Promoting Non-Strict Programming

— Draft —

Olaf Chitil

University of Kent
United Kingdom

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 Haskell library 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.

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 [4] 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, a Haskell library, that semi-automatically identifies such problematic functions amongst a large number and that suggest how the function could be less strict. The aim is to provide a tool that supports programming for non-strictness and thus modularity.

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:

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

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

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

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

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

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

I believe that programmers mostly only consider 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 monoton 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:

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

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:

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

Besides the function `unzip'` which is clearly not least-strict, other examples spring to mind. The efficient pretty printing function of [1] is least-strict whereas Wadler's [8] is not. An efficient breadth-first numbering algorithm by Chris Okasaki [7] proves to be undesirably strict in a non-strict language, but a less strict variant can be defined.

3 The Tool

I have implemented the prototype of a tool that tests whether a given function is least strict. The tool is a small Haskell library and thus just as easy to use as the random testing tool QuickCheck [2]. For our example function `unzip'` we find:

```
*Main> test 10 (unzip' :: [(Int,Int)] -> ([Int],[Int]))
Input: _|_
Current output: _|_
Propose output: (_|_, _|_)
```

So the tool clearly says that `unzip'` is not least-strict. For input $\perp$ it produces output $\perp$, but to be least-strict it should produce $(\perp, \perp)$. The numeric 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 `unzip'` with a type, because the tool can only test monomorphic functions; for polymorphic functions it would not know how to generate test arguments.

If we test the predefined function `unzip`, then we find that it is not least-strict for the very same reason as `unzip'`. Still it is far less strict than `unzip'`. The tool does not expose the difference, because it currently only shows one input-output pair which demonstrates least strictness. Showing all input-output pairs for which the function is unnecessarily strict leads to a flood of information. In the future a sensible compromise will need to be found.

For a function that is least-strict the tool stops after testing a number of arguments (the numeric argument relates to the maximal size of tested arguments):

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

4 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

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

Unfortunately there usually is an infinite number of total values v' with $v \sqsubseteq v'$. Hence we cannot compute $\bigsqcup\{f v' | v \sqsubseteq v', v' \text{ is total}\}$. Therefore the tool only considers a small number of v' and takes the greatest lower bound of the function outputs.[1] So the test data generated by the tool consists of a number of partial inputs v plus a number of total completions v' for each v. For each

[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.

test data set the inequation above is checked. Without loss of generality only partial values that contain exactly one $\perp$ are generated. Because the tool usually considers a small number of v's and only a small number of v''s for each v, it may give wrong answers in both directions. The tool may not notice that a function is overly strict or it may incorrectly identify a function as overly strict and propose an output that is too defined. In practice I have not yet come across either case, but the user needs to be aware of these limitations.

Test data generation is similar to the testing tools QuickCheck [2] and Gast [5]. Like the later, 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. We use the scrap-your-boilerplate generics extensions [6] of the Glasgow Haskell compiler to give 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 mindboggling 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 overstrictness is performed with the help of the ChasingBottoms library [3]. 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, the tool only works with the Glasgow Haskell compiler.

5 Future Work

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 the prototype of a tool that identifies functions that are not least strict and that proposes a less strict variant.

This research has just started and the next step is to apply the tool to a number of real programs. How many functions are not least strict? Is it easy to make a function definition less strict, based on the proposals given by the tool? Do less strict functions actually use less space or time and are their definitions no more complex?

I do not claim that aiming for least strictness will remove all space leaks from non-strict programs, sharing, for example, has been completely ignored, but I hope it will be a step towards more efficient non-strict programs.

Acknowledgements

I thank Frank Huch for the `unzip'` 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. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
3. 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.
4. J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
5. 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.
6. 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).
7. C. Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *International Conference on Functional Programming*, pages 131–136, 2000.
8. P. Wadler. A prettier printer. In *The Fun of Programming*, chapter 11, pages 223–244. Palgrave Macmillan, 2003.