Towards a skeleton based parallelising compiler for SML

Greg Michaelson and Andrew Ireland and Peter King

Department of Computing and Electrical Engineering,
Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS

Abstract. A design for a skeleton based parallelising compiler for a pure functional subset of Standard ML is presented. The compiler will use Structural Operational Semantics based prototype instrumentation to determine communication and processing costs at sites of higher order function use. Useful parallelism will be identified through performance models for equivalent algorithmic skeletons. Parallelism will be optimised through proof plan driven program transformation. The compiler will generate C utilising MPI for inter-process communication.

1 Introduction

1.1 Implicit and explicit functional parallelism

For parallelism to be exploited in programs, it must first be identified. In principle, extracting parallelism which is somehow implicit in programs would relieve programmers from low level implementation considerations to do with communication/processing costs, topology and process placement. Pure functional languages appear particularly promising for implicit parallelism as referential transparency suggests that parallelism may be found at all levels of a program. Alas, exploiting implicit parallelism has proved somewhat challenging.

Early systems for functional parallelism[1,2] were based on combinator graph reduction on variants of data flow architectures. However, simple SK combinators are too fine grain for efficient parallel implementation. While supercombinator lifting[3] coarsens granularity, it can also lead to inefficiencies in implementing algorithm specific process topologies on a general purpose graph reduction architecture. Strictness analysis[3] can further focus parallelism but may be costly.

One alternative is to introduce explicit constructs to identify sites of parallelism. At simplest this may take the form of a 'par' annotation to indicate that the function and argument expressions in a function application may be evaluated in parallel[4]. More elaborate schemes like Caliban[5] enable the design of parallel process topology in a functional style. This approach finds it apogee in coordination languages[6]. However, explicit parallelism requires the programmer to have deep understanding of the implementation implications of their designs. There are still no well enunciated methodologies for parallel programming: ad-hoc development often leads to disappointing performance and time consuming post-hoc tuning.

1.2 Higher order functions and skeletons

Another approach is to constrain sites of parallelism to calls to higher order functions(HOFs) with associated algorithmic skeletons[7]. A skeleton is a general purpose template with known generic characteristics which may be instantiated with a problem specific process. For a parallel skeleton, the characterisation normally consists of a performance model, parameterised on data sizes and processing costs. Thus, given sizes and costs for a specific instance, the utility of skeleton use may be assessed. Considerable research has been carried out into the development and characterisation of parallel skeletons for common HOFs, for example the well known map/farm and fold/divide and conquer correspondences.

A major benefit of HOF/skeleton use is the growing body of associated proof and transformation techniques, best exemplified by the Bird Meertens Formalism[8]. These enable semantics preserving program manipulation to rearrange composed HOFs and hence the underlying skeletons, in search of optimal parallelism.

Note that HOF identified parallelism is not explicit in the sense of necessarily obliging the use of equivalent skeletons. HOF use may indicate potential rather than mandatory parallelism whose utility is assessed by costs in conjunction with performance models. Furthermore, a compiler might lift HOFs from a program for skeleton realisation. Hence, the use of HOFs with skeletons straddles implicit and explicit parallelism.

Now, the choice of supported HOFs affects both ease of programming and determination of useful parallelism. At one end of the metaphorical scale, stringent restrictions on programming constructs and data structures enable complete identification of useful parallelism at compile time. For example, in FiSh[9] the shape of data must always be statically determinate, and HOFs must be shape preserving. This leads to impressive sequential performance and should form a basis for very accurate parallel cost prediction, and mechanical optimisation of parallelism and data distribution. Similarly, NESL[10] and HOPP[11], though far less radical than FiSh, both offer a fixed set of HOFs as sites of parallelism over data of known structure, to enable accurate static cost analysis.

However, such restrictions necessarily cannot address arbitrary algorithmic parallelism. Here, static analysis must be augmented with empirical measurement of sequential code processing representative data. It may be argued plausibly that such prediction is hopelessly tied to the test data and cannot be generalised to arbitrary cases. The same is true for fine tuning of parallel code. So it goes.

This approach is used in several extant systems. For example, Busvine's PUFF[12], which compiles SML to occam2, uses instrumentation to identify useful parallelism in linear recursion. Feldcamp et al[13] investigate parallelism in a variety of HOFs through instrumentation. Both achieve accurate performance prediction, but based on absolute timings on specific sequential implementations, greatly limiting the portability of their systems.

Bratvold's SkelML[14] is a skeleton based parallelising compiler, from a pure functional subset of Standard ML(SML) to occam2. It is based on sequential program instrumentation through Structural Operational Semantics (SOS)[15]. SkelML uses an SOS based interpreter to count rule firings during sequential program evaluation. This provides an implementation independent measure of program behaviour, in particular for HOF argument functions. SkelML supports a range of common HOFs with equivalent algorithmic skeletons. Skeleton performance models are instantiated using the SOS measures to determine useful parallelism. However, SkelML, like PUFF, is oriented specifically to the Meiko Computing Surface architecture, based on T800 transputers.

Our new project, to generate predictably portable C with MPI from SML, draws on much of the work discussed aboved, in particular SkelML. Its salient features are discussed in the following sections.

2 Portable skeleton based compilation

The first stages of any compiler should carry out the usual lexical, syntactic and semantic checks. We assume that appropriate stages are readily available.

At simplest, the final stage of a skeleton based parallelising compiler requires some representation of the original program annotated to indicate which HOFs are to be implemented as skeletons. Suppose a skeleton is a template for a process topology with slots for arguments and for inter-process I/O. We assume that a sequential code generator for the language is available. The skeleton based compiler should:

- instantiate skeletons for nominated HOFs with sequential code for function arguments and for interprocess I/O. The latter may be generated from the type signatures for the HOFs' arguments.
- generate sequential code for the outer layer of the program with calls to the instantiated skeletons[1]

[1] Skeleton instantiation might consist of copying and in-lining, or of true parameter passing.

The central stage must identify useful parallelism in HOF use. Again, we assume that an SOS based interpreter is available to exercise the program with representative data. We now need to derive likely skeleton behaviour from this instrumentation.

For each abstract syntax construct with an associated SOS rule, we know the C that will be generated by the sequential compiler. Thus, we may devise a test suite for abstract syntax construct to determine its actual costs when the equivalent code is run on a specific architecture. This test suite need only be run once on a new architecture for subsequent use to convert SOS rule costs into actual costs.

Suppose a skeleton performance model is parameterised on individual process and inter-process communications costs. Suppose that interprocess communication consists of data traversal, byte stream data transmission and reception, and data reconstruction[2]. Transmission and reception costs may again be determined once through a standard test suite on a new architecture. For shapely types, traversal and reconstruction costs can be determined analytically through type information. For arbitrary types, typical costs may again be assessed through sequential evaluation of traversal/reconstruction code with representative data.

Thus, this stage should:

- run the program on the SOS interpreter
- for each site of HOF use:
  - convert SOS costs to architecture specific costs
  - run the inter-process I/O tests and converts them
  - instantiate the performance model with the costs
  - annotate the skeleton if the parallelism is useful

3 Program transformation

Much of the time, HOF use will not identify fruitful parallelism. Here, the program may be transformed to regroup composed HOFs and skeletons, a technique used in SkelML and Darlington et al's system[16]. Referential transparency ensures that program transformation will also transform instrumentation information uniformly, so reinstrumentation is only required where new intermediate data types and processing functions are introduced. This raises three important areas for investigation.

First of all, it is necessary to identify and prove appropriate transformations for the HOFs the system supports. For each new HOF, a set of likely transformations might be supplied to the system to be proved en masse. Alternatively, the system or user might generate plausible candidate transformations from known paradigms and attempt to prove them directly. Subsequently, if the known transformations are not applicable to a particular program, the system or user might hypothesise appropriate transformations for proof in situ.

Secondly, where HOF use is not exploitable it may be possible to lift instances of known HOFs from programs. For a given program, there may be a number of different implicit HOFs. Thus, it is important to constrain HOF lifting, to concentrate on finding those which will have most effect on parallel behaviour.

Finally, even for simple compositions of HOFs, the number of applicable transformations grows very quickly and it becomes computationally prohibitive to try them all. Once again, program transformations must be constrained, to focus on those which are known to have significant impacts.

We view transformation as manipulating proofs rather than programs and will build upon Bundy's proof planning approach[17] to constructing automated theorem provers. A proof plan is a common pattern of reasoning which defines a family of proofs and may be expressed in a metalogic in which properties of the object-level structure of proofs can be represented. Promising results have already obtained using the proof planning approach in verifying transformations[18]. The dynamic creation of new transformations is a synthesis task which proof plans support well[19].

[2] Traversal may be combined with transmission, and reconstruction with reception.

More generally, Madden[20] has shown how proof plans can be used to control search in optimising programs through proof transformation. Here, the proof plan meta-logic has a significantly smaller search space than for object-level search.

In addition, proof plan based program synthesis may exploit meta-theoretic properties which characterise certain classes of synthesis proof[21]. Proof plans provide an ideal framework for exploiting the intimate connections between the correctness and constraint of HOF lifting. Proof plan representation could also be used to characterise performance criteria which could further constrain the search for synthesis proofs and thus the dynamic creation of transformations.

4 Design summary

Our proposed system is shown in figures 1 to 4.

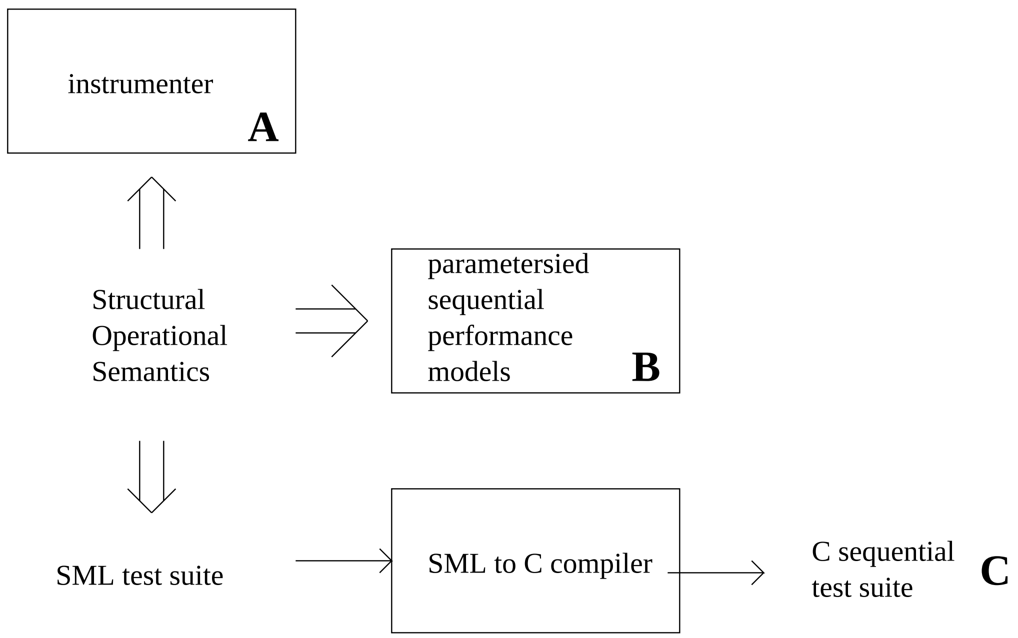

Fig. 1. SOS based instrumenter, sequential models and test suites

Figure 1 shows the role of the SOS for SML. We aim to achieve SOS based SML instrumenter(A) generality by developing parameterised sequential parallel performance models(B) equivalent to the SML SOS rules. A C test suite for the rules(C), generated from an SML test suite, will then provide architecture specific coefficients to instantiate the performance models.

Figure 2 shows the role of HOFs. We will build C test suites(F) for each HOF's skeleton. As for the SOS rules, running the test suites on a new architecture will provide coefficients to instantiate the skeleton models. We will also prove appropriate transformations for the HOFs the system supports(D).

Figure 3 shows the compilation and execution of the test suites (C and F) on a MIMD architecture to provide parameters(G and H) for the sequential and skeleton models.

Figure 4 shows the overall system in use. A SML prototype and test data sets are given to the instrumenter(A). Instrumentation is passed to the analyser which is based on the parameterised sequential(B) and skeleton(E) performance models and instantiated with the architecture specific sequential(G) and skeleton(H) model parameters. Each HOF is called with an argument function, which may itself be a HOF call. At each level, the performance model is provided with communication and processing information from the instrumentation to provide an overall assessment of

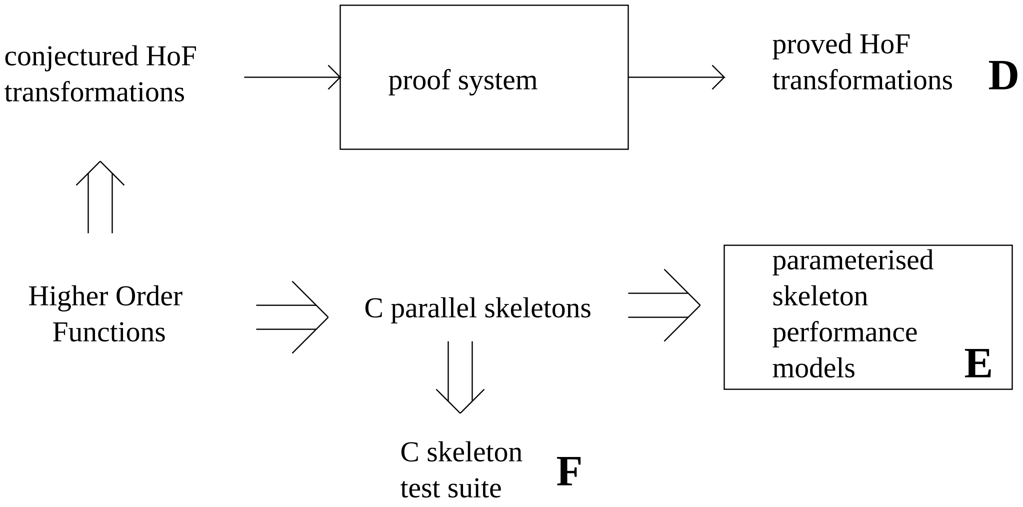

Fig. 2. HOF based transformations, skeletons, models and test suites

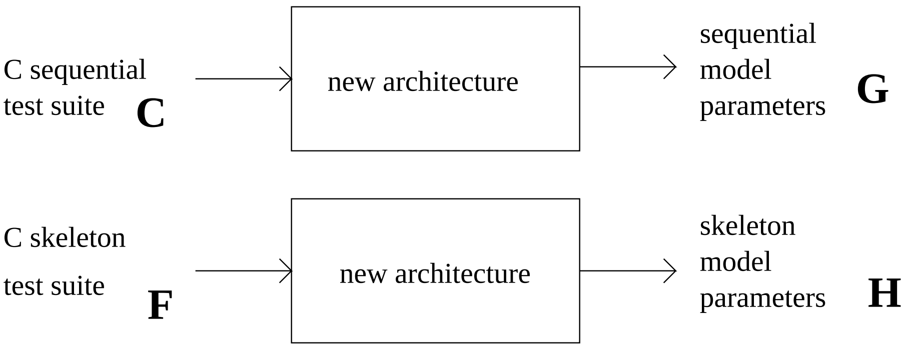

Fig. 3. Model parameters for new architecture

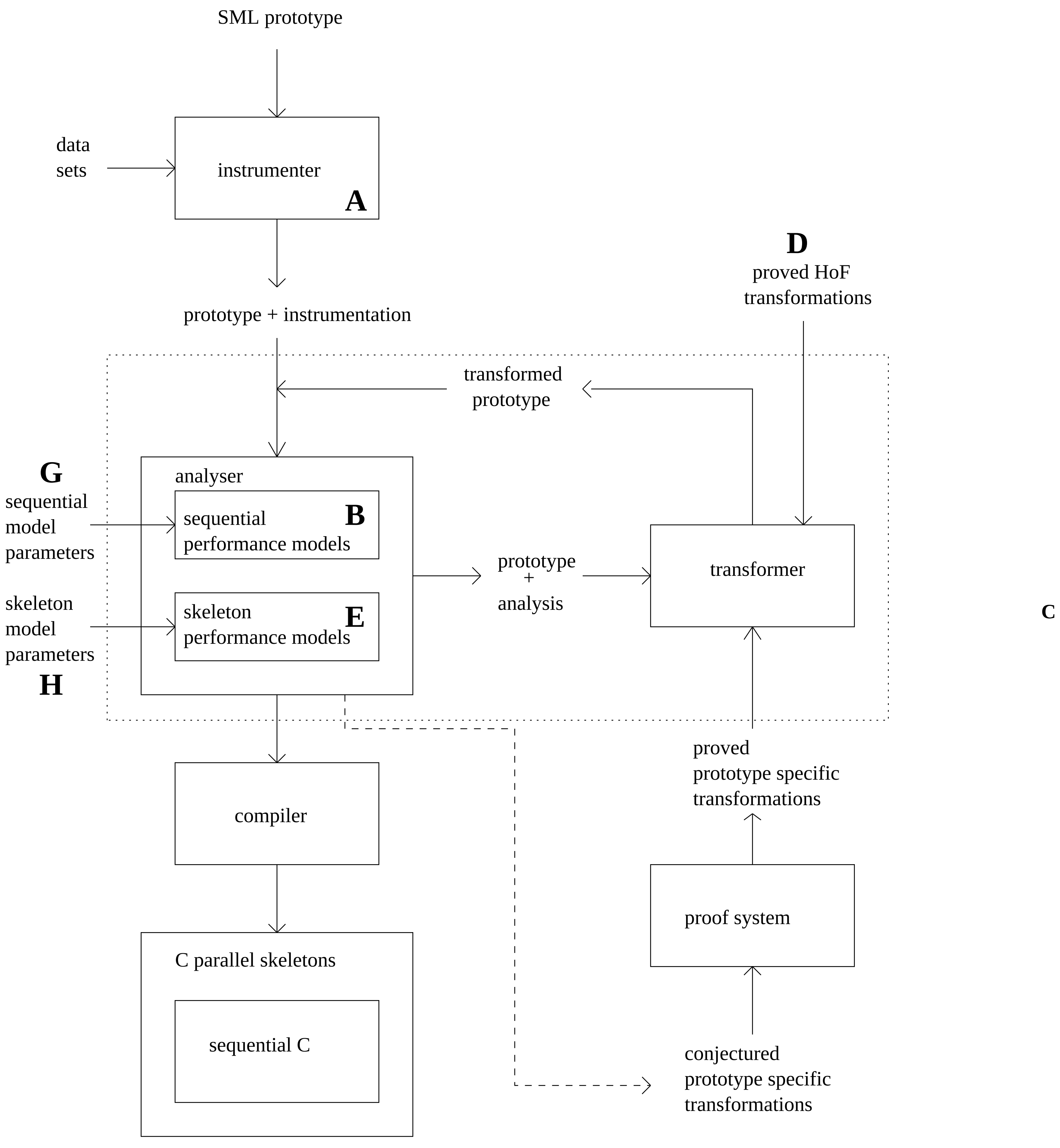

Fig. 4. Prototype instrumentation, transformation and compilation

potential parallelism at that level. For non-HOF arguments, the sequential models are used. If the analyser finds appropriate parallelism then the compiler generates a C program using skeletons for each parallelisable HOF instantiated with sequential code for non-parallelisable arguments. If the analyser cannot find exploitable parallelism then the system will attempt to transform the prototype.

5 Conclusion

We have presented an overview of the design of a parallelising compiler from a pure functional subset of SML to C with MPI. The design is intended to support portability of parallel programs across different MIMD platforms. The initial target platform will be the Fujitsu AP1000 though we will also use a local Meiko Computing Surface during system development.

While this project is undoubtedly optimistic, all the components for the system are mature: their integration should represent a plausible demonstration of the feasibility of automatic optimising parallel compilation.

6 Acknowledgements

This work is funded by EPSRC grant GR/L42889. We are pleased to acknowledge the support of the Imperial College Fujitsu Parallel Computing Research Centre and of Harlequin Ltd.

References

1. J. Darlington and M. Reeve. ALICE — a multiprocessor reduction machine for the parallel evaluation of applicative languages. In *Proceedings of ACM Symposium on Functional Languages and Computer Architectures*, pages 65–76. October 1981.
2. S.L. Peyton-Jones, C. D. Clack, J. Salkild, and M. Hardie. GRIP — a high performance architecture for parallel graph reduction. In G. Kahn, editor, *Proceedings of IFIP Conference on Functional Programming Languages and Computer Architectures*, pages 98–112. Springer Verlag LNCS 274, November 1985.
3. S.L. Peyton-Jones. *The implementation of functional programming languages.* Prentice-Hall, 1987.
4. P.W. Trinder, K. Hammond, J.S. Mattson, and A.S. Partridge. GUM: a portable parallel implementation of haskell. In *Proceedings of Conference on Programming Language Design & Implementation.* Philidelphia, 1996.
5. P. Kelly. *Functional programming for loosely coupled multiprocessors.* Pitman, 1989.
6. J. Darlington, Y-K. Guo, H. W. To, and J. Yang. Functional skeletons for parallel coordination. In S.HAidi, K. Ali, and P. Magnusson, editors, *Proceedings of EuroPar'95*, pages 55–69. Springer-Verlag, August 1995.
7. M. I. Cole. *Algorithmic skeletons: structured management of parallel computation.* Pitman, 1989.
8. R. S. Bird. Introduction to the theory of lists. In *Logic of programming and calculi of discrete design*, volume 36, pages 3–42. NATO ASI Series F, Springer Verlag, 1987.
9. C. B. Jay and P. A. Steckler. The functional imperative: shape! Technical report, Department of Computer Science, University of Technology Sydney, 1997.
10. G. E. Blelloch. NESL: a nested data-parallel language. Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, September 1995.
11. R. Rangaswami. A cost analysis for a higher order parallel programming model. In *PhD thesis.* Department of Computer Science, University of Edinburgh,, February 1996.
12. D. Busvine. Implementing recursive functions as processor farms. *Parallel Computing*, 19:1141–1153, 1993.
13. D. Feldcamp, H. V. Sreekantaswamy, A. Wagner, and S. Chanson. Towards a skeleton based parallel programming environment. In A. Veronis & Y. Paker, editor, *Transputer Research and Applications 5*, pages 104–115. IOS Press, 1992.
14. T. Bratvold. Skeleton-based parallelisation of functional programs. In *PhD thesis.* Dept of Computing & Electrical Engineering, July 1995.
15. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1990.

16. J. Darlington, A. J. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. J. Kelly, K. M. Sephton, and D. W. Sharp. Structured parallel functional programming. In H. Glaser and P. Hartel, editors, *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 31–51. CSTR 91-07, Department of Electronics and Computer Science, University of Southampton, 1991.
17. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk & R. Overbeek, editor, *Proceedings of 9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988.
18. A. Ireland and A. Bundy. Extensions to a generalization critic for inductive proof. In M.A. McRobbie & J.K Slaney, editor, *Proceedings of 13th Conference on Automatic Deduction*, pages 47–61. Springer Lecture Notes in Artificial Intelligence No. 1104, 1996.
19. A. Bundy, A. Smaill, and J. Heskith. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–116. 1990.
20. P. Madden. Automatic program optimization through proof transformation. In D. Kapur, editor, *11th Conference on Automated Deduction*, number 607, pages 446–461. Springer-Verlag Lecture Notes in AI, 1992.
21. J. Heskith, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail recursive programs. In D. Kapur, editor, *11th Conference on Automated Deduction*, number 607, pages 310–324. Springer-Verlag Lecture Notes in AI, June 1992.