritmust használunk.

A képletből az $A(n)$ rész az inverze az Ackermann fügvénynek, ami azt jelenti, hogy a legkisebb k -ra $n \leq Ack(k, k)$ és ez a szám sosem lesz 5-nél nagyobb n semmilyen valós értékére.

Most beszéljünk az implementációs stratégiáról. Először is szükség van különböző típusok reprezentálására, a típusváltozókat is beleértve. Map-et készítünk az AST csócsókból és a typusokból. Olyan halmazaokat keresünk, amelyekbe több típus is beletartozik különböző szempontok szerint, ezáltal úniókat hozunk létre. Ezt követően átalagítjuk őket AST formátumra, legeneráljuk a korlátainkat, és menet közben egyesítjük az azonos únióba tartozó típusokat. Eközben jelentést készítünk, ha valamelyik egységesítésben hibát találunk.

Chapter 3

Befejezés

A PLISS nevű nyári iskola tehát nagyon sok érdekes témát érintett. Az egyetemen tanult tudásra alapozva, azon túli ismereteket adott át, problémákat vetett fel, néhány esetben meg is oldva azokat, néhány esetben pedig felkeltve a hallgató érdeklődését, olyan problémákat említett, amelyek még megoldásra várnak.

Összességében véve tehát ez a nyári iskola kitágította az átlagos tehetséges egyetemisi eddigi látókörét, olyan érdekes problémákat felvetve, amelyekről eddig nem halott, de megvolt hozzá a megfelelő tudása, hogy megértse. Az előadók és a szervezők mind elismert szakemberek voltak, akik már letettek valamit az asztalra, ezáltal motiválva a fiatalságot, hiszen ők is érhetnek el hasonló sikereket, csak meg kell dolgozniuk érte.

Chapter 4

Források

- https://pliss2019.github.io/talks.html
- Anders Moller - Michael I. Schwartzbach (2018). Static Program Analysis