Tampereen yliopisto

Mikko Esko

JULIA-OHJELMOINTIKIELEN OMINAISUUDET JA KÄYTTÖKOHTEET

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Tarkastajat: Tiina Schafeitel-Tähtinen
Elokuu 2021

TIIVISTELMÄ

Mikko Esko: Julia-ohjelmointikielen ominaisuudet ja käyttökohteet
Kandidaatintutkielma
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Elokuu 2021

Tämän tutkielman tarkoituksena on antaa lukijalle yleiskuva Julia-ohjelmointikielen keskeisimmistä ominaisuuksista ja selvittää, miten ne auttavat Juliaa toteuttamaan lupauksen käännettyjä kieliä vastaavasta suorituskyvystä. Tutkielmassa tehtiin katsaus Juliaa koskevaan ja Julialla tehtyyn tutkimukseen. Aineiston avulla selvitettiin, mihin kieltä käytetään ja miten kielen erityisominaisuuksia on hyödynnetty erilaisissa käyttökohteissa.

Julia on vuonna 2012 julkaistu korkean tason dynaaminen ohjelmointikieli, joka pyrkii tarjoamaan matalan tason käännettyjen kielten tasoista suorituskykyä. Optimoitu Julia-koodi voi yltää jopa C-kielen tasoiseen suorituskykyyn. Julian suorituskyvyn taustalla ovat ajonaikainen kääntäminen, tyyppipäättely sekä aggressiivinen optimointi. Kielen käyttäjät ovat raportoineet Julialla kirjoitetun koodin olevan sekä suorituskykyisempää että lyhyempää kuin muilla korkean tason kielillä kirjoitettu koodi. Suorituskyvyn lisäksi Julian vahvuuksia ovat kielen laajennettavuus, intuitiivinen syntaksi sekä laaja standardikirjasto.

Julian erityispiirre on metodivalinnassa käytettävä multiple dispatch, jota voidaan pitää kielen pääasiallisena paradigmana. Multiple dispatch tarkoittaa, että metodivalinta tehdään kaikkien parametrien tyyppien perusteella. Julian monipuolinen tyyppijärjestelmä yhdessä multiple dispatchin kanssa mahdollistaa vahvat abstraktiot, jotka tekevät kielestä joustavan ja laajennettavan. Kielen joustavuutta lisäävät myös laajat metaohjelmointiominaisuudet.

Tutkielman tuloksena todetaan, että Julia on erittäin ilmaisuvoimainen ja suorituskykyinen ohjelmointikieli, joka soveltuu monenlaiseen käyttöön. Kieltä on käytetty paljon erityisesti laskennallisesti vaativissa käyttökohteissa, kuten tieteellisessä laskennassa, matemaattisessa optimoinnissa ja koneoppimisessa. Julia-koodia voidaan suorittaa myös grafiikka- ja tensorisuorittimilla sekä hajautetuilla alustoilla, minkä vuoksi se soveltuu myös suurteholaskentaan. Juliaa käytetään monissa yliopistoissa opetuksessa sekä tutkimuksessa. Julian heikkoutena on vielä toistaiseksi pienehkö valikoima paketteja suositumpiin kieliin verrattuna.

Avainsanat: Julia, ohjelmointikielet, tieteellinen laskenta

SISÄLLYSLUETTELO

1. JOHDANTO

Ohjelmointikieltä valitessa ohjelmoija joutuu tekemään kompromisseja ristiriitaisten vaatimusten vuoksi. Yksi tyypillisimmistä ristiriidoista on ohjelmointikielen käyttömukavuus ja suorituskyky. Korkean tason kielet, kuten Python, MATLAB, Ruby ja R, ovat helppokäyttöisiä ja mahdollistavat matalamman tason kieliä paremman tuottavuuden. Niiden suorituskyky on kuitenkin yleensä huomattavasti huonompi kuin matalamman tason kielten, kuten C ja C++. Vuonna 2012 julkaistu Julia on verrattain uusi korkean tason dynaaminen ohjelmointikieli, jonka tavoitteena on mahdollistaa käännettyjen, matalan tason kielten tasoinen suorituskyky [1]. Julialla voidaan kirjoittaa erittäin korkean abstraktiotason koodia, mutta kieli jättää ohjelmoijalle mahdollisuuden myös matalan tason kontrolliin.

Tässä tutkielmassa tarkastellaan Julia-ohjelmointikieltä ja pyritään selvittämään, mitkä kielen ominaisuudet ovat edesauttaneet sen menestystä. Tutkimalla kielen ominaisuuksia ja käyttökohteita pyritään selvittämään, millaisia etuja kielestä on muihin ohjelmointikieliin verrattuna ja miten Julia kykenee toteuttamaan lupauksen dynaamisesta ohjelmointikielestä, jonka suorituskyky vastaa käännettyjä kieliä. Tutkielmaan sisältyy kirjallisuuskatsaus, jossa selvitetään, mitä Juliasta tiedetään. Mukaan on valittu kielen kehittäjien kirjoittamat Juliaa käsittelevät vertaisarvioidut julkaisut sekä vertaisarvioituja artikkeleja, jotka koskevat Julian suorituskykyä, Julia-paketteja tai Julialla tehtyä tutkimusta. Kielen käyttäjien julkaisemista artikkeleista on otettu mukaan ensisijaisesti ne, joissa on kerrottu, miten Julian ominaisuuksia on hyödynnetty. Aineistoa on täydennetty käytettyjen artikkelien lähdeviitteistä löytyneillä julkaisuilla. Lisäksi on käytetty lähteitä, joissa on selitetty lähdeaineistossa käytettyjä termejä, sekä Julian omilta verkkosivuilta löytyvää dokumentaatiota, josta löytyy ajantasaisin tieto kielen ominaisuuksista.

Luvussa 2 esitellään Julia-ohjelmointikielen tausta ja kerrotaan kielen toiminnasta. Tämän jälkeen luvuissa 3–5 perehdytään tarkemmin Julian keskeisimpiin ominaisuuksiin. Luvussa 6 tarkastellaan aineistosta löytyvien suorituskykytestien tuloksia ja kerrotaan mitkä tekijät mahdollistavat Julian suorituskyvyn. Lopuksi luvussa 7 esitellään Julian kirjastoja ja käyttökohteita sekä kerrotaan, millaisia hyötyjä kielestä on ollut näissä käyttökohteissa. Yhteenvedossa 8 kootaan yhteen tärkeimmät löydökset ja tuodaan esiin mahdollisia jatkotutkimuksen aiheita.

2. KIELEN ESITTELY

Ohjelmointikieltä valitessa ohjelmoija joutuu usein tekemään valinnan korkean tason kielten helppouden ja matalan tason kielten suorituskyvyn välillä. Korkeamman tason kielillä kirjoitettua koodia pidetään helpompana lukea, ymmärtää ja ylläpitää. Ne tarjoavat voimakkaampia abstraktioita, jotka paitsi lyhentävät tarvittavaa ohjelmakoodia, myös helpottavat monimutkaisten ongelmien hahmottamista. Vaikka tietokoneiden laskentatehon nopea kasvu on vähentänyt tarvetta matalan tason kielten käyttämiseen suorituskykysyistä, vaativimmat käyttökohteet, kuten koneoppiminen, tieteellinen laskenta ja matemaattinen mallintaminen, pakottavat usein edelleen ohjelmoijan valitsemaan kielen, joka mahdollistaa matalan tason kontrollin.

Tyypillinen kompromissi suorituskyvyn ja käyttömukavuuden välillä on kutsua matalan tason kielellä kuten C:llä tai Fortranilla kirjoitettuja kirjastoja esimerkiksi Pythonilla kirjoitetusta sovelluskoodista [1]. Tällainen ratkaisu tuo kuitenkin mukanaan omat ongelmansa. Mikäli kirjastokoodia tulee tarve muuttaa tai laajentaa, sovelluspuolen kehittäjä joutuu joko opettelemaan toisen kielen tai delegoimaan tehtävän toiselle kehittäjälle, joka ei välttämättä ymmärrä sovellusalueen tarpeita yhtä hyvin. Kaksikielisten ratkaisujen kokonaisvaltainen optimointi on vaikeaa, ja käytännössä sovelluspuolen kielen hitaus saattaa kirjastokoodin nopeudesta huolimatta aiheuttaa pullonkauloja. Eri kielten välinen kommunikaatio myös monimutkaistaa ja hidastaa ohjelman toimintaa eikä välttämättä salli korkeamman tason kielen hyödyntämistä parhaalla tavalla.

Julia on avoimen lähdekoodin dynaaminen ohjelmointikieli, joka pyrkii ratkaisemaan tämän niin sanotun ”kahden kielen ongelman” [2]. Dynaamisella ohjelmointikielellä tarkoitetaan ohjelmointikieltä, jossa käännösaikaisia operaatioita voidaan suorittaa ajoaikana [3]. Julian tavoitteena on yhdistää dynaamisuuden tuomat käytettävyys- ja tuottavuusedut käännettyjen ohjelmointikielten tasoiseen suorituskykyyn [1, 2, 4, 5]. Julian dynaamisuus perustuu dynaamiseen tyyppijärjestelmään sekä monipuolisiin metaohjelmointi-ominaisuuksiin. Tyyppipäättely ja valinnaiset tyyppiannotaatiot yhdessä ajonaikainen kääntämisen (engl. JIT, Just-In-Time compilation) kanssa mahdollistavat tyyppien hyödyntämisen suoritusaikana. Julian tyyppijärjestelmää käsitellään lisää luvussa 4.

Kieleen on valittu mukaan dynaamisia ominaisuuksia, joita kehittäjät pitivät hyödyllisim-

pinä. Toisaalta tiettyjä optimointeja haittaavia dynaamisia ominaisuuksia on jätetty myös pois. Esimerkkinä tästä tyypit ja ohjelmakoodi ovat Juliassa muuttumattomia (engl. immutable). Uuden ohjelmakoodin generointi ja suoritus sekä käännös- että suoritusaikana on kuitenkin mahdollista. Dynaamisuuden rajoittuneisuuden ansiosta kielen kääntäjä pystyy analysoimaan ja optimoimaan koodia silloinkin, kun koodi sisältää kutsuja käännösaikana tuntemattomiin funktioihin. [1] Esimerkiksi MATLAB ja R ovat Juliaa dynaamisempia, sillä ne sallivat funktioiden uudelleenmäärittelyn sekä näkyvyysalueen tutkimisen ja muokkaamisen [5, s. 19].

Julia luotiin alun perin erityisesti numeerisen ja tieteellisen laskennan tarpeisiin [5]. Numeerisessa ja tieteellisessä laskennassa on tärkeää kyetä kääntämään matemaattisia kaavoja ja malleja tehokkaaksi koodiksi. Aiempien dynaamisten kielten sovittaminen tähän tehtävään on lähes mahdotonta, sillä numeerisen laskennan optimoinnit riippuvat vahvasti kielen implementaation yksityiskohdista. [2] Julia on suunniteltu alusta pitäen suorituskykyä ajatellen.

Julia on toteutettu LLVM-kääntäjäinfrastruktuurin päälle. LLVM on kokoelma työkaluja LLVM-koodin (LLVM intermediate representation, ”LLVM IR”) analysointiin, käsittelyyn ja kääntämiseen [6, 7]. LLVM IR on laitteistosta ja korkeamman tason ohjelmointikielestä riippumaton matalan tason käskykanta [6, 7]. Julia käyttää LLVM:ää optimointien tekemiseen sekä natiivin konekoodin generoimiseen [1]. Kaikki LLVM:ää käyttävät kielet hyötyvät toisistaan, sillä ne voivat hyödyntää samoja LLVM-koodille tehtäviä optimointeja.

Julia-koodia ajettaessa merkkijonomuotoinen koodi jäsennetään ensin syntaksipuuksi (engl. abstract syntax tree, AST). Syntaksipuu muutetaan sitten alemman tason välimuotoon, jota Julian kääntäjä käyttää tyyppipäättelyyn ja optimointiin. Tämän jälkeen koodi käännetään LLVM IR -muotoon, jossa sille tehdään vielä lisää optimointeja LLVM:n tarjoamien työkalujen avulla. Optimointikierrosten jälkeen LLVM kääntää koodin natiiviksi konekoodiksi. Käännöksen tulokset tallennetaan välimuistiin, jotta käännöstä ei tarvitse tehdä uudelleen kun samaa metodia kutsutaan uudestaan. [5, s. 11-12] Eri välivaiheissa generoitua koodia voidaan tarkastella suoritusaikana [8, s. 827]. Tämä onnistuu funktioilla `code_lowered`, `code_typed`, `code_llvm` ja `code_native`. Besard et al. [8] ovat osoittaneet, että Julian kääntäjää voidaan laajentaa tavallisilla Julia-paketeilla ilman että itse kääntäjää tarvitsee muuttaa. Tämä kertoo kielen joustavuudesta.

Käytetyimmistä ohjelmointikielistä (C++, Python, Java) poiketen Julia ei ole luokkapohjainen olio-ohjelmointikieli. Juliassa ei siis ole lainkaan luokkia eikä täten myöskään luokka- tai instanssimetodeja. Sen sijaan sanalla metodi viitataan geneerisen funktion yhteen toteutukseen [5]. Kaikki funktiot ovat Juliassa geneerisiä, ja oikea metodi valitaan annettujen parametrien tyyppien perusteella. Tätä kutsutaan multiple dispatchiksi. Multiple dispatch on Julian keskeinen paradigma, ja siihen perehdytään vielä tarkemmin luvussa 3.

Luokkien puutteesta seuraa myös se, että metodeja ei ole kapseloitu tietotyyppien sisäl-

le. Julian kehittäjien mukaan Javassa turvallisuusominaisuutena pidetty kapselointi on numeerisessa laskennassa taakka, sillä se vaikeuttaa toiminnallisuuden lisäämistä olemassaoleville tietotyypeille [2, s. 85]. Julian kulttuurissa korostuukin koodin uudelleenkäyttö ja yhteistoiminnallisuus eri kirjastojen välillä. On tyypillistä, että kirjastot laajentavat standardikirjaston tai toisten kirjastojen määrittelemiä geneerisiä funktioita omilla metodeillaan. Tästä saattaa toisaalta seurata metodien päällekkäisyyksiä, jolloin joudutaan lisäämään ylimääräisiä määritelmiä yksiselitteisyyden varmistamiseksi [4, s. 76].

Suurin osa Julian standardikirjastosta on kirjoitettu Julialla itsellään. Standardikirjasto on suunniteltu helposti laajennettavaksi. Lisäksi standardikirjasto hyödyntää joitakin avoimen lähdekoodin C- ja Fortran-kirjastoja esimerkiksi lineaarisen algebran toteutuksessa. [2] Myös käyttäjän koodista voidaan helposti kutsua C- ja Fortran-kirjastoja `ccall`-avainsanan avulla. Ulkoisilla paketeilla saavutetaan yhteentoimivuus myös useiden muiden kielten kanssa. Esimerkiksi PyCall.jl-paketti mahdollistaa täyden yhteentoimivuuden Pythonin kanssa.

Julia tukee sekä asynkronista että rinnakkaista suoritusta. Asynkronisuus voidaan toteuttaa vuorottaisrutiinien avulla. Rinnakkaisuuden toteutuksessa käytetään säikeitä. `Task`-tietotyypin avulla ilmaistavat vuorottaisrutiinit voidaan suorittaa myös rinnakkain eri säikeissä. Prosessien välistä kommunikaatiota varten kieli tarjoaa `Channel`-tietotyypin. Lisäksi standardikirjastoon kuuluu `Distributed`-moduuli, josta löytyy työkaluja Julia-funktioiden suorittamiseen etäyhteyden kautta. Julian ekosysteemistä löytyy myös paketteja, joiden avulla Julia-koodia voidaan suorittaa rinnakkain grafiikkaprosessoreilla [8].

Julian asennukseen sisältyy monipuolinen interaktiivinen komentotulkki (engl. Read-Eval-Print-Loop, REPL). Julian komentotulkin kautta voi suorittaa Julia-koodin lisäksi ulkoisia komentoja sekä asentaa paketteja kielen mukana tulevan pakettienhallintajärjestelmän avulla. [9] Kieli sisältää myös useita funktioita ja makroja, jotka helpottavat komentotulkin kautta työskentelyä. Esimerkiksi `@less`-makrolla saa näkyviin kutsuttavan metodin lähdekoodin. Tietylle tyypille määriteltyjä metodeja voi tarkastella `methodswith`-funktiolla. Myös dokumentaation selaaminen onnistuu helpoiten komentotulkissa. Funktioiden, metodien, makrojen, tyyppien sekä kielen avainsanojen dokumentaatiota voi lukea `?`-komennolla. Esimerkiksi `Vector`-tietotyypin dokumentaation saa näkyviin komennolla `?Vector`. Julian komentotulkki on hyödyllinen työkalu kieleen tutustumiseen.

Julian syntaksi muistuttaa muita proseduraalisia kieliä. Koodi ajetaan järjestyksessä rivi kerrallaan kuten esimerkiksi C:ssä ja Pythonissa. Yleisimmät imperatiivisten ohjelmointikielten rakenteet kuten `if`- ja `while`-lauseet toimivat kuten muissakin kielissä. Lisäksi Julia tukee useita funktionaalisista kielistä tuttuja konsepteja kuten korkeamman tason funktioita sekä funktiokompositiota (engl. function composition).

Koodin kirjoittamista ja lukemista helpottavaa syntaksisokeria (engl. syntax sugar) on lai-

nattu muista kielistä. Listojen luomiseen voidaan käyttää Pythonin kaltaista syntaksia (engl. list comprehension). Esimerkki tästä on ohjelman 2.1 ensimmäisellä rivillä luotava lista `a`, joka suorituksen jälkeen sisältää lukujen 1–20 neliöt. Mikäli metodi ottaa ensimmäisenä parametrinaan funktion, sen määritelmän voi halutessaan kirjoittaa Rubya muistuttavalla do–end -syntaksilla metodikutsun jälkeen. Tästä on esimerkkinä ohjelman 2.1 `open`-metodikutsu, jota käytetään tiedoston käsittelyyn. Metodilla on tässä tapauksessa kolme parametria, joista ensimmäinen on funktio, joka käsittelee tiedostoon viittaavaa `IOStream`-objektia `file`. Kaksi muuta parametria kertovat halutun tiedoston nimen ja että tiedostoon halutaan kirjoitusoikeus. Alkiokohtaisten säiliöoperaatioiden suorittamiseen puolestaan käytetään MATLABista tuttua pistesyntaksia. Ohjelmassa 2.1 jokaiseen kaksiuloitteisen matriisin [1 1; 0 1] alkioon lisätään 2, ja laskun tulos (joka on tässä siis [3 3; 2 3]) tallennetaan muuttujaan `m`. Vastaavaa syntaksia voidaan käyttää yksinkertaisten matemaattisten operaattorien lisäksi myös muiden funktioiden kanssa.

```
a = [x * x for x in 1:20]  # list comprehension

# funktio ensimmäisenä parametrina
open("tiedosto.txt", "w") do file
        write(file, "data")
end

m = [1 1; 0 1] .+ 2  # alkiokohtainen yhteenlasku
```

Ohjelma 2.1. *Esimerkkejä Julian syntaksista.*

Matemaattisten kaavojen muuttamista koodiksi helpottaa Julian kattava tuki matemaattisille Unicode-symboleille. Symbolien kirjoittaminen onnistuu Julia-komentotulkissa sekä Juliaa tukevissa editoreissa LaTeX-syntaksilla. Standardikirjasto tukee monia matemaattisia merkintätapoja. Esimerkiksi funktiokutsun `issubset([1,pi], [1,2,pi])` voi vaihtoehtoisesti kirjoittaa `[1,π] ⊆ [1,2,π]`. Unicode-symboleita voidaan käyttää myös omien muuttujien, funktioiden ja operaattorien nimissä.

3. MULTIPLE DISPATCH

Multiple dispatch tarkoittaa suoritettavan metodin valitsemista useamman kuin yhden sille annetun parametrin tyypin perusteella. Kirjallisuudessa samasta asiasta käytetään joskus myös termiä ”multi-methods”. Tässä työssä käytetään termiä ”multiple dispatch”, koska se on yleisempi Juliaan liittyvässä kirjallisuudessa. Multiple dispatch muistuttaa monissa käännetyissä kielissä (esimerkiksi C++) käytettävää kuormitusta. Multiple dispatch tehdään kuitenkin suoritusaikaisten tyyppien perusteella, kun taas kuormitus perustuu käännösaikaisiin tyyppeihin [10]. Näin ollen kuormituksen avulla ei voida valita metodia ajonaikaisesti polymorfisten alityyppien perusteella. Multiple dispatch mahdollistaa siis tarkemman tyyppitiedon hyödyntämisen.

Multiple dispatch ei ole uusi konsepti. Se on peräisin 80-luvulla julkaistusta CommonLoops-järjestelmästä [11], joka oli edelleen käytössä olevan Common Lisp Object Systemin (CLOS) edeltäjä. Vaikka monet ohjelmointikielet ovat sittemmin toteuttaneet multiple dispatchin, useimmat olio-ohjelmointikielet tukevat metodin valintaa vain yhden parametrin tyypin perusteella (engl. single dispatch) [10]. Tällaisissa kielissä ohjelmoijat joutuvat multiple dispatchia tarvittaessa toteuttamaan sen itse, mikä vaatii muokkauksia useisiin eri luokkiin ja monimutkaistaa koodia [12].

Multiple dispatchia pidetään yhtenä mahdollisena ratkaisuna niin sanottuun ekspressiivisyysongelmaan (engl. the expression problem) [12]. Ongelma syntyy kun olemassaolevaa ohjelmaa halutaan laajentaa lisäämällä uusi tyyppi, joka hyödyntää olemassaolevia metodeja, sekä uusi operaatio, joka hyödyntää olemassaolevia tyyppejä, kuitenkaan muokkaamatta alkuperäistä ohjelmakoodia [13]. Olio-ohjelmointikielissä on yleensä helppo lisätä uusi aliluokka, mutta uusien metodien määrittely ei ole mahdollista. Funktionaalisissa kielissä ongelma on käänteinen: uusien funktioiden lisääminen onnistuu helposti, mutta uusien tietotyyppien lisääminen ei. Multiple dispatch sen sijaan mahdollistaa sekä uusien tyyppien että uusien metodien määrittelyn ilman muutoksia alkuperäiseen ohjelmaan tai aiemman koodin uudelleen kääntämistä.

Multiple dispatchin haittapuolena voidaan pitää sen hitautta ja monimutkaisuutta käännöksenaikaiseen metodivalintaan ja single dispatchiin verrattuna. Erikoistuneimman metodin valinta vaatii alityyppirelaation selvittämistä, mikä saattaa olla joissain tilanteissa yllättävän monimutkainen operaatio Julian ominaisuuksien, erityisesti yhdistetyyppien,

vuoksi [14]. Julian kääntäjä pyrkii minimoimaan suoritusaikaisten metodivalintojen tarvetta tyyppipäättelyn ja aggressiivisen optimoinnin avulla. Mikäli metodin määritelmä on tyyppivakaa, kääntäjän on mahdollista optimoida sen sisäiset funktiokutsut kokonaan pois (engl. inlining) käännösaikana [5]. Semanttisesti ohjelma toimii kuitenkin kuin kaikki metodivalinnat olisi tehty suoritusaikaisten tyyppien perusteella [15].

Kokemukset aiemmista tieteelliseen laskentaan suunnatuista kielistä (esimerkiksi R ja Matlab) ovat osoittaneet, että ohjelmoijat laajentavat usein kielen ydinominaisuuksia. Multiple dispatch on Julian tapa mahdollistaa tällaiset laajennukset ja lisätä siten kielen joustavuutta. [5] Koska Julian keskeiset operaatiot on toteutettu metodeina multiple dispatchia hyödyntäen, niille voidaan aina helposti lisätä erikoistuneempia toteutuksia [5, 8]. Tyypillisesti kieltä laajennetaan määrittelemällä uusi alityyppi, jonka perusteella voidaan valita erikoistetumpi metodi halutun operaation suorittamiseen.

Bezanson et al. [15] pitävät teknistä laskentaa multiple dispatchin läpimurtosovelluksena. Matemaattiseen optimointiin tarkoitettu Convex-projekti tukee väittämää [16]. Bezanson perustelee väitettä sillä, että teknisessä laskennassa optimoidun algoritmin valinta on usein niin tärkeää, että verrattain hitaan dynaamisen metodivalinnan tekeminen kannattaa [4, s. 19]. Udell et al. [16] mukaan Julian multiple dispatch sekä yksinkertaistaa että nopeuttaa matemaattisten mallien muodostamista. Lisäksi multiple dispatch erottaa datan sitä käsittelevistä metodeista, mikä mahdollistaa uusien metodien lisäämisen [16].

Muschevici et al. [12] ovat tutkineet multiple dispatchin käyttöä useissa eri kielissä. Bezanson et al. [2] vertaavat multiple dispatchin käyttöä Juliassa näihin aiemmin tutkittuihin kieliin ja toteaa, että Julia hyödyntää multiple dispatchia huomattavasti aiempia järjestelmiä laajemmin. Ero on erityisen suuri kun tarkastellaan vain Julian operaattoreita, joista osalla on useita kymmeniä tai jopa satoja eri metodeja [2]. Ääriesimerkkinä Julian `*`-operaattorilla on kielen uusimman version (1.6.1) standardikirjastossa 328 metodia. Metodeja on erikoistettu esimerkiksi lukuisille eri numero- ja matriisityyppien yhdistelmille.

Julia käyttää symmetristä multiple dispatchia [1, s. 9]. Symmetrisyydellä tarkoitetaan, että kaikkia parametreja pidetään saman arvoisina suoritettavaa metodia valittaessa [1, 17]. Joissakin muissa multiple dispatchia käyttävissä järjestelmissä metodivalinta tehdään vasemmalta oikealle, mikä vähentää epäselvyyksiä metodivalinnassa. Bezanson [4, s. 75] pitää kuitenkin tällaista lähestymistapaa liian mielivaltaisena Julian metodinvalintaan, jossa metodi valitaan funktiolle annettujen argumenttien tyyppien muodostaman monikon (engl. tuple) perusteella. Mahdolliset valinnaiset avainsana-argumentit (engl. keyword arguments) eivät vaikuta metodin valintaan, vaan ne käsitellään vasta metodivalinnan jälkeen.

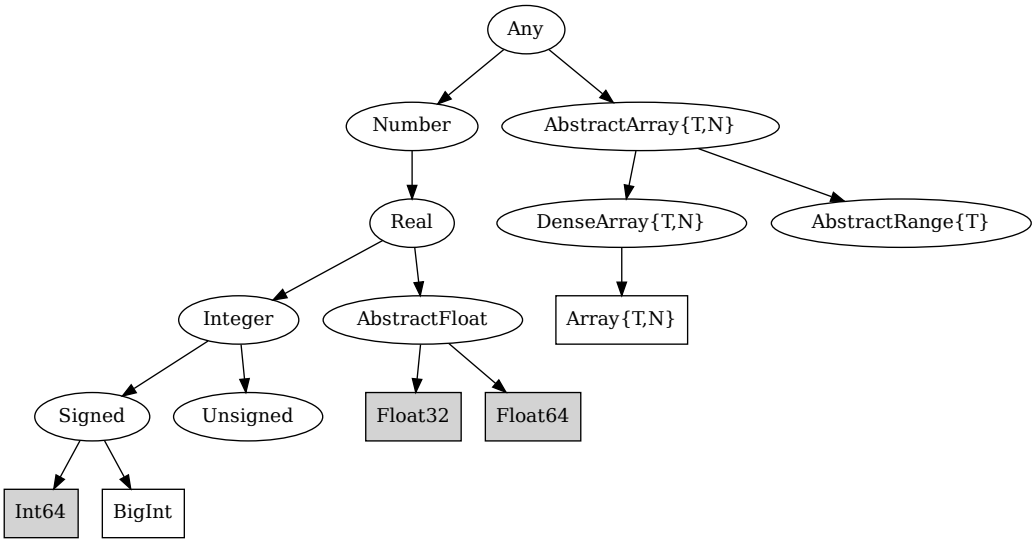
4. TYYPPIJÄRJESTELMÄ

Tyyppien käyttäminen Juliassa on valinnaista. Julian tyypit ovat konseptuaalisesti suoritusaikaisia tyyppejä (engl. run-time type). Suoritusaikaisista tyypeistä puhuttaessa käytetään englanniksi joskus myös termiä ”ilk” [17]. Juliaa on luonnehdittu jopa tyyppijärjestelmättömäksi [14]. Dynaaminen tyyppijärjestelmä ja monipuolinen tyyppi-informaatio ovat kuitenkin edellytyksiä Julian toiminnalle ja erityisesti suorituskyvylle [1]. Mahdollisimman laajan tyyppi-informaation takaamiseksi Julian kääntäjä hyödyntää tyyppipäättelyä [4]. Edellisessä luvussa käsitelty multiple dispatch käyttää tyyppejä ja niiden välisiä alityyppirelaatioita metodivalintaan [14].

Muuttujien ja arvojen tyyppejä on mahdollista määritellä valinnaisilla tyyppiannotaatioilla [1, 5, 14]. Valinnaisuudesta huolimatta Julia-pakettien metodien parametreista valtaosa on eksplisiittisesti tyypitetty [5]. Tyyppiannotaatiot myös antavat kääntäjälle lisää tietoa käsiteltävästä datasta, mikä mahdollistaa optimointeja [1, 5]. Kun metodissa esiintyvät tyypit ja niiden vaatima muisti tiedetään käännösaikana, kääntäjä pystyy usein käyttämään tehokkaampaa pinomuistia (engl. stack memory) keon (engl. heap) sijaan [8]. Tyyppiannotaatiot voivat myös helpottaa virheiden löytämistä ja parantaa koodin luettavuutta. Ne eivät kuitenkaan pysty takaamaan tyyppivirheettömyyttä (engl. soundness), vaan tyyppivirheet havaitaan vasta suoritusaikana. Julia ei tee tyyppikoersiota, vaan tyyppivirheet aiheuttavat aina ajoaikaisen poikkeuksen (engl. run-time exception).

Julian tyyppijärjestelmä on nimellinen [14]. Nimellisessä tyyppijärjestelmässä tyypit ovat samoja vain, jos niillä on sama nimi. Kaksi nimeä lukuun ottamatta määritelmältään identtistä tyyppiä eivät siis ole samoja (tai toistensa alityyppejä) toisin kuin strukturaalisissa tyyppijärjestelmissä. Juliassa ohjelmoijan tulee määritellä eksplisiittisesti halutut tyyppien väliset suhteet.

Julian tyyppihierarkiassa on kahdenlaisia tyyppejä: abstrakteja ja konkreettisia. Näistä abstrakteja tyyppejä käytetään vain tyyppihierarkian osina, esimerkiksi metodien parametrien tyyppien määrittelemiseen. Abstrakteille tyypeille ei siis voi luoda ilmentymiä (engl. instance). Jokaisella konkreettisella tyypillä on vähintään yksi abstrakti ylityyppi (engl. supertype) `Any`, joka on kaikkien muiden tyyppien ylityyppi. Abstrakteista tyypeistä poiketen konkreettisille tyypeille ei voida määritellä alityyppejä. Tyyppihierarkiaa on havainnollistettu kuvassa 4.1.



Kuva 4.1. *Pieni osa Julian tyyppihierarkiasta. Konkreettiset tyypit on merkitty suorakulmioilla. Harmaalla taustalla merkityt tyypit ovat primitiivisiä. Yksinkertaisuuden vuoksi läheskään kaikkia alityyppejä ei ole merkitty näkyviin.*

Tyypit voivat olla myös parametrisia kuten kuvan 4.1 `AbstractArray{T,N}`, joka on Julian n-uloitteisten taulujen ylityyppi. Aaltosulkeiden sisällä olevat muuttujat kuvaavat tyypille annettavia parametreja. Julia-koodissa tyyppi tulisi kirjoittaa `AbstractArray{T,N} where {T,N}`, jotta kääntäjä tietää `T`:n ja `N`:n olevan vapaita muuttujia. Tässä tapauksessa muuttuja `T` kertoo taulun elementin tyypin ja `N` kertoo kuinka moniuloitteinen taulu on. Esimerkiksi `Array{Int64,1}` tarkoittaa siis yksiuloitteista 64-bittisten kokonaislukujen listaa. Parametrisuutta hyödynnetään myös alityyppirelaatioiden ratkaisemisessa, mikä mahdollistaa esimerkiksi metodin erikoistamisen ainoastaan 2-uloitteisille kokonaislukutauluille. Tätä kutsutaan parametriseksi polymorfismiksi.

Primitiiviset ja komposiittityypit ovat konkreettisia tyyppejä. Komposiittityypit muistuttavat C-kielen `struct`-rakennetta. Ne koostuvat yhdestä tai useammasta tietokentästä (engl. field), joista kullekin voi määritellä oman tyypin. Primitiivinen tyyppi on konkreettinen tyyppi, jonka data on ainoastaan tietty vakiomäärä bittejä. Esimerkkejä primitiivisistä tyypeistä ovat lukutyypit `UInt8`, `Int64` ja `Float64` sekä merkkijonojen elementtinä käytettävä 32-bittinen `Char`. Tavallisesti ohjelmoijan ei tarvitse koskaan määritellä itse uusia primitiivisiä tyyppejä. Useimmissa muissa ohjelmointikielissä primitiiviset tyypit on määritelty kielen puolesta eikä uusien lisääminen ole edes mahdollista. [18] Juliassa standardikirjaston ja kääntäjän käyttämät mekanismit ovat kuitenkin yleensä myös kielen käyttäjien ulottuvilla, ja niinpä käyttäjä voi määritellä itse primitiivisiäkin tyyppejä.

Luomalla oman alityypin `AbstractArray`lle ohjelmoija saa pienellä vaivalla käyttöönsä

laajan kokoelman geneerisille tauluille määriteltyjä metodeja. Alityypille täytyy toteuttaa ainoastaan tietyt dokumentaatiossa määritellyt perusoperaatiot kuten indeksointiin liittyvät metodit `setindex!` ja `getindex`. [19] Julian tyyppijärjestelmä ei kuitenkaan tarkista näiden metodien olemassaoloa, vaan rajapinta on täysin muodollinen. Tarvittavien metodien puuttuminen johtaa suoritusaikaisiin virheisiin.

Juliassa ei ole niin sanottua null-tyyppiä [14]. Null-tyypin sijaan arvon puuttumisen ilmaisemiseen käytetään joskus singletoneja. Singleton-tyypit ovat Juliassa komposiittityyppejä, joilla ei ole yhtään tietokenttää [18]. Tavallisimpia singletoneja ovat `nothing`, jota käytetään esimerkiksi funktioiden paluuarvona kun ei haluta palauttaa mitään, sekä `missing`, jota käytetään kuvaamaan puuttuvaa dataa. Puutteellisen mittaus- tai tilastodatan käsittely helpottuu `missing`-singletonin avulla, sillä matemaattiset operaatiot `Missing`-tyypin ja numeroiden välillä on määritelty palauttamaan aina `missing`. Esimerkiksi `[1, missing, 3] .+ [1, 2, missing]` ei johda virheeseen vaan palauttaa `[2, missing, missing]`.

Yksinkertaisten tyyppien lisäksi Julia tukee myös tyyppien yhdisteitä (engl. union) sekä monikkoja (engl. tuple). Esimerkiksi yhdiste `Union{Int32, Float32}` on tyyppi, joka voi olla joko 32-bittinen kokonaisluku tai 32-bittinen liukuluku. Monikkotyyppi kuvaa monesta tyypistä muodostuvaa tyyppiä. Yksinkertainen esimerkki monikkotyypistä on `(1, 'a')`, jonka tyyppi on `Tuple{Int64, Char}`. Monikkotyypit ovat kovariantteja, eli monikkotyyppi on toisen monikkotyypin alityyppi, mikäli kaikki sen argumentit ovat toisen monikon vastaavien argumenttien alityyppejä [14]. Yhdiste- ja monikkotyypit voivat olla myös huomattavasti monimutkaisempia, sillä ne saattavat sisältää rekursiivisesti muita monikoita ja yhdisteitä. Tämän vuoksi alityyppirelaatioiden selvittäminen yleisessä tapauksessa on monimutkaista.

Julia käyttää semanttista alityypitystä (engl. semantic subtyping) [4]. Semanttinen alityypitys tarkoittaa, että alityyppirelaatioita ei määritellä syntaktisten sääntöjen vaan joukko-oppiin perustuvien mallien perusteella [20]. Zappa Nardelli et al. ovat formalisoineet Julian alityypitysrelaation [14]. Alityypityksen yksityiskohdat ovat monimutkaisia eikä niitä käydä tässä työssä läpi. Julian alityypityssääntöjen ratkaiseminen on todennäköisesti laskennallisesti epäkäytännöllistä yleisessä tapauksessa [4]. Käytännössä tyypit ovat kuitenkin oikeissa ohjelmistoissa niin yksinkertaisia, ettei alityypityksen laskennallinen kompleksisuus muodostu ongelmaksi.

5. METAOHJELMOINTI

Metaohjelmoinnilla tarkoitetaan ohjelman mahdollisuutta tutkia omaa toimintaansa ja muokata itseään. Julia sisältää useita metaohjelmointiin liittyviä työkaluja, ja monet kielen kirjastot myös hyödyntävät metaohjelmointiominaisuuksia hyvin laajasti [8, 21, 22]. Julian merkittävimpiä metaohjelmointiominaisuuksia ovat makrot, joilla voidaan muokata koodia ennen sen suorittamista, sekä reflektioon liittyvät metodit. Reflektiolla tarkoitetaan mahdollisuutta tutkia ja muuttaa ohjelman toimintaa sen suorituksen aikana.

Julian makrot ovat Lispin makrojen tavoin syntaktisia eli ne käsittelevät koodia syntaksipuuna. Tämä poikkeaa esimerkiksi C- ja C++-kielten tekstuaalisista makroista, jotka käsittelevät koodia puhtaana tekstinä ennen sen parsimista. Syntaktisten makrojen avulla voidaan hyödyntää kielen omaa parseria, kun halutaan laajentaa kielen syntaksia omilla lisäyksillä. Kaikkia Julian ominaisuuksia voidaan käyttää myös makrojen määritelmien sisällä [23]. Makrojen avulla voidaan luoda toimialuekohtaisia kieliä (engl. domain-specific language, DSL) Julian sisälle [22]. Syntaktisten makrojen rajoituksena on se, että niille annetun koodin täytyy olla validia Julia-koodia. Tämän vuoksi esimerkiksi parittomien sulkeiden käyttö ei ole mahdollista syntaktisilla makroilla luoduissa toimialuekohtaisissa kielissä.

Makrojen erityistapaus on ei-standardit merkkijonoliteraalit (engl. non-standard string literals). Tällaisia makrokutsuja merkitään kirjoittamalla makron nimi välittömästi ennen merkkijonon aloittavaa lainausmerkkiä. Esimerkki tällaisesta makrosta on standardikirjaston `@r_str`, joka muodostaa käännösaikana säännöllisen lausekkeen (engl. regular expression) merkkijonosta kuten `r"a.*z"i`. Merkkijonon sisällä oleva säännöllinen lauseke täsmää merkkijonoihin, jotka alkavat kirjaimella a ja loppuvat kirjaimeen z. Merkkijonon jälkeinen merkkijono `i` annetaan myös parametrina `@r_str`-makrolle. Se kertoo että säännöllisen lausekkeen tulee olla aakkoslajeja erottamaton (engl. case-insensitive).

Toinen esimerkki ei-standardien merkkijonoliteraalien käytöstä on yhteensopivuuden Pythonin kanssa toteuttavan PyCall.jl-paketin `@py_str`-makro. Makro muuttaa ilmauksen `py"range(3,6)"` Python-kutsuksi, ja tuloksena palautetaan vastaava Julian `UnitRange{Int64}`-tyypin instanssi `3:5` (Pythonin `range`-objektit eivät sisällä viimeistä lukua toisin kuin Julian `UnitRange`). Tällaiset makrot vähentävät niin sanotun

boilerplate-koodin tarvetta ja helpottavat siten ohjelmointia.

Reflektion avulla voidaan tarkastella ohjelman toimintaa suoritusaikana. Julia sisältää metodeita esimerkiksi tyyppien sisältämien kenttien tutkimiseen sekä tyyppien välisten relaatioiden selvittämiseen. Lisäksi Juliassa on useiden muiden korkean tason kielten tapaan `eval()` -metodi, jonka avulla voidaan suorittaa mielivaltaista koodia. Monesta muusta kielestä poiketen Julian `eval()` -metodilla ei kuitenkaan ole koskaan pääsyä paikalliseen näkyvyysalueeseen (engl. scope), jotta paikallisten näkyvyysalueiden optimointi olisi mahdollista. [5] Reflektio-ominaisuudet ovat hyödyllisiä esimerkiksi virheenjäljityksessä (engl. debugging) ja koodin analysoinnissa.

6. SUORITUSKYKY

Yksi Julian merkittävimmistä ominaisuuksista on suorituskyky. Useat tutkimukset ovat verranneet Julialla tehtyjen toteutusten nopeutta muihin kieliin [2, 5, 8, 22, 23, 24, 25, 26, 27, 28]. Julian on huomattu olevan pääsääntöisesti selvästi muita dynaamisia kieliä nopeampi. Staattisesti käännettyihin kieliin verrattuna tulokset ovat olleet vaihtelevia. Bezanson et al. tekemissä mikrosuorituskykytesteissä Julian suorituskyvyn on todettu olevan vertailukelpoinen C-kielen kanssa [2]. Lentosimulaatiolla tehdyissä suorituskykytesteissä Julian on huomattu olevan selvästi (noin 10x) hitaampi kuin C++ ja Java lyhyessä testissä, mutta ajettaessa samoja prosesseja useita kertoja Julian ajonaikaisen kääntämisen aiheuttama hidastus amortisoitui, jolloin ero pieneni huomattavasti [25]. Julialla kirjoitettujen grafiikkaprosessorikernelien on havaittu olevan suorituskyvyltään saman tasoisia kuin CUDA C:llä kirjoitetut grafiikkaprosessorikernelit, kun ajonaikaiseen kääntämiseen kuluvaa aikaa ei oteta huomioon [8]. Suorituskykytestien tulosten perusteella ei pystytä lopullisesti ratkaisemaan kielten välisiä nopeuseroja, sillä tulokset vaihtelevat suuresti eri tyyppisissä testeissä. Täysin toisiaan vastaavien toteutusten kirjoittaminen eri kielillä on myös hyvin vaikeaa tai jopa mahdotonta, sillä suorituskyvyltään paras tapa toteuttaa sama algoritmi saattaa olla kielestä riippuen erilainen.

Aiemmissa korkean tason kielissä yksi tärkeimmistä tavoista parantaa suorituskykyä on vektorisaatio. Vektorisaatio tarkoittaa koodin kirjoittamista muotoon, jossa käsiteltävä data on taulukossa, jonka elementit ovat samaa tyyppiä. Tämä tyyppitieto mahdollistaa vektorisoidun koodin paremman suorituskyvyn monissa dynaamisissa kielissä. [2] Esimerkiksi Pythonin Numpy-kirjastoa käytettäessä vektorisoitu koodi voi hyödyntää tehokkaita C-kielellä kirjoitettuja vektorioperaatioiden toteutuksia. Bezanson et al. [2] mukaan vektorisaatio ei kuitenkaan ole luonnollinen tapa kaikkien ongelmien ilmaisemiseen. Vektorisaatio ei myöskään toimi käyttäjän määrittämien tyyppien kanssa vaan se edellyttää kielen tai kirjaston tarjoamien tyyppien käyttämistä [2]. Julian etuna on, että Julia-koodia optimoidessa vektorisaatio ei ole välttämätöntä, vaan tavallisilla for-silmukoilla voidaan päästä samaan suorituskykyyn kielen tyyppijärjestelmän ansiosta. Tästä syystä tehokkaan koodin kirjoittaminen Julialla saattaa olla yksinkertaisempaa kuin muilla dynaamisilla kielillä.

Suorituskyvyn kannalta tärkeitä Julian ominaisuuksia ovat metodien erikoistaminen, devirtualisaatio sekä tyyppipäättely [5]. Devirtualisaatio tarkoittaa metodikutsujen ratkaisemista käännösaikana, jolloin dynaamista metodivalintaa ei tarvitse tehdä, ja metodikutsut

voidaan optimoida jopa kokonaan pois (engl. inlining). Tyyppipäättelyn avulla saatu tyyppitieto mahdollistaa devirtualisaation hyödyntämisen laajemmin.

Bezanson et al. [5] ovat tutkineet devirtualisaation ja tyyppipäättelyn vaikutusta Julian suorituskykyyn ottamalla ne pois käytöstä. Ilman tyyppipäättelyä tehdyissä suorituskykytesteissä suoritusnopeuden havaittiin hidastuvan 5.6x–2151x ja vastaavasti ilman devirtualisaatiota hidastus oli 5.3x–1905x. LLVM:n optimointien poistaminen käytöstä sen sijaan hidasti koodia vain 1.1x–7.1x. [5] Tulokset osoittavat mahdollisimman laajan tyyppitiedon ja suorituskyvyn yhteyden. Juliassa tyyppitietoa on tyyppiannotaatioiden ja tyyppipäättelyn ansiosta tarjolla runsaasti, mikä antaa kääntäjälle mahdollisuuden tehdä optimointeja, jotka eivät monissa muissa dynaamisissa kielissä olisi mahdollisia.

7. KÄYTTÖKOHTEET

Julian kehittäjien tarkoituksena oli luoda ohjelmointikieli, joka soveltuu erityisesti tieteelliseen laskentaan [5]. Tieteellinen laskenta ja siihen liittyvät paketit ovat edelleen suuri osa Julian ekosysteemiä. Saatavilla olevat kirjastot kattavat suurimman osan tieteellisen laskennan tarpeista [29]. Luonnontieteissä usein tarvittavien differentiaaliyhtälöiden muodostamiseen ja ratkaisemiseen on kehitetty DifferentialEquations.jl-paketti [21]. Juliaa käytetään luonnontieteissä esimerkiksi biologisten [30], ekologisten [28] sekä fysikaalisten [27] ongelmien ja ilmiöiden mallintamiseen.

Julia on kuitenkin yleiskäyttöinen ohjelmointikieli ja paketteja löytyy runsaasti myös muihin käyttötarkoituksiin. Niistä ei kuitenkaan ole juurikaan julkaistu vertaisarvioitua tutkimusta. Kirjoittamishetkellä Julian viralliseen pakettirekisteriin on rekisteröity yli 6000 pakettia [31]. Vanhempiin ja kirjoitushetkellä suositumpiin kieliin kuten Pythoniin ja Javaan verrattuna saatavilla olevien pakettien määrä on kuitenkin vaatimaton. Joissakin käyttökohteissa valmiiden pakettien ja kirjastojen puute voi olla syy valita jokin toinen kieli Julian sijaan.

Julian käyttäjät ovat pitäneet kielen etuna muihin kieliin verrattuna erityisesti sen suorituskykyä [23, 32, 33]. Muita käyttäjien mainitsemia etuja ovat yksinkertainen ja intuitiivinen syntaksi [30, 32], kääntäjän laajennettavuus [8, 34] sekä tuottavuus [16, 30, 33, 34, 35]. Seuraavissa aliluvuissa esitellään esimerkkejä käyttökohteista, joissa Julian vahvuudet tulevat esiin.

7.1 Interaktiivinen tietojenkäsittely - Jupyter ja Pluto.jl

Interaktiivinen ohjelmointi on ohjelmointitapa, jossa ohjelmoija muokkaa koodia suoritusaikana. Tällöin ohjelmoija voi hyödyntää välitöntä palautetta siitä mitä kukin metodi tai koodirivi tekee, mikä on erityisen hyödyllistä opeteltaessa uutta ohjelmointikieltä tai ohjelmointikielen kirjastoa. Juliaa on mahdollista käyttää interaktiivisesti joko komentorivin (REPL) tai niin sanottujen notebookkien kautta. Perinteisemmän komentorivipohjaisen työskentelyn rajoitteena on terminaalien tekstipohjaisuus. Notebookit kuten Jupyter ja Pluto.jl toimivat terminaalin sijaan selaimen kautta ja kykenevät näyttämään tekstin lisäksi myös multimediaa ja interaktiivisia komponentteja.

Jupyter on laajasti käytössä oleva notebook-toteutus, joka tukee Julian lisäksi kymmeniä muita eri ohjelmointikieliä. Pluto.jl on uudempi, varta vasten Juliaa varten kehitetty notebook-toteutus. Pluton etuna on reaktiiviset solut, jotka ovat tietoisia välisistä riippuvuuksistaan. Solun sisällön päivittäminen johtaa myös kyseisestä solusta riippuvien solujen päivittämiseen. Lisäksi Pluto-notebookit tallennetaan tavallisina Julia-kooditiedostoina jotka voidaan suorittaa myös suoraan komentoriviltä ilman notebook-ympäristöä. Notebookkeja hyödynnetään paljon opetuksessa sekä tutkimuskäytössä. Ne tarjoavat ilmaisen, avoimen lähdekoodin vaihtoehdon kaupallisille kehitysympäristöille kuten MATLAB.

7.2 Matemaattinen optimointi

JuMP on Julialla toteutettu avoimen lähdekoodin algebrallinen mallinnuskieli (engl. Algebraic Modeling Language, AML) algebrallisten ongelmien ilmaisemiseen matemaattista notaatiota muistuttavassa muodossa. JuMP on huomattavasti aiempia Pythonilla ja MATLABilla toteutettuja avoimen lähdekoodin algebrallisia mallinnuskieliä nopeampi mallien muodostamisessa. JuMP:n suorituskyky mallien muodostamisessa on kilpailukykyinen myös alan huippua edustavien kaupallisten ratkaisujen kanssa. JuMP on ensimmäinen AML, joka mahdollistaa myös käyttäjän määrittelemien funktioiden automaattisen differentioinnin. [22] JuMP:lla muodostettuja malleja voidaan ratkaista useilla eri ratkaisimilla yhtenäisen rajapinnan kautta.

JuMP on toteutettu Juliaan upotettuna toimialuekohtaisena kielenä (engl. Domain-Specific Language, DSL) syntaktisten makrojen avulla. [22] Makrot mahdollistavat mallien ilmaisemisen matemaattista notaatiota muistuttavalla syntaksilla. Convex.jl on toinen matemaattisten mallien ilmaisemiseen tarkoitettu Julia-paketti. JuMP:sta poiketen Convex.jl hyödyntää Julian multiple dispatchia mallien muodostamiseen ja sopivimman ratkaisimen valitsemiseen [16]. Juliaa käytetään matemaattiseen optimointiin useissa eri yliopistoissa sekä tutkimuksessa että opetuksessa [22].

7.3 Koneoppiminen

Koneoppiminen on nopeasti kasvava tieteenala. Monet Julian ominaisuudet kuten standardikirjaston laaja tuki moniuloitteisille matriisityypeille ja niiden operaatioille sopivat hyvin koneoppimissovellusten käyttöön. Kääntäjän laajennettavuuden ja joustavuuden ansiosta tavallista Julia-koodia voidaan suorittaa myös koneoppimissovelluksiin hyvin sopivilla grafiikka- ja tensorisuorittimilla [8, 34]. Juliaa voidaan käyttää Pythonin tavoin ”liimakielenä” alemman tason kielillä kirjoitettujen koneoppimiseen tarkoitettujen ohjelmistokehysten (esimerkiksi TensorFlow ja MXNet) kanssa, mutta Julian edut tulevat paremmin esiin käytettäessä Julialla kirjoitettuja koneoppimispaketteja. Tällaisia paketteja ovat Flux.jl sekä Koçin yliopistossa kehitetty Knet.jl.

Flux on ohjelmistokehys differentioituvaan ohjelmointiin. Flux käyttää koneoppimisessa käytettyjen gradienttien ratkaisemiseen erillisten graafirakenteiden sijaan Julian syntaksipuuta. [34] Tämä mahdollistaa gradienttien laskemisen tavallisille Julia-funktioille, jotka voivat sisältää matemaattisten operaatioiden lisäksi esimerkiksi for-silmukoita, indeksointioperaatioita sekä rekursiivisia funktiokutsuja. Koska tavalliset Julia-funktiot ovat differentioituvia, koneoppimismalleihin voidaan integroida myös Julian standardikirjaston sekä muiden Julia-pakettien toiminnallisuutta [34].

7.4 Supertietokoneet / Suurteholaskenta

Suurteholaskentaa (engl. High-Performance Computing, HPC) tehdään sadoista tai jopa tuhansista prosessoreista koostuvilla supertietokoneilla tai laskentaklustereilla. Julia on suunniteltu tukemaan samanaikaista ja hajautettua laskentaa [2]. Standardikirjaston `Distributed`-moduuli sisältää hajautettuun laskentaan tarvittavan toiminnallisuuden. Ulkoisten pakettien avulla hajautettua laskentaa voidaan tehdä myös heterogeenisilla (eli erilaisista prosessoreista, esimerkiksi grafiikkasuorittimista, koostuvilla) alustoilla [36]. Julian ohjelmointimallin etuna on nopea prototyypitys ja mahdollisuus asteittaiseen optimointiin [36]. Prototyyppejä kirjoittaessa voidaan käyttää `AbstractArray`-tyyppiä, joka on myös hajautettuun laskentaan tarkoitettujen matriisityyppien ylityyppi. Myöhemmin optimointivaiheessa metodien erikoistaminen erilaisille hajautetuille matriisityypeille vaatii vain pieniä muutoksia koodiin [36].

Esimerkki Julian käytöstä oikeiden ongelmien ratkaisemiseen supertietokoneella on Celeste.jl. Celeste on ohjelma, jota käytetään tähtitieteellisen katalogin muodostamiseen kuvantamisdatasta tilastollisten menetelmien avulla. Vuonna 2018 Celeste prosessoi 178 teratavua dataa 650 000-ytimisellä Cori Phase II -supertietokoneella alle 15 minuutissa. Celeste oli ensimmäinen kokonaan Julialla, ja todennäköisesti millään dynaamisella ohjelmointikielellä, kirjoitettu ohjelma, joka kykeni petaflopsin hetkelliseen laskentatehoon. [37] Celeste on osoitus siitä, että korkean tason kielten tarjoamia abstraktioita voidaan hyödyntää myös kaikkein vaativimmassa laskennassa.

Suorituskykytestit ovat osoittaneet Julian olevan varteenotettava vaihtoehto C:lle ja Fortranille suurteholaskennassa [35]. Julian merkittävin etu muihin suurteholaskennassa käytettyihin kieliin nähden on se, että samoihin tuloksiin voidaan päästä huomattavasti pienemmällä määrällä koodia [35, 37]. Celesten tekijöiden arvion mukaan Julialla kirjoitettu koodi on tyypillisesti 5–10 kertaa lyhyempää kuin vastaava C, C++ tai Fortran-koodi [37].

8. YHTEENVETO

Tässä tutkielmassa selvitettiin Julia-ohjelmointikielen keskeisimmät ominaisuudet perehtymällä kieltä koskeviin tieteellisiin artikkeleihin sekä kielen dokumentaatioon. Työssä esiteltiin kielen toimintaa ja ominaisuuksia sekä esimerkkejä kielen käyttökohteista. Julian on tarkoitus olla korkean tason kieli, jolla voidaan kirjoittaa suorituskyvyltään matalan tason kieliä vastaavaa koodia. Aineiston perusteella Julia myös onnistuu tavoitteessaan. Tutkielman tuloksena voidaan todeta, että Julia on erittäin ilmaisuvoimainen ja suorituskykyinen ohjelmointikieli, joka soveltuu monenlaiseen käyttöön.

Yhtenä tutkielman tarkoituksena oli selvittää mihin Juliaa on käytetty ja mitä hyötyjä kielestä on muihin kieliin verrattuna. Selvisi, että kieltä käytetään erityisesti raskasta laskentaa vaativissa käyttökohteissa kuten tieteellisessä laskennassa sekä koneoppimisessa. Julia on käytössä myös opetuksessa useissa yliopistoissa. Kielen käyttäjät ovat raportoineet Julialla kirjoitetun koodin olevan sekä lyhyempää että suorituskykyisempää kuin muilla korkean tason kielillä kirjoitettu koodi. Julian suorituskyvyn mahdollistajia ovat tyyppipäättely, aggressiivinen optimointi, metodien erikoistaminen ja ajonaikainen kääntäminen. Multiple dispatch on Julian erityisominaisuus, joka erottaa sen muista ohjelmointikielistä. Multiple dispatchin ja dynaamisen tyyppijärjestelmän yhdistelmä tekee kielestä joustavan ja helposti laajennettavan. Julia on kuitenkin verrattain uusi kieli eikä sille ole vielä saatavissa yhtä paljon kirjastoja ja paketteja kuin vanhemmille ja suositummille kielille.

Tässä tutkielmassa kieltä tutkittiin enimmäkseen lähdeaineiston avulla. Vaikka joissain lähteissä oli vertailtu eri kielillä tehtyjä toteutuksia, kielten välisiä eroja ei yleensä käsitelty kuin pintapuolisesti esimerkiksi suorituskykytestein. Tämän vuoksi Juliaa voisi jatkossa tutkia myös ohjelmointiprojektin kautta, mikä mahdollistaisi paremmin vertailun muihin kieliin. Lähdeaineistoon perustuvan tutkielman rajoitteena on myös se, että käyttäjien, jotka eivät ole huomanneet kielestä olevan erityistä hyötyä, tulokset eivät välttämättä ole päätyneet julkaistavaksi. Kielen käyttäjien kokemuksia voisi tutkia kattavammin esimerkiksi haastattelu- tai kyselytutkimuksella. Tällainen lähestymistapa voisi mahdollistaa myös eri ohjelmointikielten objektiivisemman vertailun.

Julian käytöstä tieteellisessä laskennassa on saatavilla runsaasti tietoa, mutta kielen soveltuvuutta yleiseen ohjelmointiin ei ole juurikaan tutkittu. Jatkotutkimuksen aiheita ovat Julian soveltuvuus esimerkiksi web-ohjelmointiin tai scriptaukseen. Julia voisi soveltua

myös ohjelmoinnin opetuskieleksi, sillä kieli on syntaksiltaan yksinkertainen, mutta mahdollistaa myös matalan tason konsepteihin tutustumisen.

LÄHTEET

[1] Bezanson, J., Karpinski, S., Shah, V. B. ja Edelman, A. Julia: A Fast Dynamic Language for Technical Computing. *arXiv:1209.5145 [cs]* (syyskuu 2012). arXiv: 1209.5145. URL: http://arxiv.org/abs/1209.5145 (viitattu 14. 05. 2021).

[2] Bezanson, J., Edelman, A., Karpinski, S. ja Shah, V. B. Julia: A Fresh Approach to Numerical Computing. en. *SIAM Review* 59.1 (tammikuu 2017), s. 65–98. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/141000671. URL: https://epubs.siam.org/doi/10.1137/141000671 (viitattu 14. 05. 2021).

[3] *Dynamic programming language - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. en-US. URL: https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_programming_language (viitattu 03. 06. 2021).

[4] Bezanson, J. W. Abstraction in technical computing. eng. Accepted: 2015-11-09T19:50:22Z. Thesis. Massachusetts Institute of Technology, 2015. URL: https://dspace.mit.edu/handle/1721.1/99811 (viitattu 14. 05. 2021).

[5] Bezanson, J., Chen, J., Chung, B., Karpinski, S., Shah, V. B., Vitek, J. ja Zoubritzky, L. Julia: dynamism and performance reconciled by design. en. *Proceedings of the ACM on Programming Languages* 2.OOPSLA (lokakuu 2018), s. 1–23. ISSN: 2475-1421. DOI: 10.1145/3276490. URL: https://dl.acm.org/doi/10.1145/3276490 (viitattu 14. 05. 2021).

[6] Lattner, C. ja Adve, V. LLVM: a compilation framework for lifelong program analysis transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Maaliskuu 2004, s. 75–86. DOI: 10.1109/CGO.2004.1281665.

[7] Lattner, C. *The Architecture of Open Source Applications: LLVM*. URL: http://www.aosabook.org/en/llvm.html (viitattu 07. 06. 2021).

[8] Besard, T., Foket, C. ja De Sutter, B. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30.4 (huhtikuu 2019). Conference Name: IEEE Transactions on Parallel and Distributed Systems, s. 827–841. ISSN: 1558-2183. DOI: 10.1109/TPDS.2018.2872064.

[9] *The Julia REPL · The Julia Language*. URL: https://docs.julialang.org/en/v1/stdlib/REPL/ (viitattu 06. 06. 2021).

[10] Pirkelbauer, P., Solodkyy, Y. ja Stroustrup, B. Open multi-methods for c++. en. *Proceedings of the 6th international conference on Generative programming and component engineering - GPCE '07*. Salzburg, Austria: ACM Press, 2007, s. 123. ISBN: 978-1-59593-855-8. DOI: 10.1145/1289971.1289993. URL:

http://portal.acm.org/citation.cfm?doid=1289971.1289993 (viitattu 16. 05. 2021).

[11] Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. ja Zdybel, F. CommonLoops: merging Lisp and object-oriented programming. *ACM SIGPLAN Notices* 21.11 (kesäkuu 1986), s. 17–29. ISSN: 0362-1340. DOI: 10.1145/960112.28700. URL: http://doi.org/10.1145/960112.28700 (viitattu 09. 06. 2021).

[12] Muschevici, R., Potanin, A., Tempero, E. ja Noble, J. Multiple dispatch in practice. *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. OOPSLA '08. New York, NY, USA: Association for Computing Machinery, lokakuu 2008, s. 563–582. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449808. URL: http://doi.org/10.1145/1449764.1449808 (viitattu 14. 05. 2021).

[13] Wadler, P. *The Expression Problem*. Discussion on Java-genericity mailing list. Marraskuu 1998. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (viitattu 10. 06. 2021).

[14] Zappa Nardelli, F., Belyakova, J., Pelenitsyn, A., Chung, B., Bezanson, J. ja Vitek, J. Julia subtyping: a rational reconstruction. *Proceedings of the ACM on Programming Languages* 2.OOPSLA (lokakuu 2018), 113:1–113:27. DOI: 10.1145/3276483. URL: https://doi.org/10.1145/3276483 (viitattu 14. 05. 2021).

[15] Bezanson, J., Chen, J., Karpinski, S., Shah, V. ja Edelman, A. Array Operators Using Multiple Dispatch: A design methodology for array implementations in dynamic languages. en. *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. Edinburgh United Kingdom: ACM, kesäkuu 2014, s. 56–61. ISBN: 978-1-4503-2937-8. DOI: 10.1145/2627373.2627383. URL: https://dl.acm.org/doi/10.1145/2627373.2627383 (viitattu 14. 05. 2021).

[16] Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S. ja Boyd, S. Convex Optimization in Julia. en. *2014 First Workshop for High Performance Technical Computing in Dynamic Languages*. LA, USA: IEEE, marraskuu 2014, s. 18–28. ISBN: 978-1-4799-7020-9. DOI: 10.1109/HPTCDL.2014.5. URL: http://ieeexplore.ieee.org/document/7069900/ (viitattu 14. 05. 2021).

[17] Allen, E., Hilburn, J., Kilpatrick, S., Luchangco, V., Ryu, S., Chase, D. ja Steele, G. Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance. en. *ACM SIGPLAN Notices* 46.10 (lokakuu 2011), s. 973–992. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2076021.2048140. URL: https://dl.acm.org/doi/10.1145/2076021.2048140 (viitattu 14. 05. 2021).

[18] *Types · The Julia Language*. URL: https://docs.julialang.org/en/v1/manual/types/ (viitattu 16. 07. 2021).

[19] *Interfaces · The Julia Language*. URL: https://docs.julialang.org/en/v1/manual/interfaces/#man-interface-array (viitattu 16. 07. 2021).

[20] Dardha, O., Gorla, D. ja Varacca, D. Semantic Subtyping for Objects and Classes. en. *Formal Techniques for Distributed Systems*. Toim. D. Beyer ja M. Boreale. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, s. 66–82. ISBN: 978-3-642-38592-6. DOI: 10.1007/978-3-642-38592-6_6.

[21] Rackauckas, C. ja Nie, Q. DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. en. *Journal of Open Research Software* 5.1 (toukokuu 2017). Number: 1 Publisher: Ubiquity Press, s. 15. ISSN: 2049-9647. DOI: 10.5334/jors.151. URL: http://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/ (viitattu 14. 05. 2021).

[22] Dunning, I., Huchette, J. ja Lubin, M. JuMP: A Modeling Language for Mathematical Optimization. en. *SIAM Review* 59.2 (tammikuu 2017), s. 295–320. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/15M1020575. URL: https://epubs.siam.org/doi/10.1137/15M1020575 (viitattu 14. 05. 2021).

[23] Lubin, M. ja Dunning, I. Computing in Operations Research Using Julia. eng. *INFORMS journal on computing* 27.2 (2015). Publisher: INFORMS, s. 238–248. ISSN: 1091-9856. DOI: 10.1287/ijoc.2014.0623.

[24] Greener, J. G., Selvaraj, J. ja Ward, B. J. BioStructures.jl: read, write and manipulate macromolecular structures in Julia. en. *Bioinformatics* 36.14 (heinäkuu 2020). Toim. A. Elofsson, s. 4206–4207. ISSN: 1367-4803, 1460-2059. DOI: 10.1093/bioinformatics/btaa502. URL: https://academic.oup.com/bioinformatics/article/36/14/4206/5837108 (viitattu 20. 07. 2021).

[25] Sells, R. Julia Programming Language Benchmark Using a Flight Simulation. *2020 IEEE Aerospace Conference*. ISSN: 1095-323X. Maaliskuu 2020, s. 1–8. DOI: 10.1109/AERO47225.2020.9172277.

[26] Fathurrahman, F., Agusta, M. K., Saputro, A. G. ja Dipojono, H. K. PWDFT.jl: A Julia package for electronic structure calculation using density functional theory and plane wave basis. en. *Computer Physics Communications* 256 (marraskuu 2020), s. 107372. ISSN: 00104655. DOI: 10.1016/j.cpc.2020.107372. URL: https://linkinghub.elsevier.com/retrieve/pii/S0010465520301600 (viitattu 14. 05. 2021).

[27] Sinaie, S., Nguyen, V. P., Nguyen, C. T. ja Bordas, S. Programming the material point method in Julia. en. *Advances in Engineering Software* 105 (maaliskuu 2017), s. 17–29. ISSN: 09659978. DOI: 10.1016/j.advengsoft.2017.01.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S0965997816302769 (viitattu 14. 05. 2021).

[28] Hall, K. R., Anantharaman, R., Landau, V. A., Clark, M., Dickson, B. G., Jones, A., Platt, J., Edelman, A. ja Shah, V. B. Circuitscape in Julia: Empowering Dynamic Approaches to Connectivity Assessment. en. *Land* 10.3 (maaliskuu 2021). Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, s. 301. DOI: 10.3390/

land10030301. URL: https://www.mdpi.com/2073-445X/10/3/301 (viitattu 14. 05. 2021).

[29] Kulyabov, D. S. ja Korol'kova, A. V. Computer Algebra in JULIA. en. *Programming and Computer Software* 47.2 (maaliskuu 2021), s. 133–138. ISSN: 0361-7688, 1608-3261. DOI: 10.1134/S0361768821020079. URL: https://link.springer.com/10.1134/S0361768821020079 (viitattu 14. 05. 2021).

[30] Landeros, A., Stutz, T., Keys, K. L., Alekseyenko, A., Sinsheimer, J. S., Lange, K. ja Sehl, M. E. BioSimulator.jl: Stochastic simulation in Julia. en. *Computer Methods and Programs in Biomedicine* 167 (joulukuu 2018), s. 23–35. ISSN: 0169-2607. DOI: 10.1016/j.cmpb.2018.09.009. URL: https://www.sciencedirect.com/science/article/pii/S0169260718301822 (viitattu 14. 05. 2021).

[31] *GitHub - JuliaRegistries/General: The official registry of general Julia packages*. en. URL: https://github.com/JuliaRegistries/General (viitattu 23. 07. 2021).

[32] Sukhodolov, A., Sorokina, P. ja Fedotov, A. Numerical analysis of ecology-economic model for forest fire fighting in Baikal region. *Discrete and Continuous Models and Applied Computational Science* 27 (joulukuu 2019), s. 154–164. DOI: 10.22363/2658-4670-2019-27-2-154-164.

[33] Weibezahn, J. ja Kendziorski, M. Illustrating the Benefits of Openness: A Large-Scale Spatial Economic Dispatch Model Using the Julia Language. en. *Energies* 12.6 (maaliskuu 2019), s. 1153. ISSN: 1996-1073. DOI: 10.3390/en12061153. URL: https://www.mdpi.com/1996-1073/12/6/1153 (viitattu 18. 05. 2021).

[34] Innes, M., Saba, E., Fischer, K., Gandhi, D., Rudilosso, M. C., Joy, N. M., Karmali, T., Pal, A. ja Shah, V. Fashionable Modelling with Flux. *arXiv:1811.01457 [cs]* (marraskuu 2018). arXiv: 1811.01457. URL: http://arxiv.org/abs/1811.01457 (viitattu 07. 06. 2021).

[35] Hunold, S. ja Steiner, S. Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC?: *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Marraskuu 2020, s. 20–25. DOI: 10.1109/PMBS51919.2020.00008.

[36] Besard, T., Churavy, V., Edelman, A. ja Sutter, B. D. Rapid software prototyping for heterogeneous and distributed platforms. en. *Advances in Engineering Software* 132 (kesäkuu 2019), s. 29–46. ISSN: 0965-9978. DOI: 10.1016/j.advengsoft.2019.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0965997818310123 (viitattu 14. 05. 2021).

[37] Regier, J., Pamnany, K., Fischer, K., Noack, A., Lam, M., Revels, J., Howard, S., Giordano, R., Schlegel, D., McAuliffe, J., Thomas, R. ja Prabhat, . Cataloging the Visible Universe Through Bayesian Inference at Petascale. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. ISSN: 1530-2075. Toukokuu 2018, s. 44–53. DOI: 10.1109/IPDPS.2018.00015.