

Joonatan Kuosa

Ohjelmointikielten tyypvirheet

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintutkielma
Kesäkuu 2020

TIIVISTELMÄ

Joonatan Kuosa: Ohjelmointikielten tyyppivirheet
Kandidaatintutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Kesäkuu 2020

Yksi ohjelmointikielten suurimmista eroista syntyy tyyppijärjestelmän valinnasta, joka vaikuttaa siihen voiko ohjelmoija luottaa koodin toimivuuteen ilman sen ajamista. Valinta tyyppijärjestelmästä on kielen suunnittelijalla, mutta sen vaikutukset tulevat kaikille kielen käyttäjille.

Tyyppijärjestelmät ovat nousseet suuremmaksi kiinnostuksen kohteeksi, koska dynaamisesti tyyppitetyt ohjelmat ovat saaneet pidemmän elinkaaren. Yksikkötestaus on auttanut, mutta se ei ole ollut riittävä takamaan hyvin toimivia ohjelmia. Javascriptin käyttö on lisääntynyt valtavasti ja on kehitetty Javascriptiksi kääntyviä kieliä, kuten Typescript ja ReasonML, jotka pyrkivät ratkaisemaan dynaamisen tyyppityksen ongelmia. Nämä uudet kielet mahdollistavat erilaisten tyyppijärjestelmien helpon käytön web-ohjelmissa, koska ne voidaan lisätä olemassa oleviin Javascript-ohjelmiin ilman kokonaisten ohjelmien uudelleen kirjoittamista.

Tyyppijärjestelmät on suunniteltu poistamaan virheitä, joita kutsutaan tyyppivirheiksi. Kuinka suuri luokka virheitä poistetaan, riippuu käytetystä tyyppijärjestelmästä ja siitä annetaanko ohjelmoijalle työkaluja, joilla hän voi rikkoa tyyppisääntöjä. Tässä tutkielmassa annetaan katsaus kolmeen erilaiseen tyyppijärjestelmään, jotka kattavat lähes kaikki käytössä olevat ohjelmat: täysin dynaamiset tyyppijärjestelmät, kuten Javascript, C:n kaltaiset, kuten Java ja ML:n kaltaiset, kuten Haskell.

Tässä tutkielmassa ei oteta kantaa, mikä tyyppijärjestelmä on paras, vaan annetaan esimerkkejä, mitä virheitä tyyppijärjestelmä pystyy poistamaan. ML:n tyyppijärjestelmä on kaikkein tarkin virheiden suhteen, mutta kuitenkin ML-pohjaiset kielet eivät ole yleistyneet teollisuudessa, joten ohjelmointikielen valintaan vaikuttavat muut asiat kuin tyyppijärjestelmän virheettömyys.

Avainsanat: ohjelmointikieliset, tyyppijärjestelmät, tyyppiturvallisuus, ML

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

Sisältö

1	Johdanto	1
2	Tyypijärjestelmät	2
2.1	Vahva tyypitys	2
2.2	Staatinn tyypitys	4
3	Tyypijärjestelmän turvallisuus	5
4	Tyypillisiä tyypivirheitä	7
4.1	Tyypipakotus	7
4.2	Taulukkokovarianssi	10
4.3	Taulukon ohi-indeksointi	11
4.4	Null-tyyppi	12
4.5	Null-optional	14
5	Keskustelu	16
6	Yhteenveto	17
	Lähteet	20

1 Johdanto

Tässä työssä vertaillaan ohjelmointikieliä tyyppijärjestelmän näkökulmasta ja esitetään, miten tyyppijärjestelmään kohdistuvat valinnat vaikuttavat, kuinka luotettava ohjelmointikieli on käytössä.

Tyyppijärjestelmän tarkoitus on poistaa tietynlaiset virheet kokonaan (Cardelli, 1989). Kielet, joissa ei ole tyyppijärjestelmää, kuten Assembly, eivät anna mitään järkevää palautetta, minkä takia ohjelma toimii virheellisesti, kun ohjelmoija yrittää suorittaa operaation vääranntyyppisillä arvoilla. Esimerkiksi mitä tarkoittaa merkkijonon jakaminen numerolla, ilman tyyppijärjestelmää tämä on täysin suoritettava operaatio, jonka käytös riippuu kielestä ja järjestelmästä, jossa ohjelma ajetaan. Muita käyttöjä tyyppijärjestelmälle on mm. abstraktointi, dokumentointi ja optimointi (Pierce, 2002).

Tyyppijärjestelmien merkitys on teollisuudessa nousut pinnalle viime vuosina, koska yhä monimutkaisempia ohjelmia on toteutettu viimeisen kahdenkymmenen vuoden aikana dynaamisesti tyyplitetyillä kielillä, kuten PHP, Ruby, Python ja Javascript, jotka mahdollistavat nopeamman ohjelmistokehityksen, mutta kääntäjän ja ennen ajoa tapahtuvan tyyppitarkistuksen puuttuminen aiheuttavat ongelmia monimutkaisten ohjelmien ylläpidossa ja kehityksessä, kuten Austin kertoo PHP:n käytöstä IMVU:lla (Austin, 2015). Teollisuuden ongelmia kuvaa myöskin Facebook:n kehittämä Hack kieli (Hack, 2020), joka korvaa PHP:n tyyplitetyllä variantilla.

Tutkimuskysymys johon pyritään vastaamaan on: Millaisia virheitä tyyppijärjestelmä pystyy poistamaan? Tähän pyritään vastaamaan esittämällä koodi, joka aiheuttaa tyyppivirheitä jossakin ohjelmointikielessä ja sama yritetään toistaa toisella ohjelmointikielellä, jossa on erilainen tyyppijärjestelmä. Tämä antaa hyvän kuvan, mitä eroja näissä järjestelmissä on, ja miten suuren luokan virheitä pystytään poistamaan automaattitarkistuksella. Tämä työ keskittyy vertailemaan kolmea luokkaa kielistä ML-perhe, C-tyyppiset ja dynaamisesti tyyplitetyt (kuten Lisp, Javascript ja Python). Tämä valinta tehtiin, koska se kattaa suurimman osan käytetyistä kielistä.

Menetelmänä on käytetty kirjallisuuskatsausta, jossa on käytetty julkaisuja ohjelmointikielistä ja tyyppiteoriasta. Tyyppiteoria ei ole muuttunut paljoakaan, joten se on suurimmalta osin kymmeniä vuosia vanhaa ja sen varmuus on vain vahvistunut. Tämän lisäksi on käytetty uusista trendeistä kertovia artikkeleita osoittamaan, miksi aihe on ajankohtainen. Ohjelmointikielten ja kirjastojen osalta lähdemateriaalina on käytetty kielen määrittäydokumenttia ja vastauksia kielen kehittäjiltä. Koodiesimerkit ovat tutkielman kirjoittajan tekemiä.

Aiheen ajankohtaisuudesta kertovat uudet kielet, jotka ovat korvanneet Javascript:n käyttöä, kuten Typescript (Typescript, 2020b), Dart (Dart, 2020) ja Elm

(Elm, 2020), sekä vapaaehtoisen tyyppityksen (gradual typing) lisäys Python kieleen (versiossa 3.4) (Python, 2020). Typescript on otettu käyttöön teollisuudessa erityisesti Google:n Angular2:n (Angular, 2020b, 2020a) ansiosta, joka on yksi suosituimmista Javascript frontend framework (TechMagic, 2020).

Asiaan liittyviä muita töitä on esimerkiksi Mikkosen gradu (Mikkonen, 2020), joka käsittelee Javascriptiksi kääntyvien kielten käyttöä web-kehityksessä. Mikkonen perehtyy enemmän Javascript:n ongelmiin ja miten uudet kielet, joissa on staattinen tyyppijärjestelmä, ratkaisevat niitä. Mikkosen gradussa käsitellyt kielet voidaan jakaa kahteen tässä tutkielmassa käytettyyn luokitukseen: ML-pohjaiset (Elm, ReasonML) ja C-pohjaiset (Typescript, Dart). Tässä työssä perehdytään enemmän C- ja ML- pohjaisten tyyppijärjestelmien eroihin, ja verrataan niitä dynaamisiin tyyppijärjestelmiin, kuten Javascript.

Vaikka tyyppijärjestelmä on vain pieni osa ohjelmointikieltä ja kielelle tarjolla olevat työkalut ja kirjastot ovat yleensä paljon tärkeämpi kriteeri kielen valintaan, tästä käsittelystä on hyötyä löytämään yleisiä ongelmia, mitä syntyy tyyppijärjestelmistä, miten niitä vältetään ja millaisia ratkaisuja muissa kielissä on käytetty.

Luku 2 määrittelee käsitteet heikosta, vahvasta, staattisesta ja dynaamisesta tyyppityksestä ja esittää miksi nämä jaot eivät ole hyödyllisiä. Luku 3 perehtyy tyyppi-varmuuden käsitykseen, joka antaa hyvän mittarin ohjelmointikielen tyyppijärjestelmälle. Luku 4 esittää esimerkkejä tyyppivirheistä vastaten kysymykseen: miksi ne esiintyvät tietyissä ohjelmistokielistä, mutta eivät toisissa. Luku 5 keskustellussa käydään läpi jatkamahdollisuuksia ja muita huomiota. Viimeiseksi luku 6 esittelee yhteenvedon aiheesta.

2 Tyyppijärjestelmät

Yleinen erottelu tehdään dynaamisesti ja staattisesti tyyppitettyjen ohjelmointikielten välillä, jossa staattisen tyyppityksen yleensä tunnistaa kääntäjän olemassa olosta ja dynaamisen voi kirjoittaa suoraan tulkkiin. Vahva ja heikko tyyppitys taas kertoo, miten virheet käsitellään. Karkeasti voisi sanoa, että heikosti tyyppitettyt eivät kerro miksi virhe tapahtui toisin kuin vahvasti tyyppitettyt. Staattista tyyppitystä on nykyään myös vapaaehtoisen tyyppitys, kuten Python, ja tyyppitarkistimet, kuten Flow Javascriptille. Nämä eivät käännä koodia, vain tyyppitarkastavat sen ilman, että sitä suoritetaan.

2.1 Vahva tyyppitys

Vahva tyyppitys vastaa kysymykseen, aiheuttaako tyyppivirheet tuntematonta käytöstä (undefined behaviour), esimerkiksi ohjelman kaatuminen. Heikosti tyyppitetyis-

sä kielissä, kuten C, tämä on yleistä, mutta vahvasti tyypitettyssä kielessä tyyppivirheet eivät saa aiheuttaa tuntematonta käytöstä. Esimerkiksi vahvoista tyypityksistä Java nostaa ajon aikana virheen, jos tyyppi on väärä, ja Haskell ei suostu kääntämään ohjelmaa, jossa tyypit eivät ole yksiselitteisiä.

Vahvan tyypityksen määrittäminen Liskovin mukaan on, että jokainen parametri jolla funktiota kutsutaan pitää olla tyypiltään yhteensopiva tyyppin, jolle funktio on määritelty, kanssa (Liskov & Zilles, 1974). Tietysti tämä ei ota mitään kantaa siihen, mitä tapahtuu, kun ne eivät ole yhteensopivia tai miten se estetään. Kieli on vahvasti tyypitetty, jos ohjelma ei suostu kutsumaan funktiota väärän tyyppisellä parametrilla, riippumatta siitä missä tarkistus tehdään. Tästä seuraa, että kaikki dynaamisesti tyypitettyt kielet, joissa funktion tyypit ovat aina objekteja ja tarkistus perustuu ajonaikaiseen muodon tarkistamiseen, ovat aina vahvasti tyypitettyjä.

Kieli, jossa ei ole joko staattista tai vahvaa tyypitystä, on vaikea ohjelmoida, koska ohjelmoijalla ei ole mitään tarkistusta ohjelman oikeellisuudesta ja jokainen virhe aiheuttaa tuntematonta toimintaa. Assembly on yksi niistä harvoista kielistä, joka kuuluu tähän ohjelmoijalle epämiellyttävään kategoriaan heikosti ja dynaamisesti tyypitetty (Cardelli, 1996).

Hahmotetaan heikon tyypityksen ongelmia lyhyellä C esimerkillä ohjelma 2.1. Esimerkki ohjelma on laillista C:tä, joka kuitenkin toimii virheellisesti ajonaikana, useimmiten ohjelma kaatuu, mutta riippuen ympäristöstä, jossa se ajetaan se saattaa myös jatkaa virheellistä suoritusta. Esimerkin ohjelma toimii, koska C antaa sijoittaa void-osoittimen mihin vain muuttujaan, eikä ohjelmoijan tarvitse tehdä eksplisiittisiä tyyppimuunnoksia. Tämä ominaisuus on yksi suurimpia virheiden aiheuttajia, ja vaikka C++ on edelleen heikosti tyypitetty, on se tässä tilanteessa turvallisempi, koska sen kääntäjä ei hyväksy tätä koodia. C++:n yhteensopivuus C:n kanssa, joka vaatii C-yhteensopivat tyyppimuunnokset antavat ohjelmoijalle työkaluja, joilla hän voi rikkoa tyyppiturvallisuuden, josta seuraa että C++ on heikosti tyypitetty.

```

1 char c;
2 void *p;
3 int *i;
4
5 c = 'f';
6 p = &c;
7 /* C allows implicit casting from void* */
8 i = p;
9 /* undefined behaviour */
10 *i = 1;

```

Ohjelma 2.1 Heikko tyypitys esimerkki. Muistin ohi indeksointi, koska int ei mahdu char muuttujaan.

Vahvasti ja heikosti tyyppitetyt on otettu esille täydellisyyden vuoksi, mutta tämä erottelu on harvoin hyödyllinen, koska järjestelmällisesti kaikki kielet, jotka on kehitetty 1980-luvun jälkeen, ovat vahvasti tyyppitettyjä. Ainoa mitä vaaditaan vahvalle tyyppitykselle on, että järjestelmä nostaa yksiselitteisen virheen jossain vaiheessa (yleensä kääntämisen tai ajon aikana) kaikista tyyppivirheistä. Yleisesti käytössä olevista kielistä ainoastaan Assembly ja C eivät tätä tee (Cardelli, 1996), joten keskitymme jatkossa tutkimaan, ovatko tyyppivirheet kääntämisen vai ajon aikaisia.

2.2 Staattinen tyyppitys

Kieli on staattisesti tyyppitetty, kun sen tyytit tarkistetaan ennen ohjelman suoritusta (Meijer & Drayton, 2004). Usein ajattelemme staattisen tyyppityksen olevan se, että muuttujien tyytit kirjoitetaan näkyviin. Tämä ei ole todellinen vaatimus, koska esimerkiksi pitkiäkin Haskell ohjelmia voi kirjoittaa ilman tyyppi-informaatiota. Staattisen vastakohta on dynaaminen tyyppitys. Taulukko 2.1 on esiteltynä yleisiä ohjelmointikieliä ja tehty jako tyyppijärjestelmiin sen perusteella, mitä kielen dokumentaatiossa sanotaan sen olevan.

Yleensä ajatellaan, että kielet kuuluvat jompaankumpaan luokkaan, kuten taulukko 2.1. Todellisuudessa tämä jako on huomattavasti häilyvämpi. Lisp on täysin dynaaminen ohjelmointikieli, mutta onko Typed Clojure staattinen vai dynaaminen, Clojure on Lisp variantti, joka on täysin dynaaminen, mutta Typed Clojure lisää staattisen tyyppitarkistuksen. Samoin Java on staattisesti tyyppitetty, mutta periytyminen Javassa on täysin dynaaminen ominaisuus, johon kääntäjä ei tee tyyppitarkistuksia. Onko Java sitten staattisesti tyyppitetty?

Taulukko 2.1 ohjelmointikielienten tyyppijärjestelät

Kieli	heikko/vahva	staaattinen/dynaaminen
Assembly	heikko	dynaaminen
C	heikko	staattinen
Java	vahva	staattinen
ML	vahva	staattinen
Haskell	vahva	staattinen
Lisp	vahva	dynaaminen
Python	vahva	dynaaminen
Javascript	vahva	dynaaminen

Isoissa projekteissa tyyppitarkistus on paljon tärkeämpi, kuten nähdään siitä että yritykset, kuten Microsoft ovat lisänneet tyyppityksen Javascriptiin Typescriptilla (Typescript, 2020b; Bierman, Abadi & Torgersen, 2014). Facebook kehitti oman Flow-tyypityksen Javascriptiin (Flow, 2020a) ja on käyttänyt sitä omissa ohjelmissaan, kuten Reactissa (React, 2020). Dropbox lisäsi tyyppityksen Pythoniin ottamalla

käyttöön Jukka Lehtosalon kehittämän mypy:n (mypy, 2020) palkkaamalla Lehtosalon ja tukemalla mypy:n jatkokehitystä. mypy on sittemmin integroitu viralliseen Pythoniin (Lehtosalo, 2020; Python, 2020).

Dynaamisiin kieliin jälkeinpäin lisättyä tyyppitystä kutsutaan vapaaehtoiseksi tyyppitykseksi (gradual typing / optional typing). Se poikkeaa staattisesta tyyppityksestä siinä, että tyyppi-informaatio ei ole pakollista, eikä meillä ole kääntäjää, joka pakottaisi tyyppityksen. Sen sijaan käytetään tyyppitarkistinta. Vapaaehtoinen tyyppitys on hyvä edistysaskel projekteihin, joissa on paljon dynaamisesti tyyppitettyä koodia ja antaa mahdollisuuden poistaa tyyppivirheitä ilman, että ohjelmaa tarvitsee kirjoittaa uudelleen jollain toisella kielellä. Tässä työssä ei tarkastella tarkemmin vapaaehtoisia tyyppitystä, mutta sitä koskevat samat säännöt ja huomautukset, kuin staattista tyyppitystäkin, sillä erolla, että se sallii koodin suorituksen, vaikka se ei menisi tyyppitarkastuksesta läpi.

Seuraavassa perehdytään turvallisuuden määrittämiseen ja siihen minkä takia suuri osa kielistä on dynaamisesti turvallisia kieliä. Esimerkiksi C-johdannaisille pätee, että vaikka kieli on staattisesti tyyppitetty, niin vahva tyyppitys on silti ajonaikana ratkaistava ongelma.

3 Tyyppijärjestelmän turvallisuus

Kaikki tyyppijärjestelmät vastaavat ongelmaan, miten poistetaan yksi iso yhtenäinen luokka virheitä. Tyyppijärjestelmien turvallisuus (typesystem safety/soundness) auttaa meitä ymmärtämään, mitkä kaikki virheet kuuluvat tuohon luokkaan. Tämä kertoo meille, mitä virheitä voimme olettaa tyyppijärjestelmän poistavan koodista eli mitä ei tarvitse testata ja voimme silti olla varmoja, että ohjelma toimii oikein (Cardelli, 1996; Meijer & Drayton, 2004).

Suuri osa kielistä, joita teollisuudessa käytetään, ovat dynaamisesti turvallisia, vaikka ne olisivatkin staattisesti tyyppitettyjä. Staattisesti tyyppitettyjen kielten kääntäjät pyrkivät estämään kaikki mahdolliset virheet, mutta on ominaisuuksia kuten dynaaminen muistinvaraus ja käyttäjän syöte, mille kääntäjä ei voi tehdä mitään. Erityisesti dynaamisesti varatut oliot ovat toistaiseksi aina kääntäjän voimien ulkopuolella, koska kääntäjä näkee ainoastaan muuttujan tyyppin, mutta muuttuja voi sisältää, minkä vain periytetyn tyyppin. Kielissä, kuten ML ja Haskell, ei ole oliota tai alityypitystä, joten vastaavaa ongelmaa ei synny, joten kääntäjä pystyy tarkistamaan kaikki tyytit.

Heikosti tyyppitetty kiellet, kuten C, eivät sisällä mitään tyyppi-informaatiota ajonaikana, ellei ohjelmoija ole sitä sinne tietoisesti lisännyt. Tämä on havaittu ongelmalliseksi, niin monet C-johdannaiset kielet ovat lisänneet ajonaikaisen tyyppi-informaation muuttamalla ne vahvasti tyyppitettyiksi. Vaikka nämä kielet käyttävät

staattista tyyppitystä, niiden turvallisuus on silti taattu dynaamisella, tyyppi-informaatiolla. Esimerkin ohjelma 3.1 alityypit tarkastetaan ajonaikana, kuten täysin dynaamisesti tyyppitetyissä kielissä (Lisp, Javascript, Python).

```

1 class Foo {}
2
3 class Bar : Foo {}
4
5 Foo x = new Bar;
```

Ohjelma 3.1 C#: kääntäjä luulee, että x muuttuja olisi tyyppiä *Foo*, mutta ajonaikana ohjelma tietää, että x on *Bar* tyyppiä.

Koska kääntäjällä ei ole tarpeeksi tietoa todellisesta tyyplistä, niin tyyppiturvallisuus näissä kielissä on lopulta ajonaikainen ominaisuus eli ne käyttäytyvät kuten dynaamisesti tyyppitetyt kielet. Kääntäjän hyödyllisyys näissä onkin poistaa pahimmat tyyppivirheet. Käytännössä tyyppi-informaatio on suoritettavassa ohjelmassa jokaisessa muuttujassa, ja ohjelma nostaa tyyppivirheen ajonaikana, kun ohjelma yrittää tehdä jotain, mikä ei ole sallittua. Tästä esimerkki on ohjelma 3.2.

```

1 class Foo {}
2 class Bar : Foo {}
3 class Baz {}
4
5 Bar bar = new Bar;
6 Baz baz = new Baz;
7 // Legal implicit cast
8 Foo y = bar
9 // Legal dynamic cast
10 Bar b = (Bar)x
11 // Runtime error: illegal cast
12 Bar b_ = (Bar)baz
```

Ohjelma 3.2 C#: ajonaikainen virhe dynaamisessa tyyppimuutoksessa.

Haskell ja ML eivät sisällä ajonaikaista tyyppitietoa, koska niiden käyttämä Hindley-Milner tyyppisysteemi ei sitä vaadi (Milner, 1978). ML-pohjaisten kielten oletetaan olevan täysin tarkistettu käännoaikana, joten tyyppivirheet eivät voi tapahtua ajonaikana. Siinä missä C-tyyppisissä kielissä on ajonaikainen virhe (esim. Java:n `NullPointerException`) ML käyttää non-exhausted-pattern virhettä, eli ohjelmoija ei ole ottanut huomioon kaikkia mahdollisia palautusarvoja mitä funktiosta voi tulla.

Tyyppiturvallisuus voidaan myös todistaa. Tämä on kiinnostava ominaisuus, koska jos kieli on todistetusti tyyppiturvallinen, niin kaikki ohjelmat, joita sillä voi tehdä ovat myös tyyppiturvallisia. Ainoastaan Standard ML on yleisesti käytössä olevista kielistä todistettu tyyppiturvalliseksi (Milner, 1978; Wright & Felleisen, 1994).

Standard ML ei ole teollisuudessa paljon käytetty, mutta siihen pohjattuja kieliä, kuten OcaML, F# (F#, 2020) ja Haskell (Wiki, 2020) ovat suosittuja. Standard ML -johdannaiset kielet eivät kuitenkaan ole todistettusti tyyppiturvallisiksi, koska ne lisäävät paljon ominaisuuksia, joita ei ole todistettu turvallisiksi. Erityisesti F# ja ReasonML eivät koskaan voi olla tyyppiturvallisiksi, koska ne integroituvat liian hyvin ei-turvallisiin-kieliin ilman, että ohjelmoija voisi vaikuttaa ei-turvallisen-koodin suoritukseen. Seuraavassa luvussa käydään läpi, mitkä yleisesti käytössä olevat ominaisuudet rikkovat tyyppiturvallisuutta.

4 Tyypillisiä tyyppivirheitä

Hyväksytään tässä, että tieto joka tulee ohjelman ulkopuolelta on aina tarkistettua, koska se on rajapintaongelma ei tyyppijärjestelmäongelma, vaikka laaja tyyppijärjestelmä, kuten ML:n tai Haskellin voi auttaa rajapinnan suunnittelussa. Joten keskitymme tässä tyyppivirheiksi, jotka syntyvät koodista eikä syötteestä.

Tyyppiturvallisuuden luku 3 yhteydessä esittelimme dynaamisen ja staattisen turvallisuuden. Käytämme tätä käsitettä tässä kappaleessa, kun katsomme yleisiä tyyppivirheitä ja luokittelemme ne sen mukaan.

4.1 Tyypipakotus

Ilman tyyppitystä kaikki samankokoiset muistialueet ovat valideja funktion parametreja. Virheet tapahtuvat ainoastaan muistin ohi-indeksoinnissa. Tyyppijärjestelmä lisää turvallisuutta rajoittamalla ohjelmoijan vapautta. Sivuvaikutus tästä turvallisuudesta on, että funktiota jonka suunnittelija on suunnitellut tietyille muuttujatyypeille ei voi kutsua millään muulla. Esimerkiksi funktio joka on suunniteltu reaalityypille ei voida kutsua kokonaisluvuilla. Tämä myös tekee tulostusfunktioden, jotka vaativat muuttujien muuttamisen merkkijonoiksi, luomisen monimutkaiseksi.

Tyypipakotus (type coercion) eli implisiitti tyyppimuunnos on, kun kääntäjä tai tulkki muuntaa arvon tyyppin yhdestä muodosta toiseen ilman, että ohjelmoijan tarvitsee tehdä mitään. Teoriassa tämä on hyvä ominaisuus, mutta kuten alla olevista esimerkeistä huomataan käytännössä, se aiheuttaa omia ongelmia.

C-tyylinen ratkaisu tähän ongelmaan on käyttää muistia ja makroja. Esimerkiksi printf ei ymmärrä tyypeistä mitään, vaan olettaa käyttäjän kertovan sille oikein muuttujan tyyppin ja koon parametrina. C myös tarjoaa tyyppimuunnokset, jotka ovat turvallisiksi arvoille mutta epäturvallisia viitteille. C:n isoin ongelma tyyppiturvallisuuden näkökulmasta on sen ekstensiivinen tyyppipakotusten käyttö. Erityisesti void viitteen automaattinen muuntaminen mihin vain tyyppiin.

Dynaamisesti tyyppitettyissä kielissä ainoastaan tiedon muodolla on väliä, koska

niissä ei ole ekstensiivistä tyyppijärjestelmää. Esimerkiksi Javascript objekti tyyppi tarkistaa vain, että muuttujan nimi on olemassa ja se voidaan implisiittisesti muuntaa siihen muotoon jota funktio tarvitsee. Tätä ajatusmallia kutsutaan ankka-tyypitykseksi (duck-typing), joka tarkoittaa ”jos se näyttää ankalta ja puhuu kuin ankka niin se on ankka”.

Ongelma pakotuksessa on kun muunnetaan tyyppisiä jotka eivät oikeaisti ole samanlaisia. Esimerkiksi ohjelma 4.1 on klassinen Javascript virhe, jonka jokainen Javascriptia ohjelmoinut on tehnyt. Tämä ominaisuus on määritetty ECMA 6.0 standardissa (Ecma, 2011), ja on siis tietoisesti tehty tyyppipakotus.

```
1 5 == '5' // true
```

Ohjelma 4.1 *Tyyppipakotus javascript:ssa: tyyppillinen virhe yhtäsuuruus operaattorin käytössä.*

Varmaan tunnetuin esimerkki tyyppipakotuksen käytöstä on C-kieli, joka on suunniteltu käyttämään niitä niin paljon, että jo muistinvaraus vaatii tyyppipakotuksen. Normaali tapa varata muistia C-kielessä ohjelma 4.2. Muistinvaraus palauttaa void-viitteen eli tyyppittömän viitteen. Tyyppipakotus tapahtuu kun tämä tyyppittämätön viite muunnetaan mihin vain tyyppiin automaattisesti. Koska C++-kääntäjä ei suostu tekemään tyyppipakotuksia, niin C-koodiin joka käyttää tyyppipakotuksia on lisättävä selvät tyyppimuunnokset ohjelma 4.3.

```
1 // malloc returns a void pointer not an int pointer
2 int *p = malloc(8*sizeof(int))
```

Ohjelma 4.2 *C tyyppipakotus, joka ei käänny C++ kääntäjällä.*

```
1 int *p = (int*)malloc(8*sizeof(int))
```

Ohjelma 4.3 *C++ versio 4.2.*

Yleisesti C-tyyliohjeet vastustavat tätä käytäntöä, koska se aiheuttaa virheitä myöhemmin.

Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it. (Spencer, 2020)

C++ kehittäessä Stroustrup:n yksi päätavoitteista oli tyyppiturvallisuus, niin pitkälti kun se ei riko yhteensopivuutta C:n kanssa (Stroustrup, 1996). Siinä ratkaisu tähän ongelmaan on sekä useammat versiot samasta funktiosta (procedure overloading) että tyyppimuunnokset, jotta tyyppiturvallisuus säilytetään, tyyppipakotukset on rajoitettu ainoastaan sellaisiksi, että ne eivät tuhoa tietoa esim. float -> double on ok, double -> float ei ole. Samoin myös sekä Java, että C# pyrkivät

tyyppiturvalliseen versioon C-kielestä, ja koska ne eivät vaadi yhteensopivuutta niin molemmat ovat tyyppiturvallisempia kuin C++.

Haskell ja ML ratkaisevat ongelman muuten samalla lailla kuin C++, mutta eivät sisällä tyyppipakotuksia lainkaan (O’Sullivan, Goerzen & Stewart, 2008). Sen sijaan ne käyttävät muuntajafunktiota, kuten Haskellin `show` ja `read` funktiot, jotka voidaan määrittää kaikille tyypeille ja muuntavat siitä tyyppistä merkkijonoksi tai merkkijonosta siksi tyyppiä.

ML ei sisällä mitään tyyppipakotuksia, joten reaalityyppien lisääminen kokonaislukuihin ei onnistu ohjelma 4.4, vaan se pitää kirjoittaa explisiitisti ohjelma 4.5. Haskellissa vastaava koodi toimii ohjelma 4.6, koska Haskell käyttää geneeristä `Num` tyyppiluokkaa (`type class`) (Jones et al., 1999), joka mahdollistaa matemaattiset operaatiot eri tyyppien välillä ilman tyyppipakotusta. Jos ohjelmoija pakottaa tyyppiannotaatiolla kääntäjän käyttämään `Int` tai `Double` tyyppiä, kuten ohjelma 4.7, niin Haskell toimii kuin ML, ja matemaattiset operaatiot eivät onnistu ilman tyyppimuunnosta.

```
1 val x = 1.2 + 1
2 (* error operator overload conflict *)
```

Ohjelma 4.4 ML reaalityyppien yhteenlasku ei toimi kokonaisluvuille.

```
1 val x = 1.2 + 1.0
```

Ohjelma 4.5 ML oikea tapa kirjoittaa reaalityyppien yhteenlasku.

```
1 x = 1.2 + 1
2 -- x :: Num
```

Ohjelma 4.6 Haskell reaali- ja kokonaisluvun yhteenlasku `Num` tyyppiluokalla.

```
1 x = (1 :: Int) + (1.2 :: Double)
2 -- Couldn't match expected type 'Int' with an actual type 'Double'
```

Ohjelma 4.7 Haskell reaali- ja kokonaisluvun yhteenlasku epäonnistuu ilman `Num` tyyppiluokkaa.

Haskell toimii lopusta alkuun sekä arvon- että tyyppien määrittämisessä. Kääntäjä ottaa viimeisen paikan, jossa arvoa tarvitaan (sijoitus), siitä seuraava funktio on yhteenlasku (+). Yhteenlaskua varten molemmat parametrit pitää olla samaa tyyppiä tai tyyppiluokkaa, joista `Num` tyyppiluokalle on määritetty yhteenlasku ja molemmat reaali- ja kokonaisluku kuuluvat `Num` tyyppiluokkaan, joten funktio on laillinen ja palautusarvo on `Num` tyyppiluokkaa.

```
1 (+) :: Num a => a -> a -> a
```

Ohjelma 4.8 Haskell yhteenlasku funktion määrittelmä.

Vaikka $\forall x \in N \Rightarrow x \in R$ pitää matemaattisesti totta, niin esimerkki ohjelma 4.7

ei pysty selvittämään, että tarvittava tyyppi on `Double`. Tämä johtuu siitä miten funktiot on määritelty Haskellissa, kun katsomme ohjelma 4.8 yhteenlaskun määritelmää, niin syy selviää tyypeistä. Funktio ottaa sisäänsä kaksi parametria, jotka ovat samaa tyyppiä ja palauttaa samaa tyyppiä, koska tyypit ovat eri, niin kääntäjä ei osaa sanoa onko tyyppi, mitä ohjelmoija tarkoitti `Double` vai `Int`. Tyypivirhe tässä seuraa siitä, että kääntäjä ei ymmärrä, mitä ohjelmoija haluaa tehdä, eikä siksi voi ratkaista ongelmaa yksiselitteisesti.

4.2 Taulukkokovarianssi

Hyvin monet kielet tukevat alityypitystä (subtyping), kuten olio-ohjelmointikielet ja osa ML-pohjaisista kielistä, kuten OCaml. Standard ML, millä tämän tutkielman esimerkit on kirjoitettu, ei tue alityypitystä. Alityypitys oliokielissä vastaa periyttämistä, mutta se on laajempi käsite ja voidaan toteuttaa muissakin kuin oliokielissä.

Haskell ei tue alityypistä missään muotoa. Hall ja muut antaa tähän syyksi version 1.1 tehokkuuden (Hall et al., 1996). Kun tyyppiluokat lisättiin Haskellin ratkaisuun operaattorien, kuten yhtäsuuruus ja merkkijonoksi muuttamisen, niin tarve alityypitykselle väheni. Alityypitys ja tyyppiluokat ratkaisevat hyvin samanlaisen ongelman, mutta ovat toteutukseltaan hyvin erilaiset (Cook, 2009). Ne tuodaan tässä esille kuvaamaan, miksi alityypitystä ei ole nähty tarpeelliseksi Haskellissa.

Kovarianssi kertoo, miten alityypin kompositiota käsitellään, eli jos kieli on tyyppikovariantti ja Kissa on Eläimen alityyppi niin `List<Kissa>` on `List<Eläin>` alityyppi. Kovarianssi yleisesti keskittyy mihin vain tyyppikompositioon, mutta yleensä kompositiotyyppi on taulukko tai lista joten kutsumme tätä ominaisuutta taulukkokovarianssiksi. Esimerkiksi seuraavalla algoritmilla saadaan aikaiseksi ajonaikainen tyypivirhe suurimmassa osassa oliokieliä:

1. Määritellään abstrakti tyyppi ja kaksi periytettyä tyyppiä A ja B .
2. Varataan taulukko A tyyppiä.
3. Otetaan abstraktin tyyppin viite taulusta.
4. Yritetään asettaa abstraktilla viitteellä taulukkoon tyyppiä B oleva alkio.

Tällä algoritmilla saadaan ohjelma kaatumaan ajonaikana, ja tätä on mahdollista siirtää kääntäjä virheeksi, koska periytettyjen tyyppien tyyppi-informaatio ei ole tarjolla kääntäjälle.

Taulukkokovarianssi virheen toteutus vaatii seuraavat ominaisuudet kieleltä:

1. periyttäminen ja ajonaikainen tyypintarkistus periytetyille tyypeille
2. taulukon, joka sisältää abstrakteja olentoja, muokkaus.

Ongelman voisi korjata kieltämällä abstraktin taulukon muokkauksen. Varsinkin kun taulukkokovarianssi ei ole suuressa käytössä, koska sen lisäksi että se rikkoo tyyppiturvallisuuden, se on hidas, koska abstraktityypin asetus vaatii dynaamisen tyyppimuutoksen. Kuitenkin molemmat Java ja C# rikkovat tyyppiturvallisuuden tällä ominaisuudella. Esimerkiksi ohjelma 4.9 toteuttaa taulukkokovarianssi virheen, jossa ohjelma kaatuu ajonaikana tyyppivirheeseen. Javalla on myös mahdollista toteuttaa aivan sama asia.

```
1 Giraffe[] giraffes = new[] { new Giraffe() };
  Animal[] animals = giraffes; // This is legal!
3 animals[0] = new Tiger(); // crashes at runtime with a type error
```

Ohjelma 4.9 Esimerkiksi laillista C# koodia

4.3 Taulukon ohi-indeksointi

Taulukon ohi-indeksointi on tyypillinen virhe, joka tapahtuu, kun ohjelmoija yrittää käyttää taulukon muistipaikkaa, jota ei ole olemassa, esimerkiksi ohjelma 4.10. C ja C++ ohjelmat kaatuvat tähän virheeseen (heikko tyyppitys), mutta uudemmat kielet, kuten Java heittävät ajonaikaisen virheen (vahva tyyppitys).

```
1 int [] arr = new int [2]
2 // runtime exception
3 arr [2]
```

Ohjelma 4.10 Java taulukon ohi-indeksointi

Ratkaisuja tähän ongelmaan on kehitetty, kuten C++ iteraattorit ohjelma 4.11, jotka mahdollistavat yhtenäisen syntaksin indeksoinnille. Vaikka niiden käyttö on suoraviivaisempaa, niin iteraattorit voivat kuitenkin aiheuttaa ajonaikaisia tyyppi-
virheitä. Samoin Pythonin for in -syntaksi ohjelma 4.12 estää ohi-indeksoinnin kun kokonaista taulukkoa käsitellään. Molemmat ratkaisut poistavat tietyt virheet, kuten indeksoinnin aloittamisen ykkösestä, mutta taulukot voivat silti aiheuttaa ajonaikaisia virheitä kun niitä käytetään silmukoiden ulkopuolella.

```
1 for (std::vector<int>::iterator iter = array.begin();
2     iter != array.end(); ++iter) {
3 }
```

Ohjelma 4.11 C++ Iteraattori

```
1 for obj in collection:
2     pass
```

Ohjelma 4.12 Python for in -syntaksi

Taulukon ohi-indeksointi on myös ML-pohjaisissa kielissä, vaikka niissä ei ole taulukkoja vaan listoja, joita käsitellään rekursiivisilla algoritmeilla. Ohjelma 4.13

on esimerkki ohjelmasta, joka ohi-indeksoi. Perinteinen tyyppijärjestelmällä ei pysty tätä ratkaisemaan mitenkään kääntämisen aikana, koska se vaatii tietoa listan koosta. Siksi tästä syntyy lähes aina ajonaikainen tyyppivirhe.

```

1 last :: [a] -> a
2 last (x:[]) = x
3 last (x:xs) = last xs

```

Ohjelma 4.13 Haskell last funktio

Ohjelma 4.13 on naiivi toteutus last-funktiolle Haskellilla, eli funktiolle joka palauttaa listan viimeisen alkion. Koodi kääntyy Glasgow-kääntäjällä ilman varoituksia ja toimii täydellisesti, kunnes sitä käyttää tyhjällä listalla, kuten ohjelma 4.14.

```

1 last [1, 2, 3, 4]
2 -- 4
3 last [1]
4 -- 1
5 last []
6 -- Exception : Non-exhaustive patterns in function call

```

Ohjelma 4.14 Haskell listan ohi-indeksointi

Jotta tyyppijärjestelmä pystyy ratkaisemaan tämän ongelman, meidän täytyy laajentaa se käsittämään myös argumenttien pituuksia, ei vain niiden tyyppiä, eli tässä tapauksessa lista, mitä vain tyyppiä kunhan se ei ole tyhjä. Tätä kutsutaan riippuvuustyypeiksi (dependent types), eli laajennamme tyyppijärjestelmän kattaamaan myös parametrin arvon ei pelkästään parametrin tyyppiä (Xi & Pfenning, 1999). Riippuvuustyyppit ovat aktiivisen tutkimuksen kohteena, mutta ne eivät ole tuettuja yleisesti käytössä olevissa kielissä. Erityisesti kumpikaan Haskell tai Standard ML ei tue niitä, vaikka Haskellin löytyykin lisäys niitä varten. Tarkempi perehtyminen näihin vaatisi tutkimuskäytössä olevan kielen, kuten Idris, Agda, Coq tai Epigram:n, käyttöä.

4.4 Null-tyyppi

Tony Hoare kuvaili null-tyyppiä, jonka hän lisäsi Algol W -kieleen 1965, Miljar-din dollarin virheenä (Hoare, 2009), vaikka rahallista arvoa on vaikea arvioida, niin null-viitteet ovat yksi suurimmista pahojen virheiden aiheuttajista ohjelmissa siitä lähtien. Null tyyppi on nyt lähes kaikissa ohjelmointikielissä, samanlaisena kuin se alunperin oli Algolissa.

Kielissä kuten Haskell ja Standard ML null (tai None) on tyyppi kuten kaikki muutkin, eikä niissä ole implisiittiiä null-tyyppiä kaikissa viitteissä (Milner, Tofte & Harper, 1990; Jones et al., 1999). Fortran, Algol 68 ja C johdannaisissa kielissä mikä vain viite voi olla null-tyyppinen. Siis null-tyyppi on erikoistapaus, joka lisätään kaikkiin viitteisiin eli kaikki viitteet ovat määritelty katkonaisena tyyppinä

(disjointed type) seuraavasti

$$\text{Ref} < F > = F \sqcup \{\text{null}\} \wedge \text{null} \notin F \quad (4.1)$$

Jokainen viite on muotoa kaava (4.1) eli se muodostuu todellisen tyyppin F ja erikoistyyppin null yhdisteestä. Null itsessään ei ole oikea viite vaan pseudoviite. Vastaava toteutetaan pseudokoodilla ohjelma 4.15.

```
1 RefType<Type> = Type | null
```

Ohjelma 4.15 Pseudokoodi: katkonainen tyyppi

Kielissä kuten Haskell ja ML, vastaava tyyppi voidaan määrittää käyttämällä kielen omaa tyyppijärjestelmää, eikä sitä lisätä automaattisesti kaikkiin viitetyyppeihin, jolloin se noudattaa kielen kaikkia tyyppisääntöjä ja kääntäjä pystyy käsittelemään sen ohjelma 4.16.

```
1 datatype tyyppi = OLEMASSA | NONE
2
3 fun f (OLEMASSA) = 1
4   | f (NONE) = 0
5
6 val x = f OLEMASSA (* x = 1 *)
7 val y = f NONE      (* y = 0 *)
```

Ohjelma 4.16 Standard ML versio C tyyppinen referenssi

Implisiitin null-tyypin ongelma on, että ohjelmoijat joutuvat lisäämään tarkistuksia kaikkialla, kuten ohjelma 4.17 ja ohjelma 4.18; jos tätä ei tee ohjelma kaatuu tyyppivirheeseen (Fähndrich & Leino, 2003). Vastaavat ajon aikaiset tarkistukset tarkoittavat, että kääntäjä ei voi tehdä mitään ohjelman oikeellisuuden varmistamiseksi ja vähentävät kääntäjän mahdollisuutta optimoida ohjelmaa. Fähndrich ja Leino kuvailee, miten null-tyyppi voidaan poistaa imperatiivista kielistä kuten C# ja Java (Fähndrich & Leino, 2003).

```
1 int f(char *str) {
2   if (str == NULL) {
3     return -1;
4   }
5 }
```

Ohjelma 4.17 C null-tarkistus

```

1 try {
2   f (obj)
3 } catch (NullPointerException &e) {
4   print("Object can't be null")
5 }

```

Ohjelma 4.18 Java : null-tarkistus try catch

Imperatiivisissa kielissä yleensä määritellään null-tyyppi seuraavasti: ”Osoitin joka ei osoita validiin muistipaikkaan”. Niin määritelmän mukaan null-tyypin dereferenssointi aiheuttaa aina ajonaikaisen virheen. Teknisesti muistipaikan ei tarvitse olla invalidi (esim. Lispissä), vaan se voi olla globaali muistialue, joka indikoi virhetilannetta, mutta se on aina virhetilanne ja sitä ei voida selvittää kääntämisenä aikana (Fähndrich & Leino, 2003). Tästä seuraten kielet, jotka sisältävät null-tyypin, eivät ole tyyppiturvallisista tältä osalta.

4.5 Null-optional

Fähndrich esittää (Fähndrich & Leino, 2003) tavan poistaa implisiitti null-tyyppi olio-ohjelmointi kielistä. Tämä on sittemmin toteutettu sekä C# versiossa 8 (C#, 2020), Typescriptissa (Typescript, 2020a) ja Flow:ssa (Flow, 2020b) että muissa oliokielissä.

Toisin kuin vanhemmissa versiossa C#, C:ssä, C++ tai Java:ssa normaali viitetyyppi ei voi sisältää null-tyyppejä. Näissä kielissä on mahdollista ohjelmoida kuten aikaisemmin ja vanhan koodin saa toimimaan lisäämällä hieman syntaksia (yleensä kysymysmerkki tyyppitykseen) kuten ohjelma 4.19. Tämä tarjoaa mahdollisuuden ohjelmoijille siirtää ison osan ajonaikaisia virheitä kääntäjän ongelmaksi ja parantaa ohjelman laatua ilman, että siirrymme lähes drakoniseen tyyppijärjestelmään, jota ML ja Haskell käyttävät.

```

1 int? maybeNull = 2
2 int notNull = 2
3
4 // ok
5 maybeNull = null
6 // compiler error
7 notNull = null

```

Ohjelma 4.19 C# nullable viitteet

Typescript tarjoaa samanlaisen menetelmän null-viitteille. Vaihdettaessa ohjelma 4.20 Javascriptistä Typescriptiin, niin vastaava koodi ohjelma 4.21 ei enää mene Typescript kääntäjästä läpi, eikä koskaan päädy ajettavaksi, koska Typescript viitteet eivät normaalisti voi olla null-tyyppisiä. Tämä parantaa sekä dokumentaatiota, kun ohjelmoija tietää koska null on sallittu, että ohjelman laatua ja vähentää

testattavaa, koska null-tyyppiä ei tarvitse testata.

```

1 const f = (a) => {
2   print(a.bar)
3 }
4
5 // Runtime null error
6 f(null)

```

Ohjelma 4.20 Javascript ajonaikainen null-virhe

```

1 const f = (a : {bar : string} ) => {
2   print(a.bar)
3 }
4
5 // Compiler error; f doesn't allow null type
6 f(null)

```

Ohjelma 4.21 Typescript viitteet eivät ole automaattisesti null

Typescriptin ongelma on se, että sen tyyppitys ei ole turvallinen kuten seuraavasta esimerkistä selviää. Tämä aiheuttaa ongelmia, koska Typescript mahdollistaa sekä tyyppisääntöjä rikkovan Typescriptin, että Javascript koodin sisällyttämistä, joten ohjelmoija ei voi olettaa Typescript ohjelman olevan koskaan tyyppiturvallinen. Eli pahimmassa tapauksessa Typescript koodi vaatii saman määrän testausta kuin vastaava Javascript koodi.

Typescriptin tyyppiturvallisuus rikotaan käyttämällä any tyyppiä ohjelma 4.22, joka on suunniteltu olemassa olevien Javascript-ohjelmien kivuttomaan siirtymiseen Typescriptiin. Tosin any tyyppistä seuraa, että mikä tahansa Typescript ohjelma, joka sisältää yhdenkin any tyyppisen muuttujan ei voida olettaa olevan tyyppiturvallinen.

```

1 interface foo {
2   bar : string
3 }
4 const f = (a : foo) => {
5   conso(a.bar)
6 }
7
8 let a : any = null
9
10 // casting away the typechecking -> Runtime error
11 f (a as foo)

```

Ohjelma 4.22 Typescript ei ole tyyppiurvallinen, koska ohjelmoija voi heittää tyyppityksen roskikseen kun siltä tuntuu.

5 Keskustelu

Tutkielmassa pyrittiin vastaamaan kysymykseen: Millaisia virheitä tyyppijärjestelmä pystyy poistamaan? Tähän löydettiin yksittäistapauksia yleisistä virheistä ja käsiteltiin missä määrin eri ohjelmointikielten tyyppijärjestelmät pystyvät estämään näitä virheitä. Täten tutkielma tarjosi katsauksen ongelmiin, mitä tyyppijärjestelmät ratkaisevat.

Työn rajausten takia syvällisempi perehtyminen tyyppijärjestelmien teoriaan ja tyyppivirheiden luokitteluun on jätetty pois. Jatkon kannalta teoreettisempi perehtyminen ML:n ja C:n tyyppijärjestelmien eroihin olisi hyödyllinen, mutta erityisesti ML vaatii paljon syvällisemmän perehtymisen tyyppiteoriaan kuin tässä on ollut mahdollista.

Myöskin tämä tutkielma ei käsitelty syitä, miksi ML-pohjaiset kielet eivät ole saaneet yhtä suurta käyttäjäkuntaa kuin C-pohjaiset. Tämä olisi tärkeää, jos halutaan ymmärtää, miksi teollisuudessa tyyppiturvallisuutta ei ole nähty kriittisenä ominaisuutena ohjelmointikielissä.

Jatkossa olisi hyvä tarkastella tarkemmin gradual tyyppitys systeemien hyötyjä verrattuna täysin staattiseen tyyppitykseen Javascript eco-systeemissä, kuten ReasonML, TypeScript ja Flow vertailulla. Työn laajuuden takia pois suljettiin myös muita uusia mielenkiintoisia kieliä ja varianteja, kuten Typed Clojure, Swift, Go, Dart ja Elm. Suurimmaksi osaksi nämä kuuluvat isompien kategorioiden sisälle, kuten Go on C-variantti ja Elm on ML-variantti, mutta niissä on kaikissa omat erikoisuutensa. Esimerkiksi Dart ratkaisee saman ongelman kuin Typescript, mutta siinä on täysin turvallinen tyyppijärjestelmä toisin kuin Typescriptissa.

Tässä työssä ei myöskään otettu kantaa tyyppipolymorfismisiin, kuten dynaamisten kielten anka-tyypitykseen (duck-typing) tai parametriseen polymorfiaan, jonka Hindley-Milner tyyppijärjestelmä toteuttaa. Käsittely näiden kahden välisestä eroista on tärkeä, mutta sen käsittely olisi poistanut työn muista osiosta. Asiasta kiinnostuneille Strachey (Strachey, 1967/2000) tarjoaan kattavan käsittelyn asiaan.

Mielenkiintoiseksi jatkokysymykseksi jää, missä määrin projektin koko tai ikä vaikuttaa tarvittavaan tyyppijärjestelmään. Anekdoottien, kuten (Austin, 2015), perusteella jossain kohtaan projektia dynaaminen tyyppitys muodostuu niin isoksi ongelmaksi ja muutokset tulevat niin työläiksi, että niitä ei enään voida tehdä. Samoin se miten paljon resursseja isot yritykset kuten, Facebook ja Microsoft ovat laittaneet staattiseen tyyppitykseen Javascript eco-systeemissä kertoo, että he uskovat sen olevan tärkeä heidän omissa projekteissa. Meillä ei kuitenkaan ole mitään selkeää käsitystä tai mittareita siitä, minkälaisessa tai kokoisessa projektissa haitat dynaamisesta tyyppityksestä tulevat isoksi ongelmaksi.

Vaikkakin esimerkit staattisesti tyyppiturvallisista kielistä (Standard ML ja Haskell) ovat täysin funktionaalisia, ei tyyppiturvallisuus kuitenkaan vaadi funktionaalista kieltä. Imperatiivisissa kielissä ongelmaksi muodostuu periyttäminen ja null viitteet, jotka kuitenkin jossain määrin on ratkaistu esimerkiksi C# kielessä ja sen yhteensopivuudessa F#:n kanssa.

6 Yhteenveto

Tässä tutkielmassa on näytetty, mitä ongelmia tyyppijärjestelmät on tarkoitettu ratkaisemaan. Esiteltiin karkea jako erilaisista ohjelmointikielistä, joka auttaa ymmärtämään, mitä tyyppivirheitä ohjelmoija voi olettaa valitun kielen korjaavan ja miten. Esitettiin tyyppiturvallisuuden käsite, jota havainnollistettiin erilaisilla esimerkeillä, missä ja miten ohjelmointikieliet rikkovat turvallisuutta.

Staattinen tyyppiturvallisuus on noussut selkeästi tärkeäksi asiaksi ohjelmointikielen valinnassa. Ratkaisu vaihtoehdot staattisen tyyppityksen toteuttamiseen ovat kuitenkin hyvin moninaiset. Olemassa olevaan dynaamiseen tyyppitykseen voidaan hitaasti lisätä vapaaehtoinen tyyppitys, jonka isoin etu on se, että se ei vaadi projektin uudelleen kirjoitusta. Vastaavasti sen isoin haitta tulee sen edusta eli se ei takaa staattista tyyppivarmuutta, koska se on suunniteltu lisättäväksi dynaamisesti tyyppitettyyn ohjelmaan. Esimerkkejä näistä ovat Flow ja mypy. Varsinkin jos tähän listaan lisätään Typescript, joka on välimuoto ratkaisu, niin nämä ovat varmasti nopeiten yleistyvät, koska niiden käyttöönotto on helppoa.

Uusille projekteille usein parempi ratkaisu olisi lähteä täysin staattisesta tyyppityksestä. Näistä meillä on kaksi kilpailevaa vaihtoehtoa C-tyyppiset, kuten Java, ja ML-tyyppiset, kuten ReasonML ja Elm. Huomattavaa on, että C-tyyppiset ovat olleet lähes ainoa suuressa käytössä oleva tyyppijärjestelmä, mutta ML on viimeaikoina herättänyt kiinnostusta teollisuudessa. Isoin ero näiden välillä on tyyppijärjestelmän monimutkaisuus. ML on monimutkaisempi ja vaikeampi oppia, mutta huomattavasti turvallisempi, kun taas C-tyyppinen on paljon helpompi valinta jo pelkästään suuren käyttäjäkunnan takia.

Lähteet

- Angular (2020a). *Source code*. URL: <https://github.com/angular/angular> (viitattu 17. 03. 2020).
- (2020b). *Typescript*. URL: <https://angular.io/guide/typescript-configuration> (viitattu 17. 03. 2020).

- Austin, Chad (2015). *The Long-Term Problem With Dynamically Typed Languages*. URL: <https://chadaustin.me/2015/04/the-long-term-problem-with-dynamically-typed-languages/> (viitattu 13.04.2020).
- Bierman, Gavin, Martín Abadi ja Mads Torgersen (2014). ”Understanding typescript”. Teoksessa: *European Conference on Object-Oriented Programming*. Springer, s. 257–281.
- C# (2020). *Specification*. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction> (viitattu 17.03.2020).
- Cardelli, Luca (1989). ”Typeful programming.” Teoksessa: *Formal description of programming concepts*, s. 431.
- (1996). ”Type systems”. *ACM Computing Surveys (CSUR)* 28.1, s. 263–264.
- Cook, William R (2009). ”On understanding data abstraction, revisited”. Teoksessa: *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, s. 557–572.
- Dart (2020). *Homepage*. URL: <https://dart.dev/> (viitattu 13.04.2020).
- Ecma, TC39 (2011). *Standard ECMA-262 ECMAScript Language Specification*.
- Elm (2020). *Homepage*. URL: <https://elm-lang.org/> (viitattu 13.04.2020).
- F# (2020). *Testimonials*. URL: <https://fsharp.org/testimonials/> (viitattu 17.03.2020).
- Flow (2020a). *A Static Type Checker for Javascript*. URL: <https://flow.org/> (viitattu 17.03.2020).
- (2020b). *Flow: Maybe Type*. URL: <https://flow.org/en/docs/types/maybe/> (viitattu 17.03.2020).
- Fähndrich, Manuel ja K Rustan M Leino (2003). ”Declaring and checking non-null types in an object-oriented language”. Teoksessa: *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, s. 302–312.
- Hack (2020). *Language*. URL: <https://hacklang.org/> (viitattu 13.04.2020).
- Hall, Cordelia V et al. (1996). ”Type classes in Haskell”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.2, s. 109–138.
- Hoare, Tony (2009). *InfoQ presentation: Null References: The Billion Dollar Mistake*. URL: <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/> (viitattu 17.03.2020).
- Jones, Simon Peyton et al. (1999). *Haskell 98 report, February 1999*. URL: <http://haskell.org/onlinereport> (viitattu 17.03.2020).
- Lehtosalo, Jukka (2020). *Our Journey to Type Checking 4 Million Lines of Python*. URL: <https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python> (viitattu 17.03.2020).

- Liskov, Barbara ja Stephen Zilles (1974). ”Programming with abstract data types”. *ACM Sigplan Notices* 9.4, s. 50–59.
- Meijer, Erik ja Peter Drayton (2004). ”Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages”. Teoksessa: Citeseer.
- Mikkonen, Juuso (2020). ”Statically typed programming languages in the JavaScript ecosystem: A type system perspective”. en. G2 Pro gradu, diplomityö, s. 73. URL: <http://urn.fi/URN:NBN:fi:aalto-202001261829>.
- Milner, Robin (1978). ”A theory of type polymorphism in programming”. *Journal of computer and system sciences* 17.3, s. 348–375.
- Milner, Robin, Mads Tofte ja Robert Harper (1990). *The Definition of Standard ML*. MIT Press.
- mypy (2020). *Optional Static Typing for Python*. URL: <http://mypy-lang.org/> (viitattu 17.03.2020).
- O’Sullivan, Bryan, John Goerzen ja Donald Bruce Stewart (2008). *Real world haskell: Code you can believe in*. ”O’Reilly Media, Inc.”
- Pierce, Benjamin C. (2002). *Types and programming languages*. MIT press.
- Python (2020). *Gradual typing PEP*. URL: <https://www.python.org/dev/peps/pep-0484/> (https://www.python.org/dev/peps/pep-0484) (viitattu 17.03.2020).
- React (2020). *React source code*. URL: <https://github.com/facebook/react> (viitattu 17.03.2020).
- Spencer, Henry (2020). *Ten commandments for C Programmers*. URL: http://doc.cat-v.org/henry_spencer/ten-commandments (viitattu 31.03.2020).
- Strachey, Christopher (1967/2000). ”Fundamental concepts in programming languages”. *Higher-order and symbolic computation* 13.1-2, s. 11–49.
- Stroustrup, Bjarne (1996). ”A history of C++ 1979–1991”. Teoksessa: *History of programming languages—II*, s. 699–769.
- TechMagic (2020). *React vs Angular vs Vue.js — What to choose in 2020?* URL: <https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d> (viitattu 07.06.2020).
- Typescript (2020a). *Handbook: Basic Types*. URL: <https://www.typescriptlang.org/docs/handbook/basic-types.html> (viitattu 06.06.2020).
- (2020b). *Homepage*. URL: <https://www.typescriptlang.org/> (viitattu 17.03.2020).
- Wiki (2020). *Haskell Used by Companies*. URL: <https://github.com/erkmos/haskell-companies> (viitattu 17.03.2020).
- Wright, Andrew K ja Matthias Felleisen (1994). ”A syntactic approach to type soundness”. *Information and computation* 115.1, s. 38–94.

Xi, Hongwei ja Frank Pfenning (1999). "Dependent types in practical programming". Teoksessa: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, s. 214–227.