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.

Computational effects may be included with special constructs as programming primitives. This requires us to adapt our language and reasoning system for every additional effect, which leads us to try and include all of the commonly used computational effects. This however bloats and complicates the foundations of the language. We desire a general approach to effects where adding new effects does not require additional changes to the language itself.

One of the current approaches is based on so called *monads* [5]. As show in practice (for instance in the language Haskell) this approach is successful, but using many different effects requires chaining monads which reduces clarity of code.

Another rising approach is using algebraic effects [6, 7] and handlers [8] where computational effects are represented by operations. Operations and equations between them form an algebraic theory of effects e.g. we consider an operation representing nondeterministic choice, written as $x \oplus y$, where an example equation would be commutativity of choice i.e. $x \oplus y \sim y \oplus x$.

Operation calls are given meaning by the algebraic handlers, which intercept the call and run the instructed computations. Algebraic handlers can be seen as a generalisation of exception handlers with an additional possibility of continuing the evaluation of the program. Returning to the example of nondeterministic choice, when $x \oplus y$ is called, the handling handler has acces to the arguments of the operation (x and y) and it's continuation (function k) which

waits for a choice of one of the arguments. We implicitly handle the continuation with the same handler. When implementing handlers we have a plethora of options, such as a handler that always selects the first option by continuing with $k\ x$ or perhaps we calculate both options, $k\ x$ and $k\ y$ and choose the optimal one (for instance the one that results into a higher value). But because our effect theory includes equations, we require that our handlers respect them. In that case, handlers can be seen as homomorphisms between models of theories, motivating that only such handlers are allowed [8]. This requirement was soon dropped as there are many useful handlers which do not respect equations (and are thus homomorphisms only between absolutely free models), allowing for greater expressivity when defining handlers.

Equations remain a vital part of effect theories even though they impose restrictions on acceptable handlers. If our program prints `"algebraic"` and then `" effects"` it might as well print only `"algebraic effects"`. We can express this with the equation $\mathit{print}(x); \mathit{print}(y) \sim \mathit{print}(x \hat{} y)$. This is but one of the many examples of equations on which we rely when programming. Another example is commutativity of inserting elements into mutable sets, as expressed by the equation $\mathit{add}(x);\ \mathit{add}(y) \sim \mathit{add}(y);\ \mathit{add}(x)$. However, if we do not require handlers to respect such equations, we must prove that the program is indeed correct for every handler that we wish to use. Ensuring modularity of handlers is a great motivator for using effect theories as it allows decoupling of handlers from handled code while enabling us to reason using equations. We reason about code in the desired effect theory with no regards to the handler implementation. We separately prove that the chosen handlers respect the effect theory, enabling us to use them. Due to the separation we can switch between handlers as long as they respect the theory without impacting the correctness of our program. This approach would allow to break up certain proofs (such as in [2]) into smaller and more structured pieces, which can then be combined into proofs of more intricate programs.

When developing the theory of algebraic effects and handlers they have upgraded the type system with additional information about effects [8]. The type system keeps track of which operations might be called and also which operation calls are handled away by handlers. Further development proceeded in two directions: row-types [4] and subtyping [10]. There have also been attempts at including equations in the type system with the use of dependant types [1], but requires a more complex type system. The goal is to reintroduce equations back into languages with algebraic effects in a way that is as simple as possible and also true to the original ideas [9]. In the same way that effect systems slowly move to local effect signatures it is possible to also switch to local effect theories. While effect theories used to be global [8], forcing all handlers to respect the global theory, switching to local theories allows us to rely on effect theories in some parts of the program while we use handlers that do not respect the theory in other parts where equations are not required.

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] 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.