
JavaScriptin staattinen tyypittäminen

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

Sisältö

1	Johdanto	1
2	Peruskäsitteitä	3
2.1	Tyyppijärjestelmien luokittelu	3
2.2	ECMA-262, EcmaScript ja JavaScript	4
2.3	TypeScript	5
2.4	Flow	5
2.5	Closure-kääntäjä	6
3	JavaScript-ohjelman muuttaminen staattisesti tyyppitarkastetuksi	7
3.1	Tyyppiannotaatiot	7
3.2	Ohjelman kääntäminen ennen suorittamista	9
3.3	Työkalun vaiheittainen käyttöönotto	11
4	Staattisen tyyppityksen hyödyt	13
4.1	Virheiden havaitseminen	13
4.2	Ohjelman optimointi käännösvaiheessa	15
4.3	Tyypin määritys dokumentaationa	16
5	Ongelmat JavaScriptin staattisessa tyyppittämisessä	18
5.1	EcmaScript-yhteensopivuus	18
5.2	Tyypin automaattinen ja vaiheittainen tyyppittäminen	19

5.3	Luotettavuus, täydellisyys ja käytännöllisyys	20
6	Yhteenveto	24
	Lähdeluettelo	25

Luku 1

Johdanto

Kaikenlaisen ohjelmoinnin keskiössä on data ja yksi tärkeimmistä datan ominaisuuksista on sen tyyppi. Muuttuja “nimi” voi olla datatypiltään 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”. Ohjelmointikielet voidaan hyvin karkeasti jakaa kahteen sen mukaan miten ja missä vaiheessa niissä käsitellään muuttujien tyyppejä.

Staattisesti tyypitetyiksi kutsutaan kieliä jotka vaativat että ohjelman käsittelemien tietorakenteiden tyypit on tulkittavissa käännös aikana, eli jo ennen ohjelman suorittamista. Lähes aina ohjelmointikieli saa tämän tiedon tietotyypeistä vaatimalla koodiin erityisiä tyypinmäärittelyitä. Ohjelman koodissa voitaisiin esimerkiksi määrittää että henkilön ikä on aina numero. Tällöin ohjelmointikielen kääntäjään rakennetut tarkistukset voivat vah-
tia jo ennen koodin suorittamista ettei ohjelmoija virheellisesti yritä asettaa henkilön iäksi tekstiä tai mitään muutakaan ei-numeerista arvoa. Dynaamisesti tyypitetyissä kielissä sen sijaan vastuu oikeiden tietotyyppien käyttämisestä jätetään ohjelmoijalle, kieli ei vaadi kehittäjältä tyyppien eksplisiittistä määrittämistä eikä niiden oikeellisuutta tarkasteta ainakaan ennen ohjelman suorittamista.

Kyseessä on kielen suunnittelullinen valinta. Kielen dynaaminen tyypittäminen vaatii yleensä vähemmän varsinaisen koodin kirjoittamista saman tuloksen saavuttamiseksi, sillä erillisiä tyypinmäärittelyitä ei tarvitse kirjoittaa. Toisaalta staattinen tyypitys mah-

dollistaa monia hyödyllisiä kehitystyökaluja ja auttaa poistamaan väärin tietotyyppien käyttämisestä johtuvan bugien osajoukon. Kielen suunnittelijan on löydettävä haluamansa tasapaino kielen ominaisuuksien välillä ja arvioitava mikä on parhaaksi niihin käyttö-tarkoituksiin joihin kielen on tarkoitus hyvin soveltua.

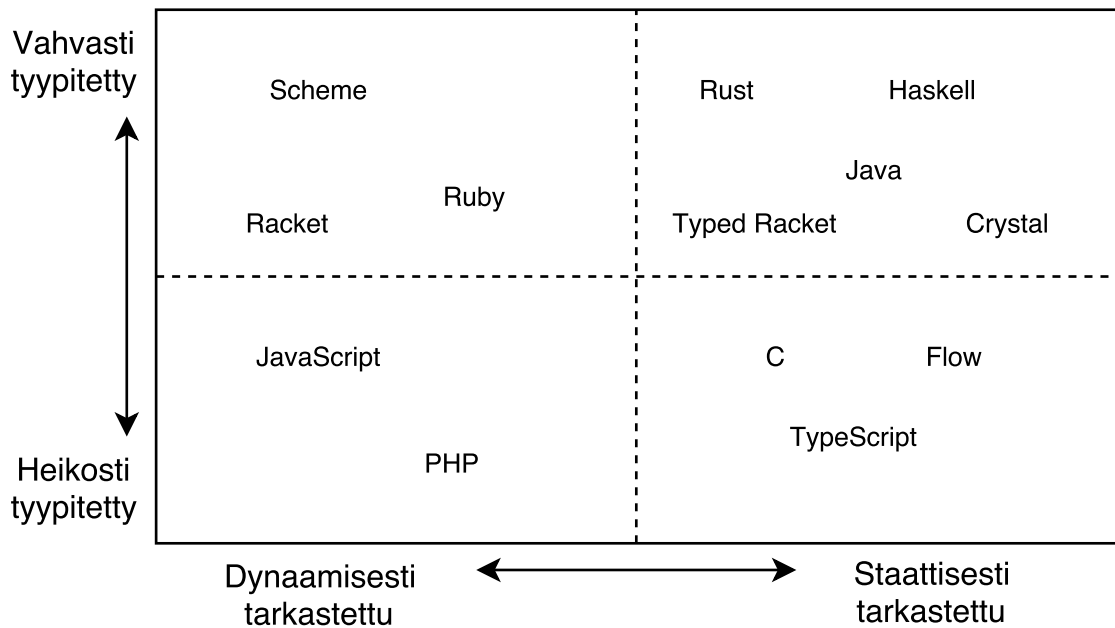
Luku 2

Peruskäsitteitä

2.1 Tyyppijärjestelmien luokittelu

Ohjelmointikielten tyyppijärjestelmien jakaminen staattisesti ja dynaamisesti tyypitarkastettuihin perustuu ohjelman kehitysvaiheeseen, jossa tarkastaminen tapahtuu. Staattisella tyypitarkastamisella viitataan ohjelman tyyppien analyysiin ennen ohjelman suorittamista, esimerkiksi käännösaikana, kun taas dynaaminen tyypitarkastus varmistaa arvojen tyyppien oikeellisuuden ohjelmaa suoritettaessa. Tyyppijärjestelmät voidaan jao- tella myös muiden ominaisuuksien perusteella esimerkiksi vahvoihin ja heikkoihin tyyp- pijärjestelmiin. Näiden termien merkitys ei ole tarkasti määritelty, mutta yleisesti niillä viitataan tapaan, jolla kieli käsittelee tarkoitettua poikkeavat, virheelliset tyypit [1]. Vah- vasti tyypitettyssä kielessä tietyn tyyppisen muuttujan vääränlainen käsittely aiheuttaisi käännös- tai ajonaikaisen virheen, kun taas heikosti tyypitettyssä kielessä arvolle voitai- siin tehdä implisiittisiä tyypimuunnoksia niiden yhteensopivuuden saavuttamiseksi.

JavaScript on dynaamisesti tarkastettu, heikosti tyypitetty kieli. Esimerkiksi ohjelma `"teksti".potenssiin(3)` antaa staattisesti tyypitarkastetussa kielessä virheen jo käännösaikana, mikäli metodia `potenssiin` ei ole tekstityypisille arvoille määri- tetty. JavaScriptiä suorittava ympäristö sen sijaan hyväksyisi ohjelman ja sallisi sen suo- rittamisen. Virhe olemattoman metodin kutsumisesta ilmenisi vasta, jos kyseinen virheen



Kuva 2.1: Tyypijärjestelmät eri ohjelmointikielissä

sisältävä koodi suoritetaan. Lisäksi esimerkiksi ilmaisu `"teksti" + 2` ei aiheuttaisi virhettä edes suoritusaikana, sillä heikoille tyypijärjestelmille ominaisesti JavaScript 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 tukevat kielen sallimia operaatioita siten, että esimerkiksi yllä esitettyä `"teksti" + 2` ohjelmaa ei enää sallittaisi. Monia muita heikoille tyypijärjestelmille 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 kah-

deksannen 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 (superset) [4] ja jatkaa JavaScriptin syntaksia tyyppimäärittelyihin käytettävällä annotaatio syntaksilla. 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 suoritussympäristöissä. TypeScript-kääntäjän voi myös määrittää muokkaamaan tulostettava koodi yhteensopivaksi vanhojen EcmaScript-standardien kanssa, mikä on hyödyllistä, jos ohjelman on tarkoitus tukea sellaisia suoritussympäristöjä, 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 tyyppimäärittelyt 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 annotaatiot 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 tyylin automaattisesti päättelemällä tai kieleen sisältyvien eksplisiittisten tyyppimäärittelyjen perusteella. Kaikki kolme tässä tutkielmassa esiteltyä JavaScriptin staattiseen tyyppitarkastukseen tarkoitettua työkalua päättelevät muuttujien tyyppiä automaattisesti, mutta vaativat paikoitellen myös eksplisiittisiä määrittelyksiä. Closure-kääntäjä lukee tyyppimäärittelyt 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 kommentteilla, 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.

Closures annotaatiot on sijoitettu kommentteihin, joten niillä ei ole ajonaikaista vaikutusta ja ohjelma on täten sellaisenaan hyväksyttävää JavaScriptiä. Toisaalta tyyppiannotaatioiden määrittely kommentteissa 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:

```
1  /**
2  * @typedef {{
3  *   nimi: string,
4  *   hinta: number
5  * }}
6  */
7  let Ostos;
```

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

```
1  type Ostos = {
2    nimi: string;
3    hinta: number;
4  };
```

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

3.2 Ohjelman kääntäminen ennen suorittamista

JavaScript koodi tulkitaan tai käännetään tyypillisesti suorittamisen yhteydessä, selaimesta tai muusta suoritussympäristöstä 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 käyttäminen ei normaalisti vaadi erillistä käännösvaihetta, useissa projekteissa ei ole sellaista käytetty. Koodin minimointi ja muu optimointi on ollut parhaiden käytäntöjen mukaista



Kuva 3.1: Vaihtoehtoisia käännösprosesseja

jo tovin, mutta tällaiset koodinkäsittelyt tehdään yleensä vasta ennen ohjelman julkaisua. Kehittäjät ovat tavanomaisesti voineet suorittaa kirjoittamansa JavaScriptin sellaiseen kehitysympäristöön. Käännösvaiheen aikavaatimus pyritään luonnollisesti pitämään mahdollisimman pienenä, mutta se on silti projektin monimutkaisuuteen ja kehitysnopeuteen vaikuttava tekijä, tulee tuleen huomioida työkalun käyttöönotossa.

Kuva 3.1 esittää erilaisia käännösprosesseja eri JavaScript-ohjelman kehityskielille. Jos koodi kirjoitetaan kaikkiin suoritussympäristöihin soveltuvalla kielen versiolla, kuten EcmaScriptin versiolla 5 (2011), käännösvaihe voidaan jättää kokonaan väliin. Kielen uudemmat versiot ovat kuitenkin tuoneet monia lisäominaisuuksia, joiden käyttäminen vanhempiä selaimia tukevissa ohjelmissa vaatii joka tapauksessa käännösvaiheen staattisesta tyyppittämisestä 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 595 000. 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 tyyppianotoidulla 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 /** Laittaa tuotteen ostoskoriin. */
2 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.lisaaTuote({ 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 yhteenspi-
 vuus täysin tyyppitarkastamattoman koodin kanssa. Sekä Flow että TypeScript tarjoavat erityisen yleisviittaustyyppin Any, jota voi käyttää kuvaamaan mitä tahansa JavaScript ar-
 voa [4]. Any-tyyppiseen muuttujaan voidaan asettaa mikä tahansa arvo ja Any tyyppinen arvo voidaan asettaa mihin tahansa muuttujaan tai funktioparametriin. Any tyyppin avulla muuten staattisesti tyyppitetystä ohjelmassa voidaan ohittaa käännösaikainen tyyppien tar-
 kistaminen sellaisten koodin osien kohdalla joiden ajonaikaista arvoa olisi muuten vaikea tai mahdotonta määrittellä käännösaikana. Näin ollen, mikäli tarve vaatii täysin annotoi-
 mattoman JavaScript moduulin käyttämistä, kaikkien kyseisestä moduulista tuotujen ar-
 vojen voidaan määrittää olevan tyyppiä Any, jolloin tarkastaja ei kiinnitä huomiota siihen miten JavaScriptillä määritettyjä funktioita tai muita arvoja käsitellään.

Luku 4

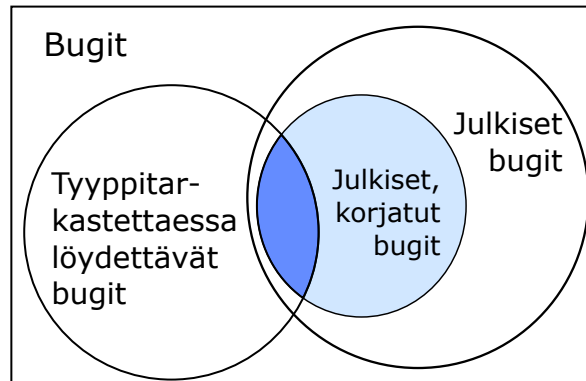
Staattisen tyyppityksen hyödyt

4.1 Virheiden havaitseminen

Kenties tärkein staattisen tyyppijärjestelmän tehtävä on havaita ja estää ohjelmoijan virheitä. Tässä esitelty 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 suoritusajasta 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



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

lähdekoodin JavaScript-projekteihin havaittiin, että vähintään 15% ilmoitetuista ja korjatuista bugeista olisi voitu havaita ja välttää jos projektin kehitykseen oltaisiin käytetty jompaakumpaa 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 tyypitarkastus 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 tyypitarkastus 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 ja jäisi helposti dynaamisessa testaamisessa huomaamatta:

```
1 if (viikonpaiva === "perjantai") {
2   // Alennus perjantaisin 5 euroa
3   ostoskori.lisaaTuote({ nimi: "astiasto", Hinta: 10 });
4 } else {
5   ostoskori.lisaaTuote({ 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 tietyissä olosuhteissa, se voi pysyä havaitsemattomana pitkään. Staattiselle analyysille konditionaalisen ohjelman tarkistaminen ei kuitenkaan ole ongelma ja tyyppiannotoituina kaikki kolme työkalua pystyvätkin osoittamaan esimerkissä olevan virheen.

4.2 Ohjelman optimointi käännösvaiheessa

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

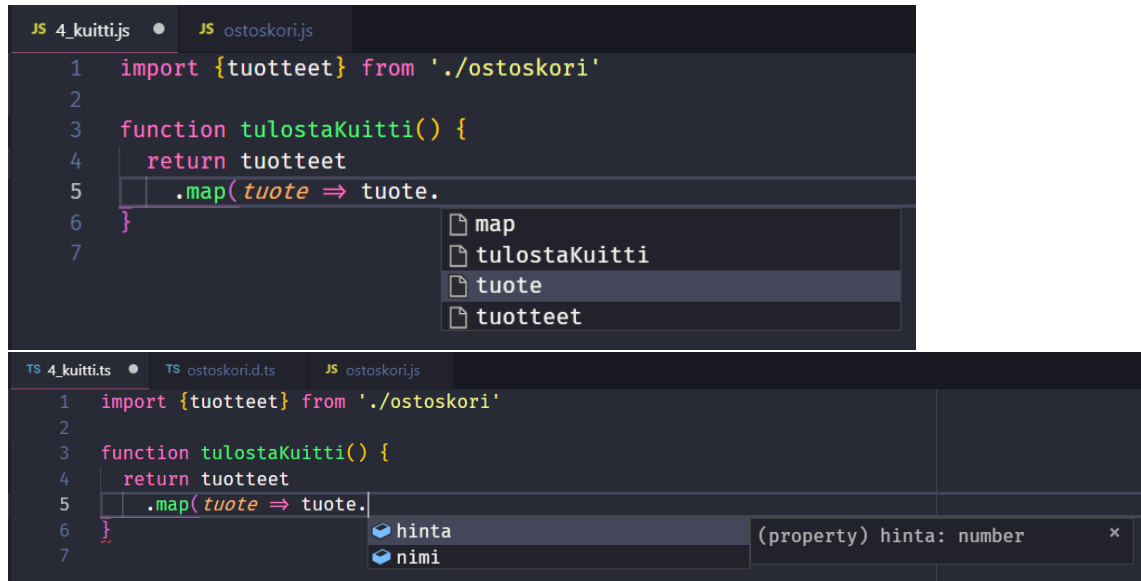
Kun JavaScriptia optimoidaan käännösvaiheessa, on kuitenkin usein hyödyllistä kiinnittää enemmän huomiota tuotetun koodin kokoon kuin suoritusajaiseen tehokkuuteen. Tyypillinen JavaScript-ohjelma ladataan sivulle saapuessa internet-yhteyden yli juuri ennen suorittamista, minkä vuoksi ladattavan koodin koon kasvattaminen minimaalisen suoritusajan edun vuoksi ei usein ole kokonaisuudessaan hyödyllistä. Closure compiler on kehitetty erityisesti JavaScript-koodin koon optimointia ajatellen. Muuttujanimet voidaan helposti uudelleennimetä ilmankin staattisen tyyppityksen 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 Tyypimäärittelyt 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 metodinimien 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 tyyppitetyissä 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 tyypejä ajonaikana [13, 14]. Tämä tekniikka yhdistettynä tavanomaisempaan tyyppien käännoaikaiseen päättelyyn voi riittää tarjoamaan melko kattavan kuvauksen jonkin muuttujan tyypestä, mutta jää silti jälkeen siitä tarkkuudesta jonka IntelliSense osaa antaa staattisesti tyyppitetyle 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 dokumentaationa jotta työtehokkuus kasvaisi merkittävästi. Vuonna 2015 toteutettu tutki-



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

mus “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 tyyppityksen 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 tärkeitä katsottuja ominaisuuksia, jotka päätettiin lisätä 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 joillekin 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 navigation) [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 tyyppityksen tuova työkalu olisi varteenotettava vaihtoehto dynaamisesti tyyppitetyn JavaScriptin rinnalla, sen kirjoittaminen ei voi vaatia liian paljon enemmän työtä kuin JavaScriptin. Staattisessa tyyppijärjestelmässä näkyvin ero dynaamisesti tyyppitettyyn on eksplisiittiset tyyppiannotaatiot, jotka voivat olla lisärasite koodin kirjoittajalle. Erityisesti vanhaa, dynaamisesti tyyppitettyä JavaScriptia refaktoroidessa staattisesti tyyppitettyyn 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 tyypimäärittelyitä.

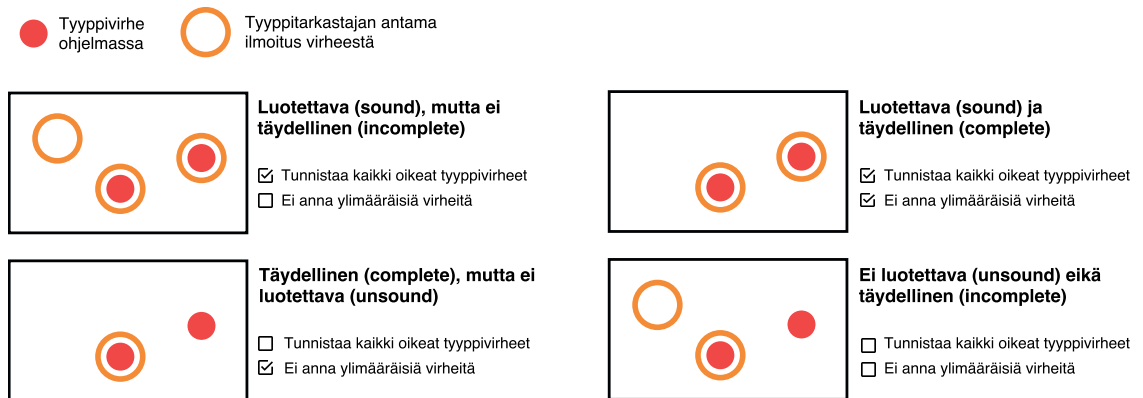
Esimerkissä 5.2 Flow pystyy päättämään tyypin

```
vertaaHintoja: (a: Ostos, b: Ostos) => number
```

automaattisesti, vaikkei esimerkissä ole yhtään eksplisiittistä tyypimäärittelyä tai muuta tavallisesta JavaScriptistä poikkeavaa. Flow tietää ennestään mikä listan sort-metodin tyyppi on, joten se pystyy päättämään myös sort-metodille annetun vertaaHintoja-funktion tyypin käyttämällä *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 päättämään että funktion palautusarvo on numero, eikä eksplisiittistä `: number` palautusarvon määrittystä 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].



Kuva 5.1: Tyypijärjestelmän luotettavuus ja täydellisyys

Jotta JavaScriptiä analysoiva tyypijärjestelmä olisi luotettava, sen on annettava virhe esimerkiksi seuraavasta ohjelmasta:

```

1 function osta(ostos) {
2   lisääTuote({
3     nimi: ostos.nimi,
4     hinta: ostos.hinta
5   });
6 }
7
8 osta({ nimi: 'juusto', hinta: 5 });
9 osta({ hinta: 5 });
    
```

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

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


```
1 function osta(ostos) {  
2   if (typeof ostos.nimi === 'string') {  
3     lisaaTuote({  
4       nimi: ostos.nimi,  
5       hinta: ostos.hinta  
6     });  
7   }  
8 }  
9  
10 osta({ nimi: 'juusto', hinta: 5 });  
11 osta({ hinta: 5 });
```

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

Esimerkit 5.3 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 tyyppitys olemassa olevaan kieleen, JavaScriptiin, siten että nykyisellään käytössä olevat kirjastot ja koodikäytännöt pystytään tyyppitarkastamaan ilman että niiden arkkitehtuuria tarvitsee merkittävästi muuttaa tyyppiturvallisuuden saavuttamiseksi.

TypeScript on struktuurallisesti tyyppitetty, minkä vuoksi seuraava koodi kääntyy ilman tyyppivirheitä.

```
1 class Ihminen {
2     constructor(public nimi: string){}
3 }
4
5 class Elain {
6     constructor(public nimi: string){}
7 }
8
9 function varaaAikaElainlaakariin(omistaja: Ihminen, lemmikki: Elain)
10 {
11     ...
12 }
13
14 varaaAikaElainlaakariin(new Elain("Musti"), new Ihminen("Jaakko"));
```

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

TypeScript-kääntäjä sallii esimerkin 5.3 koodin vaikka argumentit "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 luokainstanssit on tyypitetty nominaalisesti, mikä auttaa tässä esimerkissä mutta aiheuttaa 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>.