# 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 en 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 class. G∀ST has a generic algorithm that is able to generate elements of any type in a systematic way [7]. The user can specify any other algorithm if desired.

The advantages of this automatic testing is that it is cheap and fast. Moreover, the real code is tested. A inherent limitation of teting 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.

The generation of elements of a type works very 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 descent way. G∀ST used the same approach with the difference that functions can be derived using a generic algorithm.

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, which therefor can be generated automatically by G∀ST in the usual way. The AST is transformed to the desired function by a very simple transformation. 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 [5] 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 parser 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 manipulates 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.

It turns out that a similar representation of functions by data types is used at different places in the literature. The technique is called *defunctionalisation*,

and the function transforming the data type is usually called *apply*. This technique was introduced by Reynolds [11], and repopularized by Danvy [4]. Using defunctionalisation for generating functions and testing is new.

In section 2 we will shortly review the continuation based parser combinators. In the next section we show how functions as result can be tested for equivalence by applying them to appropriate arguments. The generation of functions as argument is treated in section 4. In section 5 we show that the effectiveness of tests can be improved by generating tailor made inputs for the parsers. In section 6 we will show how to test the entire library by defining one property for parsers, instead of by properties for individual combinators. By testing a property of the famous *fold*-function we demonstrate that our approach works also in other situations. Finally there is a conclusion.

## 2 Background: Continuation Based Parser Combinators

In order to make this paper self contained we repeat the most important parser combinators from [5]. In the continuation parser library [5] 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, but 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. This is reflected in the types:

```
:: Parser s r :== [s] → ParsResult s r
:: ParsResult s r :== [([s],r)]

:: CParser s r t:==(SucCont s r t) (XorCont s t) (AltCont s t)→Parser s t
:: SucCont s r t:==r (XorCont s t) (AltCont s t) → Parser s t
:: XorCont s t  :==(AltCont s t) → ParsResult s t
:: AltCont s t  :==ParsResult s t
```

As an example the type of the continuation parser `p = symbol '*'`, that succeeds if the first character in the input is `*`, is `CParser Char Char a`. Expanding this type to basic types yields:

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

This complicated type indicates that testing for first order properties is inadequate. The definition of the parser combinator `symbol` is:

```
symbol :: s → CParser s s t | == s
symbol s = psymbol
where psymbol sc xc ac [x:ss] | x == s = sc s xc ac ss
      psymbol sc xc ac _               = xc ac
```

The function `begin` turns a continuation parser into a standard parser by providing appropriate initial continuations. The parser takes a list of tokens as arguments and produces a list of successes. Each success is a tuple containing the remaining input tokens and the parse result.

```
begin :: (CParser s t t) → Parser s t
begin p = p (λx xc ac ss . [(ss,x):xc ac]) id []
```

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

The concatenation of two parsers, `p <&> q`, requires that the parser `q` is applied to the rest of the input left by the parser `p`. This is done by inserting `q` in the success continuation of `p`. The result of `p` is given as the first argument to `q`.

```
(<&>) infixr 6 :: (CParser s u t) (u → CParser s v t) → CParser s v t
(<&>) p q = λsc . p (λt . q t sc)
```

There are several variants of the operator `<&>`: the operator `<&` yields only the result of `p`, `&>` yields only the result of `q`, `<:&>` construct a list with the result of `p` as head and the result of `q` as tail, `<++>` appends the results of `p` and `q`, `<!&>` removes the XOR-alternatives if `p` succeeds.

The construct `p <|> q` indicates that we want all results of `p` and all results of `q`. This is achieved by putting `q` in the alternative continuation `ac` of `p`.

```
(<|>) infixr 4 :: (CParser s r t) (CParser s r t) → CParser s r t
(<|>) p q
 = λ sc xc ac ss .p (λx xc1.sc x id) id (q (λx xc1.sc x id) xc ac ss) ss
```

The operator `<!>` yields only the result of `q` if `p` has no results. This is done by putting `q` in the XOR-continuation `xc` of `p`. The success continuation of `p` takes care of removing `q` if `p` succeeds.

```
(<!>) infixr 4 :: (CParser s r t) (CParser s r t) → CParser s r t
(<!>) p q = λ sc xc ac ss
            .p (λx xc2.sc x id) (λ_.q (λx xc3.sc x id) xc ac ss) ac ss
```

The combinator `<@` applies the function `f` to the items recognized by parser `p`.

```
(<@) infixl 5 :: (CParser s r t) (r→u) → CParser s u t
(<@) p f = λ sc . p (sc o f)
```

The operator `<*>` mimics the Kleene star: it repeats parser `p` as often as possible. The results of all applications of `p` are collected in a list. It behaves like:

```
<*> :: (CParser s r t) → CParser s [r] t
<*> p = (p <&> λr . <*> p <@ λrs . [r:rs]) <!> yield []
```

## 3 Functions as result of higher order functions

Testing higher order functions that yield functions as result is relatively easy. The test system has to verify whether the correct function is produced. In most functional programming languages it is impossible to look inside functions (LISP is an exception). Hence it is impossible to decide if this function is the desired one by inspecting the function directly.

More importantly, for functions we are usually not interested in the exact definition of the function, but in its behavior. Any definition will do, if it produces the right function result to the given parameters. This implies that even if it would be possible to look inside a function directly, this would not help us. We are interested in the input/output behavior of the function instead of the algorithm it uses.

Chancing the function to be tested in such a way that it delivers a data structure instead of a function is an unattractive option: we want to test the software as it is and this does not solve the problem of testing the behavior instead of the actual definition.

Testing functions for equal input output relations is relative easy. As example we consider the function `isAlpha` and the function `isUpperOrLower` defined as

```
isUpperOrLower :: Char → Bool
isUpperOrLower c = isUpper c || isLower c
```

Using G∀ST the equivalence of the functions `isAlpha` and `isUpperOrLower` can be tested by stating a property stating that $\forall c\,.\,isAlpha\ c = isUpperOrLower\ c$. In Clean this property reads:

```
propEq :: Char → Bool
propEq c = isAlpha c == isUpperOrLower c
```

Testing this in G∀ST is done by executing `Start = test propEq`. G∀ST *proves* this property by exhaustive testing: the function `propEq` is evaluated for all possible characters. Since the number of characters is finite (and small), G∀ST is able to test it for all possible arguments and to yield *Proof* rather than *Pass* (the latter indicates a successful test for all arguments used).

In the next section we show how this approach is used to compare parsers by applying them to various inputs and comparing the results.

### 3.1 Testing basic 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 yields 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.

The property `propSymbol` can be tested directly by G∀ST by applying the function `test` to the property in the `Start`-function. 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:

```
propSymbol2 :: Char Char → Bool
propSymbol2 c d = begin (symbol c) [c,d] == [([d],c)]
```

Within a split second G∀ST proves this property by executing all possible tests. All measurements in this paper are done on a fairly moderate PC running the latest windows XP, Clean 2.1.1 and G∀ST 0.5.1.

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.

## 4 Functions as argument of higher order functions

Testing properties over higher order functions that have functions as argument is a harder problem. In these properties there is a universal quantification over functions. This implies that the test system must supply appropriate functions as argument.

A typical example of a property over higher order functions is:

$$\forall f,\, g : (x \to y)\,.\, \forall l : [x]\,.\, map\ f\ (map\ g\ l) = map\ (f \circ g)\ l.$$

For any test we need to chose concrete types for $x$ and $y$. Choosing small finite types like `Bool` or `Char` usually give good test results. The Clean version of this property where all types are `Char` is:

```
propMap :: (Char→Char) (Char→Char) [Char] → Bool
propMap f g l = map f (map g l) == map (f o g) l
```

Former versions of G∀ST where able to generate functions. The generated function of type `X→Y` converts the argument `x` to an index in a list of values `ys` of type `Y`: `λx . ys !! (toIndex x rem length ys)`. For simple functions (like `f` and `g` in `propMap`) this is adequate, but not for more complex functions (like continuation parsers). Moreover, in the generic framework the generation of values and the index function needs to be coupled. This slows down the generation of ordinary values considerable. For these reasons the existing generation of function algorithm was removed from G∀ST.

Another serious problem is that the code of a given function cannot be shown. This implies that if an counterexample would be found by G∀ST, it can only print the argument `f` and `g` as `<function>`.

As a solution for the problem of generating functions and printing them we propose to use a tailor made data structure that exactly determines the functions

that are needed in a particular test context. Instances of this data structure can be generated by the default generic algorithm used in G∀ST. Since the data type determines the needed functions exactly, the conversion from a generated instance of the data type to the corresponding function is very easy.

As example we will show how the property for the `map` function can be tested. Apart from the library functions `toUpper` and `toLower` we will use the functions `rot` and `shift` in the tests. The function `rot` rotates characters in the alphabet `n` places in the alphabet and does not change other characters, `shift` shift any character `n` places in the ascii table. These functions are defined as:

```
rot :: Int Char → Char
rot n c
    | isUpper c = 'A' + toChar ((fromChar (c-'A') + (abs n)) rem 26)
    | isLower c = 'a' + toChar ((fromChar (c-'a') + (abs n)) rem 26)
                = c

shift :: Int Char → Char
shift n c = toChar (abs (fromChar c + n) rem 256)
```

A data type representing all functions that we want to be generated as test argument and the corresponding conversion function are defined as:

```
:: Fun = Rot Int | Shift Int | ToUpper | ToLower

class apply s t :: apply s → t
instance apply Fun (Char → Char)
where
    apply (Rot n)   = rot n
    apply (Shift n) = shift n
    apply ToUpper   = toUpper
    apply ToLower   = toLower
```

We will use the class `apply` for any transformation of a data type to the corresponding function in this paper.

Now we are able to test the property for the `map` function. Instances of the type `Fun` are generated by deriving the generic generation by **derive** `ggen Fun`. Instances of this data type are converted to functions by applying `apply` to them. In `propMap2` we reuse `propMap`, the needed functions are obtained from the type `Fun`. Finally, there is a `Start`-function initiating the testing.

```
propMap2 :: Fun Fun [Char] → Bool
propMap2 f g l = propMap (apply f) (apply g) l

Start = test propMap2
```

This property passes any number of tests. In the next section we will show how this principle can be applied to continuation parsers. In order to obtain more complex parsers, the data type to represent functions will be recursive.

### 4.1 Testing parser combinators

Also for the parser combinators that compose continuation parsers, one can specify properties in the way just explained. For example the result of applying `p <|> q` to some input is equal to the concatenation of results from `p` to the same 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
```

The generation of continuation parsers needed as arguments `p` and `q` is again done with a data type and a corresponding instance of `apply`. The type `P` is a recursive data type that represents parsers that consumes lists of characters and yield a character as result.

```
:: 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 constructor Char and a character
```

The generation of instances of these data types is straightforward. The default generic generation algorithm `ggen` of G∀ST is used for the data type `P` representing the structure of the parser. For the type `Sym` we use only the characters `'a'` and `'b'` 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 'b']
```

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

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

The property to test the parser combinator `<|>` using the type `P` becomes:

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

Since the continuation parsers `x` and `y` are now represented by 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. If desired we can make a tailored instance of `genShow {|P|}` that prints the data type exactly as the functions generated by `apply`, instead of deriving the default behavior.

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

In the same spirit we can test the other combinators in the original combinator library. For instance the xor-combinator, `<!>`, only applies the second parser if the first one fails. This is expressed by the property `propXOR`:

```
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 = apply x; q = apply y
```

Testing this property reveals the problems with the original parser combinator library. One of the counterexamples found is for `(Or (Yield (Char 'b')) Fail)` as the value of `x`, `(Yield (Char 'a'))` for `y`, and the empty input `[]`. The problem is that `begin ((yield 'b' <|> fail) <!> yield 'a') []` produces the result `['ba']` instead of the desired result `['b']`. This is equivalent to the reported error that initiates this research. Since this is a unusual combination of parser combinators its in not strange that this issue was not discovered during manual tests and ordinary use of the library.

**Repetition of parsers** The parsers generated and tested above do not contain the repetition operators `<*>`. Although it is very easy to add the desired constructors to the type `P` and the function `apply`, certain instances of the generated parsers can cause serious problems. For example, the parser `<*> (yield 'a')` will produce an infinite list of `'a'`s without consuming input.

We do want to incorporate parsers containing proper applications of the operator `<*>` in our tests. This implies that we either have to prevent that parsers causing problems as illustrated above are generated (by designing a more sophisticated data type), or we have to prevent that they are actually used in the tests (by a precondition in the property). Both solutions are feasible. The selection of parsers that behave well is somewhat simpler and will be used here. Selection of well behaving parsers is done by inspection of the corresponding data structure and the operator ⟹ from G∀ST.

First we add appropriate clauses to the type `P` and the function `apply`. Since we have now a repetition it is more convenient to generate a parser that yields the list of all generated and recognized characters, than a parser yielding a single characters as we used above.

```
:: P = Fail | Yield Sym | Symbol Sym | Or P P | XOr P P | AND P P | Star P

instance apply P (CParser Char [Char] [Char])
```

where

```
    apply Fail              = fail
    apply (Yield (Char c))  = yield [c]
    apply (Symbol (Char c)) = symbol c <@ (λc=[c])
    apply (Or p q)          = apply p <|> apply q
    apply (XOr p q)         = apply p <!> apply q
    apply (AND  p q)        = apply p <++> apply q
    apply (Star p)          = (<*> (apply p)) <@ flatten
```

Generated parsers will not cause problems if they are *finite*. A parser is finite if it does not contain the parser combinators `<*>`:

```
finite :: P → Bool
finite (Or   p q) = finite p && finite q
finite (XOr  p q) = finite p && finite q
finite (AND  p q) = finite p && finite q
finite (Star p)   = False
finite other      = True
```

Parsers that need to *consume* input in order to produce a result are also safe.

```
consuming :: P → Bool
consuming Fail       = False
consuming (Yield p)  = False
consuming (Symbol c) = True
consuming (Or   p q) = consuming p && consuming q
consuming (XOr  p q) = consuming p && consuming q
consuming (AND  p q) = consuming p && consuming q
consuming (Star p)   = consuming p
```

These predicates allow us to define a class of parsers that will not produce an infinite results without consuming input as:

```
notInfiniteNonConsuming :: P → Bool
notInfiniteNonConsuming (Star p) = consuming p
notInfiniteNonConsuming p = consuming p || finite p
```

Experiments show that a little less than 8% of the generated parsers will be rejected by this predicate. Using this predicate the property for the parser combinator `<!>` can be reformulated for parsers with repetition as:

```
propXOR2 :: P P [Char] → Property
propXOR2 x y chars
 = notInfiniteNonConsuming x && notInfiniteNonConsuming y
    ⟹ case begin p chars of
         [] = begin (p <!> q) chars == begin q chars
         _  = begin (p <!> q) chars == begin p chars
where p = apply x; q = apply y
```

Despite the fact that there are more different parsers generated, this property produces a counterexample indicating an error as test case 202 (the actual number depends on the pseudo random streams used in the test data generation).

## 5 Input Generation

Apart from controlling the functions used in the properties over HOFs, it is possible to control the generation of ordinary types used in properties over HOFs. In our running example of parser combinators we used the type [Char] as input for the parsers. G∀ST will generate list of characters containing all 98 printable characters from the empty list to longer and longer lists. Although the test introduced above appear to be effective they can be improved. The parsers are generated in such a way that only the characters 'a' and 'b' will be accepted (by the definition of ggen {|Sym|} ). This implies that about 98% of the input symbols will be rejected by each instance of the parser combinator symbol. This can be improved by generating lists of characters with a limited number of characters. Without changing the instance for ggen {|Char|} in the library this can be achieved by the introduction of an additional data type and a user defined instance of ggen.

```
:: InputList = Input [Char]

ggen {|InputList|} n r = map Input l
where l = [[]: [[c:t] \\ (c,t) ← diag2 ['a'..'c'] l]]
```

The character 'c' is included to ensure that there are input symbols that need to be reject by any consuming parser. In each use we have to remove the constructor Input from the generated input. For example:

```
propXORInput :: P P InputList → Property
propXORInput x y (Input chars)
 = notInfiniteNonConsuming x && notInfiniteNonConsuming y
    ⟹ case begin p chars of
         [] = begin (p <!> q) chars == begin q chars
         _  = begin (p <!> q) chars == begin p chars
where p = apply x; q = apply y
```

This test appears indeed to be more effective. For this property G∀ST founds 319 counterexamples in the first $10^4$ tests. Using propXOR 'only' 136 counterexamples are found in this number of tests. For this property this does not matter much, one counterexample is enough to invalidate a property. In general this indicates that this algorithm yields more effective tests.

### 5.1 Generating inputs that should be accepted

In order to test whether a parser accepts the inputs it should accept, it is sufficient to use only inputs that should be accepted by the tested parser. Since we have the parsers available as data structure, it is not difficult to generate such inputs. The function PtoInput produces a list of inputs to be accepted by the parser corresponding to the given data structure of type P.

```
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 (AND  p q)         = [i++j \\ i←PtoInput p, j←PtoInput q]
PtoInput (Star p)           = take maxIter l
    where l = [[]:[ i++t \\ (i,t) ← diag2 (PtoInput p) l]]

maxIter = 10
```

The only point of interest are the repetition constructors `Star`. Here the inputs are limited to `maxIter` repetitions of the input corresponding to the argument of the repetition operator. There are two reasons for this.

First, if the parser handles inputs up to `maxIter` repetitions correctly for some descent value of `maxIter`, it is highly likely that all higher number of repetitions will be handled correctly. Test corresponding to more repetitions of the same input will not be very effective. In fact, also a much smaller value of `maxIter`, like 2 or 3, can be used.

Second, strange parsers and long inputs can produce enormous amounts of results. This is time and space consuming, but not a very effective test. As example we consider the parser `<+> (symbol 'a' <|> symbol 'a')`. Each symbol `'a'` will be recognized in two different ways. If this parser is applied to a list of $n$ characters `'a'`, the result will be a list of $2^n$ identical parse results. In order to keep testing effective we either have to remove these kind of parsers, or prevent very large inputs for such a parser. Since we do want to exclude this kind of parsers, we have chosen to limit the size of the associated inputs.

As example of the use of the generation of inputs that have to be accepted we use again the property for `<!>` combinator:

```
propXOR3 :: P P → Property
propXOR3 x y = propXOR2 x y For PtoInput (XOr x y)
```

For the first $10^4$ test cases we find now 916 counterexamples. This indicates that testing with inputs that should be accepted is even more effective as testing with pseudo random input constructed by the type `InputList`.

## 6 Direct testing of complete parsers

Above we have shown how individual parser combinators are tested effectively. This requires that at least one property is stated for each parser combinator. In this section we will show that we can also test a large set of parser combinators in one go. The idea is to construct a very simple direct parser. Given an instance of the type `P` and an input, this parser should produce all desired results.

Given a grammar and an input, it is easy to determine what the result of the parser described in section 4.1 should be:

```
results :: P [Char] → [([Char],[Char])]
results Fail              chars = []
```

```
results (Yield (Char c))  chars = [(chars,[c])]
results (Symbol (Char c)) [d:r] | c == d = [(r,[c])]
results (Symbol (Char c)) chars = []
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 (AND  p q)        chars
 = [(c3,r1++r2) \\ (c2,r1)←results p chars, (c3,r2)←results q c2]
results (Star p)          chars = repeatP p [(chars,[])]

repeatP p res
    = case [(c2,r1++r2) \\ (c1,r1) ← res, (c2,r2) ← results p c1] of
        [] = res
        r  = repeatP p r
```

This simple parser is less efficient that the parser combinator library and less flexible, but for the set of constructors defined by the type `P` it yields the list of all recognized tokens.

Using this function it is possible to state a property that has to hold for any parser that corresponds to an instance of `P`: the result of transform `p` to a parser and applying it to an input `i` should be identical to `results p i`. That is:

```
propPI :: P [Char] → Property
propPI p i = notInfiniteNonConsuming p ⟹ results p i == begin (apply p) i
```

Also here we can limit the inputs to the character lists that should be accepted by the parser:

```
propP :: P → Property
propP p = notInfiniteNonConsuming p ⟹ (propPI p For PtoInput p)
```

Also this very general property finds counterexamples corresponding to the reported problem in the original version of the library quickly. Since this property is more general it is not surprising that this property needs somewhat more tests to find a counterexample. After 279 test G∀ST reports the counterexample `(XOr (Or (Yield (Char 'a')) (Symbol (Char 'a'))) (Yield (Char 'a'))) []`. This is basically the same error as reported above. G∀ST needs less than one second to find this error.

After repairing this error we tested to library again with `PropP`. To our surprise an additional counterexample was found within 2 seconds. G∀ST reports: `Counterexample found after 791 tests: (Star (Or (Symbol (Char 'a')) (Symbol (Char 'a')))) ['a']`. The error is caused by an erroneous optimization in the parser combinator `<*>`. It appears that the parser `<*> (symbol 'a' <|> symbol 'a')` yields only one result for the input `repeat n 'a'`, instead of the desired $2^n$ identical results.

After correction of this error no new issues were found in an additional 30,000 tests. This takes 2.4 seconds. 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 incorrect libraries revealed counterexamples for each of these libraries within 2 seconds.

The final set of parser combinators can be found in the appendix.

## 7 Testing other Higher Order Functions

So far we have shown how our technique for testing higher order functions can be used for continuation based parser combinators. But our approach can be used to test any higher order function. To illustrate this, 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 [8] and is based on the Bird-Meertens theory of lists [1, 9]. 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 `propFold`. 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.

```
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 a concrete 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 (apply ex) 0 l e
```

In addition we need to generate suitable functions of type `Int Int → Int`. 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 `apply` converts instances of this data type to the desired functions:

```
instance apply Expr (Int Int → Int)
where
    apply X          = λx y.x
    apply Y          = λx y.y
    apply ConstOne   = λx y.1
    apply (SUM  a b) = λx y.apply a x y + apply b x y
    apply (DIFF a b) = λx y.apply a x y - apply 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 foldr)
```

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`. G∀ST spots this within 0.1 seconds. Although this result in itself is not new, it demonstrates the power of this approach to test higher order functions.

## 8 Conclusion

Test systems like Quickcheck and G∀ST are very suited to test properties over first order functions [3, 6]. Testing higher order functions was troublesome, since they have functions instead of data types as argument and result. The functions yielded by a higher order function are tested by supplying arguments until a data type is obtained. Until now test systems were able to generate functions as test argument in a primitive and unguided way. In this paper we have shown that the functions needed as argument can be generated by defining a data type representing the grammar for the desired functions, and a very simple function that transforms this data type to the corresponding function. This is a reinvention of ideas similar to Reynolds defunctionalisation from 1972.

By using this technique for a library of parsers combinators the test system has found a reported error as well as an until now unknown error. Since the errors occur for very unusual combinations of parser combinators it is not strange that the errors were not discovered during manual testing and ordinary use of the library. Also 25 errors injected deliberately in order investigate the power of automatic testing are found within seconds. This indicates that this way of automatic testing is very effective and efficient.

Our approach is a very general one that can also be used in any situation where higher order functions needs to be tested, or even where systematically generated functions are needed. In this paper we have show the application to simple properties over `map` (see section 4) and `fold` (see section 7), and the more advanced parser library, but it works for properties over any HOF.

### Acknowledgement

We thank Erik Zuurbier and Arjen van Weelden for indicating problems with the parser library initiating this research. Oliver Danvy pointed out the relation of our intermediate data types and defunctionalisation.

## 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.
4. Olivier Danvy and Lasse R. Nielsen. *Defunctionalization at Work.* PPDP '01 Proceedings, 2001, pp 162–174.
5. Pieter Koopman and Rinus Plasmeijer: *Efficient Combinator Parsers* In Hammond, Davie and Clack: Proc. IFL'98, LNCS 1595, pp. 120–136. 1999
6. Pieter Koopman, Artem Alimarine, Jan Tretmans and Rinus Plasmeijer: *Gast: Generic Automated Software Testing*, Peña: IFL'02, LNCS 2670, pp 84–100, 2002.

7. P. Koopman and R. Plasmeijer. Generic Generation of Elements of Types. In *Sixth Symposium on Trends in Functional Programming (TFP2005)*, Tallin, Estonia, Sep 23-24 2005.
8. Grant Malcom. *Algebraic Data Types and Program Transformations*, Thesis, 1990.
9. Lambert Meertens. *Algorithmics: Towards programming as a mathematical acitvity*, Proc. CWI Symposium 1983.
10. Rinus Plasmeijer and Marko van Eekelen: *Concurrent Clean Language Report (version 2.1.1)*, 2005. `www.cs.ru.nl/~clean`.
11. John C. Reynolds. *Definitional interpreters for higher-order programming languages*. Higher-Order and Symbolic Computation, 11(4):363-397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

## A Improved parser combinator definitions

This appendix contains the changed and tested version of the parser combinators. The types used are unchanged. The most important change is that the role of the OR-continuation and the XOR-continuation is swapped in order to get the behavior both or-combinators correctly. The basic operators `fail`, `yield` and `symbol` are basically unchanged. The definitions are slightly changed in order to reflect the change in role of the continuations `xc` and `ac`.

```
symbol :: s → CParser s s t | == s
symbol s = psymbol
where psymbol sc xc ac [x:ss] | x == s = sc s xc [] ss
      psymbol sc xc ac _               = xc ac
```

Both choice combinators also reflect the change of role of the continuations. The combinator `<|>` inserts the second parser in the continuation of `p` with alternatives that are always taken. The `<!>` operator inserts `q` in the other continuation and changes the the other or-combinator such that it checks for results.

```
(<|>) infixr 4 :: (CParser s r t) (CParser s r t) → CParser s r t
(<|>) p q = λsc xc ac ss = p sc (λac3 = q sc xc ac3 ss) ac ss

(<!>) infixr 4 :: (CParser s r t) (CParser s r t) → CParser s r t
(<!>) p q = λsc xc ac ss
 = p sc (λac2 = if (isEmpty ac2) (xc []) ac2) (q sc xc ac ss) ss
```

The and-combinator for the composition of parsers is now:

```
(<&>) infixr 6 :: (CParser s u t) (u → CParser s v t) → CParser s v t
(<&>) p q = λsc xc ac ss → p (λt xc1 ac1 → q t sc xc1 ac) xc ac ss
```

The definition of all variants of this operator (like `<&`, `&>`, and `<++>`) is not changed.

From the repeat operators `<*>` and `<+>` we removed the error by deleting the erroneous optimization in `ClistP`.

```
<*> :: (CParser s r t) → CParser s [r] t
<*> p = ClistP p []

ClistP :: (CParser s r t) [r] → CParser s [r] t
ClistP p l = (p <!&> λr → ClistP p [r:l]) <!> yield (reverse l)
```