Computational effects in computable and formalized mathematics (EFFMATH)
Project summary

Andrej Bauer
Faculty of Mathematics and Physics
University of Ljubljana
Slovenia

Abstract

This is a short summary of the research project EFFMATH, supported by Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.

Motivation and Background

In mathematics the application of a function f to an argument x always yields the same result $y = f(x)$, which depends just on x and nothing else. In programming the situation is different. A program may do other things apart from computing the result from the given input: it may get caught in a non-terminating loop, abort execution, interact with the external world through communication channels, use internal memory to store and retrieve data between invocations, compute results using random numbers, etc. All such phenomena are collectively known as *computational effects*.

A comprehensive mathematical theory of computational effects was developed by Moggi [9] under the slogan "effects are monads". It has been very successful, not only in theory but also in practice, as it heavily influenced the design of real-world programming languages, most notably Haskell. More recently, Plotkin, Power and Pretnar [10, 11, 13, 14, 12] developed a theory of computational effects based on algebra. They model computational effects as algebraic operations and homomorphisms of algebras. The homomorphisms manifest themselves in programs as *handlers*, a new concept that generalizes exception handlers, as known in mainstream programming languages.

In this project we will investigate how the theory of algebraic effects and handlers relates to two topics in foundations of mathematics, namely computable mathematics and type theory. The issues under investigation are not purely theoretical — they also have direct relation to programming practice and proof assistants, as we will explain.

Algebraic effects in computable mathematics

Computable mathematics studies computation with mathematical structures, especially those arising in analysis and topology. It is the theoretical foundation for certified scientific computation where numerical results are always guaranteed to be correct within a prescribed accuracy. (In contrast, the traditional floating-point computations sacrifice such guarantees for speed.) The mathematical technology which best addresses the needs of computable mathematics is realizability theory, a branch of logic and category theory which studies how mathematical structures and proofs of theorems may be *realized*, or implemented, by data structures and programs, respectively.

Relating algebraic effects to other computational models

Algebraic effects and handlers have not been studied systematically from the point of view of computable mathematics and realizability. We will take *core Eff* [5] as the prototypical programming

language with algebraic effects and handlers, and calibrate its expressive power in two ways:

1. We will analyze which constructive reasoning principles are realizable in Eff. Because Eff implements many kinds of computational effects it should realize various continuity and compactness theorems typical of Brouwerian intuitionism, but it does not posses sufficient intensional features to implement Russian constructivist principles, such as the formal Church's thesis. Techniques similar to those of [4] should be applicable here. The idea is to use handlers to build a tree representation of a realizer, and deduce its features by inspecting the tree.

2. We will compare (a suitable fragment of) Eff to other computational models by using the theory of typed partial combinatory algebras and applicative morphisms [8]. Applicative morphisms between Eff and other models of computation (Turing machines, λ-calculus, Gödel's T, PCF, type two machines, etc) will give us general "transfer" theorems between models of computable mathematics, such as those in [2].

Improving efficiency of exact real arithmetic

An implementation of a mathematical statement may be obtained systematically from its constructive proof via the realizability interpretation [3, 6]. The realizer which so arises is pure (does not contain any computational effects), but unfortunately tends to be quite inefficient. Nevertheless, it serves as a reference implementation against which optimized versions may be compared.

We shall study optimization with *benign* effects, i.e., uses of computational effects that improve efficiency of pure programs without altering their behavior. Examples of benign effects are short-circuiting recursive calls with exceptions, caching and memoization, local uses of mutable store, adaptive computing, and self-optimizing data structures. Specifically, we will focus on improving computation with real numbers. We have here in mind *exact* real arithmetic in which results may be computed to any desired precision. In such computations one always faces the question of error propagation: we may propagate required precision estimates backwards from the desired final precision to initial arguments, or propagate error bounds from initial arguments forwards to the final result. The latter strategy runs the risk of overestimation of required precision, and the former that of not meeting the desired final precision. Different strategies are suitable for different problems, but unfortunately it has not been easy to mix them, so every library for exact real arithmetic typically implements one fixed strategy. We shall tackle the problem as follows:

1. We implement different strategies for error propagation as different handlers. We express real-number computations using algebraic operations, which by themselves do not specify any particular evaluation strategy. We can then control whether backward or forward error propagation is used by wrapping the computation with this or that handler. Mixed strategies are possible also because handlers may be installed locally to control just a small subcomputation.

2. To further improve efficiency, and to aid automatic selection of suitable evaluations strategy, we can implement a third kind of handlers. These do not actually compute real numbers, but instead estimate how errors will propagate through the computation. The method is akin automatic differentiation, although we envision a more general approach that uses local Lipschitz constants [7].

Reasoning about algebraic effects and handlers

When a realizer is extracted from a constructive proof, it is correct by construction. But once we start optimizing it by hand in the name of efficiency, we need to prove that the modified program still works correctly, i.e., that its behavior is equivalent to the original program. Thus we need methods for proving properties of effectful programs. Of course, such methods are applicable much more widely than in computable mathematics and exact real arithmetic, for instance in certification of software for critical systems.

In [5] we developed an effect system and validated basic equational axioms that together provide a rudimentary basis for reasoning about algebraic effects and handlers. We shall continue work along these lines:

1. We will further develop the reasoning principles for algebraic effects and handlers and use them to prove correctness of various benign effects: short-circuiting computations with exceptions, caching and memoization, pure behavior of self-modifying data structures, and others. Our initial investigations show that algebraic equations are not sufficient to cover all such examples. Instead, we need an induction principle that allows us to prove properties of computations by cases, according to what operations they trigger.

2. We will extend the effect system with *effect inference* that automatically computes which algebraic effects a given piece of code may invoke. Such information helps understand, optimize, and debug code, as well as guarantee that the program will not perform certain actions (for instance, that it will not try to communicate to an outside party, or that it will only access certain parts of memory).

Algebraic effects in formalized mathematics

Our second topic is about using algebraic effects and handler to support proof assistants that are more flexible and powerful than existing ones. Proof assistants verify that inputs are correct up to judgmental equality. Proofs are easier and smaller if equalities without computational content are verified by an oracle, because proof terms for these equations can be omitted. In order to keep judgmental equality decidable, though, typical proof assistants use a limited definition implemented by a fixed equivalence algorithm. While other equalities can be expressed using propositional identity types and explicit equality proofs and coercions, in some situations these create prohibitive levels of overhead in the proof.

Voevodsky [15] has proposed a type theory HTS with two identity types, one propositional and one judgmental. This lets us hypothesize new judgmental equalities for use during type checking, but generally renders the equational theory undecidable without help from the user. Rather than reimpose the full overhead of term-level coercions for judgmental equality, we propose algebraic effect handlers as a general mechanism to provide local extensions to the proof assistant's algorithms.

Algebraic effects for type theory

We will augment the two-level type theory HTS proposed by Voevodsky with algebraic effects and handlers. This will allow us to augment terms of type theory with information on how to prove judgmental equalities, search for inhabitants of types, which unification hints to use, etc. A similar approach of augmenting terms in type theory with computations was taken by [16], except that monads were used to express the computational level.

Our initial experiments show that the workings of a proof assistant may be expressed in terms of basic operations "inhabit a type", "verify a judgmental equality", "insert a type coercion", and possibly "decompose a type into constituent parts". It is likely that they can be expressed in terms of even simpler concepts. We need to provide an operational semantics for expressions that mix type theory with algebraic operations and handlers. We can take the operational semantics of Eff as a starting point, but we need to modify it so that computations are evaluated inside function abstractions. This is necessary because the derivations found by the computations need to be uniform.

We will use the resulting language to express techniques used by proof assistants with algebraic effects and handlers. Handlers are especially convenient for implementation of various search strategies. For instance, there can be several handlers for finding type coercions that implement different unification strategies [1], such as type classes and canonical structures.

Applications to homotopy type theory

Voevodsky proposes the type theory HTS as a possible solution to difficulties with formalization of certain homotopy-theoretic constructions, most notably the notion of semi-simplicial types. We

shall test our system by expressing in it constructions from homotopy type theory which require both intensional and extensional identity types. Certainly, the construction of semi-simplicial types is one, but there are other constructions and notions which require introduction of new judgmental equalities. One such are higher inductive types. If our system turns out to be flexible enough, it will allow us to express higher inductive types with *judgmental* computation rules that seamlessly combine with the system's normalization procedures.

References

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.

[2] Andrej Bauer. A relationship between equilogical spaces and type two effectivity. *Math. Log. Q.*, 48(S1):1–15, 2002.

[3] Andrej Bauer. Realizability as connection between constructive and computable mathematics. In Tanja Grubba, Peter Hertling, Hideki Tsuiki, and Klaus Weihrauch, editors, *CCA 2005 – Second International Conference on Computability and Complexity in Analysis*, volume 326-7/2005 of *Informatik Berichte*, pages 378–379. FernUniversität Hagen, Germany, 2005.

[4] Andrej Bauer, Martin Hofmann, and Aleksandr Karbyshev. On monadic parametricity of second-order functionals. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2013)*, volume 7794 of *Lecture Notes in Computer Science*, pages 225–240. Springer, 2013.

[5] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings*, volume 8089 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[6] Andrej Bauer and Christopher A. Stone. Rz: a tool for bringing constructive and computable mathematics closer to programming practice. *J. Log. Comput.*, 19(1):17–43, 2009.

[7] Abbas Edalat. Weak topology and a differentiable operator for lipschitz maps. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008*, pages 364–375. IEEE Computer Society, 2008.

[8] J. Longley. *Realizability Toposes and Language Semantics*. PhD thesis, Edinburgh University, 1995.

[9] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[10] Gordon Plotkin and John Power. Notions of computation determine monads. In *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356, 2002.

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

[12] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[13] Gordon David Plotkin and Matija Pretnar. A logic for algebraic effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 118–129. IEEE Computer Society, 2008.

[14] Matija Pretnar. *The Logic and Handling of Algebraic Effects*. PhD thesis, School of Informatics, University of Edinburgh, 2010.

[15] Vladimir Voevodsky. A simple type system with two identity types. `https://uf-ias-2012.wikispaces.com/file/view/HTS.pdf/410120566/HTS.pdf`, February 2013.

[16] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: a monad for typed tactic programming in coq. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13*, pages 87–100, 2013.