Automatic Testing of Higher Order Functions

Pieter Koopman and Rinus Plasmeijer

Nijmegen Institute for Computer and Information Science, The Netherlands
{pieter, rinus}@cs.ru.nl

Abstract. This paper tackles a problem often overlooked in functional programming community: that of testing. Fully automatic test tools like Quickcheck and G∀ST can test first order functions successfully. Higher order functions, HOFs, are an essential and distinguishing part of functional languages. Testing HOFs automatically is still troublesome since it requires the generation of functions as test argument for the HOF to be tested. Also the functions that are the result of the higher order function needs to be identified. If a counter example is found, the generated and resulting functions should be printed, but that is impossible in most functional programming languages. Yet, bugs in HOFs do occur and are usually more subtle due to the high abstraction level.

In this paper we present an effective and efficient technique to test higher order functions by using intermediate data types. Such a data type mimics and controls the structure of the function to be generated. A simple additional function transforms this data structure to the function needed. We use a continuation based parser library as main example of the tests. Our automatic testing method for HOFs reveals errors in the library that was used for a couple of years without problems.

1 Introduction

Automatic test tools for functional languages are able to generate test cases, execute the associated tests and derive a verdict from the test results. Basically a predicate of the form $\forall x \in X : P(x)$ is replaced by a function `P :: X → Bool`. The predicate is tested by evaluating the function `P` for a large number of elements of type `X`. In Quickcheck these elements are generated in pseudo random order by a user defined instance of a type class. G∀ST has a generic algorithm that is able to generate elements of any type in a systematic way [6]. The user can specify any other algorithm if the generic algorithm is inappropriate.

The advantages of this automatic testing is that it is cheap and fast. Moreover, the real code is tested. A inherent limitation of testing is that a proof by exhaustive testing is only possible for finite types (due to generation algorithm used, Quickcheck is not able to determine when all elements are tested and never detects that a property is proven by exhaustive testing). A formal proof of a property gives more confidence, but usually works on a model of the program instead of the program itself and requires (much) user guidance. Hence, both formal proofs and testing have their own value. It is at least useful to do a quick automatic test of some property before investing much effort in a formal proof.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 148–164, 2006.

The generation of elements of a type works well for (first order) data structures. Testing properties of HOFs requires functions as test argument and hence the generation of functions by the test system. The possibilities to generate functions are rather limited. In Quickcheck functions of type `A → B` are generated by transforming elements of type `A` to an integer by a user defined instance of the class `coarbitrary`. This integer is used to select an element of type `B`. A multi-argument function of type `A → B → C` is transformed to a function `B → C` by providing a pseudo randomly generated element of type `A`. In this way all information of all arguments is encoded in a single integer. This approach is not powerful enough for more complex functions, and has as drawback that it is impossible to print these functions in a decent way. G∀ST used the same approach with the difference that functions can be derived using a generic algorithm. Using an extensional representation of functions, by providing explicit input-output pairs, is unsuited for large data types since it is usually impossible to determine the arguments that will occur.

In this paper we show how functions of the desired form can be generated systematically. The key step is to represent such a function by its abstract syntax tree, AST. This AST is represented as algebraic data type. Its instances can be generated automatically by G∀ST in the usual way. It is simple to transform the AST to the desired function. An additional advantage of using a data type as AST is that this can be printed in a generic way as well, while printing functions is impossible in functional languages like Haskell and Clean.

We illustrate this technique with a full fleshed parser combinator library. In [4] we introduced a library of efficient parser combinators. Using this library it is possible to write concise, efficient, recursive descent parsers. The parsers can be ambiguous if that is desired. Basically there are two ingredients that makes the constructed parsers efficient. First, the user can limit the amount of backtracking by a special version of the choice combinator that only yields a single result. Second, the implementation of the combinators uses continuations instead of intermediate data structures. Especially when parsed objects are processed in a number of steps before a final parse result is produced, continuation based parsers are faster than a straight forward implementation of parsers.

The price to be paid for using continuations instead of intermediate data structures, is that the implementation of the combinator becomes more complicated. Each parser has three continuations, and some of these continuations have their own continuation arguments. The parser combinators manipulate these continuations in a rather tricky way. However, the use of the combinators is independent of their implementation, and is not different for a library with a simple implementation using intermediate data types. The published combinators are tested manually by the authors and checked by many users of the library. Much to our surprise last year some errors in the library were found.

After improving the combinators we wanted to obtain more confidence in the correctness of the library. Manual testing by a number of typical examples was clearly insufficient. Using the techniques described here it was possible to test this library automatically. During these test an additional error was found.