JavaScriptin staattinen tyypittäminen

LuK -tutkielma
Turun yliopisto
Tulevaisuuden teknologioiden laitos
Tietojenkäsittelytiede
2017
Oskari Noppa

Sisältö

Luku 1
Johdanto

Ohjelmointi on pohjimmiltaan tietorakenteiden käsittelyä ja yksi tärkeimmistä tietorakenteen ominaisuuksista on sen tyyppi. Muuttuja “nimi” voi olla datatyypiltään teksti ja muuttuja “ikä” voi olla numero, eikä näitä kahta voi huolettomasti sekoittaa. Ohjelman tila ei ole järkevä jos se sanoo henkilön iän olevan “Matti”. Se miten ohjelmointikielissä käsitellään arvojen tyyppejä vaihtelee kuitenkin suuresti.

Tässä tutkielmassa käsitellään JavaScriptiä, sekä kolmea työkalua jotka lisäävät rakentavat staattisesti tarkastettavan tyyppijärjestelmän JavaScriptin päälle. JavaScript on dynaamisesti tyypitetty ohjelmointikieli, jonka alkuperäinen käyttötarkoitus oli lisätä verkkosivuille pieniä interaktiivisia ominaisuuksia, kuten lomakkeiden validointia. JavaScriptillä toteutettavien ohjelmien koko, monimutkaisuus ja tärkeys on kuitenkin viime vuosien aikana kasvanut alkuperäistä tarkoitusperää suuremmaksi, kun sillä on alettu toteuttaa esimerkiksi kartta-, kirjoitus- ja hallintapalveluita jotka toimivat selaimessa, siten ettei käyttäjän tarvitse asentaa erillistä ohjelmaa palvelun käyttöön.

Tutkielmassa esitellään TypeScript, Flow ja Closure-kääntäjä, joista jokainen on kehitetty työkaluksi parempien JavaScript-ohjelmien kehittämiseksi. Tarkastelussa ilmenee, että staattinen tyypitys voi nopeuttaa ohjelman kehittämistä, vähentää ohjelmoijan tekemien virheiden määrää ja parantaa valmiin ohjelman tehokkuutta. Toisaalta nähdään myös, että valinta staattisen ja dynaamisen tyypityksen välillä sisältää kompromisseja, etenkin kun tyyppijärjestelmä on erillinen työkalu eikä kieleen alusta asti kehitetty ominaisuus.

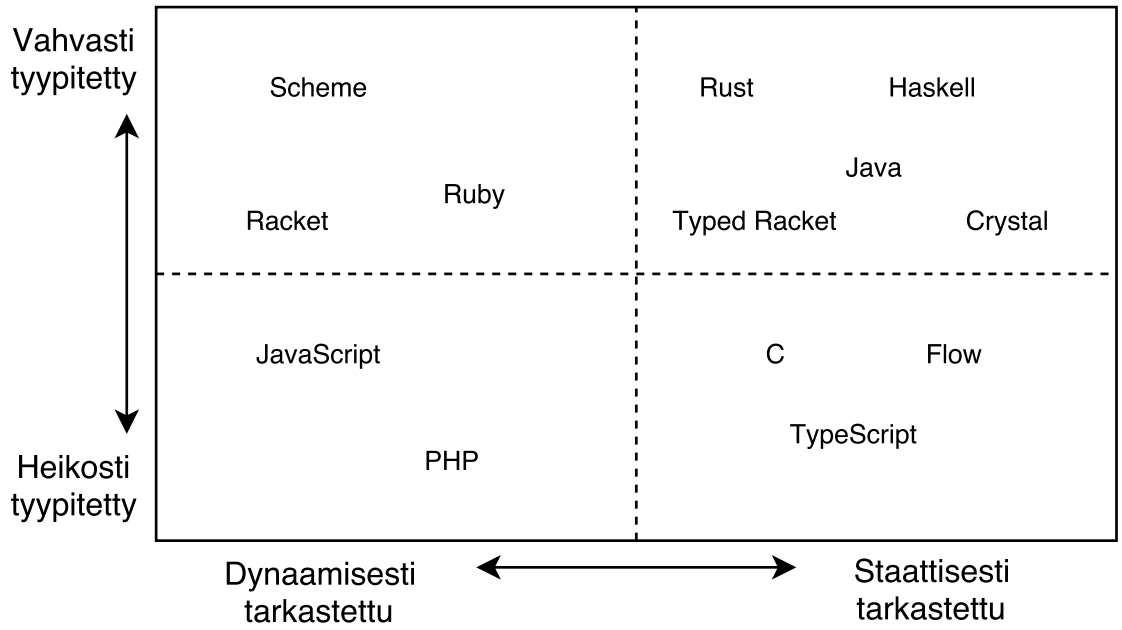
Luku 2

Peruskäsitteitä

2.1 Tyyppijärjestelmien luokitteleminen

Ohjelmointikielten tyyppijärjestelmien jakaminen staattisesti ja dynaamisesti tyyppitarkastettuihin perustuu ohjelman kehitysvaiheeseen, jossa tarkastaminen tapahtuu. Staattisella tyyppitarkastamisella viitataan ohjelman tyyppien analyysiin ennen ohjelman suorittamista, esimerkiksi käännösaikana, kun taas dynaaminen tyyppitarkastus varmistaa arvojen tyyppien oikeellisuuden ohjelmaa suoritettaessa. Tyyppijärjestelmät voidaan jaotella myös muiden ominaisuuksien perusteella esimerkiksi vahvoihin ja heikkoihin tyyppijärjestelmiin. Näiden termien merkitys ei ole tarkasti määritelty, mutta yleisesti niillä viitataan tapaan, jolla kieli käsittelee tarkoitetusta poikkeavat, virheelliset tyypit [1]. Vahvasti tyypitetyssä kielessä tietyn tyyppisen muuttujan vääränlainen käsittely aiheuttaisi käännös- tai ajonaikaisen virheen, kun taas heikosti tyypitetyssä kielessä arvolle voitaisiin tehdä implisiittisiä tyyppimuunnoksia niiden yhteensopivuuden saavuttamiseksi.

JavaScript on dynaamisesti tarkastettu, heikosti tyypitetty kieli. Esimerkiksi ohjelma `"teksti".potenssiin(3)` antaa staattisesti tyyppitarkastetussa kielessä virheen jo käännösaikana, mikäli metodia `potenssiin` ei ole tekstityyppisille arvoille määritetty. JavaScriptiä suorittava ympäristö sen sijaan hyväksyisi ohjelman ja sallisi sen suorittamisen. Virhe olemattoman metodin kutsumisesta ilmenisi vasta, jos kyseinen virheen sisältävä koodi suoritetaan. Lisäksi esimerkiksi ilmaisu `"teksti" + 2` ei aiheuttaisi virhettä edes suoritusaikana, sillä heikoille tyyppijärjestelmille ominaisesti JavaScript



Kuva 2.1: Tyyppijärjestelmät eri ohjelmointikielissä

muuttaisi numeron 2 tekstimuotoon ennen summausoperaation arviointia. Tässä tutkielmassa keskitytään lähinnä JavaScriptin tyyppien staattiseen ja dynaamiseen, eli käytännössä käännös- ja ajonaikaiseen tarkastamiseen. Eräät esitellyistä työkaluista myös tiukentavat kielen sallimia operaatioita siten, että esimerkiksi yllä esitettyä `"teksti" + 2` ohjelmaa ei enää sallittaisi. Monia muita heikoille tyyppijärjestelmille tavallisia ominaisuuksia jää kuitenkin tarkistamatta.

2.2 ECMA-262, EcmaScript ja JavaScript

EcmaScript on ECMA-262 standardin määrittelemä ohjelmointikieli [2, 3], jonka kehityksestä vastaa organisaatio Ecma International. *JavaScript* puolestaan on Oraclen omistama tavaramerkki jolla viitataan EcmaScript-kielen osittaisiin tai täydellisiin toteutuksiin [2]. Historiallisista syistä termejä “JavaScript” ja “EcmaScript” käytetään usein keskenään vaihtokelpoisesti. Tässä tutkielmassa termillä “JavaScript” viitataan ECMA-262:n kahdeksannen version mukaiseen EcmaScriptiin, jota kutsutaan myös nimellä EcmaScript 2017.

2.3 TypeScript

TypeScript on Microsoftin luoma ohjelmointikieli, jonka tarkoitus on auttaa JavaScript-ohjelmien kehitystä staattisen tyyppijärjestelmän avulla. Se on EcmaScriptin ylijoukko (engl. superset) [4] ja jatkaa JavaScriptin syntaksia tyyppimäärittelyihin käytettävällä annotaatiosyntaksilla. Jokainen validi JavaScript-ohjelma on syntaksiltaan ja ajonaikaiselta käyttäytymiseltään validi TypeScript-ohjelma. TypeScript kuitenkin lisää kehitykseen käännösvaiheen, jossa ohjelman tyyppien oikeellisuus tarkastetaan staattisesti. TypeScript-koodi käännetään JavaScriptiksi, joka puolestaan voidaan suorittaa selaimissa tai muissa JavaScriptin suoritusympäristöissä. TypeScript-kääntäjän voi myös määrittää muokkaamaan tulostettava koodi yhteensopivaksi vanhojen EcmaScript-standardien kanssa, mikä on hyödyllistä, jos ohjelman on tarkoitus tukea sellaisia suoritusympäristöjä, jotka eivät tue uusinta EcmaScriptin versiota.

2.4 Flow

Flow on Facebookin kehittämä työkalu, joka TypeScriptin tavoin jatkaa JavaScriptin syntaksia staattisesti tarkastettavilla tyyppimäärittelyillä. Flow itsessään ei sisällä kääntäjää, vaan keskittyy yksinomaan ohjelman tyyppiturvallisuuden tarkastamiseen. Koodiin lisätyt tyyppimääritykset on kuitenkin poistettava ennen kuin JavaScript-ohjelma voidaan suorittaa. Tähän tarkoitukseen voidaan käyttää esimerkiksi Babel-kääntäjää, joka poistaa Flow-tyyppimäärittelyt ja muokkaa JavaScript-koodin yhteensopivaksi toivotun EcmaScript-version kanssa [5].

2.5 Closure-kääntäjä

Googlen Closure-kääntäjä on käännöstyökalu, jonka pääasiallinen tarkoitus on minimoida ja optimoida JavaScript-koodia käännösvaiheessa ennen tuotantoon siirtämistä. Closure sisältää kuitenkin myös tuen tyyppivirheiden tarkastamiselle käännösvaiheessa [6]. Tyypit annotoidaan erityisellä JSDoc-pohjaisilla dokumentaatiokommenteilla. Koska an-

notaatiot ovat kommenteissa eivätkä erityisenä syntaksina muun suoritettavan koodin joukossa, Closure-annotoitua JavaScriptiä ei tarvitse kääntää ennen sen suorittamista [7].

Luku 3

JavaScript-ohjelman muuttaminen staattisesti tyyppitarkastetuksi

3.1 Tyyppiannotaatiot

Tyyppijärjestelmä voi päätellä muuttujan sallitun tyypin automaattisesti päättelemällä tai kieleen sisältyvien eksplisiittisten tyyppimäärittelyjen perusteella. Kaikki kolme tässä tutkielmassa esiteltyä JavaScriptin staattiseen tyyppitarkastukseen tarkoitettua työkalua päättelevät muuttujien tyyppejä automaattisesti, mutta vaativat paikoitellen myös eksplisiittisiä määrityksiä. Closure-kääntäjä lukee tyyppimääritykset JSDoc-tyylisistä dokumentaatiokommenteista [7].

```
1 /**
2 * @param {!Array<Ostos>} ostokset
3 * @return {number} Ostosten yhteenlaskettu hinta
4 */
5 function ostoskorinHinta(ostokset) {
6   let summa = 0;
7   for (const ostos of ostokset) summa += ostos.hinta;
8   return summa;
9 }
```

Listaus 3.1: Esimerkki Closure-annotaatiosta funktiolle

Listauksessa 3.1 `ostoskorinHinta` -funktion tyyppimäärittely on toteutettu sen yläpuolella olevilla kommenteilla, jotka määrittävät tyypin `ostokset` parametrille sekä funktion palautusarvolle. TypeScript ja Flow puolestaan jatkavat ECMA-262 -spesifikaatiota erityisellä syntaksilla tyyppien eksplisiittistä määrittämistä varten.

```
1 function ostoskorinHinta(ostokset: Ostos[]): number {
```

Listaus 3.2: Esimerkki Flow tai TypeScript annotaatiosta funktiolle

Flow ja TypeScript -esimerkeissä 3.2 tyyppiannotaatiot ovat osana koodia, mikä tekee

ohjelmasta yhteensopimattoman tavallisen JavaScriptin kanssa. Ohjelma on käännettävä JavaScriptiksi ennen suorittamista. Annotaatioiden syntaksi ja merkitys ei myöskään ole välttämättä suoraan selvä JavaScript-ohjelmoijalle, joka pahimmassa tapauksessa voi kokea lisätyt tyyppimäärittelyt vaikeasti luettavina.

Closuren annotaatiot on sijoitettu kommentteihin, joten niillä ei ole ajonaikaista vaikutusta ja ohjelma on täten sellaisenaan hyväksyttävää JavaScriptiä. Toisaalta tyyppiannotaatioiden määrittely kommenteissa voi olla runsassanaista ja hankalaa, mikä kasvattaa niiden kirjoittamiseen vaadittua työmäärää [4, 8].

Aiemmassa esimerkissä esitelty tyyppi `Ostos` pitäisi määritellä Closurea varten muiden annotaatioiden tapaan dokumentaatiokommentteja käyttäen:

```
/**
* @typedef {{
*   nimi: string,
*   hinta: number
* }}
*/
let Ostos;
```

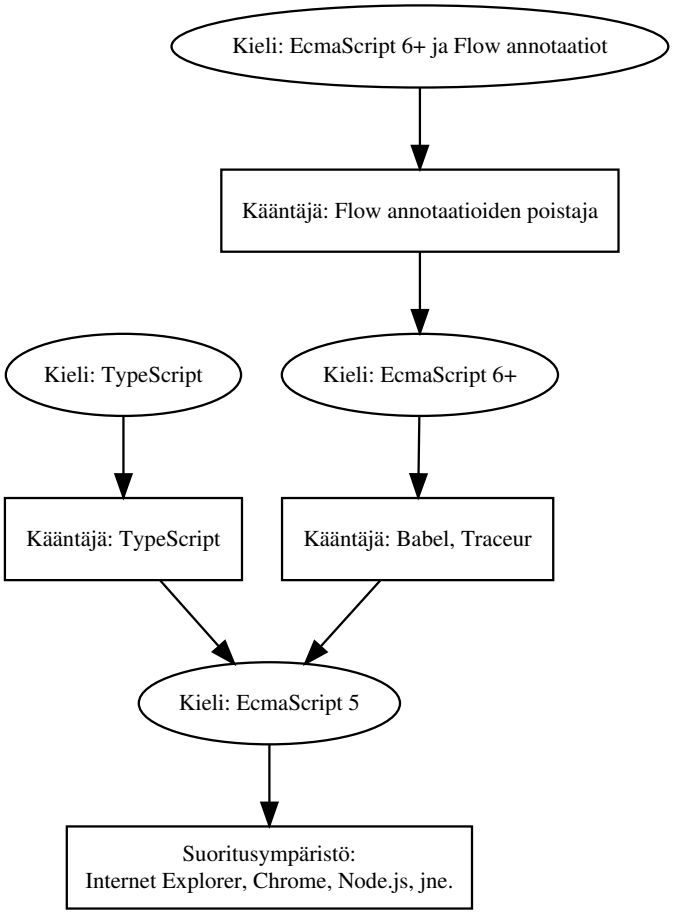
TypeScript ja Flow tarjoavat käännösaikaisen tyypin (type alias) määrittelyyn tiiviimmän ja helppolukuisemman syntaksin [4]:

```
type Ostos = {
  nimi: string;
  hinta: number;
};
```

Tällainen tyypin määritteleminen vaikuttaa ainoastaan käännösvaiheen tyyppitarkastukseen, eikä määrittely tuota tietorakenteita tai muuta sisältöä suoritettavaan JavaScript-ohjelmaan.

3.2 Ohjelman kääntäminen ennen suorittamista

JavaScript koodi tulkataan tai käännetään tyypillisesti suorittamisen yhteydessä, selaimesta tai muusta suoritusympäristöstä löytyvän ”moottorin” toimesta. EcmaScript-standardin mukaista koodia suorittamaan suunnitellut moottorit, kuten V8 tai SpiderMonkey, eivät kuitenkaan osaa käsitellä TypeScript- tai Flow-annotaatioilla merkattua koodia. Näinollen TypeScript- tai Flow-annotoitu koodi on välttämätöntä kääntää muotoon jossa annotaatiot on poistettu ja jäljellä on enää standardinmukainen JavaScript. Koska JavaScriptin käyt-



Kuva 3.1: Vaihtoehtoisia käännösprosesseja

täminen ei normaalisti vaadi erillistä käännösvaihetta, useissa projekteissa ei ole sellaista käytetty. Koodin minimointi ja muu optimointi on ollut parhaiden käytäntöjen mukaista jo tovin, mutta tällaiset koodinkäsittelyt tehdään yleensä vasta ennen ohjelman julkaisua. Kehittäjät ovat tavanomaisesti voineet suorittaa kirjoittamansa JavaScriptin sellaisenaan kehitysympäristössä. Käännösvaiheen aikavaatimus pyritään luonnollisesti pitämään mahdollisimman pienenä, mutta se on silti projektin monimutkaisuuteen ja kehitysnopeuteen vaikuttava tekijä, joka tulee huomioida työkalun käyttöönotossa.

Kuva 3.1 esittää erilaisia käännösprosesseja eri JavaScript-ohjelman kehityskielille. Jos koodi kirjoitetaan kaikkiin suoritusympäristöihin soveltuvalla kielen versiolla, kuten EcmaScriptin versiolla 5 (2011), käännösvaihe voidaan jättää kokonaan väliin. Kielen uudemmat versiot ovat kuitenkin tuoneet monia lisäominaisuuksia, joiden käyttäminen van-

hempia selaimia tukevissa ohjelmissa vaatii joka tapauksessa käännösvaiheen staattisesta tyypittämisestä riippumatta. Uusia JavaScript- ominaisuuksia hyödyntäville projekteille koodin kääntäminen ei siis tule täysin uutena vaiheena.

3.3 Työkalun vaiheittainen käyttöönotto

JavaScript-kirjastojen hallintaan tarkoitettu rekisteri npm on yksi suurimmista ohjelmistoekosysteemeistä [9]. Tutkinnon kirjoittamisen hetkellä rekisterin kotisivu, npmjs.com ilmoitti ladattavissa olevien pakettien määräksi yli 800 000. Avoimessa jakelussa olevien kirjastojen lisäksi JavaScriptiä käyttävillä kehittäjillä voi olla suuri määrä valmista JavaScript-koodia, jota voi hyödyntää uusissa projekteissa. Jotta TypeScript, Flow ja Closure olisivat hyödyllisiä työkaluja JavaScript-ohjelmien kehitykseen, on niiden oltava yhteensopivia sellaisen JavaScript-koodin kanssa jonka kehitykseen ei kyseistä työkalua ole käytetty.

TypeScript ja Flow tukevat erityisiä määrittelytiedostoja, joiden sisältämällä tyyppiannotoidulla koodilla määritetään kirjaston tai muun JavaScript-koodin ulkoisen rajapinnan tyyppimäärittelyt. Näiden tiedostojen kirjoittamiseen käytetty syntaksi on muuten sama kuin muissakin tiedostoissa, muuta niiden funktio- ja metodimäärittelyistä on jätetty implementaatiot kokonaan pois. Tiedoston ei ole tarkoitus olla osana varsinaista suoritettavaa koodia, vaan se palvelee ainoastaan kuvauksena sellaisen koodin tyyppimäärittelystä, jonka käsittelyä TypeScript ja Flow eivät muuten voisi valvoa. TypeScriptillä voitaisiin esimerkiksi kirjoittaa seuraava tiedosto `ostoskori.d.ts`, jonka tehtävä on annotoida toista tiedostoa `ostoskori.js`.

```
1 export const tuotteet: ReadonlyArray<Ostos>;
2
3 /** Lisää tuotteen ostoskoriin. */
4 export function lisääTuote(ostos: Ostos): void;
```

Listaus 3.3: Esimerkki TypeScript määrittelytiedostosta ostoskori.d.ts

Tämän jälkeen TypeScript tiedostosta käsin voidaan kutsua tätä JavaScript-funktiota siten, että TypeScript valvoo tyyppien oikeellisuutta.

```
1 import * as ostoskori from "./ostoskori";
2
3 ostoskori.lisääTuote({ nimi: "juusto", hinta: 5 });
```

Listaus 3.4: JavaScript-koodin kutsuminen TypeScript tiedostosta tuotesivu.ts

Joissain tapauksissa kirjastoille tai kehittäjän omalle aiemmin kirjoitetulle JavaScript-koodille ei kuitenkaan ole valmiita TypeScript-tyyppimäärittelyjä eikä niitä syystä tai toisesta voida luoda ennen muun kehityksen jatkamista. Kaikkien kolmen työkalun tärkeimpiin ominaisuuksiin kuuluu tuki vaiheittaiselle käyttöönotolle, eli käytännössä yhteenspivuus täysin tyyppitarkastamattoman koodin kanssa. Sekä Flow että TypeScript tarjoavat erityisen yleisviittaustyypin Any, jota voi käyttää kuvaamaan mitä tahansa JavaScript arvoa [4]. Any-tyyppiseen muuttujaan voidaan asettaa mikä tahansa arvo ja Any tyyppinen arvo voidaan asettaa mihin tahansa muuttujaan tai funktioparametriin. Any tyypin avulla muuten staattisesti tyypitetyssä ohjelmassa voidaan ohittaa käännösaikainen tyyppien tarkistaminen sellaisten koodin osien kohdalla joiden ajonaikaista arvoa olisi muuten vaikea tai mahdotonta määritellä käännösaikana. Näin ollen, mikäli tarve vaatii täysin annotoimattoman JavaScript moduulin käyttämistä, kaikkien kyseisestä moduulista tuotujen arvojen voidaan määrittää olevan tyyppiä Any, jolloin tarkastaja ei kiinnitä huomiota siihen miten JavaScriptillä määritettyjä funktioita tai muita arvoja käsitellään.

Luku 4

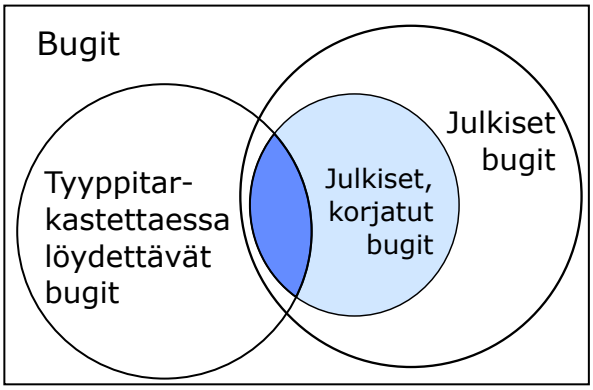
Staattisen tyypityksen hyödyt

4.1 Virheiden havaitseminen

Kenties tärkein staattisen tyyppijärjestelmän tehtävä on havaita ja estää ohjelmoijan virheitä. Tässä esitellyt työkalut, mahdollisesti Closure–kääntäjää lukuunottamatta, onkin kehitetty erityisesti tätä tarkoitusta varten.

Kaikki kolme työkalua antaisivat käännösvirheen jos esimerkeissä 3.1 ja 3.2 esiteltyä funktiota kutsuttaisiin virheellisesti esimerkiksi listalla hintaa kuvaavia numeroita, sillä funktion parametrin on annotoitu olevan lista “Ostos”-tyyppimääritelmän mukaisia objekteja. Esimerkiksi virheellinen kutsu `ostoskorinHinta([5, 10, 15])` ei itse asiassa aiheuttaisi suoritettaessa ohjelman keskeyttävää virhettä. `ostos.hinta` ilmaisu on sallittu vaikka muuttuja `ostos` olisikin arvoltaan numero eikä objekti. Tällöin ilmaisun arvo on `undefined` ja lausekkeen `summa += ostos.hinta` jälkeen `summa` muuttujan arvo on erityinen ei-numeroa kuvaava `NaN` [10]. Käännösaikaisen tarkistamisen merkitys korostuu erityisen hyödylliseksi tämänkaltaisen ohjelmointivirheen kohdalla, sillä virhe ei välttämättä ole muutoin helposti havaittavissa. Funktiokutsu ei aiheuttaisi helposti todennettavaa suoritusaikaista virhettä, joten ei-toivottu palautusarvo `NaN` saattaisi kiertää ohjelman operaatioiden välillä pitkällekin aiheuttaen muita loogisia virheitä.

Vuonna 2017 tehdyssä tutkimuksessa TypeScriptin ja Flown vaikutuksesta avoimen lähdekoodin JavaScript-projekteihin havaittiin, että vähintään 15% ilmoitetuista ja korjatuista bugeista olisi voitu havaita ja välttää jos projektin kehitykseen oltaisin käytetty jom-



Kuva 4.1: To Type or Not to Type: Quantifying Detectable Bugs in JavaScript -tutkimuksen käsittelemät bugit [11].

paakumpaa näistä työkaluista [11]. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript -tutkimuksen arvioinnissa huomioitiin lisäksi, että tulos on tutkimusmenetelmästä johtuen mitä luultavimmin alempi kuin tällaisen muutoksen tuoma todellinen vaikutus. Tutkimus toteutettiin muuntamalla avoimen lähdekoodin JavaScript-kirjastoja ensin staattisesti tyypitettyyn muotoon ja sitten testaamalla kuinka hyvin tyyppitarkastus havaitsi ennalta tunnettuja bugeja aiheuttavan koodin. Sen ulkopuolelle jäivät bugit joita ei oltu vielä korjattu tai havaittu, sekä bugit jotka kehittäjä oli havainnut jossain kehitysvaiheessa ennen virheellisen ohjelman julkaisua. Staattisen tyyppitarkastus luultavasti auttaisi vähentämään myös näitä bugeja.

Kuvaajan 4.1 esittämässä tilanteessa suuri osa bugeista ei ole julkisia, eli kehittäjät eivät ole havainneet bugia eivätkä käyttäjät ole ilmoittaneet sellaisesta. Esimerkiksi seuraavanlainen koodi saattaisi aiheuttaa vaikeasti havaittavan virheen joka jäisi helposti dynaamisessa testaamisessa huomaamatta:

```
1 if (viikonpaiva === "perjantai") {
2   // Alennus perjantaisin 5 euroa
3   ostoskori.lisääTuote({ nimi: "astiasto", Hinta: 10 });
4 } else {
5   ostoskori.lisääTuote({ nimi: "astiasto", hinta: 15 });
6 }
```

Listaus 4.1: Vaikeasti havaittavan virheen aiheuttava koodiesimerkki

Esimerkin koodi ei toimi oikein perjantaisin, sillä `Ostos` -tyyppisen objektin ominaisuus `hinta` on virheellisesti kirjoitettu isolla alkukirjaimella. Koska bugi toistuu vain tietyis-

sä olosuhteissa, se voi pysyä havaitsemattomana pitkään. Staattiselle analyysille konditionaalisen ohjelman tarkistaminen ei kuitenkaan ole ongelma ja tyyppiannotoituna kaikki kolme työkalua pystyvätkin osoittamaan esimerkissä olevan virheen.

4.2 Ohjelman optimointi käännösvaiheessa

Aivan ensimmäiset tyyppijärjestelmät, kuten Fortranin staattinen tyypitys, kehitettiin laskutoimitusten suoritusajan optimointia varten [12]. Myös uudemmissa kielissä muuttujien tyypeistä saatavilla olevaa tietoa voidaan käyttää ajonaikaisen turvallisuuden varmistavien tarkistusten optimointiin.

Kun JavaScriptia optimoidaan käännösvaiheessa, on kuitenkin usein hyödyllistä kiinnittää enemmän huomiota tuotetun koodin kokoon kuin suoritusaikaiseen tehokkuuteen. Tyypillinen JavaScript-ohjelma ladataan sivulle saapuessa internet-yhteyden yli juuri ennen suorittamista, minkä vuoksi ladattavan koodin koon kasvattaminen minimaalisen suoritusajan edun vuoksi ei usein ole kokonaisuudessaan hyödyllistä. Closure–kääntäjä on kehitetty erityisesti JavaScript-koodin koon optimointia ajatellen. Muuttujanimet voidaan helposti uudelleennimetä ilmankin staattisen tyypityksen apua, mutta sen lisäksi myös joukko muita koodia keventäviä optimointeja on käytettävissä kun kääntäjällä on muuttujien tyypeistä saatava tieto käytettävissä. Closure–kääntäjä osaa uudelleennimetä myös luokkamuuttujien nimiä lyhyemmiksi, poistaa käyttämätöntä koodia, sekä korvata yksinkertaisia funktiokutsuja siirtämällä funktion sisältämät ohjeet kutsuntapaikalle (engl. function inlining).

4.3 Tyyppimäärittelyt dokumentaationa

Eksplisiittisesti kirjoitetut tyypit sekä editorin antama tieto muuttujien tyypeistä voi toimia myös aiemmin kirjoitetun koodin dokumentaationa ja kuvauksena siitä, miten moduulia on tarkoitus käyttää. Nykyaikasten editorien vakio-ominaisuuksiin kuuluu kirjoittamisen tukeminen automaattisilla ehdotuksilla. Ehdotukset nopeuttavat pitkien metodini-

Kuva 4.2: IntelliSensen tarjoamat ehdotukset JavaScriptille (yllä) ja TypeScriptille (alla). JavaScriptille annetuissa ehdotuksissa on turhia, tiedostossa käytettyihin sanoihin perustuvia ehdotuksia.

mien kirjoittamista ja toimivat eräänlaisena dokumentaation lähteenä, sillä ohjelmoija voi tutkia luokan tai paketin tarjoamaa sisältöä ehdotettuja nimiä selaamalla. Automaattisia ehdotuksia on mahdollista tarjota myös dynaamisesti tyyppitarkastetuille kielille, mutta koodipohjan kasvaessa ja muuttuessa monimutkaisemmaksi ehdotusten tarkkuus on vaikea pitää samalla tasolla kuin staattisesti tyypitetyissä kielissä. Huonoimmillaan ehdotetut muuttuja- ja metodinimet valitaan yksinkertaisesti listaamalla avoimista tiedostoista löytyviä nimiä, välittämättä sen enempää siitä onko nämä metodit määritetty juuri kyseiselle tyypille. Edistyneemmät ehdotusmoottorit kuten Visual Studiossa käytetty IntelliSense, suorittavat osaa JavaScript-koodista taustalla ja analysoivat siten muuttujien tyyppejä ajonaikana [13, 14]. Tämä tekniikka yhdistettynä tavanomaisempaan tyyppien käännösaikaiseen päättelyyn voi riittää tarjoamaan melko kattavan kuvauksen jonkin muuttujan tyypistä, mutta jää silti jälkeen siitä tarkkuudesta jonka IntelliSense osaa antaa staattisesti tyypitetylle koodille.

Automaattisten ehdotusten vaikutuksesta ohjelmointityön tehokkuuteen ei kuitenkaan ole yleisesti hyväksyttyä, tutkittua varmuutta. Vaikka metodien nimet olisivatkin ohjelmoijan nähtävillä editorissa, ne eivät välttämättä sellaisenaan tarjoa tarpeeksi hyötyä do-

kumentaationa jotta työtehokkuus kasvaisi merkittävästi. Vuonna 2015 toteutettu tutkimus “An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability using TypeScript and JavaScript in MS Visual Studio” testasi staattisen tyypityksen ja automaattisten ehdotusten tehokkuutta antamalla osallistujille toteutettavaksi ohjelmointitehtävän JavaScriptillä ja TypeScriptillä, automaattisten ehdotusten kanssa ja niitä ilman [15]. Tutkimuksen mukaan automaattiset ehdotukset eivät toisi tilastollisesti merkittävää parannusta ohjelmointitehtävän ratkomisnopeuteen. Samassa tutkimuksessa kuitenkin nähtiin että TypeScriptiä käyttäneet koehenkilöt suoriutuivat tehtävästä nopeammin, automaattisilla ehdotuksilla tai ilman. Voi olla, että automaattisia ehdotuksia tehokkaampi työkalu on itse kääntäjä, joka paljastaa virheet koodissa ennen kuin ohjelmoijan tarvitsee kokeilla koodin toimivuutta käytännössä.

Luku 5

Ongelmat JavaScriptin staattisessa tyypittämisessä

5.1 EcmaScript-yhteensopivuus

TypeScript pyrkii noudattamaan EcmaScript-spesifikaatiota mahdollisimman tarkasti kaikkien sellaisten ominaisuuksien suhteen jotka eivät nimenomaan liity staattiseen tyypittämiseen [16]. TypeScriptin kehityksen alkuvaiheissa, ennen EcmaScript 2015-spesifikaation valmistumista, JavaScriptista kuitenkin puuttui joitain tärkeiksi katsottuja ominaisuuksia, jotka päätettiin lisätätä TypeScriptiin mahdollisista yhteensopimusongelmista huolimatta. TypeScriptissä on esimerkiksi syntaksi nimiavaruuden määrittämiseen, vaikka EcmaScriptiin myöhemmin lisätyt *moduulit* ajavat saman asian [17]. TypeScript tukee nykyään sekä alkuperäistä nimiavaruus-ominaisuuttaan että EcmaScriptin moduuleita.

Flow ja Closure-kääntäjä lisäävät koodiin ainoastaan staattista tyypitystä koskevia ominaisuuksia, kuten tyyppimäärittelyjä, joten niissä ei ole nimiavaruuksien kaltaisia koodin suoritukseen vaikuttavia ominaisuuksia. EcmaScript kuitenkin kehittyy nopeasti ja sen päälle rakentavien työkalujen on pysyttävä tahdissa mukana ollakseen hyödyllisiä kehittäjille.

Kaikki kolme työkalua pyrkivät tukemaan EcmaScriptin uusinta viimeisteltyä versiota. Lisäksi Flow tarjoaa tyyppitarkastusta joillekkin kokeellisille ominaisuuksille joiden EcmaScript-määrittely ei vielä ole täysin valmis mutta jotka luultavasti tullaan lisäämään myöhemmin. Esimerkiksi *null-turvallinen ketjutus* (engl. optional chaining tai safe navi-

gation) [18] on vielä suunnitteluvaiheessa oleva kielen ominaisuus, mutta Flow tarjoaa jo staattisen tyyppitarkastuksen sille. Uusien ominaisuuksien aikaisessa käyttöönotossa on JavaScriptin kehityksen kannalta se hyvä puoli, että kehittäjäyhteisö pääsee kokeilemaan ja antamaan palautetta ennen kuin ominaisuuden määrittely on lyöty lukkoon. TypeScript puolestaan pyrkii vastedes implementoimaan ainoastaan valmiita ominaisuuksia, sillä keskeneräisen ominaisuuden yksityiskohdat tulevat suurella todennäköisyydellä muuttumaan useaan otteeseen suunnitteluvaiheen aikana ja sitä käyttäneet kehittäjät voivat myöhemmin joutua *refaktoroimaan* koodiaan.

5.2 Tyyppien automaattinen ja vaiheittainen tyypittäminen

Jotta staattisen tyypityksen tuova työkalu olisi varteenotettava vaihtoehto dynaamisesti tyypitetyn JavaScriptin rinnalla, sen kirjoittaminen ei voi vaatia liian paljon enemmän työtä kuin JavaScriptin. Staattisessa tyyppijärjestelmässä näkyvin ero dynaamisesti tyypitettyyn on eksplisiittiset tyyppiannotaatiot, jotka voivat olla lisärasite koodin kirjoittajalle. Erityisesti vanhaa, dynaamisesti tyypitettyä JavaScriptia refaktoroidessa staattisesti tyypitettyyn muotoon on kätevää jos jokaista muuttujaa ja funktiota ei tarvitse muokata uuteen kieleen siirryttäessä.

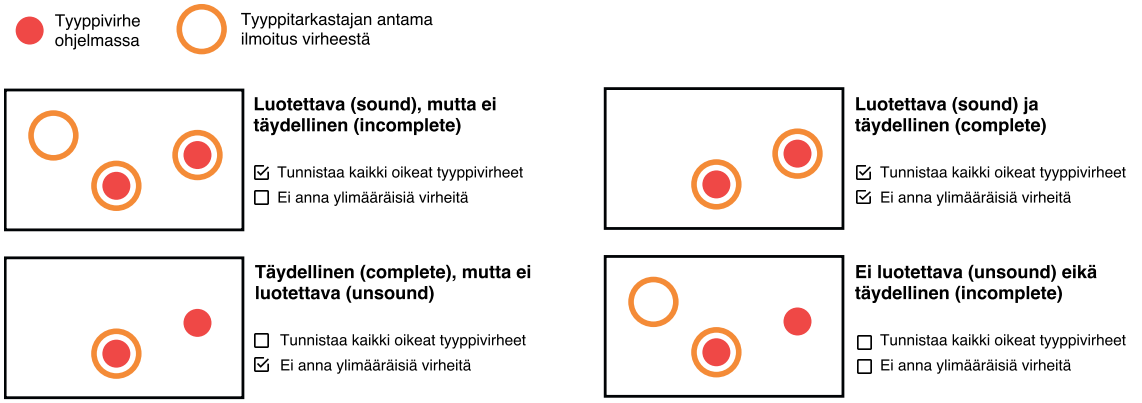
TypeScript, Flow ja Cosure-kääntäjä tukevat melko pitkälle kehittynyttä tyyppien automaattista päättelyä (engl. type inference).

```
1  const vertaaHintoja = (a, b) => a.hinta - b.hinta;
2  tuotteet.sort(vertaaHintoja);
```

Listaus 5.1: Flow pystyy tulkitsemaan vertaaHintoja-funktion tyypin automaattisesti, ilman eksplisiittisiä tyyppimäärittelyitä.

Esimerkissä 5.2 Flow pystyy päättelemään tyypin

`vertaaHintoja: (a: Ostos, b: Ostos) => number` automaattisesti, vaikkei esimerkissä ole yhtään eksplisiittistä tyyppimäärittelyä tai muuta tavallisesta JavaScriptista poikkeavaa. Flow tietää ennestään mikä listan sort-metodin tyyppi on, joten se pystyy päättelemään myös sort-metodille annetun vertaaHintoja-funktion tyypin käyttä-



Kuva 5.1: Tyyppijärjestelmän luotettavuus ja täydellisyys

mällä *kutsumispaikkaan perustuvaa päättelyä* (engl. call-site inference). TypeScript ei tue kutsumispaikkaan perustuvaa päättelyä samalla tavalla kuin Flow, mikä onkin yksi isoimmista eroista näiden kahden työkalun välillä. Liian pitkälle viety tyyppien automaattinen päättely voi johtaa ongelmiin sekavien virheviestien tai työkalun hitauden muodossa, minkä vuoksi TypeScript vaatii nimettyjen funktioiden argumenteille aina eksplisiittiset tyypit. Kun funktion signatuuriin lisätään `(a: Ostos, b: Ostos)`, myös TypeScript pystyy automaattisesti päättelemään että funktion palautusarvo on numero, eikä eksplisiittistä `: number` palautusarvon määritystä tarvita. Molempien työkalujen tapauksessa on hyvä huomata että tyyppien automaattinen päättely on viety paljon pidemmälle kuin se funktion sisäinen muuttujan tyypin päättely jollaista nähdään esimerkiksi `var`-avainsanalla C#:issa ja Javassa.

5.3 Luotettavuus, täydellisyys ja käytännöllisyys

Tyyppijärjestelmän luotettavuus (engl. soundness) kuvaa sitä, kuinka suuren osan mahdollisista ohjelmointivirheistä se estää. Täysin luotettava (engl. sound) tyyppijärjestelmä estää kaikki sellaiset virheet jotka sen on tarkoitus estää [19]. Täydellisyys (engl. completeness) puolestaan kertoo salliiko tyyppijärjestelmä kielen sellaiset ominaisuudet jotka eivät olisi ajonaikana tyyppivirheitä [12, 19].

Jotta JavaScriptiä analysoiva tyyppijärjestelmä olisi luotettava, sen on annettava virhe

esimerkiksi seuraavasta ohjelmasta:

```
function osta(ostos) {
  lisääTuote({
    nimi: ostos.nimi,
    hinta: ostos.hinta
  });
}

osta({ nimi: 'juusto', hinta: 5 });
osta({ hinta: 5 });
```

Listaus 5.2: Virheellinen JavaScript-ohjelma: Lisätyllä tuotteella ei ole nimeä.

Toisaalta jotta JavaScriptiä analysoiva tyyppijärjestelmä olisi täydellinen, sen on sallittava tämä korjattu versio ylläolevasta ohjelmasta:

```
function osta(ostos) {
  if (typeof ostos.nimi === 'string') {
    lisääTuote({
      nimi: ostos.nimi,
      hinta: ostos.hinta
    });
  }
}

osta({ nimi: 'juusto', hinta: 5 });
osta({ hinta: 5 });
```

Listaus 5.3: Toimiva JavaScript-ohjelma: Virheelliseltä kutsulta on suojauduttu tarkistuksella.

Esimerkit 5.2 ja 5.3 toimivat odotetulla tavalla Flow:ssa. TypeScript vaatii eksplisiittisen tyyppiannotaation osta-funktiolle, mutta toimii muuten samalla tavalla. Flow, TypeScript ja Closure eivät kuitenkaan ole täydellisiä tai kokonaan luotettavia. Monimutkaisemmissa tilanteissa virheitä saattaa jäädä nappaamatta tai toimiva ohjelma voidaan merkata virheelliseksi.

JavaScriptiä käännöskohteena käyttävät mutta muuten sen syntaksista ja semantiikasta eroavat uudet kielet, kuten Dart, Elm ja ReasonML on voitu kehittää toivotunlaiseksi ilman painetta olla yhteensopiva vanhan koodin kanssa. TypeScript ja Flow on sen sijaan kehitetty lisäämään staattinen tyypitys olemassa olevaan kieleen, JavaScriptiin, siten että nykyisellään käytössä olevat kirjastot ja koodikäytännöt pystytään tyyppitarkastamaan ilman että niiden arkkitehtuuria tarvitsee merkittävästi muuttaa tyyppiturvallisuuden saavuttamiskesi.

TypeScript on strukturaalisesti tyypitetty, minkä vuoksi seuraava koodi kääntyy ilman

tyyppivirheitä.

```
1 class Ihminen {
2     constructor(public nimi: string){}
3 }
4 class Eläin {
5     constructor(public nimi: string){}
6 }
7 function varaaEläinlääkäri(omistaja: Ihminen, lemmikki: Eläin){}
8
9 varaaEläinlääkäri(new Eläin("Musti"), new Ihminen("Jaakko"));
```

Listaus 5.4: Loogisen virheen sisältävä, mutta ilman virheitä kääntyvä TypeScript-ohjelma.

TypeScript-kääntäjä sallii esimerkin 5.3 koodin vaikka arugmentit "omistaja"ja "lemmikki"ovat väärin päin, sillä molempien luokkien rakenne on sama; molemmissa on pelkkä tekstimuotoinen ominaisuus "nimi". Flow:ssa sama virhe ei menisi läpi. Siinä luokkainstanssit on tyypitetty nominaalisesti, mikä auttaa tässä esimerkissä mutta aiehuttaa ongelmia muissa tilanteissa. Projekti saattaa esimerkiksi sisältää kaksi versiota samasta kirjastosta jonkin toisen kirjaston kautta, mikä voi aiheuttaa yhteensopimusongelmia kun käytännössä saman luokan tyyppejä ei lueta keskenään yhteensopiviksi.

Luku 6

Yhteenveto

Lähdeluettelo

[1] Transition to OO programming - Safety and strong typing. URL http://www.cs.cornell.edu/courses/cs1130/2012sp/1130selfpaced/module1/module1part4/strongtyping.html.

[2] JavaScript language resources. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources.

[3] Standard ECMA-262 - ECMAScript 2017 Language Specification. URL https://www.ecma-international.org/publications/standards/Ecma-262.htm.

[4] TypeScript Language Specification. URL https://github.com/Microsoft/TypeScript/blob/b8fbf884d0f01c1a20bb921cc0a65d6c1a517ee8/doc/TypeScript%20Language%20Specification.pdf, versio 1.8.

[5] Installing and setting up Flow for a project. URL https://flow.org/en/docs/install/.

[6] Closure Compiler. URL https://developers.google.com/closure/compiler/.

[7] Annotating JavaScript for the Closure Compiler. https://github.com/google/closure-compiler/wiki/Annotating-JavaScript-for-the-Closure-Compiler.

[8] Anders Hejlsberg. Microsoft Build 2014. TypeScript, maaliskuu 2014. URL https://channel9.msdn.com/Events/Build/2014/3-576.

[9] Shriam Rajagopalan Erik Wittern, Philippe Suter. A Look at the Dynamics of the JavaScript Package Ecosystem. 2016.

[10] Ecma International. *ECMA-262 Section 12.8.3*, kesäkuu 2017. URL https://www.ecma-international.org/ecma-262/8.0/index.html#sec-additive-operators, versio 8.0.

[11] Earl T. Barr Zheng Gao, Christian Bird. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. 2017.

[12] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[13] Previewing Salsa – the New JavaScript Language Service in Visual Studio "15", huhtikuu 2016. URL https://blogs.msdn.microsoft.com/visualstudio/2016/04/08/previewing-salsa-javascript-language-service-visual-studio-15/.

[14] JavaScript Language Service in Visual Studio, tammikuu 2017. URL https://github.com/Microsoft/TypeScript/wiki/JavaScript-Language-Service-in-Visual-Studio#unsupported-patterns.

[15] Stefan Hanenberg Lars Fischer. An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability using TypeScript and JavaScript in MS Visual Studio. lokakuu 2015.

[16] TypeScript Design Goals, syyskuu 2014. URL https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals.

[17] Ryan Cavanaugh, lokakuu 2014. URL https://github.com/Microsoft/TypeScript/issues/16#issuecomment-57645069.

[18] Gabriel Isenberg Claude Pache, 2018. URL https://github.com/tc39/proposal-optional-chaining.

[19] CSE341: Programming Languages Winter 2013 Unit 6 Summary, 2013. URL https://courses.cs.washington.edu/courses/cse341/13wi/unit6notes.pdf.