Modern society increasingly depends on complex computer systems, so, for our comfort and safety, we need them to be reliable, easy to maintain, and extendible with new features. These requirements are especially hard to attain on the software side. We all fall victim to this from time to time, when a 'bug' causes a program not to behave as expected, or crash altogether. In some systems, such as those controlling moving vehicles or medical equipment, it is of paramount importance that such incidents are avoided.

One of the goals of computer science is to improve this situation by proposing new methodologies of creating software and ensuring its correctness. One of such methodologies, which has been getting a lot of attention recently, is *functional programming*, in the form of *functional programming languages* or *functional features* added to mainstream programming languages. Its strength lies in a very precise, highly mathematical approach and modularity, which allows the programmer to compose programs out of small, easy-to-understand components. As a result, functional programming helps the programmer to quickly deliver software of high quality.

Within the research area of functional programming, a recent advancement has been made by the introduction of *algebraic effects*. They allow the programmer to structure and reason about their programs using tools from abstract algebra (a branch of mathematics), reaching high levels of precision and readability of the code. Algebraic effects have been an active topic of research in the programming languages community, both on the more practical side—resulting in a couple of new, experimental programming languages, and extending the existing ones—and the theoretical side, finding applications even in remote areas, such as quantum computing.

One important open problem in the area of algebraic effects is a discrepancy between how they are implemented in practical applications and how they are understood on the theoretical side. In the latter setting, algebraic effects come with a natural notion of *algebraic specification*, which allows one to formally describe some properties of the components used in the code. Then, these properties are always valid for the component, ensuring that it will never misbehave. Moreover, such an algebraic specification can determine the component in a unique way, which would free the programmer from coming up with a (possibly erroneous) implementation. The challenge, and the topic of this project, is to try to find out a way to express the power of algebraic effects with specifications in practical programming.

The problem arises, because the theoretical descriptions of algebraic effects use mathematical structures that are not directly expressible in programming. Hence, this project proposes to introduce alternative structures to describe algebraic effects, ones that will be more amenable for practical engineering. In particular, this project proposes to use *continuations*, which are structures used to express the control flow of programs, that is, broadly speaking, the order in which different tasks are performed by the machine.

The goals of this project include a better understanding of both algebraic effects and continuations on the theoretical level, and inventing new mathematical structures that can be translated into practical implementations. Additionally, a set of experimental tools are to be developed to automate the process of extracting implementations from specifications. In the long run, it leads to development of new programming methodologies, which support efficiency and robustness of the software development process.