

Benchmarking implementations of lazy functional languages II – Two years later

Pieter H. Hartel

Department of Computer Systems, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
pieter@fwi.uva.nl

Abstract

Six implementations of different lazy functional languages are compared using a common benchmark of a dozen medium-sized programs. The experiments that were carried out two years ago have been repeated to chart progress in the development of these compilers.

The results have been extended to include all three major Haskell compilers. Over the last two years, the Glasgow Haskell compiler has been improved considerably. The other compilers have also been improved, but to a lesser extent. The Yale Haskell compiler is slower than the Glasgow and Chalmers Haskell compilers. The compilation speed of the Clean compiler is still unrivalled.

Another extension is a comparison of results on different architectures so as to look at architectural influences on the benchmarking procedure. A high-end architecture should be avoided for benchmarking activities, as its behaviour is uneven. It is better to use a midrange machine if possible.

1 Introduction

In the previous benchmark paper [10], which will be referred to as the benchmark-I paper, five compilers for lazy functional languages were benchmarked. It is interesting to see what progress has been made in the development of these compilers during the past two years. The benchmark procedure has been extended to take into account the third major Haskell compiler, from Yale.

The benchmark has been executed on different architectures to look at the influence of architectural parameters on the reliability of the results. Previously, certain anomalies could only be explained by referring to the complicated architecture on which the experiments were performed. The inclusion of results for different architectures make it possible to firm up some conclusions about the results.

To chart purely the progress made in the various compiler developments the benchmarking procedure has not been altered, only extended. The benchmark-I results are therefore directly comparable to the benchmark-II results.

The next section summarises the main points from the benchmark-I paper. Section 3 describes an extension to the benchmarking procedure. Section 4 presents the new results

obtained. Section 5 compares the results obtained using different architectures. The last section gives the conclusions.

2 A summary of the method

In the benchmark-I paper, four languages and five implementations were used. These were Clean (version 0.8) [19] and its compiler from Nijmegen [15]; a subset of Miranda¹ [18] and our own FAST/FCG compiler [9, 13]; Lazy ML [2] and its compiler from Chalmers [3] and finally Haskell [12] with the compilers from Chalmers [1] and Glasgow [17]. In this paper the third major Haskell compiler from Yale [7] is also included. A summary of the offerings of the four languages is presented in the benchmark-I paper.

A new version of Clean (version 1.0) is currently under development. The results presented here apply to version 0.8 of Clean. It is hoped that the new Clean results may be included in a future version of the benchmark.

Benchmarking different languages requires a common benchmark program to be written in different syntaxes and using different primitive functions. This is possible, as we have demonstrated with the single program used in the Pseudoknot benchmark [11]. The effort involved in translating by hand a larger set of programs would be considerable. More importantly, it is difficult to do so fairly. It is easy to make mistakes and to introduce bias towards one of the languages. In particular it is difficult to avoid bias towards the language in which the benchmarks were originally written. The solution adopted here is to write all the benchmarks in one language, whilst taking into account two important considerations to reconcile the languages.

Firstly, we restrict the use of special syntax to those forms that can be translated mechanically into a functionally equivalent, and equally efficient syntax, in all the languages with which we are concerned. The main problem here is that the Haskell compilers need to resolve operator overloading, whereas the other compilers do not. Compile and run time resolution of operator overloading has consistently been avoided by mechanically annotating each operator with the exact type of its operands. A minor problem is that some compilers (Clean and FAST) support comparison operators only on numeric data. In the benchmarks, such operators are therefore only applied to numeric data, with explicit annotation of the exact types of the operands.

Secondly, we use a set of essential primitive functions that must be present in any language for it to be of interest

¹Miranda is a trademark of Research Software Ltd.

for general purpose programming purposes. The first problem here is that some systems offer single precision floating point numbers (FAST and Yale Haskell) and others offer double precision floats. This difference could not be ironed out, so the results for essentially floating point programs will have to be interpreted with care. The second problem is that all compilers except Clean offer arrays. An array implementation based on binary trees has been provided for Clean. The third problem is that complex numbers use lazy data constructors in Haskell, but strict constructors in the other systems. This affects one of the 14 benchmark programs only (the `fft` program).

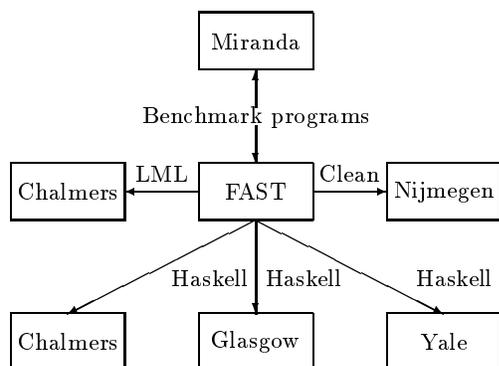


Figure 1: The organisation of the benchmarking procedure, showing how the benchmarks are translated into LML, Haskell and Clean.

The benchmark programs are written in Miranda and translated to LML, Haskell or Clean by the FAST compiler, which is also capable of autonomous compilation. The generated LML, Haskell and Clean code is further processed by the appropriate compiler. This procedure is shown in Figure 1.

The benchmark set contains small and medium-sized programs, each of which is run on a realistic input data set. Refer to the benchmark-I paper for a summary description of the individual programs.

The benchmark programs were all compiled while optimising for execution speed. The compile time and run time options used are shown in Table 1 together with the characteristics of the two platforms used.

Most compilers generate stand-alone executables, the exception being the Yale Haskell compiler. This system is embedded in a Lisp system (CMU Common Lisp). As part of the compilation process a compiled Lisp image is generated. This image can be executed under control of the CMU common Lisp system and the entire process can be timed. From the total time taken, the time necessary just to start the Common Lisp system (9 seconds on the SUN 4/690) are subtracted. This brings the Yale Haskell timing as close as possible to the timing of the stand-alone binaries generated by the remaining compilers.

Execution time measurements of the stand-alone executables (or the Lisp image in the case of Yale Haskell) were taken using the `UNIX /bin/time` command, taking the sum of user and system time as the total execution time. Each executable has been run a large number of times, on a quiet

Systems used:

SUN 4/690: a dual processor machine with 64M memory and 64K cache (1 way associative) under SunOS 4.1.2.

SPARC 10/41, a single processor machine with 128M of memory, 20K primary instruction cache (4 way associative), 16K primary data cache (5 way associative) and 1M secondary cache under Solaris 2.3

In the (Bourne) shell commands below, the variable `$P` represents the name of the program, `$I` is the input parameter (Only used at command level by the FAST/FCG system). The heap sizes shown all correspond to a maximum of 48M space.

Clean version 0.8.2 → 0.8.4:

```
fast -clean $P.i
clm $P -o $P.clean.out
$P.clean.out -b -h 48m -nt -s 8m
```

FAST/FCG version 31 → 34:

```
fast -fcg $P.i
$P.fcg.out -v 1 -h 12000000 -s 2000000 $I
```

LML version 0.999.4 → 0.999.7:

```
fast -lml $P.i
lmlc -H20000000 -O $P.lml.m fast2lml.o -o $P.lml.out
$P.lml.out -H48000000 -A1000000 -V1000000
```

Chalmers Haskell version 0.999.4 → 0.999.7:

```
fast -hbc $P.i
hbc -H40000000 -O -ihbclib: $P.hbc.hs -o $P.hbc.out
$P.hbc.out -H48000000 -A1000000 -V1000000
```

Glasgow Haskell version 0.11 → 0.22:

```
fast -ghc $P.i
ghc -Rmax-heapsize 60M -Rmax-stksize 2M -O \
-ighclib -Lghclib -lfast $P.ghc.hs \
-syslib hbc -o $P.ghc.out
$P.ghc.out +RTS -H48M -K8M
```

Yale Haskell version — → 2.3b (with CMUCL 17e):

```
fast -yale $P.i
cmu-haskell
:compile $P
cmu-haskell
:(setf ext:*bytes-consed-between-gcs* 24000000)
:load $P
:(ext:gc)
:run $P
```

Table 1: The computer system used and the compiler and run time options. The versions used previously and the current versions are shown as “benchmark-I” → “benchmark-II”.

system, taking the best execution time as the ultimate performance measure, because it minimises the error in the time measurement. The real times were found to be at most 1 second higher than the execution time in over 83% of all measurements. This implies that the reported times are a reliable indication of program performance, in the sense that the machines used were not otherwise loaded.

The measured execution times include garbage collection time. Each stand-alone executable has been run 10 times with each of six different heap sizes: 8, 16, 24, 32, 40 and 48M. The Yale Haskell system has been run likewise, while setting the `bytes-consed-between-gcs` parameter to 4, 8, 12, 16, 20 and 24M. The setting of this parameter causes the garbage collector to be run after the specified amount of space has been allocated. As the CMU Common Lisp system uses a two-space garbage collector, this setting should correspond closely to the maximum heap sizes of 8 ... 48M for the other systems.

In all cases, the selected parameters allow a two-space copying garbage collector two semi-spaces of 4, 8, ... 24M each. One of the heap sizes will give the best execution time. A small heap means slow execution because of a large number of garbage collections. In this case the user time will be high. Having a large heap slows execution down because of excessive paging activity [8]. This will result in a high system time. As the sum of the system and user time is taken as the total execution time, there will be an optimum in the execution time. The benchmarking procedure is designed to find this optimum.

All implementations were benchmarked using the default garbage collector. For the Glasgow Haskell compiler, this is a generational garbage collector; the other systems use a two-space copying garbage collector.

3 Optimising prelude functions

A bone of contention with the benchmark-I method has been the use of the prelude functions such as `map`, `filter`, `foldr` etc. These functions come with the benchmark programs, but are also available in the standard prelude provided by most of the compilers. These often-used prelude functions can be optimised. In particular, the Yale Haskell compiler uses special versions of the prelude functions that are amenable to the `foldr/build` deforestation optimisation [5]. None of the other compilers provide this optimisation (yet). To gauge the effectiveness of this (and possibly other optimisations) the Haskell programs have been compiled and run in two versions: using the built-in Haskell prelude functions and using the prelude functions supplied with the original Miranda versions of the code. This has no significant influence on the execution time measured for the Glasgow and Chalmers Haskell compilers. As one might expect, the Yale Haskell compiler produces faster code when using the Yale Haskell prelude functions. However, the improvements are slight with three exceptions: `wang` runs 20% faster and `listcopy` and `listcompr` run approximately 40% faster using the Yale Haskell prelude. These results tally with the findings reported by Gill and Peyton Jones using the Glasgow Haskell system. They report [6] that none of the `nofib` benchmark [16] programs show a significant change, when run with or without a form of deforestation similar to that used in the Yale Haskell system [14]. It is not sure whether the performance improvements due to deforestation are sustained when the overall performance of the Yale Haskell sys-

tem is brought in line with its competitors. Compiler optimisations can not be studied properly in isolation, as the many points of interaction may cause the performance of the system as a whole to deteriorate as well as improve [4].

In the results that follow, the Haskell programs all use the built-in Haskell prelude functions. The other compilers use the translated Miranda versions. This introduces a small incompatibility between the benchmark-I and the benchmark-II results. The reason for making this particular choice is that future developments might well lead to more successful optimisations if the Haskell preludes are used. This would make it necessary to switch over to Haskell preludes at a future date.

4 New results

Table 2 shows the new compile time and run time performance measurements. The compilation speed is reported in lines per minute of real time, where the number of lines in the original Miranda program determines the size of the program. For each compiler the minimum and maximum compilation speed is reported, as found over the whole range of benchmarks. The compilation speed of the various implementations has barely changed over the past two years. The compilation speed of the Clean system is still unrivalled.

The execution time for each compiled program is measured in seconds. For each stand-alone executable the best time out of $10 \times 6 = 60$ runs is reported. The heap size required to achieve this optimum is shown next to the execution time. Fixing the heap size to the same value for all experiments shows somewhat larger execution times, but the relative ranking of the compilers does not change.

Each row in Table 2 bears at least one star. The stars mark the best result for that particular row. This shows that it depends to some extent on the application which compiler generates the fastest code. The stars awarded for benchmark-I are shown here for ease of reference as dots. Clean, FAST/FCG and LML produced the fastest code for benchmark-I. For benchmark-II the fastest code is generated by the Glasgow Haskell compiler. This compiler has been improved considerably [17]. The major improvements are the provision of a strictness analyser (which was previously lacking) and a completely rewritten simplifier. Combined with a careful tuning of the system, a considerable overall improvement is the result.

The Clean compiler and the two Chalmers compilers have been improved over the whole range of programs, but not significantly so. No improvements have been made to the FAST/FCG compiler, so a stand-still of two years is sufficient to be demoted from the top of the league.

We have it on good authority (but alas undocumented) that on larger programs [16] than the ones used here, the Chalmers and Glasgow Haskell compilers are closer than would appear from the measurements presented here.

At first there appears to be various remarkable differences in the amount of space used to achieve fast execution. Compare the SPARC 10/41 performance for `listcopy` as delivered by the Chalmers and Glasgow Haskell systems. The former uses only 8M; the latter requires 48M. Another program, `fft`, shows quite the opposite behaviour. Figure 2 shows the best execution times of `listcopy`, with each of the six heap sizes used, for both Chalmers and Glasgow Haskell. There is little variation in the execution times and it just so happens that the slopes of the two curves are in oppo-

language compiler	input	Clean	FAST/FCG	LML	Haskell						
					Chalmers		Glasgow		Yale		
Compilation speed in lines per minute real time, SUN 4/690											
minimum		★ 219	25	94	22	48	62				
maximum		★ 1384	285	369	316	153	270				
Heap space in Mbyte and execution time in seconds, SUN 4/690											
event	400000	16M	24	32M · 24	16M	37	16M	34	32M ★ 19	48M	109
wang	250	48M	19	16M · 22	24M · 18	24M · ★ 15	40M ★ 15	48M	48M	48M	45
fft	5	48M	9	40M · ★ 6	24M · ★ 6	48M	8	8M	11	48M	44
genfft	7	8M · ★ 6	8M	8	8M	7	8M	7	24M	8	48M (
listcompr	1000	8M	20	8M · ★ 6	8M	8	8M	8	40M	34	48M 73
listcopy	1000	8M ★ 21	8M · 26	8M	28	8M	29	48M	41	48M	84
wave4	4000	24M	104	48M · ★ 30	32M	86	48M	127	16M	119	48M 505
sched	11	8M	8	8M · 7	8M	15	8M	14	8M ★ 4	48M	38
ida	6	8M · 26	8M · 28	32M	45	8M	33	16M ★ 19	48M	97	
typecheck	600	8M	51	8M · 54	8M	66	8M	69	8M ★ 50	48M	1041
solid	13	16M · 41	32M · 54	8M	39	8M	38	48M ★ 32	40M	171	
complab	10	16M	41	32M · 38	16M	50	16M	48	40M ★ 36	48M	(
transform	500	8M	79	16M · 78	24M	105	8M	96	16M ★ 65	48M	257
parstof	40	8M	31	8M · 36	8M	66	32M	59	8M ★ 25	48M	3413

(a) The new compilation and execution results using the SUN 4/690

language compiler	input	Clean	FAST/FCG	LML	Haskell							
					Chalmers		Glasgow		Yale			
Heap space in Mbyte and execution time in seconds, SPARC 10/41												
event	400000	24M	14	32M	13	24M	19	24M	19	24M ★ 10	-	-
wang	250	48M	12	16M	12	24M	9	24M	9	32M ★ 7	-	-
fft	5	48M	6	24M ★ 3	32M ★ 3	40M	4	8M	6	-	-	
genfft	7	8M	4	8M	4	8M ★ 3	8M	4	24M	4	-	-
listcompr	1000	8M	10	8M ★ 3	8M	5	8M	4	48M	20	-	-
listcopy	1000	8M ★ 11	8M	13	8M	14	8M	14	48M	22	-	-
wave4	4000	24M	50	48M ★ 15	32M	47	40M	67	16M	61	-	-
sched	11	8M	4	8M	4	8M	9	8M	8	8M ★ 3	-	-
ida	6	8M	13	8M	15	8M	23	8M	17	16M ★ 11	-	-
typecheck	600	8M ★ 27	8M	29	8M	35	8M	36	8M	29	-	-
solid	13	16M	23	32M	32	8M	21	8M	24	48M ★ 18	-	-
complab	10	16M ★ 22	24M ★ 22	24M	22	24M	29	16M	26	40M ★ 22	-	-
transform	500	8M	41	16M	42	16M	52	8M	48	24M ★ 37	-	-
parstof	40	8M	16	8M	19	8M	35	16M	30	8M ★ 14	-	-

(b) The new execution results using the SPARC 10/41.

Table 2: The compile time and run time results of the benchmarking (★ = Best execution time for Benchmark-II; · = Best execution time for Benchmark-I; (= Stack overflow in CMU Common Lisp; - = results not available yet)

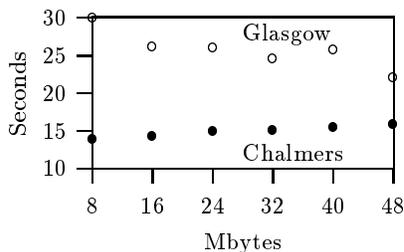


Figure 2: Best execution times (seconds) on the SPARC 10/41 of `listcopy` for Glasgow and Chalmers Haskell as a function of the heap size (Mbyte).

site directions. The amounts of heap space reported thus do not imply that one compiler requires significantly more heap space than another to achieve fast execution. The exception to this rule is the Yale Haskell compiler which runs between 1.3 and 2.5 times slower with the smallest heap size of 8M.

Some of the more specific remarks that were made in the benchmark-I paper need to be reconsidered.

It is no longer the case that the Haskell compilers from Chalmers and Glasgow are less mature than the other, older compilers. Work is in progress to improve the Yale Haskell performance. Better results will hopefully appear in a future version of this paper.

The implementation of arrays should generally be improved. This is demonstrated by the relatively slow execution of the code generated by the all but the FAST/FCG compiler on the (array intensive) `wave4` program.

All systems provide floating point support at comparable levels (programs that use floating point arithmetic are `wang`, `fft`, `wave4` and `solid`). The results do not indicate that single precision (FAST/FCG and Yale Haskell) or double precision (all other compilers) floating point support has an effect on these programs.

Both the FAST compiler and the Glasgow Haskell compiler generate C programs; the Yale Haskell compiler generates Lisp code and the remaining compilers generate assembler programs. Going via C is not a disadvantage. Going via Lisp apparently does not offer an easy route to high performance. It should be pointed out that embedding the system in Lisp enables the Yale Haskell programming environment to be more elaborate than that offered by the other compilers.

To give a rough indication of the progress that has been made in constructing lazy functional compilers over the last two years, consider the task of running all 14 the programs through the five benchmark-I compilers. The execution times for benchmark-I add up to 2969 seconds, those for benchmark II add up to 2536 seconds. This implies a progress of 17%. Most of this is due to the Glaswegian efforts.

5 Comparing different architectures

Some performance differences that were observed in the benchmark-I project could not be explained properly. To investigate this further, all stand-alone binaries have been run on a SUN 4/690 (the same machine as for benchmark-I, although over the last two years the operating system has undergone various updates) as well as on a second, rather

SUN	SunOS	memory	cache (I/D)
4/40	4.1.1	48M	64K
4/65	4.1.1	64M	64K
4/75	4.1.1	64M	64K
4/50	4.1.1	48M	64K
4/370	4.1.1	32M	64K
4/690	4.1.2	64M	64K
SS10/30	4.1.3	32M	1M+20/16K
SS10/41	5.3	128M	1M+20/16K
SS20/51	5.3	32M	1M+20/16K

Table 3: Nine different SUN systems used to benchmark `listcopy` and `listcompr`.

different machine: a SPARC 10/41 (c.f. Table 1). The SUN 4/690 that has been used is a high-end model. The SPARC 10/41 is a more modern machine, but is a midrange model. Nevertheless, the SPARC 10/41 is between 1.5 and 2 times as fast as the SUN 4/690 on the benchmark set. A highly optimised, high-end machine responds in a less even fashion to varying loads than an average, midrange machine. Comparing the timings obtained on these different machines, using the *same* binaries presents an opportunity to look at the architectural and system differences.

Consider the execution times for the programs `listcopy` and `listcompr` as measured for the Clean and Glasgow Haskell compilers. These programs are virtually the same, the difference being that `listcopy` makes an extra copy of the (rather long) output list (see the benchmark-I paper for details). Such a small amount of extra work should only make it slightly more expensive to compute the full answer. The difference between 23 and 30 seconds (Clean results, benchmark-I) can therefore only be blamed on the uneven response of the architecture to slightly different loads. The new Clean measurements on the SUN 4/690 are close, as expected (20 and 21 seconds respectively). The same problem is now manifested with the Glasgow Haskell compiler, which in the benchmark-I results showed an acceptably small difference (33 versus 36 seconds), but now 34 versus 41 seconds. Looking at the results obtained with the SPARC 10/41 shows that both Clean and Glasgow Haskell yield similar performance on the two programs: 10 versus 11 seconds for Clean and 20 versus 22 seconds for Glasgow Haskell. The two programs `listcopy` and `listcompr` were also run on seven other systems, with various different characteristics as shown in Table 3. Only the SUN 4/690 showed anomalous behaviour.

The two Chalmers compilers and the FAST/FCG compiler use an optimisation, which definitely should make the execution times of `listcompr` and `listcopy` come out rather differently (see benchmark-I for details).

This observed uneven behaviour of the SUN 4/690 does not invalidate the results that have been reported. The unevenness of the architecture may affect any of the 14×5 experiments (14×6 for benchmark-II). This observation is confirmed by the awarding of the stars to the various programs and compilers. The SUN 4/690 awards (Table 2-a) are slightly more in favour of Glasgow Haskell than the SPARC 10/41 awards, (Table 2-b) but the difference is small.

6 Conclusions

Six compilers for lazy functional languages have been benchmarked using a set of 14 medium-sized programs.

Two years of development can make a significant difference to the quality of a compiler. Implementors who want to keep up with the developments are not wise to lapse into such a long period of inactivity.

The Glasgow Haskell compiler now generates the fastest code of the six compilers. There is a small gap between this compiler and four of the others. The performance of the Yale Haskell compiler is significantly worse than that of the other two Haskell compilers, but improvements will be made shortly.

Deforestation seems to be an interesting compiler optimisation, but does not fulfill its promise at this moment. Some evidence has been found that deforestation works for Yale Haskell.

A high-end architecture is less suitable for benchmarking purposes than midrange models. The uneven response to varying loads of a high-end architecture makes it difficult to obtain predictable results.

The benchmarks are available from `ftp.fwi.uva.nl`, file `pub/functional/packages/benchmark.tar.Z`.

Acknowledgements

The help of Lennart Augustsson, John van Groningen, Sandra Loosemore, Jim Mattson, Will Partain and John Peterson with the benchmarking of the Haskell, LML and Clean compilers was greatly appreciated. Marcel Beemster, Hugh McEvoy, Jon Mountjoy, Henk Muller and Wim Vree have helped to improve the paper.

References

- [1] L. Augustsson. HBC user's manual. Programming Methodology Group Distributed with the HBC compiler, Depart. of Comp. Sci, Chalmers, S-412 96 Göteborg, Sweden, 1993.
- [2] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The computer journal*, 32(2):127-141, Apr 1989.
- [3] L. Augustsson and T. Johnsson. Lazy ML user's manual. Programming methodology group report, Dept. of Comp. Sci, Chalmers Univ. of Technology, Göteborg, Sweden, 1990.
- [4] M. Beemster. Optimizing transformations for lazy functional languages. Technical report in preparation, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1994.
- [5] A. J. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *6th Functional programming languages and computer architecture*, pages 223-232, Copenhagen, Denmark, Jun 1993. ACM, New York.
- [6] A. J. Gill and S. L. Peyton Jones. Cheap deforestation in practice: An optimiser for Haskell. In *Proc. IFIP, Vol. 1*, pages 581-586, Hamburg, Germany, Aug 1994.
- [7] The Yale Haskell Group. *The Yale Haskell Users Manual (version Y2.3b)*. Dept. of Comp. Sci, Yale Univ. (Distributed with the Yale Haskell compiler), Jul 1994.
- [8] K. Hammond, G. L. Burn, and D. B. Howe. Spiking your caches. In K. Hammond and J. T. O'Donnell, editors, *Functional programming*, pages 58-68, Ayr, Scotland, Jul 1993. Springer-Verlag, Berlin.
- [9] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. *Software—practice and experience*, 24(2):127-173, Feb 1994.
- [10] P. H. Hartel and K. G. Langendoen. Benchmarking implementations of lazy functional languages. In *6th Functional programming languages and computer architecture*, pages 341-349, Copenhagen, Denmark, Jun 1993. ACM, New York.
- [11] P. H. Hartel et al. Pseudoknot: A Float-Intensive benchmark for functional compilers. In J. R. W. Glauert, editor, *Implementation of Functional Languages*, pages 13.1-13.34. School of Information Systems, Univ. of East Anglia, Norwich, UK, Sep 1994.
- [12] P. Hudak, S. L. Peyton Jones, and P. L. Wadler (editors). Report on the programming language Haskell - a non-strict purely functional language, version 1.2. *ACM SIGPLAN notices*, 27(5):R1-R162, May 1992.
- [13] K. G. Langendoen and P. H. Hartel. FCG: a code generator for lazy functional languages. In U. Kastens and P. Pfahler, editors, *Compiler construction (CC 92)*, LNCS 641, pages 278-296, Paderborn, Germany, Oct 1992. Springer-Verlag, Berlin.
- [14] S. J. Loosemore. *Secrets of the Yale Haskell optimizer revealed*. Dept. of Comp. Sci, Yale Univ. (Distributed with the Yale Haskell compiler), Jul 1993.
- [15] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *3rd Parallel architectures and languages Europe (PARLE)*, LNCS 505/506, pages 202-220, Veldhoven, The Netherlands, Jun 1991. Springer-Verlag, Berlin.
- [16] W. Partain. The nofib benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Functional programming*, pages 195-202, Ayr, Scotland, Jul 1992. Springer-Verlag, Berlin.
- [17] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. Joint Framework for Information Technology (JFIT) Technical Conference*, pages 249-257, Keele, England, Mar 1993. DTI/SERC.
- [18] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture*, LNCS 201, pages 1-16, Nancy, France, Sep 1985. Springer-Verlag, Berlin.
- [19] M. C. J. D. van Eekelen, H. Huitema, E. G. J. M. H. Nöcker, J. E. W. Smetsers, and M. J. Plasmeijer. Concurrent Clean language manual - version 0.8. Technical report 92-18, Dept. of Comp. Sci, Univ. of Nijmegen, The Netherlands, Aug 1992.