Computer Science at Kent

StrictCheck: a Tool for Testing Whether a Function is Unnecessarily Strict

Olaf Chitil

Technical Report No. 2-11
June 2011

Abstract In a non-strict functional programming language functions that yield the same result for all total arguments can still differ for partial arguments, that is, they differ in their strictness. Here a tool for Haskell is presented that enables the programmer to easily check whether a given function is least-strict; if it is not least-strict, then the tool suggests how to make it less strict.

Copyright © 2011 University of Kent
Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

1 Introduction

Non-strict functional programming languages such as Haskell and Clean allow the definition of non-strict functions. Language implementations are based on lazy evaluation, which defers computation until results are needed. John Hughes [7] showed how lazy evaluation provides a mechanism for composing a program from small, flexible, reusable parts. An intermediate data structure can be the glue between components. Such an intermediate data structure does not incur significant space costs, because only small bits of the data structure are produced on demand immediately before they are consumed and then their space can be reclaimed by the garbage collector. Additionally this programming style naturally implements online algorithms, which quickly produce part of an output before completely having processed all inputs.

In practice, however, a program often requires far more space with lazy evaluation than with standard eager evaluation, a phenomenon known as "space leak". I suspect that space leaks often occur because functions are not non-strict enough. When strict and non-strict functions are mixed, intermediate data structures have to be constructed completely and then they use significant amounts of space. Similarly a single strict function in a sequential composition of functions can destroy the desired online behaviour of a program.

So the programmer has to identify functions that are too strict, that consume large parts of their arguments to produce small parts of their results. Such a function should be redefined less strictly. Here I present a tool, StrictCheck, that semi-automatically identifies such problematic functions amongst a large number and that suggest how the function could be less strict. StrictCheck shall support programming for non-strictness and thus modularity.

The paper is structured as follows. In the next section we specify what it means for a function to be too strict and introduce the notion of least-strictness. Section 3 describes how we can test for least-strictness and outlines the implementation of StrictCheck. Subsequently Section 4 demonstrates how StrictCheck can be used at the hand of several examples. Practical experience uncovered a number of shortcomings of StrictCheck which are discussed in Section 5. Section 6 gives a larger case study and Section 7 concludes.

This paper is an extended version of a draft paper presented at IFL 2006 [2]. StrictCheck can be downloaded from `http://www.cs.kent.ac.uk/~oc`.

2 Least-Strictness

Consider the following innocuous Haskell definition of a function which takes a list of tuples and transforms it into the list of first components and the list of second components:

```
unzip2 :: [(a,b)] -> ([a],[b])
unzip2 = foldr (\(a,b) (as,bs) -> (a:as,b:bs)) ([],[])
```

Is `unzip2` equal to Haskell's predefined function `unzip`? No, it is not:

```
Prelude> unzip [(True,False),error "ups"]
([True*** Exception: ups
Prelude> unzip2 [(True,False),error "ups"]
*** Exception: ups
```

Haskell's predefined function is *less* strict than `unzip2`. For example

`unzip [(True,False),⊥] = (True:⊥,False:⊥)` but
`unzip2 [(True,False),⊥] = ⊥`

For all *total* arguments, that is, values that do not contain $\bot$, both functions yield the same results.

I believe that programmers mostly consider only total arguments when defining their functions. The specification only makes requirements on the function result for total arguments. However, for nearly every function there exist many *variants* that yield the same results for the total arguments but differ for *partial* arguments. These variants cannot produce arbitrary results, because functions defined in a functional programming language are always monotone and continuous. For a partial argument the function has to return a value that is an approximation of the function result of any total completion of the partial argument. Hence for any function f:

$$fv \sqsubseteq \bigsqcup\{fv' | v \sqsubseteq v', v' \text{ is total}\}$$

A function f is *least-strict* iff for a partial input the function returns the greatest lower bound of all outputs produced by total completions of the partial input:

$$fv = \bigsqcup\{fv' | v \sqsubseteq v', v' \text{ is total}\}$$

The function `unzip2` is clearly not least-strict.

3 Implementation

How do we test whether a function is not least-strict? Well, function f is not least-strict if there exists a partial argument v such that

$$fv \sqsubset \bigsqcup\{fv' | v \sqsubseteq v', v' \text{ is total}\}$$

Unfortunately there usually is an infinite number of total values v' with $v \sqsubseteq v'$. Hence we cannot compute $\bigsqcup\{fv' | v \sqsubseteq v', v' \text{ is total}\}$. Therefore StrictCheck, the tool for testing whether a given function is least-strict, only considers a small number of v''s and takes the greatest lower bound of the function outputs.[1] So the test data generated by

[1] The number of v''s is still subject to experimentation, but only a small number such as 3 or 4 seems to be sufficient in practice.

StrictCheck consists of a number of partial inputs v plus a number of total completions v' for each v. Without loss of generality only partial values v that contain exactly one $\perp$ are generated. For example, if the function input is of type `[(Int,Int)]` the following test data is used:

v	v'
$\perp$	`[]`, `[(0,0)]`, `[(1,1)]`, `[(0,0),(0,0)]`, ...
`[`$\perp$`]`	`[(0,0)]`, `[(1,1)]`, `[(0,0),(0,0)]`, `[(0,0),(1,1)]`, ...
`(0,0):`$\perp$	`[(0,0)]`, `[(0,0),(0,0)]`, `[(0,0),(1,1)]`, ...
$\vdots$	$\vdots$

For each test data set the inequation at the start of this section is checked.

StrictCheck may still give wrong answers. It may report a false positive, because it applies a function only to a finite number of arguments v. If the test succeeds for all these arguments, StrictCheck claims that the function is least-strict, even though an argument v may exist for which the least-strictness property is violated.

StrictCheck may also report a false negative, because for every argument v it can only compare the function result with the greatest lower bound for a finite number of total completions v'. If the test fails for these, StrictCheck says that the function is not least-strict, although there may be a total completion v' that makes the test succeed. For example, the function `short` defined as

```
short :: [Bool] -> Bool
short [] = True
short [_] = True
short [_,_] = True
short [_,_,_] = True
short _ = False
```

is least-strict, but if too few v' are checked, StrictCheck will suggest

```
short _|_ = True
```

which is clearly wrong.

StrictCheck is just a small Haskell library, like the testing tools QuickCheck [5] and SmallCheck [14] and Gast [9] for the programming language Clean. Test data generation is similar to these testing tools. Like SmallCheck and Gast and unlike QuickCheck, test data is not generated randomly but by systematically enumerating values by size, because if a function is not least-strict, we expect this to be exposed already by small arguments. The scrap-your-boilerplate generics extensions [10] of the Glasgow Haskell compiler enables a single definition of test data generation for all types that are instances of the automatically derivable classes `Typeable` and `Data`. Test data generation proved to be unexpectedly complex and seems to require the direct use of the mind-boggling function

```
gunfold :: Data a => (forall b r. Data b => c (b -> r) -> c r)
                  -> (forall r. r -> c r) -> Constr -> c a
```

instead of one of its more user-friendly instances. The actual check for over-strictness is performed with the help of the ChasingBottoms library [6]. This library provides a (non-pure) test `isBottom`, the meet operator $\sqcup$ over partial values, and a function for converting partial values into printable strings. Because of the libraries used, StrictCheck works only with the Glasgow Haskell compiler.

4 Using StrictCheck

StrictCheck is easy to use. It provides functions `test1`, `test2`, `test3`, etc. for testing functions with the stated number of arguments. For our example function `unzip2` we find:

```
*Main> test1 10 (unzip2 :: [(Int,Int)] -> ([Int],[Int]))
Function seems not to be least strict.
Input(s): _|_
Current output: _|_
Proposed output: (_|_, _|_)
  Continue?  y
Function seems not to be least strict.
Input(s): [(0, 0)_|_
Current output: _|_
Proposed output: ([0_|_, [0_|_)
  Continue?  y
Function seems not to be least strict.
Input(s): [(0, 0), (0, 0)_|_
Current output: _|_
Proposed output: ([0, 0_|_, [0, 0_|_)
  Continue?
```

Note that the partial list `0:`$\perp$ is written as `[0_|_` and `0:0:`$\perp$ as `[0, 0_|_`.

The first argument (here 10) of the test function is a measure of the number of arguments for which the function given as second argument is tested for least-strictness. We have to annotate `unzip2` with a type, because the tool can only test monomorphic functions; for polymorphic functions it would not know how to generate test arguments. After every proposal for a more defined output StrictCheck asks the user whether it shall continue testing and making proposals.

StrictCheck clearly says that `unzip2` is not least-strict. For input $\perp$ it produces output $\perp$, but to be least-strict it should produce $(\perp, \perp)$. However, the failure to produce the tuple for input $\perp$ is of little practical concern. Indeed, if we test the predefined function `unzip`, then we find that it is not least-strict for the input $\perp$ either. However, the subsequent proposals of StrictCheck show that `unzip2` does not produce any output for a partial input list. Thus StrictCheck shows clearly that the function is spine-strict and it proposes a less strict function like `unzip`.

For a function that is least-strict the tool stops after testing a number of arguments:

```
*Main> test1 10 (True:)
Function is least-strict for 255 partial inputs.
```

Testing a large number of functions of the Haskell Prelude, StrictCheck judges only a very small number to be least-strict. However, many proposals for making a function less strict are actually impossible to implement. Consider the logical conjunction operation:

```
*Main> test2 10 (&&)
Function seems not to be least-strict.
Input(s): (_|_, False)
Current output: _|_
Proposed output: False
  Continue?  y
Completed 4 test(s).
```

StrictCheck suggests

```
  ⊥ && False = False
```

However, the function already has the properties

```
  ⊥ && ⊥ = ⊥
  False  && ⊥ = False
```

A function meeting all three equations is not sequential and hence not definable in Haskell. All functions definable in Haskell or most other programming languages are *sequential*. For formal definitions of sequentiality see Section 4 of [13]. Intuitively a function is sequential when to produce a result the function inspects (parts of) its arguments in a sequential order. Haskell's predefined `(&&)` first inspects the first argument and then, if necessary, the second argument.

StrictCheck often proposes outputs that would make the function non-sequential. Here is another example:

```
*Main> test2 5 ((++) :: String -> String -> String)
Function seems not to be least-strict.
Input(s): (_|_, "a")
Current output: _|_
Proposed output: "_|__|_
  Continue?  y
Function seems not to be least-strict.
Input(s): (_|_, "aa")
Current output: _|_
Proposed output: "_|__|__|_
  Continue?  y
Function seems not to be least-strict.
Input(s): (_|_, "aaa")
Current output: _|_
Proposed output: "_|__|__|__|_
```

StrictCheck says that the result list of `(++)` should be at least as long as the second argument, even if the first argument is $\perp$. However, `(++)` is already non-strict in its second argument, that is, it produces a partial result even if the second argument is $\perp$. A sequential list concatenation has to test one argument first and hence cannot be non-strict in both arguments.

Some proposals by StrictCheck can be implemented but are still undesirable in practise:

```
*Main> test1 5 (reverse :: [Int] -> [Int])
Function seems not to be least-strict.
Input(s): [0_|_
Current output: _|_
Proposed output: [_|__|_
  Continue?  y
Function seems not to be least-strict.
Input(s): [0, 0_|_
Current output: _|_
Proposed output: [_|_, _|__|_
  Continue?  y
Function seems not to be least-strict.
Input(s): [0, 1_|_
Current output: _|_
Proposed output: [_|_, _|__|_
```

Reversing a list should preserve the length of a list and we can define list reversal such that it produces the output list already while consuming the input list. However, the definition is more complex and it does not save space during the computation because the list elements cannot be determined until the whole input list has been traversed.

Despite frequent useless proposals StrictCheck can also provide some interesting information. For example, it finds that the standard definition of the function that produces all prefixes of a given list is unnecessarily strict. For a partial input list it could produce one more element in the output list:

```
*Main> test1 10 (inits :: String -> [String])
Function seems not to be least-strict.
Input(s): _|_
Current output: _|_
Proposed output: ""_|_
  Continue?  y
Function seems not to be least-strict.
Input(s): "a_|_
Current output: ""_|_
Proposed output: "", "a"_|_
  Continue?  y
Function seems not to be least-strict.
```

```
Input(s): "aa_|_
Current output: ["", "a"_|_
Proposed output: ["", "a", "aa"_|_
```

Inspecting the definition of the function

```
inits :: [a] -> [[a]]
inits []     =  [[]]
inits (x:xs) =  [[]] ++ map (x:) (inits xs)
```

the source of the strictness becomes clear and it can easily be rectified:

```
inits xs = [[]] ++ initsNonEmpty xs

initsNonEmpty [] = []
initsNonEmpty (x:xs) = map (x:) (inits xs)
```

A good example for the usefulness of StrictCheck is provided by a tree-traversing algorithm by Chris Okasaki [12]. The function `bfNum` takes a tree and creates a tree of the same shape, but with the values at the nodes replaced by the numbers $1, 2, \ldots$ in breadth-first order.

```
*Main> test1 5 (bfNum :: Tree Int -> Tree Int)
Function seems not to be least-strict.
Input(s): T E 0 _|_
Current output: _|_
Proposed output: T E 1 _|_
  Continue?  y
Function seems not to be least-strict.
Input(s): T E 0 (T E 0 _|_)
Current output: _|_
Proposed output: T E 1 (T E 2 _|_)
  Continue?  y
Function seems not to be least-strict.
Input(s): T E 0 (T E 1 _|_)
Current output: _|_
Proposed output: T E 1 (T E 2 _|_)
```

Surprisingly the function is fully strict. Instead we would expect to get a substantial part of the output already for a partial input. The definition of `bfNum` is short, but the algorithm is not trivial and hence the cause of the strictness not obvious. The algorithm uses two queues of trees, but all operations on queues are sufficiently non-strict. It turns out that the algorithm twice reverses the order of intermediate trees; this double reversal makes the function fully strict.

In an appendix Okasaki [12] gives an alternative breadth-first numbering algorithm `bfnum` based on ideas of Jones and Gibbons [8]. That algorithm uses a cyclic definition that works only in a non-strict language.

```
*Main> test1 5 (bfnum :: Tree Int -> Tree Int)
Function seems not to be least strict.
Input(s): T E 0 (T (T E 0 (T E 0 E)) 0 _|_)
Current output: T E 1 (T (T E 3 _|_) 2 _|_)
Proposed output: T E 1 (T (T E 3 (T E _|_ E)) 2 _|_)
  Continue? y
Function seems not to be least strict.
Input(s): T E 0 (T (T E 0 (T E 0 E)) 1 _|_)
Current output: T E 1 (T (T E 3 _|_) 2 _|_)
Proposed output: T E 1 (T (T E 3 (T E _|_ E)) 2 _|_)
```

Clearly this algorithm is far less strict, but not yet least strict. However, a tiny modification, making the pattern matching on a non-empty list lazy by adding a tilde (`~(k:ks)`), makes it least strict:

```
Main> test1 4 (bfnum' :: Tree Int -> Tree Int)
Completed 438 test(s).
Function seems to be least strict.
```

So in a non-strict language this algorithm is clearly superior.

5 Features Required for Future Versions of StrictCheck

Using StrictCheck uncovered some shortcomings of the current version that have to be addressed in the future.

The main practical problem with StrictCheck is that most proposed outputs could only be produced by a non-sequential function. The definition of least-strictness is based on monotonicity and continuity of functions, but it completely disregards whether a least-strict function is sequential. Hence in the future StrictCheck needs to test whether a proposed output could be produced by a sequential variant of the given function. Such a test seems non-trivial but feasible. StrictCheck will still make some useless proposals as in the `reverse` example, but filtering for non-sequentiality would substantially reduce useless proposals by StrictCheck.

Generic generation of test data makes StrictCheck quick and easy to use, but in many cases user-defined test data generation similar to QuickCheck and SmallCheck is desirable. User defined test data generation is needed when test data for a function has to meet certain pre-conditions. For example, when testing the parser of a compiler we need to generate mostly valid programs as test input, but generic string generation will produce hardly any valid programs. The same problem occurs when the test data are elements of an abstract data type. For example, I tried to use StrictCheck to show that the efficient pretty printing function of [1] is least-strict whereas Wadler's [15] is not. However, both implementations represent documents by an abstract data type and

generic test generation generated numerous elements of the abstract data type representation that are invalid within the abstract data type. Clearly the documents should be generated by the pretty printing combinators of the library.

StrictCheck only tests first-order functions. Functional values raise two new problems. First, functional test data has to be generated. There are many ways of generating functions and in many applications user-defined function generation may be desirable. Second, functions which may be embedded in output values have to be compared, which can only be done for a finite number of arguments. The implementation would require substantial changes, because the scrap-your-boilerplate generics extension of Haskell used for both test data generation and comparison of outputs cannot handle functional values. Sequentiality of higher-order functions is also known to be an undecidable property [11]. Hence full strictness testing of higher-order functions seems unachievable, but partial support could be useful. Note that the popular testing tool QuickCheck [5] also provides only rather limited support for higher-order functions.

6 A Case Study

Despite the shortcomings listed in the preceding section, StrictCheck can already be applied usefully to real programs.

A compiler is the canonical example of a program that is built as a composition of several phases. In a non-strict language one would expect the compiler to still parse part of the input program while already generating part of the output code. Although dependencies between different parts of the compiled program — especially the declarations and uses of variables — suggest that it is infeasible to write a compiler that works in a small constant space, one would hope its space requirements to be less than linear in the size of the program, possibly linear in the size of the symboltable.

About 10 years ago I wrote for teaching purposes a compiler that translates a simple imperative programming language into code for an abstract stack machine. It consists of 950 lines distributed over 13 modules.

With StrictCheck we test the generation of code from the abstract syntax tree. Code generation uses the symboltable, which is created by contextual analysis of the syntax tree.

```
*Main> test1 10 (\t -> generateCode t (analyse t))
Function seems not to be least-strict.
Input(s): _|_
Current output: _|_
Proposed output: [_|__|_
  Continue?  y
Function seems not to be least-strict.
Input(s): Program [] _|_
Current output: _|_
Proposed output: [_|__|_
```

```
  Continue?  y
Function seems not to be least-strict.
Input(s): Program [] (StmtSeq _|_)
Current output: _|_
Proposed output: [_|__|_
  Continue?  y
Function seems not to be least-strict.
Input(s): Program [] (StmtSeq [Assignment "" (Variable "")_|_)
Current output: _|_
Proposed output: [Lod _|_, Sto _|_, _|__|_
  Continue?  y
Function seems not to be least-strict.
Input(s): Program [] (StmtSeq [Assignment "" (Variable ""),
Assignment "" (Variable "")_|_)
Current output: _|_
Proposed output: [Lod _|_, Sto _|_, Lod _|_, Sto _|_, _|__|_
  Continue?
```

The first three proposals relate to the fact that the list of stack machine instructions is always non-empty, because it always ends with the instruction `Halt`. Like in the `reverse` example we could achieve this non-strictness, but it would be useless. The following proposals, however, point to a serious shortcoming within the compiler. For a part of the abstract syntax tree the compiler cannot already produce the corresponding machine instructions, as we would expect.

The function `generateCode` calls the function

```
translateStatementSequence :: Symboltable -> Int ->
                              StatementSequence ->
                              (MachineProgram,Length)
```

So we test this function next. This function also requires the symboltable, but even for an undefined symboltable it should already produce some output:

```
*PsaCodeGenerator> test2 10 (translateStatementSequence undefined)
Function seems not to be least-strict.
Input(s): (0, _|_)
Current output: _|_
Proposed output: (_|_, _|_)
  Continue?  y
Function seems not to be least-strict.
Input(s): (0, StmtSeq _|_)
Current output: _|_
Proposed output: (_|_, _|_)
  Continue?  y
Function seems not to be least-strict.
```

```
Input(s): (0, StmtSeq [Assignment "" (Variable "")_|_)
Current output: _|_
Proposed output: ([Lod _|_, Sto _|__|_, _|_)
  Continue?  y
Function seems not to be least-strict.
Input(s): (0, StmtSeq [Assignment "" (Variable ""), Assignment ""
(Variable "")_|_)
Current output: _|_
Proposed output: ([Lod _|_, Sto _|_, Lod _|_, Sto _|__|_, _|_)
```

We see that function `translateStatementSequence` is also far too strict. The function mainly uses a function `translateStatement`, so we test that:

```
*PsaCodeGenerator> test2 10 (translateStatement undefined)
Function seems not to be least-strict.
..
Function seems not to be least-strict.
Input(s): (0, Assignment "" (Compound (Variable "") Plus _|_))
Current output: ([Lod _|__|_, _|_)
Proposed output: ([Lod _|_, _|_, _|_, _|__|_, _|_)
  Continue?  y
Function seems not to be least-strict.
Input(s): (0, Assignment "" (Compound (Variable "") Plus
(Compound (Variable "") Plus _|_)))
Current output: ([Lod _|_, Lod _|__|_, _|_)
Proposed output: ([Lod _|_, Lod _|_, _|_, _|_, _|_, _|__|_, _|_)
```

The function `translateStatement` is not least-strict, but the counter-examples are of the same trivial kind as for the `reverse` example of the last section. Even for a partial expression a minimal number of corresponding machine instructions is known. Significantly, the function does return some machine instructions for a partial abstract syntax tree. So the function cannot be the reason for `translateStatementSequence` being spine-strict. Therefore we now scrutinise its definition:

```
translateStatementSequence symboltable start (StmtSeq stmtlist)
    = foldl combineTranslate ([], 0) stmtlist
    where
    ...
```

The definition uses `foldl`! The function `foldl` does not produce any result before it has traversed the whole list argument. StrictCheck can be used to demonstrate this spine-strictness:

```
*PsaCodeGenerator> test1 10 (foldl (&&))
...
```

```
Function seems not to be least-strict.
Input(s): (False, [False, False, False, False_|_)
Current output: _|_
Proposed output: False
```

So we have to rewrite the definition of `translateStatementSequence` such that it does not use `foldl`; for example as follows:

```
translateStatementSequence symboltable start (StmtSeq stmtlist) =
    combineTranslate 0 stmtlist
    where
    combineTranslate :: Length -> [Statement] ->
                        (MachineProgram,Length)
    combineTranslate offset [] = ([],offset)
    combineTranslate offset (stmt:stmts)
      = (stmtCode ++ code,codeLength)
      where
      (stmtCode, stmtCodeLength) =
        translateStatement symboltable (start+offset) stmt
      (code,codeLength) =
        combineTranslate (offset+stmtCodeLength) stmts
```

With the new definition the code generator is far less strict, as StrictCheck demonstrates:

```
*Main> test1 10 (\t -> generateCode t (analyse t))
...
Function seems not to be least-strict.
Input(s): Program [] (StmtSeq [Assignment "" (Variable ""),
Assignment "" (Variable "")_|_)
Current output: [Lod _|_, Sto _|_, Lod _|_, Sto _|__|_
Proposed output: [Lod _|_, Sto _|_, Lod _|_, Sto _|_, _|__|_
```

The function is not least-strict, but all the counter-examples are trivial and hence we are happy with the function.

7 Conclusions

In this paper I claimed that the modular structure and performance of non-strict functional programs can be improved by making functions less strict, more non-strict. I have defined least-strictness and presented StrictCheck, a tool that identifies functions that are not least-strict and that proposes a less strict variant.

A number of examples and in particular the case study demonstrate that StrictCheck provides useful insights into the strictness behaviour of functions, uncovers unnecessary

strictness and thus supports finding less strict alternative definitions for functions. These alternative definitions are usually not more complex, sometimes even much simpler.

Two extensions of StrictCheck are clearly desirable, filtering non-sequential functions and allowing user-defined test data generators. The second is mainly an implementation issue, but the first is the topic of future research [3, 4].

Acknowledgements

I thank Frank Huch for the `unzip2` example and Jan Christiansen for discussions about the topic.

References

[1] O. Chitil. Pretty printing with lazy dequeues. *Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):163–184, January 2005.

[2] O. Chitil. Promoting non-strict programming. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL 2006*, pages 512–516, Budapest, Hungary, September 2006. Eotvos Lorand University. Technical Report No 2006-SO1.

[3] J. Christiansen. Sloth — a tool for checking minimal-strictness. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 160–174. Springer-Verlag, 2011.

[4] J. Christiansen and D. Seidel. Minimally strict polymorphic functions. In *Proceedings of the 13th international symposium on Principles and Practice of Declarative Programming*, PPDP'11. ACM, 2011.

[5] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[6] N. A. Danielsson and P. Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In D. Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, LNCS 3125, pages 85–109. Springer-Verlag, July 2004.

[7] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[8] G. Jones and J. Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report No. 71, University of Auckland, 1993. (Also known as IFIP Working Group 2.1 working paper 705 WIN-2., 1993.

[9] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Pena, editor, *IFL 2002, Implementation of Functional Programming Languages*, LNCS 2670, pages 84–100, 2002.

[10] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[11] R. Loader. Finitary PCF is not decidable. *Theoretical Computer Science*, 266(1-2):341–364, 2001.

[12] C. Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *International Conference on Functional Programming*, pages 131–136, 2000.

[13] C.-H. L. Ong. Correspondence between operational and denotational semantics: the full abstraction problem for PCF. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol 4*, pages 269–356. Oxford University Press, 1995.

[14] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 37–48, 2008.

[15] P. Wadler. A prettier printer. In *The Fun of Programming*, chapter 11, pages 223–244. Palgrave Macmillan, 2003.