Understanding recursion

Lorenzo Clemente

The verification of functional programs is a main challenge for the model checking community. Functional features appear in all major modern programming languages such as, e.g., C++, Java, Haskell, Python, Scala, Scheme, and Erlang. At the basis of functional programming is recursion, as illustrated by the following "hello world" of functional programming, i.e., the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, 13 \ldots$:

$$f(n+2) = f(n+1) + f(n), \text{ with } f(0) = 0 \text{ and } f(1) = 1.$$

However, allowing recursion together with data manipulation (here in the form of the arithmetic operation '$+$' on natural numbers) immediately leads to undecidability of all non-trivial properties about such programs, i.e., there exists no algorithm which always terminates and replies "yes/no" for such properties. By removing the data and leaving arithmetic operations uninterpreted (i.e., without "executing" them), one obtains a *recursion scheme*, which faithfully models the control flow of the program:

$$F \to 0 \mid 1 \mid F + F. \tag{1}$$

In this case, we obtain a recursion scheme which looks the same as a classic context-free grammar. However, symbols such as "F" can themselves have arguments, thus one gets a higher order scheme in the general case.

In this project, we propose three problems related to schemes and verification. In the first problem, we add to recursion schemes information about execution times. In this way we can say things such as "running symbol F takes two time units, and running symbols 0 and 1 takes one time unit". With such specification, the term "$0+0$" is allowed ($1+1=2$ time units), but the term "$0+0+0$" is not ($1+1+1=3$ time units). This concrete specification is clearly satisfiable since $2 \leq 5$, but this is not always the case, and in general it might be very difficult to establish which one is the case. We proposed a concrete model called *timed pushdown automata*, which is equivalent to first order schemes with timing information as outlined above, and we have shown that there exists an algorithm (running in time doubly exponential in the size of the automaton) that can decide properties of this model. In this project we aim at providing a more efficient algorithm (with singly exponential running time), as well as introducing more expressive primitives for the construction of the automaton.

In the second problem we address higher order schemes, which are a generalisation of recursion schemes as presented above with higher-order recursion, i.e., where nonterminal symbols can have functional arguments. Given a scheme as in (1) above, one could ask for example whether it generates all terms of the form 0, 1, $0+0$, $0+1$, $0+(1+1)$,..., or more generally all terms starting from constants $0, 1$ and with a binary constructor '$+$'. While this question is undecidable for schemes, it can partially be circumvented by over-approximating the scheme by what is called its *downward closure*, which intuitively means that any term generated by the scheme also generates all its subterms. We have recently shown how to compute downward closures for linear terms (i.e., terms with no binary symbol, unlike '$+$' above), and we plan to extend this result to arbitrary terms.

Finally, in the last problem we address a different model called *Petri nets*, which essentially are non-recursive programs with counters, and a problem called *regular separability*. While two Petri nets might be very different from each other, in the regular separability problem we look for "simple" reasons which can already tell the two nets apart. This is a very challenging and difficult open problem in the area.