Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming

Ralf Lämmel
Vrije Universiteit, Amsterdam

Simon Peyton Jones
Microsoft Research, Cambridge

Abstract

We describe a design pattern for writing programs that traverse data structures built from rich mutually-recursive data types. Such programs often have a great deal of "boilerplate" code that simply walks the structure, hiding a small amount of "real" code that constitutes the reason for the traversal.

Our technique allows most of this boilerplate to be written once and for all, or even generated mechanically, leaving the programmer free to concentrate on the important part of the algorithm. These generic programs are much more adaptive when faced with data structure evolution because they contain many fewer lines of type-specific code.

Our approach is simple to understand, reasonably efficient, and it handles all the data types found in conventional functional programming languages. It makes essential use of rank-2 polymorphism, an extension found in some implementations of Haskell. Further it relies on a simple type-safe cast operator.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Languages

Keywords

Generic programming, traversal, rank-2 types, type cast

1 Introduction

Suppose you have to write a function that traverses a rich, recursive data structure representing a company's organisational structure, and increases the salary of every person in the structure by 10%. The interesting bit of this algorithm is performing the salary-increase — but the code for the function is probably dominated by "boilerplate" code that recurses over the data structure to find the specified department as spelled out in Section 2. This is not an unusual situation. On the contrary, performing queries or transformations over rich data structures, nowadays often arising from XML schemata, is becoming increasingly important.

Boilerplate code is tiresome to write, and easy to get wrong. Moreover, it is vulnerable to change. If the schema describing the company's organisation changes, then so does every algorithm that recurses over that structure. In small programs which walk over one or two data types, each with half a dozen constructors, this is not much of a problem. In large programs, with dozens of mutually recursive data types, some with dozens of constructors, the maintenance burden can become heavy.

Generic programming techniques aim to eliminate boilerplate code. There is a large literature, as we discuss in Section 9, but much of it is rather theoretical, requires significant language extensions, or addresses only "purely-generic" algorithms. In this paper, we present a simple but powerful design pattern for writing generic algorithms in the strongly-typed lazy functional language Haskell. Our technique has the following properties:

- It makes the application program adaptive in the face of data type (or schema) evolution. As the data types change, only two functions have to be modified, and those functions can easily be generated because they are not application-specific.
- It is simple *and* general. It copes with arbitrary data-type structure without fuss, including parameterised, mutually-recursive, and nested types. It also subsumes other styles of generic programming such as term rewriting strategies.
- It requires two extensions to the Haskell type system, namely (a) rank-2 types and (b) a form of type-coercion operator. However these extensions are relatively modest, and are independently useful; they have both been available in two popular implementations of Haskell, GHC and Hugs, for some time.

Our contribution is one of synthesis: we put together some relatively well-understood ideas (type-safe cast, one-layer maps) in an innovative way, to solve a practical problem of increasing importance. The paper should be of direct interest to programmers, and library designers, but also to language designers because of the further evidence for the usefulness of rank-2 polymorphic types.

The code for all the examples is available online at:

`http://www.cs.vu.nl/Strafunski/gmap/`

The distribution comes with generative tool support to generate all datatype-specific boilerplate code. Our benchmarks show that it is possible to get the run-time performance of typical generic programs reasonably close to the hand-coded boilerplate-intensive counterparts (Section 10).