Parsing Mixfix Operators

Nils Anders Danielsson[1,⋆] and Ulf Norell[2]

[1] University of Nottingham
[2] Chalmers University of Technology

Abstract. A simple grammar scheme for expressions containing mixfix operators is presented. The scheme is parameterised by a precedence relation which is only restricted to be a directed acyclic graph; this makes it possible to build up precedence relations in a modular way. Efficient and simple implementations of parsers for languages with user-defined mixfix operators, based on the grammar scheme, are also discussed. In the future we plan to replace the support for mixfix operators in the language Agda with a grammar scheme and an implementation based on this work.

1 Introduction

Programming language support for user-defined infix operators is often nice to have. It enables the use of compact and/or domain-specific notations, especially if a character set with many symbols is used. The feature can certainly be abused to create code which the intended audience finds hard to read, but the inclusion of user-defined infix operators in a number of programming languages, including Haskell (Peyton Jones 2003), ML (Milner et al. 1997), Prolog (Sterling and Shapiro 1994), and Scala (Odersky 2009), suggests that this is a risk which many programmers are willing to take.

Some languages, such as Coq (Coq Development Team 2009), Isabelle (Paulson et al. 2008), and Obj (Goguen et al. 1999), take things a step further by supporting the more general concept of *mixfix* (also known as *distfix*) operators. A mixfix operator can have several name parts and be infix (like the typing relation $\_\vdash\_:\_$), prefix (`if_then_else_`), postfix (array subscripting: `_[_]`), or closed (Oxford brackets: $[\![\_]\!]$). With mixfix operators the advantages of binary infix operators can be taken one step further, but perhaps also the disadvantages.

An important criterion when designing a programming language feature is that the feature should be easy to understand. In the case of mixfix operators the principle explaining how to parse an arbitrary expression should be simple (even though abuse of the feature may lead to a laborious parsing process). Mixfix operators are sometimes perceived as being difficult in this respect. Our aim with this work is to present a method for handling mixfix operators which is elegant, easy to understand, and easy to implement with sufficient efficiency.

We show how to construct a simple expression grammar, given a set of operators with specified precedence and associativity (see Sect. 3). We want to avoid

⋆ The author would like to thank EPSRC for financial support.