

# Putting Multi-stage Annotations to Work

## A Personal Research Statement for IFIP WG 2.11

Walid Taha  
Rice University, Houston, TX, USA,  
taha@cs.rice.edu

Techniques such as program generation, partial evaluation, just-in-time compilation, and run-time code generation respond to the need for general purpose programs which do not pay unnecessary run-time overheads. The thesis of our research is that a uniform, principled, high-level, and practical view of these diverse techniques is possible through *staged computation*, a paradigm for the development of maintainable, high-level yet high-performance software. Programming languages can support staging by providing high-level *annotations* to allow the programmer to break down the cost of a computation into distinct *stages*. Depending on their expressivity, such languages are called two-level, multi-level, or multi-stage languages. Two level languages can express programs with two stages. Multi-level languages allow staging of staged programs. Multi-stage languages extend multi-level languages with the ability to execute future-stage programs in the current stage. Over the past decade, there have been many advances in study of all these classes of languages. While most of my work has been on the last class of languages, I am interested in all three classes.

A premier applications of two-level languages has been in modeling the semantics of macros, partial evaluation, and runtime code generation. Macros are computations that happen before all the programs inputs are available. Partial evaluation takes a generic program and some of its inputs, adds staging annotations to it, and executes it. Executing the annotated program automatically produces a new, *specialized program*. For a wide range of applications, specialized programs are orders of magnitude faster than the original programs. Runtime code generation systems produce new sequences of machine instructions at runtime, using the latest available information about the context of the running computation. Runtime code generation has demonstrated its effectiveness in demanding applications such as operating systems. There have also been advances in the development of mathematical models of two-level, multi-level, and multi-stage languages. These models have been used to prove profound guarantees about these languages. For ex-

ample, a program generator written using these languages not only is type-safe, but we are guaranteed that *any generated program will also be type safe*. This provides an extraordinary level of assurance about the quality of the generated code.

Yet, compared to these advances in the design of these languages, the practical aspects still pose many open questions. For example, until recently only interpreters had been developed for multi-stage languages. Now that we have compiled implementations such as MetaOCaml, can demonstrate the effectiveness of staging in interesting applications such as dynamic programming algorithms, numerically intensive computation, and web applications? These questions can only be answered by extensive, realistic case studies. The goal of such case studies should be a better understanding of staging as an integral aspect of realistic software design.

To demonstrate the viability of compiled multi-stage programming as a *practical* unifying framework for various program generation technologies, there is a need for investigating many distinct and useful strategies for the compilation of multi-stage programming languages, which differ in how a “future-stage” value is represented. One strategy represents future-stage values by source-code, and corresponds to high-level program generation. Using this source-code strategy means that generation involves symbolic computation and that executing future-stage code involves a just-in-time compilation phase. This is the strategy used in current implementations of MetaOCaml. Another strategy would be to represent future-stage values by machine-code and corresponds to run-time code generation. Using this strategy means that no source code is available at runtime, and no runtime compilation phase is needed. Clearly, it has the advantage of having much faster runtime “compilation” times. An open question is whether there is an upper limit to the quality of the generated code in such a setting.

**About the Author:** Walid Taha is an assistant professor at the Computer Science Department at Rice University. His past research work has focused on the semantics and type systems for multi-stage programming language, and he has been the principle investigator on NSF awards addressing various aspects of these topics. He has been actively involved in helping bring together a research community concerned with all aspects of program generation. He has founded an organized events such as SAIG’01, SAIG’01, GPCE’02, and IFIP WG 2.11.