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

Despite algebraic effects gaining popularity in mainstream functional programming, the field of explicit equations between effects remains largely unresearched. While there is promising work with dependant types [1] there are many other approaches that result in simpler type systems. This requires less changes to operational and denotational semantics making such approaches more favourable for implementation and general use. While changes to the type system are kept to a minimum, types now include equations, which describe programs, which yet again depend on types. The system must be defined with great care to ensure that we avoid any and all circular definitions.

In order for the type system to ensure that handlers respect the desired effect theory we require a logic where such properties can be expressed and proven. If possible, the use of different logics should be used, allowing us to switch to a more expressive logic when needed.

My research will focus on establishing a more powerful reasoning system, aided by local effect theories in their original form, to allow for proofs that do not require concrete handlers. This way we obtain true modularity even in reasoning. This opens up possibilities in fields such as statistical programing where using effects is a natural approach but many properties of code need to be ensured. We further ease reasoning in presence of code abstraction because many crucial properties of abstracted code can be expressed with equations without revealing the implementation.

Research questions

- How to extend the type system to include local theories while paying attention to not introduce cyclic definitions?
- How many logics can we use in our type system and how to choose a suitable logic?
- How does the choice of logic impact our language and its denotational semantics?
- Is the problem of handlers respecting effect theories decidable? If undecidable, how do we design procedures similar to quickcheck [3] to at aid us with catching errors?
- How do we adapt properties such as safety, soundness, adequacy, and contextual equivalence in presence of equations?
- In which applicative cases are we aided by the use of equations? Can we improve approaches to fields such as statistical programing by using improved reasoning.

Expected results and original contribution to science

By reintroducing equations we return to the original concept of algebraic effects. We further improve it by introducing local theories, which help alleviate restrictions on handler definitions. Improved reasoning capabilities will allow algebraic effects to anchor in mainstream languages and will remove a layer of mysticism surrounding handlers.

I suspect that there will be many options in the choice of logic, ranging from logics that express the current system without equations to far more powerful logics that greatly benefit users. When introducing algebraic handlers [8] they have shown that determining whether a handler respects an effect theory is undecidable and it is most likely that it is the case with many expressive logics. However a procedure that automatically generates tests for handlers can be used to detect errors in handler code before attempting formal proofs, reducing the burden of additional work.

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.