Efektijärjestelmät

Lauri Alanko

Helsinki 6.5.2003

Tyyppiteoria ja ohjelmointikielet -seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Tyyppijärjestelmät tarkastelevat ohjelmien syntaktista rakennetta ja tekevät niiden pohjalta tarkistuksia ja päätelmiä ohjelman suoritusaikaisista ominaisuuksista. Tavanomaiset tyyppijärjestelmät keskittyvät yhteen ominaisuuteen: minkälaisia arvoja missäkin ohjelman osassa liikkuu. Tyyppijärjestelmä tekee konservatiivisen arvion arvoista, joita kukin muuttuja tai lauseke voi suorituksen aikana saada, ja tätä arviota kutsutaan tyypiksi.

Ohjelmilla on kuitenkin myös muita kiinnostavia ominaisuuksia. Monissa kielissä ohjelmat voivat suorituksensa aikana aiheuttaa sivuvaikutuksia (engl. *side effects*) jotka monimutkaistavat kielen semantiikkaa. *Efektijärjestelmä* on tyyppijärjestelmä, joka tekee tarkistuksia ja päätelmiä sivuvaikutuksista, joita kukin ohjelman osa voi aiheuttaa.

Efektijärjestelmillä on samat sovellustavat kuin tyyppijärjestelmillä yleensäkin. Ne parantavat ohjelman turvallisuutta, koska jo käännösaikaan voidaan tarkistaa, että ohjelma noudattaa annettuja spesifikaatioita. Lisäksi kääntäjä voi käyttää tietoa sivuvaikutuksista omiin optimointeihinsa. Jälkimmäinen tuntuu tällä hetkellä olevan keskeisin efektijärjestelmien tutkimuskohde.

Tässä kirjoituksessa käsitellään efektijärjestelmien toimintaa sekä joitain sovellusalueita, joissa efektejä voidaan hyödyntää. Painotus on staattisessa semantiikassa eli tyypityssäännöissä. Dynaaminen semantiikka (suoritussäännöt) sekä tyyppisääntöjen eheystodistukset on tässä sivuutettu.

Luvussa 2 esitellään peruskäsitteitä ja efektijärjestelmien yleisiä tyyppisääntöjä. Luvussa 3 näytetään, miten näitä sääntöjä voidaan soveltaa yksinkertaisessa pysähtyvyysanalyysissä ja luvussa 4 tehdään efekteistä joustavampia *aliefektien* avulla. Luvussa 5 analysoidaan poikkeuskäsittelyä ja luvussa 6 näytetään, kuinka *polymorfismi* tekee siitä yleiskäyttöisempää. Luvuissa 7 ja 8 käsitellään viitemuuttujien hallintaa ja luku 9 päättää kirjoituksen.

Tekstissä pohjaudutaan pitkälti Benjamin Piercen kirjan [Pie02] eri osiin ja pyritään seuraamaan sen esityskäytäntöjä.

2 Perusteet

2.1 Sivuvaikutukset ja efektijärjestelmät

Monet λ-kalkyylin variantit ja jotkin ohjelmointikieletkin ovat ”puhtaita”: lausekkeen suorittamisella ei ole muuta merkitystä kuin sen palauttama arvo, ja tämä palautettu arvo riippuu ainoastaan lausekkeen rakenteesta. Monet kiinnostavat ohjelmointikielten ominaisuudet kuitenkin rikkovat tätä sääntöä: sijoitukset, poikkeukset ja kaikenlainen vuorovaikutus ympäröivän maailman kanssa ovat ominaisuuksia, joissa lausekkeen suoritus joko vaikuttaa muuhunkin kuin paluuarvoon, tai riippuu jostain

$$
\begin{array}{lll}
t & ::= & x \\
 & & t_1 t_2 \\
 & & \lambda x : T.t \\
T & ::= & T_1 \xrightarrow{\varepsilon} T_2 \\
\varepsilon & ::= & \emptyset \\
 & & \varepsilon_1 \cup\ e_2
\end{array}
$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T, \emptyset} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \xrightarrow{\varepsilon}_3 T_2, \varepsilon_1 \qquad \Gamma \vdash t_2 : T_1, \varepsilon_2}{\Gamma \vdash t_1 t_2 : T_2, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \qquad \text{(T-APP)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2, \varepsilon}{\Gamma \vdash \lambda x : T_1.t : T_1 \xrightarrow{\varepsilon} T_2, \emptyset} \qquad \text{(T-ABS)}$$

Kuva 1: Kalkyyli $\lambda_{\rightarrow,\varepsilon}$

muustakin kuin lausekkeen rakenteesta.

Sivuvaikutusten vaikutusalue on usein arvaamattoman suuri: muuttujaan voidaan sijoittaa yhdessä osassa ohjelmaa, ja sitä voidaan lukea jossain aivan muualla, eikä osien välillä ole mitään ilmeistä yhteyttä. Tämä hankaloittaa ohjelman toiminnan ymmärtämistä.

Efektijärjestelmä auttaa tässä. Se analysoi ohjelman sivuvaikutuksia ja tarvittaessa rajoittaa niiden käyttöä. Käytännössä tämä tapahtuu laajentamalla tyyppijärjestelmän tyypitysrelaatiota niin, että tyypityslause kertoo paitsi sen, millaisen arvon lausekkeen suorittaminen voi tuottaa, myös sen, mitä sivuvaikutuksia suorituksen aikana voi tapahtua.

2.2 Peruskalkyyli

Tutkitaan ensin yksinkertaisesti tyypitettyä λ-kalkyyliä efekteillä höystettynä ja kutsutaan sitä vaikka nimellä $\lambda_{\rightarrow,\varepsilon}$. Sen syntaksi ja tyypityssäännöt löytyvät kuvasta 1. Tämä kieli on itsessään tietysti hyödytön, sillä siinä ei ole yhtään hyvintyypitettyä termiä eikä sivuvaikutuksia joita analysoida, mutta se muodostaa yhteisen pohjan laajennuksille.

Kielen termit ovat samanlaisia kuin $\lambda_{\rightarrow}$:ssakin, mutta tyyppejä on laajennettu: funktiotyypissä $T_1 \xrightarrow{\varepsilon} T_2$ kerrotaan paitsi argumentin ja paluuarvon tyyppi, myös sen *efekti*, arvio sivuvaikutuksista, joita funktion kutsuminen voi tuottaa.

Efekti on konservatiivinen approksimaatio mahdollisista sivuvaikutuksista aivan samoin kuin tyyppi on konservatiivinen approksimaatio mahdollisista arvoista. Käytännössä ε on äärellinen joukko primitiivisivuvaikutusten lajeja. Puhtaassa $\lambda_{\rightarrow,\varepsilon}$:ssa ei näitä primitiivisivuvaikutuksia ole, joten efektit tässä vaiheessa ovat vain tyhjiä joukkoja.

Tiukan formaalisti ajatellen efektitermeille pitäisi määritellä omat yhtälösäännöt ja operaatiot[1], mutta tässä tyydymme tulkitsemaan ne joukoiksi ja käytämme jatkossa tavallisia joukko-operaatioita.

Tyypityslauseet ovat siis muotoa $\Gamma \vdash t : T, \varepsilon$. Tässä ε kertoo, minkälaisia sivuvaiku-

[1] Tarkalleen ottaen $\cup$-operaattorin tulee olla symmetrinen, assosiatiivinen ja idempotentti, ja $\emptyset$:n tulee olla sen neutraalialkio.

$$
\begin{array}{rcl}
t & ::= & \ldots \\
 & & \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \\
 & & \texttt{iszero} \mid \texttt{succ} \mid \texttt{prec} \\
 & & b \\
 & & i \\
b & ::= & \texttt{true} \mid \texttt{false} \\
i & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
T & ::= & \ldots \\
 & & \texttt{Bool} \\
 & & \texttt{Nat}
\end{array}
$$

$$\Gamma \vdash b : \texttt{Bool}, \emptyset \qquad \text{(T-Bool)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool}, \varepsilon_1 \quad \Gamma \vdash t_2 : T, \varepsilon_2 \quad \Gamma \vdash t_3 : T, \varepsilon_3}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \qquad \text{(T-If)}$$

$$\Gamma \vdash i : \texttt{Nat}, \emptyset \qquad \text{(T-Nat)}$$

$$\Gamma \vdash \texttt{iszero} : \texttt{Nat} \xrightarrow{\emptyset} \texttt{Bool}, \emptyset \qquad \text{(T-IsZero)}$$

$$\Gamma \vdash \texttt{succ} : \texttt{Nat} \xrightarrow{\emptyset} \texttt{Nat}, \emptyset \qquad \text{(T-Succ)}$$

$$\Gamma \vdash \texttt{prec} : \texttt{Nat} \xrightarrow{\emptyset} \texttt{Nat}, \emptyset \qquad \text{(T-Prec)}$$

Kuva 2: Luvut ja totuusarvot

tuksia termin t suorittaminen voi aiheuttaa.

Sääntö T-Var kertoo, että muuttujaan viittaaminen ei aiheuta sivuvaikutuksia.

Sääntö T-App kertoo, että funktiokutsu aiheuttaa ne sivuvaikutukset, joita itse funktiotermiä suoritettaessa tapahtuu (ε_1), ne, joita argumenttia suoritettaessa tapahtuu (ε_2), sekä ne, joita itse funktiokutsun aikana tapahtuu (ε_3). Tämä onkin kielen operationaalisen semantiikan mukaista: ensin funktiolauseke redusoidaan λ-termiksi, sitten argumenttilauseke redusoidaan arvoksi, ja sitten λ-termin vartalo redusoidaan, kun sen parametri on korvattu tällä arvolla. Jokaisessa vaiheessa voi tapahtua sivuvaikutuksia.

Sääntö T-Abs kertoo, miten λ-termit tyypitetään: muodostettu funktio voi kutsuttaessa aiheuttaa niitä sivuvaikutuksia, joita λ-termin vartalon suoritus voi saada aikaan. Koska λ-termi vain rakentaa funktion, on se itse puhtaan sivuvaikutukseton.

2.3 Luvut ja totuusarvot

Kuvassa 2 on laajennus, joka tuo kieleen kokonaisluvut ja totuusarvot, jotta kielellä voisi tehdäkin jotain. Säännöt ovat melko ilmeiset: literaalivakio ei tuota sivuvaikutuksia. Ainoa monimutkaisempi sääntö on T-If. Se kertoo, että ehtolauseen sivuvaikutukset ovat yhdiste ehdon *ja* kummankin haaron sivuvaikutuksista. Tosiasiassa tietysti kerralla aiheutuvat vain jomman kumman haaran sivuvaikutukset, mutta tätä efektijärjestelmämme ei pysty kuvaamaan.

$$
\begin{array}{lll}
t & ::= & ... \\
 & & \texttt{rec}\ fx.t \\
\varepsilon & ::= & ... \\
 & & \bot
\end{array}
\qquad
\frac{\Gamma, f : T_1 \xrightarrow{\varepsilon \cup \bot} T_2, x : T_1 \vdash t : T_2, \varepsilon}{\Gamma \vdash \texttt{rec}\ fx.t : T_1 \xrightarrow{\varepsilon} T_2, \emptyset} \quad (\text{T-Rec})
$$

Kuva 3: Vapaa rekursio ja pysähtyvyysanalyysi

3 Pysähtyvyysanalyysi

3.1 Efektijärjestelmä rekursiolle

Kaikkein yleisin ohjelmointikielissä esiintyvä sivuvaikutus, vaikka sitä ei yleensä ajatella sellaisena, on pysähtymättömyys. Se kuitenkin täyttää sivuvaikutuksen ehdot: lauseke, jonka suoritus ei pääty, ei vaikuta ainoastaan paluuarvoonsa, vaan koko ohjelman toimintaan. Yksikin päättymättömyys missä hyvänsä kohdassa ohjelmaa tekee koko ohjelman suorituksesta päättymättömän. Kyseessä on siis globaali efekti.

Pysähtymisongelma on tietysti ratkeamaton, joten täydellinen pysähtyvyysanalyysi on mahdotonta. Silti on mahdollista tehdä konservatiivisia approksimaatioita niin, että tyyppijärjestelmä kertoo edes joistain lausekkeista, että niiden suorituksen tiedetään varmasti pysähtyvän. Niin kauan kuin kielessä ei ole rekursiivisia tyyppejä, ainoa keino luoda termi, jonka suoritus ei pysähdy, on käyttää vapaata rekursiota erityisen syntaktisen rakenteen avulla. Tähän efektipohjainen pysähtyvyysanalyysi perustuu.

Kuvassa 3 esitetään yksinkertainen laajennus, joka tuo kieleen `rec`-konstruktion rekursiivisten funktioiden määrittelemiseen. Lauseke $\texttt{rec}fx.t$ määrittelee funktion, jonka argumentti on x ja vartalo t, mutta vartalossa muuttuja f on sidottu funktioon itseensä. Se on otettu käytännön syistä tässä primitiiviksi, mutta sen voi määritellä myös `fix`-operaattorilla[Pie02, 11.11]:

$$\texttt{rec}\ fx.t = \texttt{fix}\ \lambda f.\lambda x.t$$

Efektijärjestelmää laajennetaan ottamalla mukaan uusi efekti $\bot$, joka kuvaa pysähtymättömyyttä. Sääntö T-Rec kertoo, että rekursiivisesti määritelty funktio voi kutsuttaessa aiheuttaa efektit ε jos sen vartalon suoritus aiheuttaa efektit ε *kun oletetaan, että rekursiiviset kutsut voivat olla pysähtymättä.*

Täten siis ε:n on pakko sisältää $\bot$, jos vartalossa tehdään yhtään suoraa rekursiivista kutsua. Mutta jos f:ään ei vartalossa viitata, sääntö toimii aivan kuten T-Abs, joten `rec`-rakenteella voisi hyvin korvata λ:n.

3.2 Soveltaminen kääntäjässä

Tällainen karkea pysähtyvyysanalyysi on ohjelmoijalle hyödytöntä. Tosimaailman ohjelmat käyttävät kuitenkin vapaata rekursiota, joten efektijärjestelmä kertoisi ohjelmoijalle melkein kaikesta, että se ei välttämättä pysähdy. Eikä auttaisi mitään, vaikka ohjelmoija voisikin tunnistaa "ongelmakohdat", koska niille ei kuitenkaan mitään voi tehdä.

Kääntäjälle tämä vähäinenkin tieto on kuitenkin hyödyllistä. Kun kielessä on sivuvaikutuksia, monien optimointien laillisuus riippuu siitä, millaisia sivuvaikutuksia optimoitavan lausekkeen osilla on. Eräs kääntäjien usein hyödyntämä ekvivalenssi on:

$$\frac{\Gamma \vdash t_1 : T_1, \emptyset \qquad \Gamma, x : T_1 \vdash t_2 : T_2, \varepsilon}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 \cong t_2[x/t_1] : T_2, \varepsilon}$$

Toisin sanoen `let`-sidotun muuttujan voi korvata määrittelyllään[2], *jos* siinä ei ole sivuvaikutuksia. Tästä yhtäläisyydestä on hyötyä kääntäjälle molempiin suuntiin. Vasemmalta oikealle kääntäjä voi laventaa (inline) muuttujan, ja näin paljastaa uusia optimointimahdollisuuksia. Oikealta vasemmalle kääntäjä taas voi säästää turhaa työtä varmistamalla, että monta kertaa esiintyvä lauseke lasketaankin vain kerran.

Oleellista on, että tämän ekvivalenssin oikeellisuuden varmistamiseen tarvitaan efektisääntöjä: jos t_1:n suoritus voisi jäädä jumiin, yhtäläisyys ei enää pätisikään.

4 Aliefektit

Nyt järjestelmässä on kahdenlaisia lausekkeita: puhtaita (efekti $\emptyset$) ja epäpuhtaita (efekti jotain muuta, esim $\perp$). Järjestelmä kuitenkin pitää ne täysin erillisinä. Tämä on harmillista, sillä monessa yhteydessä halutaan, että sivuvaikutuksetonta lauseketta voidaan käyttää siellä, missä sivuvaikutuksetkin sallitaan. Tämä saadaan aikaan yksinkertaisella säännöllä: lausekkeen efektiin voidaan vapaasti lisätä muita efektejä. Formaalisti tämä *aliefektisääntö* on seuraava:

$$\frac{\Gamma \vdash t : T, \varepsilon \qquad \varepsilon \subset \varepsilon'}{\Gamma \vdash t : T, \varepsilon'} \tag{E-Sub}$$

Tämä ei kuitenkaan ole vielä itsessään kovin hyödyllistä. Nyt voidaan luoda funktio tyyppiä $T_1 \xrightarrow{\perp} T_2$, joka ei kuitenkaan sisällä `rec`-lauseketta. Mutta vielä ei voida käyttää puhdasta funktiota tyyppiä $T_1 \xrightarrow{\emptyset} T_2$ argumenttina funktiolle, joka parametrityyppi on $T_1 \xrightarrow{\perp} T_2$. Koska kyseessä on tyyppien eikä efektien yhteensopivuus, tarvitaan tähän selvästi jonkinlaista alityypitystä.

Alityypityksen perusteet on esitetty Piercen kirjassa [Pie02, luku 15]. Kun mukaan tuodaan efektijärjestelmä, tarvitsee vain yhden uuden alityypityssäännön:

$$\frac{\varepsilon \subset \varepsilon'}{T_1 \xrightarrow{\varepsilon} T_2 \leq T_1 \xrightarrow{\varepsilon}' T_2} \tag{S-Eff}$$

[2]Tietysti varmistaen, että t_1:n vapaita muuttujia ei sidota

$$t \quad ::= \quad ... \quad \texttt{raise}\ E \quad \texttt{try}\ t_1\ \texttt{with}\ E \longrightarrow t_2$$
$$\varepsilon \quad ::= \quad ... \quad E$$

$$\Gamma \vdash \texttt{raise}\ E : T, E \qquad \text{(T-RAISE)}$$

$$\frac{\Gamma \vdash t_1 : T, \varepsilon_1 \qquad \Gamma \vdash t_2 : T, \varepsilon_2}{\Gamma \vdash \texttt{try}\ t_1\ \texttt{with}\ E \longrightarrow t_2 : T, \varepsilon_1 \setminus E \cup \varepsilon_2} \qquad \text{(T-TRY)}$$

Kuva 4: Poikkeukset

Näin saavutetaan haluttu intuitio: funktiotyyppi $T_1 \xrightarrow{\varepsilon} T_2$ tarkoittaa funktiota, joka voi aiheuttaa *korkeintaan* efektit ε, mutta mahdollisesti vähemmänkin.

5 Poikkeukset

Poikkeuskäsittely on nykyään varsin suosittu ominaisuus ohjelmointikielissä. Se on myös käytännössä ainoa alue, jolla jotain efektijärjestelmiä muistuttavaa on päässyt valtavirran ohjelmointikieliin.

Kuvassa 4 on laajennus joka tuo kieleen yksinkertaiset poikkeukset. Termikieleen tuodaan kaksi uutta rakennetta. Lauseke `raise` E nostaa poikkeuksen, joka siirtää kontrollin lähimmän ympäröivän `try` ... `with` $E \longrightarrow$ ... -lausekkeen käsittelijähaaraan (missä E on sama poikkeustunniste). Piercen kirja [Pie02, luku 14] käsittelee poikkeusten semantiikkaa tarkemmin.

Pieni ero kirjan esitykseen tässä on, että poikkeukset eivät ole ajonaikaisia arvoja eikä niillä ole omaa tyyppiä, vaan poikkeustunniste E saa arvoja jostain (tässä määrittelemättömästä) joukosta, ja heitettävän poikkeuksen tunniste täytyy mainita `raise`-lausekkeessa.

Nyt heitettäviä poikkeuksia voidaan analysoida staattisesti efektijärjestelmän avulla. Lisätään poikkeustunnisteet efekteihin: jos lausekkeen efekti sisältää E:n, tarkoittaa se, että lausekkeen suoritus voi nostaa poikkeuksen E siitä ulos.

Sääntö T-RAISE kertoo yksinkertaisesti, että poikkeuksen E nosto on lause, joka voi aiheuttaa poikkeuksen E (eikä muuta). Koska nostolauseke ei koskaan palaa kutsujalleen, voidaan sen tulkita olevan minkä tyyppinen hyvänsä.

Sääntö T-TRY puolestaan kertoo, että lause, joka käsittelee poikkeuksen, ei voi itsessään enää aiheuttaa sitä poikkeusta, ellei käsittelijä nosta sitä uudestaan (tämän takia *ensin* ε_1:stä poistetaan E, ja vasta sitten lisätään ε_2.

Esitetty efektijärjestelmä on tarkoituksellisen yksinkertainen. Järjestelmää voi laajentaa käsittelemään todellisissa ohjelmointikielissä esiintyviä monimutkaisempia poikkeusominaisuuksia[PL99].

6 Polymorfismi

Edellä esitetty efektijärjestelmä poikkeuksille muistuttaa pitkälti esim. Javan poikkeustarkistuksia[3]. Javan poikkeustarkistukset ovat kuitenkin harmillisen rajoitettuja: metodin poikkeukset eivät voi riippua sen argumenteista. Tällaiselle ominaisuudelle olisi kuitenkin käyttöä, ja se onkin yksinkertaista lisätä efektijärjestelmään.

Tarkastellaan lauseketta $\texttt{double} = \lambda f : \texttt{Nat} \xrightarrow{?} \texttt{Nat}.\lambda x.\texttt{Nat}.f(fx)$. Tämän on tarkoitus olla yleiskäyttöinen funktio, joka soveltaa ensimmäistä argumenttiaan kahdesti toiseen argumenttiinsa. Mikä efektin ? tulisi olla? Jos se on $\emptyset$, niin `double`:a voi soveltaa vain puhtaisiin funktioihin, mikä rajoittaa sen käyttökelpoisuutta. Jos taas efekti on suurempi, niin sallitut argumentit lisääntyvät, mutta `double` fx:n efektikin on suurempi, vaikka argumentti olisikin puhdas funktio. Tämäkään ei käy päinsä.

Ratkaisu on ottaa käyttöön *efektipolymorfismi* ja tehdä tyypeistä parametroituvia efektien suhteen. Teknisesti mekanismi on pitkälti sama kuin `let`-polymorfismissa [Pie02, 22.7]. Ainoa ero on siinä, että vapaiden tyyppimuuttujien sijaan kvantifioidaankin vapaiden efektimuuttujien ylitse. Jos käytössä on pelkkä efektipolymorfismi, `double`:lle saataisiin tyyppiskeema:

$$\forall a.(\texttt{Nat} \xrightarrow{a} \texttt{Nat}) \xrightarrow{\emptyset} \texttt{Nat} \xrightarrow{a} \texttt{Nat}$$

Toisaalta, jos myös tyypit ovat polymorfisia, voidaan λ-lausekkeiden tyyppiannotaatiot unohtaa kokonaan ($\texttt{double} = \lambda f.\lambda x.f(fx)$), ja tällöin saataisiin tyyppiskeema:

$$\forall a.\forall X.(X \xrightarrow{a} X) \xrightarrow{\emptyset} X \xrightarrow{a} X$$

Tämä saadaan sitten ilmentymiä korvaamalla a erilaisilla efekteillä ja X erilaisilla tyypeillä. Näin saavutetaan tavoite: funktion efekti voi riippua sen argumenttien efekteistä.

Efektipolymorfismilla saadaan myös aikaan sama kuin efektipohjaisella alityypityksellä: puhdasta funktiota voidaan käyttää myös siellä missä kaivataan epäpuhdasta. Tämä saadaan aikaan antamalla puhtaille lausekkeille efektiksi $\emptyset$:n sijasta a, missä a on vapaa efektimuuttuja. Tällöin puhtaan funktion tyypiksi saadaan tyyppiskeema:

$$\forall a.T_1 \xrightarrow{a} T_2$$

Tästä sitten saadaan sekä puhtaita että epäpuhtaita instansseja korvaamalla a eri efekteillä.

[3] Paitsi että Javassa `throw`-lauseen efekti riippuu heitettävän poikkeusolion tyypistä. Tämä olisi yksinkertaista lisätä efektijärjestelmäänkin.

$$
\begin{array}{lcl}
t & ::= & \dots \\
 & & \texttt{ref}\ t \\
 & & !t \\
 & & t_1 \texttt{:=} t_2 \\
T & ::= & \dots \\
 & & \texttt{Ref}\ T \\
\varepsilon & ::= & \dots \\
 & & \texttt{a} \mid \texttt{r} \mid \texttt{w}
\end{array}
$$

$$\frac{\Gamma \vdash t : T, \varepsilon}{\Gamma \vdash \texttt{ref}\ t : \texttt{Ref}\ T, \varepsilon \cup \texttt{a}} \quad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash t : \texttt{Ref}\ T, \varepsilon}{\Gamma \vdash !t : T, \varepsilon \cup \texttt{r}} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T, \varepsilon_1 \qquad \Gamma \vdash t_2 : T, \varepsilon_2}{\Gamma \vdash t_1 \texttt{:=} t_2 : T, \varepsilon_1 \cup \varepsilon_2 \cup \texttt{w}} \quad \text{(T-Assign)}$$

Kuva 5: Viitemuuttujat

7 Viitemuuttujat

7.1 Efektijärjestelmä päivityksille

Imperatiivisille kielille ehkä kaikkein ominaisin sivuvaikutus on muuttujien arvon päivittäminen. Suuri osa efektijärjestelmien tutkimuksesta onkin tehty nimenomaan päivitettävien viitemuuttujien parissa. Päivitykset hankaloittavat ohjelman toiminnan ymmärtämistä niin ohjelmoijalle kuin kääntäjällekin, joten efektijärjestelmän tarjoamalle lisäanalyysille on käyttöä.

Kuvassa 5 esitetään yksinkertainen laajennus, joka tuo kieleen viitemuuttujat ja niiden perustoimitukset: luonti, lukeminen ja päivitys. Viitemuuttujien toimintaa käsitellään tarkemmin Piercen kirjassa [Pie02, luku 13].

Efektijärjestelmää on tässä laajennettu hyvin yksinkertaisesti: kullakin operaatiolla on oma efektinsä. Viitemuuttujan luomisen efekti on `a`, lukemisen efekti on `r` ja päivityksen efekti on `w`. Jopa näin yksinkertainen järjestelmä on kääntäjälle hyödyllinen: sen avulla kääntäjä saa paljon lisätietoa siitä, minkä lausekkeiden suoritusjärjestystä voi vaihtaa ilman, että ohjelman toiminta muuttuu. Jos kaksi lauseketta eivät kumpikaan kirjoita mihinkään viitemuuttujaan, voidaan ne suorittaa kummassa järjestyksessä hyvänsä.

7.2 Viitemuuttujat ja `let`-polymorfismi

Tällaisella efektijärjestelmällä voidaan myös kehittää yksi ratkaisu `let`-polymorfismin ja viitemuuttujien vuorovaikutuksen ongelmaan [Pie02, s. 335]. Ongelmahan on, että viitemuuttujissa olevia tyyppimuuttujia ei haluta yleistää `let`-sidonnassa, koska silloin viitteeseen voidaan kirjoittaa yhdellä tyyppi-instanssilla ja lukea toisella, ja tämä rikkoo tyyppijärjestelmän eheyden. Karkeana ratkaisuna voidaan sanoa, että `let`-sidonnassa sidottavaa lauseketta voidaan käyttää polymorfisesti *vain*, jos sillä ei ole sivuvaikutuksia. Tällöin varmistetaan myös, että se ei luo viitemuuttujia:

$$\frac{\Gamma \vdash t_1 : T_1, \emptyset \qquad \Gamma \vdash t_2[x/t_1] : T_2, \varepsilon}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2, \varepsilon} \quad \text{(T-LetPoly)}$$

$$
\begin{array}{lcl}
t & ::= & ... \\
 & & \texttt{ref}\ t \\
 & & !t \\
 & & t_1{:=}t_2 \\
T & ::= & ... \\
 & & \texttt{Ref}_\rho\ T \\
\varepsilon & ::= & ... \\
 & & \texttt{a}(\rho) \mid \texttt{r}(\rho) \mid \texttt{w}(\rho)
\end{array}
$$

$$\frac{\Gamma \vdash t : T, \varepsilon}{\Gamma \vdash \texttt{ref}\ t : \texttt{Ref}\rho\ T, \varepsilon \cup \texttt{a}(\rho)} \quad (\text{T-Ref})$$

$$\frac{\Gamma \vdash t : \texttt{Ref}_\rho\ T, \varepsilon}{\Gamma \vdash !t : T, \varepsilon \cup \texttt{r}(\rho)} \quad (\text{T-Deref})$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}_\rho\ T, \varepsilon_1 \qquad \Gamma \vdash t_2 : T, \varepsilon_2}{\Gamma \vdash t_1{:=}t_2 : T, \varepsilon_1 \cup \varepsilon_2 \cup \texttt{w}(\rho)} \quad (\text{T-Assign})$$

Kuva 6: Viitemuuttujat ja alueet

$$\frac{\Gamma \vdash t_1 : T_1, \varepsilon_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2, \varepsilon_2}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2, \varepsilon_1 \cup \varepsilon_2} \quad (\text{T-ILet})$$

Sääntö T-LetPoly sallii polymorfisen `let`-sidonnan sivuvaikutuksettomille lausekkeille ja sääntö T-ILet sallii monomorfisen sidonnan muille lausekkeille.

Tämä on vielä hyvin yksinkertainen ratkaisu, mutta monimutkaisempia ja tarkempiakin efektipohjaisia ratkaisuja on kehitetty [Wri92]. Nykyään vallitseva ratkaisu ongelmaan on kuitenkin *arvorajoitus* [Wri95]: sallitaan polymorfinen sidonta vain jos sidottava on "syntaktinen arvo": vakio, muuttuja tai λ-lauseke. Tämä on konseptuaalisesti paljon yksinkertaisempaa eikä käytännössä kovin paljon rajoittavampaa. Saattaa kuitenkin olla, että tämän ratkaisun suosio johtuu osittain siitä, että sitä käyttävissä kielissä ei ole valmiiksi efektijärjestelmää, joten sellainen täytyisi lisätä alusta pitäen.

8 Alueet

8.1 Aluekoodatut efektit

Viitemuuttujien efektijärjestelmä, jota käsiteltiin edellä, ei ole vielä kovin hyödyllinen. Kääntäjälle siitä on hieman iloa, mutta kovin imperatiivisesti kirjoitetut ohjelmat lukevat ja kirjoittavat muuttujia käytännössä kaikkialla, jolloin useimpien funktioiden efektiksi tulisi $\texttt{a} \cup \texttt{r} \cup \texttt{w}$. Tästä ei ole paljon apua. Tarvitaan tarkempi järjestelmä, joka kertoo paitsi, että joitain viitteitä luetaan ja kirjoitetaan, myös sen, *mitä* viitteitä käsitellään. Tähän on kehitetty ratkaisuksi *alueet* (regions).

Kuvassa 6 esitetään aluepohjainen efektijärjestelmä Talpinin ja Jouvelot'n tyyliin [TJ91, TJ92]. Ajatuksena on, että kukin viitemuuttuja kuuluu johonkin alueeseen ρ, ja tämä alue näkyy myös viitemuuttujan tyypissä. Myös kussakin viitteen luonti-, luku- ja kirjoitusefektissä on mukana alue, jota se koskee. Ajatellaan, että konkreettisia aluetunnisteita γ on numeroituvasti ääretön määrä.

Sääntö T-Ref kertoo, että uudella viitemuuttujalla voi olla mikä tahansa alue, ja sen luomisella on efekti, että se luo uuden viitteen siitä alueesta. Sääntö T-Deref kertoo, että kun alueen ρ viitettä luetaan, tulee siitä lukuefekti alueessa ρ. Ja sääntö T-Assign vastaavasti kertoo, että kun alueen ρ viitettä päivitetään, tulee siitä kirjoitusefekti alueessa ρ. Järjestelmä on siis kaiken kaikkiaan hyvin yksinkertainen.

Mitä tästä hyödytään? Nyt tyyppijärjestelmä pakottaa kaikki "keskenään tekemisissä olevat" viitteet samaan alueeseen γ. Seurauksena tiedetään, että kullakin viitteellä on tasan yksi alue, joten eri alueiden viitemuuttujat eivät voi osoittaa samaan paikkaan. Tämän tiedon avulla voidaan tehdä paljon tarkempaa uudelleenjärjestelyä: lausekkeet, joiden efektit ovat $\mathtt{r}(\gamma_1)$ ja $\mathtt{w}(\gamma_2)$ eivät voi mitenkään vaikuttaa toisiinsa, joten niiden keskinäisellä järjestyksellä ei ole väliä.

8.2 Efektimaskaus

Vaikka alueiden käsittely tapahtuu tyypillisesti automaattisena efekti-inferenssinä kääntäjän sisällä, olisi siitä jotain hyötyä myös ohjelmoijalle näkyvän tyyppijärjestelmän osana. Aluepohjainen efektijärjestelmä antaa jo hyvät puitteet ohjelmoijallekin nähdä, mitkä osat ohjelmaa voivat vaikuttaa mihin osiin.

Mahdollisuudet eivät lopu vielä tähän. Efektijärjestelmä antaa esimerkiksi seuraavanlaisen tyypityslauseen:

$$\vdash \mathtt{let\ x\ =\ ref\ 1\ in\ let\ y\ =\ x\ :=\ succ\ (!x)\ in\ !x} : \mathtt{Nat}, \mathtt{a}(\rho) \cup \mathtt{r}(\rho) \cup \mathtt{w}(\rho)$$

Kiinnostavaa tässä on se, että lauseke voi saada tämän efektin *kaikille* ρ:n arvoille. Toisin sanoen lausekkeella on kyllä sivuvaikutuksia, mutta ne voivat tapahtua missä alueessa hyvänsä, erityisesti myös sellaisessa, joka on *erillään* kaikista muista. Tämä tarkoittaa, että lausekkeen sivuvaikutukset *eivät näy ulospäin.*

Tämä ominaisuus näkyy puhtaan syntaktisena piirteenä efektissä: ρ esiintyy vapaana lausekkeen efektissä, mutta se ei esiinny lausekkeen tyypissä eikä tyyppikontekstissa. Tämä antaa motivaation *efektimaskauksen* sääntöön:

$$\frac{\Gamma \vdash t : T, \varepsilon \qquad \rho \notin FV(\Gamma) \land \rho \notin FV(T)}{\Gamma \vdash t : T, \varepsilon \setminus (\mathtt{a}(\rho) \cup \mathtt{r}(\rho) \cup \mathtt{w}(\rho))} \qquad \text{(T-EffMask)}$$

Tässä FV on funktio, joka antaa tyyppikontekstissa tai tyypissä vapaana esiintyvät tyyppi- ja efektimuuttujat. Efektimaskaussääntö siis tunnistaa "ulkoisesti puhtaat" lausekkeet, jotka kyllä aiheuttavat sivuvaikutuksia, mutta ainoastaan sisäisessä toteutuksessaan.

Tällaisella tekniikalla on monia sovellusmahdollisuuksia. Funktionaaliset kielet kuten ML tukevat viitemuuttujia lähinnä siksi, että vaikka ohjelmien korkean tason rakenne on tyylikästä kuvata puhtaan funktionaalisesti, jotkin matalan tason algoritmit vaativat tehokkuussyistä imperatiivista koodia. Valitettavasti sivuvaikutusten mukaantuomisen myötä menetetään monia ominaisuuksia, joihin puhtaan funktionaalisissa kielissä voi luottaa.

Efektijärjestelmät tarjoavat ratkaisun tähän. Sopivan efektijärjestelmän avulla kieli voi yhä tukea sivuvaikutusten käyttöä algoritmien toteutuksessa mutta silti taata, että puhtaiksi merkityt funktiot eivät muuta ympäristöään eivätkä riipu siitä.

8.3 Aluepohjainen muistinhallinta

Efektimaskaussäännöllä on vielä muuan kiinnostava piirre. Kun lausekkeen efektissä esiintyy vapaa alue ρ, tarkoittaa se, että mitään ρ:sta varattua viitemuuttujaa ei tulla enää käyttämään kun lauseke on suoritettu. Tämä tarkoittaa, että kaikki nämä viitemuuttujat voidaan vapauttaa lausekkeen suorituksen päätyttyä.

Tähän ajatukseen pohjautuu *aluepohjainen muistinhallinta* [TT97]. Sen kantavana ideana on, että muistinvaraus voidaan tehdä erillisistä alueista, käytännössä eräänlaisista pinoista, jotka voidaan vapauttaa kerralla kun ollaan varmoja, että mitään alueesta varattua kappaletta ei enää käytetä.

Kun alueita käytetään muistinhallintaan, lisätään aluemuuttuja *kaikkiin* tyyppeihin, joiden alkioita varataan dynaamisesti, ei pelkästään viitetyyppeihin. Lisäksi kaikki lausekkeet, jotka varaavat tällaisia alkioita, varustetaan aluekoodilla, joka kertoo, mistä alueesta alkio varataan. Näitä alueita luodaan `letregion`-lausekkeella. Esimerkiksi efektimaskaussäännössä esiintyvä lauseke muutettaisiin muotoon:

$$\texttt{letregion}\ \rho\ \texttt{in}\ t$$

Näin alueet muuttuvat tyyppijärjestelmän abstraktioista konkreettisiksi ajonaikaisiksi objekteiksi. Ylläolevassa lausekkeessa sidotaan aluemuuttuja ρ uuteen alueeseen, ja lauseke t käyttää tätä aluetta varatessaan niitä objekteja, joita ei käytetä t:n ulkopuolella. Kun lausekkeesta poistutaan, alue ja kaikki siitä varatut objektit vapautetaan kerralla.

Aluepohjainen muistinhallinta on osoittautunut toimivaksi ajatukseksi, ja sillä on onnistuneesti jopa korvattu perinteinen roskankeruuseen perustuva muistinhallinta. Sen edut eivät kuitenkaan ole olleet yksiselitteisiä, joten se ei ole levinnyt muutamia kokeellisia ML-toteutuksia pidemmälle.

9 Lopuksi

Tässä esitetyt tyyppisäännöt ovat tarkoituksellisen yksinkertaisia. Ne eivät sellaisinaan sovellu kovin hyvin niin tyyppitarkastukseen kuin tyyppi-, efekti- ja alueinferenssiinkään. Käytännössä kuitenkin useimmat efektijärjestelmät on suunniteltu käytettäväksi nimenomaan kääntäjän välikielessä, kun lähdekielessä ei ole mitään valmiita efekti- tai alueannotaatioita. Efektijärjestelmissäkin tyyppi-inferenssi on siis mahdollista.

Efektit eivät ole ainoa tapa mallintaa sivuvaikutuksia tyyppijärjestelmän keinoin. Toinen tapa, jota hyödynnetään etenkin Haskell-kielessä, ovat *monadit*: arvot, jotka kuvaavat "komentoja", jotka suorittavat sivuvaikutukset. Efektit ja monadit ovat läheisessä suhteessa toisiinsa [Wad99, BHM00]: esimerkiksi efektimaskaussääntö vastaa melko suoraan Haskellin `runST`-funktiota, jolla voi suorittaa sijoituksia käyttävän monadisen komennon ja saada sen tuloksen puhtaana arvona.

Toisaalta efektit ovat ohjelmoijan kannalta ehkä intuitiivisempia: tyyppi $T_1 \xrightarrow{\varepsilon} T_2$, "funktio, joka ottaa T_1:n ja palauttaa T_2:n, aiheuttaen suorituksensa aikana efektit ε", vastaa suoremmin ohjelmoijan käsitystä sivuvaikutuksellisen funktion toiminnasta kuin $T_1 \to \mathbf{T}_\varepsilon(T_2)$, "funktio, joka ottaa T_1:n ja palauttaa komennon, joka suorittaa sivuvaikutukset ε antaakseen T_2:n".

Onkin siis sääli, että efektijärjestelmät on pitkälti pyritty piilottamaan kääntäjän sisään pois ohjelmoijan ulottuvilta. Siinä missä tyypit auttavat ohjelmoijaa hallitsemaan ohjelman funktionaalisia piirteitä, efektit auttaisivat hallitsemaan imperatiivisia piirteitä. Toivoa sopii, että tulevaisuudessa imperatiivisten ohjelmoijien tekijöillä on käytettävissään muitakin apuvälineitä kuin Javan köykäinen poikkeustarkistus.

Lähteet

BHM00 Nick Benton, John Hughes ja Eugenio Moggi. Monads and effects. URL `citeseer.nj.nec.com/benton00monads.html`. 2000.

Pie02 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

PL99 Francois Pessaux ja Xavier Leroy. Type-based analysis of uncaught exceptions. *Symposium on Principles of Programming Languages*, sivut 276–290, 1999. URL `citeseer.nj.nec.com/pessaux99typebased.html`.

TJ91 Jean-Pierre Talpin ja Pierre Jouvelot. Polymorphic type region and effect inference. Tekninen raportti EMP-CRI E/150, 1991. URL `citeseer.nj.nec.com/talpin92polymorphic.html`.

TJ92 Jean-Pierre Talpin ja Pierre Jouvelot. The type and effect discipline. *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, sivut 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press. URL `citeseer.nj.nec.com/article/talpin93type.html`.

TT97 Mads Tofte ja Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997. URL `citeseer.nj.nec.com/tofte97regionbased.html`.

Wad99 Philip Wadler. The marriage of effects and monads. *Proceedings of the ACM SIGPLAN International Conference on Function-*

al Programming (ICFP '98), osa 34(1), sivut 63–74, 1999. URL `citeseer.nj.nec.com/article/wadler98marriage.html`.

Wri92 Andrew K. Wright. Typing references by effect inference. Teoksessa Bernd Krieg-Bruckner, toimittaja, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, osa 582, sivut 473–491. Springer-Verlag, New York, N.Y., 1992. URL `citeseer.nj.nec.com/wright92typing.html`.

Wri95 Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995. URL `citeseer.nj.nec.com/wright95simple.html`.