In virtually all aspects of modern society – from governance and administration to business, science, and social media – the demand for efficient computing and data processing escalates in an unprecedented rate. This demand is not only about quantity (of computational power, number of servers and software developers), but mostly about the complexity of the tasks set before computer systems. This raises an important question of how to manage this complexity, and what kind of tools and methodologies are possible to aid construction and maintenance of complex systems.

On the side of software, the key word is *abstraction*: the ability to think in terms of high-level logic of the system, without the need to deal with the low-level technical details altogether. One example why tools that provide abstraction seem essential is the increase of parallelisation and distribution of the systems, which means that they are realised on a number of independent computing units, be they different cores in a processor, as in a PC or a smartphone, or thousands of remote servers in a cloud-computing scenario. If managed "manually", this leads to a number of problems not directly related to the high-level logic of the system, such as organising communication and synchronisation of different parts of the system, dealing with them competing for shared resources or deadlocks (occurring when there are two computations, each waiting for the other one to perform an action, which results in a global freezing of the system).

The tool that mediates between the logic of the program and the low-level implementation of the system is a *programming language*. In recent years, to help the programmers manage the complexity of the constructed systems, the mainstream industrial software development has turned its attention to *functional programming*, which is not a new idea (as it originated in the 1950s with the LISP programming language), but provides a higher level of abstraction than, say, object-oriented programming. Functional programming is a declarative paradigm, which means that the programmer defines *what* is the intended result of the program, rather than *how* to obtain it by manipulating the internal state of the machine. An advantage of employing functional programming on a larger scale is that it has been one of the foci of academic research on programming languages and the semantics of computing, which means that one can benefit from an abundance of results about the intricacies of semantics, type systems, testing, formal reasoning, and so on.

The proposed research project is in the area of a new addition to the functional-programming landscape: **algebraic effects**. In general, *effects* is a common name for constructs in a programming language that allow the program to do more than just compute a value given some initial arguments. Examples include: performing an input/output operation, changing the value of a variable, communicating with another thread/process, throwing an exception. One strength of functional programming is that it discourages using effects if they are not necessary, as a piece of code that is *pure* (that is, does not use any effects) is much more predictable, reusable, as wall as easy to understand, maintain, test, and reason about. One of the most important questions in the programming language research today is how to incorporate effects and not lose these desired properties. Algebraic effects appear to be the most promising course of action to obtain this.

The more specific goal of this project is to study two aspects that are important for a wider adoption of algebraic effects in practical programming and in understanding their mathematical foundations. The first one is to study the issue of how effects **compose**, that is, how one can program with a number of different effects at a time. Different effects can interact with each other in a number of ways, and the things become more difficult when we employ the full power of abstraction – for example, when we cannot statically tell which effects will be in use when we execute the program.

Another aspect is incorporating **coinduction**, that is, understanding a program with effects not as a batch computation that gives some final value for the initial input values, but as an interactive system. From this angle, even specifying what it means for a program to be "correct" is more difficult, since it is more about the observable behaviour, which is more complex to capture from the formal, mathematical point of view.

The overall expected result of this project is to establish a mathematical underpinning of these two aspects of effects. This will give a firm basis for building better-behaved and more expressive programming languages. Since algebraic effects already have a great impact on the way programming languages are designed and understood, solving these two issues should have a noticeable impact on the theory and practice of programming.