Comparative Cross-Platform Performance Results from a Parallelizing SML Compiler *

Norman Scaife[1], Greg Michaelson[2], and Susumu Horiguchi[1]

[1] Japan Advanced Institute for Science and Technology,
1/8 Asahidai, Tatsunokuchi, Nomigun, Ishikawa, 923-1211
{norman,hori}@jaist.ac.jp
[2] Department of Computing and Electrical Engineering
Heriot-Watt University, Edinburgh EH14 4AS, Scotland
greg@cee.hw.ac.uk

Abstract. We have developed a compiler for Standard ML which allows instantiation of a fixed set of higher order functions with equivalent parallel algorithmic skeletons written in C and MPI. The compiler is intended to be both portable and semi-automatic. Here we discuss the performance of the code generated by the compiler, for five exemplar programs on six MIMD parallel machines. Results suggest acceptable performance and consistent cross-platform behaviour.

1 Compiler overview

We have developed a parallelising compiler [10] for a pure functional subset of Standard ML in which the sole source of parallelism is indicated by specific higher order functions (HOFs). These are translated into parallel algorithmic skeletons[3] implemented in C linked with an MPI library [8].

The main objective of this work is to build a compiler for which all parallelism is *implicit*. Our use of algorithmic skeletons allows us to partially achieve this goal. On the one hand the selection and parallel implementation of the higher-order functions is automatic. On the other hand the programmer is constrained to use the particular set of HOFs provided by our compiler.

We have focused on two HOFs; *map* and *fold*:

```
fun map [] = []
  | map f (h::t) = f h::map f t
fun foldr b f [] = b
  | foldr b f (h::t) = f h (foldr b f t)
```

map applies a function `f` to each element of a list and *fold*, shown here in left-to-right form, applies a function `f` "between" elements of a list. These functions

* This work is supported by grant number GR/L42889 from the UK's Engineering and Physical Sciences Research Council (EPSRC) and Postdoctoral fellowship P00778 of the Japan Society for the Promotion of Science (JSPS)

capture paradigmatic programming patterns and prove to have wide applications in programming.

Each function may be realised through a parallel algorithmic skeleton, a template for a generic parallel computation pattern, which may be instantiated with particular argument function instances. Thus, *map* is implemented as a *process farm*, where a *farmer* process sends list elements to *worker* processes, each of which runs the mapped function `f`. The result from each worker is returned to the farmer for reassembly into the final result list. Similarly, *fold* with an associative and commutative function is realised as a divide-and-conquer network, where, at each level, a parent divides its argument list and sends sub-lists to its children. Each child applies the folded function `f` to its sub-list and returns the result to the parent for recombination with `f`.

A novel feature of our compiler is the ability to *arbitrarily* nest these two skeletons to any depth. Nesting of HOFs is common in functional programs and a rich source of potential parallelism. However, exploiting nested parallelism is known to be a difficult problem involving many program analysis and resource allocation issues.

2 Compiler structure

We use the ML Kit [2] as the host compiler for our system. This provides basic compilation support for SML and, being written in SML itself, is well-suited to the type of analysis we wish to perform. The target compiler for our system is Objective Caml [4]. This was chosen because it is a relatively lightweight implementation of ML with low memory requirements and a fully implemented C language interface, enabling development of skeletons in C, linked with MPI libraries.

The structure of the compiler is shown in Figure 1 and is organised as a series of modules. They are controlled by an overarching executive which initialises data and calls the appropriate modules according to a set of control flags. Each module is now briefly considered.

ParseElab reads the input SML files, `file.sml`, and lexically analyses, parses and *elaborates* the SML prototype. The SML definitions [11, 12] define *elaboration* to be the implementation of the *static* semantics which performs type inference and type checking of the program.

Network takes an elaborated syntax tree and extracts a description of the *topography* of recognised HOFs from the program. We define the topography to be the relationship between HOFs in the source code and between skeletons in the executing program. This module uses the type information generated by the elaborator. The network information is saved into a separate file, `file.an` for verification purposes.

Tddf is an optional module which attempts to defunctionalize [1] the entire program. Our run-time system does not have the ability to transmit functional values which prevents the use of some parallel structures without extensive re-

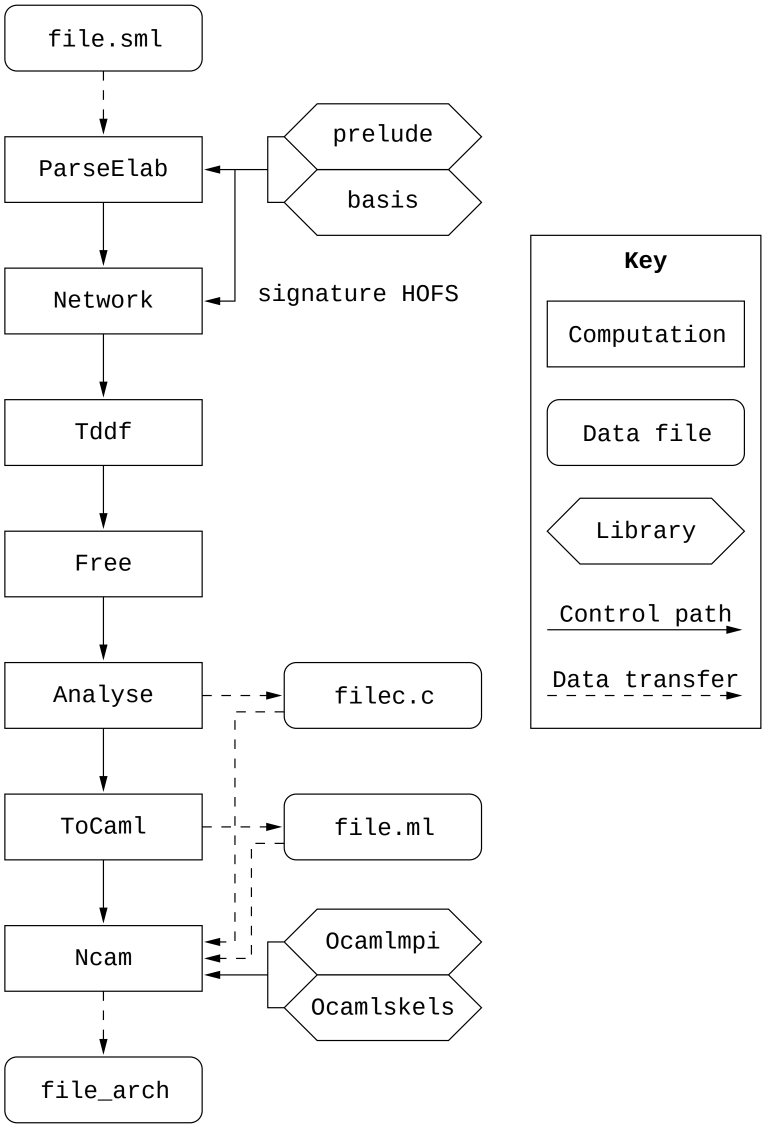


Fig. 1. The compiler modules

structuring of the program by hand. Here, closures are represented by `datatype` values which allows us to transmit partially-applied functions at run-time.

Free performs free-variable analysis and modifies functions by adding their free-variables as arguments. This ensures that free-variables are present at the point of calling a skeleton and are available for transmission at run-time. An attempt is made to limit the transmission of such values to those which will actually be required for remote computation. Note that if defunctionalization is performed then free-variable analysis is not required since free values are bound up in the `datatype` representations.

Analyse is the back-end processing required to transform nominated HOFs into skeleton calls and add code to launch the skeletons at runtime. This involves registering functions with the runtime system, passing free values to the runtime system and also reconstructing the types of skeleton results. A C file, `filec.c`, containing code to construct the network description including the names of registered functions is also output by this phase.

ToCaml translates the `Core` SML into Objective Caml. This is mostly a direct conversion between semantic entities but requires some compilation support to iron out minor differences, particularly between the two type systems. The output from this phase is an Objective Caml file `file.ml` with the prelude code prepended. Other support for the final compilation is also generated at this point, however, including a `Makefile` to perform (local or remote) compilation and running of the executable.

Ncam compiles the output into an executable file. This is not a module in the main compiler but is actually a Perl script which calls the appropriate Caml and C compilers, generating the correct compiler flags and library options. `Ncam` knows about the architectures that the system runs on and is installed on a particular architecture using automatic configuration methods. Thus, if the output from the compiler is not to be run on the local machine it has to be down loaded to the target machine for compilation by the remote `Ncam` installation.

3 Parallel machines

We have ported the compiler to a Network of Ultra SPARC Workstations (NOW), an Intel Celeron Beowulf, an IMB SP2, a Fujitsu AP3000, a Cray T3E and a Sun Enterprise 10000. System characteristics are:

Machine	Processor	Procs[a]	OS
NOW	UltraSPARC IIi 300MHz	10	Solaris 2.8
Beowulf	Intel Celeron 533MHz	32	Linux 2.2.16
IBM RS/6000 SP2	PowerPC 332MHz	68(32)	AIX 3.4
AP3000	UltraSPARC 300MHz	76(16)	Solaris 2.6
Cray T3E-1200E[b]	Alpha 21164 600MHz	128(64)	UNICOS/mk 2.0.5.40
Sun Enterprise 10000	UltraSPARC2 250MHz	32	Solaris 2.7

[a] Numbers in brackets are the available processors due to scheduling.
[b] Compiled into *bytecode* only.

The communications setup for each machine is as follows:

Machine	Network	MPI
NOW	Ethernet	LAM 6.3.2
Beowulf	Ethernet	LAM 6.3.2
IBM RS/6000 SP2	GigaSwitch 150MB/s	Native
AP3000	AP-Net 200Mb/s	Fujitsu MPIAP
Cray T3E-1200E	1.2GB/s	Native
Sun Enterprise 10000	GigaPlane-XB[a]	MPICH 1.2.1

[a] Shared memory

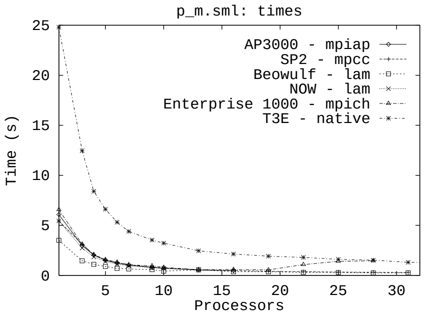
Note that several implementations of MPI are available for each machine, for instance the SP2 can use LAM, MPICH or IBM's own implementation of MPI. This is also true for the AP3000 but only Cray's native MPI library was available for the T3E. The native MPI libraries were preferred for each machine. MPICH has not been installed on the Beowulf so LAM was also chosen for the NOW machine for consistency. MPICH was used for the Sun Enterprise since it is implemented for shared memory.

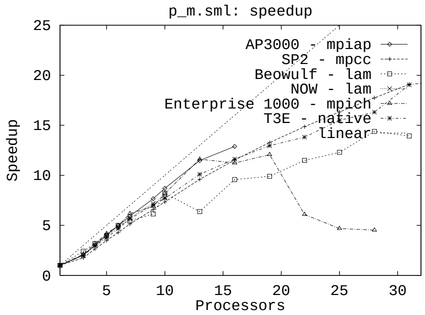
4 Exemplar programs

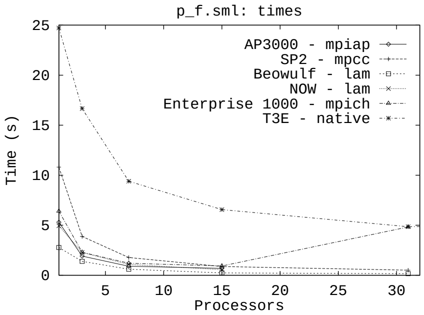
The following sections present results for the bare `map` and `fold` skeletons, and for the four exemplars, on the six systems. None of the exemplars requires defunctionalization so free-value analysis was used in all cases. The runtimes are the average of three executions of the program. Results are not available for the T3E for some exemplars due to a problem with the runtime system.

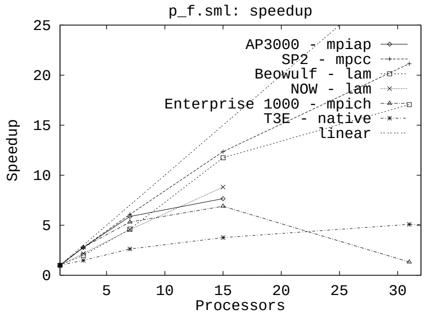
4.1 Bare skeletons

The first test is based on single, non-nested skeletons, originally intended to aid in developing parallel performance models. `p_m.sml` and `p_f.sml` implement a `pmap` and a `pfold` call, respectively. The computation is a delay loop, which can be set at runtime and communications includes padding to set the communications size. The runtimes and speedups for conditions under which a steady-state has been reached, and where neither communication nor computation dominates the execution, are as follows:









The performance results for the AP3000, Beowulf, SP2 and NOW machines are remarkably similar. The T3E runtimes are much slower due to the bytecode execution but otherwise show the same trends. The Enterprise machine's performance tails off dramatically after about 10-15 processors but this is probably due to the machine usage - the runs were not performed on a quiescent machine.

More importantly, both skeletons give useful speedup on all of the given architectures. This gives confidence in applying these skeletons in a new application since the cost of expressing an algorithm using them is counter-balanced by the known performance benefits of the skeletons.

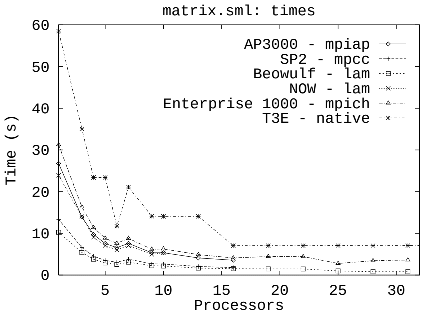
4.2 Matrix Multiplication

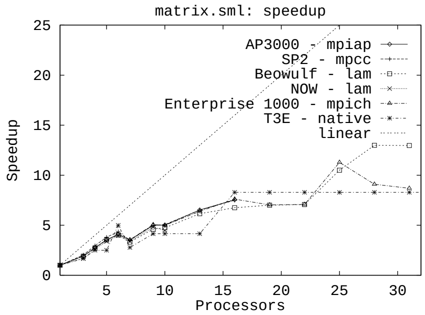
This program [10] calculates the product of two arbitrary-length integer matrices. It has also been used as an exemplar for Hamdan's EKTRAN skeleton-based system [6]. The structure is a nested `map` where the two skeletons are very closely coupled:

```
fun inner Arow Bcol = dotprod Arow Bcol
fun outer BT Arow = map (inner Arow) BT
fun mmult A B = map (outer (transpose B)) A
```

The `mmult` function multiplies two matrices by mapping the matrix-vector operation `outer` over the rows of matrix `A`. This function, in turn, maps the `dotprod` vector-vector operation over the columns of matrix `B`. Note that the entire matrix `B` is passed as an argument to the intermediate skeletons resulting in transmission at runtime. There are methods which mitigate this such as Gentleman's algorithm [5] but these are not expressible using a functional `map`. The performance of the nested `map` implementation is shown.

These runs are for a single non-nested map up to 6 processors. From 7 processors and greater the parallel nested map is implemented. The allocation of processors to skeletons is done at runtime using dummy profile information which assumes that all skeletons perform equal amounts of work. Here we have demonstrated that nested implementation can lead to successful speedup, even if it is not required by this application.



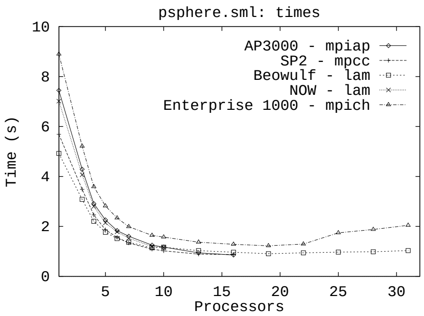


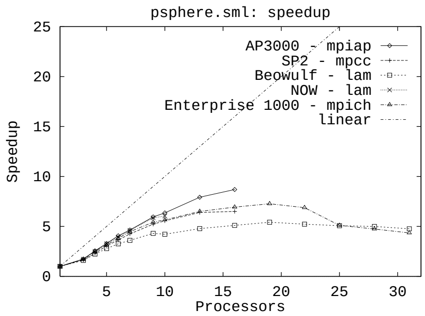
4.3 Ray tracer

This ray tracing algorithm is implemented as a nested `map` structure. The top level code is a map of the tracing function over the pixels of the image. `indxs` generates a list of `(x,y)` indices for the image:

```
val ind = indxs 0 (winsize - 1) 0 (winsize - 1)
fun ff (x,y) =
  ((x,y),tracepixel spheres lights x y firstray scrnx scrny)
val image = map (seqmap ff) ind
```

In practice, the two maps are so closely coupled and the decomposition domain (pixels) is too fine-grained to give speedup from the inner `map`. The results are for a 256×256 image with the single outer `map` implemented in parallel. Thus, `seqmap` is an implementation of `map` which is not parallelised by the compiler.





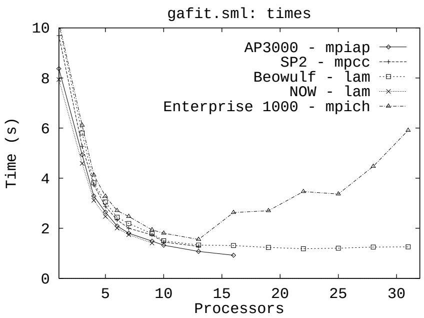
There is considerable tail-off in performance due to the amount of data transmitted to and returned by the remote tasks. Again, however, the program behaves in a similar manner on all the architectures.

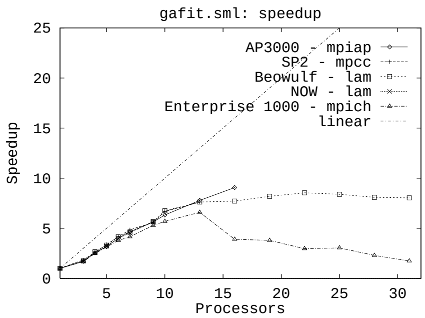
4.4 Genetic algorithm

We have implemented a subset of the libraries for RPL2 (Reproductive Plan Language) for developing parallel genetic algorithms. It is based around a simple flat imperative language with data-parallel iterators for manipulating genome pools. RPL2 was developed originally at the Edinburgh Parallel Computing Centre in a joint project with British Gas [13] investigating pipe-network optimisation problems. It is now maintained and distributed by Quadstone Ltd. In this subset we have implemented a linear equation solver applied to data generated by our profiler [9]. Here we have a triply-nested `map` structure: mapping over genome lists, mapping over genomes and calculating the fitness function itself. The innermost `map` is far too fine-grained for efficient parallel execution:

```
fun nextgeneration gsPop = map pfitness (breed gsPop)
fun run population =
  <until convergence> run (map nextgeneration population)
```

A bug in our current runtime system prevents nested structures from appearing within iterative constructs so the results are presented for a non-nested implementation. There is a lengthy sequential initialization cost which takes about 10 times as long as a single generation (iteration) but convergence takes about 50 to 100 generations so the iteration time dominates. The results presented are for a single iteration.





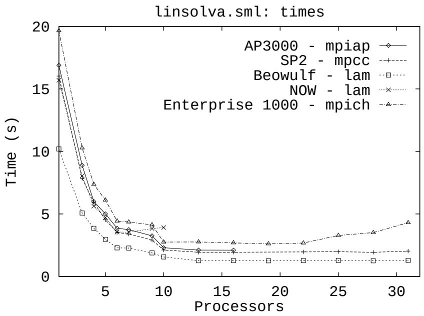
Here we see a typical speedup profile for our `map` skeleton under significant computational loads. There is again a considerable agreement between the various architectures in terms of both runtimes and speedups.

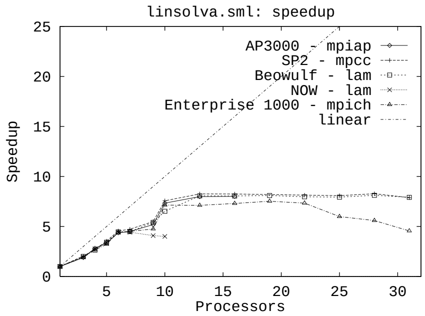
4.5 Linear equation solver

This algorithm finds an exact solution of a linear system of equations of the form $Ax = b$, where all values are in the integer domain, and is discussed in detail by Loidl [7]. The original algorithm was coded in Haskell and has been directly translated into SML. The method is to map the problem onto multiple *homomorphic images*, i.e.. modulo a prime number, and solve in the homomorphic domain. The final solution is generated by applying the Chinese Remainder Algorithm to the homomorphic solutions. Parallelism is a single `map` over the images as nested parallelism within the image was found to be too fine-grained:

```
fun g_hS p = get_homSol aN bN p
val (xList:big_int list list) = map g_hS boundPrimes
```

The function `get_homSol` solves the equation (`aN x = bN`) modulo prime number `p`, the value `boundPrimes` is a list of prime numbers estimated to be the number of primes required for a solution.





For this application there are significant differences between the runtimes but with almost identical speedup profiles. The difference is probably due to the performance of the arbitrary precision number library provided by Objective Caml.

5 Conclusions

We have noted considerable consistency in the behaviour of our exemplars across the platforms. We would expect this for the Beowulf and the NOW as they are linked by similar interconnection technologies. It is interesting that, while the four specialised MIMD architectures are built from varying processors and interconnections, they display comparable processing/communication behaviours as reflected by speedup and comparable absolute processing times. This may reflect a long term convergence of computer technology.

The software components that make up the compiler, and the tools used to realise it, also contribute substantially to this consistency. In particular, these results suggest that the GNU C compiler, which generates code for skeletons, and the OCaml back end, which generates code for sequential program constructs, offer excellent consistency of generated code behaviour across platforms. The different MPI implementations also display consistent cross-platform behaviours.

Our compiler achieves respectable performance and speedup across six MIMD systems, confirming the utility of the automatic extraction of parallelism from functional programs by realising higher order functions as algorithmic skeletons. Parallel programming is hard even with deep knowledge of underlying parallel software and hardware architectures, and the automatic parallelisation of imperative programs has been most successful when restricted to regular data parallelism. Algorithmic skeletons offer an appropriate level of granularity for exploring less regular forms of parallelism. The corresponding higher order functions offer a high degree of abstraction to programmers, minimising necessary awareness of low-level parallel implementation concerns. Finally, functional languages provide a succinct, expressive and formally based medium for writing parallel programs.

Work continues on refining the compiler. Our priorities are to:

- identify useful parallelism in higher order functions from profiling. We use sequential rule counts, from an interpreter based on the SML structural operational semantics, to instantiate linear equations relating counts to parallel times. We are currently exploring numerical and genetic approaches to finding stable solutions to the equations;
- add additional algorithmic skeletons to the compiler, in particular for function composition realised as a pipeline;
- benchmark the compiler on exemplars offering multiple sites of useful, deeply nested parallelism.

References

1. J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In *Proceedings of the ACM SIGPLAN ICFP '97*, pages 25–37. ACM, Jun 1997.
2. L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit (Version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.
3. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.
4. G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
5. W. M. Gentleman. Some Complexity Results for Matrix Computations on Parallel Processors. *JACM*, 25:112–115, 1978.
6. M. Hamdan. *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 2000.
7. H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.
8. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994.
9. G. Michaelson and N.Scaife. Parallel functional island model genetic algorithms through nested skeletons. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on the Implementation of Functional Languages*, pages 307–313, Aachen, September 2000. {ISSN 0935-3232}.
10. G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing*, to appear 2000.
11. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
12. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
13. P. Surry. RPL2 Functional Specification. Technical Report EPCC-PAP-RPL2-FS 1.0, University of Edinburgh/British Gas, 1993.