Doctoral dissertation proposal

Žiga Lukšič
Mentor: doc. dr. Matija Pretnar

???

??? but in english

Overview of current research on the topic

The use of math based concepts in programming language design has ??? as shown by increasing integration of functional programming ideas in mainstream languages. A language with strong mathematical foundations allows us to present programs as mathematical objects, allowing for strong tools and procedures in formal reasoning of program correctness. As many programs accept input and return the computed value it is natural to interpret them as functions. Yet simple functions do not change their environment, so it is difficult to model computational effects such as printing text, reading data from sensors, or communication with a distant server.

Računske učinke lahko v programski jezik dodajamo kot posebne konstrukte, vendar moramo temu primerno za vsak dodani učinek prilagoditi pomen naših programov in pristope k formalnemu dokazovanju. Ker s tem omejimo uporabnike jezika na zgolj že vgrajene učinke, bi želeli dodati vse pogosto rabljene učinke, 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]. Monadični pristop k učinkom deluje, kot dokazujejo programski jeziki kot je Haskell, vendar uporaba različnih učinkov zahteva veriženje monad kar pri veliki količini učinkov zmanjšuje preglednost kode.

Drug pristop k računskim učinkom je uporaba algebrajskih učinkov [6, 7] in prestreznikov [8], kjer računske učinke predstavimo z pripadajočimi operacijami, te pa so med seboj povezane z algebrajskimi enačbami. 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 $x \oplus y \sim y \oplus x$.

Operacijam pomen dodelimo z algebrajskimi prestreznikov, ki prestrežejo klic operacije in zaženejo izbran izračun. Algebrajske prestreznike si lahko predstavljamo kot posplošitev prestreznikov izjem, le da ima uporabnik 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 zažene nadaljevanje $k\ x$, 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]. Ker veliko zanimivih in uporabnih prestreznikov ne zadošča enačbam teorij (in so zato homorfizmi zgolj med absolutno prostimi modeli), so enačbe pogosto izpuščene zavoljo večje ekspresivnosti prestreznikov.

Kljub omejevanju možnih prestreznikov, 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, takšne predpostavke niso samoumevna. Zagotavljanje modularnosti prestreznikov motivira uporabo teorij učinkov, saj želimo kodo programa ločiti od kode uporabljenih prestreznikov. Lastnosti kode dokazujemo v želeni teoriji učinkov in se pri tem ne oziramo na implementacijo prestreznikov, temveč zgolj upoštevamo enačbe teorije. Nato pokažemo, da izbrani prestrezniki spoštujejo izbrano teorijo in s tem zagotovimo, da jih lahko uporabimo na programu. Ker v dokazu kode ne uporabimo definicije prestreznikov lahko prestreznike zamenjamo z drugimi, ki spoštujejo teorijo učinkov, in s tem ne vplivamo na pravilnost skupnega dokaza. S tem pristopom nekatere dokaze (kot npr. v [2]) razbijemo na manjše in strukturirane kose, ki jih nato komponiramo v dokaze kompleksnejših programov.

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' [9]. Prav tako se je pričelo tudi dodajanja enačb v sisteme tipov [1] vendar zahteva uporabo zahtevnejših odvisnih tipov. Zato želimo enačbe dodati v tipe na preprostejši način, ki zahteva manjšo prilagoditev jezika in je bolj v skladu z izvornim pristopom [8]. 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 enačb.

Design of research

Kljub temu, da se je teorija algebrajskih učinkov uveljavila v sferi funkcijskega programiranja, ostaja področje eksplicitnih enačb med učinki še precej neraziskano področje. Kljub začetkom dela z odvisnimi tipi [1] ostaja še mnogo neraziskanih pristopov, ki nam nudijo preprostejši sistem tipov. S tem pridobimo preglednejšo operacijsko in denotacijsko semantiko, zaradi česar so takšni sistemi primernejši za implementacijo in splošnejšo uporabo. Kljub manj obsežnim spremembam tipi vsebujejo enačbe, ki govorijo o programih, ti pa se spet zanašajo na tipe, torej je pomembno, da poskrbimo za neciklične definicije. Poudarek mojega pristopa bo predvsem na preprostem sistemu formalnega dokazovanja pri uporabi algebrajskih učinkov, kjer bo ključnega pomena dokazovanje pravilnosti brez prisotnosti konkretnih prestreznikov, saj lahko zgolj v tem primeru zagotovimo modularnost. S tem se odprejo aplikativna področja kot je na primer statistično programiranje, kjer je uporaba učinkov naravna, hkrati pa je nujno zagotoviti določene lastnosti kode. Prav tako olajšamo dokazovanje v prisotnosti abstrakcije, saj lahko lastnosti programov, katerih koda je nedostopna, izrazimo z enačbami.

Research questions

- Kako prilagodimo tipe, da vključujejo lokalne teorije, brez cikličnih definicij?
- Koliko različnih logik imamo na izbiro in kako izbrati primerno?
- Kako izbira logike vpliva na jezik in denotacijsko semantiko jezika?
- Je problem, ali prestreznik spoštuje teorijo, odločljiv? Ali lahko v primeru neodločljivosti problem delno rešimo z orodji kot je quickcheck [3]?
- Kako prilagoditi lastnosti kot so varnost, zdravost, zadostnost in kontekstno ekvivalenco v prisotnosti enačb?
- V katerih aplikativnih primerih nam uporaba enačb pomaga? Ali lahko s pomočjo modularnih dokazov izboljšamo pristope na področjih kot je statistično programiranje?

Expected results and original contribution to science

Z vrnitvijo enačb bo teorija algebrajskih učinkov ponovno takšna, kot je bila izvorno zamišljena. Hkrati bo teorija nadgrajena z uporabo lokalnih teorij, s čimer odstranimo omejitve pri pisanju prestreznikov. Izboljšanje možnosti dokazovanja bodo tako omogočile prodor algebrajskih učinkov v širše sfere uporabe in odstranile mističnost algebrajskih prestreznikov. Domnevam, da bo na voljo

mnogo različnih logik, s katerimi bo možno izraziti tako trenutni sistem prestreznikov, kjer enačbe ne igrajo vloge, kot tudi sisteme kjer enačbe uporabnikom pomagajo. Ker se je pri uvedbi prestreznikov [8] izkazalo, da je problem ali prestreznik spoštuje teorijo neodločljiv, pričakujem za ekspresivne logike podoben rezultat.

References

[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] 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.