Generic Programming with Adjunctions

Ralf Hinze

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@cs.ox.ac.uk
http://www.cs.ox.ac.uk/ralf.hinze/

Abstract. Adjunctions are among the most important constructions in mathematics. These lecture notes show they are also highly relevant to datatype-generic programming. First, every fundamental datatype—sums, products, function types, recursive types—arises out of an adjunction. The defining properties of an adjunction give rise to well-known laws of the algebra of programming. Second, adjunctions are instrumental in unifying and generalising recursion schemes. We discuss a multitude of basic adjunctions and show that they are directly relevant to programming and to reasoning about programs.

1 Introduction

Haskell programmers have embraced functors [1], natural transformations [2], monads [3], monoidal functors [4] and, perhaps to a lesser extent, initial algebras [5] and final coalgebras [6]. It is time for them to turn their attention to adjunctions.

The notion of an adjunction was introduced by Daniel Kan in 1958 [7]. Very briefly, the functors L and R are adjoint if arrows of type $\mathsf{L}\,A \to B$ are in one-to-one correspondence to arrows of type $A \to \mathsf{R}\,B$ and if the bijection is furthermore natural in A and B. Adjunctions have proved to be one of the most important ideas in category theory, predominantly due to their ubiquity. Many mathematical constructions turn out to be adjoint functors that form adjunctions, with Mac Lane [8, p.vii] famously saying, "Adjoint functors arise everywhere."

The purpose of these lecture notes is to show that the notion of an adjunction is also highly relevant to programming, in particular, to datatype-generic programming. The concept is relevant in at least two different, but related ways.

First, every fundamental datatype—sums, products, function types, recursive types—arises out of an adjunction. The categorical ingredients of an adjunction correspond to introduction and elimination rules; the defining properties of an adjunction correspond to β-rules, η-rules and fusion laws, which codify basic optimisation principles.

Second, adjunctions are instrumental in unifying and generalising recursion schemes. Historically, the algebra of programming [9] is based on the theory of initial algebras: programs are expressed as folds, and program calculation is based on the universal property of folds. In a nutshell, the universal property

J. Gibbons (Ed.): Generic and Indexed Programming, LNCS 7470, pp. 47–129, 2012.

formalises that a fold is the unique solution of its defining equation. It implies computation laws and optimisation laws such as fusion. The economy of reasoning is further enhanced by the principle of duality: initial algebras dualise to final coalgebras, and correspondingly folds dualise to unfolds. Two theories for the price of one.

However, all that glitters is not gold. Most, if not all, programs require some tweaking to be given the form of a fold or an unfold and thus make them amenable to formal manipulation. Somewhat ironically, this is in particular true of the "Hello, world!" programs of functional programming: factorial, the Fibonacci function and append. For instance, append does not have the form of a fold as it takes a second argument that is later used in the base case.

In response to this shortcoming a plethora of different recursion schemes has been introduced over the past two decades. Using the concept of an adjunction many of these schemes can be unified and generalised. The resulting scheme is called an adjoint fold. A standard fold insists on the idea that the control structure of a function ever follows the structure of its input data. Adjoint folds loosen this tight coupling—the control structure is given implicitly through the adjunction.

Technically, the central idea is to gain flexibility by allowing the argument of a fold or the result of an unfold to be wrapped up in a functor application. In the case of append, the functor is essentially pairing. Not every functor is admissible: to preserve the salient properties of folds and unfolds, we require the functor to have a right adjoint and, dually, a left adjoint for unfolds. Like folds, adjoint folds are then the unique solutions of their defining equations and, as is to be expected, this dualises to unfolds.

These lecture notes are organised into two major parts. The first part (Section 2) investigates the use of adjunctions for defining 'data structures'. It is partly based on the "Category Theory Primer" distributed at the Spring School [10]. This section includes some background material on category theory, with the aim of making the lecture notes accessible to readers without specialist knowledge.

The second part (Section 3) illustrates the use of adjunctions for giving a precise semantics to 'algorithms'. It is largely based on the forthcoming article "Adjoint Folds and Unfolds—An Extended Study" [11]. Some material has been omitted, some new material has been added (Sections 3.3.3 and 3.4.2); furthermore, all of the examples have been reworked.

The two parts can be read fairly independently. Indeed, on a first reading I recommend to skip to the second part even though it relies on the results of the first one. The development in Section 3 is accompanied by a series of examples in Haskell, which may help in motivating and comprehending the different constructions. The first part then hopefully helps in gaining a deeper understanding of the material.

The notes are complemented by a series of exercises, which can be used to check progress. Some of the exercises form independent threads that introduce more advanced material. As an example, adjunctions are closely related to the