Metaprogramming on the Proof Level

Gergely Dévai

This research is about the usability of metaprogramming techniques for construction of program correctness proofs.

In metaprogramming one uses two languages: a base language and a meta language. The compilation is done in two phase: first a precompiler processes the meta language constructs and generates a result written purely in the base language. Examples vary from macro assemblers to template metaprogramming in C++. In these classical applications metaprogramming is used to generate programs.

The goal of formal programming methods is to produce verified programs. These methods try to increase reliability of software by formally proving its properties. In order to achieve this goal, one has to construct correctness proofs of programs. There are two main approaches: writing the program first and proving its properties afterwards (verification), or deriving the program from its specification using rules that guarantee soundness (correctness by construction).

We use the second approach. In our system the programmer writes specification first, then refines it by more detailed specification statements. This process is called stepwise behavioral refinement and it results in a correctness proof of the algorithm. Our system checks this proof and generates the program automatically, which is then correct by construction.

We investigate how to support proof construction using metaprogramming techniques. That is, the base language is the proof language and metaprogramming elements are used to generate proof fragments. Let us summarize the advantages of this approach.

- *Decreased proof-length.* If we identify often used proof fragments and generalize them to proof templates we can decrease the length of the proof by calling the templates instead of repeating the proof fragment several times.

- *Templates make the system extensible.* In our system we specify only a small subset of instructions. The proof for programming constructs like conditional statements, loops, procedure calls are generated by templates. This means that the verification conditions for these constructs are not built into our system. If one needs a new kind of loop that is not supported yet, can develop a template for the new construct. This is not possible in systems where the supported language elements are hard-wired in the system core.

- *Meta programming is safe at the proof level.* Primitive metaprogramming constructs, like macros sometimes generate inefficient or erroneous program fragments. This can also happen when metaprogramming is used to generate proof. Fortunately the generated proof is checked afterwards and all the errors are reported to the programmer already in compile time.

- *The trusted base can be minimal.* In every verification framework we have to trust the system: its proof engine, the rules about different language constructs etc. In our system this trusted base is minimal: it consists of the module that checks the validity of proofs and the specification of primitive instructions. All other features are provided by templates, and, according to the previous point, erroneous templates can not corrupt the soundness of the system.

In this paper we present a set of metaprogramming constructs that we have found useful in proof generation. These include different types of proof-templates, conditions, passing proof fragments as parameters and generation of template definitions by templates. We also present how to use these features to implement proofs for different programming language constructs.