Scalable Reasoning about Concurrent Imperative Programs

Filip Sieczkowski

Anyone who has ever written even a single computer program should know how easy it is to make a mistake that would result in an incorrect result. However, what most people might not recognise is that this problem does not disappear entirely with experience: whenever we use a computer to solve a complex problem, we can be almost certain that some errors will be made. The process of *testing* the software are indispensable when it comes to noticing the existence of a problem or locating it within a particular part of the program, but they can never give us absolute certainty that an error isn't lurking somewhere in the code. It is the goal of the methods of *program verification* to provide us with such guarantees, and our project lies within that area of research.

One of the fundamental divisions within formal verification of programs is the distinction between *automated* methods, which can analyse the code of a program without the necessity of human interaction, and the *manual* techniques, where the crucial part of the process is the human interaction, often with some aid of the machine. This distinction stems from the known limits on the expressive power of computers: most of the interesting properties of programs lie squarely outside what computers can reason about on their own. Hence, automatic verification systems concentrate on showing that certain *classes of errors* are absent from the program, but do not consider the *full correctness* of the program. The human-driven methods, on the contrary, tend to allow the user to formulate precise *specifications*, i.e., intended behaviour of programs and their parts, and then prove that these specifications are met by the program code.

In general, program verification is a very difficult problem, and the research in the area has been carried out for more than half a century. However, it was only at the turn of the last century that we discovered techniques that allow us to reason effectively about programs in realistic programming languages. The crucial aspect of these techniques, known as separation logics, is their *modularity* — a property that allows us to construct proofs of correctness of larger programs from the proofs of smaller parts, while *abstracting* from a lot of their internal details. In a very real sense this process resembles the process of constructing the programs themselves. Thus, these techniques can be applied to programs of size and complexity much beyond what we could tackle before.

One of the most important problems tackled in the area of formal verification in the last decade was devising reasoning techniques for *concurrent* programming languages, i.e., languages in which multiple processes communicate with each other through a shared state which they all can modify. This model of computation is much more complicated than the traditional sequential models, where the program is isolated from the outside world — while at the same time, with an increasing number of cores of the modern day multiprocessors, it is a model that is increasingly more common in the programming practice.

In this project we take up both the questions of full correctness of concurrent programs and their automated analysis — as well as questions lying at the border of the two areas. Firstly, we work in the area of concurrent separation logics, where we want to devise proof methods that simplify the reasoning about programs, while retaining the high expressivity of a lot of the recent work. Secondly, we want to provide novel ways to guarantee that a program's execution is *deterministic*, i.e., that for any given input, the execution can only have a single result. Such a property means in particular that the order in which concurrent processes communicate does not matter, which should significantly simplify the reasoning about the program's execution — and consequently enhance the programmer's understanding of the code. The final task that we are going to work on is particularly interesting, though. We plan to research the possibility of utilising the specifications of concurrent libraries and data structures in the automated reasoning about the code that only *uses* these libraries. In practice, developing such techniques would mean that it is sufficient to prove small, crucial components of a vast concurrent program in order to automatically establish some guarantees about the entirety of the code.