Tampereen yliopisto

Pauli Jaakkola

TULKIN ASTEITTAINEN MUUTTAMINEN KÄÄNTÄJÄKSI

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Maaliskuu 2019

TIIVISTELMÄ

Pauli Jaakkola: Tulkin asteittainen muuttaminen kääntäjäksi
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikka
Maaliskuu 2019

Tässä opinnäytetyössä tutkitaan tulkin muuttamista kääntäjäksi kasvattamalla käännöksen osuutta käännösvaihe kerrallaan. Tämä on vaihtoehto kääntäjän kirjoittamiselle suoraan tai tulkin muuttamiselle kääntäjäksi osittaisevaluaattorilla. Menetelmän toimivuus todennetaan muuttamalla puhtaasti tulkattu pienehkön funktionaalisen ohjelmointikielen toteutus kääntäjän ja virtuaalikoneen yhdistelmäksi. Havaitaan välikielten tulkkien antavan huomattavasti tukea toimivan kääntäjän kirjoittamiseen.

Avainsanat: ohjelmointikielet, tulkit, kääntäjät, C*K-tilakoneet, CPS

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

AST	Abstrakti syntaksipuu (**A**bstract **S**yntax **T**ree) on puumuotoinen väliesitys, joka ei sisällä semanttisesti merkityksettömiä yksityiskohtia kuten sulkuja.
CEK	Tapa esittää ohjelmointikielen operatiivinen semantiikka tilakoneena, jonka tila koostuu ohjelmafragmentista (**C**ontrol), muuttujan arvot sisältävästä ympäristöstä (**E**nvironment) ja kontekstista tai jatkumasta (**K**ontinuation).
CESK	CEK-tilakoneen laajennus, jossa mukana on myös muisti (**S**tore) imperatiivisten operaatioiden tai rekursiivisen sidonnan määrittelemiseksi.
CPS	**C**ontinuation **P**assing **S**tyle; väliesitys, jossa operaatioiden argumenttien on oltava(jonkin määritelmän mukaan) triviaaleja eli hyvin yksinkertaisia ja jossa operaatiota seuraavat operaatiot ovat operaatiolle annettavassa jatkumasssa.
Esijärjestys	Sellainen puun läpikäynti, jossa ensin käsitellään solmu itse ja sitten sen lapset.
Jatkuma	Funktio, jonka kutsuminen vastaa lopun ohjelman suoritusta. Jatkuman ollessa funktion parametri sen tarkoitus on sama kuin matalammalla tasolla funktion paluuosoitteella ja kutsujan aktivaatiotietueella.
Jälkijärjestys	Sellainen puun läpikäynti, jossa ensin käsitellään solmun lapset ja vasta sitten solmu itse.
Jäsentäminen	Tekstin lukeminen muistiin tietorakenteeksi, esimerkiksi lähdekoodista syntaksipuuksi.
Madaltaminen	Ohjelman muuttaminen eksplisiittisempään 'matalamman tason' muotoon.
Monikko	Monikko eli 'tuple' on parin yleistys eli tietue, jonka kentillä on nimien sijaan numerot. Yleisesti käytetty muun muassa funktionaalisissa ohjelmointikielissä, relationaalisissa tietokannoisssa ja Pythonissa.
Osittaisevaluointi	Ohjelman suorittaminen niiltä osin kuin se on käännösaikana tiedossa olevien syötteiden valossa mahdollista. Tuottaa residuaaliohjelman, joka käyttäytyy muuten kuten alkuperäinen ohjelma mutta ei tarvitse käännösaikaisia syötteitä ja on tehokkaampi.

Peritty attribuutti	Ominaisuus, jonka puun solmun lapset perivät (esi)vanhemmaltaan. Usein toteutettu käytännössä puun läpikäyntifunktion lisäparametrina, joka välitetään läpikäynnin rekursiivisissa kutsuissa.
Peruslohko	Ohjelmalauseiden jono, jossa vain ensimmäiseen lauseeseen kohdistuu hyppyjä ja josta lähtee hyppyjä vain viimeisestä lauseesta.
RISC	**R**educed **I**nstruction **S**et **C**omputing; tietokonearkkitehtuurityyli, jossa kukin käsky sisältää vain yhden operaation.
SSA	**S**ingle **S**tatic **A**ssignment; väliesitys, jossa muuttujat alustetaan aina määritelmänsä yhteydessä eikä niihin voi sijoittaa ja jossa datavuon yhtyminen esitetään eksplisiittisesti ϕ-funktioilla tai peruslohkojen parametreina.
Sulkeuma	Sulkeuma on ensiluokkaisen funktion ajonaikainen ilmentymä, jossa on funktion koodin lisäksi mukana myös sen vapaiden muuttujien arvot.
Tavukoodi	Virtuaalikonemuotoisen tulkin käskykanta. Nimi tulee siitä, että monissa virtuaalikoneissa käskyn operaatio on tavun kokoinen.
Välikieli, väliesitys	Ohjelman olomuoto jossain muussa käännösprosessin vaiheessa kuin alussa lähdekoodina tai lopussa kohdekielisenä koodina.

1 JOHDANTO

Ohjelmointikielten tarkoitus on tietorakenteiden ja algoritmien kuvaaminen sekä yleensä myös niiden automaattinen suoritus. Suoritukseen tarvitaan ohjelmointikielen toteutus eli tulkki, kääntäjä tai jokin näiden yhdistelmä. Tyypillisten ohjelmointikielten yleiskäyttöisyys, korkea abstraktiotaso ja suorituskykyvaatimukset tekevät niiden toteuttamisesta haastavaa. Erityisen vaikeaa on uuden ohjelmointikielen ensimmäisen toteutuksen luominen, koska saatavilla on ainoastaan kielen mahdollisesti keskeneräinen tai tulkinnanvarainen määritelmä.

Tulkin toteuttaminen operatiivisen semantiikan pohjalta korkean tason ohjelmointikielellä on yleensä melko suoraviivaista. Kääntäjän toteuttaminen on vaikeampaa, koska semantiikkaa ei toteuteta suoraan vaan muuntamalla ohjelma toiselle kielelle, jota osataan jo suorittaa. Kääntämällä saavutetaan kuitenkin helpommin suorituskykyinen tai muuten haluttu kohdeympäristö. Niinpä varsinkin uutta ohjelmointikieltä toteutettaessa olisi hyödyllistä pystyä muuttamaan tulkki kääntäjäksi. Osittaisevaluaattorin ja Futamuran projektioiden avulla tämä voidaan tehdä täysin automaattisesti. Tässä työssä tutkitaan suoraviivaisempaa menetelmää, joka toisaalta vaatii paljon enemmän käsityötä.

Käytetyn menetelmän lähtökohtana on yksinkertainen tulkki. Vähitellen tulkista ja sen toteuttamasta kielestä poistetaan ominaisuuksia ja kirjoitetaan käännösvaihe edellisestä kielen versiosta uuteen yksinkertaistettuun kieleen. Välikielten tulkkeja voidaan hyödyntää käännösvaiheiden suunnittelussa ja testauksessa. Työn loppuun mennessä syntyneistä käännösvaiheista on muodostunut tavanomainen käännösputki ja tulkista on jäljellä enää suoraviivainen virtuaalikone.

Aluksi käsitellään ohjelmointikielten määrittelyn ja toteutuksen teoreettista taustaa olennaisilta osin. Sitten määritellään työssä tulkattavan ja käännettävän Complot-kielen semantiikka. Näiden pohjatietojen jälkeen esitellään kielen tulkki, kohdevirtuaalikone sekä muutoksen vaiheet. Tässä osassa kiinnitetään erityistä huomiota tulkkien hyödyntämismahdollisuuksiin käännösvaiheiden suunnittelussa. Lopuksi tehdään yhteenveto ja esitellään työn aikana ilmenneitä jatkokysymyksiä.

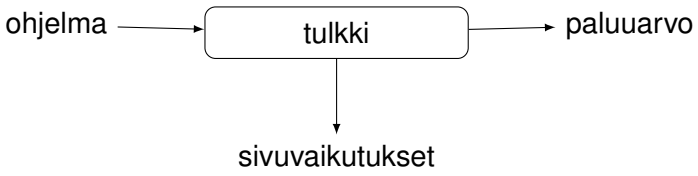
2 OHJELMOINTIKIELTEN TOTEUTUS

Ohjelmointikieliä käytetään kuvaamaan tietorakenteita, algoritmeja ja prosesseja yksikäsitteisesti jollain tarkoituksenmukaisella abstraktiotasolla. Käytännössä ohjelmointikielillä kirjoitettuja ohjelmia myös suoritetaan tietokoneella.

Ohjelman suorittaminen tietokoneella voidaan mahdollistaa kahdella tavalla: suorittamalla suoraan eli **tulkkaamalla** tietokoneen prosessorilla tai tulkkiohjelmalla tai **kääntämällä** tulkattavaan muotoon kääntäjällä[15, s. 13]. Kääntäminen tarkoittaa siis ohjelmointikielten yhteydessä samaa kuin luonnollisten kielten, mutta tulkkaaminen ei.

2.1 Tulkki

Kielen L **tulkki** on funktio, joka ottaa syötteenään L-kielisen ohjelman P_L ja suorittaa sen eli palauttaa sen mitä P_L:n kuuluu palauttaa ja aiheuttaa ne sivuvaikutukset, jotka P_L:n kuuluu aiheuttaa (kuva 2.1)[15, s. 14].

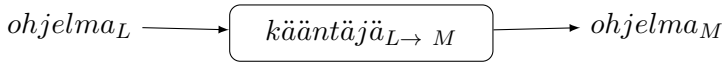


Kuva 2.1. *Tulkki*

Useimpien kielten tapauksessa tulkki on toinen ohjelma. Myös tietokoneen prosessori on tulkki, joka suorittaa omaa konekieltään[15, s. 6–9].

2.2 Kääntäjä

Kielen L **kääntäjä** on funktio, joka ottaa syötteenään L-kielisen ohjelman P_L ja tuottaa M-kielisen ohjelman P_M, joka käyttäytyy suoritettaessa samalla tavalla kuin P_L käyttäytyisi (kuva 2.2)[15, s. 15].



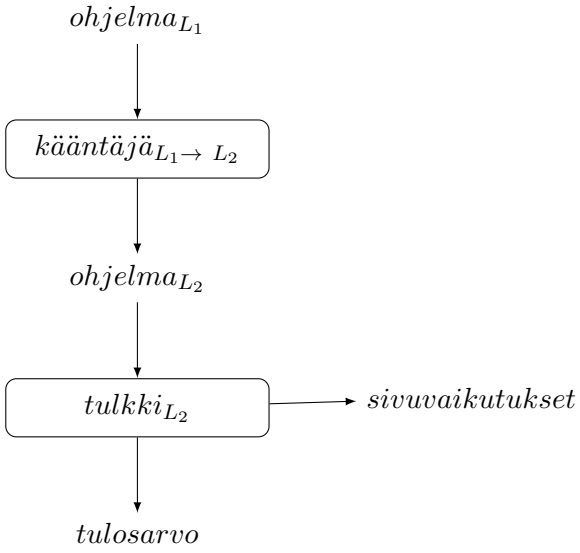
Kuva 2.2. *Kääntäjä*

Perinteisimmillään kääntäjä tuottaa konekielisen ohjelman, joka voidaan suorittaa suo-

raan tietokoneen prosessorilla. Kohdekieli voi kuitenkin olla mikä tahansa [15, s. 41], esimerkiksi virtuaalikoneen käskykanta (’tavukoodi’) tai jopa sama kuin lähtökieli kuten Javascriptin pakkaamisessa (’minification’).

2.3 Kääntäjän ja tulkin yhdistelmä

Yleisessä tapauksessa ohjelmointikielen toteutus sisältää sekä kääntäjän että tulkin. Tällöin kääntäjä muuntaa ohjelman tulkille sopivampaan muotoon ja tulkki suorittaa sen tässä muodossa (kuva 2.3). [15, s. 17]



Kuva 2.3. *Tulkki ∘ kääntäjä*

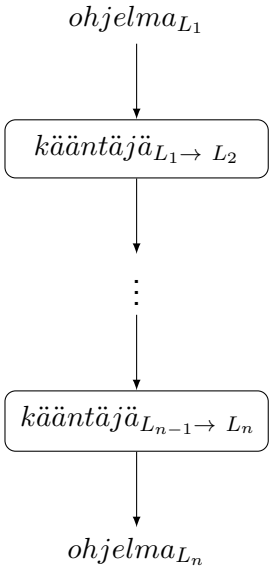
Esimerkiksi Python-kielen alkuperäinen toteutus CPython [42] sisältää lähdekoodia tavukoodiksi muuntavan kääntäjän ja tuota tavukoodia suorittavan tulkin. CPythonin tapauksessa kääntäjä ja tulkki ovat saman ohjelman osia eikä tavukoodia aina edes kirjoiteta välissä tiedostoon.

Voidaan ajatella, että pelkän kääntäjän sisältävissä kielitoteutuksissa tulkkikomponentti on vain ulkoistettu jonkin toisen kielen toteutukselle. Yksinkertaisetkin tulkit taas yleensä suorittavat jonkinlaisen käännöksen niiden muuttaessa lähdekoodin syntaksipuuksi ennen sen suorittamista[15, s. 17]. Minkäänlaista väliesitystä (luku 2.4) rakentamattomat tulkit ovat harvinaisia, koska silmukoiden ja funktioiden toistuva jäsentäminen on hidasta. Lisäksi jäsentämisen erottaminen tulkkaamisesta on parempaa työnjakoa ja tekee tulkin koodista helpommin ymmärrettävää. [31, luku 9]

2.4 Käännösvaiheet

Käännösprosessi jaetaan usein vaiheisiin, joista kukin on ikään kuin oma osakääntäjänsä (kuva 2.4). Kuvan kieliä L_2-L_{n-1} kutsutaan **välikieliksi** (Intermediate Language, IL). Koska välikielet ovat yleensä merkki- tai käskyjonojen sijaan helpommin analysoitavia ja

muokattavia puita tai yleisempiä graafeja, usein käytetään mieluummin nimitystä **väliesitys** (Intermediate Representation, IR). [2, s. 136]



Kuva 2.4. *Käännösvaiheet*

Käännösvaiheen tarkoitus on yleensä joko tarkistaa, että ohjelma on hyväksyttävä esimerkiksi muuttujien näkyvyysalueiden suhteen, madaltaa ohjelmaa eli muuttaa se väliesitykseen, joka on lähempänä lopullista tavoitekieltä tai optimoida[1] sitä esimerkiksi vähentämällä sen suoritusaikaa, muistin käyttöä tai koodin määrää. Jotkin käännösvaiheet kuten rekisterien varaaminen (luku 5.3.2) sekä madaltavat että optimoivat ohjelmaa. Väliesityksen vaihdokset ajoittuvat yleensä madaltaviin käännösvaiheisiin.

[1] Yleensä ohjelmaa ei kuitenkaan saada optimaaliseen muotoon vaan sen ominaisuudet vain paranevat jonkin verran.

3 COMPLOTIN OPERATIIVINEN SEMANTIIKKA

Tässä luvussa esitellään työssä tulkattavan ja käännettävän Complot-ohjelmointikielen semantiikka. Complot on vahvasti dynaamisesti tyypitetty funktionaalinen ohjelmointikieli. Sen syntaksi (liite A) on lähellä C-sukuisia kieliä ja semantiikka Schemeä [37]. Semantiikka määritellään tässä operatiivisesti CESK-tilakoneena [14, s. 165– 167]. Operatiivinen semantiikka valittiin denotatiivisen tai aksiomaattisen sijaan, koska se on lähempänä konkreettista tulkkia. CESK-tilakone taas valittiin, koska jatkumia käyttävänä se sopii hyvin yhteen CPS:n (luku 5.2) kanssa ja optimoi lisäksi häntäkutsut. Häntäkutsujen optimointi on välttämätöntä, koska yksinkertaisuuden vuoksi Complotissa ei ole lainkaan erillisiä silmukkarakenteita. CESK-tilakoneen **S**torea eli muistia tarvitaan tässä semantiikassa vain lohkojen rekursiivisen sidonnan mallintamiseen.

$$
\begin{aligned}
(\llbracket \texttt{\{}s\ \overline{s^*}\ e\texttt{\}} \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) &\longrightarrow (s, \mathcal{E}'_l, \mathcal{E}'_d, \sigma'', \texttt{block}(\kappa, \mathcal{E}'_l, \mathcal{E}'_d, [\overline{s^*}], e)) \\
&\quad \texttt{where}\ (x^*_l, x^*_d) = defs([s, \overline{s^*}]) \\
&\qquad (a^*_l, \sigma') = alloc(\sigma, \|x^*_l\|);\ (a^*_d, \sigma'') = alloc(\sigma', \|x^*_d\|) \\
&\qquad \mathcal{E}'_l = \mathcal{E}_l[\overline{x^*_l \mapsto a^*_l}];\ \mathcal{E}'_d = \mathcal{E}_d[\overline{x^*_d \mapsto a^*_d}] \\
&\quad (\llbracket \texttt{\{}e\texttt{\}} \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) \longrightarrow (e, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) \\
(\_, \_, \_, \sigma, \texttt{block}(\kappa, \mathcal{E}_l, \mathcal{E}_d, s :: s^*, e)) &\longrightarrow (s, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{block}(\kappa, \mathcal{E}_l, \mathcal{E}_d, s^*, e)) \\
(\_, \_, \_, \sigma, \texttt{block}(\kappa, \mathcal{E}_l, \mathcal{E}_d, [], e)) &\longrightarrow (e, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa)
\end{aligned}
$$

$$
\begin{aligned}
(\llbracket x = e \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) &\longrightarrow (e, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{def}(\kappa, \mathcal{E}_l, \mathcal{E}_d, x)) \\
(v, \_, \_, \sigma, \texttt{def}(\kappa, \mathcal{E}_l, \mathcal{E}_d, x_l)) &\longrightarrow (\_, \mathcal{E}_l, \mathcal{E}_d, \sigma[\mathcal{E}_l[x_l] := v], \kappa) \\
(v, \_, \_, \sigma, \texttt{def}(\kappa, \mathcal{E}_l, \mathcal{E}_d, x_d)) &\longrightarrow (\_, \mathcal{E}_l, \mathcal{E}_d, \sigma[\mathcal{E}_l[x_d] := v], \kappa) \\
(\llbracket x_l \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) &\longrightarrow (\sigma[\mathcal{E}_l[x_l]], \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) \\
(\llbracket x_d \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) &\longrightarrow (\sigma[\mathcal{E}_d[x_d]], \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa)
\end{aligned}
$$

Kuva 3.1. *Complotin lohkojen ja muuttujien semantiikka*

Complotin CESK-tilakoneen siirtymärelaatio on kuvien 3.1 ja 3.2 sääntöjen refleksiivinen transitiivinen sulkeuma [14, s. 10] eli sääntöjä sovelletaan, kunnes ohjelmafragmentti on pelkistynyt vakioarvoksi tai jäänyt jumiin ('stuck') eli ikuiseen silmukkaan, jossa tila ei koskaan vaihdu.

Kuvassa 3.1 on esitetty Complotin lohkojen $\texttt{\{}s^*\ e\texttt{\}}$ ja lohkomuuttujien semantiikka. Lohkon suorituksen aluksi varataan lohkon tasolla määriteltäville muuttujille x^*_l ja x^*_d muisti-

paikat a_l^* ja a_d^* muistista σ sekä talletetaan muuttujien ja muistipaikkojen yhteys ympäristöihin $\mathcal{E}_l'$ ja $\mathcal{E}_d'$. Muistipaikat alustetaan erityiseen arvoon, jonka lukeminen aiheuttaa ajonaikaisen virheen. Sitten lohkon lauseet s^* ja lohkon arvon tuottava lauseke e suoritetaan vasemmalta oikealle. Muuttujien määrittelyt suoritetaan evaluoimalla ensin yhtäsuuruusmerkin oikean puolen lauseke arvoksi ja kirjoittamalla sitten tuo arvo muistiin. Muuttujan arvo saadaan hakemalla ensin osoite ympäristöstä ja sitten osoitteella varsinainen arvo muistista.

$$(\llbracket \{\overline{m^+}\} \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) \longrightarrow (\langle [\overline{m^+}], \mathcal{E}_l \rangle, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa)$$

$$\begin{aligned}
(\llbracket e\ \overline{e^+} \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) &\longrightarrow (e, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{callee}(\kappa, \mathcal{E}_l, \mathcal{E}_d, [\overline{e^+}])) \\
(\llbracket o\ e\ \overline{e^*} \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) &\longrightarrow (e, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{args}(\kappa, \mathcal{E}_l, \mathcal{E}_d, [\overline{e^*}], o, [])) \\
(\llbracket o \rrbracket, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) &\longrightarrow (primcall(o, []), \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa)
\end{aligned}$$

$$\begin{aligned}
&(v, _, _, \sigma, \texttt{callee}(\kappa, \mathcal{E}_l, \mathcal{E}_d, e :: e^*)) \\
&\qquad \longrightarrow (e, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{args}(\kappa, \mathcal{E}_l, \mathcal{E}_d, e^*, v, [])) \\
&(v, _, _, \sigma, \texttt{args}(\kappa, \mathcal{E}_l, \mathcal{E}_d, e :: e^*, f, v^*) \\
&\qquad \longrightarrow (e, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{args}(\kappa, \mathcal{E}_l, \mathcal{E}_d, e^*, f, v :: v^*) \\
&(v, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{args}(\kappa, \mathcal{E}_l', \mathcal{E}_d', [], \langle \llbracket \overline{x^*} \mid _ \Rightarrow _ \rrbracket :: m^*, \mathcal{E}_l'' \rangle, v^*))\ \texttt{if}\ \|x^*\| \neq \|v :: v^*\| \\
&\qquad \longrightarrow (v, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{args}(\kappa, \mathcal{E}_l', \mathcal{E}_d', [], \langle m^*, \mathcal{E}_l'' \rangle, v^*) \\
&(v, _, _, \sigma, \texttt{args}(\kappa, _, \mathcal{E}_d, [], \langle \llbracket \overline{x^*} \mid e_c \Rightarrow e_b \rrbracket :: m^*, \mathcal{E}_l \rangle, v^*))\ \texttt{if}\ \|x^*\| = \|v :: v^*\| \\
&\qquad \longrightarrow (e_c, \mathcal{E}_l', \mathcal{E}_d', \sigma''', \texttt{cond}(\kappa, \mathcal{E}_l', \mathcal{E}_d', e_b, \langle m^*, \mathcal{E}_l \rangle, v :: v^*, \mathcal{E}_d)) \\
&\qquad\ \texttt{where}\ ((x_l^*, v_l^*), (x_d^*, v_d^*)) = paramDefs([\overline{x^*}], reverse(v :: v^*)) \\
&\qquad\qquad (a_l^*, \sigma') = alloc(\sigma, \|x_l^*\|);\ (a_d^*, \sigma'') = alloc(\sigma', \|x_d^*\|) \\
&\qquad\qquad \sigma''' = \sigma''[\overline{a_l^* := v_l^*}, \overline{a_d^* := v_d^*}] \\
&\qquad\qquad \mathcal{E}_l' = \mathcal{E}_l[\overline{x_l^* \mapsto a_l^*}];\ \mathcal{E}_d' = \mathcal{E}_d[\overline{x_d^* \mapsto a_d^*}] \\
&(v, _, _, \sigma, \texttt{args}(\kappa, \mathcal{E}_l, \mathcal{E}_d, [], o, v^*) \\
&\qquad \longrightarrow (primcall(o, reverse(v :: v^*)), \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa) \\
&(\texttt{false}, _, _, \sigma, \texttt{cond}(\kappa, \mathcal{E}_l, _, _, f, v :: v^*, \mathcal{E}_d)) \\
&\qquad \longrightarrow (v, \mathcal{E}_l, \mathcal{E}_d, \sigma, \texttt{args}(\kappa, \mathcal{E}_l, \mathcal{E}_d, [], f, v^*)) \\
&(\texttt{true}, _, _, \sigma, \texttt{cond}(\kappa, \mathcal{E}_l, \mathcal{E}_d, e, _, _, _)) \longrightarrow (e, \mathcal{E}_l, \mathcal{E}_d, \sigma, \kappa)
\end{aligned}$$

Kuva 3.2. *Complotin funktioiden ja primitiivioperaatioiden semantiikka*

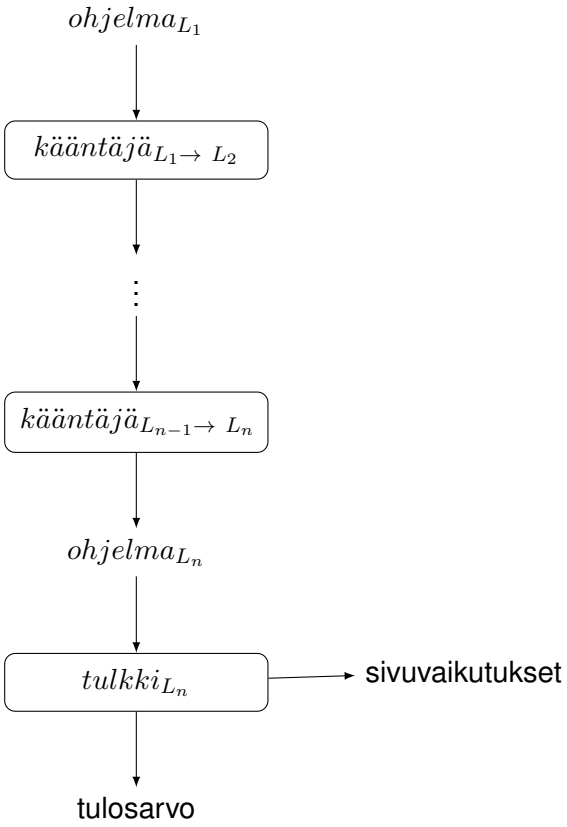
Complotin funktioiden ja primitiivioperaatioiden operatiivinen semantiikka on kuvassa 3.2. Funktioliteraalin $\{\overline{m^+}\}$ evalointi tapahtuu tallentamalla metodit m^+ staattisen ympäristön $\mathcal{E}_l$ kanssa sulkeumaksi $\langle [\overline{m^+}], \mathcal{E}_l \rangle$. Funktiokutsun $e\ e^*$ suoritus aloitetaan evaluoimalla ensin kutsuttava funktio ja sitten argumentit vasemmalta oikealle. Primitiivioperaation (vaikkapa kokonaislukujen summaaminen `__iAdd`) evaluointi alkaa muuten samoin, mutta itse operaatiota ei ole evaluoitava.

Funktiokutsun loppuosa on semantiikan monimutkaisin. Sulkeuman metodeista haetaan lineaarisesti ensimmäinen, jonka parametrien x^* määrä on sama kuin saatujen argument-

tien. Sitten argumentit kytketään parametreihin ympäristöjen ja muistin kautta ja evaluoidaan metodin esiehto e_c. Mikäli esiehto on tosi, jatketaan metodin vartalon e_b evaluoitiin, muuten palataan etsimään seuraavaa oikean parametrimäärän omaavaa metodia. Parametriviittaukset evaluoidaan täysin samoin kuin lohkomuuttujat. Primitiivioperaatiokutsun loppuosan taas määrittelee metafunktio $primcall$, joka sisältää primitiivioperaatioiden toteutukset eli esimerkiksi `__iAdd`:in tapauksessa argumenttien yhteenlaskun ja tuloksen palauttamisen. $Primcall$:in yksityiskohdat jätetään tässä avoimeksi, koska ne eivät ole semantiikan kannalta olennaisia ja saattavat käytännönläheisinä vaihdella hyvinkin paljon eri toteutusten välillä ja ajan kuluessa.

4 TULKISTA KÄÄNTÄJÄKSI

Tässä työssä yhdistetään kuvan 4.1 mukaisesti luvun 2.3 ajatus tulkin ja kääntäjän sekä luvun 2.4 ajatus useiden kääntäjien yhdistämisestä.



Kuva 4.1. *Tulkki ∘ kääntäjä ∘ ... ∘ kääntäjä*

Kielen L_1 tulkki voidaan korvata kielen L_2 tulkin ja $L_1 \rightarrow L_2$ -kääntäjän yhdistelmällä. Tässä uudessa yhdistelmässä kielen L_2 tulkki taas voidaan korvata kielen L_3 tulkilla ja kääntäjällä L_2:sta L_3:een. Kieli L_i laaditaan aina niin että se on lähes sama kuin L_{i-1}, mutta hieman lähempänä lopullista kohdekieltä L_n. Lopulta päädytään kielen L_1 toteutukseen, joka koostuu modulaarisesta $L_1 \rightarrow L_n$ kääntäjästä ja kielen L_n tulkista. Kieleksi L_n voitaisiin myös valita jokin kieli, jolle tulkki on valmiiksi saatavilla.

4.1 Lähtökohta: jäsennyspuun tulkkaaminen

Kääntäjäksi muuttamamme tulkki perustuu läheisesti luvun 3 CESK- tilakoneeseen. Eksplisiittisen muistin (semantiikan **S**tore) sijaan tulkki kuitenkin hyödyntää Racketin implisiittistä kekoa, joten käytännössä se on lähempänä CEK-tilakonetta [14, s. 100–105]. Lisäksi tilasiirtymien ketjuttamiseen käytetään transitiivisen sulkeuman mainitsemisen sijaan häntäkutsuja ja virhetilanteiden ilmaisemiseen Racketin poikkeuskäsittelyä ikuisen silmukan sijaan.

Lisäksi tulkki on selkeyden vuoksi jaettu kolmeksi eri funktioksi `eval`, `continue` ja `apply`. Continue-funktio sisältää tilakoneen siirtymät, joissa ohjelmafragmentti on pelkistynyt vakioksi ja apply-funktio toteuttaa funktiokutsut eval-funktion hoitaessa loput eli lähinnä tilanteet, joissa ohjelmafragmentti vaatii vielä pelkistämistä. Jaottelu osoittautui hyödylliseksi myös CPS-muunnoksen (luku 5.2.1) suunnittelussa.

4.2 Kohdetulkki: virtuaalikone

Kohdekieleksi L_n voitaisiin valita jokin kieli, jolle tehokas tulkki on jo olemassa, kuten x86- tai ARM-konekieli tai Java-virtuaalikoneen tavukoodi [36][35][25]. Tällöin vastuullemme jäisi pelkkä kääntäjä. Tässä työssä näin ei tehty prosessorien ja virtuaalikoneiden työn kannalta täysin epäolennaisten yksityiskohtien käsittelyn välttämiseksi.

Sen sijaan lopullisena käännöskohteena toimii erityisesti Complotille suunnitellun virtuaalikoneen tavukooditiedosto. Virtuaalikoneen käskykanta sisältää samat primitiivioperaatiot (kuten aritmetiikka ja monikoiden (’tuple’) luominen) kuin tulkkikin. Toisaalta virtuaalikoneeseen ei ole sisäänrakennettu mitään erityistä pinomekanismia funktiokutsuja varten, koska kääntäjä toteuttaa pinon jatkumasulkeumina.

Complot-virtuaalikoneen käskyt ovat 32-bittisiä kokonaislukuja. Olkoon vähiten merkitsevän bitin indeksi 0. Bitit 0-7 kertovat, mikä käsky on kyseessä. Bitit 8-15, 16-23 ja 24-39 sisältävät käskyn operandit, joita voi olla yhdestä kolmeen. Käskystä riippuen kukin operandi on joko hypyn pituus, kohderekisteri, lähderekisteri tai lähdevakion indeksi funktion vakiotaulukossa. Mikäli operandi voi olla joko rekisteri tai vakion indeksi, käytetään operandin vähiten merkitsevää bittiä kertomaan kumpi on kyseessä. Käytössä on siis korkeintaan $2^7 = 128$ rekisteriä. Rekisterien määrää voitaisiin lisätä erityisillä laajennuskäskyillä kuten Selfin tavukoodissa[4], mutta tätä ei ole toteutettu. Toisaalta kääntäjä voi käyttää muistia, jos rekisterit loppuvat [1, luku 11], mutta tätäkään ei ole toteutettu, koska käytännössä 128 rekisteriä on melkein aina paljon enemmän kuin tarpeeksi[1].

Tavukooditiedostojen formaatti on suunnilleen sama kuin PUC Luan vastaavien [27, s. 3–14]. Olennaisin ero on se, ettei Complotin tavukooditiedostoissa esiinny sisäkkäisiä funkioita.

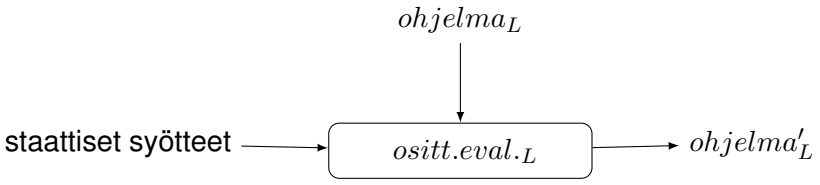
[1] Jos funktiossa tarvitaan yli 128 paikallista muuttujaa on ohjelma niin ongelmainen, että voi olla parempi, ettei se edes käänny. Makroilla tai muutoin ohjelmallisesti luodun lähdekoodin kanssa rajoite saattaisi kuitenkin tulla vastaan kohtuullisissakin käyttötapauksissa.

Virtuaalikoneemme erottaa siis fyysisistä prosessoreista suoritusnopeuden lisäksi lähinnä hieman suurempi rekisterien määrä ja operaatioiden korkeampi abstraktiotaso. Virtuaalikoneen fyysisiä prosessoreita lähestyvä abstraktiotaso toivottavasti lisää tämän työn vakuuttavuutta. Käytännöllisempi toteutus pitäisi virtuaalikoneen abstraktiotason korkeammalla kääntämisen helpottamiseksi ja suoritusnopeuden lisäämiseksi. Nimittäin päinvastoin kuin fyysisissä prosessoreissa, joissa RISC on pitkään vallannut alaa, virtuaalikoneissa monimutkaiset käskyt ovat eduksi, koska emulointisilmukan kiertäminen on itsessään hidasta [13].

Virtuaalikone on toteutettu Rust-ohjelmointikielellä [43], joka pyrkii olemaan turvallisempi ja modernimpi vaihtoehto virtuaalikoneiden toteutuksessa yleensä käytettäville C:lle ja C++:lle. Matalahkon tason ohjelmointikielen käyttö on tarpeen muun muassa tavukoodikäskyjen nopean suorituksen sekä toteutuskielestä poikkeavan muistinhallinnan toteuttamiseksi.

4.3 Vaihtoehtona osittaisevaluointi

Tulkin muuttamiseen kääntäjäksi voitaisiin käyttää myös osittaisevaluointia. **Osittaisevaluaattori** on kääntäjä, joka erikoistaa kielellä L kirjoitetun ohjelman sen staattisille eli jo käännösaikana tiedossa oleville syötteille (kuva 4.2). Nimi tulee siitä, että osittaisevaluaattori evaluoi ohjelman niiltä osin kuin pelkkien staattisten syötteiden perusteella ja todellisen ajoympäristön puuttuessa[2] on mahdollista. [20]



Kuva 4.2. *Osittaisevaluointi*

Osittaisevalointia voidaan käyttää optimoivana käännösvaiheena, jolloin staattiset syötteet ovat lähdekoodissa esiintyvät vakiot ja kyseessä siis erityisen tehokas vakioiden levittämisen muoto [44][45]. Käyttämällä osittaisevaluaattorin syötteinä tulkkeja ja osittaisevaluaattoreita Futamuran projektioiden (liite B) mukaisesti voidaan sitä käyttää myöskin tulkin automaattiseen muuttamiseen kääntäjäksi [20, s. xi].

Futamuran projektiot ovat elegantti ja yleispätevä tapa tulkkien muuttamiseen kääntäjiksi. Ei ole kuitenkaan mitään syytä olettaa, että tällä tavalla tuotetut kääntäjät olisivat tehokkaita, tuottaisivat tehokasta koodia tai olisivat ihmisten luettavissa. Käytännössä nopeat ja tehokasta koodia tuottavat osittaisevaluaattorimaiset työkalut kuten Truffle [49] ja RPython [33] vastaavat lähinnä Futamuran ensimmäistä projektiota ja ovat monimutkaisia jopa verrattuna tavallisiin kääntäjiin.

Tämän kandidaatintyön tavoitteena taas on tuottaa yksinkertainen ja luettava kääntäjä,

[2] Suurin rajoite on I/O:n täydellinen puuttuminen.

jota on mahdollista jatkokehittää tulkista riippumatta tavalliseen tapaan. Kääntäjien pitäisi kaikkien ohjelmistojen tavoin olla ymmärrettäviä ja niiden jatkokehityksen suoraviivaista.

5 MUUTOSVAIHEET

Tässä luvussa tulkin ominaisuuksia korvataan yksi kerrallaan käännösvaiheilla. Käytännössä tämä tarkoittaa sitä, että kirjoitetaan uusi tulkki, josta käsiteltävä toiminnallisuus puuttuu ja uusi käännösvaihe, joka palauttaa ominaisuuden yksinkertaisemmiksi operaatioiksi. Sitten liitetään yhteen jo toteutetut käännösvaiheet, uusi käännösvaihe ja tulkki. Näiden muutosten jälkeen uuden käännösputken toiminnallisuutta ja suorituskykyä voidaan testata suhteessa aiemmin koottuihin putkiin.

5.1 Alkupää (frontend)

Kääntäjän alkupään (’frontend’) tarkoitus on muuttaa käännettävä ohjelma ihmisystävällisestä lähdekoodista kääntäjän keskiosan (’middle end’) käyttämään yksinkertaistettuun ja säännölliseen muotoon. Lisäksi alkupään vastuulla on yleensä tarkistaa, että ohjelma on syntaktisesti ja semanttisesti hyväksyttävä. Näiden tehtävien laatu ja laajuus riippuvat suuresti lähdekielestä.

Tämän työn Complot-kääntäjän alkupää jäsentää lähdekoodin syntaksipuuksi, josta se sitten korvaa keskiosalle sopivilla ratkaisuilla tarpeen muuttujien lohkomuotoisille näkyvyysalueille (’block scope’), rekursiiviset määritelmät, dynaamiset muuttujat sekä metodien valinnan esiehtojen perusteella. Koska Complot on dynaamisesti tyypitetty kieli, kääntäjän semanttiset tarkistukset koostuvat vain määrittelemättömien ja samassa lohkossa useita kertoja määriteltyjen muuttujien havaitsemisesta. Toisaalta pienimmätkin syntaksivirheet havaitaan, koska ne tekevät koko käännösyksikön syntaksipuun muodostamisesta ja sitä kautta käännösprosessin jatkamisesta mahdotonta.

5.1.1 Jäsentäminen

Jäsentäminen tekee kieliopin mukaisesta lähdekoodista syntaksipuun ja antaa syntaksivirheistä virheilmoitukset. Työn tavoitteen kannalta jäsennysvaihe ei ole kovin olennainen, sillä lähtötulkkikin tulkkaa lähdekoodin sijaan jäsentäjän laatimaa abstraktia syntaksipuuta (AST). Tässä työssä jäsentäjä toteutettiin Racketin parser-tools -kirjaston selaaja- ja jäsentäjägeneraattoreilla [30].

5.1.2 Alfatisointi

Complotin staattisten muuttujien näkyvyysalueet noudattavat lähdekoodin lohkorakennetta. Tämä on nähtävissä myös staattisen ympäristön $\mathcal{E}_l$ käsittelystä luvussa 3. Lohkorakenteen ansiosta on yksinkertaista löytää lähdekoodista kaikki kohdat, joissa staattista muuttujaa käytetään tai löytää jossain tietyssä paikassa käytetyn muuttujan määritelmä. Tämä helpottaa sekä ohjelmoijan että kielen toteutuksen ohjelmien semantiikkaa koskevaa päättelyä.

Kääntäjälle muuttujan käytöstä määritelmään ja määritelmästä käyttöön siirtymisen pitää kuitenkin olla paitsi johdonmukaista myös yksinkertaista ja nopeaa. Lisäksi CPS- ja sulkeumamuunnokset (luvut 5.2.1 ja 5.2.3) poistavat lohkorakenteen ensin tavallisten lohkojen ja sitten myös funktioiden osalta.

Alfatisointi on käännösvaihe, joka muuttaa kaikki staattisten muuttujien nimet uniikeiksi. Tämän jälkeen muuttujien nimet voidaan yhdistää määritelmiin käyttämättä lohkorakennetta, koska muuttujan peittyminen saman nimisellä määritelmällä sisemmässä lohkossa ei ole enää mahdollista. Käytöt voidaan tällöin kytkeä määritelmiin koko käännösyksikön kattavien hajautustaulujen kautta tai jopa suoraan osoittimilla. Tässä työssä käännösvaiheet käyttävät ensin mainittuja hajautustauluja, joita myös **symbolitauluiksi** [2, s. 103] kutsutaan.

Koska Complotissa ei ole sijoitusoperaattoria on kukin nimi myös yksimääritelmäinen. Myös yksimääritelmäisyys helpottaa tiettyjä käännösvaiheita kuten kopioiden ja vakioiden levittämistä (luku 5.2.4) huomattavasti, koska datavuoanalyysia [2, luku 17] ei tarvita [2, s. 417–419].

Myös myöhemmät käännösvaiheet joutuvat muutoksia tehdessään säilyttämään muuttujien nimien yksikäsitteisyyden ja yksimääritelmäisyyden. Tämä ei kuitenkaan ole vaikeaa vaan vaatii vain hieman huolellisuutta.

Alfatisointi voidaan toteuttaa käymällä syntaksipuu läpi esijärjestyksessä käyttäen perittynä attribuuttina muuttujan vanhan nimen uuteen assosioivaa hakurakennetta. Hakurakenne vastaa siis tulkin staattista ympäristöä, mutta läpikäyntijärjestys noudattaa suoritusjärjestyksen sijaan syntaksipuun rakennetta. Kohdattaessa muuttujan määritelmä generoidaan muuttujalle uusi uniikki nimi. Koska tavallisten lohkojen määritelmät ovat rekursiivisia, pitää tavallisten lohkojen määrittelemät nimet lisätä hakurakenteeseen ennen lohkon alilausekkeiden läpikäyntiä. Sekä muuttujien määritelmät että käytöt korvataan hakurakenteesta löytyvillä. Alfatisointi toimii myös tarkistavana käännösvaiheena, koska määrittelemättömät staattiset muuttujat aiheuttavat siinä virheen. [39, s. 45]

Tämän työn toteutuksessa uudet nimet generoidaan selkeyden vuoksi vanhojen pohjalta Racketin `gensym`-funktiolla. Tehokkaampaa olisi käyttää pelkkiä kokonaislukuja [7] tai osoittimia muuttujan määritelmään [5][23].

5.1.3 Staattisen sidonnan linearisointi

Semantiikkaa (luku 3) noudattaen tulkkimme sitoo muuttujanimet kaksivaiheisesti rekursiivisten määritelmien mahdollistamiseksi. Tässä käännösvaiheessa tämä kaksivaiheisuus siirretään tulkista itse ohjelmaan jakamalla muuttujien määrittelyt muuttujan luomiseen ja sen alustamiseen määrittelyn mukaisella arvolla. Viittaukset muuttujaan muutetaan eksplisiittisiksi latauksiksi muuttujasta. Alustamattoman muuttujan lukeminen tuottaa ajoaikaisen virheen.

Mikäli nimeen ei viitata ennen sen määrittelyä, ei kaksivaiheisuutta tarvita. Tämä optimointi välttää ajoaikaisia muistin varauksia, lukuja ja kirjoituksia. Myös käännökseen tarvittava muisti ja aika vähenee, koska ohjelman väliesitykset ovat pienempiä.

Myös merkittävä osa niistä tapauksista, joissa nimeä käytetään ennen alustusta voitaisiin havaita jo tässä käännösvaiheessa[10]. Complotin määritelmä nimittäin sallii tehdä näistä käännösvirheitä ajonaikaisten virheiden sijaan.

5.1.4 Dynaamisen ympäristön konkretisointi

Tässä käännösvaiheessa dynaamisen ympäristön kuljetus ja sen hyödyntäminen dynaamisten muuttujien määrittelyissä ja käytöissä siirretään tulkista itse ohjelmaan. Samalla dynaamisten muuttujien määrittelyt jaetaan luomiseen ja alustukseen samaan tapaan kuin tehtiin staattisille muuttujille luvussa 5.1.3.

Dynaaminen ympäristö toteutetaan ajonaikaisena muuttumattomana hakurakenteena. Kussakin tavallisessa lohkossa luodaan uusi dynaaminen ympäristö lisäämällä ulomman lohkon dynaamiseen ympäristöön sisemmässä lohkossa määriteltävät muuttujat. Uusi ympäristö sidotaan tuoreeseen staattiseen muuttujaan lohkon alussa. Funktiot taas saavat dynaamisen ympäristön nyt lisättävänä parametrina ja vastaavasti kulloinenkin dynaaminen ympäristö välitetään jokaisessa funktiokutsussa nyt lisättävänä argumenttina. Menettely vastaa läheisesti Reader-monadin toteutusta[19, s. 28].

Todennäköisesti useimmissa lohkoissa ei määritellä yhtään dynaamista muuttujaa. Tällöin voidaan uuden dynaamisen ympäristön luomisen sijaan vain käyttää suoraan ulomman lohkon dynaamista ympäristöä. Sen sijaan kaksivaiheisuuden välttäminen vaatisi paljon kehittyneempää analyysia kuin staattisten muuttujien tapauksessa, koska dynaaminen ympäristö välitetään kutsuttaville funktioille, joiden määritelmän löytäminen on ensiluokkaisten funktioiden läsnäollessa erittäin haastavaa[38].

Dynaamisen ympäristön konkretisointi voidaan toteuttaa käymällä syntaksipuu läpi esijärjestyksessä käyttäen dynaamisen ympäristön nimeä perittynä attribuuttina. Tätä perittyä attribuuttia vaihdetaan funktioiden ja dynaamisia muuttujia määrittelevien tavallisten lohkojen kohdalla. Jälkimmäisessä tapauksessa lisätään lohkon alkuun uuden ympäristön luova koodi, joten kuten luvussa 5.1.3 lohkossa määriteltävien muuttujien nimet pitää selvittää ennen lohkon alilausekkeiden läpikäyntiä. Muuttujien määrittelyt ja viittaukset kor-

vataan alustuksilla ja luvuilla samaan tapaan kuin luvussa 5.1.3, mutta nyt joudutaan lisäämään myös muuttujan dynaamisesta ympäristöstä hakeva koodi. Lisäksi dynaamisen ympäristön nimi lisätään funktiokutsujen parametrilistoihin.

5.1.5 Metodinvalintapuiden konkretisointi

Kuten luvussa 3 esitettiin, Complot-funktioilla voi olla useita eri metodeja, joilla on eri määrä parametreja ja/tai eri esiehdot. Yksinkertainen tapa toteuttaa tämä on välittää funktioiden lähdekoodista löytyvät argumentit yhdessä monikossa. Tällöin funktiokutsuissa siis ei enää vaihtele argumenttien määrä ja laatu vaan monikkoargumentin koko ja sisältö. Luvussa 5.1.4 lisätty dynaaminen ympäristö välitetään aina, joten se voidaan välittää monikon ulkopuolella.

Oikean metodin koodi voidaan tässä uudessa esityksessä löytää kaksitasoisella valintapuulla, joka tarkastelee ensin argumenttimonikon pituutta ja sitten esiehtojen tuloksia. Virheellisiä kutsuja varten valintapuun molempiin asteisiin lisätään oletushaara, johon laitetaan oikean virheen ajoaikana tuottava koodi. Menetelmä on yksinkertainen versio muun muassa OCamlin ja Haskellin hahmontunnistuksen toteutustekniikoista[28].

Argumenttien määrän arviointi voitaisiin tehdä myös case-rakenteella, mutta tämän työn väliesityksistä sellaista ei yksinkertaisuuden vuoksi löydy. Esiehdot taas kannattaa aina arvioida lähdekoodin järjestyksessä, koska se kuitenkin ratkaisee tasapelit ja turhat esiehtojen evaluoinnit voivat viedä mielivaltaisen ajan. Tässä työssä valintapuun molemmat asteet on siis toteutettu if–else-rakenteilla.

5.2 Keskiosa (middle end)

Kääntäjän keskiosan (’middle end’) tarkoitus on suorittaa ne madallukset, jotka ovat enimmäkseen riippumattomia sekä käännösputken lähde- että kohdekielestä. Funktionaalisille kielille tyypilliseen tapaan Complotissa näitä ovat CPS- ja sulkeumamuunnokset. Lisäksi kääntäjän keskiosa on yleensä vastuussa suurimmasta osasta tärkeimpiä optimointivaiheita. Niitä Complot-kääntäjäämme ei kuitenkaan ole toteutettu, sillä ne eivät olleet työn tavoitteiden kannalta olennaisia.

Complot-kääntäjämme keskiosa käyttää väliesityksenään eräänlaista CPS:ää (‘Continuation Passing Style’). CPS:ssä funktiokutsujen ja primitiivisten operaatioiden argumenttien on oltava **triviaaleja** [1, luku 2] eli meidän tapauksessamme muuttujan nimiä tai skalaarivakioita. Lisäksi hyppyjen ja funktioista paluiden kohteena käytetään pelkkien lohkonimien (’label’) ja implisiittisen pinon sijaan sulkeumia, jotka sisältävät koodiosoitteen lisäksi myös vapaiden muuttujien eli ’ympäristön’ tallennetut arvot [39, s. 37]. Näitä kutsutaan **jatkumiksi** (’continuation’) [1, luku 1.1].

Kääntäjämme käyttämä CPS koostuu funktioista, joisa kukin sisältää on joukon jatkumia, jotka voivat viitata vapaasti toisiinsa. Se muistuttaa siis ainakin MLtonin [50, s. 9] ja Applen Swift-kääntäjän [40] käyttämää SSA:n varianttia sillä erotuksella että peruslohko-

ja eli jatkumia voi paitsi kutsua myös käsitellä arvoina eli esimerkiksi tallentaa muuttujiin ja välittää argumentteina. Kääntäjässämme tätä laajennusta käytetään vain funktioiden paluujatkumien välittämisessä, mutta myös lähdekooditason jatkumien (kuten Schemen `call-with-current-continuation`) toteuttaminen olisi tässä järjestelyssä helppoa. SSA:n tapaan kunkin jatkuman sisällä on peruslohko, jonka voidaan ajatella koostuvan useista jatkumista, joita kutsutaan saman tien kun ne on määritelty. Toisin kuin esimerkiksi Guilen CPS:ssä näitä jatkumia ei ole ruvettu nimeämään, sillä se hidastaisi useimpia analyyseja ilman vastaavaa hyötyä[2, s. 361–362].

5.2.1 CPS-muunnos

CPS-muunnos on käännösvaihe, joka siirtää ohjelman CPS-väliesitykseen. CPS-muunnosta on tutkittu paljon ja algoritmista esiintyy monia variaatioita. Osa vaihtelusta johtuu kohde-CPS:n eroavaisuuksista: millaisia lausekkeita pidetään triviaaleina, sidotaanko jatkumia sisäkkäin ja niin edelleen. [8] Olennaisempaa on kuitenkin algoritmin rakenne ja tuotetun CPS-koodin laatu.

Naiivi CPS-muunnos nimeää kaikki argumentit, vaikka ne olisivat jo valmiiksi triviaaleja. Näitä turhia välivaiheita kutsutaan admistratiivisiksi β-redekseiksi.[1] CPS-muunnos voi myös tuottaa jatkumia, jotka vain kutsuvat toista jatkumaa argumentteina omat parametrinsa eli adminstratiivisia η-redeksejä.[2] Erityisesti adminstratiiviset η-redeksit olisi hyvä välttää, koska samalla tulemme optimoineeksi häntäkutsut [1, luku 1.1].

Monet tutkijat ovat huomanneet, että adminstratiiviset β-redeksit voidaan välttää antamalla muunnosfunktiolle parametrina jatkuman koodin sijaan sen tuottava funktio [8, s. 800]. Sabry ja Felleisen taas kehittivät algoritmin, jossa jatkumaparametri esitetään C*K-tilakoneiden tapaan linkitettyinä jatkumatietueina [34]. Kumpikin näistä algoritmeista tuottaa kuitenkin η-redeksejä ja enemmän kuin naiivi CPS-muunnos, mutta tämä on helppo korjata[8, s. 802].

Adminstratiiviset redeksit voitaisiin optimoida pois myöhemmin[1, luku 6], mutta tämä vaatii erillisiä optimoivia käännösvaiheita ja kuluttaa enemmän muistia. Tässä työssä käytettiin Sabryn ja Felleisenin algoritmia, jolla paitsi vältettiin admistratiiviset redeksit voitiin myös luontevasti käyttää CEK-tilakonemaisen tulkin toimintaa lähtö- ja vertailukohtana CPS-muunnoksen kehittämisessä.

5.2.2 Kriittisten kaarien rikkominen

Kriittinen kaari on sellainen suunnatun graafin kaari, joka ei ole ainoa lähtösolmustaan alkava eikä myöskään ainoa päätössolmuunsa loppuva kaari [29, luku 13.3]. CPS-väliesityksemme tapauksessa graafin solmut ovat jatkumia ja kaaret jatkumien mainintoja eli käyttöjä (’use’). Kriittiset kaaret vaikeuttavat sulkeumamuunnosta (luku 5.2.3) ja rekis-

[1] β-redeksille $(\lambda x.e)\ a$ voitaisiin välittömästi suorittaa β-reduktio muotoon $e[x \mapsto a]$.
[2] η-redeksille $\lambda x.fx$ voitaisiin välittömästi suorittaa η-reduktio muotoon f.

terien varaamista (luku 5.3.2). Kaikilla kriittisen kaaren lähtösolmusta lähtevien kaarien kohdejatkumilla pitäisi nimittäin olla sama vapaiden muuttujien välitystapa ja rekisterien lähtötila, mikä ei ole yleisessä tapauksessa mahdollista.

Näiden ongelmien välttämiseksi kriittiset kaaret rikotaan heti CPS-muunnoksen jälkeen η-laajentamalla eli lisäämällä kaaren lähtö- ja päätösjatkumien väliin uusi jatkuma, joka vain kutsuu päätösjatkumaa omalla parametrillaan. Tietenkin seuraavien käännösvaiheiden pitää välttää luomasta uusia kriittisiä kaaria aivan kuten alfatisoinnin jälkeiset käännösvaiheet säilyttävät muuttujanimien yksikäsitteisyyden. Toinen vaihtoehto olisi rikkoa kriittiset kaaret vasta, kun niistä tulee ongelma, mutta se olisi huonompaa vastuunjakoa ja aiheuttaisi luultavasti myös saman logiikan toistoa.

5.2.3 Sulkeumamuunnos

Complot käyttää ensisijaisesti staattista sidontaa, koska staattisesti sidottujen muuttujien suhteen on helpompi järkeillä käännösaikana. Tämä helpottaa ohjelmointityötä ja mahdollistaa myös tehokkaamman koodin tuottamiseen dynaamisin muuttujiin verrattuna. Tehokkuus tulee siitä, että viittaukset voidaan rekisterien varauksessa (luku 5.3.2) yhdistää määritelmiin jo käännösaikana ja järjestää niin, että muuttujan arvo on ajoaikana suoraan rekisterissä tai korkeintaan osoittimen indeksoinnin päässä[39, s. 9].

Tämän luvun loppuosassa 'jatkuma' viittaa myös funktioihin, sillä itse asiassa funktion kutsuminen on vain sen aloitusjatkuman kutsumista. Rekisterien varaamisen lisäksi haasteena on sulkeumien muodostuminen eli tähän asti staattinen ympäristö on funktion määritelmää tai jatkumaviittausta evaluoitaessa yhdistetty jatkuman koodin kanssa sulkeumaksi. Totesimme kuitenkin juuri, että staattisen ympäristön kaltaisista hakurakenteista olisi tarkoitus ja mahdollista päästä eroon.

Itse asiassa jatkuma ei tarvitsekaan staattista ympäristöä sinänsä vaan ainoastaan **vapaiden muuttujiensa** arvot. Jatkuman vapaat muuttujat ovat ne muuttujat, joihin jatkuma viittaa mutta joiden määritelmät ovat sen ulkopuolella[48, s. 76]. Tarvitaan siis vain jokin johdonmukainen tapa vapaiden muuttujien arvojen välittämiseksi jatkumalle sen määritelmän tai kutsujen kohdalla. Itse asiassa muiden kuin vapaiden muuttujien välittäminen aiheuttaa muistivuodon, joka muun muassa Appelin [1, luku 12] mukaan ei ole vain tehotonta vaan semanttinen virhe kielen toteutuksessa. Joskus myös pinon ylivuotoa häntärekursiosta huolimatta pidetään vastaavanlaisena virheenä[37, s. 11].

Vapaiden muuttujien välittämiseen määritelmien yhteydessä voidaan käyttää sulkeumia, mutta staattisen ympäristön sijaan sulkeumaan tallennetaan vain vapaiden muuttujien arvot taulukkoon pakattuna. Funktion tai jatkuman koodi saa sulkeuman ylimääräisenä parametrina ja viittaukset vapaisiin muuttujiin muutetaan sulkeuman taulukon indeksoinniksi. Tämä 'litteiksi sulkeumiksi' ('flat closures') kutsuttu strategia on yleispätevä, yksinkertainen ja tehokas. Siihen liittyvä runsas kopiointi määritelmien yhteydessä on tehokkaampaa kuin toinen ääripää eli hitaat haut muuttujaviittausten yhteydessä ja mahdollisesti jopa edellä mainittu mielivaltaisen kokoinen muistivuoto. [11, s. 88–97]

Mikäli jatkuman kaikki kutsukohdat tiedetään ja sen vapaat muuttujat ovat kaikissa niissä näkyvissä tai niiden arvot sinne välitettävissä voidaan vapaiden muuttujien arvot välittää jatkumalle kutsujen yhteydessä ylimääräisinä argumentteina [18]. Tämän 'lambda-liftingiksi' kutsutun strategian käyttäminen kaikkien jatkumien kohdalla vaatisi vaikeahkoa ja kallista analyysia [38], mutta saman funktion jatkumien kutsuessa toisiaan se on helppoa ja vältytään paljolta muistin varaamiselta sulkeumille.

Complot-kääntäjä käyttää pääasiallisesti litteitä sulkeumia ja yksinkertaisissa paikallisella analyysilla havaittavissa tilanteissa lambda-liftingiä. Pelkkien litteiden sulkeumien käyttäminen olisi yksinkertaisempaa ja melko tehokasta, mutta sulkeumien varaaminen vain samassa funktiossa käytetyille jatkumille vaikutti suorastaan hölmöltä[3].

5.2.4 Kopioiden ja vakioiden levittäminen

Vapaan muuttujan muuttaminen jatkuman parametriksi johtaa usein tilanteeseen, jossa parametrin kohdalla välitetään aina sama argumentti. Tämä johtuu siitä, että tuon muuttujan määritelmä dominoi (katso luku 5.3.2) kyseistä jatkumaa. Tällaiset parametrit voidaan poistaa jatkumilta ja niiden käytöt korvata vastaavilla argumenteilla.

Turhien jatkumaparametrien poistaminen on itse asiassa erikoistapaus **kopioiden levittämisestä** ('copy propagation') [2, s. 359]. Sulkeumamuunnoksessa tai muista syistä syntyneen turhan muuttujien tai vakioiden kopioinnin poistamiseksi käännettävästä ohjelmasta lisättiin sulkeumamuunnoksen jälkeen yksinkertainen kopioiden ja vakioiden levittämisvaihe [2, s. 418–419].

SSA-muotoa ('Single Static Assignment') käytettäessä tilannetta, jossa edellä kuvattuja turhia jatkumaparametreja eli ϕ-funktioita ei esiinny kutsutaan minimaaliseksi SSA-muodoksi ja sen saavuttamiseen käytetään Iterated Dominance Frontiers -algoritmia [2, s. 404–408]. Tuo algoritmi on kuitenkin paljon monimutkaisempi ja hieman hitaampi [24] kuin lineaarinen kopioiden levitys vaikka se tekee vähemmän parannuksia ohjelmaan.

5.3 Loppupää (backend)

Kääntäjän loppupää ('backend') muuttaa keskiosan käyttämän väliesityksen kohdekielelle. Complot-kääntäjämme loppupää koostuu tyypilliseen tapaan käskyjen valinnasta, rekisterien varaamisesta ja koodin generoimisesta. Käskyjen valinta (luku 5.3.1) ja rekisterien varaaminen (luvut 5.3.2 ja 5.3.3) muuttavat keskiosan operaatiot ja muuttujat vastaamaan kohdekielen käskyjä ja rekistereitä. Niillä on madaltamisen lisäksi suuri vaikutus myös tuotetun koodin suorituskykyyn [3, s. 5][16, s. 1–2]. Koodin generointi (luvut 5.3.4-5.3.7) taas on lähinnä madaltavaa koostuen koodin kirjoittamisen lisäksi muutamista kirjanpidollisista oheistoiminnoista kuten hyppyjen ja vakioiden esittämisestä kohdearkkitehtuurin tukemilla tavoilla.

[3] 'As I tell my compiler students now, there is a fine line between "optimization" and "not being stupid".' [12, s. 4]

Koska kohteenamme on virtuaalikone, käskyjen valitseminen on varsin triviaalia. Myös rekisterien varaaminen on helpompaa, kun niitä on runsaasti saatavilla. Koodin generointikin on varsin suoraviivaista ja yleisillä arkkitehtuureillakin siinä olisivat haasteena lähinnä käskykannan binäärikoodauksen epäsäännöllisyydet.

5.3.1 Käskyjen valinta

Ensin loppupään pitää varmistaa, että kaikki primitiivisinä kohdellut operaatiot on mahdollista toteuttaa suoraan kohdearkkitehtuurin yksittäisillä käskyillä. Koska kohteena on virtuaalikone, tässä vaiheessa tarvitsee vain varmistaa kunkin käskyn käyttävän korkeintaan kahta argumenttia. Käytännössä tämä toteutetaan muuttamalla moniparametriset käskyt jonoiksi korkeintaan kaksiparametrisia. Esimerkiksi monikoiden luonti ositetaan muistin varaukseen ja kunkin jäsenen arvon kirjoittamiseen.

Mikäli kohdearkkitehtuuri olisi fyysinen suoritin tai vaikkapa Java-virtuaalikone, jouduttaisiin kaikki primitiivisinä pidetyt operaatiot korvaamaan toisilla. Tähän voitaisiin käyttää esimerkiksi makrolaajennus[3, s. 13–30]- tai puidensovitus[3, s. 31–76]-tekniikoita.

5.3.2 Rekisterien varaaminen

Kohdevirtuaalikoneessa ei ole rajatonta määrää nimettyjä muuttujia vaan rajattu määrä numeroituja rekistereitä. Rekisterien varauksessa muuttujien nimet korvataan rekisterien numeroilla. Oletamme, että virtuaalikoneen 128 rekisteriä riittävät kaikille muuttujille, jolloin muuttujia ei tarvitse pitää rekisterien sijaan muistissa (’spilling’ [2, s. 219]), eikä siirrellä rekisterien välillä (’live-interval splitting’ [32, s. 911]). Toisaalta CPS:n käytön voidaan ajatella SSA:n tavoin jo aiheuttaneen muuttujien jakautumista useaan muuttujaan lohkoparametrien välityksen kautta [16, s. 4].

Oppikirjoissa suositeltu tapa rekisterien varaamiseksi on funktion interferenssigraafin rakentaminen ja rekisterien varaaminen sen pohjalta graafinväritysheuristiikoilla kuten IRC (Iterated Register Coalescing). Interferenssigraafissa funktion muuttujat ovat solmuja ja kahden solmun välinen kaari kertoo muuttujien interferoivan eli olevan yhtä aikaa elossa jossain kohtaa funktiota [2, s. 219].

Interferenssigraafin rakentamiseen vaadittava eläväisyysanalyysi on kuitenkin mahdollisesti hidas ja interferenssigraafi voi myös viedä paljon tilaa [16, s. 33]. Lisäksi graafin värittäminen on NP-täydellinen ongelma, joten siihen käytetään heuristisia, jopa iteratiivisia menetelmiä. Linear Scan -algoritmit käyttävät nopeampia heuristiikoita ja tuottavat hitaampaa koodia kuin IRC[32].

Käytettäessä SSA:ta tai CPS:ää voidaan eläväisyysanalyysi ja heuristiikat kuitenkin kokonaan välttää. Nimittäin kun ohjelman näkyvyys noudattaa dominanssisääntöä, ohjelman interferenssigraafi on jänteellinen (’chordal’) eli kolmioitu (’triangulated’). [16, s. 40]

Kolmioitu graafi ei sisällä yhtään kolmea kaarta pidempää indusoitua sykliä [16, s. 107].

Aligraafi on indusoitu, kun se sisältää kaikki päägraafin aligraafiin kuuluvien solmujen väliset kaaret [16, s. 105]. Tällöin on olemassa täydellinen eliminaatiojärjestys eli järjestys, jossa graafi voidaan värittää ja siis rekisterit varata niin, että niitä tarvitaan mahdollisimman vähän [16, s. 108].

Peruslohkon sisällä täydellinen eliminaatiojärjestys on muuttujien esittelyjärjestys eli suoritusjärjestys ja Hackin SSA-algoritmi [16] muistuttaa Wangin [46] intuitiivista rekisterien varausta abstraktilla tulkkauksella. Kokonaiselle kontrolligraafille täydellinen eliminaatiojärjestys vastaa dominaattoripuun esijärjestystä ja algoritmi laajenee yksinkertaisesti Hackin esittämään muotoon [16, s. 54][4]. Esijärjestyksen noudattaminen on myös sikäli ilmeinen ratkaisu, että tällöin jatkuman vapaiden muuttujien rekisterit on varmasti varataan ennen jatkuman käsittelyä.

Rekisterien varaamiseen valittiin Hackin algoritmi [16, s. 54], koska se on yksinkertainen toteuttaa ja suoritusajaltaan lineaarinen ohjelman koon suhteen. Algoritmin hyvät ominaisuudet riippuvat SSA:n tai CPS:n käytöstä, mutta CPS:ää oli jo muutenkin päätetty käyttää.

Myös Hackin algoritmin käyttämiseksi täytyy ensin suorittaa eläväisyysanalyysi. Algoritmi ei kuitenkaan tarvitse kokonaista interferenssigraafia vaan muuttujien elinkaarten loput riittävät Wangin [46, s. 3–4] tapaan. Toisin kuin muissa mainituissa rekisterien varausalgoritmeissa, Hackin menetelmässä myös ‘spilling’ suoritettaisiin erillisenä vaiheena ennen itse rekisterien varausta eikä sen aikana [16, s. 44–52].

Sulkeumamuunnoksessa funktiokutsujen paluujatkumista tehtiin uusia sisääntulokohtia kontrolligraafille. Tämä helpottaa rekisterien varausta sikäli, että rekisterien tallentaminen funktiokutsujen yhteydessä tuli jo tehtyä jatkumien sulkeumamuunnoksen yhteydessä.

Toisaalta sisääntulokohtien lisääminen vaikuttaisi tekevän Hackin algoritmin vaatiman dominaattoripuun muodostamisesta mahdotonta, koska perinteinen dominanssirelaation määritelmä

> Ohjelman piste p_1 dominoi toista pistettä p_2 joss suoritus ei voi päästä sisääntulokohdasta p_2:een kulkematta p_1:n kautta. [2, s. 379]

olettaa että sisääntulokohtia on vain yksi. Puu voidaan kuitenkin muodostaa luomalla aluksi kuvitteellinen tyhjä juurijatkuma, jonka määritellään dominoivan kaikkia sisääntulokohtia. Tässä työssä dominaattoripuun muodostamiseen käytettiin Cooperin, Harveyn ja Kennedyn yksinkertaista ja käytännössä nopeaa dominanssialgoritmia[6].

Linear Scan kompastuu kontrollivuon jakautumiseen [16, s. 34] ja Wangin algoritmi yhtymiseen [46, s. 7]. Hackin algoritmi selviää näistä vaivatta lohkoparametrien (tai ϕ-funktioiden) ansiosta. Voidaan kuitenkin ajatella, että Hackin tavoitteena on siirtää hankalat lisäongelmat kuten muistin käyttö muuttujien säilyttämiseen [16, s. 44–52] ja siirtokäskyjen minimointi [16, s. 59–74] erillisiin vaiheisiin. Eikä seuraava vaihe (5.3.3) ole lainkaan niin elegantti ja optimaalinen kuin Hackin menetelmän muut osat. Hackin muut-

[4] Hack käyttää lohkoparametrien sijaan ϕ-funktioita, joten parametrit tulevat käsiteltyä tavallisina käskyinä.

tujien yhdistämisheuristiikkoja [16, s. 63–70] ei niiden monimutkaisuuden vuoksi lähdetty tässä työssä toteuttamaan.

5.3.3 Jatkumakutsujen korvaaminen hypyillä

Ohjelman rekisterikäytös ei ole vieläkään täysin määritelty vaan jatkumat ovat parametrisoituja ja jatkumaa kutsuttaessa argumentit siirretään yhtäaikaisesti parametreille varattuihin rekistereihin. Koska nyt kaikkien muuttujien sijaintirekisterit ovat tiedossa, parametrit ovat olemassa enää näitä rinnakkaisia siirtoja varten. [16, s. 55]

Prosessoreissa eikä virtuaalikoneessaammekaan ole tällaisia rinnakkaisia siirtokäskyjä. Niinpä siirrot pitää sarjallistaa. [16, s. 55] Sarjallistuksen jälkeen jatkumat eivät enää tarvitse parametreja, koska argumentit välitetään rekistereissä kuin globaaleissa muuttujissa ikään. Jatkumakutsut voidaan siis toteuttaa pelkkinä hyppykäskyinä. SSA-maailmassa tässä vaiheessa vastaavasti poistetaan ϕ-funktiot, jota kutsutaan SSA:n 'tuhoamiseksi' ('SSA destruction') [16, s. 15], vaikka oikeastaan jo muuttujien korvaaminen rekistereillä kumoaa dominanssinäkyvyysominaisuuden.

Tarvittavat siirrot muodostavat enimmäkseen puumaisten suunnattujen graafien joukon. Siirrot voidaan sarjallistaa lähtemällä graafien lehdistä. Kun lehtiä ei ole enää jäljellä, jää pelkkiä syklejä. Syklit voidaan toteuttaa 'rikkomalla' ensin sykli vapaan rekisterin tai muistipaikan avulla tai sitten swap-käskyjen[5] sarjoilla mikäli rekistereitä ei ole vapaana eikä muistiin haluta tätä varten koskea. [16, s. 55– 58] Koska virtuaalikoneessamme on runsaasti rekistereitä, syklit voitaisiin rikkoa lisärekisterillä. Kääntäjän tuottamien siirtokäskyjen määrän vähentämiseksi virtuaalikoneeseen lisättiin kuitenkin tätä varten swap-käsky.

5.3.4 Jatkumagraafin linearisointi

Seuraavaksi jatkumagraafi muutetaan abstraktiotasoltaan assembly-kieltä vastaavaksi käskyvuoksi. Käytännössä jatkumille vain valitaan järjestys ja asetetaan ne peräkkäin. Seuraavaan jatkumaan kohdistuvat hypyt voidaan jättää tässä vaiheessa turhina pois. Järjestykseksi valittiin käänteinen jälkijärjestys, koska tällöin asyklisen jatkumagraafin[6] kaikki hypyt suuntautuvat eteenpäin. Kun lisäksi `if`:ien tosi-vaihtoehto tulee ennen epätosi-vaihtoehtoa, saadaan ne toteutettua pelkällä yksittäisellä ehdollisella haarautumiskäskyllä. Jatkumakutsut on nyt korvattu hypyillä, mutta ne kohdistuvat vielä jatkumien nimiin.

5.3.5 Hyppyjen pituuksien laskeminen

Seuraavaksi lasketaan hyppyjen pituudet. Tämä tehdään laskemalla kunkin jatkuman alun indeksi käskyvirrassa ja käymällä sitten käskyvirta läpi ohjelmalaskuria simuloiden

[5] Swap-käskyn puuttuessa voidaan hyödyntää XOR swap -algoritmia.

[6] Tässä kääntäjässä kaikki jatkumagraafit ovat asyklisia, koska lähdekielessä ei ole silmukkarakenteita eikä rekursion silmukaksi muuttavia optimointeja ole toteutettu.

ja lisäten kohteiden nimien rinnalle hypyn pituus ja suunta $\pm$(jatkuman indeksi - ohjelmalaskuri). Tässä mukaillaan siis tuloskoodin tulkkia lähtökoodin tulkin sijaan, mutta joka tapauksessa tulkin seuraamisesta on hyötyä vielä näinkin alhaisella tasolla.

Hyppyjen pituudet voitaisiin laskea myös samalla kun graafia linearisoidaan, mutta tällöin eteenpäin suuntautuvat hypyt jouduttaisiin alustamaan paikanpitäjällä ja korjaamaan sitten, kun hypyn kohde tavataan ja pituus saadaan laskettua ('back patching'). Koska kaikki hypyt suuntautuvat eteenpäin joutuisimme tekemään tämän kaikille hypyille. Tämä olisi hankalaa Nanopass-kehyksen puitteissa ja imperatiivisena vaikeampi toteuttaa virheettömästi. Se ei välttämättä olisi edes nopeampaa, koska $2O(n) = O(n)$ ja kaksivaiheinen menetelmä hyötyy lineaarisesti muistia käsittelevänä enemmän prosessorin datavälimuistista.

5.3.6 Vakioiden taulukointi

Kuten luvussa 4.2 kerrottin, virtuaalikoneen käskyissä vakioargumentille on tilaa vain 7 bittiä olioviitteiden ('object reference'[21, luku 1]) ollessa osoittimen kokoisia eli 32 tai 64 bittiä. Niinpä vakioviitteet eivät suoraan mahdu käskyihin, vaan vakioargumentit on koodattu 7-bittisinä indekseinä käskyn sisältävän tavukoodifunktion vakiotaulukkoon.

Tässä käännösvaiheessa kunkin funktion koodi käydään vain läpi alusta loppuun, kerätään vakiot taulukkoon ja laitetaan niiden paikalle käskyihin vakion indeksi taulukossa. Kukin arvo laitetaan taulukkoon vain kerran, joten tämä järjestely myös säästää hieman tilaa.

5.3.7 Tavukoodin tuottaminen

Tässä vaiheessa koodin abstraktiotaso vastaa virtuaalikoneen abstraktiotasoa. Koodi tarvitsee enää viedä virtuaalikoneelle muodossa, jota se osaa lukea. Tähän päätettiin käyttää binäärimuotoisia tavukooditiedostoja, jotta kääntäjää ja virtuaalikonetta voidaan kehittää toisistaan riippumatta. Tiedostoformaatista kerrottiin tarkemmin luvussa 4.2. Tavukooditiedostojen luominen voidaan myös välttää esimerkiksi Unixin stdout–stdin-putkituksella.

Tavukoodin kirjoittaminen ja lukeminen on täysin suoraviivaista. Se ei tietyssä mielessä edes ole käännösvaihe, sillä koodin informaatiosisältö tai looginen rakenne ei muutu siinä lainkaan. Fyysinen rakenne toki muuttuu radikaalisti. Toisin olisi, mikäli tähän koodin generoimiseen olisi integroitu muita toimenpiteitä. Esimerkiksi assemblerien perustoiminnallisuuteen sisältyy usein ainakin hyppyetäisyyksien laskeminen ja hyppykäskyjen valinta hypyn pituuden perusteella[9].

6 TULOKSET JA JOHTOPÄÄTÖKSET

Työssä onnistuttiin muuttamaan tulkki kääntäjäksi, joka tuottaa virtuaalikoneen tavukooditiedostoja. Kääntäjässä on 18 käännösvaihetta, jotka käyttävät yhtätoista erilaista ohjelmaesitystä. Kaikille kymmenelle väliesitykselle on myös oma tulkki, mikä helpottaa käännösvaiheiden testaamista.

Käännösvaiheiden järjestyksen muuttaminen osoittautui työlääksi, koska järjestyksen lisäksi piti usein vaihtaa myös käännösvaiheiden syöte- ja tuoteväliesitykset eli muuttaa myös vaiheiden koodia. Staattisen tyypityksen tavoin Nanopass-kehys esti näin ohjelmointivirheitä, mutta tarkistusten hinta osoittautui korkeahkoksi. Vertailun vuoksi todettakoon, että muiden muassa kääntäjäkehys LLVM[41], Haskell-kääntäjä GHC[17] sekä Lua-virtuaalikone LuaJIT[26] tekevät suurimman osan käännösvaiheistaan ja erityisesti optimoinneistaan väliesitystä vaihtamatta. Optimointien optimaalinen järjestys kun on vaikea saavuttaa ilman jatkuvaa tyyppien vaihteluakin [29, luku 11.4].

Nanopass-kääntäjäkehys automatisoi kaavamaisia puiden läpikäyntejä varhaisissa käännösvaiheissa, mutta CPS-muunnoksen jälkeen siitä ei ollut vastaavaa hyötyä, koska väliesitys koostui lähinnä litteistä listoista syvien puiden sijaan. Nanopass-kehyksen kehitystä paljolti ohjannut Chez Scheme -kääntäjä [12] onkin aina käyttänyt enimmäkseen syviä puuväliesityksiä toisin kuin useimmat imperatiivisten kielten kääntäjät. Myöskään tämän työn CPS-väliesitysten esikuvina toimineet MLtonin SSA [50] tai Guilen CPS [7] eivät sisällä syviä lausekepuita.

Erinäisten käännösvaiheiden havaittiin vastaavan abstraktia tulkkausta ja niiden kehittämisen siksi helpottuvan tulkin lähdekoodin saatavuudesta. Erityisesti muuttujien käsittelyssä käännösvaiheet hyödynsivät luontevasti ympäristötietorakenteita tulkkien tapaan. CPS-muunnoksesta tuli jokseenkin helpommin ymmärrettävä C*K-tilakonetta mukaillen kuin tyypillisemmistä korkeamman asteen funktioita hyödyntävistä ratkaisuista.

Tulkkaamisella ja käännösvaiheilla on kuitenkin yleensä myös selkeitä eroja. Tulkki suorittaa ehtolauseista aina vain yhden haaran ja hyppää funktiokutsujen kohdalla kutsuttuihin funktioihin kun taas käännösvaiheet käyvät yleensä läpi koko väliesityksen rakenteellisella induktiolla käsitellen kaikki ehtolauseiden haarat ja hyppäämättä funktioiden määritelmiin. Jotkin optimoivat käännösvaiheet kuten funktiokutsujen laajentaminen (’inlining’) [1, luvut 6.1 ja 7], harva ehdollinen vakioiden levitys (’sparse conditional constant propagation’) [47] ja osittaisevaluointi [20] muistuttavat paljon enemmän tulkin toimintaa, mutta suurinta osaa näistä ei toteutettu.

Hienostuneet optimoinnit jätettiin pois, koska tavoitteena oli vain oikeellinen kääntäjä agressiivisesti optimoivan sijaan. Pari yksinkertaista optimoivaa käännösvaihetta toteutettiin, jotta muiden käännösvaiheiden ei tarvitsisi varoa tuottamasta lainkaan tehottomia koodinpätkiä. Tehokkaampien optimointien toteuttaminen on eräs selkeä jatkotutkimuskohde.

Toinen ilmeinen jatkotutkimuskohde eli fyysisen prosessorin hyödyntäminen ei vaatisi olennaisia muutoksia lähestymistapaan, vaan ratkaistavaksi tulisi lähinnä käskyjen valinta ja virtuaalikoneen palveluiden (kuten automaattinen muistinhallinta) muuttaminen ajoympäristökirjastoksi (’runtime library’).

Tulkin muuttaminen kääntäjäksi osoittautui hyödylliseksi tekniikaksi. Olin aiemminkin yrittänyt kirjoittaa kääntäjiä kehittelemilleni ohjelmointikielille, mutta toisin kuin tulkkien kanssa projektit jäivät aina jumiin. Tämän kandidaatintyön kanssa jäin jumiin vain raportin viimeistelyyn. Pitäisiköhän soveltaa diplomityössä ohjelmakoodin muuttamista kirjaksi[22]?

LÄHDELUETTELO

[1] A. Appel. *Compiling with Continuations*. Cambridge University Press, 2007. ISBN: 9781107393288.

[2] A. W. Appel ja J. Palsberg. *Modern Compiler Implementation in Java*. 2nd. New York, NY, USA: Cambridge University Press, 2003. ISBN: 052182060X.

[3] G. Blindell. *Instruction selection: Principles, methods, and applications*. Tammikuu 2016, 1–177.

[4] C. Chambers, D. Ungar ja E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. *LISP and Symbolic Computation* 4.3 (heinäkuu 1991), 243–281. ISSN: 1573-0557. DOI: 10.1007/BF01806108. URL: https://doi.org/10.1007/BF01806108.

[5] C. Click ja M. Paleczny. A Simple Graph-based Intermediate Representation. *SIGPLAN Not.* 30.3 (maaliskuu 1995), 35–49. ISSN: 0362-1340. DOI: 10.1145/202530.202534. URL: http://doi.acm.org/10.1145/202530.202534.

[6] K. Cooper, T. Harvey ja K. Kennedy. A Simple, Fast Dominance Algorithm (tammikuu 2006).

[7] *CPS in Guile*. Marraskuu 2018. URL: https://www.gnu.org/software/guile/manual/html_node/CPS-in-Guile.html.

[8] O. Danvy, K. Millikin ja L. R. Nielsen. On one-pass CPS transformations. English. *Journal of Functional Programming* 17.6 (2007), 793–812.

[9] N. G. Dickson. A Simple, Linear-Time Algorithm for x86 Jump Encoding. *CoRR* abs/0812.4973 (2008). arXiv: 0812.4973. URL: http://arxiv.org/abs/0812.4973.

[10] D. Dreyer. A Type System for Well-founded Recursion. *SIGPLAN Not.* 39.1 (tammikuu 2004), 293–305. ISSN: 0362-1340. DOI: 10.1145/982962.964026. URL: http://doi.acm.org/10.1145/982962.964026.

[11] R. K. Dybvig. Three Implementation Models for Scheme. UMI Order No. GAX87-22287. Tohtorinväitöskirja. Chapel Hill, NC, USA, 1987.

[12] R. K. Dybvig. The Development of Chez Scheme. *SIGPLAN Not.* 41.9 (syyskuu 2006), 1–12. ISSN: 0362-1340. DOI: 10.1145/1160074.1159805. URL: http://doi.acm.org/10.1145/1160074.1159805.

[13] M. A. Ertl ja et al. *The structure and performance of efficient interpreters*. 2003.

[14] M. Felleisen, R. B. Findler ja M. Flatt. *Semantics Engineering with PLT Redex*. 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.

[15] M. Gabbrielli ja S. Martini. *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer London, 2010. ISBN: 9781848829145. URL: https://books.google.fi/books?id=U-uBhJCRw3QC.

[16] S. Hack. Register Allocation for Programs in SSA Form. Tohtorinväitöskirja. Universität Karlsruhe, lokakuu 2007. ISBN: 978-3-86644-180-4. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532.

[17] *Home – The Glasgow Haskell Compiler*. [online], https://www.haskell.org/ghc/. Elokuu 2018.

[18] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag New York, Inc., 1985, 190–203. ISBN: 3-387-15975-4. URL: http://dl.acm.org/citation.cfm?id=5280.5292.

[19] M. P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Berlin, Heidelberg: Springer-Verlag, 1995, 97–136. ISBN: 3-540-59451-5. URL: http://dl.acm.org/citation.cfm?id=647698.734150.

[20] N. D. Jones, C. K. Gomard ja P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 0-13-020249-5.

[21] R. Jones, A. Hosking ja E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC, 2011. ISBN: 1420082795, 9781420082791.

[22] D. E. Knuth. Literate Programming. *Comput. J.* 27.2 (toukokuu 1984), 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. URL: http://dx.doi.org/10.1093/comjnl/27.2.97.

[23] R. Leißa, M. Köster ja S. Hack. A graph-based higher-order intermediate representation. English. IEEE Computer Society, 2015, 202–212. ISBN: 9781479981618;1479981613;

[24] T. Lengauer ja R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1.1 (tammikuu 1979), 121–141. ISSN: 0164-0925. DOI: 10.1145/357062.357071. URL: http://doi.acm.org/10.1145/357062.357071.

[25] T. Lindholm, F. Yellin, G. Bracha ja A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390590X, 9780133905908.

[26] *LuaJIT*. [online], http://luajit.org/luajit.html. Elokuu 2018.

[27] K.-H. Man. *A No-Frills Introduction to Lua 5.1 VM Instructions*. 2006. URL: http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf.

[28] L. Maranget. Compiling Pattern Matching to Good Decision Trees. *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ML '08. Victoria, BC, Canada: ACM, 2008, 35–46. ISBN: 978-1-60558-062-3. DOI: 10.1145/1411304.1411311. URL: http://doi.acm.org/10.1145/1411304.1411311.

[29] S. S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[30] S. Owens. *Parser Tools: lex and yacc-style Parsing*. [online], https://docs.racket-lang.org/parser-tools/. Marraskuu 2018.

[31] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1st. Pragmatic Bookshelf, 2009. ISBN: 193435645X, 9781934356456.

[32] M. Poletto ja V. Sarkar. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21.5 (syyskuu 1999), 895–913. ISSN: 0164-0925. DOI: 10.1145/330249.330250. URL: http://doi.acm.org/10.1145/330249.330250.

[33] A. Rigo ja S. Pedroni. PyPy's Approach to Virtual Machine Construction. *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, 944–953. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176753. URL: http://doi.acm.org/10.1145/1176617.1176753.

[34] A. Sabry ja M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation* 6.3 (marraskuu 1993), 289–360. ISSN: 1573-0557. DOI: 10.1007/BF01019462. URL: https://doi.org/10.1007/BF01019462.

[35] D. Seal. *ARM Architecture Reference Manual*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201737191.

[36] T. Shanley. *X86 Instruction Set Architecture*. Mindshare Press, 2010. ISBN: 0977087859, 9780977087853.

[37] A. Shinn, J. Cowan ja A. A. Gleckler. *Revised^7 Report on the Algorithmic Language Scheme*. 2013.

[38] O. G. Shivers. Control-flow Analysis of Higher-order Languages of Taming Lambda. UMI Order No. GAX91-26964. Tohtorinväitöskirja. Pittsburgh, PA, USA, 1991.

[39] G. L. Steele Jr. *Rabbit: A Compiler for Scheme*. Tekninen raportti. Cambridge, MA, USA, 1978.

[40] *Swift Intermediate Language (SIL)*. Marraskuu 2018. URL: https://github.com/apple/swift/blob/master/docs/SIL.rst.

[41] *The LLVM Compiler Infrastructure*. [online], http://llvm.org/. Elokuu 2018.

[42] *The Python programming language*. Marraskuu 2018. URL: https://github.com/python/cpython.

[43] *The Rust Programming Language*. [online], https://www.rust-lang.org. Elokuu 2018.

[44] V. F. Turchin. The Concept of a Supercompiler. *ACM Trans. Program. Lang. Syst.* 8.3 (kesäkuu 1986), 292–325. ISSN: 0164-0925. DOI: 10.1145/5956.5957. URL: http://doi.acm.org/10.1145/5956.5957.

[45] O. Waddell ja R. K. Dybvig. Fast and effective procedure inlining. *Static Analysis*. Toim. P. Van Hentenryck. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, 35–52. ISBN: 978-3-540-69576-9.

[46] Y. Wang ja R. K. Dybvig. Register Allocation By Model Transformer Semantics. English (2012).

[47] M. N. Wegman ja F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13.2 (huhtikuu 1991), 181–210. ISSN: 0164-0925. DOI: 10.1145/103135.103136. URL: http://doi.acm.org/10.1145/103135.103136.

[48] R. Wilhelm ja D. Maurer. *Compiler Design*. International computer science series. Addison-Wesley Publishing Company, 1995. ISBN: 9780201422900. URL: https://books.google.fi/books?id=oa5QAAAAMAAJ.

[49] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon ja M. Wolczko. One VM to Rule Them All. *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581. URL: http://doi.acm.org/10.1145/2509578.2509581.

[50] L. Ziarek, S. Weeks ja S. Jagannathan. Flattening tuples in an SSA intermediate representation. *Higher-Order and Symbolic Computation* 21.3 (syyskuu 2008), 333–358. ISSN: 1573-0557. DOI: 10.1007/s10990-008-9035-3. URL: https://doi.org/10.1007/s10990-008-9035-3.

A COMPLOTIN KIELIOPPI

Syntaksi määritellään yleisesti käytössä olevilla merkintätavoilla. Leksikaalinen syntaksi määritellään alkionimien (’token’) säännöllisinä kielioppeina, jotka ilmaistaan säännöllisinä lausekkeina [2, s. 18–21]. Lausekesyntaksi annetaan kontekstivapaana kielioppina [2, s. 38–41].

Complotissa ei ole lainkaan varattuja avainsanoja. Infiksioperaattorit ovat ML:n tapaan vain funktiokutsuja, joten ohjelmoija voi määrittää ne uudelleen tai lisätä omia operaattoreitaan. Operaattorien sidontajärjestys ja assosiatiivisuus on kuitenkin sidottu kieliopissa niiden ensimmäisiin merkkeihin kuten Scalassa, joten jäsentäjän ei tarvitse ymmärtää jäsentämänsä koodin semantiikasta edes näiden operaattorien ominaisuuksien määrittelyjä.

A.0.1 Leksikaalinen

Complotin alkionimien kieliopit on annettu kuvassa A.1 säännöllisinä lausekkeina. Näiden kielioppien symboleina toimivat yksittäiset merkit. Alkionimi tunnistetaan yksikäsitteisesti jo ensimmäisestä merkistä, joten rivien järjestyksellä ei ole merkitystä.

A.0.2 Kontekstivapaa

Complotin kontekstivapaa kielioppi on annettu kuvassa A.2. Tämän kieliopin symboleina toimivat edellisen luvun alkionimien ilmentymät eli tekstialkiot (’token’ tai ’lexeme’)

$$
\begin{aligned}
LEX\_IDENT &\rightarrow [\_A-Za-z]^{+} \\
DYN\_IDENT &\rightarrow \$\ [\_A-Za-z]^{*} \\
STRING &\rightarrow \text{”}\ [\hat{}\text{”}]*\ \text{”} \\
INT &\rightarrow (\prime+\prime \mid -)?\ [0-9]^{+} \\
CHAR &\rightarrow \text{'}\ [\hat{}\text{'}]\ \text{'} \\
WHITESPACE &\rightarrow \backslash s^{+} \\
LINE\_COMMENT &\rightarrow \#\ [\hat{}\backslash n]* \\
\\
constituent &\rightarrow [\hat{}\backslash s\backslash(\backslash)[]\{\}\text{”}\text{‘};;]
\end{aligned}
\qquad
\begin{aligned}
OP_1 &\rightarrow \backslash|\ constituent^{+} \\
OP_2 &\rightarrow \hat{}\ constituent^{*} \\
OP_3 &\rightarrow \&\ constituent^{*} \\
OP_4 &\rightarrow (=|!)\ constituent^{+} \\
OP_5 &\rightarrow (<|>)\ constituent^{*} \\
OP_6 &\rightarrow (\backslash+|-)\ constituent^{*} \\
OP_7 &\rightarrow (\backslash*|/|\%)\ constituent^{*}
\end{aligned}
$$

Kuva A.1. *Complotin leksikaalinen syntaksi*

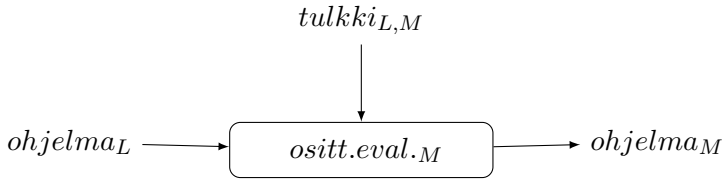
$$
\begin{aligned}
prog &\rightarrow body \\
body &\rightarrow (stmt\ '{;}')^*\ expr \\
stmt &\rightarrow id\ '{=}'\ expr \\
&\mid expr \\
expr &\rightarrow infix_1 \\
infix_p &\rightarrow infix_p\ OP_p\ infix_{p+1} \\
&\mid infix_{p+1} \\
infix_7 &\rightarrow infix_7\ OP_7\ call \\
&\mid call \\
call &\rightarrow simple^+ \\
simple &\rightarrow '('\ expr\ ')' \\
&\mid '\{'\ (case\ '{;}')^*\ case\ '\}' \\
&\mid '\{'\ body\ '\}' \\
&\mid id \\
&\mid datum
\end{aligned}
\qquad
\begin{aligned}
case &\rightarrow id^+\ '|'\ expr\ '{=>}'\ expr \\
datum &\rightarrow '('\ exprs\ ')' \\
&\mid STRING \\
&\mid INT \\
&\mid CHAR \\
id &\rightarrow LEX\_IDENT \\
&\mid DYN\_IDENT \\
&\mid '('\ OP_{[1,7]}\ ')' \\
exprs &\rightarrow \epsilon \\
&\mid expr\ ',' \\
&\mid (expr\ ',')^+\ expr
\end{aligned}
$$

Kuva A.2. *Complotin kontekstivapaa syntaksi*

[2, s. 40]. Infiksi-operaattorien yhteydessä alaindeksi p on mikä tahansa sidontatiukkuus (’precedence’) 1-7 eli säännöt $infix_p$ ja $infix_7$ muodostavat ‘makrolaajennettavan’ säännön.

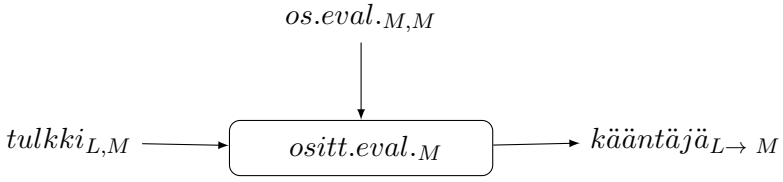
B FUTAMURAN PROJEKTIOT

Mikäli osittaisevaluaattorin käsiteltäväksi ohjelmaksi annetaan kielellä M kirjoitettu kielen L tulkki ja staattisiksi syötteiksi tulkattava L-kielinen ohjelma, tuottaa se kielellä M kirjoitetun tulkin, joka osaa tulkata vain yhden ohjelman eli toisin sanoen tulkattava ohjelma tulee käännetyksi kielelle M (kuva B.1). Tätä kutsutaan **Futamuran ensimmäiseksi projektioksi**. [20, s. 13]



Kuva B.1. *Futamuran ensimmäinen projektio*

Futamuran ensimmäinen projektio mahdollistaa tulkin ja osittaisevaluaattorin yhdistelmän käyttämisen kääntäjänä. **Futamuran toisen projektion** mukaan taas kielen M osittaisevaluaattoria voidaan käyttää myös kielellä M kirjoitetun osittaisevaluaattorin erikoistamiseen kielellä M kirjoitetun kielen L tulkin suhteen (kuva B.2). Tällöin saadaan kuvasta B.1 versio, jossa tulkkia ei enää tarvitse antaa syötteenä eli kääntäjä kielestä L kielelle M. [20, s. 13]



Kuva B.2. *Futamuran toinen projektio*

Futamuran kolmas projektio menee vielä yhden askeleen pidemmälle eli erikoistaa osittaisevaluaattorin sen itseensä suhteen tuottaen kuvasta B.2 version, jossa osittaisevaluaattoria ei enää tarvitse antaa syötteenä eli ohjelman, joka muuttaa minkä tahansa tulkin kääntäjäksi. [20, s. 14]