Adding Types to Untyped Languages

Matthias Felleisen

PLT
Northeastern University, Boston, Massachusetts
matthias@ccs.neu.edu

Abstract

Over the last 15 years, we have experienced a programming language renaissance. Numerous scripting languages have become widely used in industrial and open-source projects. They have supplemented the existing mainstream languages—C++ and Java—and, in contexts such as systems administration and web programming, they have started to play a dominant role.

While each scripting language comes with its own philosophy, their designers share an antipathy to types. As a result, these languages come without a static type system. Most script developers initially welcome this freedom, but soon discover that the lack of a type system deprives them of an essential maintenance tool.

My keynote explains my team's approach to equip such languages with a type system. The goal of our work is to empower programmers so that they can gradually enrich scripts with types on a module-by-module basis as they perform maintenance work on the system. Naturally, we wish to ensure type soundness so that the type annotations are meaningful, and we wish to accommodate the programming idioms of the original language in order to keep the overhead of type enrichment low.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Language Constructs and Features*]: Modules, Packages

General Terms Languages, Design

Keywords Software Contracts, Scheme, Type Systems

From Scripts to Programs, One step at a Time

When scripts in untyped languages grow into large systems, maintaining them becomes difficult. A lack of explicit and meaningful type declarations in scripting languages means that programmers must re-discover critical pieces of design information every time they wish to change the program. This analysis step slows down the maintenance process and may even introduce errors when the maintainer recovers only a part of a function's intended signature.

The conventional way to address the problem is to port the system from the scripting language to a typed language. This solution comes with serious disadvantages, however. First, it requires rewriting code that works well in many cases, often because it has been in use for a long time. Of course, any change to working code may introduce errors and should therefore be avoided if possible. Second, mainstream type-safe languages do not accommodate smooth and type-safe interactions with dynamically typed scripting languages. Programmers are thus forced to port entire systems, not just pieces, and they then end up maintaining both old and new versions.

My team and I are working on an alternative to the whole sale port of systems, namely the gradual enrichment of scripts with types. Our approach involves the development of a typed sister language and sound interoperation of typed and untyped code. To this end, we first equip the language with a module system that supports both the typed and untyped variant of the language. The second key piece is a behavioral contract system for module interfaces. In general these contracts can monitor all kinds of invariants as even higher-order values flow from one module to another [2]; here it helps us enforce types dynamically and thus enables sound interoperability between the typed and untyped world [6]. The third and final piece is a type system that accommodates the idioms of the original language as much as possible [5, 7] so that enriching untyped modules with types requires few changes to existing code.

Over the past four years, we have validated this recipe with the development of Typed Scheme, a typed sister language to PLT Scheme [1]. Experiments with around 10,000 lines of code shows that the type-enrichment of existing PLT Scheme modules requires changes to some five percent of the lines in order to appease the type checker. Although this percentage is low, it is not ideal and many challenges remain. Our next steps will hopefully introduce contracts for polymorphic types [3], add types for PLT Scheme's first-class classes and units [4], and provide tools for synthesizing type annotations for untyped modules.

References

[1] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced macrology and the implementation of Typed Scheme. *Scheme Workshop 2007*. Tech. Rep. DIUL-RT-0701, Université Laval, Quebec.

[2] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pp. 48–59, 2002.

[3] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Third Symposium on Dynamic Languages*, pp. 29–40, 2007.

[4] T. Stephen Strickland and Matthias Felleisen. Contracts for first-class modules. In *Fifth Symposium on Dynamic Languages*, pp. 27–38, 2009.

[5] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *European Symposium on Programming*, pp. 32–46, 2009.

[6] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Second Symposium on Dynamic Languages*, pp. 964–974, 2006.

[7] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, pp. 395–406, 2008.

TLDI'10, January 23, 2010, Madrid, Spain.
ACM 978-1-60558-891-9/10/01.