Formalizing Abstract Machines

Internship Proposal

Advisor: Beniamino Accattoli (INRIA & LIX, École Polytechnique)

Email: beniamino.accattoli@inria.fr

The mathematical model behind functional programming languages is the \( \lambda \)-calculus, while the execution model is usually given as an environment-based abstract machine. Abstract machines are *machines* because they have a set of instructions whose complexity can be easily measured, while \( \beta \)-reduction—the only computational rule of the \( \lambda \)-calculus—is a non-atomic rule whose complexity is not evident. On the other hand, abstract machines are *abstract* because they omit many details that should be taken into account in an actual implementation, like the garbage collector.

There is an extensive literature about how to relate abstract machines to the \( \lambda \)-calculus. Recently, a new approach has been introduced by Accattoli, Barenbaum, and Mazza in [ABM14], that is based on the *linear substitution calculus* (LSC), a simple framework that is halfway \( \lambda \)-calculus and abstract machines, and that arises as a decomposition of \( \lambda \)-calculus via linear logic and rewriting theory. The idea is that the study of the relationship

\[
\lambda\text{-calculus} \leftrightarrow \text{Abstract machine}
\]

that usually amounts to proving that the machine correctly implements a given strategy in the \( \lambda \)-calculus, can be decomposed by studying the following two relationships

\[
\lambda\text{-calculus} \leftrightarrow \text{LSC} \leftrightarrow \text{Abstract machine}
\]

The advantage of such a factorization is that it cleanly disintangle the two tasks performed by abstract machines, *i.e.* the *decomposition of the substitution process*, that takes place at the level of the LSC, and *the search for the next redex*, that is what machines add with respect to the LSC. Moreover, the decomposition can be used to develop simple and modular complexity analyses of abstract machines. Namely, one is interested in bounding the number of instructions in the machine with respect to the number of \( \beta \)-steps in the \( \lambda \)-calculus. These analyses, again, can be done in two modular steps, bounding first the cost of the substitution process and then the cost of searching for the next redex.

In the last two years, a wide range of abstract machines has been decomposed via the LSC and then analyzed using basic complexity theory [ABM14, AC15, ABM15]. Such a new quantitative approach to abstract machines led to insights and improvements over the abstract machine at work in the Coq proof assistant [AC15].

The aim of the internship is to formalize the relationship between the simplest abstract machines, *i.e.* Krivine Abstract Machine, and the LSC. First, the correctness and completeness of the implementation will be addressed, and then, hopefully, the complexity analysis.

The interest in such a formalization goes beyond the usual fine-grained understanding of its proof. The theorem indeed relates the \( \lambda \)-calculus and the LSC, that are calculi with binders, *i.e.*
settings where terms are considered up to renaming, with abstract machines, where instead the renaming has to be handled explicitly. Formal reasoning in presence of binders is a notoriously subtle issue. Therefore, the formalization is meant to explore how naturally one can formally reason about systems that require different approaches to binders.

The internship will take place in the INRIA Parsifal team and the formalization will be carried out in the Abella proof assistant, a recent tool actively developed by the team jointly with the University of Minneapolis. Abella has been designed exactly to deal with languages with binders. An extensive tutorial for Abella [BCG+14] as well as various formalizations about $\lambda$-calculus are available on the Abella website (http://abella-prover.org/), as well as in some papers, for instance [Acc12].

The internship is meant to be an introduction to the research in many different and yet related topics, as abstract machines, complexity analysis, linear logic, formalization in presence of binders, rewriting, etc. The focus of the internship is on the formalization activity rather than in developing new results. Formalized reasoning, however, is here intended more as a pedagogical tool than as a goal in itself. Its purpose is in fact threefold: to test the ability of the student, to fill eventual gaps in the technical background of the student, and to provide a skill that is becoming more and more important for researchers in proof theory and in the theory of programming languages. A successful internship may lead to a PhD scholarship, where more challenging and open-ended topics—more likely related to the complexity analysis of functional languages than to formalizations—will be addressed.

Candidate Profile and Background

The ideal internship candidate should have played with a proof assistants and have some basic knowledge about the $\lambda$-calculus. No knowledge of complexity theory, rewriting theory, or linear logic, nor experience with implementations is required. The concepts involved in the internship are quite basic. The most important skill is a tendency for abstract, simple, and esthetic reasoning.

References


