
JavaScriptin staattinen tyypittäminen

LuK -tutkielma
Turun yliopisto
Tulevaisuuden teknologioiden laitos
Tietojenkäsittelytiede
2019
Oskari Noppa

Sisältö

1	Johdanto	1
2	Peruskäsitteitä	3
2.1	Tyyppijärjestelmät	3
2.2	Luokittelu	3
2.3	EcmaScript ja JavaScript	5
2.4	TypeScript	5
2.5	Flow	6
2.6	Closure	6
3	Käyttöönotto	7
3.1	Tyoppiannotaatiot	7
3.2	Käännösprosessi	9
3.3	Työkalun vaiheittainen käyttöönotto	10
4	Staattisen tyypitarkastuksen hyödyt	14
4.1	Virheiden havaitseminen	14
4.2	Ohjelman optimointi käännösvaiheessa	16
4.3	Tyypinmäärittelyt dokumentaationa	18
5	Tyypitarkastamisen ongelmat	21
5.1	EcmaScript-yhteensopivuus	21

5.2	Automaattinen ja vaiheittainen tyypittäminen	22
5.3	Luotettavuus, täydellisyys ja käytännöllisyys	23
6	Yhteenveto	27
	Lähdeluettelo	29

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 olisi järkevä jos henkilön iäksi sallittaisiin "Matti". iän olevan "Matti". Se miten ohjelmointikielissä käsitellään arvojen tyyppejä vaihtelee kuitenkin suuresti.

Tässä tutkielmassa käsitellään JavaScript-ohjelmointikieltä, sekä kolmea työkalua jotka rakentavat staattisesti tarkastettavan tyyppijärjestelmän JavaScriptin päälle.

JavaScriptin 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 tarkoituspää suuremmaksi, kun sillä on alettu toteuttaa esimerkiksi kartta-, kirjoitus- ja hallintapalveluita jotka toimivat selaimessa, siten ettei käyttäjän tarvitse asentaa erillistä tietokoneohjelmaa palvelun käyttöön. JavaScriptin käyttö on levinnyt verkkosivujen asiakaspuolen käyttöliittymän toteutuksesta tietokone- ja älypuhelinsovelluksiin sekä verkon asiakas-palvelin-arkkitehtuurissa myös palvelinsovelluksien kieleksi.

Tutkielmassa esitellään TypeScript, Flow ja Closure-kääntäjä, joista jokainen on tarkoitettu työkaluksi sellaisten ohjelmien kehittämiseen, jotka muuten kehitettäisiin JavaScriptillä. Päämääränä on tarkastella kuinka *dynaamisesti tyyppitetty* JavaScript voidaan täydentää staattisesti tyyppitarkastetuksi TypeScriptin, Flow'n, tai Closure-kääntäjän avulla, sekä mitä hyötyä siitä voi olla. Työkalujen hyötyjä ja haittoja vertaillaan sekä toi-

siinsa että tavalliseen JavaScript-koodiin.

Luku 2

Peruskäsitteitä

2.1 Tyyppijärjestelmät

Tyyppijärjestelmät, tai *tyyppiteoria*, on tutkimusalue matematiikan ja filosofian alalla [1]. Tarkka "tyyppijärjestelmän" määritelmä riippuu siitä minkä tieteenalan ja -haaran näkökulmasta sitä käsitellään. Tässä tutkielmassa tyyppijärjestelmiä tarkastellaan käytännönläheisesti ohjelmointikielten näkökulmasta metodina, jolla voidaan osoittaa virheellisissä ohjelmissa olevia puutteita luokittelemalla ohjelman rakenteita tyypeiksi ja vertaamalla niitä ohjelmointikielen tyypisääntöjä vasten.

2.2 Luokittelu

Ohjelmointikielten tyyppijärjestelmien jakaminen staattisesti ja dynaamisesti tyypitarkastettuihin (puhekielessä usein: staattisesti ja dynaamisesti "tyypitettyihin") 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.

Esimerkiksi ohjelma `"teksti".potenssiin(3)` antaa staattisesti tyypitarkastetussa kielessä virheen jo käännösaikana, mikäli metodia `potenssiin` ei ole tekstityypisille arvoille määritetty. Staattisesti tyypitarkastetussa kielessä muuttujien mahdolliset tyypit analysoidaan käännösaikana, ennen ohjelman suorittamista. Kääntäjä nä-



Kuva 2.1: Tyypijärjestelmät eri ohjelmointikielissä

kee, että lausekkeen vasen puoli, `"teksti"`, on tyypiltään teksti ja oikea puoli on metodin `potenssiin` kutsu. Kääntäjä voi todeta kielen tyoppisääntöjen perusteella, ettei tekstimuotoisessa arvossa ole metodia `"potenssiin"` ja pysäyttää virheellisen ohjelman käsittelyn.

Tyypijärjestelmät voidaan jaotella myös muiden ominaisuuksien perusteella esimerkiksi vahvoihin ja heikkoihin tyypijärjestelmiin. Näiden termien merkitys ei ole tarkasti määritelty, mutta yleisesti niillä viitataan tapaan, jolla kieli käsittelee tarkoitettusta poikkeavat, virheelliset tyypit [2]. Vahvasti tyypitettyssä kielessä tietyn tyyppisen muuttujan vääränlainen käsittely aiheuttaisi käännös- tai ajonaikaisen virheen, kun taas heikosti tyypitettyssä kielessä arvolle voitaisiin tehdä implisiittisiä tyyppimuunnoksia niiden yhteensopivuuden saavuttamiseksi.

JavaScript on dynaamisesti tarkastettu, heikosti tyypitetty kieli. JavaScriptiä suorittava ympäristö hyväksyisi ohjelman ja sallisi sen suorittamisen. Virhe olemattoman metodin kutsumisesta ilmenisi vasta jos ohjelmaa testataan käytännössä ja kyseinen virheen sisältävä osa koodia suoritetaan. Lisäksi esimerkiksi lausekkeen `"teksti" + 2` laskeminen ei aiheuttaisi virhettä edes suoritusaikana, sillä heikoille tyypijärjestelmille ominai-

sesti JavaScript muuttaisi numeron 2 tekstimuotoon ennen summausoperaation arviointia ja antaisi tulokseksi `"teksti2"`, mikä ei välttämättä ollut koodin alkuperäinen tarkoitus. 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` lauseketta ei enää sallittaisi.

2.3 EcmaScript ja JavaScript

EcmaScript on ECMA-262 standardin määrittelemä ohjelmointikieli [3, 4], jonka kehityksestä vastaa organisaatio Ecma International. *JavaScript* puolestaan on Oraclen omistama tavaramerkki jolla viitataan EcmaScript-kielen osittaisiin tai täydellisiin toteutuksiin [3]. Historiallisista syistä termejä “JavaScript” ja “EcmaScript” käytetään usein keskenään vaihtokelpoisesti. Tässä tutkielmassa termillä “JavaScript” viitataan ECMA-262-spesifikaation kahdeksannen version mukaiseen EcmaScriptiin, jota kutsutaan myös nimellä EcmaScript 2017.

2.4 TypeScript

TypeScript on Microsoftin luoma ohjelmointikieli, jonka tarkoitus on auttaa JavaScript-ohjelmien kehitystä staattisen tyyppijärjestelmän avulla. Se on EcmaScriptin ylijoukko (engl. superset) [5] ja jatkaa JavaScriptin syntaksia tyyppimäärittelyihin käytettävällä annotaationsyntaksilla. Jokainen validi JavaScript-ohjelma on syntaksiltaan ja ajon aikaiselta 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.

2.5 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 [6].

2.6 Closure

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 [7]. Tyyppit 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 [8]. Kehittäjä voi suorittaa koodin sellaisenaan ilman aikaavievää käännösprosessia. Kun ohjelma on valmis, se voidaan haluttaessa ajaa Closure-kääntäjän läpi tiedostokoon ja suoritussopeuden optimoinniksi.

Luku 3

Käyttöönotto

3.1 Tyyppiannotaatiot

Tyyppijärjestelmä voi päätellä muuttujan sallitun tyypin automaattisesti tai kielen syntaksin tarjoamien eksplisiittisten tyypinmäärittelyjen perusteella. Kaikki kolme tässä tutkielmassa esiteltyä JavaScriptin staattiseen tyypitarkastukseen tarkoitettua työkalua päättävät muuttujien tyyppejä automaattisesti, mutta vaativat paikoitellen myös eksplisiittisiä määrittäyksiä. Closure-kääntäjä lukee tyypinmäärittelyt JSDoc-tyylisistä dokumentaatiokommenteista [8].

```
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 tyypinmäärittely 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. Flow ja TypeScript -esimerkissä 3.2 tyyppiannotaatiot ovat osana koodia, mikä tekee ohjelmasta yhteensopimattoman tavallisen JavaScriptin kanssa. Ohjelma on käännettävä JavaScriptiksi en-

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

Listaus 3.2: Esimerkki Flow tai TypeScript annotaatiosta funktiolle

nen suorittamista. Annotaatioiden syntaksi ja merkitys eivät myöskään ole välttämättä suoraan selviä 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 kommentteissa voi olla runsassanaista ja hankalaa, mikä kasvattaa niiden kirjoittamiseen vaadittua työmäärää [5, 9].

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;
```

Listaus 3.3: Uuden tyyppin määrittely Closures kommenttisyntaksilla

TypeScript ja Flow tarjoavat käännoaikaisen tyyppin (type alias) määrittelyyn tiiviimmän ja helppolukuisemman syntaksin [5]:

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

Listaus 3.4: Uuden tyyppin määrittely TypeScriptissä tai Flow'ssa

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

3.2 Käännösprosessi



Kuva 3.1: Vaihtoehtoisia käännösprosesseja

JavaScriptillä kirjoitettua koodia ei tavallisesti tarvitse erikseen kääntää ennen suoritusta. JavaScriptiä suorittava ohjelma, esimerkiksi selain, tulkkaa EcmaScript-standardin mukaista koodia sellaisenaan tai *JIT-kääntää* sen optimoituun muotoon automaattisesti suorituksen ohessa. TypeScript- tai Flow-annotoitu koodi ei kuitenkaan ole

validia JavaScriptiä, eikä selain tai muu JavaScriptia suorittamaan suunniteltu ohjelma tavallisesti osaa sellaista koodia käsitellä. Näinollen TypeScript tai Flow-annotoitu koodi on välttämätöntä kääntää muotoon jossa annotaatiot on poistettu ja jäljellä on enää standardinmukainen JavaScript.

Kääntämisen tarve vaikuttaa ohjelman kehitysprosessiin. Koska JavaScriptin käyttäminen 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 sellaiseen 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. Käännösvaiheen tarpeellisuus ja vaikutus kehitysprosessiin riippuu käytetystä kielestä ja sen versiosta, sekä siitä missä ympäristöissä sitä halutaan suorittaa. Esimerkiksi Internet Explorer 11 -selain tukee ainoastaan EcmaScriptin vanhaa viidettä versiota (2011). Jos koodi kirjoitetaan kaikkiin suoritussympäristöihin soveltuvalla EcmaScriptin versiolle, käännösvaihe voidaan jättää kokonaan väliin. Kielen uudemmat versiot ovat kuitenkin tuoneet monia lisäominaisuuksia, kuten syntaksin *luokkien* määrittelyyn, joiden käyttäminen vanhempia 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 ohjelmistokekosysteemeistä [10]. Tämän tutkielman kirjoittamisen hetkellä rekisterin kotisivu, npmjs.com, ilmoitti julkisesti saatavilla olevien pakettien määräksi yli 800 000. Avoi-

messa jakelussa olevien kirjastojen lisäksi JavaScriptiä käytävillä 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` (Listaus 3.5).

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

Listaus 3.5: Esimerkki TypeScript määrittelytiedostosta `ostoskori.d.ts`

Tämän jälkeen TypeScript tiedostosta käsin voidaan kutsua tätä JavaScript-funktiota siten, että TypeScript valvoo tyyppien oikeellisuutta (Listaus 3.6).

```
1 import * as ostoskori from "./ostoskori";
2
3 ostoskori.lisääTuote({ nimi: "juusto", hinta: 5 });
4 // Vääränlainen kutsumistapa aiheuttaisi käännoaikaisen virheen:
5 ostoskori.lisääTuote('juusto', 5); // Expected 1 arguments, but got 2
```

Listaus 3.6: JavaScript-koodin kutsuminen TypeScript tiedostosta `tuotesivu.ts`

JavaScript-kirjaston kehittäjä voi tarjota erillisen tyyppitiedoston kirjastonsa lähdekoodin mukana, tai muut kirjastoa käyttävät kehittäjät voivat oma-aloitteisesti luoda sellaisen tyyppitiedostoja keräävään julkaisuarkistoon (engl. repository). TypeScriptin tyyppitys-

tiedostoille on *DefinitelyTyped* [11] ja Flowlle vastaavasti *flow-typed* [12]. Vaikka Flow ja TypeScript ovatkin monin tavoin hyvin samankaltaisia, eivät ne kuitenkaan ole täysin yhteensopivia. Yhdelle tyyppijärjestelmälle luotuja annotaatiotiedostoja ei siis yleensä voi suoraan käyttää toisella järjestelmällä, joskin niiden muuntaminen toisen tyyppijärjestelmän käytettäväksi on useimmiten melko suoraviivaista ja ainakin osin automatisoitavissa työkaluilla kuten *flowgen* (TypeScript-tyypitykset Flow-tyypityksiksi) [13], *clutz* (Closure-tyypitykset TypeScript-tyypityksiksi) [14] ja *tsickle* (TypeScript-annotoitu koodi Closure-annotoiduksi) [15]. Joitain perustavanlaatuisia eroja työkalujen välillä voi silti olla vaikea tai mahdotonta siirtää tyyppijärjestelmästä toiseen tekemättä kompromisseja. TypeScriptissä esimerkiksi kaikki *enum*ejä lukuunottamatta on *rakenteellisesti* (engl. structural) tyypitetty, kun taas Flow:ssa esimerkiksi luokkien instanssien tyypejä vertaillaan *nimellisesti* (engl. nominal), mikä aiheuttaa tyyppijärjestelmien käyttäytymisessä eroja jos jotkin kaksi luokkaa ovat rakenteeltaan täysin samat. Tämän tutkielman viimeisestä luvusta löytyy esimerkki 5.4 juuri tällaisesta tilanteesta; siinä luokat `Ihminen` ja `Eläin` ovat rakenteeltaan samanlaisia ja käsitellään siksi TypeScriptissä kuin ne olisivat sama luokka, mutta Flow:ssa kahtena erillisenä luokkana jotka eivät ole keskenään vaihtokelpoisia.

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. JavaScript-kirjaston rajapinta voi olla liian iso tyypitettäväksi projektin aikatauluun sopivalla tahdilla, tai se saattaa olla suunniteltu käyttämään sellaisia dynaamisia JavaScriptin ominaisuuksia joita ei helposti voida tyypittää TypeScriptin, Flow'n tai Closuresn tarjoamalla tyyppijärjestelmällä. Kaikkien kolmen työkalun tärkeimpiin ominaisuuksiin kuuluu tuki vaihteittaiselle käyttöönnotolle, eli käytännössä yhteenspivuus täysin tyyppitarkastamattoman koodin kanssa.

Sekä Flow että TypeScript tarjoavat erityisen yleisviittaustyyppin `Any`, jota voi käyttää kuvaamaan mitä tahansa JavaScript-arvoa [5]. `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 määrittellä 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.

```
1 const koriElementti: null | HTMLElement =
2   document.getElementById('kori')
3 // Tiedetään että elementti on dokumentissa eikä null.
4 const koriElementtiEiNull: HTMLElement = koriElementti as any
5 koriElementtiEiNull.innerHTML = '<h1>Ostoskori</h1>'
```

Listaus 3.7: Esimerkki Any-tyypillä sivuutetusta TypeScript virheestä

Esimerkin 3.7 `getElementById` -funktio kutsu palauttaisi `null` jos kori elementtiä ei löytyisi dokumentista, mutta virhe on päätetty ohittaa määrittämällä eksplisiittisesti muuttujan tyyppiä ensin `any` ja sitten `any` tyyppistä `HTMLElement` tyyppiä. Flow sallii tyyppivirheiden sivuuttamisen myös erityisillä kommentilla.

```
1 const koriElementti: null | HTMLElement =
2   document.getElementById('kori')
3 // $FlowFixMe Tiedetään että elementti on dokumentissa eikä null.
4 const koriElementtiEiNull: HTMLElement = koriElementti
5 koriElementtiEiNull.innerHTML = '<h1>Ostoskori</h1>'
```

Listaus 3.8: Esimerkki kommentilla sivuutetusta Flow virheestä

Esimerkissä 3.8 on sama tilanne kuin esimerkissä 3.7, ohjelmassa on päätetty sivuuttaa `null` -palautusarvon mahdollisuus. Tällä kertaa muuttujaa ei ole konvertoitu `any` -muotoon, vaan turvattomasta tyyppimuunnoksesta normaalisti annettava käännösvirhe on sivuutettu `$FlowFixMe` -kommentilla.

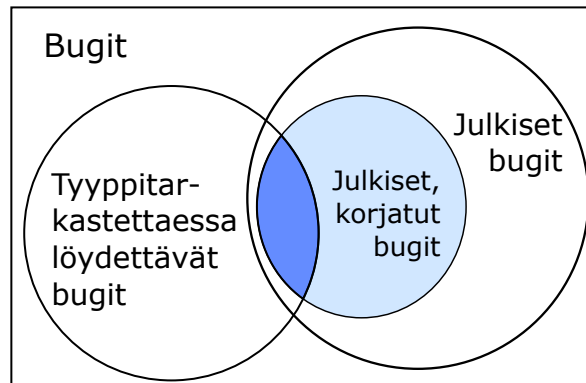
Luku 4

Staattisen tyyppitarkastuksen hyödyt

4.1 Virheiden havaitseminen

Kenties tärkein tyyppijärjestelmän tehtävä on havaita ja estää ohjelmoijan virheitä. Tyypipisäännöt eivät kykene tunnistamaan yleisesti kaikenlaisia ongelmia koodissa — bugin lähteenä voi esimerkiksi olla väärinymmärrys, jossa ohjelman vaatimukset on tulkittu ja toteutettu väärin. Vaikka tyyppijärjestelmät eivät kykenisikään ehkäisemään kaikkia virheitä, niillä voidaan kuitenkin havaita tietyn luokan virheet systemaattisesti. Johdannossa annettiin esimerkki ohjelman virheellisestä tilasta, jossa henkilön ikää kuvaavaan muuttujaan oli päätnyt tekstimuotoinen arvo, "Matti". Tämä on yksinkertainen esimerkki virheestä joka olisi luultavasti helposti havaittavissa tyyppijärjestelmän avulla. Myös dynaamiset tyyppijärjestelmät estävät tällaiset tyyppivirheet suoritusaikana, mutta vasta jos virheen sisältävä osa koodia suoritetaan. Tässä tutkielmassa esitelty työkalut, mahdollisesti Closure-kääntäjää lukuunottamatta, onkin kehitetty erityisesti estämään tyyppivirheitä osana kehitysprosessia, jo ennen koodin suorittamista.

Kaikki kolme työkalua antaisivat käännösvirheen, jos esimerkeissä 3.1 ja 3.2 esiteltyä ostoskorifunktiota kutsuttaisiin virheellisesti esimerkiksi listalla hintaa kuvaavia numeroita, sillä funktion parametrin on annotoitu olevan lista `Ostos`-tyyppimääritelmän mukaisia objekteja. Koska JavaScript on paitsi dynaamisesti myös heikosti tyyplitetty, `ostoskorinHinta([5, 10, 15])` ei itse asiassa aiheuttaisi suoritettaessa ohjelman keskeyttävää virhettä. Ilmaisuuksella `ostos.hinta` on sallittu vaikka muuttuja `ostos` olisikin



Kuva 4.1: To Type or Not to Type: Quantifying Detectable Bugs in JavaScript -tutkimuksen käsittelemät ohjelmistovirheet [17].

arvoltaan numero eikä `Ostos`-olio. Tällöin ilmaisun arvo olisi `undefined` ja lausekkeen `summa += ostos.hinta` jälkeen `summa`-muuttujan arvo olisi erityinen ei-numeroa kuvaava `NaN` [16]. Tyypitarkistamisen 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 [17] TypeScriptin ja Flown vaikutuksesta avoimen lähdekoodin JavaScript-projekteihin havaittiin, että vähintään 15% ilmoitetuista ja korjatuista bugeista olisi voitu havaita ja välttää jos projektin kehitykseen oltaisiin käytetty jompaakumpaa näistä työkaluista. 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 tyyppitettyyn muotoon ja sitten mittaamalla kuinka hyvin tyypitarkastus havaitsi ennalta tunnettuja bugeja aiheuttavan koodin. Sen ulkopuolelle jäivät ohjelmistovirheet joita ei oltu vielä korjattu tai havaittu, sekä ohjelmistovirheet jotka kehittäjä oli havainnut jossain kehitysvaiheessa ennen virheellisen ohjelman julkaisua. Staattisen tyypitarkastus luultavasti auttaisi vähentämään myös näitä bugeja.

Kuva 4.1 luokittelee tutkimuksessa [17] kiinnostuksen kohteena olleet ohjelmointivir-

heet. Tutkimuksessa käsiteltiin ainoastaan julkisia, jo korjattuja bugeja. Suuri osa ohjelmointivirheistä jää kuitenkin luultavasti tämän joukon ulkopuolelle, sillä ohjelmissa voi olla bugeja joita kehittäjät eivät ole vielä havainneet ja joista käyttäjät eivät ole ilmoittaneet. Esimerkiksi ohjelma 4.1 saattaisi aiheuttaa vaikeasti havaittavan virheen joka jäisi helposti huomaamatta manuaalisesti tehtävässä testauksessa ja jopa yksikkötesteissä

```
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 4.1 ei toimi oikein perjantaisin, sillä `Ostos`-tyyppisen objektin ominaisuus `hinta` on virheellisesti kirjoitettu isolla alkukirjaimella. Koska ohjelmistovirhe toistuu vain tietyissä olosuhteissa, se voi pysyä havaitsemattomana pitkään. Staattinen analyysi etsii tyyppivirheitä kaikista koodin haaroista ja tyyppiannotoituina kaikki kolme työkalua pystyvätkin osoittamaan esimerkissä olevan virheen.

4.2 Ohjelman optimointi käännösvaiheessa

Aivan ensimmäiset tyyppijärjestelmät, kuten Fortranin (1957) staattinen tyyppitys, kehitettiin laskutoimitusten suoritusajan optimointia varten [1]. Myös uudemmissa kielissä tyypeistä saatavilla olevaa tietoa voidaan käyttää ajonaikaisen esimerkiksi turvallisuuden varmistavien tarkistusten optimointiin.

Kun JavaScriptiä optimoidaan käännösvaiheessa, on kuitenkin usein hyödyllistä kiinnittää enemmän huomiota tuotetun koodin kokoon kuin suoritusajaiseen tehokkuuteen. JavaScriptin yleisin käyttökohde ovat verkkosivut, joissa selain lataa lähdekoodin sivulle saapuessa internet-yhteyden yli juuri ennen ohjelman suorittamista. Ladattavan JavaScript-koodin määrä vaikuttaa siihen kauanko sivun lataus kestää, joten 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 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).

```
1 class Hintalaskuri {
2   constructor() {
3     this.ostostenHintaYhteensa = 0
4     this.arvonlisavero = 0.14
5   }
6   lisääTuote(hinta) {
7     this.ostostenHintaYhteensa += hinta
8   }
9   yhteensa() {
10    return this.ostostenHintaYhteensa +
11           this.ostostenHintaYhteensa * this.arvonlisavero
12   }
13 }
14 const laskuri = new Hintalaskuri()
15 laskuri.lisääTuote(5)
16 console.log(laskuri.yhteensa())
```

Listaus 4.2: Esimerkki JavaScript-koodista jota Closure osaa optimoida

Esimerkissä 4.2 on yksinkertainen JavaScript-tiedosto, jota Closure voi optimoida kooltaan pienemmäksi ja ajonaikaiselta suorituskyvyltään nopeammaksi.

```
1 var $laskuri$$=new function(){this.$a$=0};
2 $laskuri$$.$a$+=5;
3 console.log($laskuri$$.$a$+.14*$laskuri$$.$a$);
```

Listaus 4.3: Closures optimoima JavaScript-koodi

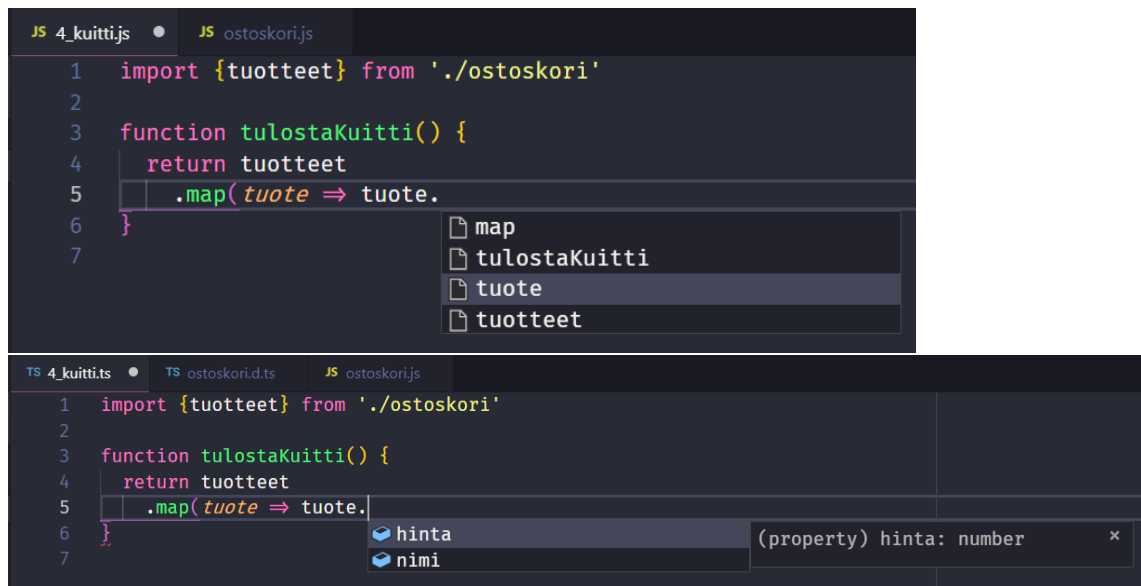
Listauksessa 4.3 näkyy Closures (versio 20190215.0.2) tuottama koodi. Pitkä luokan `Hintalaskuri` jäsenmuuttuja `ostostenHintaYhteensa` on uudelleennimetty lyhyeen muotoon `a`. Closure näkee että luokan `Hintalaskuri` metodeja `lisääTuote` ja `yhteensa` kutsutaan vain yhdessä kohdassa koodia, joten se on

siirtänyt metodien ohjeet luokan ulkopuolelle, alkuperäiselle kutsupaikalle, ja muuttanut parametriviittaukset viittauksiksi alkuperäisen kutsun argumentteihin.

`arvonlisavero`-luokkamuuttuja on poistettu kokonaan ja korvattu käyttöpaikalla `.14 numeroliteraalilla`, mikä paitsi pienentää koodin kokoa myös saattaa parantaa suorituskenaikaista tehokkuutta, sillä koodia suorittavan moottorin ei tarvitse noutaa `arvonlisavero`-luokkamuuttujaa luokan *instancsista*. Käännöksen tulos on vaikeasti luettavaa, mutta se onkin tarkoitettu JavaScript-moottorin suoritettavaksi eikä ohjelmoijan luettavaksi. Ongelmatilanteissa, lopullista koodia tutkittaessa, käännetty koodi voidaan palauttaa alkuperäiseen muotoonsa ohjelmoijan luettavaksi käyttämällä *lähdekarttoja* (engl. source map) [18, 19], jotka kuvaavat minimoidun lähdekoodin yhtymäkohtia alkuperäiseen.

4.3 Tyypinmäärittelyt dokumentaationa

Nyky aikaisten editorien vakio-ominaisuuksiin kuuluu kirjoittamisen tukeminen automaattisilla ehdotuksilla. Yksinkertaisimmillaan ehdotukset voivat perustua avatuissa tiedostoissa käytettyihin sanoihin, joita editori ehdottaa käytettäväksi uudelleen koodia kirjoitettaessa. Alkeellisetkin ehdotukset voivat nopeuttaa kirjoittamista silloin kun ne sattuvat osumaan oikeaan, mutta suurempi hyöty saadaan kun ehdotusten taustalla on syvempää koodin analyysia. Kun editori tai ehdotukset tarjoava editorin lisätyökalu ymmärtää koodissa olevia rakenteita ja niiden tyyppejä, ehdotukset ovat täsmällisempiä ja perustuvat kontekstiin johon uutta koodia ollaan juuri kirjoittamassa. Kuvassa 4.2 näkyy *VSCode*-editorin antamat ehdotukset eräille JavaScript- ja TypeScript-tiedostoille. TypeScriptiä kirjoittaessa editori ymmärtää että `tuote.`-ilmaisun jälkeen ainoat järkevät ehdotukset ovat `Ostos`-tyyppisen muuttujan ominaisuuksien nimiä, eli että ohjelmoija haluaa hyvin suurella todennäköisyydellä kirjoittaa joko `tuote.hinta` tai `tuote.nimi`. Ehdotuksen voi hyväksyä helposti enteriä painamalla, jolloin ehdotettua koodinpätkää ei tarvitse kirjoittaa käsin.



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

Eksplisiittisesti kirjoitetut tyypit sekä editorin antama tieto muuttujien tyypeistä voivat toimia myös aiemmin kirjoitetun koodin dokumentaationa ja kuvauksena siitä, miten moduulia on tarkoitus käyttää. Ehdotukset 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 yhtä hyvänä kuin staattisesti tyyppitetyissä kielissä. Huonoimmillaan ehdotetut muuttuja- ja metodinimet valitaan yksinkertaisesti listaamalla avoimista tiedostoista löytyviä nimiä, välittämättä sen enempää siitä ovatko nämä metodit määritetty juuri kyseiselle tyyppille. Edistyneemmät ehdotusmoottorit, kuten Visual Studiossa käytetty IntelliSense, suorittavat osaa JavaScript-koodista taustalla ja analysoivat siten muuttujien tyyppejä ajonaikana [20, 21]. 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 tyyppitetylle 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 tutkimus [22] testasi staattisen tyyppityksen ja automaattisten ehdotusten tehokkuutta antamalla osallistujille toteutettavaksi ohjelmointitehtävän JavaScriptillä ja TypeScriptillä, automaattisten ehdotusten kanssa ja niitä ilman. 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 kääntäjä, joka paljastaa virheet koodissa ennen kuin ohjelmoijan tarvitsee kokeilla koodin toimivuutta käytännössä.

Luku 5

Tyypitarkastamisen ongelmat

5.1 EcmaScript-yhteensopivuus

TypeScript pyrkii noudattamaan EcmaScript-spesifikaatiota mahdollisimman tarkasti kaikkien sellaisten ominaisuuksien suhteen jotka eivät nimenomaan liity staattiseen tyypittämiseen [23]. TypeScriptin kehityksen alkuvaiheissa, ennen EcmaScript 2015 -spesifikaation valmistumista, JavaScriptista kuitenkin puuttui joitain tärkeiksi katsottuja ominaisuuksia, jotka päätettiin lisätä TypeScriptiin mahdollisista yhteensopivuusongelmista huolimatta. TypeScriptissä on esimerkiksi syntaksi nimiavaruuden määrittämiseen, vaikka EcmaScriptiin myöhemmin lisätyt *moduulit* ajavat saman asian [24]. 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 tuleviin määrittelyihin. Esimerkiksi *null-turvallinen ketjutus* (engl. optional chaining tai safe navigation) [25] 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 Automaattinen ja vaiheittainen tyypittäminen

Jotta staattisen tyypityksen tuova työkalu olisi vartenotettava 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 JavaScriptiä 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 kehittyntä tyyppien automaattista päättelyä (engl. type inference).

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

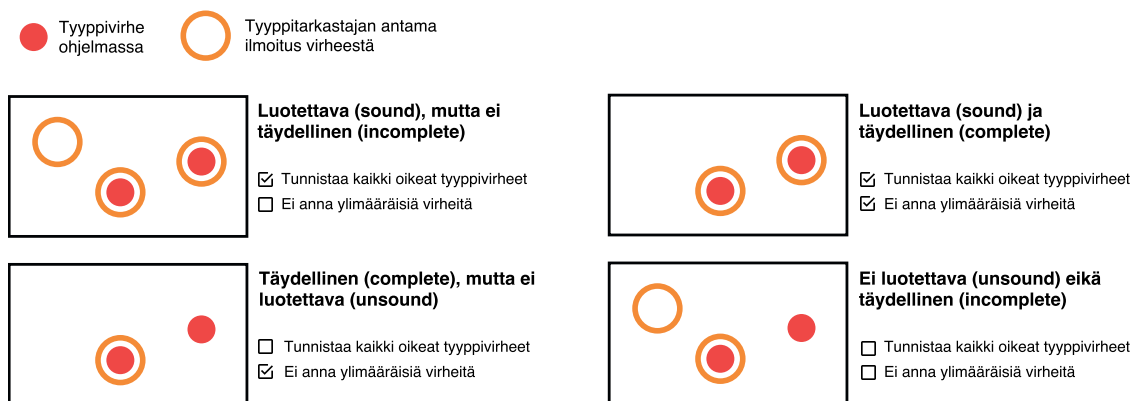
Listaus 5.1: Flow pystyy tulkitsemaan vertaaHintoja-funktion tyyppin automaattisesti, ilman eksplisiittisiä tyyppimäärittelyitä.

Esimerkissä 5.1, Flow pystyy päättämään muuttujan `vertaaHintoja` tyyppin

`(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 päättämään myös `sort`-metodille annetun `vertaaHintoja`-funktion tyyppin käyttämällä *kutsumispaikkaan perustuvaa päättelyä* (engl. call-site inference). TypeScript ei tue kut-

sumispaikkaan 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, minä vuoksi TypeScript vaatii nimettyjen funktioiden parametreille aina eksplisiittiset tyyppit. Kun funktion signatuuriin lisätään `(a: Ostos, b: Ostos)`, myös TypeScript pystyy automaattisesti päättelemään että funktion palautusarvo on numero, eikä eksplisiittistä `: number` palautusarvon määrittystä tarvita. Molempien työkalujen että tyyppien automaattinen päättely on viety paljon pidemmälle kuin se funktion sisäinen muuttujan tyyppin päättely, jollaista nähdään esimerkiksi `var`-avainsanalla C#:issa ja Javassa.

5.3 Luotettavuus, täydellisyys ja käytännöllisyys



Kuva 5.1: Tyypijärjestelmän luotettavuus ja täydellisyys

Tyypijärjestelmän luotettavuus (engl. soundness) kuvaa sitä, kuinka suuren osan mahdollisista ohjelmointivirheistä se estää. Täysin luotettava (engl. sound) tyypijärjestelmä estää kaikki sellaiset virheet jotka sen on tarkoitus estää [26]. Täydellisyys (engl. completeness) puolestaan kertoo salliiko tyypijärjestelmä kaikki kielen sellaiset ominaisuudet jotka eivät olisi ajonaikana tyypivirheitä [1, 26].

Jotta JavaScriptiä analysoiva tyypijärjestelmä olisi luotettava, sen on annettava virhe esimerkiksi seuraavasta Listauksen 5.2 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 tyyppijärjestelmä olisi täydellinen, sen on sallittava tämä korjattu versio ylläolevasta ohjelmasta:

```
1 function osta(ostos) {  
2   if (typeof ostos.nimi === 'string') {  
3     lisääTuote({  
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 5.2 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 tunnistamatta tai toimiva ohjelma voidaan merkitä virheelliseksi.

JavaScriptiä käännöskohteena käyttävät mutta muuten sen syntaksista ja semantiikasta eroavat uudet kielet, kuten Dart [27], Elm [28] ja ReasonML [29] 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.

TypeScriptin tyyppijärjestelmä on *rakennepohjainen* (engl. structural), koska sen katsottiin sopivan paremmin siihen tyyliin jolla JavaScriptiä tavallisesti käytetään. Rakennepohjaisessa tyyppijärjestelmässä keskitytään objektien muotoon eikä *nimelliseen* (engl. nominal) arvoon, minkä vuoksi myös 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 varoituksia kääntyvä TypeScript-ohjelma.

TypeScript-kääntäjä sallii esimerkin 5.4 koodin vaikka argumentit parametreille

`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 tyyhitetty 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.

JavaScriptissä on useita yleisesti käytettyjä rajapintoja jotka ovat hyödyllisiä mutta vaikeita tyyppittää staattisesti. Esimerkiksi suosittu kirjasto *lodash* löytyy funktio `get` [30], jota voidaan kutsua seuraavasti

```
1 import {get} from 'lodash';
2
3 const object = { a: [{ b: { c: 3 } }], d: null };
4 get(object, 'a[0].b.c'); // Palauttaa numeron 3
```

Listaus 5.5: get-funktiolla voidaan palauttaa arvo syvältä objektin sisältä välittämättä mahdollisesti puuttuvista arvoista.

Staattisen tyyppijärjestelmän on vaikea päätellä esimerkin get-kutsun palautusarvo tai tarkistaa toisen argumentin oikeellisuus. TypeScriptiä, Flow'ta tai Closurea käyttävä ohjelmoija joutuu joko luopumaan tyyppiturvallisuudesta tässä osassa koodia tai kirjoittamaan jonkin huomattavasti monimutkaisemman version säilyttääkseen tyyppiturvallisuuden.

Luku 6

Yhteenveto

Tyypijärjestelmä on tärkeä ohjelmointikielen ominaisuus, jolla on suuri vaikutus ohjelmistokehittäjän käyttökokemukseen. Eksplisiittiset tyypimäärittelyt vaativat lisää kirjoitettavaa koodia ja liian tiukka tyypijärjestelmä voi rajoittaa sitä mitä kielellä voi tehdä, sillä harva tyypijärjestelmä on täydellinen (engl. complete). Dynaamisesti tyypitetyn koodin nopeasta kirjoitustahdista saatavat hyödyt jäävät kuitenkin vähäisiksi jos lopputuotos ei toimi odotetulla tavalla. Bugit heikentävät käyttäjien tyytyväisyyttä ohjelmaan ja voivat pahimmillaan aiheuttaa pysyvää vahinkoa saadessaan ohjelman toimimaan virheellisesti. Staattinen tyypitys ohjaa kirjoitettua koodia turvalliseen suuntaan kehitysvaiheen alusta loppuun ja torjuu tietynlaisia ohjelmointivirheitä tehokkaammin kuin esimerkiksi ajonaikaista käyttäytymistä testaavat *unit testit*. Työn viidennen luvun viimeisessä esimerkissä esitelty get-kutsu on yksinkertainen ja helppolukuinen, mutta jos muuttujan `object` tyyppiä myöhemmin muutetaan toiseen muotoon muistamatta päivittää myös get-kutsua, virheellisestä get-kutsusta voi muodostua ohjelmaan hankala bugi.

Tässä tutkielmassa on esitelty kolme työkalua, jotka lisäävät dynaamisesti tyypitettyyn JavaScriptiin käännösaikaisen tyypitarkastuksen sekä syntaksin tyyppien eksplisiittiseen määrittämiseen. Flow, TypeScript ja Closure sallivat asteittaisen siirtymisen dynaamisesti tyypitarkastetusta staattisesti tarkastettuun, sekä staattisen tarkastuksen ohittamisen niissä osissa koodia joihin sitä olisi liian vaikeaa tai mahdotonta lisätä. Samankaltaisuus ja yhteensopivuus jo ennestään suosituksen JavaScriptin kanssa yhdistettynä tyypiturvallisuuteen, koodin kirjoittamista helpottaviin työkaluihin ja selkeämmin dokumen-

toituun koodiin ovat nostaneet erityisesti TypeScriptin käytön nopeaan kasvuun.

Lähdeluettelo

- [1] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [2] Transition to OO programming - Safety and strong typing. URL <http://www.cs.cornell.edu/courses/cs1130/2012sp/1130selfpaced/module1/module1part4/strongtyping.html>.
- [3] JavaScript language resources. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources.
- [4] Standard ECMA-262 - ECMAScript 2017 Language Specification. URL <https://www.ecma-international.org/publications/standards/Ecma-262.htm>.
- [5] TypeScript Language Specification. URL <https://github.com/Microsoft/TypeScript/blob/b8fbf884d0f01c1a20bb921cc0a65d6c1a517ee8/doc/TypeScript%20Language%20Specification.pdf>, versio 1.8.
- [6] Installing and setting up Flow for a project. URL <https://flow.org/en/docs/install/>.
- [7] Closure Compiler. URL <https://developers.google.com/closure/compiler/>.

-
- [8] Annotating JavaScript for the Closure Compiler. <https://github.com/google/closure-compiler/wiki/Annotating-JavaScript-for-the-Closure-Compiler>.
 - [9] Anders Hejlsberg. Microsoft Build 2014. TypeScript, maaliskuu 2014. URL <https://channel9.msdn.com/Events/Build/2014/3-576>.
 - [10] Shriam Rajagopalan Erik Wittern, Philippe Suter. A Look at the Dynamics of the JavaScript Package Ecosystem. 2016.
 - [11] DefinitelyTyped – repositorio TypeScript tyypitystiedostoille. URL <https://github.com/DefinitelyTyped/DefinitelyTyped>.
 - [12] flow-typed – repositorio Flow tyypitystiedostoille. URL <https://github.com/flow-typed/flow-typed>.
 - [13] Flowgen – kääntäjä TypeScriptist-tyypitystiedostoista Flow-tyypitystiedostoiksi, 2016. URL <https://github.com/joarwilk/flowgen>.
 - [14] Clutz – kääntäjä Closure-tyypitystiedostoista TypeScript-tyypitystiedostoiksi, 2016. URL <https://github.com/angular/clutz>.
 - [15] Tsickle – kääntäjä TypeScriptistä Closureksi, 2014. URL <https://github.com/angular/tsickle>.
 - [16] 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.
 - [17] Earl T. Barr Zheng Gao, Christian Bird. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. 2017.
 - [18] MDN: How to use a source map, 2018. URL https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map.

- [19] Closure Compiler - Source Map documentation, tammikuu 2019. URL <https://github.com/google/closure-compiler/wiki/Source-Maps>.
- [20] 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/>.
- [21] JavaScript Language Service in Visual Studio, tammikuu 2017. URL <https://github.com/Microsoft/TypeScript/wiki/JavaScript-Language-Service-in-Visual-Studio#unsupported-patterns>.
- [22] 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.
- [23] TypeScript Design Goals, syyskuu 2014. URL <https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals>.
- [24] Ryan Cavanaugh. Kommentti TypeScript-ominaisuusehdotuksessa Suggestion: "safe navigation operator", i.e. `x?.y`, lokakuu 2014. URL <https://github.com/Microsoft/TypeScript/issues/16#issuecomment-57645069>.
- [25] Gabriel Isenberg Claude Pache. Optional Chaining for JavaScript, 2018. URL <https://github.com/tc39/proposal-optional-chaining>.
- [26] CSE341: Programming Languages Winter 2013 Unit 6 Summary, 2013. URL <https://courses.cs.washington.edu/courses/cse341/13wi/unit6notes.pdf>.
- [27] Dart-kielen kotisivut, 2019. URL <https://www.dartlang.org/>.
- [28] Elm-kielen kotisivut, 2019. URL <https://www.dartlang.org/>.

-
- [29] ReasonML-kielen kotisivut, 2019. URL <https://reasonml.github.io/>.
- [30] Lodash-kirjaston dokumentaatio - get-funktio. URL <https://lodash.com/docs/4.17.11#get>.