J. Functional Programming 1 (1): 1–000, January 1993 © 1993 Cambridge University Press

Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark

Pieter H. Hartel 1, Marc Feeley 2, Martin Alt 3, Lennart Augustsson 4, Peter Baumann 5, Marcel Beemster 6, Emmanuel Chailloux 7, Christine H. Flood 8, Wolfgang Grieskamp 9, John H. G. van Groningen 10, Kevin Hammond 11, Bogumił Hausman 12, Melody Y. Ivory 13, Richard E. Jones 14, Jasper Kamperman 15, Peter Lee 16, Xavier Leroy 17, Rafael D. Lins 18, Sandra Loosemore 19, Niklas Röjemo 20, Manuel Serrano 21, Jean-Pierre Talpin 22, Jon Thackray 23, Stephen Thomas 24, Pum Walters 25, Pierre Weis 26, Peter Wentworth 27

Abstract

Over 25 implementations of different functional languages are benchmarked using the same program, a floating-point intensive application taken from molecular biology. The principal aspects studied are compile time and

1 Dept. of Computer Systems, Univ. of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, e-mail: pieter@fwi.uva.nl
2 Départ. d'informatique et r.o., Univ. de Montréal, succursale centre-ville, Montréal H3C 3J7, Canada, e-mail: feeley@iro.umontreal.ca
3 Informatik, Universität des Saarlandes, 66041 Saarbrücken 11, Germany, e-mail: alt@cs.uni-sb.de
4 Dept. of Computer Systems, Chalmers Univ. of Technology, 412 96 Göteborg, Sweden, e-mail: augustss@cs.chalmers.se
5 Dept. of Computer Science, Univ. of Zurich, Winterthurerstr. 190, 8057 Zurich, Switzerland, e-mail: baumann@ifi.unizh.ch
6 Dept. of Computer Systems, Univ. of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, e-mail: beemster@fwi.uva.nl
7 LIENS, URA 1327 du CNRS, École Normale Supérieure, 45 rue d'Ulm, 75230 PARIS Cédex 05, France, e-mail: Emmanuel.Chailloux@ens.fr
8 Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge Massachusetts 02139, USA, e-mail: chf@lcs.mit.edu
9 Berlin University of Technology, Franklinstr. 28-29, 10587 Berlin, Germany, e-mail: wg@cs.tu-berlin.de
10 Faculty of Mathematics and Computer Science, Univ. of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, e-mail: johnvg@cs.kun.nl
11 Dept. of Computing Science, Glasgow University, 17 Lilybank Gardens, Glasgow, G12 8QQ, UK, e-mail: kh@dcs.glasgow.ac.uk
12 Computer Science Lab, Ellemtel Telecom Systems Labs, Box 1505, S-125 25 Älvsjö, Sweden, e-mail: bogdan@erix.ericsson.se
13 Computer Research Group, Institute for Scientific Computer Research, Lawrence Livermore National Laboratory, P. O. Box 808 L-419, Livermore, CA 94550, e-mail: ivory1@llnl.gov
14 Dept. of Computer Science, Univ. of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK, e-mail: R.E.Jones@ukc.ac.uk
15 CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, e-mail: jasper@cwi.nl
16 Dept. of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue Pittsburgh, Pennsylvania 15213, USA, e-mail: petel@cs.cmu.edu
17 INRIA Rocquencourt, projet Cristal, B.P. 105, 78153 Le Chesnay, France. e-mail: Xavier.Leroy@inria.fr
18 Departamento de Informática, Universidade Federal de Pernambuco, Recife, PE, Brazil, e-mail: rdl@di.ufpe.br
19 Dept. of Computer Science, Yale Univ., New haven, Connecticut, e-mail: loosemore-sandra@cs.yale.edu
20 Dept. of Computer Systems, Chalmers Univ. of Technology, 412 96 Göteborg, Sweden, e-mail: rojemo@cs.chalmers.se
21 INRIA Rocquencourt, projet Icsla, B.P. 105, 78153 Le Chesnay, France. e-mail: Manuel.Serrano@inria.fr
22 European Computer-Industry Research Centre, Arabella Straße 17, D-81925 Munich, Germany. e-mail: jp@ecrc.de
23 Harlequin Ltd, Barrington Hall, Barrington, Cambridge CB2 5RG, England, e-mail: jont@harlequin.co.uk
24 Dept. of Computer Science, Univ. of Nottingham, Nottingham, NG7 2RD, UK, e-mail: spt@cs.nott.ac.uk
25 CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, e-mail: pum@cwi.nl
26 INRIA Rocquencourt, projet Cristal, B.P. 105, 78153 Le Chesnay, France. e-mail: Pierre.Weis@inria.fr
27 Dept. of Computer Science, Rhodes Univ., Grahamstown 6140, South Africa, e-mail: cspw@cs.ru.ac.za

execution time for the various implementations that were benchmarked. An important consideration is how the program can be modified and tuned to obtain maximal performance on each language implementation.

With few exceptions, the compilers take a significant amount of time to compile this program, though most compilers were faster than the then current GNU C compiler (GCC version 2.5.8). Compilers that generate C or Lisp are often slower than those that generate native code directly: the cost of compiling the intermediate form is normally a large fraction of the total compilation time.

There is no clear distinction between the runtime performance of eager and lazy implementations when appropriate annotations are used: lazy implementations have clearly come of age when it comes to implementing largely strict applications, such as the Pseudoknot program. The speed of C can be approached by some implementations, but to achieve this performance, special measures such as strictness annotations are required by non-strict implementations.

The benchmark results have to be interpreted with care. Firstly, a benchmark based on a single program cannot cover a wide spectrum of 'typical' applications. Secondly, the compilers vary in the kind and level of optimisations offered, so the effort required to obtain an optimal version of the program is similarly varied.

---

1 Introduction

At the Dagstuhl Workshop on Applications of Functional Programming in the Real World in May 1994 (Giegerich and Hughes, 1994), several interesting applications of functional languages were presented. One of these applications, the Pseudoknot problem (Feeley *et al.*, 1994) had been written in several languages, including C, Scheme (Rees and Clinger, 1991), Multilisp (Halstead Jr, 1985) and Miranda† (Turner, 1985). A number of workshop participants decided to test their compiler technology using this particular program. The first point of comparison is the speed of compilation and the speed of the compiled program. The second point is how the program can be modified and tuned to obtain maximal performance on each language implementation available.

The initial benchmarking efforts revealed important differences between the various compilers. The first impression was that compilation speed should generally be improved. After the workshop we have continued to work on improving both the compilation and execution speed of the Pseudoknot program. Some researchers not present at Dagstuhl joined the team and we present the results as a record of a small scale, but exciting collaboration with contributions from many parts of the world.

As is the case with any benchmarking work, our results should be taken with a grain of salt. We are using a realistic program that performs a useful computation, however it stresses particular language features that are probably not representative of the applications for which the language implementations were intended. Implementations invariably trade-off the performance of some programming features for others in the quest for the right blend of usability, flexibility, and performance on 'typical' applications. It is clear that a single benchmark is not a good way to measure the overall quality of an implementation. Moreover, the performance of an implementation usually (but not always) improves with new releases as the implementors repair bugs, add new features, and modify the compiler. We feel that our choice of benchmark can be justified by the fact that it is a real world application, that it had already been translated into C and several functional languages, and that we wanted to compare a wide range of languages and implementations. The main results agree with those found in earlier studies (Cann, 1992; Hartel and Langendoen, 1992).

Section 2 briefly characterises the functional languages that have been used. The compilers and interpreters for the functional languages are presented in Section 3. The Pseudoknot application is introduced in Section 4. Section 5 describes the translations of the program to the different programming languages. The benchmark results are presented in Section 6. The conclusions are given in the last section.

† Miranda is a trademark of Research Software Ltd.

language	source	ref.	typing	evaluation	order	match
		SML family				
Caml	INRIA	Weis (1993)	strong, poly	eager	higher	pattern
SML	Committee	Milner *et al.* (1990)	strong, poly	eager	higher	pattern
		Haskell family				
Clean	Nijmegen	Plasmeijer and van Eekelen (1994)	strong, poly	lazy	higher	pattern
Gofer	Yale	Jones (1994)	strong, poly	lazy	higher	pattern
LML	Chalmers	Augustsson and Johnsson (1989)	strong, poly	lazy	higher	pattern
Miranda	Kent	Turner (1985)	strong, poly	lazy	higher	pattern
Haskell	Committee	Hudak *et al.* (1992)	strong, poly	lazy	higher	pattern
RUFL	Rhodes	Wentworth (1992)	strong, poly	lazy	higher	pattern
		Lisp family				
Common Lisp	Committee	Steele Jr (1990)	dynamic	eager	higher	access
Scheme	Committee	Rees and Clinger (1991)	dynamic	eager	higher	access
		Parallel and concurrent languages				
Erlang	Ericsson	Armstrong *et al.* (1993)	dynamic	eager	first	pattern
Facile	ECRC	Thomsen *et al.* (1993)	strong, poly	eager	higher	pattern
ID	MIT	Nikhil (1991)	strong, poly	eager non-strict	higher	pattern
Sisal	LLNL	McGraw *et al.* (1985)	strong, mono	eager	first	none
		Intermediate languages				
ΓCMC	Recife	Lins and Lira (1993)	strong, poly	lazy	higher	access
Stoffel	Amsterdam	Beemster (1992)	strong, poly	lazy	higher	case
		Other functional languages				
Epic	CWI	Walters and Kamperman (1995)	strong, poly	eager	first	pattern
Opal	TU Berlin	Didrich *et al.* (1994)	strong, poly	eager	higher	pattern
Trafola	Saarbrücken	Alt *et al.* (1993)	strong, poly	eager	higher	pattern
		C				
ANSI C	Committee	Kernighan and Ritchie (1988)	weak	eager	first	none

Table 1. *Language characteristics. The source of each language is followed by a key reference to the language definition. The remaining columns characterise the typing discipline, the evaluation strategy, whether the language is first- or higher-order, and the pattern-matching facilities.*

2 Languages

The Pseudoknot benchmark takes into account a large number of languages and an even larger number of compilers. Our aim has been to cover as comprehensively as possible the landscape of functional languages, while emphasising typed languages.

Of the general purpose functional languages, the most prominent are the eager, dynamically typed languages Lisp and Scheme (the Lisp family); the eager, strongly typed languages SML and Caml (the SML family); and the lazy, strongly typed languages Haskell, Clean, Miranda and LML (the Haskell family). These languages are sufficiently well known to obviate an introduction. There are also some variants of these languages, such as the Gofer and RUFL variants of Haskell. The syntax and semantics of these variants is sufficiently close to that of their parents that no introduction is needed.

Four of our functional languages were designed primarily for concurrent/parallel applications. These are Erlang, an eager, concurrent language designed for prototyping and implementing reliable real-time systems; Facile, which combines SML with a model of higher-order concurrent processes based on CCS; ID, an eager, non-strict, mostly functional, implicitly parallel language; and Sisal, an eager, implicitly parallel functional language designed to obtain high performance on commercial scalar and vector multiprocessors.

The concurrent/parallel capabilities of these four languages have not been used in the Pseudoknot benchmark, so a further discussion of these capabilities is not relevant here. It should be pointed out however, that because these languages were intended for parallel execution, the sequential performance of some may not be optimal (See Section 6.3.3).

Two of the functional languages are intended to be used only as intermediate languages, and thus lack certain features of fully fledged programming languages, such as pattern matching. These languages are ΓCMC, a Miranda based language intended for research on the categorical abstract machine (Lins, 1987); and Stoffel, an intermediate language designed to study code generation for high level languages on fine-grained parallel processors. The Stoffel and ΓCMC compilers have been included because these compilers offer interesting implementation platforms, not because of the programming language they implement.

A further three functional languages were designed for a specific purpose: Epic is a language for equational programming, which was primarily created to support the algebraic specification language ASF+SDF (Bergstra *et al.*, 1989); Trafola is an eager language that was designed as a transformation language in a compiler construction project; and Opal is an eager language that combines concepts from algebraic specification and functional programming in a uniform framework.

Finally, C is used as a reference language to allow comparison with an imperative language.

Table 1 provides an overview of the languages that were benchmarked. The table is organised by language family. The first column of the table gives the name of the language. The second column gives the source (i.e. a University or a Company) if a language has been developed in one particular place. Some languages were designed by a committee, which is also shown. The third column of the table gives a key reference to the language definition.

The last four columns describe some important properties of the languages. The typing discipline may be *strong* (and static), *dynamic*, or *weak*; a strong typing discipline may be monomorphic (*mono*) or polymorphic (*poly*). The evaluation strategy may be *eager*, *lazy* or *eager* with *non-strict* evaluation. The language may be *first order* or *higher order*. Accessing components of data structures may be supported by either pattern-matching on function arguments, local definitions and/or as part of case expressions (*pattern*, *case*), by compiler generated access functions to destruct data (*access*), or not at all (*none*). The reader should consult the references provided for further details of individual languages.

3 Compilers

Having selected a large number of languages, we wished to provide comprehensive coverage of compilers for those languages. For a number of languages, we set out to benchmark more than one compiler, so as to provide direct comparisons between different implementations of some prominent languages as well as between the languages themselves.

For the Lisp family we use the CMU common Lisp native code compiler, and the Bigloo and Gambit portable Scheme to C compilers.

compiler	version	source	ref.	FTP / Email
Bigloo	1.7	INRIA Rocquencourt	Serrano (1994)	Ftp: ftp.inria.fr: /INRIA/Projects/icsla/Implementations/
Caml Light	0.61	INRIA Rocquencourt	Leroy (1993)	Ftp: ftp.inria.fr: /lang/caml-light/
Caml Gallium		INRIA Rocquencourt	Leroy (1992)	Email: Xavier.Leroy@inria.fr
Camloo	0.2	INRIA Rocquencourt	Serrano and Weis (1994)	Ftp: ftp.inria.fr: /lang/caml-light/
CeML	0.22	LIENS	Chailloux (1992)	Email: Emmanuel.Chailloux@ens.fr
Clean	1.0b	Nijmegen	Smetsers *et al.* (1991)	Ftp: ftp.cs.kun.nl: /pub/Clean/
CMU CL	17e	Carnegie Mellon	MacLachlan *et al.* (1992)	Ftp: lisp-sun1.slisp.cs.cmu.edu: /pub/
Epic	0.8	CWI	Walters and (1995) Kamperman	http://www.cwi.nl/ gipe/epic.html
Epic-C	0.2	CWI	Walters and (1995) Kamperman	http://www.cwi.nl/ gipe/epic.html
Erlang	6.0.4	Ellemtel AB	Hausman (1994)	commercial Email: erlang@erix.ericsson.se
Facile	Antigua	ECRC	Thomsen *et al.* (1993)	Email: facile@ecrc.de
FAST	33	Southampton/ Amsterdam	Hartel *et al.* (1994)	Email: pieter@fwi.uva.nl
Gambit	2.3	Montréal	Feeley and Miller (1990)	Ftp: ftp.iro.umontreal.ca: /pub/parallele/gambit/
ΓCMC	0.1	Recife Brazil	Lins and Lira (1993)	Email: rdl@di.ufpe.br
Gofer	2.30a	Yale	Jones (1994)	Ftp: nebula.cs.yale.edu: /pub/haskell/gofer/
Haskell	0.999.6	Chalmers	Augustsson (1993)	Ftp: ftp.cs.chalmers.se: /pub/haskell/chalmers/
Haskell	0.22	Glasgow	Peyton Jones *et al.* (1993)	Ftp: ftp.dcs.glasgow.ac.uk: /pub/haskell/glasgow/
Haskell	2.1	Yale	Yale Haskell group (1994)	Ftp: nebula.cs.yale.edu: /pub/haskell/yale/
ID	TL0 2.1	MIT/Berkeley	Nikhil (1991)	Email: chf@lcs.mit.edu
LML	0.999.7	Chalmers	Augustsson and Johnsson (1990)	Ftp: ftp.cs.chalmers.se: /pub/haskell/chalmers/
LML (OP–TIM)	Pre-rel.	Nottingham/ Kent	Thomas (1995)	Email: spt@cs.nott.ac.uk
MLWorks	n.a.	Harlequin Ltd.	Harlequin Ltd. (1994)	commercial
Miranda	2.018	Research Software Ltd.	Turner (1990)	commercial Email: mira-request@ukc.ac.uk
Nearly Haskell	Pre rel.	Chalmers	Röjemo (1995)	Ftp: ftp.cs.chalmers.se: /pub/haskell/nhc/
Opal	2.1c	Berlin	Schulte and Grieskamp (1991)	Ftp: ftp.cs.tu-berlin.de: /pub/local/uebb/ocs
RUFL	1.8.4	Rhodes	Wentworth (1991)	Ftp: cs.ru.ac.za: /pub/rufl/
Sisal	12.9.2	LLNL	Cann (1992)	Ftp: sisal.llnl.gov /pub/sisal
SML/NJ	1.07	AT&T Bell Labs.	Appel (1992)	Ftp: research.att.com: /dist/ml/
Stoffel		Amsterdam	Beemster (1993)	Email: beemster@fwi.uva.nl
Trafola	1.2	Saarbrücken	Alt *et al.* (1993)	Email: alt@cs.uni-sb.de

Table 2. *Compiler details consisting of the name of the compiler and/or language, the University or Company that built the compiler, a key reference to the description of the implementation and the address from which information about the compiler can be obtained.*

compiler	compiler options	execution options	collector	float
Bigloo	–unsafe –O4		mark-scan	double
Caml Light			gen.	double
Caml Gallium			gen.	double
Camloo	–unsafe –O4		mark-scan	double
CeML	–O		1-space	single
Chalmers	–c –Y–S –H50Mg – –Y–A500k –cpp		2-space	single
Clean		–nt –s 10k –h ...	2-space/ mark-scan	double
CMU CL	(speed 3) (safety 0) (debug 0) (compilation-speed 0)		2-space	single
Epic	-s80		mark-scan	single
Epic-C			mark-scan	single
Erlang BEAM	–fast	–h 600000	2-space	double
Facile			gen.	double
FAST	–fcg	–v 1 –h ... –s 400K 1	2-space	single
Gambit		–:h4096	2-space	double
ΓCMC			2-space	double
Glasgow	–O –fvia–C –O2–for–C	+RTS –H1M	gen.	single
Gofer	–DTIMER		2-space	single
ID	strict, merge–partitions (tlc: opt)		none	double
LML Chalmers	–H24000000 –DSTR –c		2-space	single
LML(OP–TIM)	LMLC: –H24000000 –DSTR –c –fno–code –fout–flic; SPGC: –c –i		2-space	single
MLWorks	no details available[a]		2-space	double
Miranda		/heap ...; /count	mark-scan	double
NHC(HBC)	–H30M		2-space	single
NHC(NHC)	–h2M		1-space	single
Opal	opt=full debug=no		refcount	single
RUFL	–w	–m300	mark-scan	double
RUFLI	–iw	–m300 –r32000	mark-scan	double
Sisal	–cpp –seq –O –c atan2 –cc=–O		refcount	double
SML/NJ			gen.	double
Stoffel	–O2 (for C)		2-space	double
Trafola	–TC –INLINE 1	–HE 8000000	1-space	sgl/dbl
Yale	see CMU CL		2-space	single

[a] MLWorks is not yet available. Compilation was for maximum optimisation, no debugging or statistics collection was taking place at runtime.

Table 3. *Compilation and execution options. The type of garbage collector is one of 2-space (non-generational 2-space copying collector); mark-scan; gen. (generational with two or more spaces); 1-space (mark-scan, one space compactor); or reference counting. Floating-point arithmetic used is either single- or double-precision.*

For the SML family we use: SML/NJ, an incremental interactive native code compiler; MLWorks, a commercial native code compiler; Caml Light, a simple byte-code compiler; Camloo, a Caml to C compiler derived from the Bigloo Scheme compiler; Caml Gallium, an experimental native-code compiler; and CeML, a compiler that has been developed to study translations of Caml into C.

For Haskell we use the Glasgow compiler, which generates either C or native code; the Chalmers native code compiler and the Yale compiler, which translates Haskell into Lisp. A large subset of Haskell is translated into byte-code by the NHC compiler. The Haskell relatives RUFL and Gofer can both compile either to native code or to a byte code. The Clean native code compiler from Nijmegen is used for Clean. For Miranda, the Miranda interpreter from Research Software Ltd is used, as well as the FAST compiler, which translates a subset of Miranda into C. For LML the Chalmers LML native code compiler is used, as well as a modified version that translates into a low-level intermediate form based on FLIC. After extensive optimisations (Thomas, 1993) this LML(OP–TIM) back-end generates native code.

For the four concurrent/parallel languages we use the Erlang BEAM compiler, a portable compiler that generates C; the Facile compiler, which uses the SML/NJ compiler to translate the SML code embedded in Facile programs into native code; the ID compiler, which translates into an intermediate data flow representation that is subsequently compiled into C by the Berkeley Tl0 back-end; and the Sisal compiler, which compiles via C with special provisions to update data structures in place, without copying.

Epic is supported by a so-called hybrid interpreter, which allows the combination of interpreted and compiled functions. Initially, all functions are translated to, essentially, byte-code. Then, individual functions can be translated into C, and can be linked to the system. This leads to a stratum of possibilities, with, in one extreme, all functions being interpreted, and in the other, all functions being compiled and only the dispatch overhead being paid. In this document, the two extremes are being benchmarked under the names Epic, and Epic-C, respectively.

Of the four remaining languages Opal, ΓCMC and Stoffel are translated into C whereas Trafola is translated into an interpreted code.

An overview of the compilers that have been used may be found in Table 2. Since this table is intended for reference rather than comparisons, the entries are listed in alphabetical order. The first column gives the name of the language and/or compiler, the second shows the source of the compiler. A key reference that describes the compiler is given in the third column. The last column gives instructions for obtaining the compiler by FTP or email.

To make the best possible use of each of the compilers, compilation and runtime options have been selected that should give fast execution. We have consistently tried to optimise for execution speed. In particular no debugging information, run time checks or profiling code have been generated. Where a '–O' option or higher optimisation setting could be used to generate faster code, we have done so. The precise option settings that were used for each compiler are shown in columns 2 and 3 of Table 3. The fourth column shows what type of garbage collection is used, and the last column indicates whether single or double floating-point precision was used. Where alternatives were available, we have chosen single-precision, since this should yield better performance.

4 Application

The Pseudoknot program is derived from a 'real-world' molecular biology application (Feeley *et al.*, 1994). In the following sections the program is described briefly from the point of view of its functional structure and its main operational characteristics. The level of detail provided should be sufficient to understand the later sections that describe the optimisations and performance analyses of the program. For more detail on the biological aspects of the program, the reader is referred to Feeley et al. (1994).

4.1 Functional behaviour

The Pseudoknot program computes the three-dimensional structure of part of a nucleic acid molecule from its primary structure (i.e. the nucleotide sequence) and a set of constraints on the three-dimensional structure. The program exhaustively searches a discrete space of shapes and returns the set of shapes that respect the constraints.

More formally, the problem is to find all possible assignments of the variables $x_1, \ldots, x_n$ ($n = 23$ here) that satisfy the structural constraints. Each variable represents the 3D position, orientation, and conformation (i.e. the internal structure) of a nucleotide in the nucleic acid molecule. Collectively they represent the 3D structure of the molecule. There are four types of nucleotides (A, C, G, and U), which contain from 30 to 34 atoms each. To reduce the search space, the domains of the variables are discretized to a small finite set (i.e. $x_i \in D_i$). These domains are dependent on the lower numbered variables (i.e. D_i is a function of $x_1, \ldots, x_{i-1}$) to take into account the restricted ways in which nucleotides attach relatively to one another. The constraints specify a maximal distance between specific atoms of the molecule.

The heart of the program is a backtracking search. For each possible assignment of x_1, all possible assignments of x_2 are explored, and so on until all variables are assigned a value. A satisfactory set of

floating-point operations		square root and trigonometric functions	
$\times$	3,567,672		
$+$	2,798,571	$\sqrt{\ }$	69,600
$>, \leq, <$	129,656	arctan	40,184
$-$	330,058	cos	40,184
$/$	40,184	sin	40,184
total	6,866,141	total	190,152

Table 4. *Breakdown of the 'real' work involved in the Pseudoknot problem as counted by the FAST system. The floating-point operations occurring in the trigonometric functions and the square root are not counted separately.*

assignments is a *solution*. As the search deepens, the constraints are checked to prune branches of the search tree that do not lead to a solution. If a constraint is violated, the search backtracks to explore the next possible assignment of the current variable. When a leaf is reached, the current set of assignments is added to the list of solutions. For the benchmark, there are 50 possible solutions.

The computation of the domains is a geometric problem which involves computations on 3D transformation matrices and points (3D vectors). Notable functions include `tfo_combine` (multiplication of 3D matrices), `tfo_align` (creation of a 3D matrix from three 3D vectors), and `tfo_apply` (multiplication of a 3D matrix by a 3D vector). Another important part of the program is the conformation database of nucleotides. This database contains the relative position of all atoms in all possible conformations of the four nucleotides (a total of 46 conformations). This data is used to align nucleotides with one another and to compute the absolute position of atoms in the molecule.

The program used in the present benchmarking effort is slightly different from the original (Feeley *et al.*, 1994). The latter only computed the number of solutions found during the search. However, in practice, it is the location of each atom in the solutions that is of real interest to a biologist, since the solutions typically need to be screened manually by visualising them consecutively. The program was thus modified to compute the location of each atom in the structures that are found. In order to minimise I/O overhead, a single value is printed: the distance from the origin to the farthest atom in any solution (this requires that the absolute position of each atom be computed).

4.2 Operational behaviour

The Pseudoknot program is heavily oriented towards floating-point computations, and floating-point calculations should thus form a significant portion of the total execution time. For the C version (executed on machine 10 c.f. Table 7) this percentage was found to be at least 25%.

We also studied the extent to which the execution of the functional versions is dominated by floating-point calculations, using state-of-the-art compilers for the eager SML and the lazy FAST versions of the program. The time profile obtained for the MLWorks compiler for SML suggests that slightly over 50% of the run time is consumed by three functions, `tfo_combine`, `tfo_align` and `tfo_apply`, which do little more than floating-point arithmetic and trigonometric functions. This means that half the time, little more than the floating-point capability of this implementation is being tested, and some of that functionality is actually provided by an operating system library.

Statistics from the lazy FAST compiler show that with lazy evaluation the most optimised version of the program does about 7 million floating-point operations, excluding those performed by the 190 thousand trigonometric and square root function calls. A detailed breakdown of these statistics is shown in Table 4. Overall, the program makes about 1.5 million function calls and claims about 15 Mbytes of space (the maximum live data is about 30 Kbytes).

5 Translations, annotations and optimisations

The Pseudoknot program was hand-translated from either the Scheme or the C version to the various other languages that were benchmarked. All versions were hand-tuned to achieve the best possible performance for each compiler. The following set of guidelines were used to make the comparison as fair as possible:

1. Algorithmic changes are forbidden but slight modifications to the code to allow better use of a particular feature of the language or programming system are allowed.
2. Only small changes to the data structures are permitted (e.g. a tuple may be turned into an array or a list).
3. Annotations are permitted, for example strictness annotations, or annotations for inlining and specialisation of code.
4. All changes and annotations should be documented.
5. Anyone should be able to repeat the experiments. So all sources and measurement procedures should be made public (by `ftp` somewhere).
6. All programs must produce the same output (the number 33.7976 to 6 significant figures).

The optimisations and annotations made to obtain best performance with each of the compilers are discussed in the following subsections. We will make frequent reference to particular parts of the program text. As the program is relatively large it would be difficult to reproduce every version in full here. The reader is therefore invited to consult the archive that contains most of the versions of the program at `ftp.fwi.uva.nl`, file `/pub/computer-systems/functional/packages/pseudoknot.tar.Z`.

The guidelines above were designed on the basis of the experience gained at the Dagstuhl workshop with a small subset of the present set of implementations. We tried to make the guidelines as clear and concise as possible, yet they were open to different interpretations. The problems we had were aggravated by the use of different terminology, particularly when one term means different things to people from different backgrounds. During the process of interpreting and integrating the benchmarking results in the paper we have made every effort to eradicate the differences that we found. There may be some remaining differences that we are unaware of.

In addition to these unintentional differences, there are intentional differences: some experimenters spent more time and effort improving their version of Pseudoknot than others. These efforts are documented, but not quantified, in the sections that follow.

5.1 Sources used in the translations

The translation of the Pseudoknot program into so many different languages represented a significant amount of work. Fortunately this work could be shared amongst a large number of researchers. The basic translations were not particularly difficult or interesting, so we will just trace the history of the various versions. The optimisations that were applied will then be discussed in some detail in later sections.

The Scheme version of the Pseudoknot benchmark was used as the basis for the Bigloo, Gambit, CMU Common Lisp, and Miranda versions.

The Miranda source was used to create the Clean, FAST, Erlang, ΓCMC, Gofer, Haskell, Stoffel and SML sources. The Haskell source was subsequently used to create the ID, RUFL and LML sources, and together with the SML source to create the Opal source. The SML version was subsequently used as the basis for the translation to Caml, Epic and Facile.

The Sisal version is the only functional code to have been derived from the C version of the program.

Some typed languages (RUFL, Opal) require explicit type signatures to be provided for all top level functions. For other languages (SML/NJ) it was found to be helpful to add type signatures to improve the readability of the program.

All but two of the sources were translated by hand: the Stoffel source was translated by the FAST compiler from the Miranda source and the Epic source was produced by a translator from (a subset of) SML to Epic, which was written in Epic.

5.2 Splitting the source

Most of the compilers that were used have difficulty compiling the Pseudoknot program. In particular the C compilers, and also most of the compilers that generate C, take a long time to compile the program. For example, GCC 2.5.8 requires more than 900 seconds (on machine 10, See table 7) to compile the program with the –O optimisation enabled. The bundled SUN CC compiler takes over 300 seconds (with the same option setting and on the same machine).

The reason it takes so long to compile Pseudoknot is because the program contains four large functions (which in C comprise 574, 552, 585 and 541 lines of code, respectively) which collectively build the conformation database of nucleotides. These functions contain mostly floating-point constants. If the bodies of these four functions are removed, leaving 1073 lines of C code, the C compilation time is reduced to approximately 13 seconds, for both SUN CC and GCC. Since the functional versions of the program have the same structure as the C version, the functional compilers are faced with the same difficulty.

In a number of languages that support separate compilation (e.g. Haskell, LML and C), the program has been split into 6 separate modules. Each global data structure is placed in its own module which is then imported by each initialisation module. The main program imports all of these modules. Splitting the source reduced the compilation times by about 8% for GCC and 3% for SUN CC. As the main problem is the presence of large numbers of floating-point constants in the source, this is all we could hope for.

The NHC compiler is designed specifically to compile large programs in a modest amount of space. There are two versions of this compiler: NHC(HBC), which is the NHC compiler when compiled by the Chalmers Haskell compiler HBC; and NHC(NHC), which is a bootstrapped version. The monolithic source of the Pseudoknot program could be compiled using less than 8 MB heap space by NHC(NHC), whereas NHC(HBC) requires 30 MB heap space. HBC itself could not compile the monolithic source in 80 MB heap space, even when a single-space garbage collector was used.

A number of the functional compilers that compile to C (e.g. Opal and FAST) generated such large or so many C procedures that some C compilers had trouble compiling the generated code. For example, the C code generated by Epic-C consists of many functions occupying 50000 lines of code. It is generated in 3.5 minutes, but had to be split by hand in order for the gcc compiler to compile it successfully (taking 2.5 hours; both times on machine 10 c.f. Table 7).

As a result of the Pseudoknot experience, the Opal compiler has been modified to cope better with extremely large functions such as those forming the nucleotide database.

One way to dramatically reduce C compilation time at the expense of increased run time is to represent each vector of floating-point constants in the conformation database as a single large string. This string is then converted into the appropriate numeric form at run time. For the Bigloo compiler, which uses this technique to successfully reduce compilation time, the run time penalty amounted to 30% of the total execution time.

5.3 Purity

To allow a fair comparison of the quality of code generation for pure functions, none of the functional versions of Pseudoknot exploit side-effects where these are available in the source language.

5.4 Typing

Most of the languages are statically typed, in which case the compilers can use this type information to help generate better code. Some of the compilers for dynamically typed languages can also exploit static type information when this is provided.

For example, the Erlang version of Pseudoknot used guards to give some limited type information, as in the following function definition where X1 etc. are typed as floating-point numbers.

```
> pt_sub({X1,Y1,Z1},{X2,Y2,Z2})
>       when float(X1),float(Y1),float(Z1),
```

```
>            float(X2),float(Y2),float(Z2) -> {X1-X2,Y1-Y2,Z1-Z2}.
```

Similarly, for Common Lisp, type declarations were added to all floating-point arithmetic operations and to all local variables that hold floating-point numbers. This was unnecessary for the Scheme version, which already had calls to floating-point specific arithmetic functions.

5.5 Functions

Functional abstraction and application are the central notions of functional programming. Each of the various systems implements these notions in a different way. This, in turn, affects the translation and optimisation of the Pseudoknot program.

There is considerable variety in the treatment of function arguments. Firstly, some languages use curried arguments; some use uncurried arguments; and some make it relatively cheap to simulate uncurried arguments through the use of tuples when the normal argument passing mechanism is curried. Secondly, higher-order languages allow functions to be passed as arguments to other functions; whereas the first-order languages restrict this capability. Finally, even though most languages support pattern-matching, some do not allow 'as-patterns', which make it possible to refer to a pattern as a whole, as well as to its constituent parts.

There are several other issues that affect the cost of function calls, such as whether function bodies can be expanded 'in-line', and whether recursive calls can be transformed into tail recursion or loops. These issues will now be discussed in relation to the Pseudoknot program, with the effects that they have on the performance where these are significant.

5.5.1 Curried arguments

The SML/NJ source of the Pseudoknot program is written in a curried style. In SML/NJ version 1.07, this proved to have a relatively small effect on performance (less than 5% improvement compared with an uncurried style). For the older version of the SML/NJ compiler used for the Facile system, however (version 0.93), some of the standard compiler optimisations appear to be more effective on the uncurried than on the curried version of the program. In this case the difference was still less than 10%.

5.5.2 Higher-order functions

The Pseudoknot program occasionally passes functions as arguments to other functions. This is obviously not supported by the three first order languages Sisal, Erlang and Epic. The Sisal code was therefore derived from the C program, where this problem had already been solved. Erlang took the alternative approach of eliminating higher-order calls using an explicit application function `p_apply`. For example `reference` is called by:

```
> p_apply(reference,Arg1,Arg2,Arg3) -> reference(Arg1,Arg2,Arg3).
```

where `reference` is a constant (it is a static function name). In Epic a similar mechanism was used.

Higher-order functions are generally expensive to implement so many compilers will make attempts to reduce how often such functions are used. In most cases higher-order functions simply pass the name of a statically known function to some other function. These cases can be optimised by specialising the higher order function. Many compilers will specialise automatically, in some cases this has been achieved manually. For example for Yale Haskell the functions `atom_pos` and `search` were inlined to avoid a higher order function call.

5.5.3 Patterns

Some functions in the Pseudoknot program first destruct and then reconstruct a data item. In CeML, Haskell, LML and Clean as-patterns have been used to avoid this. As an example, consider the following Haskell fragment:

```
> atom_pos atom v@(Var i t n) = absolute_pos v (atom n)
```

Here the rebuilding of the constructor `(Var i t n)` is avoided by hanging on to the structure as a whole via the variable `v`. The Epic compiler automatically recognises patterns that occur both on the left and the right hand side of a definition; such patterns are never rebuilt.

This optimisation has not been applied universally because as-patterns are not available in some languages (e.g. Miranda). In FAST, a similar effect has been achieved using an auxiliary function:

```
> atom_pos atom v     = absolute_pos v (atom (get_nuc v))
> get_nuc (Var i t n) = n
```

The benefits of avoiding rebuilding a data structure do not always outweigh the disadvantage of the extra function call, so this change was not applied to the other languages.

Neither of the two intermediate languages support pattern matching. To access components of data structures ΓCMC uses access functions; Stoffel uses case expressions.

5.5.4 Inlining

Functional programs normally contain many small functions, and the Pseudoknot program is no exception. Each function call carries some overhead, so it may be advantageous to inline functions, by expanding the function body at the places where the function is used. Small functions and functions that are only called from one site are normally good candidates for inlining. Many compilers will automatically inline functions on the basis of such heuristics, and some compilers (e.g. Opal, Chalmers Haskell, Glasgow Haskell) are even capable of inlining functions across module boundaries.

For Clean, FAST, Trafola and Yale Haskell many small functions (in particular the floating-point operator definitions) and constants were inlined explicitly.

5.5.5 Tail recursion and loops

Tail recursive functions can be compiled into loops, but some languages offer loop constructs to allow the programmer to express repetitive behaviour directly. In ID and Sisal the recursive function `get_var` is implemented using a loop construct. In Epic, this function coincides with a built-in polymorphic association table lookup, which was used instead. In ID the backtracking search function `search` has also been changed to use a loop instead of recursion.

5.6 Data structures

The original functional versions of the Pseudoknot program use lists and algebraic data types as data structures. The preferred implementation of the data structures is language and compiler dependent. We will describe experiments where lists are replaced by arrays, and where algebraic data types are replaced by records, tuples or lists.

For the lazy languages strictness annotations on selected components of data structures and/or function arguments also give significant performance benefits.

5.6.1 Avoiding lists

The benchmark program computes 50 solutions to the Pseudoknot constraint satisfaction problem. Each solution consists of 23 variable bindings, that is one for each of the 23 nucleotides involved. This creates a total of $50 \times 23 = 1150$ records of atoms for which the distance from the origin to the furthest atom must be computed. These 1150 records each contain between 30 and 34 atoms, depending on the type of the nucleotide (33 for type A, 31 for type C, 34 for type G and 30 for type U). The sizes of these records of atoms are determined statically so they are ideal candidates for being replaced by arrays. The advantage of using an array instead of a list of atoms is the amortised cost of allocating/reclaiming all atoms at once. A list of atoms is traversed linearly from the beginning to the end, so the unit access cost of the array does not give an extra advantage in this case. This change from lists to arrays has been implemented in Caml, ID, and Scheme.

In the Sisal code, the problem described above does not arise: instead of building the 1150 records, a double loop traverses the 50×23 records. A further loop computes the maximum distance to the origin. Consequently, no intermediate lists or arrays are created.

The local function `generate` within `p_03'` was replaced by an ID array comprehension; in Sisal a loop construct was used.

5.6.2 Avoiding algebraic data types

Some of the algebraic data type constructors in the Pseudoknot program are rather large, with up to 34 components. This leads to distinctly unreadable code when pattern matching on arguments of such types and it may also cause inefficiencies.

For Sisal all algebraic data types were replaced by arrays, since Sisal compilers are specifically optimised towards the efficient handling of arrays.

For SML/NJ the 12 component coordinate transformation matrix `TFO` was changed to an array representation. This was found not to make a significant difference.

For the Caml Gallium compiler, some of the algebraic data types have been converted into records to guide the data representation heuristics; this transformation makes no difference for the other Caml compilers, Caml light and Camloo.

Trafola, Epic and RUFL implement algebraic data types as linked lists of cells, which implies a significant performance penalty for the large constructors used by Pseudoknot.

5.6.3 Strictness annotations

The Pseudoknot program does not benefit in any way from lazy evaluation because all computations contained in the program are mandatory. It is thus feasible to annotate the data structures (i.e. lists, algebraic data types and tuples) in the program as strict. Those implementations which allowed strictness annotations only had to annotate the components of algebraic data types as strict to remove the bulk of the lazy evaluations. The Gofer, Miranda, NHC, RUFL and Stoffel compilers do not permit strictness annotations, but a variety of strictness annotations were tried with the other compilers.

For Yale, Chalmers Haskell and the two LML compilers, all algebraic data types were annotated as strict; for ΓCMC the components of `Pt` and `TFO` were annotated as strict; for Clean, the components of `Pt`, `TFO` and the integer component of `Var` were annotated as strict; for FAST all components of these three data types were annotated as strict. For Yale Haskell the first argument of `get_var` and the arguments of `make_relative_nuc` were also forced to be strict. This is permissible since the only cases where these arguments are not used give rise to errors, and are thus equivalent to demanding the value of the arguments.

Depending on the compiler, strictness annotations caused the Pseudoknot execution times to be reduced by 50%–75%.

5.6.4 Unboxing

The Pseudoknot program performs about 7 million floating-point operations. Unless special precautions are taken, the resulting floating-point numbers will be stored as individual objects in the heap (a 'boxed' representation). Representing these values as unboxed objects that can be held directly in registers, on the stack, or even as literal components of boxed structures such as lists, has a major impact on performance: not only does it reduce the space requirements of the program, but the execution time is also reduced since less garbage collection is required if less space is allocated.

There are a number of approaches that can be used to avoid maintaining boxed objects: Caml Gallium, SML/NJ, Bigloo and Gambit provide an analysis that will automatically unbox certain objects; CMU common Lisp and Glasgow Haskell provide facilities to explicitly indicate where unboxed objects can safely be used. Our experience with each of these techniques will now be described in some detail, as it provides useful insight into the properties of this relatively new technology.

The Caml Gallium compiler employs a representation analysis (Leroy, 1992), which automatically exploits an unboxed representation for double-precision floating-point numbers when these are used monomorphically. Since the Pseudoknot benchmark does not use polymorphism, all floating-point numbers are unboxed. This is the main reason why the Gallium compiler generates faster code than most of the other compilers.

The latest version of the SML/NJ compiler (version 1.07) also supports automatic unboxing through a representation analysis (Shao, 1994). However, unlike Caml Gallium, it does not directly exploit special load and store instructions to transfer floating-point numbers to and from the FPU. Changing this should improve the overall execution time for this compiler.

In an attempt to find better performance, a large number of variations were tried with the SML/NJ compiler. The execution time was surprisingly stable under these changes, and in fact no change made any significant difference, either good or bad, to the execution speed. In the end, the original transcription of the Scheme program, with a type signature for the main function was used for the measurements. A similar result was found for the MLWorks compiler, where a few optimisations were tried, and found to give only a marginal improvement (of 2%). The MLWorks timings apply to essentially the same source as the SML/NJ timings. MLWorks generates slightly faster code than SML/NJ for this program.

The SML/NJ implementation of the Pseudoknot program actually performs better on the DECstation 5000 than on the SPARC. On the DECstation 5000 it runs at 55% of the speed of C, whereas on the SPARC it runs at only 36% of the speed of C. We suspect that this is mainly due to memory effects. Previous studies (Diwan *et al.*, 1994) have shown that the intensive heap allocation which is characteristic of the SML/NJ implementation interacts badly with memory subsystems that use a write-no-allocate cache policy, as is the case of the SPARC; in contrast, the use of a write-allocate policy coupled with what amounts to sub-block placement on the DECstation (the cache block size is four bytes) supports such intensive heap allocation extremely well.

The Bigloo compiler uses a two-step representation analysis. The first step is a control flow analysis that identifies monomorphic parts of the program. The second step improves the representation of those objects that are only used in these monomorphic parts. Unfortunately, it is not possible to avoid boxing entirely because some data structures are used heterogeneously in the Scheme source (e.g. floating-point numbers, booleans, and vectors are contained in the same vector). Even so, of the 7 million floating-point values that are created by the Pseudoknot program, only 700 thousand become boxed.

The Gambit compiler uses two simple 'local' methods for reducing the number of floating-point numbers that are boxed. Firstly, intermediate results for multiple argument arithmetic operators, such as when more than two numbers are added, are never boxed. This means that only 5.3 million of the 7 million floating-point results need to be considered for boxing. Secondly, Gambit uses a lazy boxing strategy, whereby floating-point results bound to local variables are kept in an unboxed form and only boxed when they are stored in a data structure, passed as a function argument, or when they are live at a branch (i.e. at a function call or return). Of the 5.3 million floating-point results that might need to be boxed, only 1.4 million actually become boxed. This optimisation decreases the run time by roughly 30%.

In the Epic implementation specialised functions were defined for the two most common floating point expressions (two- and three-dimensional vector inproduct), leading to a 41% reduction of function calls and (un)boxing. Although the new functions were trivially written by hand, their utilization was added automatically by the addition of two rewrite rules to the – otherwise unaltered – SML-to-Epic translator. This is possible because Epic, unlike many functional languages, does not distinguish constructor symbols from defined function symbols. Consequently, laws (in the sense of Miranda: (Thompson, 1986); in Epic *all* functions are defined by laws) can be introduced, which map specific patterns such as $x_1 * x_2 + x_3 * x_4$, to semantically equivalent, but more efficient patterns which use a newly introduced function (i.e., $\mathit{inprod2}(x_1, x_2, x_3, x_4)$).

In the Common Lisp version of the program, the `Pt` and `TFO` data types were implemented as vectors specialised to hold untagged `single-float` objects, rather than as general vectors of tagged objects. This is equivalent to unboxing those floating-point numbers.

Cost Centre	%time	%alloc
tfo_combine	18.0	4.7
tfo_apply	15.9	0.0
p_o3'	8.3	25.5
tfo_align	6.2	1.9
dgf_base	5.9	21.6
get_var	5.9	0.0
absolute_pos	4.7	24.1
...	...	...

(a) Original profile (by time)

Cost Centre	%time	%alloc
get_var	11.1	0.0
tfo_combine	10.5	26.5
p_o3'	8.5	13.0
pseudoknot_constraint	7.8	9.9
search	7.8	2.6
tfo_align	5.2	5.6
pt_phi	5.2	0.0
tfo_apply	5.2	0.0
...	...	...

(c) Maximum map (by time)

Cost Centre	%time	%alloc
tfo_apply	11.1	0.0
tfo_combine	10.5	23.2
search	9.9	2.3
pseudoknot_constraint	8.2	8.7
get_var	7.0	0.0
var_most_distant	6.4	8.7
tfo_align	5.8	4.9
...	...	...

(b) Strict types (by time)

Cost Centre	%time	%alloc
tfo_combine	9.7	26.5
p_o3'	7.7	13.0
pseudoknot_constraint	4.6	9.9
mk_var	2.0	6.6
tfo_align	10.2	5.6
tfo_inv_ortho	2.6	5.6
...	...	...

(d) Maximum map (by allocation)

Table 5. *Time and allocation profile of Pseudoknot from the Glasgow Haskell system by function, as a percentage of total time/heap allocations.*

The Glasgow Haskell compiler has provisions for explicitly manipulating unboxed objects, using the type system to differentiate boxed and unboxed values (Peyton Jones and Launchbury, 1991). The process of engineering the Pseudoknot code to reduce the number of boxed floating-point numbers is a good illustration of how the Glasgow profiling tools can be used. We therefore present this aspect of the software engineering process in detail below.

The version of Pseudoknot which was used for Chalmers Haskell ran in 10.2 seconds when compiled with the Glasgow Haskell Compiler for machine 16 c.f. Table 7. The raw time profiling information from this program (See Table 5-a) shows that a few functions account for a significant percentage of the time used, and over 80% of the total space usage. Three of the top four functions by time (`tfo_combine`, `tfo_apply` and `tfo_align`) manipulate `TFO`s and `Pt`s, and the remainder are heavy users of the `Var` structure. Since these functions can be safely made strict, they are prime candidates to be unboxed, as was also done with the Common Lisp compiler.

By unboxing these data structures using a simple editor script and changing the pattern match in the definition of `var_most_distant_atom` so that it is strict rather than lazy, an improvement of roughly a factor of 3 is obtained. This is similar to the improvements which are possible by simply annotating the relevant data structures to make them strict as with the Chalmers Haskell compiler. However, further unboxing optimisations are possible if the three uses of the function composition `maximum . map` are replaced by a new, specialised function `maximum_map` as shown below. This function maps a function whose result is a floating-point number over a list of arguments, and selects the maximum result. It is not possible to map a function directly over a list of unboxed values using the normal Prelude `map` function, because unboxed values cannot be passed to polymorphic functions.

```
> maximum_map :: (a->Float#) -> [a]->Float#
> maximum_map f (h:t) =
>    max f t (f h)
>    where max f (x:xs) m = max f xs (let fx = f x in
```

Version	seconds	Mbytes	(Residency)
Original	10.0 + 0.2	36.8	(55K)
Strict Types	3.5 + 0.3	10.1	(53K)
Maximum Map	1.8 + 0.1	7.6	(46K)

Table 6. *Time and Heap Usage of three Pseudoknot variants compiled for machine 16 by the Glasgow Haskell compiler.*

```
>                                       if fx `gtFloat#` m then fx else m)
>           max f [] m = m
>           max :: (a->Float#) -> [a] -> Float# -> Float#
```

This optimisation is suggested indirectly by the time profile (Table 5-b) which shows that the top function by time is `tfo_apply`. This is called through `absolute_pos` within `most_distant_atom`. Merging the three nested function calls that collectively produce the maximum value of a function applied to a list of arguments allows the compiler to determine that the current maximum value can always be held in a register (an extreme form of deforestation (Wadler, 1990)). When this transformation is applied to the Haskell source, the total execution time is reduced to 1.8 seconds user time (still on machine 16). An automatic generalised version of this hand optimisation, the foldr/build transformation (Gill and Peyton Jones, 1994), has now been incorporated into the Glasgow Haskell compiler.

The final time profile (Table 5-c) shows `get_var` and `p_o3'` jointly using 20% of the Haskell execution time with `tfo_combine`, `tfo_align` and `tfo_apply` accounting for a further 20%. (The minor differences in percentage time for `tfo_combine` in Tables 5-c and 5-d are probably explained by sampling error over such a short run). While the first two functions could be optimised to use a non-list data structure, it is not easy to optimise the latter functions any further. The total execution time is now close to that for C, with a large fraction of the total remaining time being spent in the Unix mathematical library. Since the allocation profile (Table 5-d) suggests that there are no space gains which can be obtained easily, it was decided not to attempt further optimisations. The overall time and space results for Glasgow Haskell are summarised in Table 6. In each case, the heap usage reported is the total number of bytes that were allocated, with the maximum live data residency after a major garbage collection shown in parentheses.

The foldr/build style deforestation of `maximum . map` has also been applied to the ID, SML and Scheme sources. For SML/NJ this transformation, and other, similar deforestation transformations made no measurable improvement (though several led to minor slowdowns).

5.6.5 Single threading

In a purely functional program, a data structure cannot normally be modified once it has been created. However, if the compiler can detect that a data structure is modified by only one operation and that this operation executes after all other operations on the data structure (or can be so delayed), then the compiler may generate code to modify the data structure in place. The Sisal compiler includes special optimisations (preallocation (Ranelletti, 1987) and copy elimination (Gopinath and Hennesy, 1989)) that make safe destructive updates of data structures possible. In order to exploit this, the Sisal version of the Pseudoknot program was written so as to expose the single threaded use of some important data structures. An example is given below, where the array `stack` is single threaded, so that the new versions `stack1` and `stack2` occupy the same storage as the original `stack`:

C (Pseudo code)	Sisal revised
	`> let`
add new element to stack	`>   stack1 := array_addh(stack,element)`
increment stack counter	
call pseudoknot_domains	`>   stack2 := pseudoknot_domains(stack1, ... )`
	`> in`
decrement stack counter	`>   array_remh(stack2)`
	`> end let`

In principle, this code is identical to the C code. The Sisal compiler realises that there is only a single consumer of each stack. It tags the data structure as mutable and generates code to perform all updates in place. Consequently, the Sisal code maintains a single stack structure similar to the C code, eliminating excessive memory usage and copy operations. As in the C code, when a solution is found, a copy of the stack is made to preserve it. The Sisal code runs in approximately 85KB of memory and achieves execution speeds comparable to the C code.

5.7 Floating-Point Precision

When comparing our performance results, there are several reasons why floating-point precision must be taken into account. Firstly, it is easier to generate fast code if single-precision floating-point numbers are used, since these can be unboxed more easily. Secondly, both memory consumption and garbage collection time are reduced, because single-precision floating-point numbers can be represented more compactly. Thirdly, single-precision floating-point arithmetic operations are often significantly faster than the corresponding double-precision operations.

Traditionally, functional languages and their implementations have tended to concentrate on symbolic applications. Floating-point performance has therefore been largely ignored. One notable exception is Sisal, which is intended more as a special-purpose language for scientific computations than as a general-purpose language.

Since single-precision gives sufficient accuracy for the Pseudoknot program on our benchmark machine, and since single-precision operations are faster than double-precision operations on this architecture, compilers that can exploit single-precision arithmetic are therefore at some advantage. The advantage is limited in practice by factors such as the dynamic instruction mix of the code that is executed: for example, for the GNU C version of Pseudoknot overall performance is improved by only 12% when single-precision floating-point is used; for the Trafola interpreter, however, performance was improved by 16%; and for the Opal compiler, performance was improved by a factor of 2.

6 Results

Comparative time measurements are best done using a single platform. However, many of the compilers are experimental and in constant flux. They are therefore difficult to install at another site in a consistent and working state. Therefore we have decided to collect the compiled binaries of the Pseudoknot program, so as to be able to execute all binaries on the same platform. The measured execution times of the programs are thus directly comparable and accurate.

The compile times are not directly comparable. To make a reasonable comparison possible, a relative measure has been defined. This is somewhat inaccurate, but we think that this is quite acceptable, since for compile times it is more the order of magnitude that counts than precise values. The relative unit ‘pseudoknot’ relates compilation time to the *execution* time of the C version of the Pseudoknot program, where both are measured on the *same* platform. The more obvious alternative of comparing to the C compilation times of Pseudoknot was rejected because not all architectures that are at stake here use the same C compiler (See Table 7). The Pseudoknot is computed as:

$$\text{relative speed} \;=\; \frac{1000 \times \text{C execution time}}{\text{compilation time}}$$

no.	SUN machine	mem.	cache	op. system	processor	C compiler
1	4/50	32 M	64 K	SunOS 4.1.3.	standard	gcc 2.5.8
2	4/75	64 M	64 K	SunOS 4.1.3.	standard	gcc 2.5.8
3	4/330	96 M	128 K	SunOS 4.1.1.	standard	gcc 2.5.4
4	4/630MP	64 M	64 K	SunOS 4.1.2	SUNW	gcc 2.4
5	4/670	64 M	64 K	SunOS 4.1.3.	standard	gcc 2.5.7
6	4/670MP	64 M	64 K	SunOS 4.1.3	TMS390Z55	gcc 2.5.7
7	4/670	64 M	64 K	SunOS 4.1.3.	TI Supersparc	gcc 2.5.7
8	4/670	64 M	64 K	SunOS 4.1.2.	Cypress CY605	gcc 2.4.5
9	4/670MP	64 M	1 M	SunOS 4.1.3.	SUNW, system 600	gcc 2.5.8
10	4/690	64 M	64 K	SunOS 4.1.2.	standard	gcc 2.5.8
11	4/690MP	64 M	64 K	SunOS 4.1.3	ROSS 40MHz Super	cc
12	4/690	64 M	1 M	SunOS 4.1.3.	standard	gcc 2.5.8
13	SPARC 10/30	32 M	1 M	SunOS 4.1.3.	TMS390Z55	gcc 2.5.8
14	SPARC 10/41	64 M	1 M	SunOS 4.1.3.	standard	gcc 2.5.7
15	SPARC 10/41	96 M	1 M	SunOS 4.1.3.	standard	gcc 2.5.8
16	SPARC 10/41	96 M	20K/32K+1M	SunOS 4.1.3.	TMS390Z50	gcc 2.5.7
17	SPARC 10/41	128 M	20K/32K+1M	Solaris 2.3	standard	gcc 2.5.8
18	SPARCStat. 5	64 M	16K/8K	SunOS 4.1.3	standard	gcc 2.6.0
19	SPARCStat. 20	128 M	16K/20K+1M	SunOS 4.1.3	Supersparc	gcc 2.5.8

Table 7. *Details of the SUN machines and C compilers used to compile the Pseudoknot program. The type of the machine is followed by the size of the memory (in MB), the size of the cache (as a total or as instruction/data + secondary cache size), the operating system name and version, and the type of processor used. The last column gives the C compiler/version that has been used on the machine.*

To 'compile at 1000 knots' thus means to take the same amount of time to compile as it takes the C version of Pseudoknot to execute.

With so many compilers under scrutiny, it is not surprising that a large number of machines are involved in the different compilations. The most important characteristics of the SUN machines may be found in Table 7. The table is ordered by the type of the machine.

To measure short times with a reasonable degree of accuracy, the times reported are an average of either 10 or 100 repeated runs. The resulting system and user times divided by 10 (100) are reported in the Tables 8 and 9.

6.1 Compile time

Table 8 shows the results of compiling the programs. The first column of the table shows the name of the compiler (c.f. Table 2). The second column 'route' indicates whether the compiler produces native code ('N'), code for a special interpreter ('I'), or compiles to native code through a portable C back-end ('C'), Lisp ('L'), or Scheme ('S'), or through a combination of these back-ends. The third column gives a reference to the particular machine used for compilation (c.f. Table 7). The next three columns give the user+system time and the space required to compile the Pseudoknot program. Unless noted otherwise, the space is the largest amount of space required by the compiler, as obtained from `ps -v` under the heading SIZE. The column marked 'C-runtimes' gives the user+system time required to execute the C version of the Pseudoknot program on the same machine as the one used for compilation. The last two columns 'pseudoknots' show the relative performance of the compiler with respect to the C-runtimes.

It is possible to distinguish broad groups within the compilers. The faster compilers are, unsurprisingly, those that compile to an intermediate code for a byte-code or similar interpreter, and which therefore perform few, if any, optimisations. NHC is an outlier, perhaps because unlike the other compilers in this group, it is a bootstrapping compiler.

With the exception of Bigloo and Camloo, which are faster than many native compilers, implementa-

compiler	route	mach.	times user	+	sys	space Mb	A/H[a]	C-runtimes user	+	sys	pseudoknots user	+	sys
Compiled via another high level language: C, L(isp) or S(cheme)													
Bigloo	C	5	56.5	+	6.4	7.5	A	3.0	+	0.1	53	+	16
Camloo	S+C	5	98	+	17.6	4.6	A	3.0	+	0.1	31	+	6
Sisal	C	9	112	+	13.3	2.4	A	1.4	+	0.1	12	+	8
Gambit	C	15	167	+	4.2	8.7	A	1.7	+	0.1	10	+	24
Yale	L	11	610	+	186	14	H	4.7	+	0.1	8	+	1
ΓCMC	C	4	332	+	11	13	A	2.7	+	0.3	8	+	27
FAST	C	10	450	+	40	100	A	2.7	+	0.1	6	+	2
Opal	C	2	1301	+	19	15	A	3.0	+	0.1	2	+	5
Glasgow	C	16	564	+	30	47	A	1.3	+	0.1	2	+	3
Erlang BEAM	C	1	> 1 Hour			8	A	3.3	+	0.1			
CeML	C	5	> 1 Hour			35	A	3.0	+	0.1			
ID	C	15	> 1 Hour			64	A	2.8	+	0.1			
Epic-C	C	10	> 2 Hours			12.4	A	2.7	+	0.1			
Stoffel	C	17	> 2 Hours			25	A	1.3	+	0.1			
Compiled into native code													
Clean	N	8	30	+	10	9	A	2.7	+	0.1	90	+	10
RUFL	N	10	41.6	+	8	3	A	2.7	+	0.1	65	+	12
CMU CL	N	11	118	+	25	14	H	4.7	+	0.1	40	+	4
Caml Gallium	N	7	45.9	+	2.0	3.8	A	1.4	+	0.1	31	+	50
SML/NJ	N	19	40.3	+	2.3	35	A	0.9	+	0.1	22	+	43
LML Chalmers	N	18	78.7	+	24.0	14.2	A	1.3	+	0.1	17	+	4
LML(OP–TIM)	N	18	85.5	+	13.5	13.6	A	1.3	+	0.1	15	+	7
Facile	N	14	123	+	2.5	11.3	A	1.7	+	0.1	14	+	40
MLWorks	N	3	394	+	19	14.4	R	4.9	+	0.1	12	+	5
Chalmers	N	13	181	+	45	50	A	1.3	+	0.1	7	+	2
Interpreted													
Gofer	I	10	6.7	+	0.7	3	A	2.7	+	0.1	403	+	143
RUFLI	I	10	9.1	+	1.7	1	A	2.7	+	0.1	297	+	59
Miranda	I	10	12.5	+	0.8	13	A	2.7	+	0.1	216	+	125
Caml Light	I	6	29.7	+	1.1	2.3	A	2.7	+	0.1	91	+	91
Trafola	I	10	31.4	+	11.5	6	A	2.7	+	0.1	86	+	9
Epic	I	10	114	+	1.6	8.4	A	2.7	+	0.1	24	+	62
NHC(HBC)	I	13	122	+	7.3	30	A	1.6	+	0.1	13	+	14
NHC(NHC)	I	13	560	+	5.0	8.7	A	1.6	+	0.1	3	+	20
C compilers													
SUN CC –O	N	10	325	+	26	8	A	2.7	+	0.1	8	+	4
GNU GCC –O	N	10	910	+	97	21	A	2.7	+	0.1	3	+	1

[a] A = Mbytes allocated space; H = Mbytes heap size; R = Mbytes maximum resident set size

Table 8. *Results giving the time (user+system time in seconds) and space (in Mbytes) required for compilation of the Pseudoknot program. The 'pseudoknots' give the relative speed with respect to the execution (not compilation) of the C version.*

tions that generate C or Lisp are the slowest compilers. Not only does it take extra time to produce and parse the C, but C compilers have particular difficulty compiling code that contains large numbers of floating-point constants. The worst case example is the Stoffel compiler. It takes 216 seconds to generate the C code and more than 2 hours to compile the C (on machine 17 c.f. Table 7). Most of this time is spent compiling the function that initialises the data structures containing floating-point numbers. As the bottom two rows of the table show, C compilers also have particular difficulty compiling the hand written C version of the Pseudoknot program due to this phenomenon.

The faster compilers also generally allocate less space. This may be because the slower compilers generally apply more sophisticated (and therefore space-intensive) optimisations.

6.2 Execution time

All programs have been executed a number of times (on machine 10 c.f. Table 7) with different heap sizes to optimise for speed. The results reported in Table 9 show the best execution time, inclusive of garbage collection time. The first column of the table shows the name of the compiler/interpreter (c.f. Table 2). The second column 'route' duplicates the 'route' column from Table 2. The third column states whether floating-point numbers are single- or double-precision. Columns 4 and 5 give the user and system time required to execute the Pseudoknot program. The last column shows the space required, which unless noted otherwise, represents the largest amount of space required by the program, as obtained from `ps -v` under the heading SIZE.

The product moment correlation coefficient calculated from all compilation speeds (as reported in pseudoknots in Table 8) and execution times (as reported in seconds in Table 9) is 0.70. This shows that there is a strong correlation between compilation time and execution speed: the longer it takes to compile, the faster the execution will be. Only the Clean implementation offers both fast compilation and fast execution. The set of Caml compilers offers a particularly interesting spectrum: Caml Gallium is a slow compiler which produces fast code; Caml Light compiles quickly, but is relatively slow; and Camloo is intermediate between the two.

The Epic-C code generator was designed to allow selected, individual functions to be compiled, thus providing an almost continuous spectrum of possibilities from fully interpreted to fully compiled code. The present facilities for compiling code provide little improvement over interpreted code at the cost of huge compilation times. The reason is that the C code faithfully mimics each interpreter step without optimizations, such as the use of local variables or loops. This results in C functions which behave identical to their interpreted counterparts. As much as 90% of the speedup of Epic-C with respect to epic was achieved by compiling 6 of the 170 functions occurring in the Epic version of Pseudoknot.

For the compiled systems there is a very rough relationship between execution speed and heap usage: faster implementations use less heap. There does not, however, seem to be any correlation between non-strictness and heap usage.

6.2.1 Summary

Overall, the (eager) Sisal compiler achieved the best performance. The next best implementation is the (lazy) Glasgow Haskell compiler for a heavily optimised version of the program. The next group of compilers are for Lisp, Scheme and SML, which generally yield very similar performance. An outlier is the Bigloo optimising Scheme compiler, whose performance is more comparable to most of the non-strict implementations (Chalmers, FAST, ID, Stoffel, Yale, and Glasgow Haskell on less optimised code), which form the next obvious group.

Unsurprisingly, perhaps, the interpretive systems yield the worst performance. The interpreters for Caml Light, Epic, NHC, and Trafola (which compile to an intermediate byte-code, which is then interpreted) are, however, significantly faster than their conventional brethren, Gofer, RUFLI and Miranda (which interpret a representation that is closer to the program than a byte-code). Interpreters for strict languages (Caml Light, Epic) do seem on the whole to be faster than interpreters for non-strict languages (NHC, Gofer, RUFLI, Miranda).

compiler	route	float	time(s) user	+	sys	space Mb	A/H[a]
Compiled via another high level language (C or Lisp)							
Glasgow	C	single	3.9	+	0.2	1	A
Opal	C	single	4.7	+	0.5	0.8	A
CeML	C	single	8.7	+	0.6	2	A
FAST	C	single	11.0	+	0.5	1	A
Yale	L	single	11.9	+	7.2	14	H
Epic-C	C	single	43.9	+	2.9	23	A
Sisal	C	double	3.7	+	0.2	0.7	A
Gambit	C	double	6.2	+	0.7	4.4	A
Camloo	S+C	double	11.2	+	1.5	4.9	A
ID	C	double	11.6	+	2.9	14	A
Bigloo	C	double	11.7	+	2.4	4.9	A
ΓCMC	C	double	14.7	+	1.1	22	A
Stoffel	C	double	26.6	+	2.1	5.6	A
Erlang BEAM	C	double	31.8	+	4.5	11	A
Compiled into native code							
CMU CL	N	single	5.8	+	3.3	14	H
LML(OP–TIM)	N	single	7.7	+	0.3	1.2	A
Chalmers	N	single	12.1	+	1.0	3	A
LML Chalmers	N	single	12.5	+	0.4	2.1	A
Caml Gallium	N	double	5.1	+	0.5	0.3	A
Clean	N	double	5.1	+	0.8	2.5	A
MLWorks	N	double	6.3	+	0.1	0.3	A
SML/NJ	N	double	6.9	+	1.2	2.6	A
Facile	N	double	15.5	+	4.3	7.9	A
RUFL	N	double	87	+	2.8	3	A
Interpreted							
Epic	I	single	56	+	2.8	21	A
Trafola	I	single	124	+	6.3	10.7	A
NHC	I	single	176	+	5.7	2.6	A
Gofer	I	single	370	+	12.0	3	A
Caml Light	I	double	52	+	7.4	0.3	A
RUFLI	I	double	529	+	13.0	4	A
Miranda	I	double	1156	+	34.0	13	A
C compilers							
GNU GCC	C	single	2.4	+	0.1	0.3	A
GNU GCC	C	double	2.7	+	0.1	0.3	A

[a] A = Mbytes allocated space; H = Mbytes heap size

Table 9. *The execution times (user+system time in seconds) and space (MB) of Pseudoknot as measured on platform 10.*

6.3 Analysis of Performance Results

Apart from the issues already discussed, such as floating-point precision, many language and implementation design issues clearly affect performance. This section attempts to isolate the most important of those issues.

6.3.1 Higher-Orderness

It is commonly believed that support for higher-order functions imposes some performance penalty, and the fact that the fastest language system (Sisal) is first-order may therefore be significant. Unfortunately, the other first-order implementations (Erlang and Epic) yield relatively poor performance. Sisal is also the only monomorphic language studied, and polymorphism is known to exact some performance penalty, so results here must be inconclusive.

6.3.2 Non-Strictness

As the Glasgow Haskell compiler shows, if the compiler can exploit strictness at the right points, the presence of lazy evaluation need not be a hindrance to high performance. This implementation is actually faster than most of the strict implementations.

Generally, however, non-strict compilers do not achieve this level of performance, typically offering only around 75% of the performance of eager implementations such as SML/NJ or Gambit, or 50% of the performance of CMU Common Lisp, and only after the exploitation of strictness through unboxing and similar optimisations. Without these features, on the basis of the Glasgow results, performance can be estimated as just under a quarter of the typical performance of a compiler for an eager language. For these compilers and this application, support for laziness therefore costs directly a factor of 3, with a further 50% probably attributable to the use of different implementation techniques for predefined functions etc., which are needed to allow for the possibility of laziness.

The difference between the Yale Haskell and Common Lisp results is due partly to use of tagged versus untagged arrays, and partly to the overhead of lazy lists in Haskell. These were the only significant differences between the hand-written Common Lisp code and the Lisp code produced by the Yale Haskell compiler. The Haskell code generator could be extended to use untagged arrays for homogeneous floating-point tuple types as well, but this has not yet been implemented.

The LML(OP-TIM) compiler generates faster code than the corresponding Chalmers LML compiler because if a case alternative unpacks strict arguments, LML(OP-TIM) takes into account that the unpacked values are evaluated (in Weak Head Normal Form).

6.3.3 Concurrency/Parallelism Support

Several of the compilers benchmarked here include support for concurrency or parallelism. In some cases (e.g. Facile, Glasgow Haskell), this support does not affect the normal sequential execution time. In other cases (e.g. ID, Gambit and Erlang BEAM) it is not possible to entirely eliminate the overhead of parallelism.

The low performance recorded by the Erlang BEAM compiler reflects the fact that Erlang is a programming language primarily intended for designing robust, concurrent real-time systems. Firstly, a low priority has been placed on floating-point performance. Secondly, to support concurrent execution requires the implementation of a scheduling mechanism and a notion of time. Together, these add some appreciable overhead to the Erlang BEAM execution.

6.3.4 Native Code Generation

It is interesting that several of the compilers that generate fast code compile through C rather than being native compilers. Clearly, it is possible to compile efficient code without generating assembler directly. Space usage for these compilers is also generally low: the compilers have clearly optimised both for time and space.

6.3.5 Language Design

Of the languages studied, Sisal is the only one that was specifically designed for 'numeric' rather than 'symbolic' computations, and clearly the design works well for this application. Floating-point performance has traditionally taken second-place in functional language implementations, so we may hope that these results spur other compiler writers to attempt to duplicate the Sisal results.

7 Conclusions

Over 25 compilers for both lazy and strict functional languages have been benchmarked using a single floating-point intensive program. The results given here compare compilation time and execution time for each of the compilers against the same program implemented in C. Compilation time is measured in terms of 'pseudoknots', which are defined in terms of the execution time of the benchmark program. The execution times of all compiled programs are reported in seconds as measured on a single machine.

Benchmarking a single program can lead to results which cannot easily be generalised. Special care has been taken to make the comparison as fair as possible: the Pseudoknot program is not an essentially lazy program; the different implementations use the same algorithm; all of the binaries were timed on one and the same machine.

The effort expended by individual teams translating the original Scheme program, and subsequently optimising their performance varied considerably. This is the result of a deliberate choice on the part of the teams carrying out the experiments. Firstly, this variability gives some compilers an advantage (but not an unfair advantage). Secondly, the aim of the Pseudoknot benchmark is specifically to get the best possible performance from each of the implementations (using the guidelines discussed in Section 5). There is a wide variability in the kind and level of optimisation offered by each compiler. The programming efforts required to make these optimisations effective will thus be as varied as the offerings of the compilers themselves. This should be kept in mind when interpreting our results.

Turning to the benchmark itself, we observe that the C version of the program spends 25% of its time in the C library trigonometric and square root routines. This represents the core of the application: the remaining work is 'overhead' that should be minimised by a good implementation. While this pattern may not generally hold for scientific applications, the program is still useful as a benchmark, since the 'real' work it does (the trigonometric and floating-point calculations) is so clearly identifiable. Not all the benchmark implementations are capable of realising this, but some implementations do extremely well.

Because the benchmark is so floating-point intensive, implementations that used an unboxed floating-point representation had a significant advantage over those that did not. Implementations that were capable of exploiting single-precision (32-bit) floating-point have some additional advantage, though this is significant only for the faster implementations, where a greater proportion of execution time is spent on floating-point operations.

To achieve good performance from lazy implementations, it proved necessary to apply strictness annotations to certain commonly-used data structures. When appropriate strictness annotations are used, there is no clear distinction between the runtime performance of eager and lazy implementations, and in some cases the performance approaches that of C.

Inserting these strictness annotations correctly can be a fine art, as demonstrated by the efforts of the Glasgow team. While the behaviour of the Pseudoknot program is not sensitive to incorrectly placed strictness annotations, in general lazy functional programs are not so well behaved in this respect, and considerable effort might be expended introducing annotations without changing the termination properties of a program. To make lazy functional languages more useful than they are now, clearly more effort should go into providing users with simple to use and effective means of analysing and improving the performance of their programs.

The benchmark proved to stress compilers more than expected: the compilation times for most compiled implementations (including the two C compilers) was surprisingly high. Generating C as intermediate code, however, does not necessarily make the compiler slow, as demonstrated by the performance of the Bigloo and Camloo compilers. However, generating *fast* C does often lead to high compilation times.

The Pseudoknot benchmark represents a collaborative effort of an unprecedented scale in the functional programming community. This has had a positive influence on the work that is taking place in that community. Firstly, researchers learn of the techniques applied by their co-authors in a more direct way than via the literature. Secondly, researchers are more strongly motivated to apply new techniques because of the competitive element. Thirdly, using a common benchmark always points at weaknesses in systems, that were either known and put aside for later, or uncovered by the benchmarking effort. The Pseudoknot benchmark has been the trigger to improve a number of implementations. Finally, researchers working on implementations of the different language families are brought closer together, so that the functional programming community as a whole may emerge stronger.

Acknowledgements

The comments of the anonymous referees were very useful.

Mark Jones produced the Gofer version of the Pseudoknot program.

Will Partain and Jim Mattson performed many of the experiments reported here for the Glasgow Haskell compiler. The work at Glasgow is supported by an SOED Research Fellowship from the Royal Society of Edinburgh, and by the EPSRC AQUA and PARADE grants.

The work at Nijmegen is supported by STW (Stichting voor de Technische Wetenschappen, The Netherlands).

Jon Mountjoy performed some of the experiments for the RUFL implementation.

The ID version of the Pseudoknot program was the result of a group effort. Credit goes to Jamey Hicks, R. Paul Johnson, Shail Aditya, Yonald Chery and Andy Shaw.

Zhong Shao made several important changes to the SML/NJ implementation, and Andrew Appel and David MacQueen provided general support in the use of this system. David Tarditi performed several of the SML/NJ experiments.

John T. Feo and Scott Denton of Lawrence Livermore National Laboratory collaborated on the Sisal version of the Pseudoknot program.

The ΓCMC version of Pseudoknot was the result of team work with Genésio Cruz Neto and Ricardo Lima. The ΓCMC group is supported by CNPq. (Brazilian Government) grants 40.9110/88-4, 46.0782/89.4, and 80.4520/88-7.

Marc Feeley was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

M. Alt, C. Fecht, C. Ferdinand, and R. Wilhelm. The Trafola-S subsystem. In B. Hoffmann and B. Krieg-Brückner, editors, *Program development by specification and transformation, LNCS 680*, pages 539–576. Springer-Verlag, Berlin, May 1993.

A. W. Appel. *Compiling with Continuations.* Cambridge Univ. Press, Cambridge, England, 1992.

J. Armstrong, M. Williams, and R. Virding. *Concurrent programming in Erlang.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

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.

L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The computer journal*, 32(2):127–141, Apr 1989.

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.

M. Beemster. The lazy functional intermediate language Stoffel. Technical report CS-92-16, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1992.

M. Beemster. Optimizing transformations for a lazy functional language. In W.-J. Withagen, editor, *7th Computer systems*, pages 17–40, Eindhoven, The Netherlands, Nov 1993. Eindhoven Univ. of Technology.

J. A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification,*. The ACM Press in co-operation with Addison-Wesley, ACM Press Frontier Series, 1989.

D. C. Cann. The optimizing SISAL compiler: version 12.0. Manual UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, California, Apr 1992.

D. C. Cann. Retire FORTRAN? a debate rekindled. *CACM*, 35(8):81–89, Aug 1992.

E. Chailloux. An efficient way of compiling ML to C. In P. Lee, editor, *ACM SIGPLAN Workshop on ML and its Applications*, pages 37–51, San Francisco, California, Jun 1992. School of Comp. Sci, Carnegie Mellon Univ., Pittsburg, Pennsylvania, Technical report CMU-CS-93-105.

K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and implementation of an algebraic programming language. In J. Gutknecht, editor, *Programming Languages and System Architectures, LNCS 782*, pages 228–244, Zurich, Switzerland, Mar 1994. Springer-Verlag, Berlin.

A. Diwan, D. Tarditi, and E. Moss. Memory subsystem performance of programs with copying garbage collection. In *21st Principles of programming languages*, pages 1–14, Portland, Oregon, Jan 1994. ACM, New York.

M. Feeley and J. S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Lisp and functional programming*, pages 119–130, Nice, France, Jul 1990. ACM, New York.

M. Feeley, M. Turcotte, and G. Lapalme. Using Multilisp for solving constraint satisfaction problems: an application to nucleic acid 3D structure determination. *Lisp and symbolic computation*, 7(2/3):231–246, 1994.

R. Giegerich and R. J. M. Hughes. Functional programming in the real world. Dagstuhl seminar report 89, IBFI GmbH, Schloss Dagstuhl, D-66687 Wadern, Germany, May 1994.

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.

K. Gopinath and J. L. Hennesy. Copy elimination in functional languages. In *16th Principles of programming languages*, pages 303–314, Austin, Texas, Jan 1989. ACM, New York.

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.

R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM transactions on programming languages and systems*, 7(4):501–538, Oct 1985.

Harlequin. *MLWorks draft documentation*. Harlequin Ltd, Cambridge, England, 1994.

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.

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.

B. Hausman. Turbo erlang: Approaching the speed of C. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, Boston/Dordrecht/London, Mar 1994.

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.

M. P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Dept. of Comp. Sci, Yale Univ., New haven, Connecticut, May 1994.

B. W. Kernighan and D. W. Ritchie. *The C programming language - ANSI C*. Prentice Hall, Englewood Cliffs, New Jersey, second edition edition, 1988.

X. Leroy. Unboxed objects and polymorphic typing. In *19th Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, Jan 1992. ACM, New York.

X. Leroy. *The Caml Light system, release 0.61*. Software and documentation distributed by anonymous FTP on `ftp.inria.fr`, 1993.

R. D. Lins. Categorical Multi-Combinators. In G. Kahn, editor, *3rd Functional programming languages and computer architecture, LNCS 274*, pages 60–79, Portland, Oregon, Sep 1987. Springer-Verlag, Berlin.

R. D. Lins and B. O. Lira. ΓCMC: A novel way of implementing functional languages. *J. Programming Languages*, 1(1):19–39, Mar 1993.

R. A. MacLachlan. CMU common Lisp user's manual. Technical report CMU-CS-92-161, School of Comp. Sci, Carnegie Mellon Univ., Jul 1992.

J. R. McGraw, S. K. Skedzielewski, S. Allan, R. Oldehoeft, J. R. W. Glauert, C. Kirkham, B. Noyce, and R. Thomas. Sisal: Streams and iteration in a single assignment language. Language reference manual version 1.2 M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, California, Mar 1985.

R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

R. S. Nikhil. ID version 90.1 reference manual. Computation Structures Group Memo 284-2, Laboratory for Comp. Sci, MIT, Cambridge Massachusetts, Jul 1991.

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.

S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 636–666, Cambridge, Massachusetts, Sep 1991. Springer-Verlag, Berlin.

M. J. Plasmeijer and M. C. J. D. van Eekelen. *Concurrent Clean - version 1.0 - Language Reference Manual, draft version*. Dept. of Comp. Sci, Univ. of Nijmegen, The Netherlands, Jun 1994.

J. E. Ranelletti. *Graph transformation algorithms for array memory memory optimization in applicative languages.* PhD thesis, Comp. Sci. Dept, Univ. of California at Davis, California, Nov 1987.

J. A. Rees and W. Clinger. *Revised⁴ Report on the Algorithmic Language Scheme.* MIT, Cambridge, Massachusetts, Nov 1991.

N. Röjemo. Highlights from nhc – a space-efficient Haskell compiler. In *6th Functional programming languages and computer architecture*, pages 282–292, La Jolla, California, Jun 1995. ACM, New York.

W. Schulte and W. Grieskamp. Generating efficient portable code for a strict applicative language. In J. Darlington and R. Dietrich, editors, *Phoenix Seminar and Workshop on Declarative Programming*, pages 239–252, Sasbachwalden, West Germany, Nov 1991. Springer-Verlag, Berlin.

M. Serrano. Bigloo user's manual. Technical report 0169, INRIA-Rocquencourt, France, Dec 1994.

M. Serrano and P. Weis. $1 + 1 = 1$: an optimizing Caml compiler. In *ACM-SIGPLAN Workshop on ML and its applications*, pages 101–111. Research report 2265, INRIA Rocquencourt, France, Jun 1994.

Z. Shao. *Compiling Standard ML for Efficient Execution on Modern Machines.* PhD thesis, Princeton Univ, Princeton, New Jersey, Nov 1994.

S. Smetsers, E. G. J. M. H. Nöcker, J. van Groningen, and M. J. Plasmeijer. Generating efficient code for lazy functional languages. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 592–617, Cambridge, Massachusetts, Sep 1991. Springer-Verlag, Berlin.

G. L. Steele Jr. *Common Lisp the Language.* Digital Press, Bedford, second edition, 1990.

S. Thomas. *The Pragmatics of Closure Reduction.* PhD thesis, University of Kent at Canterbury, Canterbury, UK, 1993.

S. Thomas. The OP-TIM – a better PG-TIM. Technical report NOTTCS-TR-95-7, Dept. of Comp. Sci., Univ. of Nottingham, England, 1995.

S. Thompson. Laws in Miranda. In *Lisp and functional programming*, pages 1–12, Cambridge, Massachusetts, Aug 1986. ACM, New York.

B. Thomsen, L. Leth, S. Prasad, T.-S. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile antigua release – programming guide. Technical report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany (The reference manual and license agreement are available by anonymous ftp from ftp.ecrc.de.), 1993.

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.

D. A. Turner. *Miranda system manual.* Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1 1XP, England, Apr 1990.

P. L. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

H. R. Walters and J. F. Th. Kamperman. Epic: Implementing language processors by equational programs. Technical report in preparation, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.

P. Weis and X. Leroy. *Le langage Caml.* InterÉditions, 1993.

E. P. Wentworth. Code generation for a lazy functional language. Technical report 91/19, Dept. of Comp. Sci, Rhodes Univ., Dec 1991.

E. P. Wentworth. RUFL reference manual. Technical report 92/1, Dept. of Comp. Sci, Rhodes Univ., Jan 1992.