

Benchmarking implementations of lazy functional languages

Pieter H. Hartel

Koen G. Langendoen

Department of Computer Systems, University of Amsterdam

Abstract

Five implementations of different lazy functional languages are compared using a common benchmark of a dozen medium size programs. The benchmarking procedure has been designed such that one set of programs can be translated automatically into different languages, thus allowing a fair comparison of the quality of compilers for different lazy functional languages.

Aspects studied include compile time, execution time, ease of programming determined by the availability of certain key features, and the quality of the documentation. The Clean compiler from Nijmegen and the FAST compiler from Southampton and Amsterdam generate the best code. The Nijmegen Clean compiler is much faster than all the others. The LML system from Chalmers is the most robust. The Haskell compilers from Chalmers and Glasgow provide the most comprehensive functionality.

1 Introduction

To take stock of the current state of affairs, a number of compilers for lazy functional languages have been benchmarked. We have looked at aspects of their use that are important to the programmer using the language. No attention has been given to parallel implementations, even though most of the implementations that were benchmarked also support parallel execution, when suitable hardware is available.

The overriding concern has been to give the reader a *clear* and *realistic* assessment. In particular the use of sophisticated measures has been avoided, and a presentation style is used that one might find in a consumer guide. The reader may look at the tables and decide at first glance whether the properties of interest are satisfied by a particular implementation. The choice of properties is something of a compromise between criteria that are easy to understand (which compiler generates faster code?) and sometimes obscure properties of the implementations of lazy functional languages (which language does not support local function definitions?).

The next section motivates our choice of languages and implementations. Section 3 describes the procedure that has been used to benchmark a common set of benchmarks with different languages. The experimental results are presented in Section 4. The last section gives our conclusions.

2 The choice of languages and implementations

There are several major efforts to implement a state-of-the-art lazy functional language. For the main part these efforts are addressed at implementing Haskell [12]. Haskell compilers

are under development at several places, including Glasgow University, Yale University and Chalmers University. The Chalmers and Glasgow compilers have been benchmarked. We hope to also report on the Yale compiler in a future version of the paper.

Miranda¹ [18, 19] and LML [1, 2] are slightly older and in some respects simpler languages than Haskell. This means that they are both easier to learn and easier to implement. The LML compiler has been developed at Chalmers University. A Miranda system can be purchased from Software Research Ltd. of Canterbury. Unlike the other implementations, the Miranda system does not compile programs to machine code but interprets an abstract machine code. It would not be fair to compare the runtime of interpreted code with that of compiled code. Instead of using Miranda, we use our own compiler. This is the FAST compiler (developed at Southampton University [6, 7]) together with the FCG code generator (developed at Amsterdam University [9]). The FAST compiler implements Intermediate, a language that has a syntax similar to Miranda. Intermediate provides more primitive functions than Miranda, but does not support operator overloading and several other features of Miranda.

Concurrent Clean [11, 20] is a language primarily intended to be used as target language for a compiler frontend but it has been extended towards a more comprehensive programming language. The implementation is from Nijmegen University.

These lazy functional languages have many features in common, the most important being that they are all type safe, with strong polymorphic type checking. Strong typing is such a powerful help in developing reliable programs, without incurring runtime performance penalties, that we felt restricting our attention to such languages is justified.

Table 1 provides a summary of the facilities that are provided by the languages of interest. The Miranda column is present to enable comparisons with Intermediate. Not all of the facilities listed are actually used by the benchmark programs.

The first category *basic data types* lists the basic language facilities, some of which are discussed in detail in the next section. All languages support scalar data types of various sorts. Arbitrary precision integers are not used in the benchmarks.

The category *data structures* shows that tuples, algebraic data types and lists with pattern matching on these data structures are provided by all languages. General arrays are provided by Intermediate and Haskell. LML provides a subset of the Haskell array primitives. Clean provides a builtin type for strings based on arrays of 8-bit characters but no general arrays. Strings in the benchmarks are always implemented as lists of characters.

In the category *pattern matching* a distinction is made between matching on arguments in ordinary function definitions (of the form *function pattern ... pattern = expression*), and matching on conformal definitions (of the form *pattern=expression*). Pattern matching on arguments is fully supported by all languages, but in Clean it is not possible to write a conformal definition such as `(x:xs)=list_of_numbers` to separate the first element `x` from the rest. Clean only supports conformal definitions such as `(a,b)=pair_of_numbers`, which dissects a 2-tuple into its constituents. (Note that the example programs in this paper are written in Miranda syntax; Clean, LML and Haskell syntax are all different, see also Section 3). Guards can be used in all languages to constrain the pattern matching on function arguments.

The third item in the category *pattern matching* states that in all languages except Miranda and Intermediate, tuples are considered refutable patterns. i.e. given the definition of `refute` as below, an application of `refute ⊥` yields `⊥` in such languages, but the value 1 in

¹Miranda is a trademark of Research software Ltd.

	Clean [20]	Miranda [19]	Intermediate [7]	LML [2]	Haskell [12]
Basic data types					
bool, char, int, float	+	+	+	+	+
complex	–	–	+	–	+
bitwise logical	+	–	+	+	–
arbitrary precision integers	–	+	–	+	+
Data structures					
tuples and lists	+	+	+	+	+
algebraic data types	+	+	+	+	+
general arrays	–	–	+	□	+
special character arrays	+	–	–	–	–
Pattern matching					
on arguments	+	+	+	+	+
on conformal definitions	□	+	+	+	+
refute (a,b) = 1 is lazy	–	+	+	–	–
Program structure					
where clauses	+	+	+	+	+
let expressions	–	–	+	+	+
modules	+	+	–	+	+
abstract data types	□	+	–	□	+
local functions	–	+	+	+	+
list comprehensions	–	+	+	+	+
arithmetic sequences	–	+	+	+	+
constant applicative forms	–	+	+	+	+
Typing					
polymorphic types	+	+	+	+	+
operator overloading	–	+	–	–	+
polymorphic =, <, etc.	–	+	–	+	+
Interfacing and program development					
foreign language calls	+	–	+	+	+
strictness annotations	+	–	–	+	–
use of prof and dbx (UNIX)	–	–	□	□	□
use of special profiling	–	–	–	+	+
I/O and system interfacing	+	□	–	□	+
support for parallelism	+	–	□	□	□
Miscellaneous					
quality of the manual	–	+	□	□	+
available via ftp	+	–	–	+	+

+ = best/feature provided
 □ = feature not completely provided (see text)
 – = worst/feature not provided

Table 1: Language facilities

Miranda and Intermediate.

```
> refute (a,b) = 1
```

The category *program structure* gives an indication of the syntax provided by the various languages. A “let” expression may be written where an unadorned expression is permitted. A “where” clause may appear only at the end of a function definition. For example in the contrived program below, the variable `m` is available both in the function body and in the guards. This could not have been achieved with a “let” expression.

```
> fac n = 1          , if m <= 0
>          = n * fac m , otherwise
>          where
>          m = n-1
```

All languages except Intermediate provide modules and a facility for separate compilation. Proper abstract data types are provided by Miranda and Haskell. Clean and LML use the module facility to achieve the information hiding required for the support of abstract data types. Intermediate does not provide abstract data types. Local function definitions, list comprehensions and arithmetic sequences are supported by all languages except Clean.

The final row in the category *program structure* shows that all languages except Clean support constant applicative forms (CAFs). A CAF is a parameterless function, which, when used more than once, is still evaluated only once. Clean evaluates a parameterless function each time it is called. To overcome this problem, all parameterless functions may be grouped under the “where” of the main expression, which will avoid the recomputation. This, however, incurs a certain overhead as the parameterless functions are no longer globally defined, so they will have to be passed explicitly to the functions that use these definitions. The lack of support for CAFs makes Clean less suitable as a target language for lazy functional languages that do require CAFs.

The category *typing* shows that all languages are polymorphic. Operator overloading and polymorphic comparisons are only supported by some languages. These issues are discussed further in the next section.

In the category *interfacing and program development* a brief characterisation is given of how to use lazy functional programs in the real world. Most languages have a way of interfacing to functions written in other languages.

Clean and LML allow the programmer to annotate programs for example to declare data structures to be strict in certain components, which is intended to help the compiler generate faster code. Not all data structures can be annotated, in Clean for example tuples may be annotated but not algebraic data types. The Haskell language does not define annotations, but the two Haskell compilers allow for programs to be annotated.

Some implementations provide support for the standard UNIX debugging and profiling tools `prof` and `dbx`. The FCG code generator, which has been used with the FAST compiler for the benchmarking, does not support debugging and profiling, but there are two other (slower) code generators available for FAST that do support the use of these tools (see [7]). A profiling system specifically designed for lazy functional languages [16] is available with the Haskell and LML systems.

Clean and Haskell provide the most advanced I/O facilities but all benchmark programs print a simple answer and none of them read input from a file. This does not mean that

the programs do not require input data, but merely that a specific input data set has been included into the program text. This has been done so that program and input data are guaranteed to be the same for all runs of a particular program. Only the most basic form of I/O has thus been used and we have not seriously looked at the I/O systems of the languages.

Clean is the only language specifically designed for parallel execution. Haskell and LML support annotations for parallelism and Intermediate has a number of primitives to spark parallel execution.

The last category *miscellaneous* describes the documentation and availability of the compilers. The marks given for the quality of the language manual represents our opinion, which is by no means objective. LML, Clean and Haskell implementations are available via anonymous ftp.

3 Reconciling the language features and primitives in the benchmarks

Benchmarking different languages requires a set of benchmark programs to be written in different syntaxes and using different primitive functions. This is difficult when done on an ad hoc basis because 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 the benchmarks are originally written in. The solution adopted is to write all the benchmarks in one language, whilst taking into account the following considerations:

1. Restrict the use of special syntax to those forms that can be translated mechanically into functionally equivalent, efficient syntax of all the languages of concern.
2. Use a set of essential primitive functions that must be present in any language to be of interest for general purpose programming purposes.

The next sections further elaborate these considerations.

3.1 The use of special syntax

To reconcile all the languages, two features of Haskell and Miranda had to be avoided:

- Avoid operator overloading. This feature is not present in LML, Clean and Intermediate. Instead of relying on the compiler to resolve operator overloading, all benchmark programs are written using functions and operators that explicitly indicate the required type of their operands. This information is available (through explicit type specifications) even to those compilers that are capable of deriving the information. This is done to ensure that the fastest code can be generated in all cases and thus to avoid bias. For example in the case of the example program `fac`, the Haskell compiler will be told explicitly that the operators `*`, `<=`, etc. are restricted to type `Int`, as opposed to the more general type `Integer`. Addition and comparison on type `Int` correspond to single machine instructions, whereas such operations on type `Integer` require many machine instructions.
- Avoid polymorphic comparisons (`=`, `>=`, etc.) on arbitrary data structures. Only comparisons on basic values such as integers and floating point numbers are assumed to

present as efficient primitive functions. Comparisons on data structures are explicitly programmed in the benchmark programs. Clean and Intermediate do not support polymorphic comparisons.

It is difficult to implement these two facilities without introducing runtime penalties, so it is best not to use either so as to arrive at a fair comparison between languages with and without overloading and polymorphic comparisons.

3.2 The use of essential primitives

To also reconcile the implementations of the languages it would have been ideal to use only a common sub-set of the primitive functions provided by all. Most primitives (integer and floating point arithmetic, boolean and character operations, etc.) are provided by all languages. However there are also primitives that are needed in many application areas, in particular in science and engineering, which are not provided by all implementations under scrutiny. These facilities are arrays, complex numbers, and logical operations on small bit vectors (of no more than say 30 elements). To overcome this problem the following measures were taken:

Arrays are primitive to Haskell. The Haskell style arrays are also fully implemented in Intermediate. LML has a restricted set of array primitives, whereas Clean does not currently offer arrays (other than character arrays). Without arrays it is impossible to implement some applications efficiently. An implementation of arrays has been provided in the form of library functions where it was lacking. The benchmark contains a mix of programs for different application areas, only a few use arrays.

Complex numbers are an essential ingredient of computations in science and engineering. Complex numbers are provided as standard by Haskell and Intermediate. They are programmed in both Clean and LML as *strict* pairs of two floating point numbers, where the program annotations for strictness have been used. This means that the two floating point components of a complex number are actually evaluated before a complex number is created. The complex numbers provided by Intermediate are also strict. The built in complex numbers of Haskell are lazy, which gives Haskell compilers a disadvantage on one benchmark program that uses complex numbers.

Short bit vectors and operations such as bitwise or and logical shifts are provided as primitives by all compilers. Bit vectors are used by two benchmark programs.

To implement the missing primitives at the source language level rather than at the target level is a disadvantage. However only few benchmark programs use the whole range of primitives and this will be taken into account when drawing conclusions about the performance of such benchmarks. Table 2 provides a summary of the support of essential data types and primitives that are used in the benchmark suite.

3.3 Benchmarking procedure

The benchmark programs are written using the Miranda program development system. After completing the development, the benchmark programs are compiled using the FAST compiler. On output the FAST compiler either produces an executable program or, depending on a compiler switch, a Haskell, LML, or Clean program. These programs are then further

language	Clean	Intermediate	LML	Haskell
int	32 bit	31 bit	32 bit	31 bit
float	64 bit	31 bit	64 bit	64 bit
complex	2*float!	2*float!	2*float!	2*float
arrays	added	builtin	partial	builtin
bitwise	builtin	builtin	builtin	library

added = implemented in the language itself
 partial = partially implemented in the language itself
 builtin = builtin primitive of the language
 library = provided by the language system in a library
 ! = strict complex numbers

Table 2: How the essential data types and primitives are implemented in each language

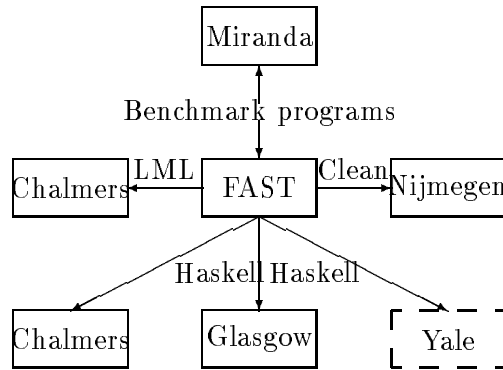


Figure 1: The organisation of the benchmarking, showing how the benchmarks are translated into LML, Haskell and Clean

processed by the appropriate compiler. This practical procedure is shown schematically in Figure 1.

The route from a Miranda program to a Haskell or LML program is straight forward because LML and Haskell support local function definitions, list comprehensions and arithmetic sequences. In this case a mere syntax change is all that is required.

The route to a Clean program is somewhat arduous as this relies on the FAST compiler to compile list comprehensions and arithmetic sequences into ordinary function calls and to perform lambda lifting. These program transformations are done as efficiently as possible (see [14]) and share the algorithm that the FAST compiler itself uses to compile to an executable program. The lack of CAF support in Clean causes programs to evaluate parameterless functions over and over again. To avoid this bias, all global parameterless functions in the Clean programs were mechanically grouped under the “where” of the main expression, as described in Section 2. This treatment has not been applied to programs in any of the other languages, as they properly support CAFs.

4 Experiments

The benchmarking procedure has been applied to a set of 13 application programs. The compilation and execution time of each program could thus be measured to provide a quantitative measure of the quality of each compiler.

4.1 Benchmark applications

Whilst gathering benchmark programs we looked for applications from different areas, to explore the support for various kinds of programming activities. The benchmark set contains small and medium size programs, each of which is run on a realistic input data set. Table 4 summarises the characteristics of the programs and provides references to their origin. The programs are sorted on the number of lines of source text. This count is exclusive of comments, blank lines, and the standard library functions provided by Miranda. The largest program comprises 1200 lines hence some programs are “real” [13], most are “imaginary”. The column *primitives* lists the data structures and primitive functions used by the program, to give an impression of its character. There are a few numerical applications (**wang**, **fft**, **wave4** and **solid**). The **event** program embodies the core of a simulation program. The programs **sched** and **ida** implement search algorithms typically found in artificial intelligence applications. An image processing application is present in the form of **complib**. The majority of the programs are parts of compilers (**listcompr**, **typecheck**, **transform** and **parstof**).

4.2 Results

The benchmark programs were all compiled with option settings that should give fast execution. The compile time and runtime options used are shown in Table 3 together with the characteristics of the system used. To achieve best performance no debugging, runtime checks or profiling code has been generated. The stand alone executables were timed on a UNIX system using `/bin/time`, 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 system, taking the best execution time as the ultimate performance measure, because it minimises the error in the time measurement. The real times were always found to be at most a few seconds higher

System used: a dual processor SUN 4/690 with 64Mbyte of real memory and 64Kbyte cache under SunOS 4.1.2.

In the shell commands below, the variable `$P` represents the name of the program, `$I` the input parameter for the Intermediate system. The heap sizes shown all correspond to 16Mbyte.

Clean version 0.8.2:

```
fast -clean $P.i
clm $P -o $P.clean.out
$P.clean.out -b -h 16m -nt -s 1m
```

Intermediate version 29:

```
fast -fcg $P.i
$P.fcg.out -v 1 -h 4000000 -s 200000 $I
```

LML version 0.998:

```
fast -lml $P.i
lmlc -H20000000 $P.lml.m fast2lml.o -o $P.lml.out
$P.lml.out -H8000000
```

Chalmers Haskell version 0.998:

```
fast -hbc $P.i
hbc -H40000000 -ihbclib: $P.hbc.hs -o $P.hbc.out
$P.hbc.out -H8000000
```

Glasgow Haskell version 0.10:

```
fast -ghc $P.i
ghc -Rmax-heapsize 60M -flet-float -O -Ighclib \
    -Lghclib -lfast $P.ghc.hs -o $P.ghc.out
$P.ghc.out +RTS -H16M -K1M
```

Table 3: The system used and the compiler and runtime options

than the cpu time, which implies that the reported times are a reliable indication of program performance.

Each executable has been run 50 times with each of the five different heap sizes that we tried: 4, 8, 12, 16 and 20 Mbytes. This allows for two semi-spaces with a two-space copying garbage collector of 2 Mbyte, 4Mbyte etc. Table 5 shows compile time and run time performance measurements. The compilation speed is reported in lines per minute real time. The execution time for each compiled program is measured in seconds. For each compiler the minimum and maximum compilation speed is reported, as found over the whole range of benchmarks. For each executable we report the best time out of $50 \times 5 = 250$ runs. 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 5 bears one asterisk, which marks the best result for that particular row. This shows that it depends to some extent on the application which compiler generates the fastest code, but in general, Clean and FAST produce the fastest code. The Clean system has the added advantage that it compiles much faster than the others.

We have been able to find an explanation for the following differences in execution times:

maturity and tuning We think that the Haskell compilers generate worse code than the other three compilers because of the complexity of the language. It takes longer to build a working compiler for a complex than a simple language, hence it takes longer before performance tuning can begin to produce results. This is pure speculation, but we do know for a fact that the Clean and LML compilers have been around for much longer than the other compilers, so we would expect them to be more finely tuned.

irrefutable tuples Tuples in Miranda and Intermediate are irrefutable. This means that a compiler for these languages cannot prove functions such as `refute strict` (see the discussion of pattern matching in Section 2). The FAST compiler therefore generates worse code than the other compilers for functions that operate on tuples. The `wang` program suffers from this effect, because the sparse matrices it manipulates are represented as lists of 4-tuples.

efficient array support The `fft` and `wave4` programs run much faster using FAST than with the other compilers. This is because of the efficient array support provided by FAST. The array implementations in Haskell and LML are probably not optimal, while the array support for Clean has been implemented in Clean itself, using (lazy) lists. In principle, the foreign language call mechanism could be used implement arrays. However to also enable the garbage collector to operate on arrays thus allocated is a major research issue and therefore well beyond the scope of a benchmarking effort.

top level append A difference in the implementations with a significant effect on the performance becomes apparent when the numbers in the two rows for `listcompr` and `listcopy` are compared. `listcopy` is identical to `listcompr`, except that it prints every output character using the function `copy` defined thus:

```
> copy []      = []
> copy (x:xs) = x:copy xs
```

The difference is due to the fact that printing output in a functional program can be expensive: strings, and printed output in particular, are manufactured by appending

program	lines	reference	primitives	short description
event	84	[10]	list, data	Event driven simulation of a set-reset fliflop. The state after 100000 transitions is calculated.
wang	100	[24, 21]	list, tuple, float	Wang's algorithm for solving system of linear equations based on a tri-diagonal 100×100 matrix.
fft	130	[8]	list, tuple, float, bit, complex, array	Two 512-point fast Fourier transforms, one using arrays and one using lists.
genfft	210	[8]	list, bit	Generation of synthetic FFT programs to calculate 4, 8, 16, 32 and 64 point transforms.
listcompr	229	[14, Ch. 7]	list, data	Translation of the list comprehensions in a program into ordinary recursive functions.
wave4	247	[21]	list, tuple, float, array	Calculation of the water heights in a square area of 8×8 grid points of the North Sea over a period of 150 time steps.
sched	250	[21]	list, data	Calculation of an optimum schedule of 11 parallel jobs with a branch and bound algorithm.
ida	256	[5]	list, data	Solution of a particular configuration of the 15-puzzle using the iterative deepening algorithm.
typecheck	360	[14, Ch. 9]	list, tuple, data	Polymorphic type checking of a set of function definitions and printing of the type signatures.
solid	605	[4]	list, tuple, data, float	Point membership classification algorithm from a solid modeling library for computational geometry.
complab	653	[17]	list, tuple, data	Image processing application that labels all four connected pixels into objects.
transform	834	[22]	list, tuple, data	Transformation of 9 programs represented as synchronous process networks into master/slave style parallel programs.
parstof	1192	[3]	list, tuple, data	Lexing and parsing based on Wadler's parsing method [23] of a 600 line program.

arrays = uses arrays

bit = uses logical operators on small bit vectors

complex = uses complex arithmetic

data = uses algebraic data types

float = uses floating point arithmetic

list = uses lists

tuple = uses tuples

Table 4: The benchmark programs with an indication of their size and purpose

language compiler	Clean		Intermediate		LML		Haskell			
							Chalmers		Glasgow	
Compilation speed in lines per minute real time										
minimum	*132		19		112		70		22	
maximum	*1277		266		337		265		143	
Heap space in Mbytes and execution time in seconds										
event	4M	7.2	4M	*5.5	8M	10.3	4M	19.9	8M	13.5
wang	16M	*2.8	4M	3.3	4M	4.1	4M	4.3	16M	3.4
fft	4M	9.2	4M	*0.9	4M	5.1	4M	4.8	4M	3.3
genfft	4M	1.9	4M	*1.7	8M	2.7	4M	2.9	20M	2.7
listcopy	4M	*4.0	4M	4.6	20M	4.5	12M	4.4	8M	6.3
listcompr	4M	5.3	8M	*1.2	4M	1.7	4M	1.7	8M	5.7
wave4	8M	7.8	4M	*1.3	4M	7.6	12M	17.2	4M	11.9
sched	8M	17.2	4M	*8.7	8M	16.8	8M	17.7	4M	10.1
ida	4M	9.6	4M	9.3	8M	16.2	8M	13.4	8M	*8.9
typecheck	4M	*9.9	4M	10.5	4M	15.4	20M	16.2	8M	12.3
solid	8M	16.2	4M	*12.2	4M	26.6	4M	21.3		?
complab	4M	*2.0	4M	2.2	4M	2.9	4M	3.7	4M	3.1
transform	4M	2.9	4M	*2.6	4M	3.2	4M	3.4	4M	2.8
parstof	4M	*3.1	8M	46.6?	4M	5.7	8M	5.2	20M	75.3?
*	=	Best execution time				?	=	Compiler problem		

lists (using the `++` operator) into one large output list. If the sole purpose of the appends is to produce the printed output of the program, then a simple optimisation is possible that may save work. This so called “top level append optimisation” works as follows: when the printer encounters an application of `append`, it prints the first argument, then prints the second argument and then returns, without constructing the concatenation of the two lists in the heap. This particular optimisation is only possible when the top level expression is of the form `"..." ++ "..."`, but not when some other function (i.e. `copy`) is posed in between the top level printer and the `++` operations, as is the case in `listcopy`. The only programs to produce large outputs are `listcopy` and `listcompr`. The others typically print a few dozen characters. When large outputs are produced, the effect is dramatic, as for example the FAST performance of `listcompr` is 1.2 seconds, which jumps to 4.6 seconds for `listcopy`, just by making it impossible to use the top level append optimisation. The top level append optimisation originates from the Chalmers compilers.

target language The FAST compiler and the Glasgow Haskell compiler generate C programs, while the remaining compilers generate assembler programs. Both C-generating systems impose special requirements on the C compiler to be used: the Glasgow Haskell compiler generates good code only when using version 2.2.1 or later of the GNU C compiler. The FCG code generator for the FAST compiler generates one large C function for each compiled program since in general C compilers do neither support explicit global register allocation nor lightweight function-call sequences; unfortunately, not all C compilers can handle really large functions.

It is interesting to note that using C as a high-level assembler does not mean generating bad code. Generating C does mean however, that considerable tuning is required to produce C programs that the C compiler properly understands [9]. From the point of view of performance the claim that generating C implies portability is not true, because switching from one C compiler to another requires a considerable amount of tuning. Using a C compiler instead of an assembler to generate an executable object file also increases compilation time significantly.

Two problems remain unresolved at the time of writing: The Glasgow Haskell compiler runs into a segmentation fault when compiling the programs `solid`, `transform` and `parstof` with the default stack size. Increasing the stack space allows `transform` and `parstof` to be compiled, but not `solid`. We hope to have a complete set of figures soon.

The second problem is, that the execution times for the `parstof` program with the FAST and Glasgow Haskell compilers is completely out of line with the other results. We are still investigating why this is the case.

5 Conclusions

Five compilers for lazy functional languages have been benchmarked using a set of 13 medium size programs, which were chosen from different application areas.

The benchmarking procedure has been designed such that one set of programs could be translated automatically into different languages. Compilers for different languages could thus be compared. The translation is fair because it does not introduce a bias towards one language in particular. This can not be achieved by manual translation of non-trivial programs into

different languages. To our knowledge this is the first systematic, comparative study of the quality of compilers for lazy functional languages.

Haskell, the most comprehensive programming language is also the most difficult to compile efficiently. The performance measurements indicate that the simpler the language, the better the runtime performance.

For programs using arrays (`fft` and `wave4`), the best performance is more than 10 times better than the worst. This is due to the fact that arrays are not built-in facilities of all the languages. The small programs in the benchmark show a varied performance ratio of up to 6 times. For the larger programs (`typecheck`, `solid`, `complab`, `transform` and `parstof`), the best performance is no more than twice the worst performance. In general the difference in runtime performance between all compilers seems so small that one may wonder whether the limit has been reached of what is possible with current compiler technology.

The Nijmegen Clean system compiles about 10 times faster than the other compilers and generates good code as well. This is also the only system that will compile on small machines (with a few Mbyte memory). Only a few problems were found with the system during the experiments, Clean is reasonably robust. The compiler gives good error messages. Clean is a simple language compared to the other languages. It lacks certain important features (e.g. CAFs and local function definitions). The Clean manual gives extensive coverage of the Clean I/O system but the language itself is not well documented. This should be rectified when the new book by Plasmeijer and van Eekelen appears [15].

The FAST compiler implements a language that is similar to Miranda. This compiler implements a comprehensive set of primitive functions and generates the best code. The compiler itself gives poor error messages. The FAST system is unsuitable for program development, which should be done using the Miranda system. Compilation of large programs is very slow. The system is still under development and not robust. There is a users's manual available for FAST. We should like to point out again that we have an interest in the FAST compiler and FCG code generator.

The LML compiler is robust and reliable. We experienced problems with all the compilers except with LML. The generated code is good. If the program being compiled contains typing errors, the compiler reports one such error at the time, which is inconvenient, in particular when considering the fact that compilation is slow. The LML documentation is good if somewhat terse.

The Haskell compiler from Chalmers and the LML compiler have many passes in common. The code generated by the Haskell compiler is sometimes better, sometimes worse than the LML code. Compilation of Haskell programs is significantly slower because of the complexity of Haskell. Error reporting in the Chalmers Haskell compiler has the same problem as the LML compiler. Haskell is well documented. The Chalmers Haskell manual is good.

The Glasgow Haskell compiler that has been used is a brand new release, which still has some problems. The runtime performance is good and can be expected to improve considerably in the future. The documentation is good.

Acknowledgements

We thank Lennart Augustsson, Marcel Beemster, Henk Muller, Eric Nöcker, Will Partain, Simon Peyton Jones and Rinus Plasmeijer for their comments on a draft version of the paper. The help of Lennart Augustsson, Sandra Loosemoore, Eric Nöcker, Will Partain and John

Peterson with the Haskell, LML and Clean compilers is gratefully acknowledged.

The FAST compiler represents joint work with Hugh Glaser and John Wild, which is supported by the Science and Engineering Research Council, UK, under grant No. GR/F 35081, FAST: Functional programming for ArrayS of Transputers.

The benchmarks are available upon request.

References

- [1] L. Augustsson and T. Johnsson. The Chalmers lazy-ML compiler. *The computer journal*, 32(2):127–141, Apr 1989.
- [2] 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.
- [3] M. Beemster. The lazy functional intermediate language Stoffel. Technical report CS-92-16, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1992.
- [4] J. R. Davy. *Using Divide and Conquer for Parallel Geometric Evaluation*. PhD thesis, School of Computer Studies, University of Leeds, England, Sep 1992.
- [5] J. Glas, R. F. H. Hofman, and W. G. Vree. Parallelization of branch-and-bound algorithms in a functional programming environment. In H. Kuchen and R. Loogen, editors, *4th Parallel implementation of functional languages*, pages 287–298, Aachen, Germany, Sep 1992. Aachener Informatik-Berichte 92-19, RWTH Aachen, Fachgruppe Informatik.
- [6] P. H. Hartel, H. W. Glaser, and J. M. Wild. On the benefits of different analyses in the compilation of functional languages. In H. W. Glaser and P. H. Hartel, editors, *Implementation of functional languages on parallel architectures*, pages 123–145, Southampton, England, Jun 1991. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England.
- [7] P. H. Hartel, H. W. Glaser, and J. M. Wild. FAST compiler user’s guide. Technical report, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, Dec 1992.
- [8] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language – a case study: the fast Fourier transform. Technical report CS-92-02, Dept. of Comp. Sys, Univ. of Amsterdam, Presented at ATABLE-92, Montréal, Canada, Jun 1992.
- [9] 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)*, LNCS 641, pages 278–296, Paderborn, Germany, Oct 1992. Springer-Verlag.
- [10] H. L. Muller. *Simulating computer architectures*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Feb 1993.
- [11] 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, *Parallel architectures and languages Europe (PARLE)*, LNCS 505/506, pages 202–220, Veldhoven, The Netherlands, Jun 1991. Springer-Verlag.

- [12] ed. P. Hudak, ed. S. L. Peyton Jones, and ed. P. L. Wadler. Report on the programming language Haskell - a non-strict purely functional language, version 1.2. *SIGPLAN notices*, 27(5):1–162, May 1992.
- [13] W. Partain. The nofib benchmark suite of Haskell programs (draft). Internal report, Dept. of Comp. Sci, Univ. of Glasgow, Scotland, Aug 1992.
- [14] S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.
- [15] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional programming and parallel graph rewriting*. Addison Wesley, Reading, Massachusetts, 1993.
- [16] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. Technical report 172, Dept. of Comp. Sci, Univ. of York, England, Apr 1992.
- [17] Q. F. Stout. Supporting divide-and-conquer algorithms for image processing. *Journal parallel and distributed computing*, 4(1):95–115, Feb 1987.
- [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.
- [19] D. A. Turner. *Miranda system manual*. Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1 1XP, England, Apr 1990.
- [20] 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.
- [21] W. G. Vree. *Design considerations for a parallel reduction machine*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1989.
- [22] W. G. Vree and P. H. Hartel. Fixed point computation for parallelism. Technical report CS-92-07, Dept. of Comp. Sys, Univ. of Amsterdam, Jul 1992.
- [23] P. L. Wadler. How to replace failure by a list of successes, a method for exception handling, backtracking, and pattern matching in lazy functional languages. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 115–128, Nancy, France, Sep 1985. Springer-Verlag.
- [24] H. H. Wang. A parallel method for tri-diagonal equations. *ACM transactions on mathematical software*, 7(2):170–183, Jun 1981.