

Challenges in Program Generation

Yukiyoshi Kameyama

Department of Computer Science, University of Tsukuba, Japan

Abstract. Program generation is ubiquitous in our programming lives. We often hand-generate shell scripts. *Lex/yac* generate, given a specific definition for a lexical grammar, a lexer and a parser which work for the given definition. *Compiler* generates, given a source program, an object program that runs on a given hardware or a virtual machine. On the other hand, domain experts want to write custom generators which generate programs specific to their domain, but it is not always easy for them, being non-professional programmers, to write such generators correctly.

We are interested in supporting domain-experts to write custom generators in a *safe* way. Specifically, we want to provide a means to guarantee the following safety properties *before* the generators actually generate programs:

- the generated “program” is syntactically correct (that is, it is actually a program.)
- the generated “program” does not cause type errors during execution (if the target language has static type system.)
- the generated “program” should not have free variables, or any other reference to undefined entities.

Multi-stage programming [3] is a successful solution to all of these problems. Today we can use safe multi-stage programming languages such as *MetaOCaml*.

Yet, there remain several important challenges in this area, as follows:

- is it possible to use imperative language, rather than functional language, or is it possible to use computational effects (such as control effects and mutable variables) in program generation ?
- is it possible to use a language without a static type system for program generation ?

The first problem is important, since many clever program generation methods use memoization, sharing, and exchange of code fragments such as *let-insertion*, to generate efficient programs. Without effects, we often find difficulty in writing a good program generator. We reported some solutions to this problem in our earlier work [1, 2].

The second one is an interesting problem as we face a number of dynamically typed programming languages used in generating programs, such as *Unix Shell*’s, *Lisp/Scheme*, *Perl*, *Ruby*, and *JavaScript*.

In this talk we address the second problem, and give a (partial) solution to it. We choose *JavaScript* as the target language (in which generated programs are written), and adopt a program-analysis approach to this problem. We approximate the behavior of future-generated programs in terms of types, and then

compute an appropriate type and a stage for each program. The type preservation theorem for the type system assures that, if a given program generator passes the test, no bad behavior (such as generating programs with free variables) may occur during the execution of the generator. We also argue how we can extend our method to computational effects in a safe way.

Acknowledgements. This work is based on the work by Hirobumi Takahashi and Yuichiro Kokaji.

References

1. Yuki-yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Closing the stage: from staged code to typed closures. In Robert Glück and Oege de Moor, editors, *PEPM*, pages 147–157. ACM, 2008.
2. Yuki-yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: staging with delimited control. In *PEPM*, pages 111–120, 2009.
3. W. Taha and M. F. Nielsen. Environment classifiers. In *POPL*, pages 26–37, 2003.