Dispozicija doktorske disertacije

Žiga Lukšič
Mentor: dr. Matija Pretnar

Učinki in Enačbe (Effects and Equations)

Pregled področja in dosedanjih raziskav

Pri zasnovi programskih jezikov je ključnega pomena močno matematično ogrodje jezika, ki zagotavlja smiselnost in pravilnost napisane kode. Programe lahko predstavimo kot matematične objekte, kar nam omogoča boljši vpogled v delovanje, hkrati pa pridobimo močna orodja za analizo in formalno dokazovanje programov. Vendar matematični objekti, kot na primer funkcije, ne vplivajo na svoje okolje. Tako so računski učinki, kot na primer izpisovanje besedila na zaslonu, odčitavanje podatkov iz senzorjev ali komunikacija z oddaljenim strežnikom, velik izziv za matematično formulacijo.

Računske učinke lahko dodajamo kot primitivne mehanizme v programski jezik, vendar moramo temu primerno za vsak učinek prilagoditi tudi pomen naših programov in pristope k formalnemu dokazovanju. Ker smo s tem omejeni na zgolj že vgrajene učinke, bi zato želeli dodati vse učinke, ki jih pogosto potrebujemo, s čimer pa postajajo temelji našega jezika kompleksnejši. Zato želimo k problemu računskih učinkov pristopiti z rešitvami, ki omogočajo strukturirano dodajanje učinkov brez prilagajanja programskega jezika.

Ena od takšnih rešitev temelji na t.i. *monadah* [5], kjer vrednosti ovijamo v posebne konstruktorje, funkcije pa dvignemo tako, da lahko sprejemajo tako prilagojene vrednosti, vse pa je med seboj povezano s posebnimi zakoni monad. Takšen pristop k učinkom deluje, kot dokazujejo programski jeziki kot recimo Haskell, vendar ima svoje pomanjkljivosti. Uporaba monad spremeni način pisanja kode, uporaba različnih učinkov zahteva veriženje monad s posebnimi pretvorniki in pri veliki količini učinkov zmanjšuje preglednost kode.

Drug pristop k računskim učinkom je uporaba algebrajskih učinkov in prestreznikov. Področje se je pričelo z uvedbo zgolj algebrajskih učinkov [6, 7], kjer vsak računski učinek predstavimo z pripadajočo operacijo, operacije pa so med seboj povezane z enačbami. Tako operacije skupaj z enačbami tvorijo algebrajsko teorijo učinkov. Za primer lahko za operacijo vzamemo nedeterministično izbiro med dvema elementoma, kar pišemo kot $x \oplus y$, primer enačbe v teoriji nedeterministične izbire pa je recimo komutativnost, torej $x \oplus y \sim y \oplus x$.

Operacija sama po sebi zgolj nosi podatke, zato ji moramo pripisati pomen v programskem jeziku. Ne želimo spreminjati programskega jezika za vsako

dodano operacijo, temveč želimo enotni pristop k interpretaciji operacij, kar je motiviralo razvoj algebrajskih prestreznikov [9]. Algebrajske prestreznike si lahko predstavljamo kot posplošitev prestreznikov napak, le da imamo v tem primeru možnost nadaljevanja programa. Na primer, če v kodi uporabimo nedeterministično izbiro $x \oplus y$, operacijo prestreže dodeljeni prestreznik, kateri ima dostop do argumentov operacije (x in y) in pa možnost nadaljevanja programa (funkcija k), ki pričakuje izbiro enega od elementov. Na nadaljevanju programa vedno implicitno uporabimo prestreznik, ki je prestregel operacijo. Pri implementaciji prestreznikov imamo proste roke, torej lahko napišemo prestreznik, ki vedno izbere levi element, torej požene $k\ x$, lahko pa izračunamo obe možnosti, $k\ x$ in $k\ y$ in se odločimo za boljšo izbiro (npr. tisto, ki vrne večjo vrednost). Ker pa med operacijami veljajo enačbe teorije učinkov, jih morajo prestrezniki temu primerno spoštovati. Takšni prestrezniki delujejo kot homomorfizmi med modeli teorij, zaradi česar so v izvorni obliki bili dovoljeni zgolj prestrezniki, ki zadoščajo enačbam [8].

Možna razširitev prestreznikov je dodatna preslikava, ki jo uporabimo na končni vrednosti programa, kar omogoča, da prestrezniki spremenijo tip programov na katerih jih uporabimo. Na primer, na programu, ki uporablja nedeterministično izbiro, lahko uporabimo prestreznik, ki zbere vse možne rezultate programa. Če program vrne vrednost x jo pretvori v seznam z enim elementom $[x]$. Če pa program sproži operacijo $x \oplus y$ z kontinuacijo k (na kateri že implicitno uporabimo prestreznik), najprej pridobimo vse rezultate pri prvi izbiri, $l_1 = kx$, nato pa še vse rezultate pri drugi izbiri, $l_2 = ky$ in nato vrnemo sestavljen seznam $l_1 @ l_2$. Vendar opisani prestreznik ne spoštuje komutativnosti nedeterministične izbire (saj le ta vpliva na vrstni red elementov v končnem seznamu). Ker veliko zanimivih in uporabnih prestreznikov ne zadošča enačbam teorij (in so zato zgolj morfizmi med prostimi modeli), so enačbe pogosto izpuščene zavoljo večje ekspresivnosti prestreznikov.

Kljub omejevanju možnih prestreznike, so enačbe pomemben del teorije računskih učinkov. Če naš program izpiše niz ``` ``algebrajski'' ``` in nato še `" učinki"`, bi lahko enako dosegli z zgolj enim izpisom `"algebrajski učinki"`. To lahko izrazimo z enačbo $print(x); print(y) \sim print(x \hat{} y)$. Ko pišemo programe se pogosto zanašamo na enačbe, na primer, da vrstni red pri dodajanju elementov v množice ni pomemben, torej $add(x);\ add(y) \sim add(y);\ add(x)$. Vendar če želimo, da je program modularen glede izbire prestreznika, takšna predpostavka ni samoumevna. Enačbe med učinki so prav tako ključnega pomena za formalno dokazovanje, zaradi česar je pomembno, da poiščemo način kako povrniti enačbe v teorijo učinkov brez omejevanja pri pisanju prestreznikov.

Tipi nam pri programiranju pomagajo tako pri razumevanju kot pri varnosti. Če nek prestreznik spoštuje določene enačbe, lahko to morda izrazimo v njegovem tipu. Že pri uvedi algebrajskih prestreznikov so sistem tipov nadgradili z informacijami o učinkih [8]. To je obsegalo tako informacije o tem, kateri učinki se lahko sprožijo v programu, kot tudi katere učinke prestreznik prestreže. Razvoj sistema tipov in učinkov se je nadaljeval v dveh smereh: t.i. 'row-types' [4] in 'subtyping' [10]. Hkrati se je pričel tudi razvoj dodajanja enačb v odvisne tipe [1]. Enačbe v tipih omogočajo sledenje teorij učinkov. Da

prestrezniku dodelimo želen tip, se moramo najprej prepričati, da zares spoštuje želeno teorijo učinkov, s čimer poskrbimo za ohranjanje varnosti kode. Tako kot se uveljavlja uporaba učinkov z lokalno signaturo, lahko tudi enačbe dodajamo lokalno, s čimer pridobimo lokalne teorije učinkov. V izvornemu pristopu je teorija učinkov globalna [8], torej morajo teoriji zadoščati vsi prestrezniki. Ker pa mnogi uporabni prestrezniki ne spoštujejo teorij, jih lahko pri uporabi lokalnih teorij kljub temu uporabljamo v delih programov, kjer ne zahtevamo nobenih enačb. Ker pomen učinkov določajo prestrezniki, so prestrezniki edini konstrukt, ki lahko spremeni lokalno teorijo.

Uporaba enačb nam pri sklepanju omogoča, da ločimo kodo programa od kode uporabljenih prestreznikov. Tako lahko formalno dokazujemo lastnosti kode v želeni teoriji učinkov in nato ločeno pokažemo, da naši prestrezniki to teorijo spoštujejo. Prav tako lahko prestreznike zamenjamo s poljubnimi drugimi prestrezniki, ki spoštujejo teorijo, in se dokaz ne spremeni. S tem pristopom nekatere dokaze (kot npr. v [2]) razbijemo na manjše in bolje strukturirane kose, hkrati pa lahko za kompleksnejše programe dokaze komponiramo.

Opis raziskovalne vsebine

Razvoj zanesljivih in ekspresivnih sistemov tipov izboljša varnost programov in formalno dokazovanje lastnosti programov. Kot pri pristopu z odvisnimi tipi [1] želim razširiti sistem tipov in učinkov z dodatnimi informacijami o enačbah, ki veljajo v želeni teoriji učinkov. S tem olajšamo dokazovanje in omogočimo dokazovanje pravilnosti kode ob uporabi poljubnih prestreznikov, ki spoštujejo izbrano teorijo učinkov. Hkrati uporaba enačb omogoča razvoj metod optimizacije, ki temeljijo na teorijah učinkov. Da je razširitev primerna želim definirati z uporabo denotacijske semantike, kjer programe prestavimo z matematičnimi objekti. Z enačbami povezujemo programe, zato jih modeliramo z relacijami na matematičnih objektih. Potrebno je poskrbeti, da programe, ki so enaki na podlagi enačb, slikamo v matematične objekte, ki so v pripadajoči relaciji. Problem ali prestrezniki spoštujejo izbrano teorijo rešujemo z uporabo formalnih sistemov logike s katerimi lahko to izrazimo. Pri izbiri logike imamo na voljo več možnosti, izbrana logika pa nam določa katere prestreznike sprejmemo.

Raziskovalna vprašanja

- Ali implementacija enačb v programski jezik z algebrajskimi učinki vzpostavi želeno strukturo teorije učinkov?
- Katerim lastnosti mora zadoščati programski jezik, da v pripadajoči denotacijski semantiki veljajo želene lastnosti?
- Kako izbira logike vpliva na jezik in denotacijsko semantiko jezika?

- Kako lahko delno preverimo, da prestrezniki spoštujejo teorijo z orodji kot je quickcheck [3]?
- Kako prilagoditi lastnosti kot so varnost, zdravost, adekvadnost in kontekstno ekvivalenco?
- Aplikativna uporaba enačb med učinki.

Pričakovani rezultati in prispevek k znanosti

Kljub temu, da se je teorija algebrajskih učinkov uveljavila v sferi funkcijskega programiranja, ostaja področje eksplicitnih enačb med učinki še precej neraziskano področje. Smer mojih raziskav, napram sorodnim pristopom z odvisnimi tipi [1], zahteva manj specializiran sistem tipov in je bolj v skladu z izvornimi idejami [8]. Dopolnitev tipov z enačbami med učinki olajša dokazovanje lastnosti programov, kar je ključnega pomena v mnogih uporabnih aspektih, kot npr. statistično programiranje, kjer je varnost naše kode ključnega pomena za kredibilnost rezultatov. Podobno velja za dokazovanje programov, ki uporabljajo knjižnice, kjer lahko lastnosti tuje kode izrazimo z enačbami, brez da odstranimo abstrakcijo in razkrijemo kodo. V primeru dovolj močne denotacijske semantike in kontekstne ekvivalence enačbe prav tako odprejo vrata v optimizacijo na podlagi učinkov, kjer za varnost optimizacij jamčimo preko enačb.

Literatura

[1] D. Ahman. Handling fibred algebraic effects. *PACMPL*, 2(POPL):7:1–7:29, 2018.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[3] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In M. Odersky and P. Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279. ACM, 2000.

[4] D. Hillerström and S. Lindley. Liberating effects with rows and handlers. In J. Chapman and W. Swierstra, editors, *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, pages 15–27. ACM, 2016.

[5] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[6] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[7] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[8] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In G. Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[9] M. Pretnar. *Logic and handling of algebraic effects*. PhD thesis, University of Edinburgh, UK, 2010.

[10] A. H. Saleh, G. Karachalias, M. Pretnar, and T. Schrijvers. Explicit effect subtyping. In A. Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 327–354. Springer, 2018.