DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46071 Valencia (SPAIN)

DSIIC
Departamento de Sistemas
Informáticos y Computación

Informe Técnico / Technical Report

Ref. No.:	DSIC-II/13/07
	Pages: 7
Title: The Buggy Benchmarks Collection of Haskell Programs	
Author(s): Josep Silva	
Date: June 12, 2007	
Keywords: Debugging, Haskell	

V° B°
Leader of research Group Author(s)

The Buggy Benchmarks Collection of Haskell Programs

1 Introduction

Benchmarks are essential for researchers in order to compare the performance of their techniques, methods and tools. In fact, the efficiency of two different program transformations with respect to the same input program can only be suitably compared by using the same implementation of this program. All the benchmarks used in experiments must be publicly known, in order to avoid toy or tricky implementations which are very specific and "prepared" for a particular language or tool. This problem was solved in Haskell by the definition of the '*nofib* benchmark suite' [5] which consists of a publicly[1] available collection of Haskell programs specialized for different purposes (e.g. memory usage, I/O operations, etc.) and written by different people with different styles of programming.

This collection of benchmarks is widely used. It has been used during the last ten years for measuring the performance of different tools and prototypes of researchers from most different fields of functional programming; giving rise to continuous new updates and versions of the benchmarks. However, this suite has a particular (and expected) characteristic which make it mostly useless for the researchers which are developing debugging techniques: All the programs in the suite are correct.

When comparing the performance of two debuggers, it is essential to fix not only the program being debugged, but also the bugs that this program contains. For instance, consider two algorithmic debuggers A and B, and two versions $\mathcal{P}_1$ and $\mathcal{P}_2$ of the same program $\mathcal{P}$, each of them with a different bug. If A finds the bug in $\mathcal{P}_1$ after, say, forty two steps, and B finds the bug in $\mathcal{P}_2$ after, say, ten steps; then, none conclusion can be deduced; because the bug is different!!!

In order to compare two different debugging methods, it is essential to use exactly the same programs with exactly the same bugs. Bugs can also be tricky or toy, being very easy to solve for a specific technique; therefore, they should be also public and form part of the standard benchmarks collection being used.

In this work, we present a collection of buggy benchmarks based on the '*nofib* benchmark suite' of Haskell. Our work has been based on the principle that "*bugs appear randomly in the source code being unpredictable*", but taking always into account that this benchmarks collection can be used by different debugging techniques, and thus, the bugs in the suite must be heterogeneous, in the sense that they must produce various different effects on the output.

Because a particular bug can be more appropriate for a particular debugging technique (such as tracing, program slicing or algorithmic debugging) but less appropriate for another technique we make distinction between different kinds of bugs. In addition, bugs can appear in a function definition which is called in every execution (i.e., the result is always buggy), or in a function definition which

[1] http://darcs.haskell.org/nofib/

is only called in some particular executions (i.e., the result is only buggy for some particular inputs). Consequently, we have first classified bugs in different kinds depending on their effects over the behavior of the program (i.e., wrong result, non-termination...), and we have distributed them between the *nofib* collection of benchmarks. In this work, we describe the construction and the current state of our buggy benchmarks collection of Haskell programs.

The rest of the paper is organized as follows: In the next section we describe and classify the kinds of bugs used in the suite. Section 3 describes the collection of benchmarks, their origin, and their bugs. In Section 3.1 we show how to download, use and participate in the development of nofib-buggy. Some implementation details are presented in Section 4. Finally, Section 5 concludes.

2 Who Needs Correct Programs?

Correct benchmarks are useless for researchers and developers of debuggers. In order to test their methods and tools they need programs that contain bugs. Bugs can appear at compile time or at runtime. In this work, we focuss on bugs at runtime (exceptions and logic errors), and we distinguish between three kinds of bugs depending on their effect on the benchmark's behavior:

– <u>Bugs of type 1</u> produce a wrong result as the output of the program.

– <u>Bugs of type 2</u> produce the non-termination of the program.

– <u>Bugs of type 3</u> produce an exception (e.g. dividing by zero) at runtime.

Of course, the effect of the bug is only achieved whenever the part of the program which contains the bug is executed.

For instance, the program in Example 1 has a bug of type 1.

Example 1. The following function definition belongs to the *nofib-buggy* benchmark suite (subset *imaginary*, benchmark *rfib*):

```
nfib :: Double -> Double
--- BUG:
--- The following line contains a bug:
nfib n = if n <= 1
         then 1
         else nfib (n-1)+nfib (n-2)
--- CORRECT:
--- nfib n = if n < 1
---          then 1
---          else nfib (n-1)+nfib (n-2)
```

The comments indicate which line contains a bug and how this line should be if the bug was corrected.

Program	Description	Origin	Bug
anna	Strictness analyser	Julian Seward (Manchester)	3
compress	Text compression	Paul Sanders (BT)	2
fluid	Fluid-dynamics program	Xiaoming Zhang (Swansea)	2
gamteb	Monte Carlo photon transport	Pat Fasel (Los Alamos)	1
gg	Graphs from GRIP statistics	Iain Checkland (York)	1
hpg	Haskell program generator	Nick North (NPL)	1
infer	Hindley-Milner type inference	Philip Wadler (Glasgow)	3
lift	Fully-lazy lambda lifter	David Lester (Manchester) & Simon Peyton Jones (Glasgow)	1
maillist	Mailing-list generator	Paul Hudak (Yale)	2
mkhprog	Haskell program skeletons	Nick North (NPL)	1
parser	Partial Haskell parser	Julian Seward (Manchester)	1
pic	Particle in cell	Pat Fasel (Los Alamos)	3
prolog	"mini-Prolog" interpreter	Mark Jones (Oxford)	2
reptile	Escher tiling program	Sandra Foubister (York)	1
veritas	Theorem-prover	Gareth Howells (Kent)	1

Fig. 1. Buggy benchmarks collection: Real subset

3 The *nofib-buggy* Benchmark Suite

The first release of the *nofib-buggy* benchmarks collection contains 43 programs. In order to keep the structure of the *nofib* benchmark suite, all the benchmarks have been classified into three subsets:

Real subset It is composed of real programs whose resource consumption is not too small (runtime greater than five secs.) nor too big (they can be executed in a typical lab PC). Moreover, these programs are called 'real' because they were written with the objective of being useful and used by other people different from the programmer. In the first release, the buggy Real subset contains 15 programs (they are listed in Figure 1).

Spectral subset They are medium-size programs which do not address the requirements to be part of the Real subset. In the first release, the Spectral subset contains 14 programs (they are listed in Figure 2).

Imaginary subset It is composed of toy programs that can be used to test tools. They are usually easy to scale, so that the amount of work they do can be changed with a little effort. In the first release, the buggy Imaginary subset contains 14 programs (they are listed in Figure 3).

All the programs in the suite have been developed by different people from different universities and research areas. This fact ensures that programs are written with diverse programming styles.

Program	Description	Origin	Bug
boyer	Gabriel suite 'boyer' benchmark	Denis Howe (Imperial)	2
cichelli	Perfect hashing function	Iain Checkland (York)	1
clausify	Propositions to clausal form	Colin Runciman (York)	2
fish	Draws Escher's fish	Satnam Singh (Glasgow)	1
knights	Knight's tour	Jon Hill (QMW)	2
life	Game of life	John Launchbury (Glasgow)	2
mandel	Mandelbrot sets	Jon Hill (QMW)	2
minimax	Tic-tac-toe (0s and Xs)	Iain Checkland (York)	1
multiplier	Binary-multiplier simulator	John O'Donell (Glasgow) &	2
pretty	Pretty-printer	John Hughes (Chalmers)	2
primetest	Primality testing	David Lester (Manchester)	3
rewrite	Rewriting system	Mike Spivey (Oxford)	2
scc	Strongly-connected components	John Launchbury (Glasgow)	2
sorting	Sorting algorithms	Will Partain (Glasgow)	3

Fig. 2. Buggy benchmarks collection: Spectral subset

Program	Description	Origin	Bug
bernouilli	Bernouilli numbers	Haskell cafe	1
digits-of-e1	e number	Dale Thurston	1
digits-of-e2	e number	John Hughes (Chalmers)	3
exp3_ 8	Power computation	Lennart Augustsson (Chalmers)	3
gen_ regexps	Regular expressions	Wentworth	1
integrate	Integration	Anonimous	1
paraffins	Paraffins generator	Steve Heller	3
queens	n-queens problem	LML dist	3
rfib	Fibonacci numbers	Anonimous	1
tak	Recursive function	Anonimous	1
wheel-sieve1	Spirals of primes	Colin Runciman (York)	1
wheel-sieve2	Spirals of primes	Colin Runciman (York)	1
x2n1	Equation calculus	Lennart Augustsson (Chalmers)	1

Fig. 3. Buggy benchmarks collection: Imaginary subset

3.1 How to Use the Benchmark Suite

Developing a benchmark suite is a hard work. We appreciate collaboration to augment the suite with new buggy programs, or new bugs that can be added to existing programs and which are proved interesting for debugging tools.

However, new buggy programs should meet some requirements[2]:

1. Bugs must be perfectly classified: The kind of bug introduced in the program must be specified including a header at the beginning of the program.

2. Bugs must be useful: Real bugs—those introduced involuntarily—are useful. Artificial bugs—those introduced on purpose—that produce interesting effects or that are particularly difficult to find in general are useful. Artificial

[2] Further instructions can be found at: http://einstein.dsic.upv.es/buggy-nofib

bugs designed to be corrected by a particular tool are not.

3. Bugs must be perfectly identified: A warning specifying where the bug is, and a comment with the corrected statement must be placed in the benchmark (see Example 1).

4. Bugs should be also hidden: It is also desirable another version of the buggy program without any indication about where the bug is. This may help people debugging the program to face a real challenge. The corrected version of the program must be provided in a separate file specifying where the bug is.

The buggy nofib suite is public. Nevertheless, people who want to publish research results derived from the use of *nofib-buggy*'s programs must follow some rules:

1. Be honest with your results: If you evaluate your debugger with a set of benchmarks you must include all these benchmarks in your results, and not only those which reported a high performance.

2. Cite the suite: The benchmarks used in your experiments must be perfectly specified including their version so that the bug used is identified. Fixing the benchmark and the bug allows other researchers to compare your results with their results.

3. Share your results and experiments: Together with your results you must provide all the necessary information so that other researchers can repeat your experiments.

4 Implementation

The buggy nofib suite is available as a DARCS repository (for information about DARCS be referred to http://www.abridgegame.org/darcs/).

To download it, the command

```
darcs get --partial http://einstein.dsic.upv.es/darcs/nofib-buggy
```

can be used. Note that the `--partial` option is necessary.

We are currently developing a web portal to get access to the repository. Figure 4 shows a list of benchmarks shown by the portal after a user introduced a query. This web portal uses a database with control of versions and provides search capabilities. The beta-version of the portal, together with some other materials of the suite are publicly available at:

`http://einstein.dsic.upv.es/nofib-buggy`

Fig. 4. Nofib-buggy web portal

5 Conclusions

Bugs are not always bad. Designers and implementers of debugging tools need buggy programs to test and measure the performance of their methods and tools. In this work we presented the *nofib-buggy* suite, a buggy benchmarks collection of Haskell programs which extends the so-called *nofib* benchmark suite. The suite classifies programs along two dimensions: the size of the program and the kind of bug it incorpores. On the one hand, according to *nofib*'s classification, benchmarks belong to the *imaginary*, *spectral* or *real* subset depending on their size and purpose. On the other hand, bugs a re of three kinds depending on their effect: bugs that produce a wrong result, bugs that produce non-termination, and bugs that produce an exception at runtime. The whole suite is publicly available as a DARCS repository and also accesible through a web portal.

6 Acknowledgements

I greatly thank Alejandro Fernández for his valuable help with the implementation of the *nofib-buggy* benchmark suite. I also thank José Iborra for helping me with the publication of the suite.

References

1. M. Berry, D. Chen, and P. Koss. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–85, Fall 1989.
2. Gerard J. Holzmann. The Logic of Bugs. In *SIGSOFT FSE*, pages 81–87, 2002.
3. David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
4. Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: A Benchmark for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
5. Will Partain. The nofib benchmark suite of haskell programs. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 195–202. Springer, 1992.
6. Rafael H. Saavedra and Alan J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, 1996.
7. E. Taillard. Benchmarks for Basic Scheduling Problems. *European Journal of Operations Research*, 64:278–285, 1993.