Chapter XI

Static Type Systems: From Specification to Implementation

Pablo E. Martínez López,
LIFIA, Facultad de Informática, UNLP, Argentina

Abstract

Static type systems are fundamental tools used to determine properties of programs before execution. There exist several techniques for validation and verification of programs based on typing. Thus, type systems are important to know for the practitioner. When designing and implementing a technique based on typing systems, there is usually a gap between the formal tools used to specify it, and the actual implementations. This gap can be an obstacle for language designers and programmers. A better understanding of the features of a type system and how they are implemented can help enormously to the good design and implementation of new and powerful verification methods based on type systems. This chapter addresses the problem of specifying and implementing a static type system for a simple language, but containing many of the subtleties found in bigger, mainstream languages. This contributes to the understanding of the techniques, thus bridging the gap between theory and practice. Additionally, the chapter contains a small survey of some of the existing developments in the static typing area and the static analysis based on typing.

Copyright © 2007, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.
Introduction

When someone speaks about verification and validation of programs, it is not very common that a static type system comes to one’s mind. However, static typing techniques have been a foundation for many of the developments in the theory and practice of this area of computing science.

A static type system (Cardelli, 1997; Cardelli & Wegner, 1985; Curry & Feys, 1958; Hindley, 1969; Hindley, 1995; Milner, 1978; Pierce, 2002) is a fundamental tool used to determine properties of programs before execution — for example, the absence of some execution errors — and to provide rudimentary documentation when coding solutions in a programming language. The main motivation for the use of these systems is that every program with a type calculated statically — that is, based only on the text of the program and not on its computation — is free from some kinds of errors during execution. The classic motto coined by Robin Milner is “well-typed programs cannot go wrong” (1978). This guarantees a certain correctness of the code and also helps omit specific checks in the executable code to avoid these kinds of problems, thus obtaining a more efficient program.

When designing static type systems, a trade-off between expressiveness and decidability has to be made. The computation prescribed by a static type system is limited — it may happen that given a type system, a decidable inference algorithm does not exist; it is usually said that the system is not decidable. So, the choices are: Design a decidable system discarding some number of correct programs, or design a precise system but with a noncomputable notion of typing. Both alternatives have been thoroughly studied. When choosing the first alternative, the goal is to maximize the expressiveness of the system — that is, the number of correct accepted programs be as large as possible, while minimizing the number of incorrect accepted ones — without losing decidability. In this line of research appear polymorphic type systems (Damas & Milner, 1982; Jim, 1996; Milner, 1978; Reynolds, 1983) as the one of ML (Clément, Despeyroux, Despeyroux, & Kahn, 1986; Rémy & Pottier, 2004) or Haskell (Peyton Jones & Hughes, 1999), and systems with subtypes (Henglein & Rehof, 1997; Mitchell, 1991; Palsberg, Wand, & O’Keefe, 1997), overloading (Jones, 1994a; Jones, 1996; Thatte, 1992), recursive types (Tiuryn & Wand, 1993), records (Gaster & Jones, 1996; Rémy, 1989), and so on.

In the second case, the idea is to design semiautomatic tools to help in the construction of programs, maximizing the ability to perform automatic type inference. In this line of research appear Girard’s System F (Girard, 1989; Reynolds, 1974), and dependent type systems as the Type Theory of Martin-Löf (Nordström, Petersson, & Smith, 1990) — on which a tool like Alf (Thorsten, Altenkirch, Gaspes, Nordström, & von Sydow, 1994) is based — and the calculus of constructions (Coquand & Huet, 1988) — on which the Coq proof assistant (Bertot & Castéran, 2004; The Coq proof assistant, 2004) is based.

This chapter concentrates on the first of these two choices.

Static type systems are an integral part of the definition of a language, similar to the grammar defining the syntax of programs. Furthermore, just as there is a tool implementing the grammar — the parser — there is a tool implementing the type system. It can be either a type checker, or a type inferencer (the difference between typechecking and type