JavaScriptin staattinen tyypittäminen

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

Sisältö

Luku 1

Johdanto

Kaikenlaisen ohjelmoinnin keskiössä on data ja yksi tärkeimmistä datan 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”. 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 tyyppimäärittelyitä. Ohjelman koodissa voitaisiin esimerkiksi määrittää että henkilön ikä on aina numero. Tällöin ohjelmointikielen kääntäjään rakennetut tarkistukset voivat vahtia 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 tyyppimäärittelyitä ei tarvitse kirjoittaa. Toisaalta staattinen tyypitys mah-

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

Luku 2

Peruskäsitteitä

2.0.1 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 määritelmästä eriävät tai virheelliset tyypit[1]. Vahvasti tyypitetyssä kielessä tällainen aiheuttaisi yleensä virheen, heikosti tyypitetyssä kielessä tyypeille voitaisiin tehdä implisiittisiä tyyppimuunnoksia tyyppien 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 ohjel-

man suoritus evaluoi kyseisen ilmaisun. Lisäksi esimerkiksi ilmaisu `"teksti" + 2` ei aiheuttaisi virhettä edes suoritusaikana, sillä heikoille tyyppijärjestelmille ominaisesti JavaScript muuttaisi numeron 2 string-muotoon ennen summausoperaation arviointia. Tässä tutkielmassa keskitytään lähinnä JavaScriptin tyyppien staattiseen ja dynaamiseen, eli käytännössä käännös- ja ajonaikaiseen tarkastamiseen. Jotkin 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.

Luku 3

Työkalun käyttöönotto

3.1 Tyyppiannotaatiot

Tyyppijärjestelmä voi päätellä muuttujan sallitun tyypin automaattisesti päättelemällä tai kieleen sisältyvien eksplisiittisten tyyppimäärittelyjen perusteella. Esimerkiksi Standard ML kykenee tulkitsemaan kaikkien muuttujien arvot automaattisesti, kun taas Java vaatii niiden olevan eksplisiittisesti annotoituja. Suurin osa staattisesti tyyppitarkastetuista kielistä kuitenkin putoaa jonnekkin välimaastoon. Haskell, C#, Crystal ja monet muut sisältävät syntaksin tyyppien eksplisiittiselle määrittämiselle mutta tarjoavat myös kielikohtaisesti vaihtelevan tuen tyyppien automaattiselle päättelylle. On hyvä huomioida että vaikka tyyppien tulkinta tapahtuu automaattisesti, niin se suoritetaan tässä tapauksessa silti käännös- eikä suoritusaikana. Esimerkiksi yllä mainittu Crystal on tarvittavien annotaatioiden vähyydestä huolimatta staattisesti tyyppitarkastettu, toisin kuin syntaksin inspiraation lähteenä ollut Ruby.

Kaikki kolme tässä esiteltyä JavaScriptin staattiseen tyyppitarkastukseen tarkoitettua työkalua päättelevät tyyppejä automaattisesti, mutta vaativat myös eksplisiittisiä määrityksiä paikoitellen. Closure-kääntäjä lukee tyyppimääritykset JSDoc-tyylisistä dokumentaatiokommenteista [2]. TypeScript ja Flow puolestaan jatkavat ECMA-262 -spesifikaatiota erityisellä syntaksilla tyyppien eksplisiittistä määrittelyä varten. Kumpikaan näistä lähes-

tymistavoista ei ole täysin ongelmaton. Dokumentaatiokommentit voivat olla hankala ja runsassanainen formaatti monimutkaisille tyyppiannotaatioille, mikä kasvattaa niiden kirjoittamiseen vaadittua työmäärää [3]. Toisaalta jo opitun JavaScriptin syntaksin jatkaminen uudella syntaksilla vaatii uusien merkintöjen oppimmista ja saattaa vaikeuttaa kirjoitetun koodin ymmärtämistä etenkin tottumattomille lukijoille.

```
1  /**
2  * @typedef {{
3  *   hinta: number
4  * }}
5  */
6  let Ostos;
7
8  /**
9  * @param {!Array<Ostos>} ostokset
10 * @return {number} Ostosten yhteenlaskettu hinta
11 */
12 function ostoskorinHinta(ostokset) {
13   let summa = 0;
14   for (const ostos of ostokset) summa += ostos.hinta;
15   return summa;
16 }
17
18 /** @type {number} */
19 const hinta = ostoskorinHinta([{ hinta: 5 }, { hinta: 10 }]);
20
21 //// Kommenteista poistaminen aiheuttaa virheen tarkastettaessa
22 // ostoskorinHinta([5, 10]);
```

Listaus 3.1: Esimerkki Closure-annotaatiosta funktiolle

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

function ostoskorinHinta(ostokset: Ostos[]): number {
  let summa = 0;
  for (const ostos of ostokset) summa += ostos.hinta;
  return summa;
}

const hinta: number = ostoskorinHinta([{ hinta: 5 }, { hinta: 10 }]);

//// Kommenteista poistaminen aiheuttaa virheen tarkastettaessa
// ostoskorinHinta([5, 10]);
```

Listaus 3.2: Esimerkki Flow tai TypeScript annotaatiosta funktiolle

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 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 koodin 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 työkalun käyttöönotossa tulee huomioida.

3.3 Työkalun vaiheittainen käyttöönotto

Kun kyseessä on jo olemassa olevan kielen, JavaScriptin, muuttaminen staattisesti tyyppitarkastetuksi, eräs merkittävä tekijä on että suuria määriä koodia voidaan jo olla kehitetty alkuperäisellä kielellä ilman tyyppitarkastuksista huolehtimista. Näin ollen tärkeäksi tekijäksi nousee vaiheittainen käyttöönotto. On tärkeää että olemassa olevaa annotoimatonta koodipohjaa voi yhä käyttää uuden, staattisesti tyyppitarkastetun koodin kanssa. Flow-tarkastukset lisätään olemassa olevaan JavaScript-koodiin erityisellä kommentilla. JavaScriptin muuttamisessa TypeScriptiksi riittää yleensä tiedostopäätteen muuttaminen. Closure toimii ilman erillistä muunnosta, kunhan tarvittava määrä dokumentaatiopohjaisia tyyppimäärittelyitä on annettu. Sekä Flow että TypeScript tarjoavat myös erityisen tyypin ”any” joka on kaikkien muiden tyyppien ylä- ja alatyyppi.

Luku 4

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. Esimerkeissä kommentoitu 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` [4]. 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.

Luku 5

Ohjelman optimointi käännösvaiheessa staattisen analyysin perusteella

Luku 6

Tyyppimäärittelyt dokumentaationa

Luku 7

Ongelmat JavaScriptin staattisessa tyypittämisessä

7.1 Vaikeasti analysoitavat suunnittelumallit

Luku 8

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

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

[4] 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.