Automated Transformation of BMF Programs

Brad Alexander and Andrew Wendelborn*
University of Adelaide, South Australia

November 2003

Abstract

Transformation is key to any program improvement process and a key to successful transformation is the medium in which it takes place. BMF (Bird-Meertens Formalism) is a medium specialised for program improvement via incremental transformation. While much theoretical work exists, there has, to date, been a paucity of actual implementations employing BMF in an automated process of program improvement. In this paper we describe such an implementation, targeted to distributed architectures.

1 Overview

Transformation is the key to any program improvement process. Highly transformable notations pave the way for the application of deep and pervasive program improvement techniques. Functional programming languages are more amenable to transformation than their more traditional imperative counterparts. Moreover, functional programs specify only true dependencies between values, making improvements that reveal and exploit parallelism much easier. Some functional programming notations are more transformable than others. BMF is a functional notation intentionally developed as a medium for transformational program development[1].

1.1 BMF

BMF is a set of theories of various aggregate types. Each theory has a open-ended set of operations over its type and an open-ended set of transformation rules that apply to these operations. Typically, at some level, BMF programs are expressed as a sequence of composed functions of the form:

$$f_n \circ \cdots \circ f_2 \circ f_1$$

where input data flows into the function f_1, whose output flows to f_2 and so on until the output eventually emerges from f_n. A concrete example of a BMF program is:

$$sqrt \circ sqr$$

which uses the square and square-root functions to find the absolute value of some numerical input. Primitive functions that take other functions as parameters (second-order functions) are commonly used to raise operations to the level of an aggregate type. For example, the

*Authors' Address: Department of Computer Science, University of Adelaide, Adelaide 5005, Australia. Phone: +61 8 303-5681, E-mail: brad@cs.adelaide.edu.au, andrew@cs.adelaide.edu.au

Obstacle	**Remedy**
Infinite transformation space	Use goal-oriented transformation.
Case explosion	Use normalising transformations to limit cases appearing in code.
Hard-to-transform recursive definitions	Ban recursion (for now).
Eureka steps	Avoid completely/Put trust in programmer.
No knowledge of run-time data	Perform only statically justifiable improvements.

Table 1: Obstacles to using BMF and our remedies

map function (denoted $*$) applies a given function to each element of an aggregate input type. The function $sqr*$ takes an aggregate containing numbers and returns an aggregate containing the squares of those numbers[1]. Another important function, reduce (denoted $/$) uses a given binary operator to to reduce an aggregate to a single value. For example, the function $+/$ returns the sum of a aggregate containing numbers.

Note that all of the functions above have no explicitly-named input parameters. Programs composed of such functions are said to be in *point-free* form. Point-free programs are very easily transformed using equational rules. One such rule is:

$$f * \circ g* = (f \circ g)*$$

This rule could, for example, be applied to the program: $(sqrt) * \circ (sqr)*$ to produce: $(sqrt \circ sqr)*$ which is semantically the same but more efficient because it produces less intermediate data.

With the right set of rules very impressive, and provably correct, feats of program development can be achieved[2, 3]. Other desirable features of BMF include: naturally parallel implementation of building-blocks such as map and reduce; reversible transformations; the existence of good sequential and distributed cost models[4] which can be used to quickly benchmark programs during the transformation process. Of all programming notations, BMF has perhaps unrivalled potential as medium for safe, transparent, effective, mechanisable, program improvement in both a parallel and sequential context.

1.2 Scope of this work

This work exploits some of the potential of BMF by using it as the primary intermediate form in a compilation process from a simple functional language, Adl[5], to a distributed parallel architecture. To the best of our knowledge, there is no other implementation that uses BMF to the same extent.

For all of its advantages, in using BMF there also are a number of obstacles to overcome. Table 1 summarises some of these and the remedies used in the Adl project. The first four obstacles are typically encountered in automated theorem proving. The last is more germane to program optimisation.

The first obstacle is overcome by defining rules that work, deterministically, toward some goal without recourse to cost analysis. The second obstacle is dealt with by constantly normalising code to narrow the range of cases that appear. The third obstacle, recursive definitions, arises because such definitions, though elegant and pure, throw up barriers to

[1]Any ordering in that aggregate is not affected.

transformation[2]. The fourth obstacle,the presence of *eureka steps*, occurs because some transformations need insight for their derivation/application. Insight is not a mechanical process. The Adl solution is to not derive new steps and to trust the programmer to, indirectly, inform us if an extant eureka step is applicable[3]. The last obstacle is one faced by any static program improvement process and the Adl implementation overcomes it in the usual manner.

Note that some of the compromises above preclude an exhaustive search for the *best* version of a program. Rather the aim of the process is to produce a *better* program.

Next, we very briefly summarise related work. The four sections after that describe the source-language, translation, optimisation, parallelisation/targeting respectively. Finally, we briefly summarise our findings.

2 Related work

Mottl provides a good overview of the issues surrounding automated program transformation in his thesis[7]. There is much work on program development using BMF. Bird provides one of the seminal works[1] and a good followup volume[2]. Skillicorn[8] motivated the application of BMF to parallel computing. Much related work has been done since (see [9] and [10] for some more recent examples). Martin and Nipkow[11] utilised a theorem prover to partially automate BMF derivations. The Adl implementation is more pervasive in its use and requires less assistance in its use than any work we have seen to date.

There are strong structural similarities between the parallel functions produced by the Adl implementation and functional skeletons[12]. BMF shares many features with the point-free intermediate form, FP*, used by Rao and Walinsky in their implementation of EL*[13]. Unlike our work, their work is not strongly focused on transformation within the intermediate form.

3 The source language

Adl is a simple, strict, non-recursive functional language with simple scoping rules, primitive operations over aggregates and constructs for selection and iteration. One-dimensional, nestable arrays, called vectors are well-supported. Aggregate operations such as `map` and `reduce`, the vector-generating function `iota` and vector-indexing are all provided. Aggregate operations on vectors are implicitly parallel. Record-types are supported as tuples through pattern-matching. Concrete types for most entities are inferred by the compiler. Once the few primitive building blocks are understood Adl is a relatively easy and reasonably expressive programming language to work with.

Figure 1 shows a simple Adl program to calculate a convolution over an input vector where every element of the vector is replaced with the sum of the absolute difference between it and every other element of the vector. If, for example, `remote.Adl` is given the input vector $[1, 2, 3, 4, 5]$ its output will be $[10, 7, 6, 7, 10]$.

[2]See Sand's article[6] for a lucid explanation of one reason why this is so.

[3]An example is that Adl trusts a programmer to use an associative binary operator in parallelisable versions of `reduce`.

```
main a: vof int :=
  let
    f x :=
      let
        add(x ,y) := x+y;
        abs x := if x<0 then -x else x endif;
        dist y := abs(x-y)
      in
        reduce(add, 0, map(dist, a))
      endlet
  in
    map(f, a)
  endlet
```

Figure 1: Source code for `remote.Adl`.

$$(\mathit{reduce\_exp} \circ (\mathsf{id}, (\mathit{abs} \circ (\mathsf{id}, \mathit{minus})^{\circ}) * \circ \mathsf{distl} \circ (\mathsf{id}, \pi_1)^{\circ})^{\circ} \circ \mathsf{id}) * \circ \mathsf{distl} \circ (\mathsf{id}, \mathsf{id})^{\circ} \circ \mathsf{id}$$

where

$$\begin{array}{lcl}
\mathit{reduce\_exp} & = & \mathsf{if}(\neq \circ (K_0, \# \circ \pi_2)^{\circ}, \mathit{oplus}/ \circ \mathsf{distl}, K_0 \circ \pi_1) \\
\mathit{oplus} & = & + \circ (\pi_1 \circ \pi_2, \pi_2 \circ \pi_2)^{\circ} \circ (\pi_1 \circ \pi_1, (\pi_2 \circ \pi_1, \pi_2 \circ \pi_2)^{\circ})^{\circ})^{\circ} \\
\mathit{abs} & = & \mathsf{if}(< \circ (\pi_2, K_0)^{\circ}, u- \circ \pi_2, \pi_2) \\
u- & = & \textit{primitive operation (unary minus)} \\
\mathit{minus} & = & - \circ (\pi_2 \circ \pi_1, \pi_2)^{\circ}
\end{array}$$

Figure 2: BMF translation of `remote.Adl`

4 Translation to BMF

The translation process from Adl to BMF is relatively straightforward. The main task is to eliminate explicit parameters and build infrastructure for transporting the values in scope, embedded in a nested tuple, to each function that needs them. Each translated function projects the values it needs from its input tuple by using π_i functions. Figure 2 shows the BMF translation of `remote.Adl`. While space constraints preclude a full description of this program[4] some features to note are: that every entity in the above code is a function; the original `map` ($*$) and `reduce` ($/$) functions are present in the code; there is heavy use of the distl function which distributes the first element of its input tuple over the second element of its input tuple.

The program above is correct but not efficient. Each instance of distl ensures that all values that *could* be accessed in the original program are supplied to every function invocation. This is clearly more than is needed and sets up a large copying overhead.

5 Optimisation

The optimiser is the first stage that directly transforms BMF code. The optimiser works to reduce data transported both within vectors and in tuples. It does this by applying transformations that incrementally move data-narrowing operations, such as vector indexing

[4]see [14] for a complete description of the translation process.

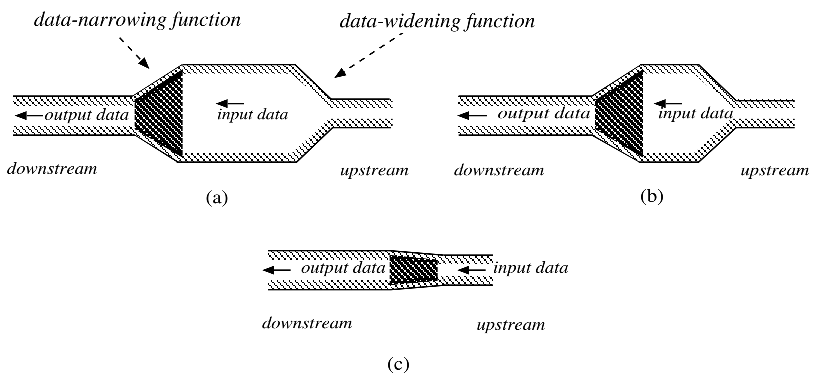

Figure 3: Snapshots of optimisation process. Data flows in the program early (part (a)), later (part(b)), and still later in the optimisation process (part (c)). The shaded areas represent the function that is the current focus of optimisation. Note that in part (c) the data widening and data narrowing functions have eliminated each other resulting in a program with much reduced data flow throughout.

$$(+/_{K_0} \circ (\mathsf{if}(< \circ(-, K_0)^\circ, u - \circ -, -)) * \circ\mathsf{distl}) * \circ\mathsf{zip} \circ (\pi_1, \mathsf{repeat} \circ (\pi_2, \# \circ \pi_1)^\circ)^\circ \circ (\mathsf{id}, \mathsf{id})^\circ$$

Figure 4: Optimised code for `Remote.Adl`

and tuple-projection as close to the start of the program as possible. The rationale behind this process is that the closer to the start of a program a data-narrowing operation is, the less time its large input data exists. As a bonus, often this process brings a data-narrowing operation and a corresponding data-widening operation into contact, causing their mutual destruction, eliminating the need for a large intermediate data structure.

Intuitively, this process can be viewed as a *front* of optimisation sweeping back to the start of the program. Figure 3 provides a conceptual view of this strategy. At every stage of this process, semantics are preserved. All transformations operate locally in the area that is the current focus of optimisation.

The optimisation process, currently, consists of two major passes, one to optimise access to vectors and one for tuples. Heavy use is made of a few sets of normalisation rules whose purpose is to shape code to fit subsequent rules. Most of the optimiser's time is spent applying sets of normalisation rules.

Figure 4 shows code produced by the optimiser for `remote.Adl`. The obvious change is the much shorter code. The other significant change is the removal of all but one instance of the inefficient distl function and this, necessary, remaining instance now distributes a lot less data. The new zip and repeat functions are used to join and copy data respectively.

To assess the extent of improvement we ran both the optimised and unoptimised versions of `remote.Adl` against an instrumented simulator that records both the time and space consumption. The results for a very small input vector of ten elements are shown in figure 5. The time consumption is in the x-axis and the space consumption is in the y-axis. The upper curve is the trace for the unoptimised version. Even with 10 elements, there is a striking difference in the performance translator and optimiser code.

For comparison, we also tested a hand-coded version of `remote.Adl` and its performance

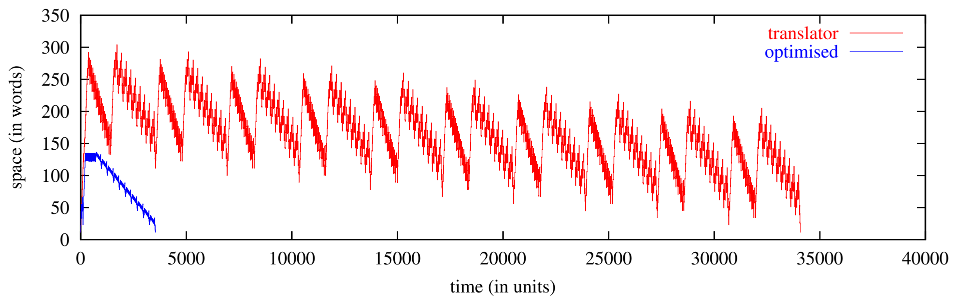

Figure 5: Traces of the sequential execution of various versions of `remote.Adl` when applied to a 10 element vector.

$$\begin{array}{l}
+\!\!+ /^{||} \circ \\
(+/_{K_0} \circ (\mathsf{if}(< \circ(-, K_0)^\circ, u - \circ -, -)) * \circ\mathsf{distl}) * *^{||} \circ \\
(\mathsf{zip}) *^{||} \circ((\pi_1, \mathsf{repeat} \circ (\pi_2, \# \circ \pi_1)^\circ)^\circ) *^{||} \circ \\
\mathsf{zip}^{||} \circ (\pi_1, \mathsf{repeat}^{||} \circ (\pi_2, \# \circ \pi_1)^\circ)^\circ \circ (\mathsf{split}_{32}, \mathsf{id})^\circ
\end{array}$$

Figure 6: Parallelised code for `remote.Adl` targeted to 32 nodes.

curve was almost coincident with that of the optimised code. These results are comparable to results achieved for other applications including, simple stencil algorithms, maximum-segment-sum, matrix addition and matrix transpose. The parallelisation process, and its results are described next.

6 Parallelisation

The same techniques used for optimisation in BMF are also used for parallelisation[5]. Transformation rules are applied to propagate functions with implicit distributed semantics through sequential BMF code. For the Adl implementation, parallelisation rules have been defined and tested. The result of applying these rules, by hand[6], to `remote.Adl` is shown in figure 6. Again, space precludes a thorough description of this code but the salient features are the parallel operations. All of these, with the exception of split are superscripted with $\|$. The numeric subscript to split indicates how many nodes the input vector will be distributed over. This parameter is expected to be supplied to the compiler by the programmer. The end result is a program that is a sequential composition of distributed functions. All communication takes place in bulk before each function starts so, for realistically sized problems, the model is constrained by bandwidth rather than latency.

We currently test our parallel programs on a detailed, highly configurable, simulator of a high-performance parallel cluster[17] with realistic settings for latency and bandwidth[7].

To confirm that the effect of optimisation carried through to parallel code we compared

[5]Roe[15] produced some good early work exemplifying how this could be done.

[6]Very close the time of writing, an automated prototype[16] has been built. Early experiments with `remote.Adl` produce code with a similar level of performance to the hand-parallelised code. The techniques used in this prototype are very similar to those employed in the optimiser.

[7]Bandwidth was rated at $3ns/byte$ and the latency was $5\mu s$. Average instruction execution time was set at $1ns$.

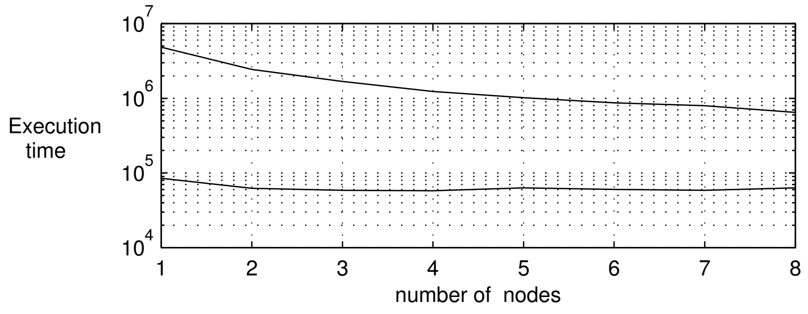

Figure 7: Performance of translator .vs optimiser code of `remote.Adl` on a simulated crossbar with 64 input values.

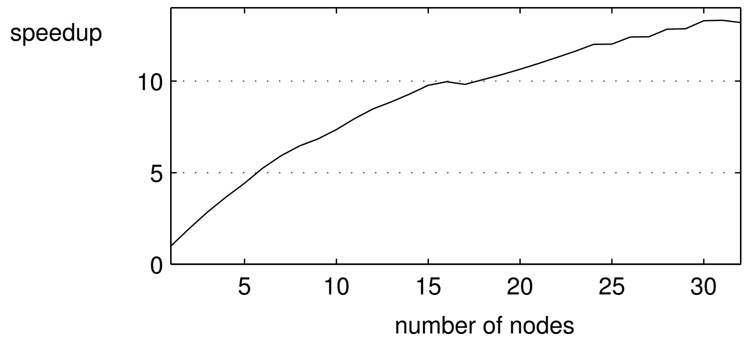

Figure 8: Performance of optimised code with 320 input values.

the performance of parallelised unoptimised code to the code shown in figure 6. The results are shown, for small data, on up to 8 crossbar-connected nodes in figure 7. The orders-of-magnitude difference in the performance of the programs confirms that the effect of optimisation does persist.

To see if reasonable speedups can be obtained with optimised code we ran the optimised program on more nodes of the same machine with more data to get the speedup shown in figure 8. Reasonable speedup is attained. The bumps at 15 and 32 nodes are artifacts of the communication primitives used. The general levelling off the curve is the effect of a decreasing computation to communication ratio as the same sized problem is spread over more processors. For `remote.Adl` the amount of levelling off decreases significantly as input data grows.

6.1 Memory management costs

In order not to overestimate speedup, the experiments above were run with the low on-node memory management costs - in line with those of a final implementation[8]. A straightforward translation[9] of BMF to an imperative language gives each function a copy of its input value and deallocates this value once the output is produced. This scheme, very similar to that

[8]High memory management costs increase the computation to communication ratio which *inflates* speedup estimates.

[9]A prototype implementation of a translator from parallel BMF to C/MPI has been very recently built[18] and early experiments have been run, showing reasonable speedup.

traditionally used by APL interpreters[19](pp 19), is robust but very costly. Fortunately, a reduction of these costs has been very successfully carried out in an implementation of the functional language,SISAL[20] by employing a raft of techniques to replace re-allocation with re-use in the imperative form. A-priori, the task should be no harder for BMF code.

7 Conclusions and Future Work

Early results are promising. A range of small programs have been translated and optimised and parallelised. Speedups are in line with expectations. This demonstrates that program improvement, incorporating optimisation and parallelisation, by automated transformation of point-free BMF is an effective compilation technique. The most novel and interesting aspect of this work thus far has been learning what it is like to compile using BMF. Some observations are:

Correctness is easy: Temporary bugs that produced incorrect code were a relative rarity.

Efficiency is still hard: Temporary bugs that produced inefficient code were common.

Gaps are costly: Stages seemed to work best when they applied a simple transformation pervasively. Any gap in a set of rules tended to limit its re-usability and, worse still, often resulted in the neglected code growing to become a bottleneck in the program.

Cleverness is best avoided: Rules with complicated tests attached, and rules that performed simultaneous tweaking of spatially separated code were the most troublesome. It is better to refashion code to fit a simpler rule than to have a clever rule.

Normalisation works: This whole exercise would be intractable without heavy use of normalisation to put things in predictable places for the benefit of subsequent rules. BMF is extraordinarily amenable to normalisation.

Normalisation is most robust if kept close: The compiler is at it most robust when normalisation is carried out just before the rule that relies upon it. This eliminates the chance of anything happening to the code in the interim.

Future work will include a completion and refinement of the prototypes for parallelisation and code generation. There is also scope for improving the earlier stages by rationalising the BMF instruction set to ease the compilation process. Medium term goals include introducing a richer type system to Adl as a pre-cursor to a more flexible partitioning scheme at the final stage. Finally, the task of producing a better improvement process is open-ended. With sets of normalisation rules to act as interfaces, special and general-purpose stages can continue to be added as needed.

References

[1] R. Bird, Lectures in Constructive Functional Programming, in *Constructive Methods in Computing Science*, Springer-Verlag, 151–216.

[2] R. Bird and O. de Moor, The Algebra of Programming, Prentice-Hall, 1996 .

[3] W.N. Chin and Z. Hu, Towards a Modular Program Derivation via Fusion and Tupling, in *The First ACM SIGPLAN Conference on Generators and Components*, Lecture Notes in Computer Science, Springer-Verlag,Vol 2487:140–155.

[4] D. Skillicorn and W. Cai, A Cost Calculus for Parallel Functional Programming, *Journal of Parallel and Distributed Computing*, 1995, Vol 28(1): 65–83.

[5] B. Alexander, D. Engelhardt, A. Wendelborn, An Overview of the Adl Language Project, in *Proceedings of Conference on High Performance Functional Computing, Denver, Colorado.*, 1995.

[6] D. Sands, Total Correctness by Local Improvement in the Transformation of Functional Programs, *ACM Transactions on Programming Languages and Systems*, 1996, Vol 18(2):175–234.

[7] M. Mottl, Automating Functional Program Transformation, Masters Thesis, 2000, `http://www.ai.univie.ac.at/ markus/msc_thesis/index.html`.

[8] D. Skillicorn, Architecture Independent Parallel Computation, *IEEE COMPUTER*, December 1990, 38–49.

[9] S. Gorlatch, Toward the Formally-Based Design of Message Passing Programs, *IEEE Transactions on Software Engineering*, March 2000, 276–288.

[10] Z. Hu, M. Takeichi, W. N. Chin, Parallelization in Calculational Forms, in *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages.*, 1998.

[11] U. Martin and T. Nipkow, Automating Squiggol, in *Programming Concepts and Methods*, North-Holland, 1990, 233–247.

[12] H. Darlington, Y. K. Guo and Y. Jing, Skeletons for structured parallel composition, in *Proceedings of the 15th ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1995.

[13] P. Rao, C. Walinsky, in *ACM SIGPLAN Notices*, 1993, Vol 28(7):112–118.

[14] B. Alexander, Mapping Adl to the Bird-Meertens Formalism,Technical Report 18, Department of Computer Science, University of Adelaide, 1994, `http://www.cs.adelaide.edu.au/users/brad/bmf.ps`.

[15] P. Roe, Derivation of Efficient Data Parallel Programs, *Seventeenth Australasian Computer Science Conference*, 1994, 621–628.

[16] J. Windows, Automated Parallelisation of code written in the Bird-Meertens Formalism, Honours Thesis, School of Computer Science, University of Adelaide, 2003, `http://www.cs.adelaide.edu.au/users/brad/students/josephw.pdf` .

[17] P. Martinaitis, Simulation and Visualisation of Parallel BMF Code, Honours Thesis, Department of Computer Science, University of Adelaide, 1998, `http://www.cs.adelaide.edu.au/users/brad/students/paulm.pdf` .

[18] D. Philp, Mapping Parallel BMF Constructs to a Parallel Machine, Masters Thesis, School of Computer Science, University of Adelaide, 2003, `http://www.cs.adelaide.edu.au/users/brad/students/deanp.pdf`.

[19] R. Bernecky, APEX - The APL Parallel Executor, Masters Thesis, University of Toronto, 1997, `http://www.snakeisland.com/ms.pdf`

[20] A report on the SISAL Language Project, *Journal of Parallel and Distributed Computing*, Vol 10(4): 349–366.