Tampereen yliopisto

Anssi Oinonen

C++ VS. HASKELL

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Lokakuu 2020

TIIVISTELMÄ

Anssi Oinonen: C++ vs. Haskell
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Lokakuu 2020

C++ ja Haskell ovat ohjelmointikieliä, joiden kehitys alkoi 1980-luvulla ja jotka edustavat kahta erilaista ohjelmointiparadigmaa. C++ perustuu imperatiiviselle ja Haskell funktionaaliselle ohjelmointiparadigmalle ja kumpikin paradigma perustuu 1930-luvulla kirjoitetuille tieteellisille tutkimuksille.

Työssä käydään läpi kummankin kielen historiaa sekä kielten kehitykseen johtaneita syitä ja niiden seurauksia kielten rakenteille tutkimalla kielten edeltäjiä, niiden synnyistä kirjoitettuja artikkeleita ja niiden määrittelyjä. C++:sta haluttiin tehokas ohjelmointikieli, jolla kirjoitetut ohjelmat olisivat helposti siirrettävissä tietokoneiden välillä, ja joka tarjoaisi ohjelmoijille korkean tason abstrahointitapoja, joiden avulla ohjelmia olisi helpompi suunnitella. Haskellin tapauksessa haluttiin kehittää puhdas ja laiska funktionaalinen kieli sovellusten kehittämis-, opetus- ja tutkimustarkoitukseen sekä yhdistää hajautunutta funktionaalisten kielten kehitystä.

C++ kehitettiin C-kielen pohjalta lisäämällä C:hen tuki Simula-tyyppisille luokille. C:n ja Simulan edeltäjät voidaan jäljittää ALGOL 60 -kieleen, mikä on osaltaan vaikuttanut C++:n ominaisuuksiin. ALGOL 60 kieli kehitettiin komiteayhteistyössä samoin kuin Haskell. Historioidensa takia C++ on kehittynyt tarjoamaan paljon hyödyllisiksi katsottuja ominaisuuksia olio-ohjelmoinnin ja imperatiivisen ohjelmointiparadigman ulkopuoleltakin, kun taas Haskellin uusien ominaisuuksien on pitänyt pystyä toimimaan laiskasti ja puhtaasti, mikä on rajoittanut uusien ominaisuuksien lisäyksiä, mutta on myös tarjonnut tilaisuuksia kehittää ohjelmointikielten toiminnallisuuksia uusilla tavoilla, kuten tyyppiluokilla ja monadeilla.

Kielten eri lähtökohdat ja tavoitteet ovat mahdollistaneet kahden erilaisen kielen kehittymisen samaan aikaan vastaamaan kasvavien ohjelmien tuomiin ongelmiin. Ne ovat tuoneet ohjelmia lähemmäs sovellusalueita ja kauemmas monimutkaisemmiksi muuttuvista tietokoneista.

Avainsanat: C++, Haskell, ohjelmointiparadigma, imperatiivinen, funktionaalinen, historia

ALKUSANAT

“Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.“

-Ludwig Wittgenstein

“A language that doesn't affect the way you think about programming, is not worth knowing.”

-Alan Perlis

SISÄLLYSLUETTELO

1. JOHDANTO

Tämän työn tarkoituksena on vertailla kahden ohjelmointikielen, Haskellin ja C++:n, syntyä ja suoria edeltäjiä, niiden erilaista lähestymistapaa ohjelmointiin, sekä miten ne tahoillaan ratkaisevat samoja ongelmia. Kielet valittiin niiden ominaisuuksien sekä samaan aikaan tapahtuneen kehityksen takia. C++ perustuu C-kieleen, joka on yhä yleisesti käytössä oleva ohjelmointikieli varsinkin sulautetuissa järjestelmissä, joissa suoritustehoa ja tilaa on rajoitetusti tarjolla. C++:n alkuperäinen tarkoitus oli lisätä C-kieleen tuki olio-ohjelmoinnille [1], joka löytyy nykyisin myös muista suosituista ohjelmointikielistä, kuten Javasta ja Pythonista. Haskell taas on alun perin kehitetty funktionaalisten kielten tutkimus- ja opetuskäyttöön sekä sovellusten ohjelmointiin ja Haskellin tarkoituksena olikin yhdistää monta eri funktionaalisista kielistä kiinnostunutta ryhmää, joista jokainen oli kehittämässä omaa kieltään, ja tarjota yhteinen pohja, jolla funktionaalista ohjelmointia voitaisiin paremmin hyödyntää [2]. Haskell on säilyttänyt puhtaan funktionaalisuutensa, mikä ohjaa ohjelmoijan kirjoittamaan ohjelmia eri tavalla C++:aan verrattuna. Sillä C ja sitä kautta C++ pyrkii antamaan ohjelmoijalle käyttöön kaikki tietokoneen tarjoamat työkalut, joten sen sisäinen rakenne simuloi tarkkaan tietokoneen rakennetta. Haskell taas keskittyy enemmän matemaattiseen täydellisyyteen perustumalla lambda-kalkyyliin, jolloin kaikkia sellaisia rakenteita, joita tietokoneen toteutus mahdollistaisi, ei voida toteuttaa rikkomatta lambda-kalkyylin periaatteita ja menettämällä siitä saatuja etuja kuten puhtautta ja laiskuutta.

Muita ohjelmointikielten vertailuja on tehty vertailemalla kielillä kirjoitettujen ohjelmien pituutta, nopeutta, muistinkulutusta, luotettavuutta ja ohjelmointiin kulunutta aikaa [3] tai tutkimalla ohjelmien tiedettyjen virheiden määrää korjattujen bugien muodossa [4; 5]. Tutkimusta on myöskin tehty keinoista, joilla ohjelmointikieliä voitaisiin verrata [6; 7]. Ohjelmointikielten vertailujen ongelmana on usein se, että vertailu suoritetaan vertailemalla ohjelmointikielillä kirjoitettuja ohjelmia, jolloin vertailun tulokseen vaikuttaa ohjelmointikielten lisäksi ohjelmoijien kyvyt, käytetyn kääntäjän tehokkuus, sekä ohjelmoitavan ohjelman ominaisuudet, jotka voivat soveltua enemmän yhdelle ohjelmointikielelle kuin toiselle. Näiden syiden takia tämä työ keskittyy vertailemaan kielillä kirjoitettujen ohjelmien sijaan kielten syntyä ja kehitystä tutkimalla kielten synnystä kirjoitettuja papereita, innoittajana toimineita ajatuksia, kielten edeltäjiä ja kehitystä. Pääpaino on kielten edeltäjissä ja innoittajissa, joita on käsitelty luvussa 2, ja joiden

pohjalta on tehty kielten vertailu luvussa 3, jossa vertaillaan kielten lähtökohtia ja kehitystä sekä tarkastellaan IO-operaatioiden toteutuksia kielissä. Luvussa 4 on yhteenveto työstä.

2. KIELTEN HISTORIA

Vuonna 1936 sekä Alonzo Church että Alan Turing julkaisivat omat todistuksensa siitä, ettei Hilbertin esittämää Entscheidungsproblemea voi ratkaista ja he tulivat samalla luoneeksi teoreettiset mallinsa tietokoneelle. Alan Turing kehitti Turingin koneen ja Alonzo Church lambdakalkyylin.

Turingin kehittämä Turingin kone koostuu nauhasta, josta syötteet luetaan ja johon ohjelman tulos kirjoitetaan, sekä laskennan suorittavasta tilakoneesta. Turingin universaalissa koneessa nauhan alussa on kuvaus Turingin koneesta, jonka jälkeen on sille tarkoitettu syöte [8]. Tämä muistuttaa läheisesti von Neumannin arkkitehtuuria, johon nykytietokoneet perustuvat: tietokoneen muisti sisältää sekä kuvauksen ohjelmasta että ohjelman käyttämän datan.

Church puolestaan kehitti lambdakalkyylin, jonka perustana toimivat puhtaat funktiot, joilla ei ole sisäistä tilaa eli ulostulo riippuu ainoastaan funktioon syötetyistä arvoista [9]. Nämä kaksi mallia, toinen tilaan perustuva ja toinen tilaton, vastaavat toiminnallisuudeltaan toisiaan Churchin–Turingin teesin mukaan.

2.1 Imperatiivinen ohjelmointiparadigma

Turingin kehittämä malli vastaa tietokoneen fyysistä rakennetta ja sitä myötä myös tietokonetta ohjaavia ohjelmia. Ensimmäisiä ohjelmointikieliä olivat konekielet, joissa ohjelma koostui konekäskyistä, joilla ohjattiin prosessoria suorittamaan laskentaa tai hakemaan ja tallentamaan syötteitä ja tuloksia. Koska aikaisissa tietokoneissa oli vain yksi prosessori, joka suoritti yhden konekäskyn kerrallaan, tuli ohjelmista luonnollisesti sarjallisia. Tästä sai alkunsa imperatiivinen ohjelmointiparadigma, jonka keskeisenä ideana on antaa tietokoneelle käskyjä, joiden järjestelmällisen suorituksen seurauksena päästään ohjelmoijan haluamaan lopputulokseen.

Tietokoneiden alkuaikoina tietokoneita pääasiassa käyttäneet matemaatikot eivät välttämättä halunneet sälyttää tietokoneelle tehtäviä, jotka voitaisiin tehdä käsin [10]. Tietokoneiden ollessa kalliita oli halvempaa tehdä kaikki mahdollinen valmisteleva työ käsin ja antaa tietokoneen tehdä vain tarvittava laskenta. Esimerkiksi Davisin mukaan ensimmäisiä tietokoneita kehittänyt John von Neumann katsoi assemblerin turhaksi, sillä ohjelmat voitiin koodata konekielellä ja assembler olisi vain kuluttanut arvokkaita tietokoneresursseja [11]. Knuthin mukaan taas Alick E. Glennie oli sitä mieltä, että ”automaattinen ohjelmointi” eli korkeamman tason ohjelmointikielet olivat yleisessä

tiedossa 1950-luvun alussa, mutta ohjelmointikielten täytyi toimia käytössä olleiden tietokoneiden asettamissa rajoissa eikä kaikkia hyödylliseksi katsottuja ominaisuuksia ollut mahdollista toteuttaa [10].

Muutos ohjelmointikielten kehityksessä tapahtui Perlisin mukaan vuosina 1956 ja 1957 korkeamman tason ohjelmointikielien tullessa markkinoille. Niiden avulla ohjelmoija pystyi keskittymään enemmän algoritmisiin kuin tietokoneen toteutukseen liittyviin ongelmiin. [12] Yksi näistä kielistä, ja Perlisin mukaan suosituin [12], oli uutta IBM 704 -tietokonetta varten kehitetty ”IBM Mathematical FORmula TRANslating System”, lyhyemmin FORTRAN [13]. FORTRANista tuli suosittu IBM:n tietokoneiden levitessä, ja tämä lisäsi kiinnostusta uusia korkeamman tason ohjelmointikieliä kohtaan [12].

Toinen muutos tapahtui, kun ohjelmien koon kasvaessa rakenteettoman ohjelmoinnin ongelmista tuli ilmeisiä. Isoja ohjelmia oli vaikea lukea ja ymmärtää, virheiden etsiminen ja korjaaminen oli vaikeaa, ja nämä ongelmat moninkertaistuivat usean ohjelmoijan projekteissa. Edsger W. Dijkstra halusi yleistää ohjelmien jakamisen osiin, joita olisi helpompi hallita ja ymmärtää ja jotka abstrahoisivat ohjelman toimintaa helposti ymmärrettäviksi moduuleiksi [14]. Näistä ajatuksista kehittyi proseduraalinen ohjelmointiparadigma, jossa ohjelma jaetaan proseduureihin, mikä helpottaa isojen ohjelmien kehitystä.

2.1.1 ALGOL

Ihmisten alkaessa nähdä korkeamman tason ohjelmointikielten edut synnytti lisääntynyt innostus kieliä kohtaan monia kieliä monissa eri paikoissa. Tietokonevalmistajilla oli omat kielensä eri koneille, ja yliopistotkin kehittivät näille koneille omia kieliään. Standardien puuttuessa niin kielten kuin tietokoneidenkin osalta oli vaikea esittää ideoita kielten välillä. Tämä johti haluun kehittää yhtenäinen standardikieli, jota eivät ohjaisi alla olevan tietokoneen konekäskyt vaan korkeamman tason algebralliset säännöt. [12]

Saksalainen Gesellschaft für Angewandte Mathematik und Mechanik, GAMM, ja yhdysvaltalainen Association for Computing Machinery, ACM, alkoivat tehdä yhteistyötä luodakseen yhteisen kansainvälisen algebrallisen kielen. Vuonna 1958 Zürichin tapaamisessa päästiin sopuun kielestä, jonka nimeksi tuli The International Algebraic Language, joka vuotta myöhemmin muuttui ALGOLiksi ja ajan myötä ALGOL 58:ksi. ALGOL 58 oli tarkoitettu perustaksi, jolta lähteä rakentamaan kieltä, ja ACM ja GAMM alkoivat kerätä kommentteja, miten kieltä voisi parantaa. Näistä muodostui 18 kuukautta myöhemmin pohja ALGOL 60 -kielelle. [12; 15]

ALGOLista oli alusta asti kolme tasoa. Ensimmäinen taso oli referenssikieli, joka oli täydellinen kuvaus kielestä, jonka komitea loi. Lisäksi oli julkaisukieli, jota käytettiin

tieteellisissä julkaisussa kuvaamaan algoritmeja. Tämä kieli soveltui paremmin käsinkirjoittamiseen ja tulostamiseen kuin referenssikieli. Julkaisukieli esimerkiksi salli alaindeksien, kreikkalaisten numeroiden ja eksponenttien käytön [16]. Lisäksi se salli sekä pisteen että pilkun käytön desimaalierottimena, mikä oli tärkeä ominaisuus transatlanttiselle kielelle [12]. Kolmantena tasona olivat tietokoneille toteutetut kielet, joista oli riisuttu referenssikielen ominaisuuksia, joita ei voitu kyseisellä tietokoneella suorittaa. Esimerkiksi tietokoneen hyväksymät merkit vaihtelivat eri tietokoneiden välillä, mikä vaikutti ominaisuuksien toteuttamiskelpoisuuteen [16]. Tämän takia ALGOLin referenssikielestä jäi muuan muassa puuttumaan tuki IO-operaatioille, sillä niiden katsottiin riippuvan liiaksi käytettävästä koneesta [12].

Sebestan mukaan ALGOL 60 onnistui tavoitteessaan tulla julkaisukieleksi, ja sitä käytettiin yli 20 vuoden ajan pääasiallisena algoritmien julkaisukielenä. ALGOL 60 vaikutti myös uusiin kieliin ja on Sebestan mukaan vaikuttanut jokaiseen vuoden 1960 jälkeen suunniteltuun imperatiiviseen kieleen. ALGOL 60:n huonona puolena oli monimutkaisuus ja vaikea ymmärrettävyys innovatiivisten ominaisuuksien takia, jotka johtivat siihen, ettei sen käyttö juuri yleistynyt Yhdysvalloissa, eikä siitä tullut suosituinta kieltä Euroopassakaan, vaikka Euroopassa pelättiinkin amerikkalaisen FORTRANin ylivaltaa. [17]

2.1.2 CPL, BCPL ja B

Combined Programming Language, CPL, kehitettiin ALGOL 60 -kielen pohjalta ja sen tarkoituksena oli tuoda kieli lähemmäs konetta. CPL:n kehittäjien mielestä ALGOL 60 oli hyvä kieli, mutta se keskittyi liiaksi kuvaamaan algoritmeja ja unohti alla olevan tietokoneen. Esimerkkinä mainitaan liukulukujen tarkkuus, joka on tietokonetta käytettäessä tärkeä tieto, mutta jota ei ALGOLissa otettu huomioon. Lisäksi he laajensivat datan kuvaamisen muotoja. [18]

Basic Combined Programming Language, BCPL, kehitettiin, koska CPL-kääntäjiä oli vaikea toteuttaa kielen monimutkaisuuden takia. Niinpä kielestä poistettiin vaikeasti toteutettavia ominaisuuksia, joista yksi oli – mielenkiintoista kyllä – tuki lambdakalkyylille. [19]

Ken Thompson kehitti B-kielen, joka Dennis Ritchien mukaan oli pienempään tilaan mahtuva BCPL, johon Thompson lisäsi omia ideoitaan. B:ssä alkoi ilmetä ongelmia siirryttäessä DEC:n PDP-7 tietokoneesta PDP-11 koneeseen. B käytti vain yhtä tietotyyppiä, solua, joka sopi hyvin yhteen vanhemman PDP-7 mallin muistipaikkoina käyttämien sanojen kanssa, mutta uudempi PDP-11 käytti muistipaikkoina tavuja, jolloin syntyi tarve käyttää vaihtelevan kokoisia muuttujia sekä kielen rakenteita, jotka

paremmin vastaisivat tietokoneen tavurakenteita. Ritchie laajensi B:tä lisäämällä siihen merkki- ja kokonaisluku-tietotyypit, niiden osoittimet ja taulukot. Lisäksi hän kirjoitti kielen kääntäjän tuottamaan PDP-11:n konekieltä ja kutsui uutta kieltä New B:ksi, josta myöhemmin kehittyi C-kieli. [20]

2.1.3 C

Dennis Ritchie kehitti C-kielen 70-luvun alussa Unix-käyttöjärjestelmää varten. Se perustui Ken Thompsonin aikaisemmin Unixille kirjoittamaan B-kieleen. Kyseiset kielet pohjaavat omat rakenteensa tietokoneen rakenteisiin ja tarjoavat monimutkaisemmat ominaisuudet kirjastoina. Kielten abstraktiot ovat kuitenkin tarpeeksi korkealla tasolla mahdollistaakseen helpon siirrettävyyden eri järjestelmien ja arkkitehtuurien välillä. [20]

C:n ja sen edeltäjien ominaisuuksia ja niiden toteutuksia määräsivät käytössä olleet tietokoneet. Esimerkiksi kirjoitettujen ohjelmien piti olla tiiviitä, jotta ne voitiin kääntää tietokoneen tilaltaan rajatussa muistissa.

Ritchie muun muassa muutti taulukoiden semantiikkaa BCPL:n semantiikasta, jossa muistiin tallennettiin osoitin taulukon ensimmäiseen alkioon, nykyisinkin C:ssä käytössä olevaan semantiikkaan, jossa taulukon nimi korvataan osoittimella taulukon ensimmäiseen alkioon. Syynä tähän muutokseen oli tuki tietueihin sijoitetuille taulukoille. [20]

```
int (*t)[5];
```
(1)

Ohjelma (1) kuvaa miltä BCPL:n tyylisen taulukon luonti näyttäisi C:ssä. Siinä näkyy myös, miten kokonaisluku-, osoitin- ja taulukko-tyyppi yhdistetään C:ssä ja tähän Ritchie otti mallia ALGOL 68 -kielestä. Luotuaan tyyppijärjestelmän, sen syntaksin sekä kääntäjän alkoi Ritchie käyttää kielestä nimeä C. [20]

Vuonna 1973 C ja sen kääntäjä olivat tarpeeksi kehittyneitä, jotta Unix-järjestelmä voitiin kirjoittaa C:llä PDP-11 -tietokoneelle. Kieli osoittautui tarpeeksi tehokkaaksi ja hyödylliseksi, jotta myös muita Unixin ohjelmia kirjoitettiin C:llä ensin PDP-11:lle ja myöhemmin muille alustoille. Tätä varten Steve Johnson kehitti pcc-kääntäjän, jonka tarkoituksena oli helpottaa C:llä kirjoitettujen ohjelmien siirtoa uusille laitteille. Tämä johti kierteeseen, jossa Unixin ja C:n suosio ruokkivat toisiaan, ja mahdollisuus siirtää C:llä kirjoitettuja ohjelmia muille laitteille ja alustoille vauhditti sen leviämistä. [20]

2.1.4 Simula

Kristen Nygaard ja Ole-Johan Dahl kehittivät SIMUlation LAnguage -kielen, josta he käyttivät nimitystä SIMULA, 1960-luvun puolivälissä ALGOL 60 -kielen pohjalta

simuloidakseen rinnakkain toimivia, monimutkaisia tietorakenteita vaativia järjestelmiä, kuten tietoliikenneverkkoja, liikennettä ja tuotantojärjestelmiä. [21]

Nygaard ja Dahl aloittivat Simulan kehityksen ajattelemalla järjestelmiä tietoverkkoina, joissa oli pysyviä aktiivisia komponentteja, jotka käsittelevät väliaikaisia passiivisia komponentteja. Laajentaessaan järjestelmien käsitettä he huomasivat, että tietoverkkojen sijaan yhteinen nimittäjä järjestelmien taustalla oli erilaiset prosessit, joita komponenteille suoritettiin. He päättivät rakentaa kielen, joka olisi ongelmakeskeinen eikä tietokonekeskeinen ja tämän seurauksena Simulan suorituskyky kärsi jonkin verran [22]. Suorituskykyongelmien seurauksena Bjarne Stroustrup päätti tehdä tehokkaamman olio-ohjelmointikielen [1].

2.1.5 C++

Bjarne Stroustrup kehitti C++:n edeltäjän, Luokallisen C:n, lisäämällä C-kieleen tuen Simula-tyylisille luokille [1]. Hän valitsi C-kielen sen tehokkuuden ja järjestelmänohjelmointikyvyn takia, mutta halusi lisätä siihen luokat, jotka tukivat korkeampia abstraktiotasoja. Stroustrupin mukaan tehokkuus ja joustavuus olivat alusta asti määrääviä tekijöitä C++:ssa. Hän arvosti Simulan ongelmalähtöisyyttä ja kielen rakenteita, jotka mahdollistivat ohjelman suunnittelun tietokoneen toteutuksen sijaan ratkaistavan ongelman kannalta. Tämä näkökulma auttoi Stroustrupia näkemään ohjelmistosuunnittelun rakentuvan luokkien päälle. Luokkarakenne auttoi myös hallitsemaan kasvavaa ohjelmaa paremmin, mutta Simulan toteutus ei skaalautunut yhtä hyvin. Tästä syystä hän päätti yhdistää Simulan luokkarakenteen C:n tehokkuuteen ja monipuolisuuteen sekä C-ohjelmien helppoon siirrettävyyteen eri koneiden välillä. [1]

C++:n tavoitteena oli tarjota yleiskäyttöisiä keinoja organisoida ohjelmia ja parantaa abstrahointitapoja. Tämä on johtanut muun muassa siihen, että C++:ssa ei ole sisäänrakennettua tukea rinnakkaiselle ohjelmoinnille, vaan kieli tarjoaa keinot, joilla käyttäjä voi itse toteuttaa rinnakkaisuuden haluamallaan tavalla. Stroustrup halusi, että C++:aa voisi käyttää kaikkialla, missä C:täkin voi. Tämä johti siihen, että C++n toteutuksen piti olla yhtä tehokas kuin C:n toteutus, eikä C:n ominaisuuksia saanut poistaa C++:sta, vaikka ne olisi katsottu rumiksi tai vaarallisiksi. [1] Tästä syystä C++:ssa näkyy edelleen ominaisuuksia, jotka periytyivät B:stä, CPL:stä ja ALGOLista.

C++:n synnyn syytä kuvaa hyvin Stroustrupin näkemys, että ohjelmointikielellä on kaksi tarkoitusta: ensinnäkin se auttaa ohjelmoijaa määrittämään, mitä ohjelma tekee, ja toiseksi se auttaa ohjelmoijaa ajattelemaan, mitä on mahdollista tehdä. Ensimmäiseen tarvitaan lähellä ongelmaa oleva kieli ja toiseen lähellä tietokonetta oleva kieli. C++:n synnyn tapaa määritti ympäristö, jossa Stroustrup kehitti Luokallisen C:n: Bell Labsissa

suosittiin käytännöllisiä ratkaisuja oikeisiin ongelmiin ja tämä johti siihen, että tehokas olio-ohjelmointikieli oli luonnollisinta rakentaa olemassa olevan kielen päälle ja kehittää sitä jatkuvasti, sen sijaan että olisi aloitettu suunnittelemaan kokonaan uutta kieltä. [1]

Ajan mittaan C++:aan on lisätty tuki muun muassa geneeriselle ohjelmoinnille, sekä muille hyödylliseksi katsotuille ominaisuuksille. Tämä kuvaa C++:n filosofiaa, että kieli on kehitetty käyttöä varten ja se jatkaa kehittymistä lisäämällä uusia, hyödyllisiksi katsottuja ominaisuuksia. [23]

2.2 Funktionaalinen ohjelmointiparadigma

Alonzo Churchin kehittämä lambdakalkyyli perustuu funktioihin [9]. Funktio ajatellaan mustana laatikkona, johon syötetään syötearvoja ja ulostulona tulee uusia arvoja. Koska funktion ulostulo perustuu vain funktion sisääntuloon, ei funktiolla ole sisäistä tilaa, joka vaikuttaisi ulostulon arvoihin, toisin kuin esimerkiksi olio-ohjelmoinnissa. Tästä seuraa myös se, että koska funktiot eivät voi muuttaa tilaansa, eikä niillä ole muitakaan sivuvaikutuksia, on funktiolla oltava ulostulo, jotta sen suorittamisella olisi mitään vaikutusta ohjelman suoritukseen.

Funktionaalista ohjelmointiparadigmaa on esitetty ratkaisuksi muun muassa hajautettuihin ohjelmiin, sillä ohjelma on helpompi jakaa osiin sivuvaikutusten puutteen vuoksi. Tämä mahdollistaa myös kääntäjän tasolla tapahtuvan optimoinnin, mikä on mielenkiintoinen ominaisuus otettaessa huomioon, että Davisin ja Knuthin ja Pardon mukaan ohjelmoinnin alkuaikoina kääntäjiä pidettiin tehottomina ohjelmoijaan verrattuna [10; 11].

2.2.1 Erot imperatiiviseen ohjelmointiin

Hughesin mukaan funktionaalista ohjelmointia verrataan usein imperatiiviseen ohjelmointiin sanomalla, että muuttujien arvoja ei voi muuttaa sen jälkeen, kun ne on kerran asetettu [24]. Tämä johtaa siihen, että muuttujien arvot ovat koko ohjelman suorituksen ajan samat, jolloin niiden laskentajärjestyksellä ei ole merkitystä ja ohjelman suorituksessa voidaan hyödyntää laiskaa laskentaa: muuttujan arvoa ei lasketa muuttujan esittelyn yhteydessä, vaan vasta kun sen arvoa tarvitaan. Knuthin ja Pardon mukaan sijoitusoperaattorin käyttö (C:ssä yhtäsuuruusmerkki ”=”) tekee selvän eron tietojenkäsittelytieteellisen ja matemaattisen ajattelun välille [10]. Sijoitusoperaation puuttuminen funktionaalisesta ajattelusta ja yhtäsuuruusmerkin käytön ollessa lähempänä matemaattista teoriaa, voidaan funktionaalisten kielten ajatella olevan lähempänä matemaattista ajattelua.

Vuonna 1978 John Backus, yksi FORTRAN kielen kehittäjistä, julkaisi Turing-palkinto puheen [25], jossa hän kritisoi imperatiivista paradigmaa ja tarjosi funktionaalista lähestymistapaa ratkaisuna ongelmiin. Tämä lisäsi tutkijoiden mielenkiintoa funktionaalisia kieliä kohtaan sekä johti suoraan Haskellin syntyyn [2].

2.2.2 Haskell

Haskell syntyi komitean yhteistyönä ja sen tarkoituksena oli tarjota yhtenäinen pohja hajanaiselle puhtaan, laiskan funktionaalisen ohjelmoinnin alalle. Komitean jäsenet olivat kehittäneet omia laiskoja funktionaalisia kieliään, mutta vaikka ne muistuttivatkin toisiaan, mikään niistä ei ollut alkanut leviämään laajempaan käyttöön. Komitea halusi perustaa kielensä Mirandaan, joka oli David Turnerin kehittämä funktionaalinen kieli, mutta Turnerin kieltäydyttyä tästä päättivät he tehdä kielen alusta asti itse. Tämä mahdollisti radikaalimpien ratkaisujen, kuten tyyppiluokkien käytön kielen kehityksessä. [2]

Haskellin tarkoitus oli tarjota laiska funktionaalinen kieli opetusta, tutkimusta ja sovellusten kehitystä varten. Kolmen vuoden aikana 1987-1990 Haskell kehittyi komitean alkuperäisistä ajatuksista Haskell 1.0:ksi [2]. Haskellin määräävimpiä ominaisuuksia ovat laiskuus ja puhtaus. Laiska laskenta mahdollistaa rakenteita, jotka eivät olisi muuten mahdollisia, kuten äärettömät listat. Laiskan laskennan toteutus taas vaatii puhdasta kieltä, sillä jotta laiska laskenta toimisi suoritusjärjestyksellä ei saa olla merkitystä, jolloin tarvitaan puhtaita funktioita, joilla ei ole sivuvaikutuksia.

Haskell on pysynyt pitkälti pienen käyttäjäryhmän kielenä, mikä on mahdollistanut uusien, innovatiivisten ja joskus ohjelmia rikkovien ominaisuuksien lisäämisen kieleen. Yksi Haskellin merkittävimpiä innovaatioita ovat tyyppiluokat, jotka määräävät mitä ominaisuuksia muuttujilla on. Tyyppiluokat sopivat myös hyvin yhteen monadien kanssa, joten Haskellin kaksi merkittävintä ominaisuutta tukevat toisiaan. [2]

3. KIELTEN VERTAILU

C++ ja Haskell ovat saman ikäluokan kieliä. Molemmat saivat alkunsa 80-luvulla, C++ kehitys alkoi vuonna 1984 [1] ja Haskellin kehitys 1987 [2]. C++98 standardi julkaistiin 1998, Haskell 98 raportti vuoden 1999 alussa. Päivitykset molempiin määrittelyihin tulivat muutaman vuoden päästä: Haskelliin 2002 ja C++:aan 2003. Molempien kielien seuraavat versiot julkaistiin myös lähekkäin toisiaan, vuonna 2010 tuli Haskell 2010 raportti ja seuraavana vuonna C++11 standardi. Koska kielet on kehitetty samaan aikaan, ovat ne joutuneet tarjoamaan omat vastauksensa samoihin ongelmiin ja kysymyksiin.

Haskellin funktionaalinen ohjelmointiparadigma on osa laajempaa deklaratiivista ohjelmointiparadigmaa. Deklaratiivinen ohjelmointiparadigma eroaa imperatiivisesta ohjelmointiparadigmasta ohjelmien ajattelun ja suunnittelun kannalta. Imperatiivisessa ohjelmointiparadigmassa ohjelmat koostuvat käskyistä, joita seuraamalla ohjelma pääsee haluttuun lopputulokseen. Deklaratiivisessa ohjelmointiparadigmassa ohjelmointi perustuu enemmän ongelman kuvailuun: ohjelmoija kertoo mitä asiat ovat ja kääntäjä päättää miten ne lasketaan. Näin kääntäjän tehtäväksi jää kääntää ohjelmoijan kirjoittama deklaratiivinen ohjelma tietokoneen suoritettavaksi imperatiiviseksi ohjelmaksi.

3.1 Kielten lähtökohdat

Stroustrup kehitti C++:n omaan tarpeeseensa yhdistämällä olemassa olevaan kieleen ominaisuuksia toisesta kielestä. Molemmat kielet, jotka hän yhdisti C++:aan, perustuivat ALGOL 60 -kieleen, mutta niiden suhde ALGOL 60:een oli hyvin erilainen. CPL-BCPL-B-C oli haarautunut ALGOL 60:sta tullakseen lähemmäs tietokonetta: CPL:n kehittäjien mielestä ALGOL ei ottanut tarpeeksi hyvin huomioon tietokoneen rakennetta. BCPL:n kehittäjän mielestä CPL ei ottanut tarpeeksi hyvin huomioon kääntäjien toteuttamisen tarpeita, ja B:n kehittäjän mielestä BCPL vei liikaa tilaa tietokoneelta. C:n kehittäjien mielestä taas B ei sopinut yhteen uusien tietokoneiden sisäisen rakenteen kanssa. Simulan tapauksessa ALGOL 60:n koettiin olevan liian kaukana ongelman kuvauksesta, jolloin järjestelmien simulointi sillä olisi ollut haastavaa ja virhealtista. Stroustrup yhdisti nämä kaksi haaraa C++:ssa yrittäen ottaa huomioon niin ongelmaa ratkaisevan ohjelmoijan tarpeet, kuin ohjelmaa ajavan tietokoneen ominaisuudet ja rajoitukset.

Haskellin tapauksessa kielen pääasiallinen tarkoitus oli tarjota yhteinen ja avoin alusta puhtaan, laiskan funktionaalisen kielen kehittämiselle ja tutkimuskäytölle. Haskellin kehityksessä innovaatioilla ja uuden kokeilulla on ollut suuri rooli ja kielen alkuperäisten kehittäjien ollessa yliopistomaailmasta ei kieltä määrittänyt niinkään yritysten tarpeet, vaan tieteellinen tutkimus ja monet kielen rakenteista perustuvat matemaattiseen teoriaan.

C++:n imperatiivinen luonne on seurausta ALGOLista ja tehokkuusvaatimukset ovat tarkoittaneet, että kielen pitää ottaa jatkuvasti huomioon alusta, jolla sillä tehdyt ohjelmat pyörivät. Toisaalta C++:n ongelmaa lähellä oleva puoli on tuonut kieleen ominaisuuksia imperatiivisen ohjelmoinnin ulkopuolelta, kuten C++11 lisäämät lambdalausekkeet. Haskellin tarkoitus taas on ollut toimia malliesimerkkinä laiskasta funktionaalisesta kielestä, jolloin uusien ominaisuuksien pitää olla yhteensopivia funktionaalisen ajattelun kanssa. Haskell on tosin joissain erikoistapauksissa joutunut sallimaan imperatiivisen ohjelmoinnin ominaisuuksia esimerkiksi laiskan laskennan tilavaatimusten arvioinnin vaikeuden takia [2].

3.2 Kielten kehitys

C++:n kehitystä on edesauttanut suuri käyttäjäryhmä, jonka tarpeisiin yritetään koko ajan vastata aiheuttamatta haittaa muille käyttäjille. Kielen tärkeimpiä teesejä on alusta asti ollut se, ettei käyttäjän tarvitse maksaa ominaisuuksista, joita hän ei käytä. Ominaisuudet, joita ohjelma ei käytä, eivät saa vaikuttaa ohjelman suoritusnopeuteen tai tilavaatimuksiin [1]. Haskellin tapauksessa kielen kehitystä on auttanut pieni, mutta intohimoinen käyttäjäryhmä [2]. Tämä on mahdollistanut kielen kehityksen suuntiin, jotka olisivat olleet mahdottomia, jos kielellä olisi ollut paljon yrityskäyttäjiä, jotka olisivat olleet riippuvaisia kielestä ja joille muutokset kielessä olisivat aiheuttaneet suuria ongelmia.

C++:aan on lisätty kielen kehittyessä muitakin ohjelmointia ja abstrahointia helpottavia ominaisuuksia olio-ohjelmoinnin lisäksi. Esimerkiksi tuki funktionaaliselle ohjelmoinnille lambdojen muodossa lisättiin C++11:ssä. Haskellin tapauksessa merkittävimmät uudet ominaisuudet 2010 versiossa olivat muun muassa tuki hierarkkisille moduulien nimille ja Foreign Function Interface, jonka avulla Haskellissa voidaan hyödyntää muilla ohjelmointikielillä kirjoitettuja ohjelman osia, sekä Haskellilla kirjoitettuja ohjelman osia voidaan hyödyntää muilla ohjelmointikielillä kirjoitetuissa ohjelmissa [26]. FFI oli lisätty Haskell 98 raportin liitteeksi jo vuonna 2003 [2], joten suuresta muutoksesta ei ollut kyse. Kun C++:n lisäykset ovat tuoneet ohjelmoijalle lisää erilaisia tapoja lähestyä ohjelmointia, ovat Haskellin lisäykset olleet paljon konservatiivisempia ja lisänneet vain ominaisuuksia, jotka ovat mahdollisimman riidattomia ja laajasti hyväksyttyjä kielen

kehittäjien keskuudessa [26]. Tästä syystä Haskellin uudistus on jäänyt vähäiseksi, toisin kuin C++:n. Nämä erilaiset lähestymistavat kuvaavat hyvin myös kielten tavoitteita, jotka Haskellin tapauksessa ovat olla de facto puhdas ja laiska funktionaalinen ohjelmointikieli [2] ja C++:n tapauksessa tarjota ohjelmoijille tehokas ohjelmointikieli, joka helpottaisi abstrahointia ja ongelmien ajattelua korkean tason rakenteilla [1].

3.3 Tapausesimerkki: IO

Kummallakin kielellä on mielenkiintoinen suhde syötteen luku- ja kirjoitusoperaatioihin. C++:ssa ei ALGOL-juurien takia ollut määritelty sisäänrakennettua IO:ta, kun taas Haskellissa ongelmaksi muodostui IO-operaatioiden sivuvaikutukset, jotka sotivat funktionaalista ajattelua vastaan. Molemmat kielet ovat ratkaisseet IO:n toteuttamisen omilla tavoillaan ja ovat käyttäneet kielen rakenteita ominaisuuden toteuttamiseen.

C++:n IO on toteutettu iostream-luokkien avulla. Tämä eroaa C:n tavasta, joka oli yleisesti käytössä C++:ssakin ennen iostreamin standardisoimista. Iostream-luokat käyttävät monia C++:n ominaisuuksia, kuten luokkarakennetta, perinnöllisyyttä, template-malleja ja geneeristä ohjelmointia. C++-ohjelman kannalta syöte ja tuloste ovat tavujonoja, joista se joko lukee tai johon se kirjoittaa tavuja. Tämä yhtenäinen näkemys mahdollistaa saman lähestymistavan luetaanpa ohjelman syöte näppäimistöltä, toisen ohjelman tulosteesta tai tietokoneella olevasta tiedostosta. Iostream-luokat abstrahoivat toteutuksen luokkien rajapintojen taakse, jolloin käyttäjälle esitetään yhtenäinen käyttöliittymä riippumatta siitä mistä syötettä luetaan tai mihin tulostetta kirjoitetaan. Myös tilalla on toteutuksessa tärkeä osa, sillä ohjelma pystyy jonon tilaa tarkkailemalla havaitsemaan ongelmia syötteen luvussa ja tulosteen kirjoittamisessa. Lisäksi ohjelmoija voi toteuttaa omia luokkiaan periyttämällä toteutuksensa iostream-luokista, jolloin niiden käyttö ei käyttäjän kannalta eroa standardikirjaston luokista. [27]

Haskellin isoimpia ongelma oli tarjota käyttäjälle IO-operaatiot. Funktionaaliset kielet eivät lähtökohtaisesti hyväksy ohjelman aikaisia syötteitä tai tulosteita, jotka voisivat aiheuttaa sivuvaikutuksia ohjelman toimintaan, joten ohjelman käytön kannalta tärkeän IO:n ja laiskuuden vaatiman puhtauden yhdistäminen oli haastava ongelma. Muissa funktionaalisissa kielissä tämä oli hoidettu imperatiivisilla rakenteilla, mutta Haskellin laiskuus esti tämän toteutuksen. Ratkaisuksi tähän ongelmaan otettiin monadin käsite kategoriateoriasta. IO-monadi ottaa syötteenä maailman tilan ja palauttaa uuden maailman tilan, sekä ulostuloarvon a. Syötettä lukiessa a:n arvo on syöte, tulostetta kirjoittaessa a:n arvo on triviaali tyhjäarvo () tulosteen ollessa uudessa maailman tilassa. Monadit mahdollistivat myös muita hyödyllisiä ominaisuuksia kuten muunneltavan tilan, satunnaislukugeneraattorit ja poikkeukset. [28]

4. YHTEENVETO

C++ ja Haskell ovat vastauksia samaan ongelmaan, mutta ne päätyivät erilaisiin vastauksiin. Sen lisäksi, että toinen edustaa Turingin imperatiivista tietokonetta ja toinen Churchin funktionaalista lambdakalkyylia, kielten syntyvaiheet ja lähtökohdat ovat hyvin erilaiset. Haskell kehitettiin matemaattisten ideoiden pohjalta toimimaan malliesimerkkinä funktionaalisesta ohjelmoinnista, kun taas C++ kehitettiin mahdollisimman tehokkaaksi ja yleishyödylliseksi kieleksi, joka ei vierasta uusia lisäyksiä – imperatiivisia tai funktionaalisia – jos ne mahdollistavat tehokkaamman ohjelmoinnin.

John Backus jakoi ohjelmointikielet kolmeen osaan: yksinkertaisiin malleihin, kuten Turing-koneet ja erilaiset automatat; lambda-laskentaan pohjautuviin kieliin ja von Neumann -arkkitehtuuriin pohjautuviin kieliin, joihin hänen mielestään valtaosa ohjelmointikielistä kuului [25]. John Backus ei innostunut funktionaalisesta ohjelmoinnista sen takia, että se olisi ollut lähempänä ratkaistavia ongelmia, vaan sen takia, koska se tarjosi tietokoneelle erilaisen rakenteen. Hän uskoi von Neumannin pullonkaulan rajoittavan ohjelmia liiaksi, jolloin ohjelmien sarjalliselle suoritukselle tarvittiin vaihtoehto, jonka funktionaalinen ohjelmointi tarjosi. Hän ei tarkoittanut funktionaalisia ohjelmia pyöriväksi koneissa, jotka käyttävät von Neumannin arkkitehtuuria, vaan koneissa, jotka voitaisiin toteuttaa ilman von Neumannin pullonkaulaa [25]. Myöhemmin kävi ilmi, että perinteiset tietokonearkkitehtuurit selviävät funktionaalisista ohjelmista nopeammin, kuin funktionaalisiin ohjelmiin erikoistuneet tietokoneet [2]. Otettaessa lisäksi huomioon Backusin kielten jakojärjestelmä, voitaisiin olio-ohjelmointi tulkita omaksi kielimallikseen, varsinkin kun olio-ohjelmointi on lähempänä ratkaistavia ongelmia kuin ratkaisuja toteuttavaa tietokonetta, minkä mukaan Backus jakoi imperatiiviset kielet von Neumannin kieliin.

Toinen mielenkiintoinen huomio Backusin puheeseen liittyen on, että vaikka funktionaalisten kielten kiinnostus on lisääntynyt, eivät tietokonearkkitehtuurin perusteet ole juurikaan von Neumannin mallista muuttuneet, ja funktionaalisten tietokoneiden suorituskyky on jäänyt von Neumannin arkkitehtuurin perustuvien koneiden jalkoihin [2]. Nykyohjelmissa ongelma ei ole niinkään tietokoneen suorituskyky, vaan ohjelmoijan ohjelmointikyky, mikä on johtanut dynaamisten, tulkittavien kielten nousuun, vaikka niiden suorituskyky olisikin heikompi. Niinpä ongelma, jonka Backus uskoi von Neumannin arkkitehtuurin pullonkaulan synnyttävän, poistui ainakin joksikin aikaa tietokoneiden tehokkuuden lisääntyessä ja ohjelmien uudeksi pullonkaulaksi on tullut ohjelmoijien suorituskyky.

Tämä työ sai alkunsa halusta verrata C++ ja Haskell-kieliä, mutta toinen mahdollinen vertailukohta olisi Haskellin ja ALGOLin välillä. Kumpikin kieli sai alkusysäyksen John Backukselta: ALGOL lähti liikkeelle Backuksen johtamasta FORTRAN-kielestä, kun taas Haskellia innoitti Backuksen FORTRANin ansiosta annetun Turing-palkinnon luento, jossa hän puhui funktionaalisen ohjelmoinnin eduista. Molempien kielien tarkoitus oli yhdistää hajaantunut kielten kehitys ja tämän seurauksena molemmat kielet syntyivät monikansallisen komitean yhteistyönä tapaamisten ollessa molemmin puolin Atlanttia. Molemmat kielet sopivat myös opetustarkoitukseen: ALGOLia käytetiin pitkään de facto algoritmien esittelykielenä, kun taas Haskell toimii hyvin funktionaalisten ominaisuuksien esittely- ja testauskielenä puhtautensa ansiosta. Lisäksi vaikuttaa siltä, että kummankin kielen yleinen käyttö jää vaatimattomaksi verrattuna niiden vaikutukseen muihin ohjelmointikieliin.

LÄHTEET

[1] B. Stroustrup, A history of C++: 1979-1991, The second ACM SIGPLAN conference on history of programming languages, 1993, pp. 271-297.

[2] P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler, A history of Haskell: being lazy with class, ACM, pp. 12-1-12-55.

[3] L. Prechelt, An empirical comparison of seven programming languages, Computer, Vol. 33, No. 10, 2000, pp. 23-29.

[4] R. Harrison, L. Samaraweera, M.R. Dobie, P.H. Lewis, Comparing programming paradigms: an evaluation of functional and object-oriented programs, Software Engineering Journal, Vol. 11, No. 4, 1996, pp. 247-254.

[5] B. Ray, D. Posnett, V. Filkov, P. Devanbu, 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, pp. 155-165.

[6] M. Felleisen, On the expressive power of programming languages, Science of computer programming, Vol. 17, No. 1-3, 1991, pp. 35-75.

[7] K. Kennedy, C. Koelbel, R. Schreiber, Defining and measuring the productivity of programming languages, The International Journal of High Performance Computing Applications, Vol. 18, No. 4, 2004, pp. 441-448.

[8] A.M. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Vol. s2-42, No. 1, 1937, pp. 230-265.

[9] A. Church, An unsolvable problem of elementary number theory, American journal of mathematics, Vol. 58, No. 2, 1936, pp. 345-363.

[10] D.E. KNUTH, L.T. PARDO, The Early Development of Programming Languages, Elsevier Inc, pp. 197-273.

[11] M. Davis, the Universal Computer: The Road from Leibniz to Turing, 3rd ed. CRC Press, 2018, .

[12] A. Perlis, The American side of the development of Algol, ACM SIGPLAN Notices, Vol. 13, No. 8, 1978, pp. 3-14.

[13] J.W. Backus, H. Herrick, I. Ziller, Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN. Preliminary report, Programming Research Group, Applied Science Division, International Business Machines Corporation, New York, 1954, .

[14] O.-. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured programming, London; New York, 1972, .

[15] P. Naur, The European side of the last phase of the development of ALGOL 60, ACM SIGPLAN Notices, Vol. 13, No. 8, 1978, pp. 15-44.

[16] J. Backus, F. Bauer, J. Green, C. Katz, J. McCarthy, A. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. Wegstein, A. van Wijngaarden, M. Woodger, P. Naur, Revised report on the algorithm language ALGOL 60, Communications of the ACM, Vol. 6, No. 1, 1963, pp. 1-17.

[17] R.W. Sebesta, Concepts of programming languages, 8th ed. Addison Wesley, Boston, 2007, .

[18] D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, C. Strachey, The Main Features of CPL, Computer journal, Vol. 6, No. 2, 1963, pp. 134-143.

[19] M. Richards, How BCPL Evolved from CPL, Computer journal, Vol. 56, No. 5, 2013, pp. 664-670.

[20] D. Ritchie, The development of the C language, ACM, pp. 201-208.

[21] O. Dahl, K. Nygaard, SIMULA: an ALGOL-based simulation language, Communications of the ACM, Vol. 9, No. 9, 1966, pp. 671-678.

[22] K. Nygaard, O. Dahl, The development of the SIMULA languages, ACM SIGPLAN Notices, Vol. 13, No. 8, 1978, pp. 245-272.

[23] B. Stroustrup, Evolving a language in and for the real world: C++ 1991-2006, ACM, pp. 4-1-4-59.

[24] J. Hughes, Why Functional programming matters, Addison-Wesley, pp. 17-42.

[25] J. Backus, Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs, Communications of the ACM, Vol. 21, No. 8, 1978, pp. 613-641.

[26] S. Marlow, Haskell 2010 language report, .

[27] S. Prata, C++ primer plus, 6th ed. Addison-Wesley, Upper Saddle River, NJ ;, 2012, .

[28] S.P. JONES, Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, 2010, .