Functional Programming with Overloading and Higher-Order Polymorphism

Mark P. Jones

Department of Computer Science, University of Nottingham, University Park,
Nottingham NG7 2RD, UK.

Abstract. The Hindley/Milner type system has been widely adopted as a basis for statically typed functional languages. One of the main reasons for this is that it provides an elegant compromise between flexibility, allowing a single value to be used in different ways, and practicality, freeing the programmer from the need to supply explicit type information.
Focusing on practical applications rather than implementation or theoretical details, these notes examine a range of extensions that provide more flexible type systems while retaining many of the properties that have made the original Hindley/Milner system so popular. The topics discussed, some old, but most quite recent, include higher-order polymorphism and type and constructor class overloading. Particular emphasis is placed on the use of these features to promote modularity and reusability.

1 Introduction

The Hindley/Milner type system [6, 19, 3], hereafter referred to as HM, represents a significant and highly influential step in the development of type systems for functional programming languages. In our opinion, the main reason for this is that it combines the following features in a single framework:

- **Type security**: soundness results guarantee that well-typed programs cannot 'go wrong'. This should be compared with the situation in dynamically typed languages like Scheme where run-time tests are often required to check that appropriate types of value are used in a particular context, and the execution of a program may terminate if these tests fail.
- **Flexibility**: polymorphism allows the use and definition of functions that behave uniformly over all types. This should be compared with the situation in monomorphically typed languages where it is sometimes necessary to produce several versions of a particular function or algorithm to deal with different types of values. Standard examples include swapping a pair of values, choosing the minimum of two values, sorting an array of values, etc.
- **Type inference**: there is an effective algorithm which can be used to determine that a given program term is well-typed and, in addition, to calculate its most general (principal) type, without requiring any type annotations in the source program. In practice, even though it is not required, programmers often choose to include explicit type information in a program as a form of