A Fast Code Generator for Point Free Form

Sean Pettge
Department of Computer Science
University of Adelaide

Supervised by Brad Alexander

This Thesis is submitted in partial fulfillment of the requirements for the Honours Degree in Computer Science

November 7, 2005

Contents

List of Figures

Acknowledgements

First, a big thanks to my supervisor, Brad Alexander, for the help and support throughout the year.
I would also like to thank my fellow honours students, for their friendship and support.
And, of course, I would like to thank my family for their support.

Abstract

Program-optimisation and program-parallelisation are challenging tasks in computer science. These tasks are often made more difficult by the properties of the language used to express program code. Bird-Meertens Formalism (BMF) is a notation designed to ease the task of program transformation by enabling changes to a program through the application of simple rewrite rules. In addition, BMF has many constructs that are readily mapped to parallel architectures.

Unfortunately, where the target architecture consists of distributed processors of conventional design, a naive mapping of BMF code to each conventional processor results in large overheads due to repeated copying of potentially large data arrays, as well as frequent allocation and deallocation of memory. There is a need for automated analysis to make this mapping produce more efficient code.

This project has focused on this automated translation, starting with the construction of a baseline automated translator that uses a simple mapping, which suffers from the problems outlined above. A more advanced translator was then constructed, which focused on moving the evaluation of routing functions - functions which only change the shape of the data, not the data itself - into the compiler, rather than having them in the produced program. This technique has produced improvements in both program execution time and memory usage over the baseline system.

Chapter 1

Introduction

Program optimisation and program parallelisation are two very challenging tasks in the field of Computer Science. These tasks are often made more difficult by the properties of the language used to express the program code. A program written in an imperative or object-oriented language typically contains many complicated dependencies, making it difficult to analyse the flow of data through the program, and to change the way the program executes without changing its meaning.

Functional languages make these processes easier, through reducing the number and types of dependencies in the program. Point free form, a type of functional notation which has no variables, simplifies these processes further, by explicitly linking the producers and consumers of data.

This means that point free form can be easily transformed through the application of simple rewrite rules, which can effect wholesale changes to the structure of a program without changing its meaning. Point free form also contains many structures, such as maps and scans, that are easily mapped onto a parallel architecture.

Unfortunately, a naive mapping of point free form code to a non-distributed architecture results in large overheads due to repeated allocation and copying of large amounts of memory. While point free code can be transformed by hand to produce a more efficient mapping, this is slow, tedious and error-prone. The automation of this transformation will greatly improve the usefulness of point free form.

This automation is also desirable because point free form is a difficult language to program in. This means that its transformational properties are perhaps best exploited when it is used as an intermediate language, with another automated system translating a different language into point free form, optimising it, and then passing on to a system like this one for conversion to executable code.

Chapter 2

Bird-Meertens Formalism

2.1 BMF

Bird-Meertens Formalism (BMF) is a dialect of point free form devised for program calculation. It is based on a functional language FP, proposed by Backus [Bac78], and developed by Bird [Bir87], [BdM96] and Meertens [Mee86]. It consists of a number of constructs which have been chosen for the power of the algebraic rules associated with them and for their programming usefulness.

A summary of the constructs in the language is given in 2.1. Our version of BMF has three base types, these being integers, floating point numbers and boolean values. The language contains two aggregate data structures. The first of these are Tuples, which contain a fixed number of elements which can be of mixed type, and are denoted using curved brackets, for example `(a,b)` is a tuple containing two elements. The other aggregate type are Vectors, which are a single dimensional array of potentially unlimited length, but containing only one type. These are denoted using square brackets, for example `[a,b,c]` is a vector of length three.

Function composition $(\cdot)$ follows normal mathematical semantics, and this ordering means that BMF programs are read from right to left, with the input coming in to the right hand side of the program, and output leaving from the left. Vector map (*) applies its argument function individually to each element of the input vector.

All applied to for Tuples, also called alltup $(^\circ)$, constructs a tuple by applying each of its input functions individually to the entire input data, and placing the results as the elements of a tuple. The identity function (`id`) simply passes its argument through unchanged. Tuple access (π_i) is used to extract individual elements of its input tuple. Constant functions ignore their input, always returning their constant value.

Binary operators operate on a tuple of two values, while unary arithmetic operators operate on a single value, and include general mathematical

Description	Symbols(s)	Semantics
Function Composition	$\cdot$	$(f \cdot g)\, x = f(g\, x)$
Vector Map	$*$	$f * [x_1, \ldots, x_{n-1}] =$ $[f(x_1), \ldots, f(x_{n-1})]$
Alltup	$(f_1, \ldots, f_n)^\circ$	$(f_1, \ldots, f_n)^\circ x =$ $(f_1\, x, \ldots, f_n\, x)$
Identity function	`id`	$\mathrm{id}\, x = x$
Tuple access	π_i	$\pi_i(a_1, \ldots, a_n) = a_i$
Constant functions	`K`	$\mathtt{K}\, x = \mathtt{K}$
Binary Arithmetic ops	$+, -, \div, \times,$ `and`$, \ldots$	$+(x, y) = x + y$ etc.
Unary Arithmetic ops	$-,$ `not`$, \ldots$	$-(x) = -x$ etc.
Left distribute	`distl`	$\mathrm{distl}(a, [x_0, \ldots, x_{n-1}]) =$ $[(a, x_0), \ldots, (a, x_{n-1})]$
Zip	`zip`	$\mathrm{zip}([x_0, \ldots, x_{n-1}],$ $[y_0, \ldots, y_{n-1}]) =$ $[(x_0, y_0), \ldots, (x_{n-a}, y_{n-1})]$
Vector enumeration	`iota`	$\mathrm{iota}\ n = [0, 1, \ldots, n-1]$
Vector length	$\#$	$\#[x_0, \ldots, x_{n-1}] = n$
Vector indexing	$!$	$!([x_0, \ldots, x_{n-1}], i) = x_i$
Vector selection	`select`	$\mathtt{select}(v, [x_0, \ldots, x_{n-1}]) =$ $[!(v, x_0), \ldots, !(v, x_{n-1})]$

Table 2.1: Structures in Bird-Meertens Formalism

operators and logical operators. Left distribute (`distl`) takes a tuple of a value and a vector, and returns a vector of tuples consisting of the value, and the corresponding element of the input vector. Zip (`zip`) takes a tuple containing two vectors, and returns a vector of tuples formed by pairing corresponding elements from each vector. If the vectors are not the same length, zip stops after using all of the values from the shorter vector. Zip can also be generalised to handle more then two inputs, but this case is not dealt with in this thesis.

Vector enumeration (`iota`) takes an integer n and returns a vector of integers with values in order from 0 to $n - 1$. Vector indexing (!) takes a tuple of a vector and an integer i, and returns the element at position i in the vector, with 0 being the first position of the first element.

Vector selection (`select`) takes a tuple of two vectors, with the second one consisting of integer indices into the first vector. It returns a vector consisting of the values in the first vector that are indexed by the indices of the second. The resulting vector is the same length as the second index vector.

There are some additional constructs in the language, which are not covered in the discussions in this thesis. These are summarised below.

Value repetition (`repeat`)takes a tuple of a value and an integer n, and returns a vector of the value repeated n times. Vector reduction (/) takes an argument of a function that has input of a tuple of two values with identical types and returns a value of the same type, and applies this function to first to the first two elements in its input vector, then to this result and the next element, and this continues until it reduces the vector to a single value. Vector scan operates in a similar way, but returns each stage of the reduction as an element of a vector.

If (`if`) and While (`while`) allow conditional evaluation or repetition of functions. `if` takes three functions as parameters, the first of which must return a boolean. The other two functions must both return the same type, and the `if` construct will evaluate one of its arguments depending on whether the first function returns true or false. `while` takes two functions as parameters, and repeatedly evaluates the first function as long as the second function returns true.

2.2 Transforming BMF

As mentioned previously, BMF code can be transformed through the application of simple algebraic rewrite rules. These rules allow the program to be changed in quite major ways without changing its meaning. This can allow for quite extensive optimisation of a program, as the way an algorithm works can be changed without changing what it does.

These transformation rules take the form of equalities, with an example in equation 2.1 showing a transformation that moves a composition of functions outside of a map. Note that the function can also be applied in reverse, moving the composition inside the map.

$$(f \cdot g)* = f * \cdot g* \tag{2.1}$$

This transformation can, for example, change the type and size of the intermediate data structure which passes the results of g into f. Applying it 'forward', as shown in the equation, would make the intermediate structure an entire list, which may make further transformations easier, for example, while applying it the other way would make the structure only a single element, which can conserve memory.

A further example is show in equation 2.2. This rule allows a map to be moved inside an alltup. This is limited by our restriction of zip to only work on two input vectors, but if zip was expanded to more vectors, so too could this rule be expanded.

$$((f, g)^\circ)* = \texttt{zip} \cdot (f*, g*)^\circ \tag{2.2}$$

Repeatedly applying these transformations to a program can produce significant improvements in both execution time and memory usage. For a more complete description of some of these transformations, refer to [Ale05].

2.3 Parallelisation of BMF

BMF contains many structures that are easily parallelisable, with many constructs having direct equivalent calls in many parallel processing APIs, such as MPI. Constructs such as map, scan and reduce can easily be mapped onto a parallel architecture, and program transformations can be used to minimise communication overheads through changing where parallel constructs occur in the program, and how much data is routed through them.

This ease of parallelisation is one of the main motivations for using BMF. It allows for easier automation of this process then in most other languages, and there has been some success in doing this [Ale04]. This thesis is not concerned with the parallelisation of BMF, but rather with BMF code that is run on a conventional processors. This code would arise during the parallelisation process as the code to be executed on each node of the parallel machine.

Chapter 3

The Baseline Compiler

The first stage consists of the construction of a baseline compiler. This compiler uses a small set of simple rules to specify how to generate C code from a BMF program. These rules, while simple, produce very inefficient code, through frequent allocation and deallocation of memory, as well as copying data to fill this memory. The reason that these rules are used is their simplicity, as they allow for easy construction of the compiler. This compiler forms a basis for comparison against which to measure the performance of the optimising compiler.

3.1 Program Translation Rules

The rules used for the translation define when memory for each constructs input and output is allocated and deallocated. the rules can be summarised as follows:

1. Space for the output of scalar valued functions is allocated just prior to the evaluation of that function.

2. Space for the output of vector valued functions is allocated just prior to the evaluation of that function

3. Space for a functions input is deallocated at the end of the function that triggered its allocation.

Using these rules means that each construct will have to create its entire output anew, with there being no reuse of the input values to any function. The constructs must allocate space for its output and then fill the data in. For some constructs, such as `id`, this entails copying the entire input to the output. For some others, a slightly changed version of the input will have to be copied into the output. This is the main source of the inefficiency of this compiler, and is the issue that the optimising compiler addresses.

3.2 Implementation of the Baseline Compiler

The baseline compiler was implemented in Haskell, a popular functional language [JHH$^+$93]. This language was chosen for its pattern matching and list handling capabilities, which are useful tools when constructing a compiler. The compiler produces C code, which was then compiled using GCC.

The compiler consists of several modules, which are listed below:

- **Main**: Contains the program entry point, testing functions and handles reading of files.
- **Parser**: Contains the lexer and parser. These were written from scratch rather then using a tool due to the simplicity of the BMF syntax. The lexer takes as input a raw text file and produces a list of symbols, from which the parser produces an abstract syntax tree (AST) using recursive descent parsing.
- **Types**: Contains definitions of the type system (covered in section 3.2.1) and the AST, as well as functions for accessing and modifying these data structures. This also performs type inferencing (covered in section 3.2.2), as well as labelling each node of the AST with a unique number, which is used for variable naming.
- **Structs**: Contains functions which manage the mapping of BMF types into C structs
- **Code**: Contains functions for storing C code templates, which are loaded from files. These templates are used for the C file start and end, as well as some of the simpler functions. Many of the other functions were found to be too complex for a template to be useful.
- **Translator**: Contains the functions to produce C code from the AST, by recursively traversing the AST.

These modules were implemented in separate Haskell modules.

3.2.1 Type System

The compiler has three basic types, these being integers, floating point numbers and boolean values. A single one of these values is represented as a struct containing a single value. The two aggregate types in the language are represented in the typing system as a type containing other types. Tuples are represented as a tuple structure, which contains a list of the types that are contained within the tuple. This is mapped into a struct that contains pointers to structs representing the contained types.

Vectors are represented within the type system as a vector structure, which contains another type struct representing the type of the elements of the vectors. This is mapped into a struct which contains a pointer to an array of structs representing the contained type, as well as an integer which holds the length of the list.

Both tuples and vectors can be arbitrarily nested within each other, and much of the complexity in the baseline system comes from dealing with this potential nesting.

3.2.2 Type Inferencing

The compiler uses a type inferencing system to determine the types used by each construct in the program being compiled. During the inferencing stage, each node of the AST is annotated with the type structures representing its input and output types. The initial type is derived from an annotation at the start of the BMF program file. The types for the nodes of the AST is then calculated through a recursive descent traversal of the AST, with structures passing on appropriate types to their child nodes - e.g. `map` will pass on the type contained within its input vector to its children.

3.3 Example of Translating a BMF Program

This section will show how a simple program is mapped into C by the baseline compiler. The program to be mapped is given in equation 3.1.

$$(+ \cdot (\mathtt{id}, \mathtt{id})^{\circ})* \tag{3.1}$$

This program takes as input a vector of numbers, and doubles each number by adding it to itself. The functions between the outer brackets are mapped to each individual element of the input. These functions take a number, and produce a tuple containing the same number twice. This tuple is then passed into the plus operator, which adds the two elements of the tuple together. These results are then recombined into a vector by the map.

The compiler does not operate on the code in this form, rather it must be changed into a form that is easier to parse (and type). This transformation is currently performed by hand, but this system is intended to be a 'back end' to a compiler, that would produce this code from a separate language. Once the program has been transformed into a suitable form, it looks as follows:

```
[[int]]
map(comp(op(plus), alltup<id,id>))
```

The notation on the first line is the initial type notation, in this case it denotes a vector of integers (the outer square brackets delimit the type notation). The second line is the program itself. It is mostly the same as the code above, with the symbols replaced with words, and the map and function composition changed to be prefix, rather then infix, with their arguments in a comma separated list in brackets. All functions which have functions as arguments are specified like this.

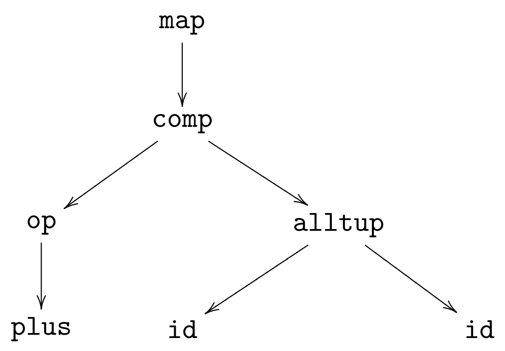

Figure 3.1: Abstract Syntax Tree of example program

The abstract syntax tree for this program is show in 3.1. This does not show the type annotations that are stored in each node of the tree. As an example of these annotations, the map at the start of the program has an input type of a vector of integers, and an output type of the same, while the alltup's input type is a single integer, and its output type is a tuple of two integers.

Each of the types that occurs in the AST is mapped to a C struct that can hold data of the correct type and form. The type definitions for this program are shown in Figure 3.2.

These structs are used during the translation process to access the programs data. The translator translates the AST into C through a recursive descent of the AST, and the body of the program produced is shown in Figure 3.3. The listing is missing the code which loads the input data (from a file), and places it into 'input', as well as the code which times the programs execution.

The inefficiencies of the baseline system can be seen in this code. It has produced numerous allocations, with several of them allocating variables that are only used to hold an unmodified copy of the current working value, and are only read from in the addition line. Eliminating these variables and allocations is the main focus of the optimisation techniques used in the second part of the project, covered in Chapter 4.

```
// Represents a single integer
typedef struct _B_Int0
{
    int var1;
} B_Int0;

// Represents a vector of integers -  the pointer points to an array
typedef struct _B_Vector1
{
    B_Int0 *var1;
    int size;
} B_Vector1;

// Represents a tuple containing two integers
// these pointers point to a single struct of the type above
typedef struct _B_Tuple2
{
    B_Int0 *var1;
    B_Int0 *var2;
} B_Tuple2;
```

Figure 3.2: Structures used in the example program

```
 /////////////////Start of Map
 output = (B_Vector1 *) malloc(sizeof(B_Vector1));
 output->size = input->size;
 output->var1 =
     (B_Int0 *) malloc (sizeof( B_Int0) * input->size);
 B_Int0 *map1;
 B_Int0 *mapo1;
 int loop1;
 for(loop1 = 0; loop1 < input->size; loop1++)
 {
     map1 = (&input->var1[loop1]);
     B_Tuple2 *cmp11;
     //////////// start of alltup template
     cmp11 = (B_Tuple2 *)malloc (sizeof(B_Tuple2));


     ///////////Start of ID template

     cmp11->var1 = (B_Int0*)malloc(sizeof(B_Int0));
     cmp11->var1->var1 = (int)map1->var1;

     /////////// End of ID
     ///////////Start of ID template

     cmp11->var2 = (B_Int0*)malloc(sizeof(B_Int0));
     cmp11->var2->var1 = (int)map1->var1;

     /////////// End of ID
     //////////////End of Alltup

     /////////////// Start of binary op(+) template

     mapo1 = (B_Int0 *) malloc(sizeof(B_Int0));
     mapo1->var1 =
 (cmp11->var1->var1) + (cmp11->var2->var1);

     //////////// End of binary op (+) template
     //////////// End of Comp

     (&output->var1[loop1])->var1 = mapo1->var1;
     free(mapo1);
}

//////////////////End of map
```

Figure 3.3: Listing of main body of $(+ \cdot (\mathtt{id}, \mathtt{id})^{\circ})*$

Chapter 4

Optimisation Techniques

This chapter covers the optimisation techniques used to attempt to improve the performance of the baseline compiler. Initially, the aim was to implement update in place analysis. This involves detecting when a function's input variable is not being used again in a program, and thus can be reused for the functions output [HB85]. However, after following the observations in [WB94], it was felt that focusing on eliminating routing functions would provide better results in the time available. Routing functions are functions that change the way in which data is addressed, rather then the data itself, and these changes in addressing can often be evaluated at compile time, rather then run time.

4.1 Routing Functions

Routing functions are functions that change the shape of data, or the way it is addressed, without changing the values of the data. These functions, such as `id`, `repeat`, π_i, and `distl`, as well as functions with routing like behaviours, such as `alltup`, perform transformations to the shape of the data.These transformations can be computed at compile time, rather than at run time.

$$+ \cdot (id, id)^{\circ} \tag{4.1}$$

As an example, consider the simple BMF code in 4.1. This code takes its input, and produces a tuple consisting of two copies of the input. It then adds these two copies, to produce its result, which is double the initial number. In the baseline system, this is literally what will happen, that two copies of the input will be created despite the fact that on a conventional processor such copying is unnecessary, as the input data can be read twice from where it is. The baseline compiler will produce code that maps the variables as shown in Figure 4.1, with a, b, and c being the variables. All three of these variables would contain exactly the same data after this code is executed.

$$\begin{tikzcd}
+ \arrow[d] & \cdot & (& \mathtt{id} \arrow[d] & , & \mathtt{id} \arrow[d] &)^\circ & \mathit{input} \arrow[l] \arrow[d] \\
d & & & c & & b & & a
\end{tikzcd}$$

Figure 4.1: How variables are bound in the baseline compiler for $+ \cdot (\mathtt{id}, \mathtt{id})^\circ$

4.2 Removal of Routing Functions

Eliminating routing functions entails evaluating these functions in the compiler, and producing appropriate mappings so that functions that use the results of the routing functions can find the data they need. For the simple program in 4.1, these mappings are not very complex - each variable is simply mapped to point to the original input. The plus must create a new variable to store its output, as we are not performing analysis to determine whether it is safe to destructively update one of its input variables. The result of this mapping can be seen in Figure 4.2.

$$\begin{tikzcd}
+ \arrow[d] & \cdot & (& \mathtt{id} \arrow[drr] & , & \mathtt{id} \arrow[d] &)^\circ & \mathit{input} \arrow[l] \arrow[dll] \\
b & & & & & a & &
\end{tikzcd}$$

Figure 4.2: How variables are bound in the optimising compiler for $+ \cdot (\mathtt{id}, \mathtt{id})^\circ$

These mappings can become more complex when functions such as distl and zip are used, as they change the way in which the original data is addressed. When these functions are mapped, their mappings will contain information on how to transform the data request so that it matches the shape of the data being pointed to. Consider the code in 4.2.

$$(+) * \cdot \mathtt{distl} \cdot (id, \mathtt{iota} \cdot \#)^\circ \tag{4.2}$$

This program takes a vector as input, and first makes a tuple containing the input and another vector containing the numbers from 0 to the length of the input - 1. The `distl` then makes this into a vector of tuples, each containing the input and a single number. The addition is then mapped across this vector, resulting in a vector containing the numbers from the input to the input$*2-1$.

The map and plus is not central to this discussion, but is included because otherwise the distl would have to produce its output directly, negating some of the routing elimination. The discussion below will not refer to the map and plus, rather it will just refer to the second part of the program. How the baseline compiler would map this is shown in Figure 4.3. Note that the $b-1$ in the `iota` and `distl` results is the value of $b-1$, not a reference to the variable b. Also note how many copies of the original input a are required to be created - one for the `id`, and then one copy for each element of the `iota` output vector. Since the output of the `iota` will have a length equal to the length of the input vector, this program has a space and time complexity of $O(n^2)$, where n is the length of the input vector.

$$\begin{tikzcd}
\texttt{distl}\cdot(\arrow[d] & \texttt{id}, \arrow[d] & \texttt{iota}\cdot \arrow[d] & \#)^\circ \arrow[d] & \mathit{input} \arrow[l] \arrow[d] \\
{[(d,0),(e,1),\ldots,(y,b-1)]} & c & {[0,1,\ldots,b-1]} & b & a
\end{tikzcd}$$

Figure 4.3: How variables are bound in the baseline compiler for code in 4.2

When the routing functions in this code are removed, all of this unnecessary copying is eliminated. The result of this mapping can be seen in Figure 4.4. The meaning of the `iota`(b) will be explained below in 4.3.

Note that the `distl` now contains numerous references to the original value a. This version performs no copying of the input, and the only part of this code which cannot be computed at compile time is the length operation. As vectors store their length as a part of their data, this is constant, giving a reduction in running time to $O(1)$, and reduction in space complexity to $O(n)$.

$$\begin{tikzcd}
\texttt{distl}\cdot(\arrow[d] & \texttt{id}, \arrow[d] & \texttt{iota}\cdot \arrow[d] & \#)^\circ \arrow[d] & \mathit{input} \arrow[l] \arrow[d] \\
{[(a,\texttt{iota}(b)),\ldots,(a,\texttt{iota}(b))]} & a & \texttt{iota}(b) & b & a
\end{tikzcd}$$

Figure 4.4: How variables are bound in the optimising compiler for code in 4.2

4.3 Handling Iota

The `iota` function, while not a routing function as such, has a structure that makes it amenable to a similar style of optimisation that improves its time and space performance. The `iota` function takes in an integer n, and returns a vector consisting of numbers running from 0 to $(n-1)$ inclusive. Given that the only difference between the output of different occurrences of `iota` is the length of the vector, it makes sense to only store this length, and to simply compute the appropriate number when it is requested.

The value to be returned can easily be calculated from the index requested - the index is simply returned (indexes in BMF are 0-based). The length is needed only when the iota is iterated over or its length is requested. This effectively reduces the time and space complexity of the `iota` from $O(n)$ to $O(1)$. So in the examples above, the `iota`(b) refers to an `iota` with length b. Note that this b would be the value of b, not a reference.

Chapter 5

The Optimising Compiler

This chapter describes the optimising compiler. This compiler uses the routing elimination methods described above to produce more efficient code from BMF.

5.1 Type System

The optimising compiler accomplishes much of its routing function elimination through an expanded typing system. This typing system has the location of a variable stored in the type information, allowing each type to know exactly how it is to be accessed. Changing this information is how most of the routing elimination is accomplished. The types also store a list of 'loop variables', which are used to index into vectors. These loop variables are assigned during type inferencing (5.2).

An additional type is also introduced, that being an `iota` type. This is used for the mapping of `iota` to a single variable representing size. The iota type is contained within a vector type, to prevent it requiring special cases whenever a function operates on a vector in the compiler.

5.2 Type Inferencing

The optimising compiler has a much expanded type inferencing section over the baseline compiler. Most of the work involved in routing function elimination takes place in this component of the compiler. As in the baseline system, this performs a recursive descent traversal of the AST, filling in input and output types. In addition, it now also fills in the information in the type trees about where their information is stored. In doing so, it also allocates the variables used in the program, as well as the loop variables used. This was done in a combined step to allow the routing function elimination code to know where new variables are declared, without requiring possibly complex analysis or tree annotations.

5.3 Allocation and Initialisation of Memory

Owing to the introduction of variable reuse in the optimising compiler, the allocation and initialisation of memory must now behave in a different way. The use of flat C arrays for vector memory storage means that it is not possible to allocate a vector one element at time, which would allow for functions to do their own allocation. For the sake of efficiency, vectors are allocated whole. One logical place for this allocation is at the first function that 'enters' the vector. By this I mean that a function like `map`, which allows the functions inside of it to access the elements of the vector.

To this end, map, and similar functions, examine their input and output type structures, and determine what variables are in the output but not in the input. These variables are then allocated and initialised. This, however, introduces a problem with 'intermediate' variables, such as those that might be produced by a composition when both of its functions must create new output structures. If the output of the first function is not used in the output of the second function, they will not be detected as needing allocation by the map. This then causes these variables to not be allocated.

One approach to fixing this is through special code to handle composition, which detects these variables, and then redeclares them as variables 'local' to the composition. These variables are changed so that they have none of the structure that exists in the other variables outside of the composition. So a composition with an intermediate value of an integer, which is inside a map, would have the local value as purely a integer, while the input and output would consist of a vector of integers. These intermediate values would be reused in this case, with the same variable used in each iteration of the map. This is safe, as the fact it is an intermediate variable means it cannot possibly be carried through in a reference in the output. As an example, consider the program in Equation 5.1.

$$(\texttt{iota} \cdot + \cdot (\texttt{id}, 5)^{\circ})* \tag{5.1}$$

The interior function takes a number as input, adds 5 to it, and then uses this as an argument to iota. This is mapped over an array of integers. The mappings to variables in this program is shown in Figure 5.1. Note that constants are inserted directly into the produced code, and thus are not stored as variables. The variable b, which is the result of the plus is the intermediate value that needs to be properly detected. It is not present in the input of the function, and nor will it be present in the output, as iota takes a copy of its input for its vector length. In this case b can be mapped to a variable local to the loop, of type integer. Note that the references to a and c are actually values indexed into a vector (c is a vector of `iota` vectors).

$$\begin{tikzcd}
output \arrow[d] & \texttt{(iota}\cdot \arrow[l] \arrow[d] & +\cdot \arrow[d] & \texttt{(id} \arrow[d] & 5)* \arrow[d] & input \arrow[l] \arrow[d] \\
c & c & b & a & 5 & a \\
Vector(int) & Vector(int) & int & int & & int
\end{tikzcd}$$

Figure 5.1: Variable mappings and types for program in Equation 5.1

5.4 Implementation of Optimising Compiler

The optimising compiler was written in Haskell, with a target language of C, the same as the baseline compiler. It consists of the following modules:

- **Main**: Contains the program entry point, testing functions and handles reading of files.
- **Parser**: Contains the lexer and parser. This was reused from the baseline compiler, with minor changes to accommodate the new type system.
- **Types**: Contains definitions of the type system (covered in section 5.1) and the AST, as well as functions for accessing and modifying these data structures.
- **Inferencing**: Contains the type inferencing system, covered in section 5.2.
- **Translator**: Contains the functions to produce C code from the AST, by recursively traversing the AST.

5.5 Example of translating a BMF program

This section demonstrates how the simple program shown in equation 5.2, the same program used in section 3.3, is translated by the optimising compiler. The optimising compiler uses a simpler system of structs to store the variables, with single scalars simply stored as a variable of the appropriate type - eg. an integer stored in `int var1`. Vectors of scalars are stored in structs of the form shown in Figure 5.2, this being the struct for a vector of integers.

$$(+ \cdot (\texttt{id}, \texttt{id})^{\circ})* \tag{5.2}$$

```
typedef struct _Vint
{
int *contents;
int size;
} Vint;
```

Figure 5.2: Structure used to store arrays of Integers

Structs for other types are similar. Vectors of vectors are represented using void pointers in a similar struct. Tuples are not directly represented, rather they are just accessed through the mapping of types and variables. As an example, a vector of tuples, would have each element of the tuple stored in a vector, which are accessed together in a similar fashion to the way FORTRAN represents ‘records’. The input program needs some manual normalisation, a requirement which simplifies the design of the compiler. This normalisation pushes compositions upwards in the program, outside of maps and alltups, to avoid intermediate values. These are covered in more detail in section 5.6. These will change the program’s AST into the shape shown in Figure 5.3

$$\begin{tikzcd}
& \texttt{comp} \arrow[dl] \arrow[dr] & & \\
\texttt{map} \arrow[d] & & \texttt{map} \arrow[d] & \\
\texttt{op} \arrow[d] & & \texttt{alltup} \arrow[dl] \arrow[dr] & \\
\texttt{plus} & \texttt{id} & & \texttt{id}
\end{tikzcd}$$

Figure 5.3: Abstract Syntax Tree of example program in equation 5.2 for optimising compiler

The optimising compiler then transforms this through a similar process to in the baseline system into a C program. The main body of the C program is shown in Figure 5.4. Again this is missing code that reads in the input from a file, does timekeeping, and output printing. The input has been read into var0, a struct of type Vint, as shown above. As can be seen, the `alltup` and `id` functions have been reduced to comments. The empty map loop that contains them is still in the program, but this can be removed through post-processing or analysis of map contents.

```
////******B_comp
int mapLoop11;
for(mapLoop11 = 0;
    mapLoop11 < ((Vint *)var0)->size; mapLoop11++)
{
    ////******B_alltup
    ////******B_id
    ////******B_id
 }

((Vint *)var1)->size = ((Vint *)var0)->size;
((Vint *)var1)->contents =
    (int *) malloc(sizeof(int) * ((Vint *)var0)->size);
int mapLoop21;
for(mapLoop21 = 0;
    mapLoop21 < ((Vint *)var0)->size; mapLoop21++)
{
    (var1)->contents[mapLoop21] =
        (var0)->contents[mapLoop21]
        + (var0)->contents[mapLoop21];
}
```

Figure 5.4: Main body of code from Optimising compiler for example program in equation 5.2

5.6 Current State of Optimising Compiler

The optimising compiler is currently implemented on a subset of the BMF language - these being `id`, `map`, $\circ$, `iota`, $+$, `distl`, `comp`, and π_i. The programs also require the manual normalisation mentioned in section 5.5, before the program is entered into the compiler to enable the programs to be translated successfully. Currently, the detection of temporary variables, as mentioned in section 5.3 is not implemented, and code that results in this situation must be avoided through the application of transformations, such as those given in section 2.2. Code that causes this is currently compositions inside of maps. The transformations given in section 2.2 can be used to move the compositions outside of the maps, where the temporary variables are the output of the first map, where the map can initialise them.

While this normalisation of the program is currently manual, it can be automated without great difficulty, due to the simple nature of the transformations required. Alternatively, implementation of the temporary variable detection given in section 5.3 would allow the compiler to work on un-normalised BMF.

Chapter 6

Results and Conclusions

This chapter covers the results obtained from the optimising compiler, comparing them to the baseline compiler and hand coded C. Three example programs are used to illustrate the performance gains made by the optimising compiler over the baseline compiler.

6.1 Results

The performance of code produced by the optimising compiler was compared to two other sources - code produced by the baseline compiler (covered in chapter 3), and hand coded C versions of the same programs. The C code has identical input loading and timekeeping code to the other two implementations. The timekeeping code in all three versions only times the actual program - it does not cover reading the input or any output of the program.

Three programs were used for the comparison. The first is the program given in equation 6.1, which is the program used as an example in sections 3.1 and 5.2. The second program is given in equation 6.2, and creates an `iota` vector of length n, and then adds n to each element. This program uses `distl`, and demonstrates the large saving the optimising compiler can generate on this type of structure. The third program is given in equation 6.3. This program is similar to the second program, but contains another `distl` that distributes the output of the first one. This program produces an `iota` vector of length n, and adds $n * 2$ to each element. This nested distribution further demonstrate the space and time savings over the baseline compiler.

$$(+ \cdot (\mathtt{id}, \mathtt{id})^{\circ}) * \tag{6.1}$$

$$(+) * \cdot \mathtt{distl} \cdot (\mathtt{id}, \mathtt{iota})^{\circ} \tag{6.2}$$

$$((+) \cdot (\pi_2 \cdot \pi_2, (+) \cdot (\pi_1, \pi_1 \cdot \pi_2)^{\circ})^{\circ}) * \cdot \mathtt{distl} \cdot (\mathtt{id}, \mathtt{distl} \cdot (\mathtt{id}, \mathtt{iota})^{\circ})^{\circ} \tag{6.3}$$

Each program was translated by the baseline compiler and the optimising compiler, and a hand coded version was produced. The full code produced for examples one and three can be found in the appendices. The results for each example can be found in tables 6.1, 6.2, and 6.3, with graphs of these results in Figures 6.3, 6.4 and 6.5. Note that the baseline compiler was unable to run on the larger data sets for program 3, as it ran out of memory, and that all results are from averages of five runs.

The results are what would be expected - the hand coded examples perform best, the baseline compiler's code worst, and the optimising compiler in-between. The performance of the optimising compiler's code comes quite close to the hand coded programs, especially in the second example. In the second example, the produced code is almost identical to the hand code, as can be seen comparing the code in Figure 6.1 with that in Figure 6.2.

The hand coded program has a bigger margin over the optimising compiler in the third example. This is due to the optimising compiler requiring a third data structure to store the intermediate result of the computation, as well as the extra pointer referencing needed to access the intermediate structure. In all three cases, the difference between the optimising compiler code and the hand coded program appears to be a linear factor, most probably caused by the extra pointer accesses.

To investigate this difference further, as well as see if the optimising compiler's results could be further improved, the optimising compilers code and the hand coded program were compiled with some different options.

First the -g option in GCC, which disables optimisations, was used to get a base for comparison, and then the -O3 option, which enables various optimisation techniques. This was done to see if the C compiler could improve the memory accesses of the optimising compiler, as it was suspected that there are locality problems with the linear accessing of the two arrays used. While there was the expected improvement in performance, the linear factor was still there, albeit smaller. The small variations from linear in some of the results are suspected to be testing artifacts, but further work is required to check this.

6.2 Future Work

There is still work to be done on the optimising compiler. An obvious direction for work in the future is extending the compiler to work on a larger subset of BMF, or perhaps the entire language. There is also work to be done on improving the way it works, with issues like the intermediate variable detection mentioned in section 5.3 needing to be resolved.

On a bigger scale, there is work to be done looking at update in place analysis for BMF, and potentially its integration with routing elimination. Update in place analysis is concerned with detecting when the results of a

```
    ((Vint *)var1)->size = var0;
    ((Vint *)var2)->size = ((Vvoid *)var1)->size;
    ((Vint *)var2)->contents =
(int *) malloc(sizeof(int) * ((Vvoid *)var1)->size);
    int mapLoop21;
    for(mapLoop21 = 0;
        mapLoop21 < ((Vvoid *)var1)->size; mapLoop21++)
    {
        (var2)->contents[mapLoop21] = var0 + mapLoop21;
    }
```

Figure 6.1: Optimising compiler generated code for main body of Example 2 (equation 6.2).

```
    output = (int *)malloc(sizeof(int) * input);
    for(i = 0; i < input; i++)
    {
        output[i] = input + i;
    }
```

Figure 6.2: Hand coded main body of Example 2 (equation 6.2).

	Execution time (ms)		
Size of input array	Baseline	Optimising	Hand Coded
50	14	1	1
100	30	2	1
200	60	3	2
500	145	5	3
1000	286	16	7
5000	1467	63	50
10000	2961	129	100
50000	19875	1294	1522
100000	32341	1329	1113
500000	152161	6589	5540
1000000	293728	13304	11210
5000000	1471206	69451	57024

Table 6.1: Results for example program 1 (equation 6.1)

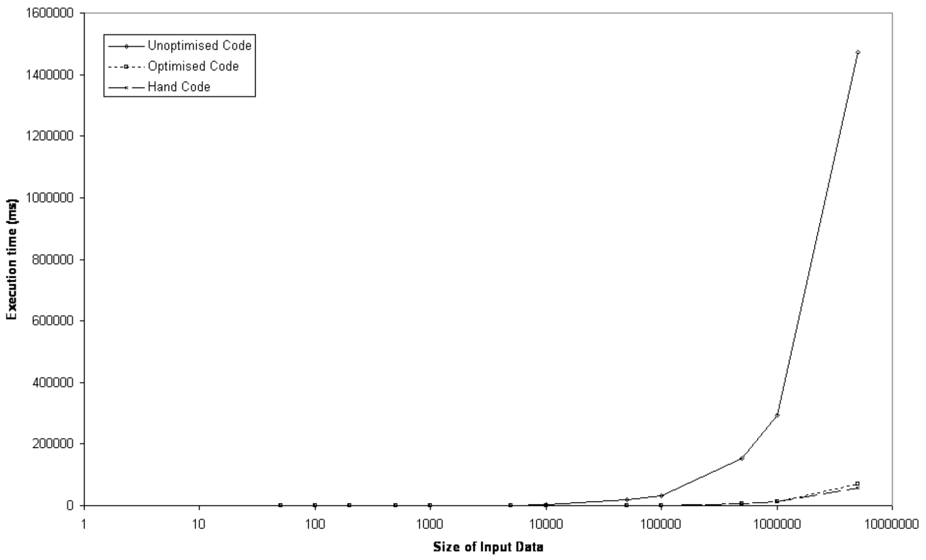

Figure 6.3: Graph of results for example program 1 (equation 6.1) Note the logarithmic X-scale.

	Execution time (ms)		
Size of input array	Baseline	Optimising	Hand Coded
10	4	1	0
50	14	1	1
100	26	1	1
500	120	3	3
1000	236	10	5
5000	1867	44	36
10000	2423	91	78
50000	16251	472	467
100000	29276	1030	900
500000	128681	4947	4734
1000000	256616	13232	12755
5000000	1256998	54580	51807

Table 6.2: Results for example program 2 (equation 6.2)

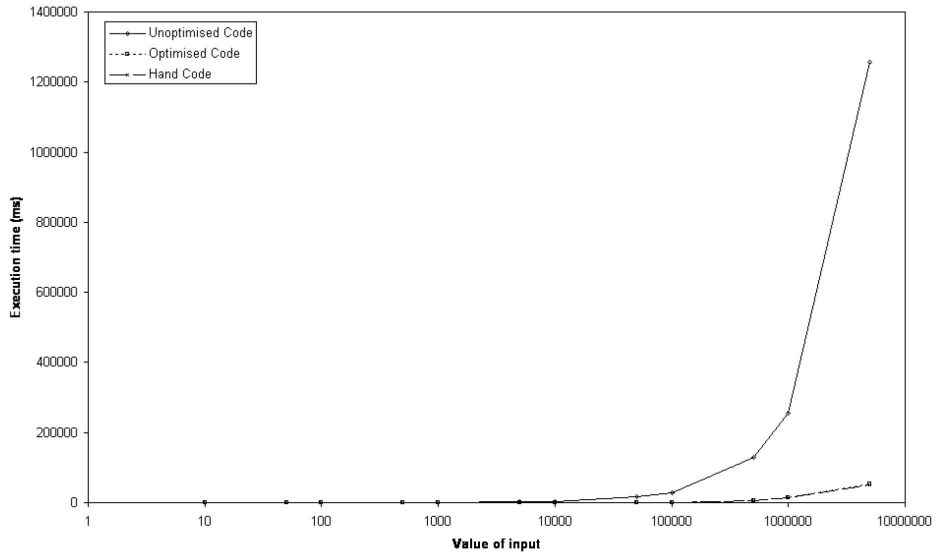

Figure 6.4: Graph of results for example program 2 (equation 6.2) Note the logarithmic X-scale.

Size of input array	Execution time (ms)		
	Baseline	Optimising	Hand Coded
10	14	1	0
50	71	1	1
100	150	2	1
500	745	9	3
1000	2136	18	5
5000	10890	84	33
10000	19165	165	72
50000	81894	909	418
100000	158676	1866	883
500000	773281	12982	4564
1000000	1539961	22895	12573
5000000	-	98473	49711

Table 6.3: Results for example program 3 (equation 6.3)

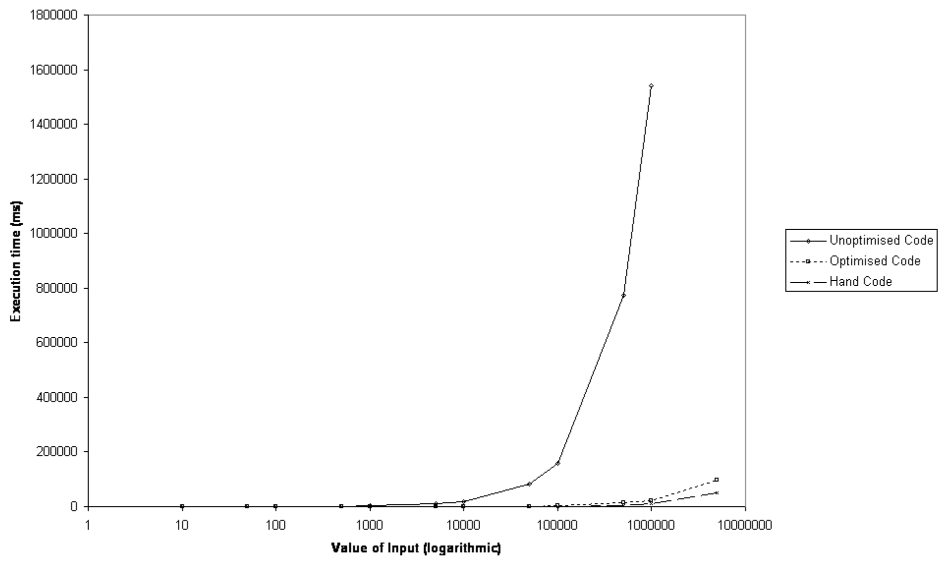

Figure 6.5: Graph of results for example program 3 (equation 6.3) Note the logarithmic X-scale.

Size of input array	Execution time (ms)	
	Optimising	Hand Coded
500	9	3
1000	17	8
5000	80	36
10000	163	79
50000	911	422
100000	1880	846
500000	11535	4485
1000000	21905	13237
2000000	42310	20637
3000000	60403	30410
4000000	80140	39550
5000000	99448	48726

Table 6.4: Results for example program 3 (equation 6.3) using -g option

	Execution time (ms)	
Size of input array	Optimising	Hand Coded
500	6	2
1000	10	5
5000	41	21
10000	95	44
50000	528	267
100000	1079	549
500000	7744	3023
1000000	14279	7479
2000000	31763	14709
3000000	39038	21438
4000000	51270	27844
5000000	63582	33898

Table 6.5: Results for example program 3 (equation 6.3) using -O3 option

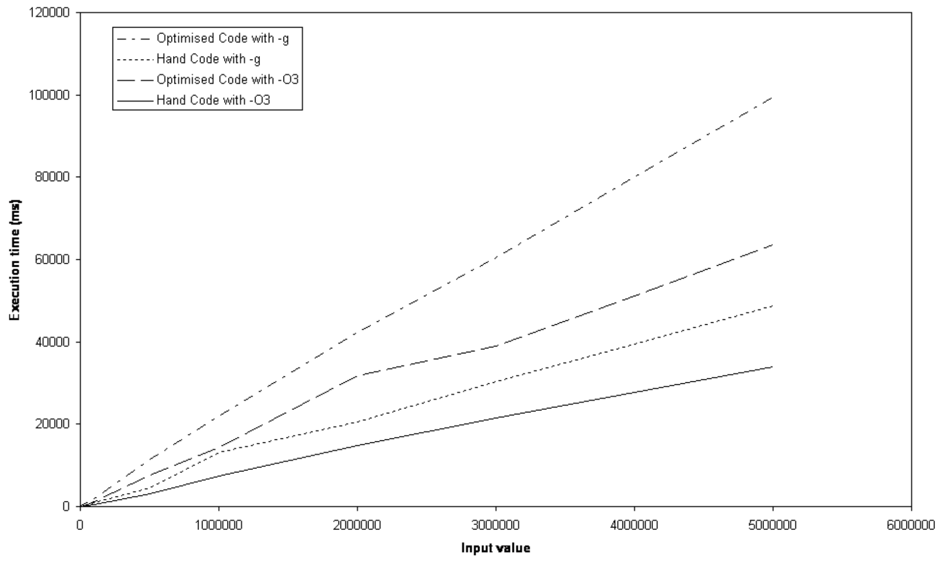

Figure 6.6: Graph of results for example program 3, comparing using -g and -O3

function can be written over its input, rather then then being written into a new piece of memory. This would produce reductions in memory usage, as well as execution time through less allocation and deallocation of memory.

6.3 Conclusion

This thesis has covered the translation of point-free BMF code into an imperative language, through two methodologies. The first was using some simple rules relating to memory allocation and deallocation, which were simple to implement but produced inefficient code. The second improves the performance through the elimination of routing functions, that is functions that only change the way data is addressed, not the values of the data. It has been shown that using these methods to move the evaluations of these functions to compile time, rather then evaluating them at run time, improves the performance of the resulting programs, both in terms of execution time and memory usage.

Appendix A

Example 1 Code

This chapter contains the code produced by the baseline compiler and the optimising compiler, as well as the hand coded version, of the program given in equation A.1. Note that the code for example 2, given in equation 6.2, is similar to the code in example 3, and has been omitted.

$$(+ \cdot (\mathtt{id}, \mathtt{id})^{\circ})* \tag{A.1}$$

A.1 Baseline Compiler Output

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/time.h>


typedef struct _B_Int0
{
    int var1;
} B_Int0;

typedef struct _B_Vector1
{
    B_Int0 *var1;
    int size;
} B_Vector1;

typedef struct _B_Tuple2
{
    B_Int0 *var1;
    B_Int0 *var2;
} B_Tuple2;


#define TRUE 1
#define true 1
#define false 0
#define FALSE 0

#define PRINT 0
//#define DEBUG

#ifdef DEBUG
#define dprint(x,y) printf(x,y)
#else
#define dprint(x,y)
#endif

/*
Format of input file:
first number an integer representing the number of entries in the file
next a number of doubles, equal to the above int.
*/

int main(int argv, char **argc)
{
    long loop;
    if(argv < 2)
    {
        printf("no argument found\n");
        exit(-1);
    }

    FILE *fd = fopen (argc[1], "r");
    if(fd == NULL)
    {
        printf("error opening input file");
        exit(-1);
    }
    struct timeval starttime;
    struct timeval endtime;
    struct timezone dummy;
```

```
char* rinputfile;

fseek(fd, 0, SEEK_END);
long filesize = ftell(fd);
//printf("filesize: %li",  filesize);

fseek(fd, 0, SEEK_SET);
rinputfile = (char *) malloc(filesize + 1);

int numread = fread(rinputfile, 1, filesize, fd);

rinputfile[filesize] = '\0';

//printf("read data: %i\n", numread);
//printf(rinputfile);
char* current;
char* point;

//long numEntries = strtol(rinputfile, &current, 10);
current = rinputfile;
//printf("numEntries: %li\n", numEntries);
B_Vector1* input;
input = (B_Vector1 *) malloc(sizeof(B_Vector1));
//input->var1  = (B_Int0*) malloc(sizeof(B_Int0) * numEntries);
//input->size = numEntries;
//long num = 0;

//for(num = 0; num < numEntries; num++)
//{
//    input->var1[num].var1 = (int) strtod(current, &point);
//    current = point;
//    //printf("read: %f \n", i[num]);
//    //printf(current);
//    //printf("\n");
//}
B_Vector1 *output;
//gettimeofday(&starttime, &dummy);

 long readnum1 = strtol(current, &point, 10);
current = point;
dprint("read in vector size %i\n", readnum1);
input->var1 =
    (B_Int0 *)malloc(sizeof(B_Int0) * readnum1);
    input->size = readnum1;
    int loopip1;
     for(loopip1 = 0; loopip1 < readnum1; loopip1++)
    {
    (&input->var1[loopip1])->var1 =
         strtol(current, &point, 10);
    current = point;
    dprint("read in int %i\n", (&input->var1[loopip1])->var1);
    }

    printf("Starting...\n")
;    gettimeofday(&starttime, &dummy);
    ////////////////Start of Map
    output = (B_Vector1 *) malloc(sizeof(B_Vector1));
    output->size = input->size;
    output->var1 =
        (B_Int0 *) malloc (sizeof( B_Int0) * input->size);
    B_Int0 *map1;
    B_Int0 *mapo1;
    int loop1;
    for(loop1 = 0; loop1 < input->size; loop1++)
    {
    map1 = (&input->var1[loop1]);
    B_Tuple2 *cmp11;
    //////////// start of alltup template
    cmp11 = (B_Tuple2 *)malloc (sizeof(B_Tuple2));


    ///////////Start of ID template

    cmp11->var1 = (B_Int0*)malloc(sizeof(B_Int0));
    cmp11->var1->var1 = (int)map1->var1;

    /////////// End of ID
    ///////////Start of ID template

    cmp11->var2 = (B_Int0*)malloc(sizeof(B_Int0));
    cmp11->var2->var1 = (int)map1->var1;

    /////////// End of ID
//////////////End of Alltup

    /////////////// Start of binary op(+) template
    mapo1 = (B_Int0 *) malloc(sizeof(B_Int0));
    mapo1->var1 =
         (cmp11->var1->var1) + (cmp11->var2->var1);

    //////////// End of binary op (+) template
//////////// End of Comp

    (&output->var1[loop1])->var1 = mapo1->var1;
    free(mapo1);
    }

    ////////////////////End of map

    gettimeofday(&endtime, &dummy);
    printf("\nFinished...\n");
    long timeElapsed = ((endtime.tv_sec - starttime.tv_sec) *
1000000L) + (endtime.tv_usec - starttime.tv_usec);
    printf("Time elapsed: %li\n", timeElapsed);
    if(PRINT)
    {
    printf("[");
    int prloop1;
    for (prloop1 = 0; prloop1 < output->size; prloop1++)
    {
    printf("    ");
        printf("%i", (&output->var1[prloop1])->var1);
    printf("");
    }
    printf("]");
    free(output);
    }
    }
```

A.2 Optimising Compiler Output

```
////example.bmf
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/time.h>
```

```
#define TRUE 1
#define true 1
#define false 0
#define FALSE 0

#define PRINT 0
//#define DEBUG

#ifdef DEBUG
#define dprint(x,y) printf(x,y)
#else
#define dprint(x,y)
#endif

typedef struct _Vint
{
    int *contents;
    int size;
} Vint;

typedef struct _Vfloat
{
    float *contents;
    int size;
} Vfloat;

typedef struct _Vdouble
{
    double *contents;
    int size;
} Vdouble;

typedef struct _Vbool
{
    int *contents;
    int size;
} Vbool;

typedef struct _Vvoid
{
    void *contents;
    int size;
```

```
} Vvoid;

int main(int argv, char **argc)
{
    long loop;
    long tempsize;
    if(argv < 2)
    {
        printf("no argument found\n");
        exit(-1);
    }

    FILE *fd = fopen (argc[1], "r");
    if(fd == NULL)
    {
        printf("error opening input file");
        exit(-1);
    }
    struct timeval starttime;
    struct timeval endtime;
    struct timezone dummy;

    char* rinputfile;

    fseek(fd, 0, SEEK_END);
    long filesize = ftell(fd);
    //printf("filesize: %li",  filesize);

    fseek(fd, 0, SEEK_SET);
    rinputfile = (char *) malloc(filesize + 1);

    int numread = fread(rinputfile, 1, filesize, fd);

    rinputfile[filesize] = '\0';

    //printf("read data: %i\n", numread);
    //printf(rinputfile);
    char* current;
    char* point;

    current = rinputfile;


    Vint *var0 = (Vint *) malloc(sizeof(Vint));
    Vint *var1 = (Vint *) malloc(sizeof(Vint));


    int input0loop0;
    tempsize = strtol(current, &current, 10);
    ((Vint *)var0)->size = tempsize;
    ((Vint *)var0)->contents =
         (int *) malloc(sizeof(int) * tempsize);
    for(input0loop0 = 0;
        input0loop0 < ((Vint *)var0)->size; input0loop0++)
    {
    (var0)->contents[input0loop0] = strtol(current, &current, 10);
    }



printf("Starting...\n");
    gettimeofday(&starttime, &dummy);
////******B_comp
    int mapLoop11;
    for(mapLoop11 = 0;
        mapLoop11 < ((Vint *)var0)->size; mapLoop11++)
    {
////******B_alltup
////******B_id

////******B_id

    }

    ((Vint *)var1)->size = ((Vint *)var0)->size;
    ((Vint *)var1)->contents =
(int *) malloc(sizeof(int) * ((Vint *)var0)->size);
    int mapLoop21;
    for(mapLoop21 = 0;
        mapLoop21 < ((Vint *)var0)->size; mapLoop21++)
    {
    (var1)->contents[mapLoop21] =
        (var0)->contents[mapLoop21] +
        (var0)->contents[mapLoop21];
    }
    gettimeofday(&endtime, &dummy);
    printf("\nFinished...\n");
    long timeElapsed = ((endtime.tv_sec - starttime.tv_sec)
* 1000000L) + (endtime.tv_usec - starttime.tv_usec);
    printf("Time elapsed: %li\n", timeElapsed);
    if(PRINT)
    {
    printf("[ ");
    int output0loop0;
    for(output0loop0 = 0;
         output0loop0 < ((Vint *)var1)->size; output0loop0++)
    {
    printf("%i\t", (var1)->contents[output0loop0]);
    }
    printf(" ]\n");
    }
}
```

A.3 Hand Coded Program

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/time.h>

#define PRINT 0

int main(int argv, char **argc)
{
    long loop;
    long tempsize;
    if(argv < 2)
    {
        printf("no argument found\n");
        exit(-1);
    }

    FILE *fd = fopen (argc[1], "r");
    if(fd == NULL)
    {
        printf("error opening input file");
        exit(-1);
    }
    struct timeval starttime;
    struct timeval endtime;
    struct timezone dummy;

    char* rinputfile;

    fseek(fd, 0, SEEK_END);
    long filesize = ftell(fd);
    //printf("filesize: %li",  filesize);

    fseek(fd, 0, SEEK_SET);
    rinputfile = (char *) malloc(filesize + 1);

    int numread = fread(rinputfile, 1, filesize, fd);

    rinputfile[filesize] = '\0';

    //printf("read data: %i\n", numread);
    //printf(rinputfile);
    char* current;
    char* point;

    current = rinputfile;

    int *input;
    int *output;

    int size = strtol(current, &current, 10);
    int i;

    input = (int *)malloc(sizeof(int) * size);

    for(i = 0; i < size; i++)
    {
        input[i] = strtol(current, &current, 10);
    }


    printf("Starting...\n");
    gettimeofday(&starttime, &dummy);

    output = (int *)malloc(sizeof(int) * size);

    for(i = 0; i < size; i++)
    {
        output[i] = input[i] + input[i];
    }

    gettimeofday(&endtime, &dummy);
    printf("\nFinished...\n");
    long timeElapsed =
        ((endtime.tv_sec - starttime.tv_sec) * 1000000L) +
        (endtime.tv_usec - starttime.tv_usec);
    printf("Time elapsed: %li\n", timeElapsed);

    if(PRINT)
    {
        printf("[ ");
        for(i = 0; i < size; i++)
        {
            printf("%i\t", output[i]);
        }
    }
    free(output);
    free(input);
}
```

Appendix B

Example 3 Code

This chapter contains the code produced by the baseline compiler and the optimising compiler, as well as the hand coded version, of the program given in equation B.1.

$$((+) \cdot (\pi_2 \cdot \pi_2, (+) \cdot (\pi_1, \pi_1 \cdot \pi_2)^{\circ})^{\circ}) * \cdot \mathtt{distl} \cdot (\mathtt{id}, \mathtt{distl} \cdot (\mathtt{id}, \mathtt{iota})^{\circ})^{\circ} \quad (B.1)$$

B.1 Baseline Compiler Output

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/time.h>


typedef struct _B_Int0
{
    int var1;
} B_Int0;

typedef struct _B_Vector1
{
    B_Int0 *var1;
    int size;
} B_Vector1;

typedef struct _B_Tuple2
{
    B_Int0 *var1;
    B_Int0 *var2;
} B_Tuple2;

typedef struct _B_Tuple3
{
    B_Int0 *var1;
    B_Tuple2 *var2;
} B_Tuple3;

typedef struct _B_Vector4
{
    B_Tuple3 *var1;
    int size;
} B_Vector4;

typedef struct _B_Vector5
{
    B_Tuple2 *var1;
    int size;
} B_Vector5;

typedef struct _B_Tuple6
{
    B_Int0 *var1;
    B_Vector5 *var2;
} B_Tuple6;

typedef struct _B_Tuple7
{
    B_Int0 *var1;
    B_Vector1 *var2;
} B_Tuple7;


#define TRUE 1
#define true 1
#define false 0
#define FALSE 0

#define PRINT 0
//#define DEBUG

#ifdef DEBUG
#define dprint(x,y) printf(x,y)
```

```
#else
#define dprint(x,y)
#endif

/*
Format of input file:
first number an integer representing the number of entries in the file
next a number of doubles, equal to the above int.
*/

int main(int argv, char **argc)
{
    long loop;
    if(argv < 2)
    {
        printf("no argument found\n");
        exit(-1);
    }

    FILE *fd = fopen (argc[1], "r");
    if(fd == NULL)
    {
        printf("error opening input file");
        exit(-1);
    }
    struct timeval starttime;
    struct timeval endtime;
    struct timezone dummy;


    char* rinputfile;

    fseek(fd, 0, SEEK_END);
    long filesize = ftell(fd);
    //printf("filesize: %li",  filesize);

    fseek(fd, 0, SEEK_SET);
    rinputfile = (char *) malloc(filesize + 1);

    int numread = fread(rinputfile, 1, filesize, fd);

    rinputfile[filesize] = '\0';

    //printf("read data: %i\n", numread);
    //printf(rinputfile);
    char* current;
    char* point;

    //long numEntries = strtol(rinputfile, &current, 10);
    current = rinputfile;
    //printf("numEntries: %li\n", numEntries);
    B_Int0* input;
    input = (B_Int0 *) malloc(sizeof(B_Int0));
    //input->var1  = (B_Int0*) malloc(sizeof(B_Int0) * numEntries);
    //input->size = numEntries;
    //long num = 0;

    //for(num = 0; num < numEntries; num++)
    //{
    //    input->var1[num].var1 = (int) strtod(current, &point);
    //    current = point;
    //    //printf("read: %f \n", i[num]);
    //    //printf(current);
    //    //printf("\n");
    //}
    B_Vector1 *output;
    //gettimeofday(&starttime, &dummy);

    input->var1 = strtol(current, &point, 10);
    current = point;
    dprint("read in int %i\n", input->var1);
    printf("Starting...\n")
;     gettimeofday(&starttime, &dummy);
    B_Vector4 *cmp1;
    B_Tuple6 *cmp11;
    //////////// start of alltup template
    cmp11 = (B_Tuple6 *)malloc (sizeof(B_Tuple6));


    ///////////Start of ID template

    cmp11->var1 = (B_Int0*)malloc(sizeof(B_Int0));
    cmp11->var1->var1 = (int)input->var1;

    /////////// End of ID
    B_Tuple7 *cmp2111;
    //////////// start of alltup template
    cmp2111 = (B_Tuple7 *)malloc (sizeof(B_Tuple7));
    ///////////Start of ID template

    cmp2111->var1 = (B_Int0*)malloc(sizeof(B_Int0));
    cmp2111->var1->var1 = (int)input->var1;

    /////////// End of ID
    cmp2111->var2 = (B_Vector1 *) malloc(sizeof(B_Vector1));
    cmp2111->var2->size = input->var1;
    cmp2111->var2->var1 =
(B_Int0 *)malloc(sizeof(B_Int0) * cmp2111->var2->size);
    int loopiota212111;
    for(loopiota212111 =
        0; loopiota212111  < cmp2111->var2->size; loopiota212111++)
    {
        cmp2111->var2->var1[loopiota212111].var1 = loopiota212111;
    }//////////////End of Alltup

    cmp11->var2 = (B_Vector5 *) malloc(sizeof(B_Vector5));
    cmp11->var2->size = cmp2111->var2->size;
    cmp11->var2->var1 =
(B_Tuple2 *) malloc(sizeof( B_Tuple2) * cmp2111->var2->size);
    int loop22111;
    for(loop22111 = 0;
        loop22111 < cmp2111->var2->size; loop22111++)
    {
    (&cmp11->var2->var1[loop22111])->var1 =
    (B_Int0 *)  malloc(sizeof(B_Int0));
    (&cmp11->var2->var1[loop22111])->var1->var1 =
    cmp2111->var1->var1;
    (&cmp11->var2->var1[loop22111])->var2 =
    (B_Int0 *)  malloc(sizeof(B_Int0));
    (&cmp11->var2->var1[loop22111])->var2->var1 =
    (&cmp2111->var2->var1[loop22111])->var1;
    }

//////////// End of Comp

//////////////End of Alltup

    cmp1 = (B_Vector4 *) malloc(sizeof(B_Vector4));
    cmp1->size = cmp11->var2->size;
    cmp1->var1 =
(B_Tuple3 *) malloc(sizeof( B_Tuple3) * cmp11->var2->size);
    int loop211;
    for(loop211 = 0; loop211 < cmp11->var2->size; loop211++)
    {
    (&cmp1->var1[loop211])->var1 =
    (B_Int0 *)  malloc(sizeof(B_Int0));
    (&cmp1->var1[loop211])->var1->var1 =
    cmp11->var1->var1;
    (&cmp1->var1[loop211])->var2 =
    (B_Tuple2 *)  malloc(sizeof(B_Tuple2));
    (&cmp1->var1[loop211])->var2->var1 =
    (B_Int0 *)  malloc(sizeof(B_Int0));
    (&cmp1->var1[loop211])->var2->var2 =
    (B_Int0 *)  malloc(sizeof(B_Int0));
    (&cmp1->var1[loop211])->var2->var1->var1 =
    (&cmp11->var2->var1[loop211])->var1->var1;
    (&cmp1->var1[loop211])->var2->var2->var1 =
    (&cmp11->var2->var1[loop211])->var2->var1;
    }

//////////// End of Comp

    /////////////////Start of Map
    output = (B_Vector1 *) malloc(sizeof(B_Vector1));
    output->size = cmp1->size;
    output->var1 = (B_Int0 *) malloc (sizeof( B_Int0) * cmp1->size);
    B_Tuple3 *map21;
    B_Int0 *mapo21;
    int loop21;
    for(loop21 = 0; loop21 < cmp1->size; loop21++)
    {
    map21 = (&cmp1->var1[loop21]);
    B_Tuple2 *cmp121;
    //////////// start of alltup template
    cmp121 = (B_Tuple2 *)malloc (sizeof(B_Tuple2));


    B_Tuple2 *cmp11121;
    cmp11121 = (B_Tuple2 *) malloc(sizeof(B_Tuple2));
    cmp11121->var1 = (B_Int0 *)  malloc(sizeof(B_Int0));
    cmp11121->var2 = (B_Int0 *)  malloc(sizeof(B_Int0));
    cmp11121->var1->var1 = map21->var2->var1->var1;
    cmp11121->var2->var1 = map21->var2->var2->var1;
    cmp121->var1 = (B_Int0 *) malloc(sizeof(B_Int0));
    cmp121->var1->var1 = cmp11121->var2->var1;
//////////// End of Comp

    B_Tuple2 *cmp21121;
    //////////// start of alltup template
    cmp21121 = (B_Tuple2 *)malloc (sizeof(B_Tuple2));


    cmp21121->var1 = (B_Int0 *) malloc(sizeof(B_Int0));
    cmp21121->var1->var1 = map21->var1->var1;
    B_Tuple2 *cmp2121121;
    cmp2121121 = (B_Tuple2 *) malloc(sizeof(B_Tuple2));
    cmp2121121->var1 = (B_Int0 *)  malloc(sizeof(B_Int0));
    cmp2121121->var2 = (B_Int0 *)  malloc(sizeof(B_Int0));
    cmp2121121->var1->var1 = map21->var2->var1->var1;
    cmp2121121->var2->var1 = map21->var2->var2->var1;
    cmp21121->var2 = (B_Int0 *) malloc(sizeof(B_Int0));
    cmp21121->var2->var1 = cmp2121121->var1->var1;
//////////// End of Comp

//////////////End of Alltup

    /////////////// Start of binary op(+) template

    cmp121->var2 = (B_Int0 *) malloc(sizeof(B_Int0));
    cmp121->var2->var1 =
(cmp21121->var1->var1) + (cmp21121->var2->var1);

    //////////// End of binary op (+) template
//////////// End of Comp

//////////////End of Alltup

    /////////////// Start of binary op(+) template

    mapo21 = (B_Int0 *) malloc(sizeof(B_Int0));
    mapo21->var1 =
(cmp121->var1->var1) + (cmp121->var2->var1);

    //////////// End of binary op (+) template
//////////// End of Comp

    (&output->var1[loop21])->var1 = mapo21->var1;
    free(mapo21);
    }

    //////////////////End of map

//////////// End of Comp

    gettimeofday(&endtime, &dummy);
    printf("\nFinished...\n");
    long timeElapsed = ((endtime.tv_sec - starttime.tv_sec) *
1000000L) + (endtime.tv_usec - starttime.tv_usec);
    printf("Time elapsed: %li\n", timeElapsed);
    if(PRINT)
    {
    printf("[");
    int prloop1;
    for (prloop1 = 0; prloop1 < output->size; prloop1++)
    {
    printf("    ");
        printf("%i", (&output->var1[prloop1])->var1);
    printf("");
    }
    printf("]");
    free(output);
    }
    }
```

B.2 Optimising Compiler Output

```
////example3.bmf
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/time.h>


#define TRUE 1
#define true 1
#define false 0
#define FALSE 0
```

```
#define PRINT 0
//#define DEBUG

#ifdef DEBUG
#define dprint(x,y) printf(x,y)
#else
#define dprint(x,y)
#endif

typedef struct _Vint
{
    int *contents;
    int size;
} Vint;

typedef struct _Vfloat
{
    float *contents;
    int size;
} Vfloat;

typedef struct _Vdouble
{
    double *contents;
    int size;
} Vdouble;

typedef struct _Vbool
{
    int *contents;
    int size;
} Vbool;

typedef struct _Vvoid
{
    void *contents;
    int size;
} Vvoid;

int main(int argv, char **argc)
{
    long loop;
    long tempsize;
    if(argv < 2)
    {
        printf("no argument found\n");
        exit(-1);
    }

    FILE *fd = fopen (argc[1], "r");
    if(fd == NULL)
    {
        printf("error opening input file");
        exit(-1);
    }
    struct timeval starttime;
    struct timeval endtime;
    struct timezone dummy;

    char* rinputfile;

    fseek(fd, 0, SEEK_END);
    long filesize = ftell(fd);
    //printf("filesize: %li",  filesize);

    fseek(fd, 0, SEEK_SET);
    rinputfile = (char *) malloc(filesize + 1);

    int numread = fread(rinputfile, 1, filesize, fd);

    rinputfile[filesize] = '\0';

    //printf("read data: %i\n", numread);
    //printf(rinputfile);
    char* current;
    char* point;

    current = rinputfile;


    int var0;
    Vint *var1 = (Vint *) malloc(sizeof(Vint));
    Vint *var2 = (Vint *) malloc(sizeof(Vint));
    Vint *var3 = (Vint *) malloc(sizeof(Vint));


    var0 = strtol(current, &current, 10);
```

```
printf("Starting...\n");
    gettimeofday(&starttime, &dummy);
////******B_comp
////******B_comp
////******B_alltup
////******B_id

////******B_comp
////******B_alltup
////******B_id

    ((Vint *)var1)->size = var0;


    //distl


    //distl

    ((Vint *)var2)->size = ((Vvoid *)var1)->size;
    ((Vint *)var2)->contents =
(int *) malloc(sizeof(int) * ((Vvoid *)var1)->size);
    ((Vint *)var3)->size = ((Vvoid *)var1)->size;
    ((Vint *)var3)->contents =
(int *) malloc(sizeof(int) * ((Vvoid *)var1)->size);
    int mapLoop21;
    for(mapLoop21 = 0;
         mapLoop21 < ((Vvoid *)var1)->size; mapLoop21++)
    {
////******B_comp
////******B_alltup
////******B_comp
    //addr

    //addr

////******B_comp
////******B_alltup
    //addr

////******B_comp
    //addr
    //addr


    (var2)->contents[mapLoop21] = var0 + var0;


    (var3)->contents[mapLoop21] =
mapLoop21 + (var2)->contents[mapLoop21];
    }
    gettimeofday(&endtime, &dummy);
    printf("\nFinished...\n");
    long timeElapsed = ((endtime.tv_sec - starttime.tv_sec) *
1000000L) + (endtime.tv_usec - starttime.tv_usec);
    printf("Time elapsed: %li\n", timeElapsed);
    if(PRINT)
    {
    printf("[ ");
    int output0loop0;
    for(output0loop0 = 0;
        output0loop0 < ((Vint *)var3)->size; output0loop0++)
    {
    printf("%i\t", (var3)->contents[output0loop0]);
    }
    printf(" ]\n");
    }
}
```

B.3 Hand Coded Program

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/time.h>

#define PRINT 0

int main(int argv, char **argc)
{
    long loop;
    long tempsize;
    if(argv < 2)
    {
        printf("no argument found\n");
        exit(-1);
    }

    FILE *fd = fopen (argc[1], "r");
    if(fd == NULL)
    {
        printf("error opening input file");
        exit(-1);
    }
    struct timeval starttime;
    struct timeval endtime;
    struct timezone dummy;

    char* rinputfile;

    fseek(fd, 0, SEEK_END);
    long filesize = ftell(fd);
    //printf("filesize: %li",  filesize);

    fseek(fd, 0, SEEK_SET);
    rinputfile = (char *) malloc(filesize + 1);

    int numread = fread(rinputfile, 1, filesize, fd);

    rinputfile[filesize] = '\0';

    //printf("read data: %i\n", numread);
    //printf(rinputfile);
    char* current;
    char* point;

    current = rinputfile;

    int input;
    int *output;

    int i;


    input = strtol(current, &current, 10);
    printf("Starting...\n");
    gettimeofday(&starttime, &dummy);
    output = (int *)malloc(sizeof(int) * input);

    for(i = 0; i < input; i++)
    {
        output[i] = input + input + i;
    }

    gettimeofday(&endtime, &dummy);
    printf("\nFinished...\n");
    long timeElapsed =
        ((endtime.tv_sec - starttime.tv_sec) * 1000000L) +
        (endtime.tv_usec - starttime.tv_usec);
    printf("Time elapsed: %li\n", timeElapsed);

    if(PRINT)
    {
        printf("[ ");
        for(i = 0; i < input; i++)
        {
            printf("%i\t", output[i]);
        }
    }
    free(output);
}
```

Bibliography

[Ale04] Brad Alexander. Automated transformation of bmf programs. In *Proceedings of the 1st International Workshop on Object Systems and Software Architectures*, pages 11–14, 2004.

[Ale05] Brad Alexander. *Compilation of Parallel applications via automated transformation of BMF Progams.* PhD thesis, School of Computer Science, University of Adelaide, 2005.

[Bac78] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, August 1978.

[BdM96] R. Bird and O. de Moor. *The Algebra of Programming.* Prentice Hall, 1996.

[Bir87] Richard Bird. A calculus of functions for program derivation. Technical Report 64, Programming Research Group, Oxford University, 1987.

[HB85] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 300–314, New York, NY, USA, 1985. ACM Press.

[JHH$^+$93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 93.

[Mee86] Lambert Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334, 1986.

[WB94] Clifford Walinsky and Deb Banerjee. A Data-Parallel FP Compiler. *Journal of Parallel and Distributed Computing*, 22:138–153, 1994.