Doctoral dissertation proposal

Žiga Lukšič
Mentor: doc. dr. Matija Pretnar

Uporaba teorij algebrajskih učinkov
Applications of algebraic effect theories

Overview of the field and current research on the topic

The use of math based concepts benefits programming language design as shown by increasing integration of functional programming ideas in mainstream languages (higher order functions, anonymous functions, patterns). 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 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* [6]. 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 [7, 8] and handlers [9] 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 the programs continuation, which awaits a choice of one of the arguments and is stored in the function k. 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). We implicitly handle the continuation with the same handler. 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 [9]. This requirement was soon dropped as there are many useful handlers which do not respect equations (and are thus homomorphisms only from absolutely free models), allowing for greater expressivity when defining handlers.

Even though equations impose restrictions on handlers they remain a vital part of the effect theory, as we can use equations to describe the interactions between effects. 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 $\mathit{add}(x);\ \mathit{add}(y) \sim \mathit{add}(y);\ \mathit{add}(x)$. Equations enable us to describe the behaviour of operations and thus allow decoupling of handlers from handled code when reasoning. 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 [3]) into smaller and more structured pieces, which can then be combined into proofs of correctness of more intricate programs.

When developing the theory of algebraic effects and handlers they have upgraded the type system with additional information about effects [2, 5]. The type system keeps track of which operations might be called and also which operation calls are handled away by handlers. There have 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 [10].

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], using simpler type systems 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 the same way that effect systems progressed to local effect signatures it is possible to also switch to local effect theories. While effect theories used to be global [9], 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.

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. This allows multiple implementations of logics.

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.

Research questions

- How to extend the type system to include local theories while avoiding cyclic definitions?
- How does the choice of logic impact our language and its denotational semantics and how do we implement suitable logics?
- How do we construct tools similar to quickcheck [4] to at aid us with catching errors?
- How do we adapt properties such as safety, soundness, adequacy, and contextual equivalence in presence of equations?
- How do equations help with reasoning in presence of abstractions, where we can describe properties of black-box programs with equations?
- In which applicative cases are we aided by the use of equations? Do such approaches benefit fields such as statistical programing [11]?

Expected results and original contribution to science

By reintroducing equations we return to the original concept of algebraic effects, improving reasoning and program safety through local theories, and allows for

effect-dependant optimisation. 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. 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.

References

[1] D. Ahman, *Handling fibred algebraic effects*, PACMPL **2**(POPL) (2018) 7:1–7:29, doi:10.1145/3158095.

[2] A. Bauer and M. Pretnar, *An effect system for algebraic effects and handlers*, Logical Methods in Computer Science **10**(4) (2014).

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

[4] K. Claessen and J. Hughes, *Quickcheck: a lightweight tool for random testing of haskell programs*, in: ICFP, ACM, 2000, pp. 268–279.

[5] O. Kammar, S. Lindley and N. Oury, *Handlers in action*, in: ICFP, ACM, 2013, pp. 145–158.

[6] E. Moggi, *Notions of computation and monads*, Inf. Comput. **93**(1) (1991) 55–92, doi:10.1016/0890-5401(91)90052-4.

[7] G. D. Plotkin and J. Power, *Adequacy for algebraic effects*, in: FoSSaCS, Lecture Notes in Computer Science **2030**, Springer, 2001, pp. 1–24.

[8] G. D. Plotkin and J. Power, *Algebraic operations and generic effects*, Applied Categorical Structures **11**(1) (2003) 69–94, doi: 10.1023/A:1023064908962.

[9] G. D. Plotkin and M. Pretnar, *Handlers of algebraic effects*, in: ESOP, Lecture Notes in Computer Science **5502**, Springer, 2009, pp. 80–94.

[10] M. Pretnar, *Logic and handling of algebraic effects*, Ph.D. thesis, University of Edinburgh, UK, 2010.

[11] M. Vákár, O. Kammar and S. Staton, *A domain theory for statistical probabilistic programming*, PACMPL **3**(POPL) (2019) 36:1–36:29.