Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is an author's version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/32303

Please be advised that this information was generated on 2018-07-07 and may be subject to change.

Testing Higher Order Functions

Pieter Koopman and Rinus Plasmeijer

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

Extended Abstract

Abstract. Automatic testing of first order functions works fine. Testing higher order functions automatically is more troublesome, it is harder to generate the functions needed as argument automatically, and these generated functions cannot be shown when a counterexample is found. Nevertheless, higher order functions can contain errors and hence need to be tested.
In this paper we present effective and efficient techniques to test higher order functions using intermediate data types. This 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. Using the data types the test engineer can control the generation of functions and print them. We use a continuation based parser library as example. Automatic testing reveals errors is the published library that was used for a couple of years without problems.

1 Introduction

In [4] we introduced an library of efficient parser combinators. Using this library it is possible to write concise recursive descent parsers that can be ambiguous if that is desired, and are efficient. Basically there are two ingredients that makes the constructed parsers efficient. First, the user limits 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.

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 has their own continuation arguments. The basic parsers and parser combinators manipulates these continuations in a rather tricky way. The use of the combinators is independent of their implementation, and has not changed compered to the library with a simple implementation using intermediate data types. The published combinators are checked by the authors, the referees of the associated publication, and many users of the library. Much to our surprise last year some errors in the library were found.

After improving the combinators, we wondered if the mythical last error was removed, or additional errors were hidden in the library. Perhaps, correcting

this error even introduced new errors. Of course some ad hoc testing of the new library was done, but experience showed that this is not enough. Systematic and automatic testing, with a tool like Quickcheck [3] or G∀ST [5], would be much better. Automatic testing requires appropriate properties, generation of the required continuation parsers, and a way to show these functions if an error would be found. Due to the complicated types of the parser combinators, the assumptions about the way the continuations are handled, and the need to print the parser fully automatic generation is not feasible. In this paper we will show how we can solve these problems by defining an intermediate data type and a simple translation function of instances of the intermediate data type to the required functions. For the generation of instances of the intermediate data type and showing them, the existing generic capabilities of the test system are used. This approach works in a broad range of situations where one needs to generate specific functions and to print them in order test higher order functions.

In section 2 we will shortly review the continuation based parser combinators. In the next sections we will show how these combinators can be tested individually. In section 4 we will show how to test the entire library based on the generation of a grammar. Using a property of the famous *fold*-function we demonstrate that this approach works also in other situations. Finally there is a discussion.

2 Background: Continuation Based Parser Combinators

Each continuation parser has four arguments:

1. The success continuation which determines what will be done if the current parser succeeds. This function gets the result of the current parser, the other continuations and the remaining input as its arguments.
2. The XOR-continuation is a function that tells what has to be done if only a single result of the parser is needed.
3. The OR-continuation determines the behavior when all possible results of the parser are needed.
4. The list of symbols to be parsed. In this paper these symbols will be characters. In general also lists of more complex tokens can be parsed.

The result of a parser is a list of tuples containing the remaining input and the results of parsing the input until this point.

As an example the type of the continuation parser `p1 = symbol '*'`, that succeeds if the first character in the input is `*`, is:

```
p1 :: (Char→([([Char],a)]→[([Char],a)])→[([Char],a)]→[Char]→[([Char],a)])
        → ([([Char],a)]→[([Char],a)])
        → [([Char],a)]
        → [Char]
        → [([Char],a)]
```

That is the result of applying `begin p1` to the input `['*abc']` will be `[(['abc'],'*')]`, while applying it to the input `['abc']` yields the empty list of results.

The function `begin` turns a continuation parser into a parser, here of type `[Char]→[([Char],Char)]`, by providing appropriate initial continuations.

See [4] for more information in this extended abstract. The full paper will contain a more complete review of the continuation based parser combinators.

3 Testing Parser Combinators

The parser combinator library contains a number of basic combinators for tasks like recognizing symbols in the input and yielding specific values. As an example we consider the parser combinator `symbol :: s → CParser s s t | == s` that should recognize the given symbol `s` in the input. A desirable property of `symbol` is that it should yield a single success when the input list starts with the given symbol. For characters as input tokens, this can be specified in G∀ST as:

```
propSymbol :: Char [Char] → Bool
propSymbol c l = begin (symbol c) [c:l] == [(l,c)]
```

Using `begin (symbol c)` instead of `symbol c` in the test makes it possible to compare parse results (lists of tuples), instead of comparing higher order functions.

This property can be tested directly by G∀ST. The result of the test is that it passes any number of tests. When we restrict the input to, for instance, lists of two characters such a property can even be proven. The property for inputs of exactly two character reads:

```
propSymbolF :: Char Char → Bool
propSymbolF c d = begin (symbol c) [c,d] == [([d],c)]
```

Within one second G∀ST proves this property by executing all possible tests.

Although this kind of properties states clearly the intended semantics of the basic parser combinators and the associated tests are useful, this does not capture the signaled problems with the combinator library.

Also for the parser combinators that compose continuation parsers, one can specify properties. The combinator `<|>` indicates a choice between the parsers given as left and right arguments. In the framework of nondeterministic parsers this implies that the result of applying `p <|> q` to some input is equal to the concatenation of results from `p` to that input and applying `q` to that input. Stated as property for G∀ST this is:

```
propOR p q input = begin (p <|> q) input == begin p input ++ begin q input
```

Again we use `begin p input` rather than `p` to be able to compare resulting data structures instead of comparing higher order functions.

In this property `p` and `q` are continuation parsers and input is list of input tokens. G∀ST has no problems with the generation of lists of characters that can be used as input.

After the first versions of G∀ST, the possibility of generating functions was removed. The generation of functions caused serious overall degrease of performance and works only satisfactory in a limited number of situations. Continuation based parser combinators heavily relay on the proper treatment of the

continuations by each and every parser. Not every function having the type of a parser is a correct parser. This implies that generic generation of parsers by the default generic algorithm of G∀ST is not adequate, even it would be reintroduced in G∀ST.

Nevertheless, a number of suitable parsers is needed for the variables `p` and `q` in order to test the property `propOR`. Specifying these parsers by hand is unattractive since it requires unnecessary human effort, and limits the testing to the listed parsers. Moreover, the listed functions cannot be printed. This implies that if an counterexample would be found by G∀ST, it can only print the argument `p` and `q` as `<function>`.

Using some low-level wizarding with cycles it is possible to define an infinite list of more and more complex continuation parsers and their textual representation. Although this solves the current problem, it is inelegant and hard to port to other situations.

Using an data type representing the grammar to be recognized by the parser and a simple conversion function from the data type to the parser, the problem of generating continuation parsers can be solved elegantly. A data type that represents parsers that consumes lists of characters and yield a character is:

```
:: P
   = Fail          // basic operator: fails for any input
   | Yield  Sym    // basic operator: yields the specified symbol for any input
   | Symbol Sym    // basic operator: recognize the specified symbol, see above
   | Or P P        // concatenation of the successes of both parsers
   | XOr P P       // successes of second parser if first parser fails
   | ANDR P P      // results of 2nd parser if parsers can be applied in given order
   | ANDL P P      // results of 1st parser if parsers can be applied in given order

:: Sym = Char Char  // Symbols are just characters
```

The generation of instances of these data types is straightforward. The default generic generation algorihm `ggen` is used for the data type `P` representing the structure of the parser. For the type symbol we use only the characters `'a'` and `'x'` in order to limit the number of characters used in the tests. This increases the number of more complicated parses used in a finite number of tests.

```
derive ggen P
ggen {|Sym|} n r = [Char 'a', Char 'x']
```

Via a direct mapping instances of the data type `P` can be transformed to the corresponding continuation parsers.

```
PtoPC :: P → (CParser Char Char Char)
PtoPC Fail              = fail
PtoPC (Yield (Char c))  = yield c
PtoPC (Symbol (Char c)) = symbol c
PtoPC (Or p q)          = PtoPC p <|> PtoPC q
PtoPC (XOr p q)         = PtoPC p <!> PtoPC q
PtoPC (ANDR p q)        = PtoPC p  &> PtoPC q
PtoPC (ANDL p q)        = PtoPC p <&  PtoPC q
```

Now we can state a property that can be used to test the parser combinator <|> with G∀ST:

```
propOR :: P P [Char] → Bool
propOR x y chars = begin (p <|> q) chars == begin p chars ++ begin q chars
where p = PtoPC x; q = PtoPC y
```

Since x and y are instances of the data type P, printing them by the generic mechanism of G∀ST reveals the structure of the combinator parsers used in the actual test clearly.

Testing such a property in G∀ST is quick. Testing this property for the first 1000 combinations of arguments takes only 0.6 seconds on a quite modest PC.

In the same spirit we can test the other combinators in the original combinator library. For instance the or-combinator <!>, that only applies the second parser if the first one fails has to obey the property:

```
propXOR :: P P [Char] → Bool
propXOR x y chars
    | isEmpty (begin p chars)
        = begin (p <!> q) chars == begin q chars
        = begin (p <!> q) chars == begin p chars
where p = PtoPC x; q = PtoPC y
```

Testing this property reveals the problems with the original parser combinator library. One of the counterexamples found is for (Or (Yield (Char 'x')) Fail) as the value of x (Yield (Char 'a')) for ys, and the empty input []. This is equivalent to the reported error that initiates this research.

In the same way we specified properties for the other operators. Using these properties we did not found any counterexamples in the new version of the library. This is considered as a strong indication that changes of the library are really an improvement.

4 Testing Parsers

The approach outlined in the previous section works fine, but can be improved at some points. Most of the generated inputs will be rejected by the generated parsers. In order to test effectively wether the parsers yield the correct results we can generate inputs based on the grammar to be accepted. All inputs accepted by a grammar can be generated by:

```
PtoInput :: P → [[Char]]
PtoInput Fail              = []
PtoInput (Yield (Char c))  = [[]]
PtoInput (Symbol (Char c)) = [[c]]
PtoInput (Or p q)          = removeDup (PtoInput p ++ PtoInput q)
PtoInput (XOr p q)         = removeDup (PtoInput p ++ PtoInput q)
PtoInput (ANDR p q)        = [i++j \\ i←PtoInput p, j←PtoInput q]
PtoInput (ANDL p q)        = [i++j \\ i←PtoInput p, j←PtoInput q]
```

Since our grammars only contain or- and and-combinators, each grammar accepts only a finite amount of inputs. If the list of inputs can be infinite, for instance if we include a Kleene star in the grammar, in order to guarantee termination.

Given a grammar and an input, it is easy to determine what the result of the parser should be:

```
results :: P [Char] → [([Char],Char)]
results Fail              chars = []
results (Yield (Char c))  chars = [(chars,c)]
results (Symbol (Char c)) []    = []
results (Symbol (Char c)) [d:r]
    | c == d
        = [(r,c)]
        = []
results (Or p q)  chars = results p chars ++ results q chars
results (XOr p q) chars
 = case results p chars of
    [] = results q chars
    r  = r
results (ANDR p q) chars
 = [ t \\ (c2,_)←results p chars, t←results q c2]
results (ANDL p q) chars
 = [ (c3,a) \\ (c2,a)←results p chars, (c3,_)←results q c2]
```

Using these tools it is very easy to verify the parsers associated to some syntax tree directly. For all inputs to be accepted by the parser, the results calculated by the function `results` above should be equal to the results obtained by applying the parser associated to the grammar to that input:

```
propP :: P → Bool
propP t = and [ results t i == begin (PtoPC t) i \\ i←PtoInput t ]
```

Also this property finds the counterexamples for the original version of the library very quickly.

In order to verify the error detecting capacity of this approach we made, by hand, 25 mutants of the library that are approved by the type system. Testing these libraries revealed counterexamples for each of these libraries within 2 seconds.

5 Testing other Higher Order Functions

In order to show that this approach can also be used for other higher order functions a property of the famous *fold* function will be tested.

The property is based on the universal property of the *fold* as stated by Malcom [6] and is based on the Bird-Meertens theory of lists [1, 7]. For any function f, elements v and e, and list l we require that $fold\ f\ v\ [e : l] = f\ e\ (fold\ f\ v\ l)$. In order to test several implementations of the *fold*function we make it an argument of the property in G∀ST. We want to specify this argument in an actual test.

The other arguments are intended as universal quantified variables and need to be generated by G∀ST. The type CLEANa a-¿a indicates the type of the higher order function f that needs to be generated in the tests.

```
propFold :: ((a a→a) a [a]→ a) (a a→a) a [a] a → Bool | == a
propFold fold f v l e = fold f v [e:l] == f e (fold f v l)
```

In order to test this with G∀ST we need to choose same data type for `a`. We will use integers here, and choose `v` to be zero.

```
propFoldInt :: ((Int Int→Int) Int [Int]→ Int) Expr [Int] Int → Bool
propFoldInt fold ex l e = propFold fold (toFun ex) 0 l e
```

Similar to above, the data type `Expr` is used to represent the functions to be generated:

```
:: Expr
   = X
   | Y
   | ConstOne
   | SUM  Expr Expr
   | DIFF Expr Expr
```

The functions `toFun` converts instances of this type to functions:

```
toFun :: Expr → (Int Int → Int)
toFun X          = λx y.x
toFun Y          = λx y.y
toFun ConstOne   = λx y.1
toFun (SUM  a b) = λx y.toFun a x y+toFun b x y
toFun (DIFF a b) = λx y.toFun a x y-toFun b x y
```

As we might expect the functions `foldr` from the standard library appears to be a valid *fold*function if we test it with:

```
Start = test (propFoldInt foldl)
```

The function `foldl` however, does not obey this property for functions like, `f x y = x`, `f x y = y`, and `f x y = x+x`. Although this result in itself is not new, it demonstrates the power of this approach to test higher order functions.

6 Discussion

The test systems G∀ST was very suited to test properties over first order functions. Generating the functions needed for testing properties over higher order functions was troublesome. In this paper we have shown that this can be solved by generating a grammar for the desired functions as data type, and a very simple function that transforms this data type to the corresponding function.

The stated properties identified the errors in the original version of the library, as well as more than two dozen of different errors injected deliberately in order investigate the power of automatic testing.

This approach can be used also in other situations where higher order functions needs to be tested in order to generate and show the needed functions as test argument.

Acknowledgement

We thank Erik Zuurbier and Arjen van Weelden for indicating problems with the parser library initiating this research.

References

1. Richard Bird. *Constructive Fununctional Programming*, Proc. MArktoberdorf International Summerschool on Constructive Methods in Computer Science ,1989.
2. A. Alimarine, R. Plasmeijer. *A Generic Programming Extension for Clean.* IFL2001, LNCS 2312, pp.168–185, 2001.
3. K. Claessen, J. Hughes. QuickCheck: A lightweight Tool for Random Testing of Haskell Programs. ICFP, ACM, pp 268–279, 2000. See also `www.cs.chalmers.se/~rjmh/QuickCheck`.
4. Pieter Koopman and Rinus Plasmeijer: *Efficient Combinator Parsers* In Proc. of Implementation of Functional Languages (IFL '98), K. Hammond, A.J.T. Davie and C. Clack (Eds.), LNCS 1595, pp. 120–136. 1999
5. Pieter Koopman, Artem Alimarine, Jan Tretmans and Rinus Plasmeijer: *Gast: Generic Automated Software Testing*, in Ricardo Peña: *IFL 2002, Implementation of Functional Programming Languages*, LNCS 2670, pp 84–100, 2002.
6. Grant Malcom. *Algebraic Data Types and Program Transformations*, Ph.D. Thesis, 1990.
7. Lambert Meertens. *Algorithmics: Towards programming as a mathematical acitvity*, Proc. CWI Symposium 1983.
8. Rinus Plasmeijer and Marko van Eekelen: *Concurrent Clean Language Report (version 2.1.1)*, 2005. `www.cs.ru.nl/~clean`.