Tuomo Hopia

Elm-ohjelmointikieli web-käyttöliittymien ohjelmoinnissa

Tietotekniikan kandidaatintutkielma

7. maaliskuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Tuomo Hopia

Yhteystiedot: tuomo.a.hopia@student.jyu.fi

Työn nimi: Elm-ohjelmointikieli web-käyttöliittymien ohjelmoinnissa

Title in English: Elm programming language in front-end web development

Työ: Kandidaatintutkielma

Sivumäärä: 31+0

Tiivistelmä: Elm-ohjelmointikieli on varsin uusi kielitulokas web-käyttöliittymien ohjelmointiin. Kieli pyrkii ratkaisemaan web-käyttöliittymien kehityksen merkittävimmät ongelmat hyvin omaperäisellä funktio-ohjelmointiin perustuvalla tavalla. Työssä tutustutaan web-käyttöliittymien ohjelmoinnin koettuihin yleisimpiin haasteisiin, jonka jälkeen tutkitaan niitä keinoja, joilla Elm ratkaisee kyseiset ongelmat. Lisäksi työssä tarjotaan katsahdus Elmin kehityksen nykytilaan ja tulevaisuuteen.

Avainsanat: funktio-ohjelmointi, funktionaalinen, web-kehitys, web-käyttöliittymien ohjelmointi, Elm, graafiset käyttöliittymät, gui

Abstract: Elm is a new functional programming language for web front-end development. Elm takes an attempt on solving the most significant problems of current web-development in its own opinionated way. This thesis introduces these problems and examines how Elm actually solves them. Furthermore, an overview of the current development focus of Elm is provided in the thesis.

Keywords: functional, programming, web-development, Elm, front-end development, graphical user interfaces, gui

Kuviot

Kuvio 1. Elm-ohjelmointikielen kuukausittaiset latausmäärät edeltävän kolmen vuoden ajalta (Npm Inc. 2018). 1
Kuvio 2. Elm-arkkitehtuurin toimintalogiikka kuvattuna........ 14
Kuvio 3. Ajonaikainen näkymä Elmissä........ 16

Sisältö

1 JOHDANTO 1

2 FUNKTIO-OHJELMOINTI 3

3 WEB-KÄYTTÖLIITTYMIEN OHJELMOINTI 5
3.1 Ongelmat web-kehityksessä 5

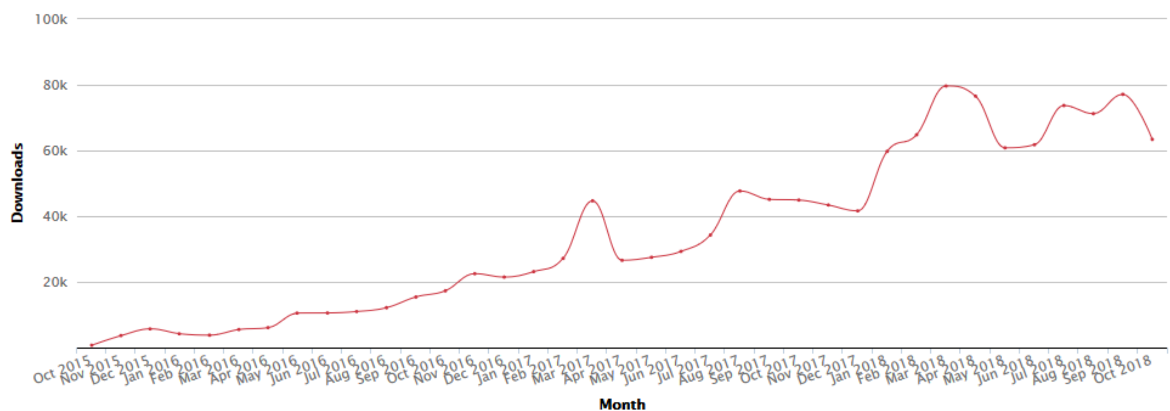
4 ELM-OHJELMOINTIKIELI 8
4.0.1 Funktionaalinen reaktiivinen ohjelmointi 9
4.1 Ominaisuudet 10
4.2 Tyyppijärjestelmä 10
4.3 Elm-arkkitehtuuri 12
4.3.1 Arkkitehtuurin toimintalogiikka 14
4.4 Kääntäjä 15
4.5 Ajonaikainen järjestelmä 15
4.5.1 Virtuaalinen dokumenttioliomalli 17
4.5.2 Yhteentoimivuus Javascriptin kanssa 17
4.6 Jatkokehitys ja haasteet 19

5 YHTEENVETO 21

KIRJALLISUUTTA 22

1 Johdanto

Web-käyttöliittymien ohjelmoinnista on tullut erittäin suosittua ohjelmistokehittäjien keskuudessa. Github-palvelun kymmenestä eniten ohjelmistokehittäjiltä tähtiä eli suosionosoituksista palvelussa saaneista kaikista avoimista ohjelmavarastoista viisi liittyy web-käyttöliittymien ohjelmointiin (Github 2018a). Vuonna 2018 Stack Overflow-tietotekniikkasivuston vuosittaiseen kyselyyn yli 100 000 vastanneesta ohjelmistokehittäjistä 38 % katsoo olevansa asiakasohjelmistokehittäjä (Stack Overflow 2018). Saman kyselyn mukaan myös web-käyttöliittymien perustan muodostavat Javascript-ohjelmointikieli sekä CSS- ja HTML-kuvauskielet ovat kärkikolmikossa ohjelmistokehittäjien eniten käyttämissä ohjelmointi- tai kuvauskielissä.



Kuvio 1. Elm-ohjelmointikielen kuukausittaiset latausmäärät edeltävän kolmen vuoden ajalta (Npm Inc. 2018).

Web-käyttöliittymien ohjelmoinnin noustessa näin suosituksi on syntynyt monia vaihtoehtoisia tapoja ohjelmoida web-käyttöliittymiä. Yksi näistä on Elm, joka on eräänlainen funktio-ohjelmoinnin sovellusaluekieli, eli ns. *Domain Specific Language* web-käyttöliittymien ohjelmointiin. Elm on tasaisesti kasvattanut suosiotaan ohjelmistokehittäjien keskuudessa kuvion 1 osoittamalla tavalla.

Itse funktio-ohjelmointilla, johon Elm-ohjelmointikieli perustuu, on hyvin pitkä historia ohjelmistokehityksessä. Ensimmäinen funktio-ohjelmointiin perustuva ohjelmointikieli on LISP, joka sai alkunsa jo vuonna 1958 ja on toiseksi vanhin edelleen

käytössä oleva ohjelmointikieli (Turner 2012, s. 5). Konrad Hinsen huomauttaa kuitenkin vuonna 2009 julkaisemassaan artikkelissa *The Promises of Functional Programming*, että vaikka funktio-ohjelmointi on ollut erittäin suosittua tietotekniikan tutkimuksessa jo monen vuosikymmenen ajan, sen käyttö tosielämän sovelluksissa on verrattain uutta (Hinsen 2009).

Karagkasidis tutki graafisten käyttöliittymien kehityksen haasteita vuonna 2008 (Karagkasidis 2008, s. 5-6). Hän näkee neljä suurta ongelmaa käyttöliittymien kehityksessä:

- graafisen käyttöliittymän luonti ja kokoaminen
- interaktio käyttäjän kanssa
- käyttäjän syötteen käsittely
- käyttöliittymän integrointi liiketoiminnallisen toiminnallisuuden kanssa

Kaikkien kokonaisvaltaisten web-käyttöliittymien ohjelmointiin tehtyjen ratkaisuiden on vastattava näihin haasteisiin. Tässä tutkielmassa tutkitaan sitä, kuinka Elm vastaa näistä kahteen ensimmäiseen. Käyttöliittymien ohjelmoinnissa paneudutaan Javascript-pohjaisen kehityksen haasteisiin ja peilataan Elmin ratkaisuja niitä vasten. Tutkimuskysymyksenä onkin, että *kuinka käyttöliittymien kehitys tapahtuu Elmillä ja kuinka Elm ratkaisee Javascript-pohjaisen käyttöliittymäkehityksen ongelmia*.

Tutkielmassa pohjustetaan aluksi kappaleessa 2 funktio-ohjelmointia teoreettisesta ja historiallisesta näkökulmasta, jonka jälkeen kappaleessa 3 perehdytään yleisesti moderniin web-käyttöliittymien ohjelmointiin sekä sen yleisimpiin ongelmiin. Itse Elm-ohjelmointikieleen keskittyvässä kappaleessa 4 tutkitaan Elmin perusteita, sen tapaa mallintaa web-käyttöliittymiä sekä Elmin tapaa ratkaista web-kehityksen yleisiä ongelmia. Sitten tarkastellaan Elmin vastaanottoa kehitysyhteisössä, sekä pohditaan lyhyesti Elmin kehityksen painopistettä tulevaisuudessa. Lopuksi yhteenvedossa kappaleessa 5 koostetaan tutkielmassa löydetyt Elmin ratkaisukeinot Javascript-pohjaisen web-käyttöliittymien ohjelmoinnin ongelmiin.

2 Funktio-ohjelmointi

Funktio-ohjelmoinnissa perusajatuksena on suorittaa funktio argumenteille (Hughes 1990, s. 1). Funktio-ohjelmoinnista puhuttaessa tulee usein esille käsite *puhtaasti funktionaalinen*. Tällä tarkoitetaan funktioita tai ohjelmointitapaa, jossa funktioiden sisällä ei voi suorittaa sivuvaikutuksia, eli esimerkiksi mutatoida dataa tai tilaa. Funktiolla on yksinkertaisesti parametri tai useita parametrejä sekä niitä vastaava paluuarvo. Puhtaassa funktio-ohjelmoinnissa jokaisella funktion argumentilla on yksi ja vain yksi lopputulos (Jones 1987). Matemaattisesti tätä voidaan ajatella siten, että jokaisella lähtöjoukon alkiolla on yksi ja vain yksi vastaava maalijoukon alkio. Kokonaisuudessaan funktio-ohjelmoinnissa ajattelumallina on, että ohjelma on lauseke, ja ohjelman suoritus lausekkeen evaluaatio (Harrison 1997; Jones 1987, s. 1; s. 10).

Pohjimmiltaan funktio-ohjelmointi perustuu yhdysvaltalaisen matemaatikon Alonzo Churchin julkaisemaan lambdakalkyylin — eräänlaiseen matemaattiseen abstraktioon (Hinsen 2009). Monet asiat funktio-ohjelmoinnissa voidaankin todistaa matemaattisesti pitäviksi, selkeyttäen ajattelumallia ja algoritmien toimintaa huomattavasti (Alic & Omanoic & Giedrimas 2016). Lambdakalkyyli perustuu kolmeen sievennykseen: alfa-, beta- ja gammasievennykyseen (Harrison 1997, s. 16). Näistä betasievennys on ohjelmoinnin näkökulmasta olennaisin, koska se kuvaa lambda-abstraktion sovellusta argumenteille (Jones 1987, s. 15). Lambdakalkyylissä Lambda-abstraktiot ovat määritelmiä, joilla merkitään uusia funktioita (Jones 1987, s. 12), mutta imperatiivisesta ohjelmoinnista tutun komentosekvenssin sijaan abstraktion sisältö on määritelmä.

Lambdanotaation mukaisesti perinteinen matemaattinen merkintätapa $f(x)$ muuttuu muotoon fx. Koska lambdanotaation mukaisesti funktioiden sovellus on vasemmalle assosioiva, saadaan notaatiosta $(f(x))(y)$ vuorostaan yksinkertaisesti fxy (Harrison 1997, s. 9). Näin voidaan antaa monta argumenttia ottaville funktioille yksi argumentti kerrallaan, ja muodostaa siten aina uusi funktio, joka ottaa yhden argumentin vähemmän. Elm, kuten Haskell-ohjelmointikielikin, nojaa vahvasti tähän

lambdakalkyylipohjaiseen vasemmanpuoleiseen assosiaatioon funktioita muodostaessa.

Lisäksi tietotekniikan kannalta laskennallisesti olennaista on myös, että sievennyksellä päästään aina yhteen samaan ratkaisuun, jos sellainen on olemassa. Church-Rosser-lause osoittaa tämän, eli että jokainen sievennysjärjestys lopulta johtaa samaan lopputulokseen (Harrison 1997, s. 19–20). Käytännössä laskennallisesti tämä tarkoittaa sitä, että laskentaa ei tarvitse lopettaa ja aloittaa alusta tietyssä pisteessä, vaan aina voidaan jatkaa uudella laskentastrategialla.

Noble ja Runciman ovat koonneet katselmuksessaan listauksen funktio-ohjelmoinnin soveltamisesta käyttöliittymien ohjelmointiin vuoteen 1994 saakka, osoittaen kuinka aikaisessa kehitysvaiheessa funktio-ohjelmoinnin implementaatiot käyttöliittymien ohjelmoinnissa olivat (Noble & Runciman 1994, s. 14, 16). Yksi aikaisimpia menestyksekkäitä käytännön ratkaisuja funktio-ohjelmoinnin soveltamisesta web-kehitykseen on Ruotsissa Ericsson-puhelinverkkoyhtiön kehittämä funktionaalinen Erlang-ohjelmointikieli ja -alusta (Armstrong 2003). Erlangin ja muutaman muun funktio-ohjelmoinnin teollisen menestystarinan johdosta 2000-luvun alusta lähtien web-kehityksessä onkin yritetty soveltaa funktio-ohjelmointia eri ohjelmointikielillä, lähinnä kuitenkin vain web-palvelinohjelmoinnissa.

2000-luvulla etenkin interaktiivisten yksisivuisten sovellusten (engl. *single-page app*) yleistyessä funktio-ohjelmointia on alettu soveltamaan myös käyttöliittymien ohjelmoinnissa. Esimerkiksi Anthony Courtney teki erilaisia funktionaaliseen reaktiiviseen ohjelmointiin perustuvia toteutuksia imperatiivisilla ohjelmointikielillä jo vuosina 2001–2003 (esim. Courtney & Elliott 2001; Courtney 2003). Elm onkin eräänlaistasta jatkumoa funktio-ohjelmoinnin ja toisaalta myös reaktiivisen funktionaalisen ohjelmoinnin soveltamista käyttöliittymäkehitykseen, nyt kuitenkin web-käyttöliittymäympäristössä.

3 Web-käyttöliittymien ohjelmointi

World Wide Web eli lyhyesti WWW on vuonna 1989 kehitetty järjestelmä, missä dokumentteja ja muita resursseja voidaan hakea internet-verkosta (The World Wide Web Consortium 2004). Verkkoselaimet hakevat WWW-järjestelmässä internetissä toimivilta verkkopalvelimilta HTML-dokumentteja eli verkkosivuja, jotka sitten piirretään käyttäjälle näkymään verkkoselaimessa. Dokumenttioliomalli on kieliriippumaton rajapinta, jonka tarkoitus on mallintaa sama verkkosivu olioista koostuvaan puumaiseen muotoon (The World Wide Web Consortium 2005a). Web-käyttöliittymien ohjelmoinnissa Javascriptilla manipuloidaan tätä dokumenttioliomallia käyttäen verkkoselainten tarjoamaa rajapintaa, jolloin muutokset näkyvät selaimessa käyttäjälle (Mozilla 2018c).

Javascript on kehitetty alun perin vuonna 1995 Netscape Navigator -verkkoselainta varten (Flanagan 2011, s. 2). Nykyään kaikki modernit verkkoselaimet sisältävät Javascript-tulkin, joten Javascript on verkkoselainympäristössä vakiintunut ohjelmointikieli. Juuri tämä tiukka assosiaatio verkkoselainten kanssa onkin johtanut Javascriptin suosioon yhdeksi maailman käytetyimmistä ohjelmointikielistä (Crockford 2008, s. 2) sekä tehnyt moderneista verkkosovelluksista ylipäätään mahdollista toteuttaa (Haverbeke 2018, s. 6). Javascript on tulkittu, heikosti ja dynaamisesti tyypitetty prototyyppipohjainen olio-ohjelmointikieli (Flanagan 2011, s. 1). Sillä on mahdollista kirjoittaa monen ajattelumallin mukaista koodia, sillä se ei aseta juurikaan rajoituksia esimerkiksi muuttumattoman datan, sivuvaikutusten tai tyypityksen suhteen.

3.1 Ongelmat web-kehityksessä

Pääosin Javascriptin heikon tyypityksen ja sitä paikkaamaan rakennetun implisiittisen tyyppimuunnosten seurauksena ohjelmistokehittäjän on oltava tarkkaavainen Javascript-ohjelmakoodia kirjoittaessa. Esimerkiksi allaoleva koodi on täysin valideja ilman virheitä ajettavia Javascript-määritelmiä paluuarvoineen.

```
NaN==NaN
// false
true+true===2
// true
{}+[]
// 0
```

Ylläolevissa määritelmissä törmätään myös epäintuitiivisiin oikeisiin ja vääriin arvoihin, eli ns. *truthy*-arvoihin (Crockford 2008, s. 106). Jos Javascriptiä ei ymmärrä tarpeeksi hyvin, tämänkaltaiset tilanteet vaikuttavat epäloogisilta ja vaikeasti ennustettavilta. Koska ylläolevankin kaltainen koodi suoritetaan ilman vikailmoituksia, ohjelmoijalle itselleen jää suuri vastuu ylläpitää korkeaa koodin laatua sekä pitää huoli siitä, että koodi suoritetaan kuten ohjelmoija olettaa.

Yleisesti Javascript nähdään varsin vikaherkkänä ohjelmointikielenä. Esimerkiksi vuoden 2018 *State of Javascript*-kyselyssä ES6-standardin mukaisen Javascriptin luotaantyöntävimmät ominaisuudet olivat järjestyksessä bugisuus ja vikaherkkyys, kömpelö ohjelmointityyli sekä pöhöttyneisyys ja monimutkaisuus kielenä. Kyselyn mukaan yli puolet Javascript-ohjelmointiin pettyneistä oli tätä mieltä. Vuotta aiemmin samankaltaisen kyselyn tuloksena suuri osa ohjelmistokehittäjä oli myös sitä mieltä, että Javascript on mennyt liian monimutkaiseksi sekä myös, että Javascriptin ekosysteemi etenee liian kovalla vauhdilla (State of Javascript 2017).

Javascriptin yleisimmät ongelmat ovat jo siis varsin maineikkaita ohjelmistokehittäjien keskuudessa. Esimerkiksi Quora-tietopalvelussa kysymykseen *miksi Javascriptiä vihataan* on poikkeuksellisen monta vastausta ja runsaasti keskustelua asiasta (Quora Inc. 2018). Javascript-ohjelmointikielen yksi aktiivisimmista kehittäjistä, Douglas Crockford 2008, kirjoitti kritiikille vastapainoksi teoksen *Javascript: The Good Parts*, jossa hän pui sitä, kuinka Javascript on maailman väärinymmärretyin ohjelmointikieli. Crockfordin mukaan Javascriptiä syytetään perusteettomasti selainten tarjoamasta rajapinnasta selainten dokumenttioliomallin manipuoloimiseksi. Kuitenkin myös Crockford joutuu teoksessaan myöntämään että monet Javascriptin ominaisuuksista ovat haitallisia hyvien ohjelmistokehityskäytänteiden ylläpidon näkökul-

masta. Etenkin vapaasti manipuoloitavat ja ajon aikana kaikkialla näkyvissä olevat globaalit tietorakenteet ovat hänen mielestään suurimpia ongelmia (Crockford 2008, s. 101). Crockfordin mukaan vapaasti manipuloitavat globaalit tietorakenteet erityisen painava ongelma siksi, että ohjelmistokehitys Javascriptillä pitkälti perustuu niihin.

Moderneissa Javascript-sovelluksissa on tyypillistä hallita sovelluksen reaaliaikaista tilaa systemaattisesti, usein jonkun tietyn kirjaston tai ajattelumallin avulla. Tilan hallintakirjastojen suosion perusteella GitHub-palvelussa (Github 2018a) tilan hallinta on selkeästi merkittävä ongelma modernissa web-kehityksessä, eikä selainympäristö tai Javascript ohjelmointikielenä tarjoa siihen minkäänlaista valmista ratkaisua. Vuoden 2017 yli 20 000 vastausta keränneessä kyselyssä Javascript-ohjelmistokehittäjät ovat nykyisiin ratkaisuihin tyytyväisiä arvosanalla 3.5 kun arvoasteikko on välillä 1–5, jossa 5 tyytyväisin (State of Javascript 2017).

Koska web-käyttöliittymät muuttuvat nykyään yhä interaktiivisemmiksi ja siten monimutkaisemmiksi, on asynkroninen eli samanaikaisuusajoon keskittyvä ohjelmointi yhä tärkeämpää. Siinä missä synkronisessa suorituksessa tapahtuu vain yksi asia kerrallaan, asynkronisessa niitä tapahtuu monta yhtä aikaa. Javascriptissä samanaikaisuusajon malli pohjautuu tapahtumasilmukkaan (Mozilla 2018a; Haverbeke 2018, s. 180). Javascriptissä asynkronista ohjelmakoodia voidaan kirjoittaa perustuen ns. callback-funktioihin tai Javascriptin promise-olioihin (Haverbeke 2018, s. 185). Promise-oliot ovat olioita, jotka esittävät operaation tilan sekä lopullisen onnistumisen tai epäonnistumisen. Kuitenkin promise-olioissakin lopulta nojataan callback-funktioihin, jotka suoritetaan operaation onnistumisen tai epäonnistumisen yhteydessä. Asynkronisen koodin kirjoittaminen heikentää koodin luettavuutta, koska asynkronista koodia ei suoriteta synkronisessa ja siten helposti ymmärrettävässä järjestyksessä. Tämä vaikeuttaa lopulta myös koodin testaamista. ECMAScript 2017 -standardissa on uutena keinona asynkroniseen ohjelmointiin määritetty ns. `async/await`-funktiot (ECMA International 2017, ks. spesifikaatio *ECMA-262*). Vaikka nämä ovat teknisesti syntaktisesti yksinkertaistettu rajapinta promise-olioiden käytölle Mozilla (2018b), pystyvät ne parantamaan koodin luettavuutta.

4 Elm-ohjelmointikieli

Korjausehdotuksia Javascript-pohjaiseen web-kehitykseen on käytännössä kahdenlaisia — toiset keskittyvät Javascriptin ongelmien rajaamiseen ja paranteluun, siinä missä muut ratkaisuehdotukset pohjautuvat Javascriptin välttelylle. Elm on esimerkki jälkimmäisestä, eli eräänlainen Javascriptin välttelystrategia, jossa ei pyritä parantamaan Javascriptin ominaisuuksia, vaan käytetään täysin uutta, puhtaasti web-käyttöliittymien ongelmien ratkaisemiseksi suunniteltua ohjelmointikieltä ja -alustaa.

Elm-ohjelmointikielen kehitti Yhdysvaltalainen Evan Czaplicki, joka edelleen aktiivisesti jatkokehittää kieltä. Elm syntyi alun perin hänen opinnäytetyönsä tuloksena vuonna 2012 Harvardin yliopistossa (Czaplicki 2012).

Czaplicki oli turhautunut senaikaiseen imperatiiviseen web-käyttöliittymien ohjelmointiin. Hän halusi löytää paremman keinon mallintaa web-käyttöliittymiä intuitiivisemmin sekä kehittäjän kannalta tuottavammin keinoin. Czaplicki oli aiemmin saanut vaikutteita funktionaalisesta reaktiivisesta ohjelmoinnista (engl. *functional reactive programming*), mutta halusi löytää parempia keinoja paikata niiden suorituskykyongelmia (Czaplicki 2012, s. 6). Alun perin suunnitellessa kielen ensimmäistä versiota Czaplicki näki kahden asian olevan Elm-kielen perusta: puhtaasti funktionaalinen graafinen käyttöliittymä, sekä täysi tuki samanaikaisesti (engl. *concurrent*) ajettavalle funktionaaliselle reaktiiviselle ohjelmoinnille (Czaplicki 2012, s. 1).

Elmistä puhuttaessa yleisesti ei kuitenkaan tarkoiteta pelkkää itse ohjelmointikieltä, vaan koko Elmin alustaa, joka mahdollistaa web-kehityksen kyseisellä kielellä. Elmin alusta kokonaisuudessaan koostuu neljästä eri komponentista:

- Elm-ohjelmointikieli
- Elm-arkkitehtuuri
- Elm-kääntäjä
- Elmin ajonaikainen järjestelmä

Koko alusta yhdessä mahdollistaa tämän sovellusaluekielen käytön web-käyttöliittymien ohjelmointiin (Poudel 2018). Kokonaisuudessaan Elm on avointa lähdekoodia, ja sitä kehitetään aktiivisesti GitHub-palvelussa (Github 2018b). Ohjelmointikielenä Elm on kuitenkin suunniteltu ratkaisemaan pelkästään web-maailman ongelmia. Tämä käy ilmi esimerkiksi Elm-projektin GitHub-profiilista, jossa on kääntäjä, kirjastot ja työkalut pelkästään web-ympäristöä varten (Github 2018b). Lisäksi Elmille ei projektin ylläpidon mukaan tulevaisuudessakaan ole suunnitteilla laajentumista web-ympäristön ulkopuolelle (Github 2018c, ks. *roadmap.md*, viimeinen kysymys).

4.0.1 Funktionaalinen reaktiivinen ohjelmointi

Alun perin funktionaalisen reaktiivisen ohjelmoinnin (tästedes FRP) konseptin esitteli Paul Hudak ja Conal Elliott vuonna 1997 julkaistussa julkaisussa *Functional Reactive Animation*. FRP:n perimmäisenä ajatuksena on mallintaa asynkronisia datavirtoja funktio-ohjelmoinnin elementein explisiittisesti mallintaen datan ajan mukaan. FRP:ssä kantavana ajatuksena on, että järjestelmää muutetaan tapahtumiin perustuen (Courtney & Elliott 2001). Kuten Czaplickin alkuperäisen Elmin projektin käynnistäneen tutkielman nimestäkin — *Elm: Concurrent FRP for Functional GUIs* — käy ilmi, oli FRP alun perin toinen Elmin tärkeimmistä ominaisuuksista (Czaplicki 2012, s. 2).

Reaktiivisella ohjelmoinnilla on kuitenkin jo pohjaa aiemmin toteutetuissa synkronisissa datavirtoihin perustuvissa ohjelmointikielissä, kuten esimerkiksi Esterel ja Lustre, ja lopulta niiden pohjalta syntynyt Lucid Synchrone-ohjelmointikieli. Nämä ohjelmointikielet perustuvat matemaattiseen samanaikaisajon malliin, ja ne on tarkoitettu reaaliaikajärjestelmien rakentamiseksi (Caspi et al. 2012, s. 2). Elmillä onkin niin paljon yhteistä synkronisten ohjelmointikielten kanssa, että Czaplicki on vuosien kehitystyön jälkeen pohtinut Elmin olevan lähempänä niitä kuin FRP:tä (Elmlang 2018b).

4.1 Ominaisuudet

Elm on ohjelmointikielenä puhtaan funktionaalinen ja kaikki data on muuttumatonta (Elm-lang 2018d). Funktionaalinen puhtaus saadaan toteutumaan siten, että käännettäessä sivuvaikutukset varsinaisesti suorittava koodi tulee Elmin alustalla Elmin ajonaikaisesta järjestelmästä (engl. *Elm runtime*). Tämä pitää itse ohjelmistokehittäjän kirjoittaman koodin funktionaalisesti puhtaana. Puhtaat funktiot muuttumattoman datan kanssa saavat aikaan viitteellisen läpinäkyvyyden (engl. *referential transparency*). Puhtaiden funktioiden ja muuttumattoman datan ansiosta Elm tekee Javascriptin kaltaisen globaalit tietorakenteet ja niiden sivuvaikutuksellisen manipuloinnin mahdottomaksi. Tällä tavoin Elm asettaa rajoitteita ohjelmistokehittäjälle, jotka samalla pakottavat kirjoittamaan selkeämpää ja helpompilukuisempaa ohjelmakoodia.

Ohjelmointikielenä Elm on kohtuullisesti Haskell-ohjelmointikielen kaltainen, sekä syntaksinsa että ominaisuuksiensa puolesta. Elm-Javascript-kääntäjä on myös kirjoitettu Haskell-kielellä (Github 2018d). Vaikka vaikutteita Haskellista on otettu runsaasti, on monet Haskellin ominaisuuksista jätetty kokonaan soveltamatta Elmiin pitääkseen kieli yksinkertaisena ja kompaktina. Esimerkiksi Haskellista poiketen, Elm ei tue omien operaattoreiden luomista (Elm-lang 2018a).

4.2 Tyyppijärjestelmä

Elm on staattisesti ja vahvasti tyypitetty ohjelmointikieli. Staattisesti tyypitetyssä ohjelmointikielessä tyyppitarkastus tehdään ennen ajoa siinä missä dynaamisesti tyypitetyssä tarkastus tehdään ajon aikana (Gabbrielli & Martini 2010, s. 202). Vahvasti tyypitetylle taas Liskov ja Zilles antoivat jo vuonna 1974 määritelmän *kun objekti annetaan kutsuvasta funktiosta kutsuttuun funktioon, sen tyypin tulee olla yhteensopiva kutsutussa funktiossa määritellyssä tyypissä*. Kolme vuotta myöhemmin Jackson laajensi määritelmän muotoon *Vahvasti tyypitetyssä kielessä kaikella datalla pitää olla selkeä tyyppinsä, ja jokaisen prosessin pitää määritellä kommunikointikeinonsa näiden tyyppien avulla* (Jackson 1977). Vahvan tyypityksen takia Elm on tyyppiturvallinen,

eikä ajonaikaisia tyyppivirheitä siksi voi esiintyä (Gabbrielli & Martini 2010, s. 201). Staattisen tyypityksen johdosta taas tyyppitarkastukset pystytään tekemään ennen ajoa ja jopa ennen ohjelmakoodin kääntämistä, antaen ohjelmoijalle palautetta epäsopivista tyypeistä ohjelmakoodissa.

Elm jakaa myös Haskellin kanssa yhden hiukan epätavallisemmista ominaisuuksistaan, eli tyyppien tunnistamisen (engl. *type inference*). Haskellin kääntäjien tapaan Elm osaa tulkita käyttäjän haluamat tyypit, kuitenkin aina mahdollisimman yleisellä tasolla ottaen huomioon kaikki mahdolliset tyypit tilanteessa. Haskellista poiketen, Elm ei kuitenkaan tue tyyppiluokkia (engl. *typeclass*), joten monet yleisemmät abstraktiot on mahdotonta kirjoittaa Elmillä. Tämä yleensä tarkoittaa kasvavaa koodin määrää, koska samoja tai samankaltaisia asioita joudutaan toistamaan koodissa.

Elmissä, kuten Haskellissakin, on vahvasti suositeltavaa ilmoittaa tyypit eksplisiittisesti ilmaisun yläpuolella olevalla rivillä. Elmin syntaksi näiden tyyppiannotaatioiden osalta poikkeaa Haskellista marginaalisesti. Siinä missä esimerkiksi funktion `map` tyyppiannotaatio Haskellissa on

```
map :: (a -> b) -> [a] -> [b]
```

on vastaava Elmin funktio tyyppiannotaatioltaan

```
map : (a -> b) -> List a -> List b
```

Vaikka taulukoissa käytetään Elmissä tyyppiannotaationa `List`-avainsanaa, on itse ohjelmakoodissa käytössä Haskellia vastaava syntaksi `[a]` taulukoiden osalta. Elmissä voidaan myös rakentaa omia datatyyppejä `type`-avainsanan avulla (Elmlang 2018e). Omien datatyyppien määrittelyn avulla on mahdollista paremmin mallintaa ohjelmaa tyyppijärjestelmää käyttäen ja vahvan tyyppijärjestelmän avulla saavuttaa pienempi vikaherkkyys ohjelmaan tekemällä ei-toivotuista tiloista tyyppiepäyhteensopivia.

4.3 Elm-arkkitehtuuri

Elmin alustan sisältämä Elm-arkkitehtuuri on järjestelmä, jonka varaan sovelluksia rakennetaan Elmissä. Elm-arkkitehtuuri muodostuu kolmesta komponentista: malli, päivitys ja näkymä. Nämä ovat sisäänrakennettuna Elmin alustaan ja yhdessä ne muodostavat ajattelumallin web-käyttöliittymien mallintamiseen Elmillä. Malli itsessään on pelkkää dataa, siinä missä näkymä ja päivitys ovat puhtaita funktioita.

Elm-arkkitehtuurin malli ylläpitää koko sovelluksen tilaa tietyllä ajanhetkellä. Tämä tila on pelkkää Elmin ymmärtämää dataa, eikä sitä näytetä käyttäjälle tässä muodossa. Elmin mallli mallinnetaan yleensä Elmin kattavalla tyyppijärjestelmällä.

Päivitys tarkoittaa Elm-arkkitehtuurissa funktiota, jolle annetaan aina ns. viesti, eli tieto siitä, mitä halutaan muuttaa ja miten, sekä malli, eli tällä hetkellä voimassaoleva tila.

```
update : msg -> Model -> Model
update msg model = ...
```

Kuten funktion ylläolevasta tyyppiannotaatiostakin selviää, päivitysfunktio laskee annetun viestin ja nykyisen tilan perusteella uuden tilan, ja palauttaa sen. Funktio itsessään on puhdas, koska se saa kaiken tarvitsemansa tiedon annetuista argumenteista. Elm-arkkitehtuurissa funktion paluuarvon tuloksena syntyy automaattisesti uusi sovelluksen tila. Tämä muistuttaa hiukan George Mealyn 1955 kehittämää äärellistä automaattia, jossa paluuarvo määräytyy sekä syötteen, että nykyisen tilan perusteella (Mealy 1955).

Elm tarjoaa puhtailla tilan päivitysfunktioillaan modernista Javascript-pohjaisesta kehityksestä tutun tilan päivitysongelman ratkaisun. Tyypillisesti Javascriptissä pidettäisiin yllä tilaa joko sivuvaikutuksellisesti manipuloimalla globaalia tilaobjektia, tai haasteellisimmillaan tila olisi sidottu suoraan käyttöliittymänäkymään, eli siis dokumenttioliomalliin. Kappaleessa ongelmat mainittu Javascript-pohjainen tilan hallintakirjasto Redux onkin mallinnettu juuri Elm-arkkitehtuuriin pohjautuen (Three Devs and a Maybe 2015). Koska Redux-kirjastolla on eniten suosiota osoitta-

via tähtiä GitHub-palvelussa (Github 2018a) kaikista tilan hallintaohjelmavarastoista, sekä myös kyselyissä pitää kehittäjien tyytyväisyydessä tilan hallintaratkaisuista kärkisijaa (State of Javascript 2017) voidaan Elmin ratkaisun tilan hallintaan luonnehtia olevan kehittäjille mieluisa ainakin konseptitasolla.

Koska tilan päivitys tapahtuu puhtaiden funktioiden kautta, pystytään Elmissä tarjoamaan mielenkiintoisia ominaisuuksia tutkia ja testata Elm-pohjaisia sovelluksia. Esimerkiksi Elmin kehitystyökaluun on integroitu eräänlainen tilan säilytystyökalu, joka tallentaa kaikki tilamuutokset reaktiivisen ohjelmoinnin periaattein, eli ajan mukaisesti jonoon. Aikaisempien tilojen tutkiminen tällä tavoin on mahdollista siksi, koska päivitysfunktiot ottavat toisena argumenttina aina sillä hetkellä voimassa olevan mallin. Osoituksena ominaisuuden pragmaattisesta hyödystä sovelluskehittäjille tämä tilan historiallinen jäljitys viedään vielä pidemmälle tuoreessa teoksessa *Trends in Functional Programming* (Horemans et al. 2018, s. 76–97). Teoksessa pohjustetaan eräänlaisen monikerroksinen tilahistorian tutkimistyökalun konsepti, jossa tilaa on mahdollista tutkia laajemminkin koko sovelluksessa Elmin ulkopuolella, esimerkiksi yhdessä palvelinohjelmiston kanssa. Tämä kuvastaa tilaongelman yleismaailmallisuutta ja voi tulevaisuudessa helpottaa merkittävästi esimerkiksi integraatiotestausta, etenkin tilallisissa reaaliaikasovelluksissa.

Näkymä Elmissä tarkoittaa Elmin ajonaikaista järjestelmää varten tuotettua kuvausta tietyllä ajan hetkellä käyttäjälle näytettävästä verkkosivusta. Elm-arkkitehtuurin mukaisesti näkymäfunktiolle annetaan argumenttina ohjelman tila (Poudel 2018). Tilan perusteella mallinnetaan itse näkymä. Käytännössä ohjelmoijalle tämä tarkoittaa Elmin `HTML`-kirjastoa käyttäen tarkoitetun HTML-näkymän mallintamista argumenttina annetun tilan perusteella.

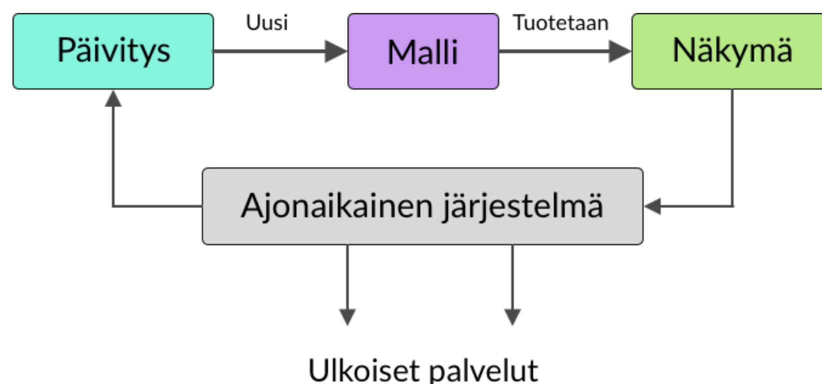
```
view : Model -> Html Msg
view model = ...
```

Näkymän kirjoittaminen ohjelmakoodiin muodossa, mitä käyttäjälle halutaan näyttää eri tilan eli mallin datan mukaan on deklaratiivista, koska dokumenttioliomallin manipulointia ei kirjoiteta eksplisiittisinä määritelminä ohjelmakoodiin imperatiivi-

sen ohjelmoinnin tapaan. Näkymäfunktion tuloksena syntyvän kuvauksen muuntaminen dokumenttioliomallin kautta käyttäjälle näkyväksi näkymäksi toteutetaan Elmissä kappaleessa 4.5.1 järjestelmän avulla.

4.3.1 Arkkitehtuurin toimintalogiikka

Ohjelmoija ei itse rakenna arkkitehtuurin mallin, päivityksen ja näkymän välistä toiminnallista logiikkaa, vaan tämän tarjoaa suoraan Elmin alusta (Poudel 2018). Ohjelmoijan tulee ainoastaan ohjelmakoodissa kuvata komponentit oikeassa Elmin kääntäjän ymmärtämässä muodossa.



Kuvio 2. Elm-arkkitehtuurin toimintalogiikka kuvattuna.

Kuvio 2 osoittaa arkkitehtuurin toiminnan käytännössä. Näkymästä lähetetään ennalta koodissa kuvattu pyyntö Elmin ajonaikaiseen järjestelmään, esimerkiksi kun nappulaa klikataan verkkosivulla. Tämän seurauksena ajonaikainen järjestelmä saattaa tehdä jonkun sivuvaikutuksen, esimerkiksi datan hakemisen verkkopalvelimelta. Lopulta ajonaikainen järjestelmä välittää tiedon viestillä päivitysfunktiolle, joka ottaa nykyisen mallin ja päivitysviestin, ja niiden perusteella lopulta tuottaa uuden mallin. Tämä näkymäfunktio saa samantien tuloksena syntyneen uuden mallin argumenttina ja tuottaa kuvauksen näkymästä Elmin ajonaikaiselle järjestelmälle, jonka vastuulla on dokumenttioliomallin manipulointi oikeanlaiseksi.

4.4 Kääntäjä

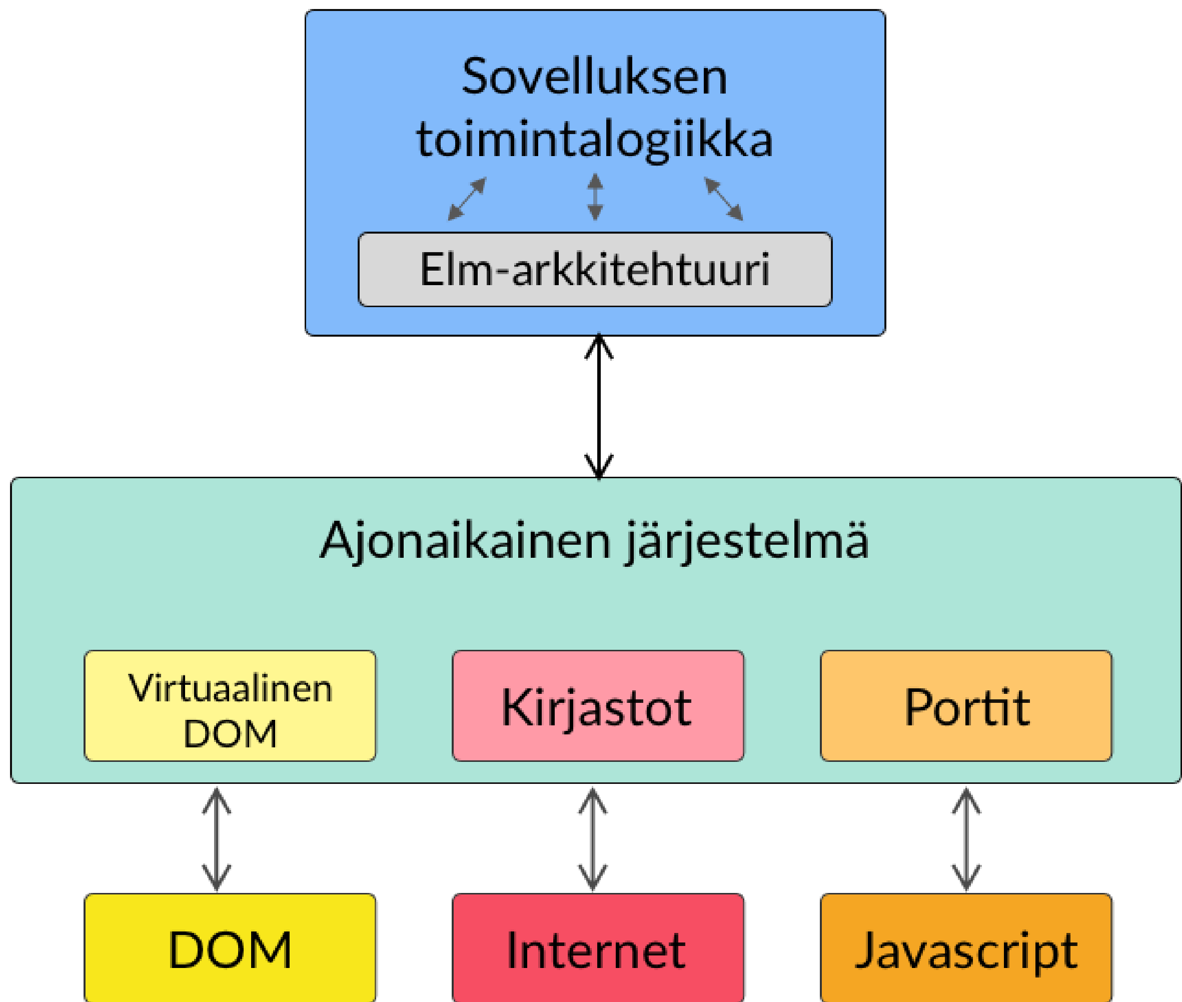
Toisin kuin tulkitut kielet kuten esimerkiksi Javascript, Elm on kääntäjän kääntämä ohjelmointikieli. Tämä johtuu siitä, että selaimet ymmärtävät käytännössä pelkästään Javascript-ohjelmointikieltä, ja siksi kaikki on lopulta käännettävä Javascript-koodiksi. Myös muut suositut web-käyttöliittymien ohjelmointiin suunnitellut kielet toimivat samoin, kuten esimerkiksi TypeScript (Microsoft 2018).

Yleisistä ohjelmakoodin tulkkien ja kääntäjien epäselvistä vikailmoituksista turhautuneena Evan Czaplicki suunnitteli Elmin kääntäjän alun perinkin ei pelkästään varsinaista ohjelmakoodin kääntämistä varten, vaan myös tehokkaaksi työkaluksi ohjelmistokehittäjän opastuksessa paremman ja virheettömämmän ohjelmakoodin tuottamiseksi (Elm-lang 2018f). Käytännössä tämä näkyy kehittäjäyhteisönkin mielestä avuliaina varoitus– ja vikailmoituksina ohjelmakoodia käännettäessä (Hicks 2017, ks. yhdeksäs kuvaaja ylhäältä), usein konkreettisten parannusehdotuksien saattelemana.

4.5 Ajonaikainen järjestelmä

Ohjelmakoodia käännettäessä Javascriptiksi Elm ei pelkästään käännä ja pakkaa ohjelmistokehittäjän kirjoittamaa Elm-koodia Javascriptiksi, vaan sen lisäksi pakkaa myös käytetyt kirjastot ja ennen kaikkea Elmin oman ajonaikaisen järjestelmän (engl. *Elm runtime*) (Poudel 2018). Elmin ajonaikainen järjestelmä sisältää valtavan määrän logiikkaa, joka toteuttaa muunmuassa sivuvaikutukset, eli esimerkiksi datan hakemisen ulkoisista lähteistä. Lisäksi järjestelmä sisältää virtuaalisen dokumenttioliomallialgoritmin, sekä Javascriptin kanssa yhteistoimintaan tehdyn viestijärjestelmän. Elmissä ohjelmistokehittäjän oma sovellus kommunikoi Elmin ajonaikaisen järjestelmän kanssa kuvion 3 mukaisesti.

Kun pelkästään Elmin oma ajonaikainen järjestelmä on vastuussa sivuvaikutusten suorittamisesta, saadaan itse kehittäjän tuottama Elm-lähdekoodi pidettyä sivuvaikutuksista täysin puhtaana. Ratkaisu on tehty puhtaasti web-kehitysympäristöön ja on siten varsin erilainen esimerkiksi verrattuna Haskellissa käytettyyn tapaan



Kuvio 3. Ajonaikainen näkymä Elmissä.

kuvata sivuvaikutukset monadien avulla (Lipovaca 2011, ks. kappale *A Fistful of Monads*). Kummatkin ratkaisut osaltaan eristävät sivuvaikutukset selkeästi, mutta Elm vie eristyksen täysin kehittäjän ulottumattomiin. Kun sivuvaikutukset saadaan eristettyä pois ohjelmistokehittäjän harteilta, Elmin oman lähdekoodin kehittäjät ja ylläpitävät voivat keskittyä työstämään Elmin omasta ajonaikaisesta järjestelmästä mahdollisimman turvallisen ja suorituskykyisen kaikille Elmiä käyttäville ohjelmistokehittäjille.

4.5.1 Virtuaalinen dokumenttioliomalli

Vaikka Javascript-tulkkien jatkuvan kehityksen ansiosta Javascript itsessään suoritetaan varsin nopeasti ja suorituskykyisesti, on dokumenttioliomallin manipulointi kuitenkin hidasta sekä myös tietokoneen laskentaresurssien puolesta kallista (Simon 2018).

Mitä interaktiivisempia verkkosovellukset ovat, sitä enemmän dokumenttioliopuuta joudutaan manipuloimaan ajon aikana. Dokumenttioliopuun muuttuessa aloitetaan uudelleenlaskentaprosessi nimeltä reflow, jossa elementtien sijainnit ja muodot lasketaan uudelleen (Simon 2018). Prosessin lopputuloksena uusi dokumenttioliopuu piirretään näkymään käyttäjälle. Reflow on siksi hyvin hidas prosessi, koska yhden elementin lisääminen, poistaminen tai muuttaminen saattaa käynnistää useita reflow-operaatioita, joissa saatetaan jopa käydä koko kyseinen dokumenttioliopuu läpi.

Turhan dokumenttioliomallin manipuloinnin välttämiseksi Elmin ajonaikaiseen järjestelmään on sisäänrakennettu eräänlainen virtuaalinen dokumenttioliomalli. Virtuaalinen dokumenttioliomalli tarkastelee näkymäfunktion tuottaman näkymän muutoksen edelliseen näkymään verrattuna, ja sitten älykkäällä vertailualgoritmilla päättää, mitä varsinaisessa dokumenttioliomallissa on muutettava, karsien kaikki ylimääräiset muutokset pois. Ajonaikaisessa järjestelmässä virtuaalinen dokumenttioliomalli pyörii Javascript-koodilla, eli sitä tulkitsee selaimen suorituskykyinen Javascript-tulkki.

4.5.2 Yhteentoimivuus Javascriptin kanssa

Koska Elm on varsin nuori sovellusaluekieli, kaikkea web-maailman toiminnallisuutta ei ole ehditty kirjoittamaan Elmille toimivaan ja testattuun muotoon. Siksi yhteensopivuus Javascriptin kanssa on oleellista ei pelkästään Elmille, vaan myös kaikille muillekin nuorille Javascriptiksi käännetyille kielille. Elm saavuttaa yhteensopivuuden eräänlaisella viestinvälitysjärjestelmällä, joka kulkee Elmissä nimellä Portit (engl. *Ports*) (Poudel 2018). Elmissä `port` on avainsana, jonka avulla luodaan

uusi viestityyppi antamalla sille pelkkä tyyppiannotaatio.

```
port lahettaja : String -> Cmd msg
port vastaanottaja : (String -> msg) -> Sub msg
```

Ylläolevassa koodilohkossa määriteltiin lähettäjäportti lähetettävälle viestille, sekä vastaanottajaportti vastaanotettaville viesteille. Tyyppiannotaation perusteella Elmin ajonaikainen järjestelmä osaa päätellä, kumpaan suuntaan portti toimii — lähetys vai vastaanotto. Viestinvälitysjärjestelmän varsinainen viestitys toimii Elmin oman ajonaikaisen järjestelmän läpi, ja tällä ylläpidetään funktionaalinen puhtaus itse kehittäjän kirjoittamassa Elm-koodissa. `Cmd` sekä `Sub` kummatkin tulevat Elmin `Platform`-kirjastosta ja ovat eräänlaisia Elmin alustan mekanismeja kuvata, mitä halutaan Elmin ajonaikaisen järjestelmän tekevän (Elm-lang 2018c). `Cmd`:n osalta voidaan tätä ajatella viestinä kuvailla Elmin ajonaikaiselle järjestelmälle välittää `String`-tyyppinen viesti eteenpäin vastaanottavalle Javascript-ohjelmalle. `Sub` vastaavasti puolestaan on pyyntö ajonaikaiselle järjestelmälle lähettää `String`-tyyppiset viestit Javascript-ohjelmasta eteenpäin kehittäjän kirjoittamalle Elm-sovellukselle.

Vastaavasti saman sovelluksen tai verkkosivun Javascript-ohjelmakoodissa olisi Elmin ajonaikaiselta järjestelmältä pyydetty tilaus, eli tapahtumakuuntelija uusia Elm-sovelluksesta saapuvia viestejä varten:

```
// sovellus-muuttujaan on sidottu Elm sovellus
sovellus.ports.lahettaja.subscribe(data => {/* ... */});
```

Koska Javascriptin integrointi Elm-sovelluksen kanssa on tehty viestijärjestelmäpohjaiseksi, saadaan Elmin parhaat ominaisuudet säilytettyä ongelmattomasti. Toisaalta viestijärjestelmän rakentaminen vaatii työtä, mutta samalla pakottaa kehittäjän määrittelemään kommunikaation Elm- ja Javascript-sovellusten välillä tarkasti. Tämän kaltainen viestinvälitysjärjestelmä muistuttaa huomattavasti yleisemmin funktio-ohjelmoinnissa – etenkin samanaikaisajoon tähtäävässä – käytettyä asynkronisesti toimivaa viestinvälitysjärjestelmää, missä yhteistä tilaa tai jaettua dataa ei ole, vaan kaikki kommunikointi tapahtuu lähetettävin ja vastaanotettavin viestein (Hebert 2013).

4.6 Jatkokehitys ja haasteet

Elmin tuoman tyyppiturvallisuuden ja puhtaiden funktioiden johdosta valtaosa käyttäjien tuottamista kolmannen osapuolen kirjastoista ovat toimintavarmempia kuin esimerkiksi Javascript-kirjastot yleisesti ovat, koska Elmin kääntäjä ei suostu kääntämään ohjelmakoodia, jossa on tyyppivirheitä tai sivuvaikutuksellisia funktioita.

Internetin keskustelupalstoilla on viime vuosina käyty aktiivista keskustelua Elmin tulevaisuuden suunnasta sekä siitä, kuinka projektia johdetaan. Vaikka latausmäärät ja käyttäjäkunta jatkaa tasaista kasvuaan, on moni ohjelmistokehittäjä turhautunut joko tästä, tai sitten eri toimintojen puutteesta. Esimerkiksi Redditissä, jossa yli 7000 Elmistä kiinnostunutta seuraajaa keskustelee Elmistä yleisesti kyseenalaistetaan, Evan Czaplickin varsin autoritäärinen tapa kontrolloida kaikkia muutoksia, lisäyksiä ja parannuksia mitä Elmin omaan lähdekoodiin tulee (Reddit 2018). Vuosittaisen web-käyttöliittymien ohjelmointiin liittyvän verkkokyselyn mukaan Elmiin pettyneiden ohjelmistokehittäjien toiseksi suurin tyytymättömyyden aihe oppimisen vaikeuden jälkeen onkin huolenaiheet projektin ylläpidosta (State of Javascript 2018). Lisäksi esimerkiksi Sandy Maguire hyvin kokeneena Haskell-ohjelmoijana valottaa turhautumistaan blogikirjoituksessaan *Elm Is Wrong* kielen puutteellisista ominaisuuksista korkeatasoisempaan abstraktioon (Maguire 2016). Maguire on osaltaan myös tyytymätön tapaan, jolla Elmin jatkokehitystä johdetaan.

Lopulta Elmiä ei kehitetä vain omassa eristetyssä ympäristössään, vaan se kilpailee aktiivisesti suosiosta muiden ratkaisujen kanssa kappaleessa 3.1 avattuihin ongelmiin. Näistä eniten samaan kohderyhmään pyrkivät vetoamaan muut funktio-ohjelmointiin perustuvat Javascriptin välttelystrategiat — kenties tunnetuimpina näistä Purescript sekä Facebookin ylläpitämä ReasonML. Czaplickikin onkin ymmärtänyt kehittäjäyhteisön tärkeyden ekosysteemin kehityksessä sekä pohtinut eri strategioita saada enemmän ohjelmistokehittäjiä aktiivisesti mukaan (Youtube, LLC 2018a,b, ks. kohdat *32:43*; *14:10*).

Kieltä ja alustaa halutaan jatkokehittää kattamaan suuremman osan web-alustan (ks. Web Platform 2018) toiminnallisuudesta (Github 2018c, ks. *roadmap.md*). Evan

Czaplickin oma alkuperäisen laajemman vision mukaisesti Elmiä aiotaan tulevaisuudessa jatkokehittää ennen kaikkea paremmin reaaliaikaohjelmointia (Czaplicki 2012, s. 39) sekä asynkronista ohjelmointia (Czaplicki & Chong 2013, s. 5) tukevaksi. Käytännössä kummatkin edellämainitut on tarkoitus jatkossa kuitenkin toteuttaa edellisessä 4.5.2 kappaleessa Erlangin prosessien välisellä viestipohjaisella actor-mallilla FRP:n sijaan (Elm-lang 2018b). Itse asiassa tätä varten on jo rakennettu eräänlainen kokeellinen ajastaja (engl. *scheduler*) Elmiin, joka on ottanut vaikutteita juuri Erlangin virtuaalikoneen ajastajasta.

Elmin kehityksessä näytetään jatkossa keskittyvän aktiivisesti tiettyihin vaikeisiin web-kehityksen haasteisiin. Yleisesti reaaliaikaiset alustat kasvattavat jatkuvasti suosiotaan, mutta vasta aika tulee lopulta näyttämään, oliko keskittyminen tähän oikea ratkaisu juuri Elmin jatkokehityksen kannalta.

5 Yhteenveto

Web-käyttöliittymien ohjelmoinnissa ollaan tällä hetkellä eräänlaisessa murrosvaiheessa, jossa Javascriptin ongelmia on nyt tunnustettu. Niitä yritetään toisaalta parannella, mutta toisaalta myös vältellä niiden kanssa tekemisiin joutumista. Elm selkeästi edustaa näistä jälkimmäistä koulukuntaa, koska on täysin uusi ohjelmointikieli. Elmiä käyttävät ohjelmistokehittäjät arvostavat eniten tapaa, jolla Elmissä ohjelmoidaan (State of Javascript 2018). Vuonna 2014 tehdyn tutkimuksen johtopäätösten mukaan funktionaalisilla ohjelmointikielillä tuotetaan jokseenkin parempilaatuista koodia, ja funktionaalisissa kielissä staattinen tyyppijärjestelmä myös osaltaan johtaa parempilaatuiseen koodiin (Ray et al. 2012). Vuoden 2017 web-kehityskyselyssä Elmin kielen osalta juuri staattinen tyyppijärjestelmä, puhtaat funktiot sekä Elm-arkkitehtuuri saavat eniten kiitosta ohjelmistokehittäjiltä (Hicks 2017).

Elmin alustan neljä komponenttia kukin tarjoavat tietynlaisen ratkaisun Javascript-pohjaisen web-käyttöliittymien ohjelmoinnin tiettyihin haasteisiin. Itse Elm-ohjelmointikieli puhtaine funktioineen pakottavat kehittäjän kirjoittamaan sivuvaikutuksetonta ja siten hyvin testattavaa ja ylläpidettävää koodia. Elm-arkkitehtuuri taas tarjoaa ratkaisuna sovelluksen tilan ja näkymän ylläpidon haasteille valmista ajatusmallia, jota käyttämällä kaikki sovelluksen tila saadaan mallinnettua systemaattisesti, ja näkymä automaattisesti vastaamaan tilamuutoksia. Dokumenttioliopuun manipuloinnin hitauden ja laskennallisen kalleuden puolestaan ratkaisee Elmin ajonaikainen järjestelmä, joka tekee varsinaisen dokumenttioliomallin manipuloinnin virtuaalisen dokumenttioliomallialgoritmin kautta. Lopulta Elm-kääntäjä antaa sekä selkeästi ymmärrettävää palautetta ohjelmakoodin laadusta ja ongelmista, että lopulta myös kääntää koodin ja paketoi kaikki paketit ja ajonaikaisen järjestelmän yhteen helposti käytettävään selaimen ymmärtämään pakettitiedostoon. Elmin ratkaisut ovat kyselyiden ja kasvavien latausmäärien perusteella jokseenkin pidettyjä, mutta toisaalta projektin ylläpito kritiikin kohteena. Lopulta kehittäjäyhteisön ja ekosysteemin menestyksekäs kehittäminen saattaakin olla ratkaisevassa asemassa Elmin tulevaisuuden suosion kannalta.

Kirjallisuutta

Alic, D., Omanovic, S. & Giedrimas, V. 2016. *Comparative analysis of functional and object-oriented programming*. In Proceedings of MIPRO 2016. https://doi.org/10.1109/MIPRO.2016.7522224.

Armstrong, J. 2003. *Making reliable distributed systems in the presence of software errors*. Final version. A Dissertation submitted tothe Royal Institute of Technology. Stockholm, Sweden:The Royal Institute of Technology.

Ray, Baishakhi., Posnett, D., Filkov, V. & Devanbu, P. 2014. *A large scale study of programming languages and code quality in github*. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering s. 155–165. https://doi.org/10.1145/2635868.2635922.

Caspi, P., Hamon, G. & Pouzet, M. 2012. *Synchronous Functional Programming:The Lucid Synchrone Experiment*. Real-Time Systems: Description and Verification Techniques: Theory and Tools, Vol. 1. Hermes publisher: Gieres, France.

Harrison, J. 1997. *Introduction to Functional Programming*. Cambridgen yliopiston opetusmateriaali. Saatavilla WWW-muodossa <URL: https://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.pdf>. Viitattu 6.12.2018.

Conal, E. & Hudak, P. 1997. *Functional Reactive Animation*. In Proceedings of the second of the second ACMSIGPLAN international conference on Functional Programming. ICFP '97, s. 263–273. New York, USA. https://doi.org/10.1145/258949.258973.

Courtney, A. 2003. *Functionally Modeled User Interfaces*. In Proceedings of the Tenth Workshop on Design, Specification and Verification of Interactive Systems, June, 2003. https://doi.org/10.1007/978-3-540-39929-2_8.

Courtney, A. & Elliott, C. 2001. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop, s. 41–69.

Crockford, D. 2008. *Javascript: The Good Parts*. O'Reilly Media Inc: Sebastopol, California, The United States of America.

Czaplicki, E. & Chong, S. 2013. *Asynchronous Functional Reactive Programming for*

GUIs. Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2013, s. 411–422. https://doi.org/10.1145/2491956.2462161.

Czaplicki, E. 2012. *Elm: Concurrent FRP for Functional GUIs*. Senior thesis, Harvardin yliopisto, Yhdysvallat. Saatavilla WWW-muodossa <URL: https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf>.

ECMA International. 2017. *ECMAScript® 2017 Language Specification (ECMA-262, 8th edition, June 2017)*. Ecma International - European association for standardizing information and communication systems. Saatavilla osoitteessa <URL: https://www.ecma-international.org/ecma-262/8.0/index.html>. Viitattu 3.2.2019.

Elm-lang.org. 2018. *Elm Syntax*. Dokumentaatio. Saatavilla WWW-muodossa <URL: https://elm-lang.org/docs/syntax>. Viitattu 21.10.2018.

Elm-lang.org. 2018. *A Farewell to FRP*. Blogikirjoitus. Saatavilla WWW-muodossa <URL: https://elm-lang.org/blog/farewell-to-frp>. Viitattu 18.12.2018.

Elm-lang.org. 2018c. *A Farewell to FRP*. Dokumentaatio. Saatavilla WWW-muodossa <URL: https://guide.elm-lang.org/interop/ports.html>. Viitattu 18.12.2018.

Elm-lang.org. 2018d. *Pure Functions*. Dokumentaatio. Saatavilla WWW-muodossa <URL: https://elmprogramming.com/pure-functions.html>. Viitattu 18.12.2018.

Elm-lang.org. 2018e. *Custom types*. Dokumentaatio. Saatavilla WWW-muodossa <URL: https://guide.elm-lang.org/types/custom_types.html>. Viitattu 19.12.2018.

Elm-lang.org. 2018f. *Compiler Errors for Humans*. Blogikirjoitus. Saatavilla WWW-muodossa <URL: https://elm-lang.org/blog/compiler-errors-for-humans>. Viitattu 19.12.2018.

Flanagan, D. 2011. *Javascript: The Definitive Guide*. O'Reilly Media: Inc. Sebastopol, California, The United States of America.

Gabbrielli, M. and Martini, S. 2010. *Programming Languages. Principles and Paradigms*. London: Springer. https://doi.org/10.1007/978-1-84882-914-5.

GitHub, Inc. 2018a. *Showing 5,690,832 available repository results*. Suoritettu ohjelmavarastohaku tähtimäärän perusteella. Saatavilla WWW-muodossa <URL: https://github.com/search?q=stars%3A%3E0&s=stars&type=Repositories/>. Viitattu 21.10.2018.

GitHub, Inc. 2018b. *Elm - Official organization for developing Elm's compiler and core tools*. Yhteisön elm-lang profiilisivu. Saatavilla WWW-muodossa <URL: https://github.com/elm-lang>. Viitattu 17.12.2018.

GitHub, Inc. 2018c. *Elm Projects*. Yhteisön elm-lang projects-ohjelmavarasto. Saatavilla WWW-muodossa <URL:https://github.com/elm/projects> Viitattu 17.12.2018.

GitHub, Inc. 2018d. *The Elm Compiler*. Elm-ohjelmointikielen kääntäjän ohjelmavarasto. Saatavilla WWW-muodossa <URL:https://github.com/elm/compiler>Viitattu 17.12.2018.

Haverbeke, M. 2018. *Eloquent Javascript*. 3rd edition. Saatavilla WWW-muodossa <URL: https://eloquentjavascript.net/Eloquent_JavaScript.pdf>. Viitattu 9.12.2018.

Hebert, F. 2013. *Learn You Some Erlang for Great Good! More On Multiprocessing*. No Starch Press. Saatavilla WWW-muodossa <URL: https://learnyousomeerlang.com/more-on-multiprocessing#secret-messages>. Viitattu 9.12.2018.

Hicks, B. 2017. *State of Elm 2017 Results*. Brian Hicks. Blogikirjoitus. Saatavilla WWW-muodossa <URL: https://www.brianthicks.com/post/2017/07/27/state-of-elm-2017-results/>. Viitattu 15.12.2018.

Hinsen, K. 2009. *The Promises of Functional Programming*. Computing in Science and Engineering. Volume 11, Issue 4 Jul-Aug, s. 86–90.

Horemans, J., Renders, B., Devriese, D. & Piessens, F. 2018. *Trends in Functional Programming*. Revised Selected Papers. 18th International Symposium, Canterbury, UK. https://doi.org/10.1007/978-3-319-89719-6.

Hughes, J. 1990. *Why Functional Programming Matters*. Research Topics in Functional

Programming, s. 17–42. Boston: Addison-Wesley Longman Publishing Co., Inc. https://doi.org/10.1093/comjnl/32.2.98.

Jackson, K. (1977). *Parallel processing and modular software construction*. Design and Implementation of Programming Languages. Lecture Notes in Computer Science. 54, s. 436–443. https://doi.org/10.1007/BFb0021415.

Jones, S.L.P. 1987. *The Implementation of Functional Programming Languages*. Series in Computer Science. Hertfordshire, the UK: Prentice-Hall International.

karagkasidis, A. 2008. *Developing GUI Applications: Architectural Patterns Revisited*. EuroPLoP 2008: 13th Annual European Conference on Pattern Languages of Programming, Irsee, Germany, July 9-13, 2008.

Lipova, M. 2011. *Learn You a Haskell for Great Good!*. E-kirja. Saatavilla WWW-muodossa <URL: http://learnyouahaskell.com/chapters>. Viitattu 7.12.2018.

Liskov, B., Zilles, S. 1974. *Programming with abstract data types*. ACM SIGPLAN Notices. 9, s. 50–59.

Maguire, S. 2016. *Elm Is Wrong*. Reasonably Polymorphic. Blogikirjoitus. <URL: http://reasonablypolymorphic.com/blog/elm-is-wrong/>. Viitattu 6.12.2018.

Mealy, G. 1955. *A Method for Synthesizing Sequential Circuits*. Bell System Technical Journal. s. 1045–1079.

Microsoft. 2018. *TypeScript in 5 minutes*. TypescriptLang.org. Saatavilla WWW-muodossa <URL: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>. Viitattu 8.12.2018.

Mozilla (a). 2018a. *Concurrency model and Event Loop*. MDN web docs. Saatavilla WWW-muodossa <URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop> Viitattu 21.10.2018.

Mozilla. 2018b. *async function*. MDN web docs. Saatavilla WWW-muodossa <URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function> Viitattu 18.12.2018.

Mozilla. 2018c. *Introduction to the DOM*. MDN web docs. Saatavilla WWW-muodossa <URL: https://developer.mozilla.org/en-US/docs/Web/

API/Document_Object_Model/Introduction> Viitattu 18.12.2018.

Noble, R. & Runciman, C. 1994. Functional Languages and Graphical User Interfaces – a review and a case study.

Npm Inc. 2018. *Elm - npm*. Saatavilla WWW-muodossa <URL: https://www.npmjs.com/package/elm>. Viitattu 20.10.2018.

Quora Inc. 2018. *Why is JavaScript so hated?*. Julkinen kysymys-vastaussivusto. Saatavilla WWW-muodossa <URL: https://www.quora.com/Why-is-JavaScript-so-hated>. Viitattu 20.10.2018.

Poudel, P. 2018. *Beginning Elm*. E-kirja. Saatavilla WWW-muodossa <URL: https://elmprogramming.com/> Viitattu 21.10.2018.

Reddit. 2018. *Is Evan Killing Elm's momentum?*. Verkkokeskustelu. Saatavilla WWW-muodossa <URL: https://www.reddit.com/r/elm/comments/7zk0dy/is_evan_killing_elms_momentum/>. Viitattu 6.12.2018.

Reimann, D. 2016. *The Elm Architecture*. Blogikirjoitus. Saatavilla WWW-muodossa <URL: https://dennisreimann.de/articles/elm-architecture-overview.html>. Viitattu 21.10.2018.

Simon, L. 2018. *Minimizing browser reflow*. Blogikirjoitus. PageSpeed Insights, Google Developer. Saatavilla WWW-muodossa <URL: https://developers.google.com/speed/docs/insights/browser-reflow>. Viitattu 7.12.2018.

Stack Overflow. 2018. *Developer Survey Results 2018*. Verkkokyselyn tulokset. Saatavilla WWW-muodossa <URL: https://insights.stackoverflow.com/survey/2018/>. Viitattu 16.12.2018.

State of Javascript. 2018. *Javascript flavors - Elm*. Verkkokyselyn tulokset. Saatavilla WWW-muodossa <URL: https://2018.stateofjs.com/javascript-flavors/elm/>. Viitattu 23.12.2018.

State of Javascript. 2018. *State Management Tools – Results*. Verkkokyselyn tulokset. Saatavilla WWW-muodossa <URL: https://2017.stateofjs.com/2017/state-management/results/>. Viitattu 23.12.2018.

Three Devs and a Maybe. 2015. *The History of React and Flux with Dan Abramov*. Podcast. Saatavilla osoitteessa <URL: https://threedevsandamaybe.

com/the-history-of-react-and-flux-with-dan-abramov/>. Viitattu 9.12.2018.

Turner, D. 2012. *Some History of Functional Programming Languages*. Proceedings of the 2012 Conference on Trends in Functional Programming - Volume 7829. https://doi.org/10.1007/978-3-642-40447-4_1.

The World Wide Web Consortium. 2005a. *Document Object Model (DOM)*. Verkkosivu. Saatavilla osoitteessa <URL: https://www.w3.org/DOM/#what>. Viitattu 9.12.2018.

The World Wide Web Consortium. 2004. *Architecture of the World Wide Web, Volume One*. Verkkosivu. Saatavilla osoitteessa <URL: https://www.w3.org/TR/webarch/>. Viitattu 18.12.2018.

Web Platform, 2018. *The Web platform: Browser technologies*. Verkkosivu. Saatavilla osoitteessa <URL: https://platform.html5.org/>. Viitattu 18.12.2018.

Youtube, LLC., 2018a. *"What is Success?" by Evan Czaplicki*. Puhe Elm-konferenssissa. Saatavilla osoitteessa <URL: https://youtu.be/uGlzRt-FYto>. Viitattu 19.12.2018.

Youtube, LLC., 2018b. *"Code is the Easy Part" by Evan Czaplicki*. Puhe Elm-konferenssissa. Saatavilla osoitteessa <URL: https://youtu.be/DSjbTC-hvqQ>. Viitattu 19.12.2018.