Optimizing a language of iterative system solvers

Robin D. Newton

September 11, 2000

Contents

List of Tables

List of Figures

1 Introduction

1.1 Preliminary remarks

The computer program – in its source form – has an ambiguous status, being a sensuous object that is also a pure abstraction. It is the outward, visible form of an inward, invisible algorithm. The nature of this form is often considered to be a trifling matter: so one may say that C and Pascal are essentially the same language, that Java is just Smalltalk with C++ syntax, and that in any case all computer languages are ultimately equivalent in as much as they have the same computational power as a Turing Machine.

Yet there is a sense that there is something of great importance in programming that is not a pure formality. It may be said that programming is an art rather than a science [17], and that software engineers should strive to produce beautiful – elegant – programs [12]. At a more mundane level, programmers often have strong opinions on such seemingly trivial matters as how source code should be indented. It would seem that a computer program has its own delights and discords. As in scholastic philosophy where beauty was analyzed in terms of *integritas sive perfectio*, *proportio sive consonantia*, and *claritas*[1] [10], so a modern programming sensibility has its own watchwords: simplicity, clarity, generality [15]. The implication is that beauty is a matter with cognitive (not just emotional) impact; a good program should be pervaded by a consistent (and appropriate) style.

Programming – being a branch of engineering – involves making trade-offs when balancing conflicting considerations. An obvious example would be the space/time trade-off in memoisation: if the result of a calculation is stored then time can be saved if the result is re-used, but the calculation is re-done on demand then this can save space needed for storage. It seems fair enough that one is forced to make this choice. However, it is not so clear cut that programmers should have to choose between clarity and efficiency in the code they write; the opposition between these goals seems to have no necessity, and a suspicion is that is that arises due to flaws in programming languages.

In this project I attempt to create a language suited to writing one particular type of program (iterative system solvers) in which it is convenient to express their algorithms; and to write a compiler for it that takes advantage of the nature of the language, performing optimizations without as much of the heavyweight analysis that these would require in the setting of a general-purpose programming language.

1.2 What is language?

One extreme of language is where – in its performance – it simply acts, without representing: things are done with words. In a marriage ceremony, words spoken ("I now pronounce you man and wife") do not represent anything beyond themselves, but are actions that are done by being said. In the jargon of analytic philosophy they are said to have illocutionary force.[2]

I think the situation is similar with a language of commands in a computing context. Consider the Unix language of files – `open`, `close`, `read`, `write`, `seek` – a *lingua franca* for data transfer. The command `read` performs a read, and

[1] Integrity or perfection, proportion or consonance, and clarity

there is not much more than can be said about it. With such a vocabulary there are no combining forms beyond simple sequences of commands. There is a wider context, of course: a C program will contain more than just file operations, and can control them with a more complex logic of its own; the operating system relates different file commands in its 'plumbing' of data channels that link the things that write data to the things that read it. One could say that the file operations do not form a language in themselves, but since they seem to form a coherent unity I would rather say that they form a sub-language that operates in a larger context. Talking of a protocol or an API rather than a language of primitive operations to my mind changes nothing.

The other extreme is where language is purely representative, but in itself does nothing. In a programming context, this is the domain of data structures. The issue of how data represents its object is important in all kinds of programming.

In the context of traditional imperative programming, Fredrick Brooks has said that breakthroughs in program development usually come through changes in data representation rather than through modification of algorithms *per se*, and that a program's use of representation – more that its control flow structures – is where the heart of a program lies: "Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I usually won't need your flowcharts; they'll be obvious."[5]

In functional programming it is traditional to regard recursive data-types as the abstract syntax of a mini-language – in effect a term algebra. Pure expressions are impotent; to achieve anything at all they have to be interpreted (for example the meaning of terms in lambda calculus is determined by an external reduction relation). But since expressions are just data they can be manipulated by computer programs. And since programs are themselves expressions in a particular language, the tendency is to regard interpretation and computation as virtually synonymous: so "The interpreter for a computer language is just another program," and again "Perhaps the whole distinction between program and programming language is misleading."[11]

In reality programming exists between these extremes. A program can be regarded as a data object – an expression in the term algebra defined by the language it is written in. But a program also has a meaning that a compiler or interpreter must respect. Knowledge of the language in question allows the program to be transformed without its meaning being changed. This is only possible because the language definition is fixed and available to the compiler writer.

I suspect that the conflict between clarity and efficiency arises because when a program is written a language is created for solving the problem at hand. Creating a module with procedural interface creates a language of commands; selecting a data representation creates a language of expressions (and with abstract data types and object-orientated programming these processes are combined). This can make a program clearer because it is expressed in a language moulded to its own needs, but the language created is not known to the compiler writer, so code optimization cannot be based on the particular nature of this special-purpose language. Also, procedural abstraction and run-time interpretation of data inevitably affect performance. To overcome this a programmer is forced to 'unpack' his little language by hand, to produce code that – like the language of legal documents – is hard to read because it is completely explicit.

1.3 BLAS

This project is based around the idea of a mini-language tailored to the creation of a type of numerical program called an *iterative system solver.* In such a program a solution to a particular numerical problem if found by iteratively taking small steps through a solution space until the process converges on an answer. The problem and the sequence of approximate solutions take the form of matrices and vectors.

To make the problem area more specific, I decided to use as a point of reference a number of programs that implemented iterative system solvers which were written by Olav Beckmann as benchmarks for his distributed linear algebra library. They were written in C, but made use of routines from the BLAS library.

BLAS stands for Basic Linear Algebra Subroutines, and defines a set of very basic linear algebra operations that might be used in a typical numerical algorithm [9]. The idea was that given a standard interface, programmers could use BLAS routines to perform common operations within their programs, and computer vendors could create implementations that they would optimize for particular platforms.

One point to make about the relation between BLAS operations and linear algebra operators is that BLAS works in an imperative manner. In linear algebra matrices and vectors are composite values; mathematical objects that do not change no matter what is done with them. BLAS works in the context of Fortran or C, so vectors and matrices are really chunks of storage in a computer memory that hold values, and operations on vectors and matrices often work by changing the value stored. For instance, if α was a scalar value and $\vec{x}$ was a vector then their product – $\alpha\vec{x}$ – would be a vector different from $\vec{x}$ [2]. An equivalent operation using BLAS might use the double-precision vector scaling routine `dscal`:

```
dscal(n,alpha,x,1);
```

But here the vector `x` is actually an array whose elements will be multiplied by `alpha` in place.

Therefore, although BLAS may be used in programs that evaluate linear expressions, the BLAS interface itself provides a language of commands rather than a language of expressions.

BLAS is divided into three 'levels'. The levels refer to the amount of loop nesting required in naive implementations of the routines. Multiplying two matrices together requires a routine with a triply nested loop (one loop inside another, itself inside a third), so matrix multiplication is done by Level 3 routines.[3] On the other hand finding the dot product of two vectors can be done with a single loop, so is performed by a Level 1 routine. Level 2 routines can be implemented with doubly nested loops. Another way of making this distinction is to say that Level 1 routines perform operations only on vectors and have linear execution time, Level 2 routines perform matrix-vector operations and have quadratic execution time, and Level 3 routines perform matrix-matrix operations and have cubic execution time.

[2] Unless the product happened to be equal to $\vec{x}$

[3] If a loop nest is tiled then this will increase the level of loop nesting, and this may well be done with actual implementations of Level 3 BLAS routines; this is why I referred to *naive* implementations.

2 Possible approaches

2.1 Partial evaluation

Programming languages can be interpreted or compiled. It therefore follows that if data within a program is interpreted in the manner of a language then it could potentially be compiled. This is the promise of partial evaluation.[13]

If the data that is interpreted by a program is known, then it is possible to do the work of interpretation statically to produce a specialized program that only contains the actions that are called for by the interpretation. This idea is clearest when the data being interpreted is a computer program in the most literal sense; the specialized interpreter would be a translation of the data program's meaning with no residual interpretive code.

An example that is often given is that of the standard C printing routine printf. This takes as its first argument a string that defines the format of its output. Given the following call in a program text:

```
printf("str = %s   var = %f", str, var);
```

the formatting string could be statically interpreted to give the following residual code:

```
print_string("str = "); print_string(str);
print_string("   var = "); print_double(var);
```

(where `print_string` and `print_double` are suitably defined routines). If this is done then there is no need for the formatting string to be interpreted at run-time, every time the code is executed.

All of this depends on a separation between the work of finding out what should be done and the work of actually doing it. The former is referred to in partial evaluation literature as the *interpretive overhead*, and the assumption is that this contributes a significant proportion of the running time of many programs; hence reducing (or eliminating) this overhead can provide a noticeable performance boost.

Andrew Berlin has described a system that uses partial evaluation to allow numerical programs to be written using the data-structuring and abstraction facilities of a modern computer language (in his case Scheme), without suffering a reduction in performance.[4] This worked by using a modified Scheme interpreter that would execute a program 'symbolically', using 'place-holders' to represent values that would not be known until run-time, and creating residual code when evaluating expressions that could not be reduced fully until these values were known. The result would be code that only contained the computational component of the original program; all the interpretive overhead created by the use of abstractions would have been eliminated.

However none of this is of much help here. The reason for this is that BLAS-style operations are heavyweight (whereas the residual operations in [4] only involve scalar values). One of the ways of identifying interpreter-like aspects of programs that is sometimes give is to say that a program is interpretive to the extent that the flow of control is governed by the data. The implementation of a BLAS routine may interpret its arguments to a certain extent – some routines may check a scalar multiplier and have special case code for when it is 1, say. But if the sizes of vectors and matrices is generally large (and there is not much interest in trying to improve performance if they are not) then most of the execution time will be spent in very straightforward loop nests where the flow of control is very predictable indeed.

In [4] an important distinction is made between data-dependent and data-independent computations. Numerical computations are for the most part data-independent, which is to say that sequences of operations in a program are not altered by the run-time values that they process. When a program is data-dependent – containing branches conditional on run-time values – then the data-dependence is confined to small regions (such as tests for convergence).

2.2 DESO

Olav Beckmann has created an implementation of a BLAS-like library suitable for use on a distributed computing system [3]. To increase performance in such a system it is necessary to consider how data is to be distributed across nodes in a system to minimize the amount of communication required. This is obviously a rather different matter to simply eliminating interpretive overhead. What is more interesting from my point of view is that it works within the context of a regular user program (in C or Fortran) that makes calls to library subroutines; in other words BLAS still has the appearance of being a language of commands.

One important advantage of a language of expressions over a language of commands is that an expression – being a data object – can be manipulated and transformed. When a program in an imperative language is compiled it is treated as a formal object, and the compiler is at liberty to alter it (for instance, by changing the order of statements) provided the overall effect of the program is in accord with the meaning of its source. The compiler can consider the interaction between statements, not just the effect of each of them individually.

Beckmann's parallel BLAS implementation creates scope for optimization by viewing groups of operations as forming an expression, but at a 'user' rather than a 'system' level (*ie* without the benefit of a compiler writer's special privileges). What happens is that in a program that performs a sequence of calls to BLAS routines, the routines do not immediately perform the desired operations but rather build up a structure that represents the operations that should be performed. At some point evaluation is 'forced', and the DESO system must do whatever is necessary to produce the same overall effect as if the operations had all been performed when their corresponding routines were called; however DESO is free to achieve the effect as it will. The optimizations that it does are mainly related to how the data being processed is distributed over a network of computers being used: it is difficult to make intelligent decisions about this when each operation is considered in isolation because no account can be taken of how the results of a single operation will subsequently be used.

There are two important points to make about this system. Firstly: the creation of both a graph that represents a sequence of BLAS calls and of an optimized execution plan happens at run-time. As well as meaning that account can be taken of state of a particular distributed computing system, this also means that there are no restrictions on the structure of the program using the library. A program can be straightforward or convoluted with regards the manner in which it makes its calls to BLAS routines; from the point of view of DESO system it is all as one: sequences of delayed operations are built up then evaluated. The second point – related to the first – is that the graphs of delayed operations represent an imperative language that lacks any control flow apart from sequential composition. If a program has to make a decision using the result of a BLAS operation then evaluation would have to be forced in order to make the result available before the decision could be made, and this process would be external to the language of sequenced BLAS operations itself.

The initial plan for this project was to do something very similar to this DESO system, but aimed at a uniprocessor machine rather than a distributed system. However, I decided that using lazy evaluation and run-time optimization in this way would be an unhelpful complication. In a distributed system it is easy to imagine how factors not known until run-time (levels of network traffic, loading of individual nodes) could impact on the choices made in execution strategy. But in a uniprocessor system I do not see this as being the case: most of the relevant factors can be known statically.

2.3 Uniprocessor loop optimizations

It is the nature of BLAS-style routines that they perform simple arithmetic operations on large amounts of data, so improving performance tends to be a matter of optimizing the way the data is accessed. In a distributed system the overhead of accessing data is dominated by the cost of sending data over a network from one node to another, and so optimization consists of finding an execution plan that minimizes these costs. In a uniprocessor system data is stored in memory, but this memory is organized into a hierarchy (caches, main memory, virtual memory) with the levels towards the top providing fastest access but having least capacity. Here improving data access time means organizing execution so that when a memory location is used repeatedly the uses are brought close together, with the intention that as far as possible data being accessed will already be held high in the memory hierarchy.

The core part of the implementation of a BLAS-style operation is a loop that in each iteration performs the part of the overall computation that goes with part of the data the operation is given; over all the iterations all the given data is covered. For example, a simple implementation of `dscal` may have each iteration multiply one element of the given vector by the given factor; a more complex implementation may unroll this loop and so multiply several elements in the vector in each iteration – but overall both implementations will multiply every element in a vector. In the body of a loop there is normally a simple relation between the particular elements in vectors and matrices being accessed, and the values of loop induction variables. In this project I have based the loops that implement BLAS-style operations on the reference BLAS implementations [19], but with no loop unrolling at all, to make the loop bodies as simple as possible. Loop unrolling could always be performed later.

So, given a program that is made up from a number of this sort of loop, reorganizing execution is a matter of reordering the execution of iterations of loop bodies. The way this is normally conceived is that a loop nest gives rise to an 'iteration space', with a point in this space for each iteration of the innermost body (which is to say a point of each combination of values that the loops' induction variables can take on). Loop interchange and tiling modify the order in which the iteration space for a loop nest is traversed; loop fusion superimposes the iteration spaces of a number of non-nested loops. All such changes must preserve the meaning of a program: if there is a pair of points in a loop nest's iteration space such that one of the points must be traversed before the other (due to a dependence relation) then any transformation must honour this in order to be valid.

Loop interchange involves changing the order in which loops are nested; in terms of an iteration space, if the space was square then it would be like traversing it row at a time rather than column at a time. Loop tiling cuts the iteration space into rectilinear tiles, then traversing the space tile at a time. Loop fusion is different because it involves combining multiple iteration spaces; if two loops were to be fused then instead of performing all the iterations of the first loop followed by all the iterations of the second, the first iterations of both loops would be executed, followed by the second iteration of both loops *etc.*

These are the standard techniques of restructuring compilers that are used for high performance computing [21]. They are not of equal interest for the purposes of my project. The reason for this is that standard BLAS implementations (of the sort that computer vendors supply) can already optimize (automatically or by hand) individual BLAS routines, but the one option that is open to me but not to them is to optimize sequences of BLAS routines. The Level 1 routines (which are the ones most commonly used in the benchmark programs) are implemented with single loops, so the scope for loop fusion with a sequence of Level 1 operations is apparent. But I do not believe there are likely to be many cases where there are profitable opportunities for loop interchange and tiling when sequences of operations are considered that would not be apparent when the operations were looked at in isolation.

Tiling is most beneficial in something like matrix-matrix multiplication, where each element in the matrices being multiplied is accessed many times. One might think that tiling could be left until after all possible loop fusion had been done, but I doubt that fusion is likely with the sort of loop nests that are usefully tileable. Consider the following sequence of operations (where `A`, `B`, `C`, `D`, `E` are matrices):

```
(1) A := B * C
(2) D := A * E
```

This can be implemented using two loop nests of depth 3, one for each multiplication. But these cannot be fused, because the result of the first multiplication must be available in its entirety before the second multiplication can begin.

3 Proposed approach

3.1 Features of the target problems

I observed that the form taken by these iterative system solvers is as follows. The variables used for the actual data being processed are matrices, vectors and scalars, but the actual numbers stored in these are of the same type (double precision reals). The contents of some variables are initialized (so as to represent the system being solved, plus a small amount of processing). A loop whose body consists of a sequence of BLAS operations is then performed a large number of times. Then (possibly after another small amount of processing) the results are presented. In the body of the loop some of the operations performed in the first iteration are not the same as those performed in subsequent iterations, and in some cases error conditions (such as particular scalar variables becoming too small) are tested and cause the program to terminate early; apart from this, control in the loop body flows in a straight line.

Beyond this, the matrices and vectors used to store data being processed, and the way in which BLAS routines were called, also showed noticeable regularities. An instance of a problem would be characterized by its ‘size’: all vectors would be of this size, all matrices would be square and have side lengths of this size. Also, in BLAS it is possible for vectors to be something other than a vector represented directly by a variable in the source program: it could be a row or column of a matrix, or part of a larger vector. In these benchmark programs this never happens: the vectors passed to BLAS routines are vector variables, the matrices are matrix variables, and the dimensions passed are all equal to the problem’s size. The operations themselves are for the most part Level 1; a Level 2 operation that sometimes crops up is `dgemv` – matrix-vector multiplication.

3.2 Loop fusion

The focus for optimization has to be the main loop of a iterative system solver; this is where most of the time will be spent. The prospect of having the body of this loop consist of a linear sequence of Level 1 BLAS operations – and these obeying the restrictions described above – makes the loop fusion look particularly straightforward. These operations can be implemented as single loops with one iteration for each position in the vectors. Consider he following example of a dot product followed by a scaling operation (written in C) that illustrates the simplicity of fusion:

```
/* prod := v.u */
prod = 0.0;
for (i = 0; i < SZ; i++)
  prod += v[i] * u[i];

/* v := alpha*v */
for (i = 0; i < SZ; i++)
  v[i] = v[i] * alpha;
```

These loops can be fused to produce the following:

```
/* Fused loop */
prod = 0.0
for (i = 0; i < SZ; i++)
{
  prod += v[i] * u[i];
  v[i] = v[i] * alpha;
}
```

In this example, for each value of `i` the value initially stored at `v[i]` is read twice. A redundant load from memory can be eliminated if the value of `v[i]` is stored in a temporary scalar variable at the beginning of the loop body; this called ‘scalar replacement’.

3.2.1 Legality

For the fusion of two loops to be legal there are a number of conditions that must be satisfied.

Loops must be adjacent if they are to be fused; there must be no statement that necessarily comes after the first but before the second. If the loops are adjacent in the program text then it is easy to see that this is the case. If there is another statement in between the loops then the partial order that is defined by the dependency relations on statements shows whether the intermediary statement creates a real separation between the loops, or whether it would be legal to reorder statements so as to make the loops physically adjacent.

Loops to be fused must also be compatible: the loops must have an equal number of iterations. In general this means that an analysis of the loop's induction variables will show the loops to have the same trip counts, but with no requirement that the loops use the same induction variable, or have their induction variables ranging over the same set of values. However, in the class of programs I am dealing with in this project *all* single loops in a program are compatible in this sense because they all have their induction variables running over the same range (fixed by the problem size), and in the same order.

It must also be the case that the fusion of two loops does not result in any backward-running loop-carried dependency. A loop-carried dependency is one where the dependency is between executions of statements in different iterations of a loop. If such a dependency ran backwards - for example, if a statement in one iteration needed to use a value produced in the following iteration - then the actual execution order (forwards) would violate the dependency.[4]

Here is an example of two loops that cannot be fused (as they stand) for this reason:

```
for (i = 1; i < 6; i++)
  v[i] = alpha * v[i];

for (i = 0; i < 5; i++)
  w[i] = v[i+1];
```

The trip count is the same for both loops, but if they were fused then in the first iteration the statement from the body of the second loop would use a value not calculated until the second iteration, *etc.* However, this sort of situation can never arise in the class of programs for which this project caters. When a loop accesses a vector (reading or writing) it always does it in the same fashion: first element in the first iteration, second element in the second, *etc.* This continues to be true when loops are fused, so it can never be the case that an element of a vector could be accessed in more that one iteration of a loop: fusion can introduce no loop-carried dependencies (forward or backward) that relate to vectors.

So vectors cannot cause loop-carried dependencies that would prevent fusion, and matrices never occur in single-loop operations; this leaves scalar variables as a possible source of trouble. Consider the following:

```
prod = 0.0
```

[4]In some cases this problem can be avoided by reversing the order in which the loop is executed, but that is not useful here - as will be seen shortly.

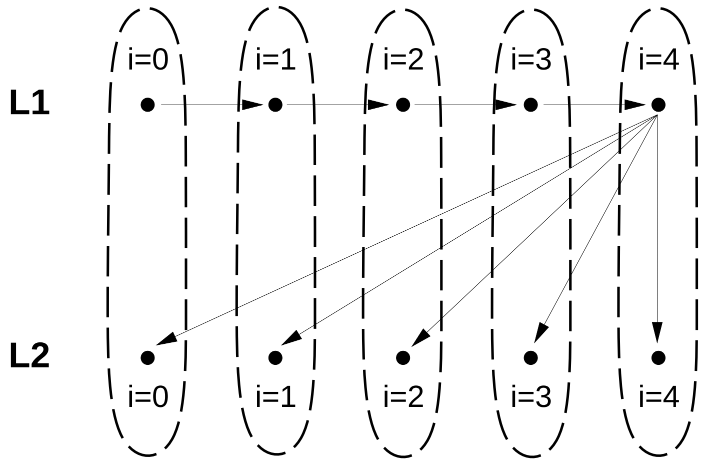

Figure 1: Iteration spaces of unfuseable loops

```
for (i = 0; i < SZ; i++) /* L1 */
  prod += v[i] * u[i];

for (i = 0; i < SZ; i++) /* L2 */
  w[i] = w[i] * prod;
```

The two loops cannot be fused because every iteration of the second loop needs a value that is not produced until the the last iteration of the first loop.[5] The iterations spaces for these loops it shown in Figure 1 (with SZ = 5). If the loops were fused (as shown by the dashed lines) then it would be impossible to satisfy all the dependencies: the iteration $\texttt{i} = 4$ depends on the the iteration $\texttt{i} = 0$ (due to the dependency carried by the first loop) but the iteration $\texttt{i} = 0$ depends on the iteration $\texttt{i} = 4$ (because of the second loop's dependency on the outcome of the first).

Instead of going to the trouble of explicitly analyzing the iteration space of a fused loop in search of backward dependencies, it is enough to say that loops cannot be fused if there is a dependence between them (direct or indirect) that is mediated by scalar variables).

In the literature it is conventional not to talk of dependencies between loops, but rather between the statements contained in loops (or even more precisely, between the variables references in statements). In this project dependencies are analyzed at the level of sub-BLAS operations, many of which are understood as 'really' being single loops; but these 'loops' are not explicit, and the statements in their bodies are not available until code-generation phase. Hence it is convenient for me to talk about dependencies as being between loops.

[5]It is immediately apparent that reversing the first loop will not help here. One can say the the first loop cannot be reversed because that would violate the loop-carried dependency that it contains in its own right. Or one can say that the first loop can be reversed (as it is a summation, and addition is associative and commutative) but that is still the value produced by the last iteration actually executed that is used by second loop.

```
((declarations
   (A)
   (b x p q r res z)
   (alpha beta bnrm2 err rho rho_o))
 (setup
   (initialise A x b)
   (:= rho_o 1.0)
   (dcopy b r)
   (dgemv "T" -1.0 A x 1.0 r)
   (:= bnrm2 (dnrm2 b))
   (:= bnrm2 (nonzero bnrm2)))
 (loop (dcopy r z)
   (:= rho (ddot r z))
   (first-time
     ((dcopy z p))
     ((:= beta (/ rho rho_o))
      (daxpy beta p z)
      (dcopy z p)))
   (dgemv "T" 1.0 A p 0.0 q)
   (:= alpha (/ rho (ddot q p)))
   (daxpy alpha p x)
   (daxpy (- alpha) q r)
   (:= rho_o rho)
   (:= err (/ (dnrm2 r) bnrm2)))
 (finish (dcopy b res)
   (dgemv "T" 1.0 A x -1.0 res)
   (:= err (dnrm2 res))
   (results b x)))
```

Figure 2: Source code for Conjugate Gradient Descent algorithm

3.3 The language

The appearance of my language is best explained with an example. In Figure 2 I have show the implementation of the Conjugate Gradient Descent benchmark in this language.

First there is a declaration of all the variables used (matrices, then vectors, then scalars). Then there are three sections (setup, loop, finish) that contain sequences of statements. The statements are written lisp-style: a space-separated list enclosed in brackets, with the operator as the first element. In the main loop 'first-time' is a control-flow operator which takes two sequences of statements; it indicates that on the first iteration of the loop the first sequence should be executed, but on subsequent iterations it should be the other sequence.

Some of the statements assignments of expressions to scalar variables, and some of them can be thought of as being like procedure calls. So the statement `(:= err (dnrm2 res))` has `err = dnrm2(SZ, res, 1)` as its rough C equivalent.[6] The statement `(daxpy (- alpha) q r)` corresponds with the statement

[6]The C version contains extra arguments to the function call because the size and stride of the vector argument are not know implicitly.

`daxpy(SZ, -alpha, q, 1, r, 1)` in C.

Scalar expressions can contain scalar variables, arithmetic operators, BLAS function calls (such as to `ddot`) and other function calls (such as to `sqrt`, or `sin`).

Most other statements correspond to calls to BLAS subroutines, but there are two that do not: `initialise` and `results`. These represent how data is initially brought into the iterative system solver, and how the results are exported. The idea is that in the end the system solving program will be compiled into C then object code, then linked into a larger program written by the user. The larger program should contain definitions of procedures `initialise` and `results` that create the initial data and displays results in any way the user sees fit. From the current point of view all that matters is that these operators take an arbitrary number of variables (of any type), and that `initialise` will write the contents of all variables passed to it whereas `results` will read the contents of all variables passed to it.

3.3.1 Absences

There are a number of features that might be expected in this program (and the language in general) but which are not present.

The size of the problem that the program solves (which determines the sizes of vectors and matrices) is not given; most – if not all – of the analysis of this program can be done in ignorance of this, so this need not be set until later. And, given that the same system solving algorithm can be used on problems of different sizes, it might be reasonable only to set this size at run time.

There is also no indication given as to how often the loop body should be iterated. In the C implementation of this algorithm there is a fixed number of iterations that is set at run time (via a command line argument to the program). A similar approach would work in my system, with the number of iterations being set at the same time as the problem size. It would be better if the conditions for terminating the loop were explicit in the program; then it would be possible for the loop to be iteration just until it had converged towards a solution sufficiently closely (according to some criterion). This cannot be expressed in the language described above, but I see no reason why in principle that would always have to be the case. The loop form could be modified to have a condition expression, like a traditional while-loop. If the conditions that decided termination were considered to be too awkward (for whatever reason) to be expressed in the language of this system then an externally defined function could be used (much like with `initialise` and `results`) as long as it had no side-effects visible from the system solving program. I do not believe that any of this would complicate the analysis needed to do loop fusion.

Another missing feature that would be needed to implement the Biconjugate Gradient Descent and Quasi-Minimal Residual benchmarks is the ability to exit the program itself with an error condition (indicating the method being performed by a program is failing). The only error conditions that feature in the benchmarks concern particular scalar variables becoming too small; I imagine that the reason for this is to prevent overflow errors where these variables are used as divisors in arithmetic expressions. I think it would probably be straightforward to add a new statement type to do this job. This would complicate the flow of control and possibly inhibit scope for optimization by making decreasing the size of basic blocks. However, given that what is being discussed here is early termination with an error – rather than early successful completion – then I suspect that this could be ignored for the most part (as there is a reduced need for program transformations to preserve exact semantics in the face of a fatal error). So, if a variable is assigned then checked for failure, no reordering of statements is prevented provided that the check is kept between the assignment and the subsequent uses of the variable. In any case, no loop fusion would be prevented as dependencies through scalars prevent fusion (for reasons given earlier).

In the example program given above, the statement `(:= bnrm2 (nonzero bnrm2))` uses a function `nonzero` which I define return its argument unchanged except when its argument is 0 when in returns 1. This is a rather ad hoc way of getting around the fact that the language does not feature the normal, useful control constructs of typical programming languages. In the C version of the benchmark, the equivalent for this statement is:

```
if (bnrm2 == 0.0)
  bnrm2 = 1.0;
```

The reasons for this are that it would be awkward to represent in the structures I have chosen to use in my system (which I will describe later), and because this sort of facility is of marginal importance in the system. In this project one of the points of the exercise is to see what can be done by avoiding implementing a full-blown, general-purpose programming language; it is inevitable that a deliberately restricted language will lack features. Working around these limitations in the manner given above is inconvenient but effective; it is simple to assume that any unrecognized functions must be defined externally. Normal control-flow constructs would only be useful in the `setup` and `finish` sections (if they were allowed in the `loop` section it would change the basic character of the language), and there is a case for saying that these two sections could be reduced simply to calls to `initialise` and `results`, and these externally defined routines could do any processing required before and after the execution of the main loop. But this would disperse the logical unity of the algorithm expressed in the program.

4 Implementation

The overall plan is fairly obvious. Parse the program text in order to produce an abstract syntax tree. Analyze this tree in order to create the graph of a flowchart that contains the program's basic operations and structure. Optimize the program by manipulating it in this graph form. Then generate executable code from the graph. The stages are broadly what one expect in a compiler; but they are simpler, reflecting the narrow interest of this project.

4.1 Parsing

I decided that programs should be represented in the manner of a lisp-style s-expression for convenience. My system is implemented in Scheme – a lisp variant – and so a program in my language appear in a Scheme source file as a quoted literal. The rather mundane matter of parsing a text file (reading in characters, tokenising, *etc*) is handled by the Scheme implementation. The most low-level form in which a program is encountered in my system is as an s-expression.

A program in the form of an s-expression is virtually an abstract syntax tree already, so I quite possibly could have done without this stage. However, I found it convenient when writing the code converts a program to flowchart form for the nodes in the tree to be explicitly tagged as to their particular nature. This extra level of explicitness would not be appropriate for the the source language.

4.2 Conversion to a flowchart

A program is converted into a flowchart. This flowchart will be manipulated so as to optimize the program, then used as the basis for code-generation at the last stage. A flowchart object contains fields for declarations of the variables used in a program, program nodes (which is to say basic program operations), a flow relation which connects a node with its successors, and a set of labels that mark out the basic blocks of the program.

A node object contains fields for an instruction and its arguments, together with other fields that hold information used in the data flow and loop fusion analyses.

The program is split up into four basic blocks. The `setup` and `finish` sections become basic blocks. The program's main loop – which in the source was allowed to contain statements whose execution was conditional on an iteration being the first iteration or not – becomes two blocks: one representing the body of the loop in the first iteration and one representing it in subsequent iterations. Duplicating the body like this means that each version of it is a straight-line sequence of commands.[7] The blocks are referred to as `setup`, `finish`, `first-loop` and `rest-loop`. The program's flow relation – as well as indicating that the program steps through each nodes sequentially – connects the end of `rest-loop` back to its own beginning (showing that it can be executed repeatedly), the end of `setup` to the beginning of `finish` (showing that the loop might never execute), and the end of `first-loop` to the beginning of `finish` (showing that the loop might terminate after just the first iteration). See Figure 3.

There will be nodes in a flowchart which have multiple successors in the flow relation. For example the end of the `rest-loop` block leads back to its beginning and also to the beginning of the `finish` block. It might be expected that a node such as this would contain an indication as to when each of these routes might be taken; a test expression. That is not the case here; the way the flow of control would be governed at run time (such as counting the loop iterations up to a maximum number) is not represented in the flowchart structure. The reason for this is that – in the system as it currently is – this issue is always handled in the same way in for all possible programs of the language; if it is not subject to variation between programs then there is no point in including it in any representation of a program. The intention is to generate structured C, not a mass of `goto` statements, so the flow relation is not used in code generation

[7]Conceivably this trick could be used to allow other sorts of condition in the loop body, with a version of the body being created for each set of circumstances. However (a) none of the benchmark programs I looked at would need this, and (b) the circumstances tested have to be known at the beginning of an iteration (which is easy when the test is "Is this the first iteration?").

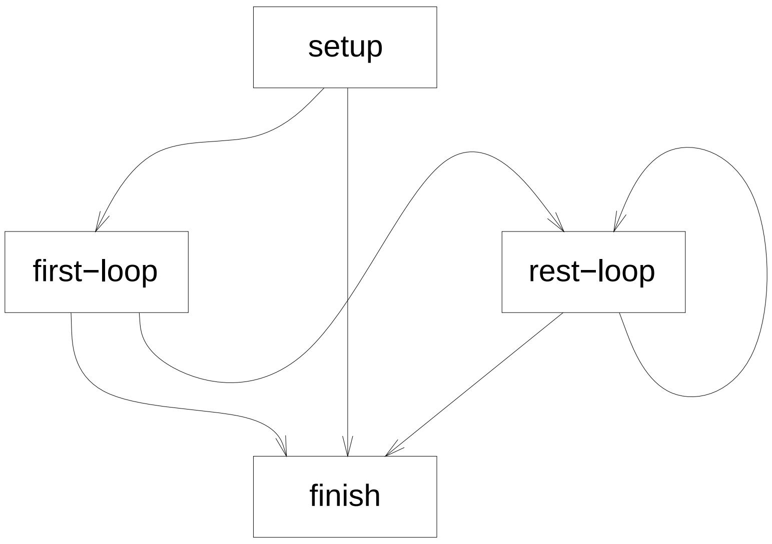

Figure 3: The basic blocks in a program flowchart

but only for data flow analysis.[8]

The nodes of the flowchart contain instructions correspond to the statements in the program source, but the correspondence is not one-to-one. Statements are broken up into primitive operations. The granularity of these primitives is not nearly as fine as would be the case in a typical compiler for a general-purpose language. As optimization will consist of the merging of single loops, the aim is to make all such loops visible – neither buried within larger operations nor implicit in the arrangement of smaller operations.

Statements representing calls to BLAS subroutines are broken up according to the loop nests of its reference implementation.

Take for example the matrix-vector multiplication routine `dgemv`. If the statement in the program was `(dgemv "N" alpha A x beta y)`, then the intended effect is that the vector y should be assigned with the product of the matrix A and the vector x - scaled up by a factor alpha - added to the old value of y - scaled up by a factor beta; or, more concisely, $\mathtt{y} \leftarrow \alpha\mathtt{A}\mathtt{x} + \beta\mathtt{y}$. If `"N"` was replaced with `"T"` then this would mean the multiplication would be between x and the transpose of A, *ie* $\mathtt{y} \leftarrow \mathtt{A}^{\mathtt{T}}\mathtt{x} + \beta\mathtt{y}$. The Fortran reference implementation splits this into two parts: the first part is a single loop that produces the effect $\mathtt{y} \leftarrow \beta\mathtt{y}$, the second part executes one of two double loops that produce the effect of either $\mathtt{y} \leftarrow \alpha\mathtt{A}\mathtt{x} + \mathtt{y}$ or $\mathtt{y} \leftarrow \alpha\mathtt{A}^{\mathtt{T}}\mathtt{x} + \mathtt{y}$.

So a `dgemv` statement in the program will result in two nodes in the flowchart. Concerning the second of these – the one that does the actual matrix-vector multiplication – it is assumed that the first argument to `dgemv` (the `"N"` or `"T"` that indicates whether is it the matrix or its transpose that it used) is a compile time constant; hence a choice can be made when the second node is being inserted into the chart between the two instructions that handle this multiplication in the two different cases. Concerning the first of the nodes – the

[8]This is the reason why it would be awkward to allow conditional statements in the `setup` or `finish` sections of programs.

one that scales the vector `y` by a factor `beta` – it seemed worthwhile to me that certain special cases should be detected: if the factor has the constant value 1.0 then this node can be omitted, if it is 0.0 then the node will contain an instruction set every element of the vector to 0.0, and if the it is −1.0 then the instruction will be one to negate every element of the vector. As an example, the statement:

```
(dgemv "T" 1.0 a x -1.0 res)
```

will produce the following nodes:

```
(55 neg (res))
(56 mvmul-t (#(lit 1.0) a x res))
```

[9] So `neg` is the instruction that negates every element in the vector given to it, and `mvmul-t` performs a multiplication between the transpose of a matrix and a vector.

Most BLAS operations are in the form of subroutines, which means that their results are seen in the changes made to the variables passed to them as arguments. Some BLAS operations are functions, which means they return a result and hence must appear in the program as part of an expression. Leaving an operation embedded in an expression would make it impossible to do the sort of optimization I am interested in here. The way around this that I have used is that for each call to a BLAS function a new scalar variable is created, the function is cast into a 'subroutine' form (taking the variable as an argument), and the variable is then used in the place where the function call was in the original expression. The subroutine form will itself be analyzed like any other BLAS subroutine call, and hence will be broken up into its constituent instructions. To give an example, the statement:

```
(:= alpha (/ rho (ddot q p)))
```

(*ie* `alpha = rho / ddot(q,p)`) will give rise to the following nodes in the flowchart.

```
(19 assign (tmp9 #(lit 0.0)))
(20 dot (tmp9 q p))
(21 assign (alpha #(aexpr / (#(var-ref rho) #(var-ref tmp9)))))
```

Here `tmp9` is the variable created to hold the result of the dot product, and `dot` is the instruction that increments this variable by the dot product of the two vectors given to it.

Another situation where it is convenient to create new variables is where a BLAS routine takes a scalar value as an argument. If this value is a literal number or a reference to a scalar variable there is no problem, but if is something more complex (a function call or an arithmetic expression) then some steps have to be taken to ensure that at run time the expression is not re-evaluated unnecessarily. It s easy enough to do that here, assigning the value of a complex expression to a new variable then passing that to the instructions that implement the BLAS operation.

[9]The format displaying a node used here is as a list of three elements: the first is the nodes position in the flowchart, then second is the name of the instruction, and the third is a list of its arguments.

instruction	argument types	effect
`(copy x y)`	vector, vector	$y \leftarrow x$
`(dot alpha x y)`	scalar, vector, vector	$\alpha \leftarrow \alpha + x \cdot y$
`(axpy alpha x y)`	scalar, vector, vector	$y \leftarrow \alpha x + y$
`(add-assign x y)`	vector, vector	$y \leftarrow y + x$
`(sub-assign x y)`	vector, vector	$y \leftarrow y - x$
`(scal alpha x)`	scalar, vector	$x \leftarrow \alpha x$
`(neg x)`	vector	$x \leftarrow -x$
`(zero x)`	vector	$x \leftarrow \vec{0}$
`(mvmul-n alpha A x y)`	scalar, matrix, vector, vector	$y \leftarrow \alpha Ax + y$
`(mvmul-t alpha A x y)`	scalar, matrix, vector, vector	$y \leftarrow \alpha A^T x + y$

Table 1: Sub-BLAS instructions

The possible instructions that can be contained in a flowchart are: `dummy` (which does nothing but is used for the labelled nodes at the beginning and end of basic blocks), `assign` (which takes the name of a scalar variable and a scalar expression as its arguments), `initialise` and `results` (which take arbitrary numbers of variables) and a large number of 'sub-BLAS' instructions. In the system in its current state, these are as given in Table 1. The instructions `add-assign` and `sub-assign` do not correspond directly with any BLAS routines, but are used for special cases of `daxpy` (which has the effect $\mathtt{y} \leftarrow \alpha\mathtt{x} + \mathtt{y}$) where the scaling factor as a constant value of 1.0 or −1.0 respectively.

Each node object, as well as holding the name of its instruction and a list of arguments, also contains lists of the variables that the operation is represents read from and writes to (which is used when data flow information is calculated) and the nature of the operation's potential for fusion.

An assignment – which assigns the value of an expression to a scalar variable – obviously writes to the variable it assigns and reads all the variables that occur in its expression. A dummy node neither reads nor writes anything. The instruction `initialise` writes to all of its arguments, and `results` reads all its arguments. The pattern of reading and writing done by the sub-BLAS operations depends on the particular operation in question. The function shown in Figure 4 is used when finding the variables written and read by a node with a particular instruction. The values `w`, `r` and `rw` are used to indicate that a particular argument in a variable that is written, read, or both (respectively), and `e` indicates that an argument is a scalar expression, in which case all variables occurring in the expression are considered to be read by the node.[10]

Assignments and calls to `initialise` and `results` cannot be fused with other nodes because they are not implemented as loops (likewise dummy operations). Operations like `mvmul-n` and `mvmul-t` are marked as unfuseable because they are implemented as double loops and my system currently only considers merging single loops. The other sub-BLAS operations are implemented as single loops, and hence are considered to be fuseable. A distinction is made between operations that are implemented with no loop-carried dependencies – and are therefore fully parallelisable - and ones which do – and are therefore inherently

[10]Other than in assignments, scalar expressions can only be literal value or references to scalar variables; more complex expressions would have been replaced by temporary variables in the process of creating the flowchart.

```
(define instr-arg-pattern
  (lambda (instr)
    (case instr
      ((dummy) '())
      ((assign) '(w e))
      ((mvmul-n mvmul-t) '(e r r rw))
      ((copy) '(r w))
      ((dot) '(rw r r))
      ((axpy) '(e r rw))
      ((scal) '(e rw))
      ((add-assign sub-assign) '(r rw))
      ((neg) '(rw))
      ((zero) '(w))
      (else (error "Unknown instruction" instr)))))
```

Figure 4: Analysis of instruction read/write patterns

sequential; these cases are demonstrated by the implementations of `scal` and `dot`:

```
/* scal: Each iteration can proceed independently of the others */
for (i = 0; i < SZ; i++)
  v[i] *= alpha;

/* dot: Each iteration depends on the previous one */
for (i = 0; i < SZ; i++)
  res += v[i] * w[i];
```

No use of this distinction is actually made in the system as it currently is, but it has been made because it might prove to be useful in future. When a sequential loop is fused with the parallel loop, the resulting loop is sequential. On a computer system where all types of loops are executed sequentially this is of no consequence, but for other systems it may be best to perform 'typed loop fusion' [8] where the degree of exploitable parallelism is never reduced by fusing loops of different types. A modern uniprocessor system, even though is lacks the ability to execute a large number of loop iterations simultaneously, may yet have a significant degree of instruction level parallelism; I can imagine that techniques to exploit this (such as software pipelining) may differ in the application to sequential and parallel loops, so it could be the case even here that typed loop fusion would be worth exploring.

4.3 Data flow analysis

Two of the four 'classic' data flow analyses are included in my system: reaching variable definitions, and live variable analysis. (The other two are analyses of available expressions and very busy expressions.) Reaching definitions are used to construct a list of dependencies between nodes, which is a necessary pre-cursor to the loop fusion stage. The liveness of variables is intended to be used in the code generation stage, in order to eliminate unnecessary writes to memory (see Section 4.3.2).

4.3.1 The monotone framework

Because two analyses were being used it made sense to me to to take advantage of the fact that are both instances of a 'monotone framework', and can be performed by the same algorithm.

In a monotone framework, performing an analysis consists of finding a canonical solution to a set of recursively defined equations.[11] The equations concern properties at the entry and exit of each node. Properties are taken from a finite complete lattice. The equations arise from monotone transfer functions that for each node relate its exit and entrance properties, and the joining of properties according to the flow relation between nodes.

I have used the worklist algorithm, taken from the presentation of it in [20]. The Scheme code for this is shown in Figure 5.

The algorithm is implemented in a generic fashion, taking parameters that define the instance of the monotone framework. The lattice in which the analysis is done is defined by its least element (`bottom`), an ordering relation (`lte?`), and a join operator (`join`). In both the analyses done here the lattice used is a powerset ordered by subset inclusion, so for both the least element is the empty set, and the join operator is set union. I used a list-based representation of sets, however there is nothing in the implementation of the algorithm given above that depends on this. A more efficient representation for sets would be as bit-vectors, but performance in this part of the system has never proved to be an issue, so I never took that step.

In a forwards analysis – such as reaching definitions – the parameter `flow` is the normal forwards flow relation: a list of all the edges in the flow chart. In a backwards analysis – such as live variables – it is the reverse flow relation that is given: all the flow chart edges with their directions reversed.[12]

The nodes themselves are referred to by numbers: $\{\mathtt{0} \ldots \mathtt{num} - \mathtt{nodes} - \mathtt{1}\}$. These numbers are the contained in the flow relation, are passed to functions `init?` and `tf`, and are the indices into the arrays `analysis` and `tf-analysis`. If it is the whole flow chart for a program that is being analyzed then these numbers will the positions of nodes in the flow chart; this is the same way nodes are referred to elsewhere in the system (such as in the textual representation of nodes used in this report). However, if it is only part of a program's flow chart that is being analyzed then a certain amount of translation has to be done.

Live variable analysis is performed for the flow chart of the whole program. If the idea is to remove writes to dead variables (*ie* removing the writing to memory of values that will never be read) then the analysis must be of the whole program in order for this optimization to be valid.

Reaching definitions analysis is only performed on the `rest-loop` block, and only with respect of the forward edges it contains (not the flow from its end back to its beginning). The reason for this is that the analysis is being done with a view to loop fusion. This optimization will only be done on the `rest-loop` block because that is where the program is likely to spend most of its execution time. The loops that may be fused are the one that arise from the implementation of certain instructions appearing in nodes. It does not make

[11]The canonical solution is the least fixed point for the equations.

[12]Another difference: in a forwards analysis, `analysis` holds properties for node entries and `tf-analysis` holds properties for node exits. With a backwards analysis it is the other way around.

```
(define mfp
  (lambda (bottom lte? join flow init? init-val tf num-nodes)
    (let ((w flow)
  (analysis (make-vector num-nodes))
  (tf-analysis (make-vector num-nodes)))
      (begin
;Step 1 - Initialisation
(do ((l 0 (1+ l)))
    ((= l num-nodes))
  (let* ((v (if (init? l) init-val bottom))
 (tf-v (tf l v)))
    (begin
      (vector-set! analysis l v)
      (vector-set! tf-analysis l tf-v))))
;Step 2 - Iteration
(do ()
    ((null? w))
  (let ((l (caar w))
(l- (cdar w)))
    (if (lte? (vector-ref tf-analysis l)
      (vector-ref analysis l-))
(set! w (cdr w))
(let*
    ((v (join (vector-ref analysis l-)
      (vector-ref tf-analysis l)))
     (tf-v (tf l- v)))
  (begin
    (vector-set! analysis l- v)
    (vector-set! tf-analysis l- tf-v)
     (lambda (p) (eq? l- (car p)))
     flow)
    (cdr w))))))))

;Step 3 - Presentation
; (could return tr-analysis as well, but there is no need)
analysis))))
```

Figure 5: Calculating canonical fixed-point

sense – at least in the system as it currently is – to fuse loops for nodes that are in different basic blocks, or to fuse across iterations of the `rest-loop` block.[13] So if a variable definition reaches a node in the `rest-loop` block from a node in an earlier block, or via the back edge of the `rest-loop`, then the only the only interesting information in this is that the definition comes from 'before' the beginning of the block.

The use of the work list algorithm is a bit excessive just to do reaching definitions analysis on a straight-line sequence of nodes. It is because the equations that the properties must satisfy are recursive that an iterative approach is used to find a fixed point for them. But in the absence of back edges the equations could be solved more directly. However, since the system contained code to handle the general case there was no compelling reason to add code to handle the more limited case.

4.3.2 Rationale for live variable analysis

A reaching definitions analysis is done because calculating dependencies is a necessary pre-cursor to loop fusion analysis. The requirement for a live variable analysis is not so strong, but I will explain my reason for including it in the system.

Suppose the source of a program contained the statements:

```
(dcopy x y)
(daxpy -1.0 y z)
```

(where `x`, `y` and `z` are all vectors). These might end up in the flow chart as the nodes:

```
(8 copy (x y))
(9 sub-assign (y z))
```

If these nodes were put in the same partition during the loop fusion stage, one might expect the generated C code to look something like this:

```
for (i = 0; i < SZ; i++)
{
  y[i] = x[i];
  z[i] -= y[i];
}
```

However, if it were known that the variable y was dead at the exit of the 'sub-assign' node, then the only place where the contents of y defined by the 'copy' node could be used would be the 'sub-assign' node. So in the fused loop the store to y could be eliminated:

[13]It could make sense under some circumstances. If it were given that the program's main loop would always be iterated at least once, then there could be fusion across the boundary between the `setup` and `first-loop` blocks. If there was sufficient information about the number of time the `rest-loop` block would be executed (or if the exact number was not important) the the loop could be unrolled, effectively allowing fusion across some iteration boundaries. But both of these cases in reality change what the basic blocks of the program are.

```
for (i = 0; i < SZ; i++)
{
  double y_i = x[i];
  z[i] -= y_i;
}
```

Furthermore, if it turned out that all writes to and reads from the contents of y were eliminated from the generated code, then there would be no need to allocate any storage space for y.

4.4 Loop fusion

In [14] the issue of how loops in a program should be fused is framed as a matter of partitioning a graph in which the nodes represent the loops being considered for fusion and the edges represent the dependence relations between loops. Loops will be fused if the nodes that represent them are put in the same partition. The problem is firstly to find a legal partition, and secondly to try and find an optimal partition (with respect to some criterion).

4.4.1 Safety

The dependencies between nodes create a partial order, and the partitioning of the graph must respect this. It must be possible to execute the partitions (or rather the fused loops arising from the partitions) in an order that accords with the ordering on the nodes; this would be violated if there were a pair of partitions that would have to be executed in one order to satisfy the one dependency and in the reverse order to satisfy another.

Also, for each partition it must be possible to order the statements in the fused body in such a way as to satisfy all the dependencies between the nodes in the partition. This can normally be achieved by ordering the statements according to the ordering of the nodes in the flowchart. But if the fused loop had a backwards-carried dependency then there would be no legal way to order the statements. So as to prevent this situation from arising, whenever two loops cannot be fused (because fusion would create an unsatisfiable dependency) a 'fusion-preventing edge' is put in the graph between the corresponding nodes. For a partitioning to be legal no partition should contain nodes connected by a fusion-preventing edge. As I said earlier, in my system it is easy to detect when fusion is impossible because it always involves a dependency via a scalar variable.

As the flowchart will contain nodes that are not fuseable – either because they represent operations that are not implemented as loops at all or because they are implemented as double loops which are currently beyond the scope of this system. The edges in the graph that involve such nodes should be marked as being fusion-preventing. That still leaves the possibility that an unfuseable node will end up in a partition with other nodes, as it is possible that a node can be put in a partition without have edges that connect to the other nodes in that partition. But this is not a serious problem. Such a node cannot be included in the fused loop produced by the partition, but can be out immediately before or after the fused loop; it does not matter which as it is self-evident that the node cannot have any dependence relation with any statement in the fused node that would prevent this.

4.4.2 Optimality

Judging how good a partition is depends on the purpose of fusion. Here fusion is to promote re-use, to bring together the execution of statements that use the same locations in memory. The approach of [14] is to weight the edges in the graph being partitioned according to the 'amount' of dependence their is between the different pairs of nodes (this will be discussed in more detail below). The problem is then to find a partition that maximizes the total weight on all the edges that do not cross partition boundaries. The paper shows that finding such an optimal solution is NP-hard, so the aim must be to find a 'good' partition, not necessarily the best.

4.4.3 Creating the graph

Four kinds of dependency need to be recognized: flow, output, anti and input. A flow dependency exists when one node writes to a variable and another node then value then reads the value written by the first. An output dependency is where two nodes write to the same variable; the last one will write the value seen by subsequent operations. An anti-dependence is where a node reads from a variable and another node then over-writes the value read. With an input dependency, two nodes read the same value from a variable. The first three of these are directed dependencies: the relative order of the two nodes in a dependence relation must be maintained in order to preserve the meaning of a program. The order in an input dependence does not need to be preserved, and the dependence simply represents a re-use of values.

Certain statements made above must be qualified to take account of input dependencies. For a partition to be valid it must respect the order constraints imposed by the dependencies; however input dependencies create no such constraints. Also, when I said that any edge representing a dependency involving a scalar variable must be a fusion-preventing edge, I should have added that this only applies for flow, anti, and output dependencies. Later on there will be talk of the 'successors' and 'predecessors' of a node; this will be with respect to the partial order on nodes created by dependencies of the three directed kinds.

In [14] flow and input dependencies are given a weight of 1, and output and anti dependencies are given a weight of 0. No account is made as to what type of variable the dependencies involve. However it seems clear that if two nodes represented operations that iterate over all the elements in some particular vector then the benefit of fusion would be much greater than if they just referred to the same scalar variable. Somewhat arbitrarily I decided to increase the weighting on dependencies involving vectors by a factor of 100.

There is no need to keep track of each dependencies kind (flow, anti, etc); all that matter from a practical point of view is what the weight on a dependency is, whether or not is prevents fusion, and whether or not it is directed (all but input dependencies are). Also, there is no need to keep track of multiple dependencies between a pair of nodes; they can be combined. Weights are summed, and if any of the individual dependencies being combined are fusion-preventing then the overall dependency is fusion-preventing (and likewise for directedness).

A dependency object has slots in it for the source and destination of the dependency edge in the graph, a weight, and flags indicating if the edge is weighted and/or fusion-preventing. Each such object corresponds to an edge in the graph, and may represent a combination of several dependencies in the program.

The dependency graph is built up using the reaching definitions analysis. A definition consists of the name of a variable and the number of a node where is is written to. The first step is to find for each definition the nodes that use it – *ie* the nodes that the definition reaches that read the defined variable - and the nodes that redefine it – *ie* the nodes that the definition reaches that write to the defined variable. From there it is a matter of saying that for a definition, all the nodes that use the definition have flow dependencies on the defining node and input dependencies on each other, and anti dependencies on redefining nodes; the redefining nodes have output dependencies on the defining node.

4.4.4 Partitioning the graph

As has been said, finding an optimal partitioning of a weighted dependency graph is NP-hard, so a sub-optimal solution will have to do if the analysis is to be tractable. In [14] two algorithms are presented. One of them is based on a reduction to another NP-hard graph problem for which there are heuristic algorithms that can produce solutions within a known bound of the optimal solution. However, this is a much more involved algorithm than the one I decided to implement, which was based on incrementally improving the solution produced by a 'greedy' algorithm.

The greedy algorithm is not presented explicitly in [14], so I have used a modified (and simplified) version of the algorithm presented in [1]. The idea is that a sequence of partitions is created, and each node is put in the earliest partition possible. To achieve this, when a partition is created all the nodes that have yet to be placed in partitions are considered in order; a node is added to the current partition if all of its immediate predecessors have been put in partitions, and the node has no fusion-preventing edges with any of the nodes already in the current partition.

The way this works, and its limitations, can be seen in this simple example:

```
(1 dot (res x y))
(2 neg (z))
(3 scal (res z))
```

The greedy algorithm will place the first two nodes in the first partition created, but has to create another partition for the third node as there is a fusion preventing dependency between the it and the first node. This is shown in Figure 6 (a). However, with a view to maximizing re-use it would be better if nodes 2 and 3 were put in a partition together (because they will both access all the elements in the vector `z`); the optimal partition is shown in Figure 6 (b).

An algorithm to improve the partitioning produced by the greedy algorithm is given in [14]. It works by moving nodes one-at-a-time between partitions in ways that increase the profitability of the partitioning. A node cannot be moved up into an earlier partition, because the greedy algorithm places each node in the earliest partition it can, but moving nodes down into later nodes may be possible. Here is the description from [14] of when it is safe and profitable to move a node n from a partition g_l to a later partition g_h:

Safety. A node $n \in g_l$ may move to g_h *iff* it has no successors in

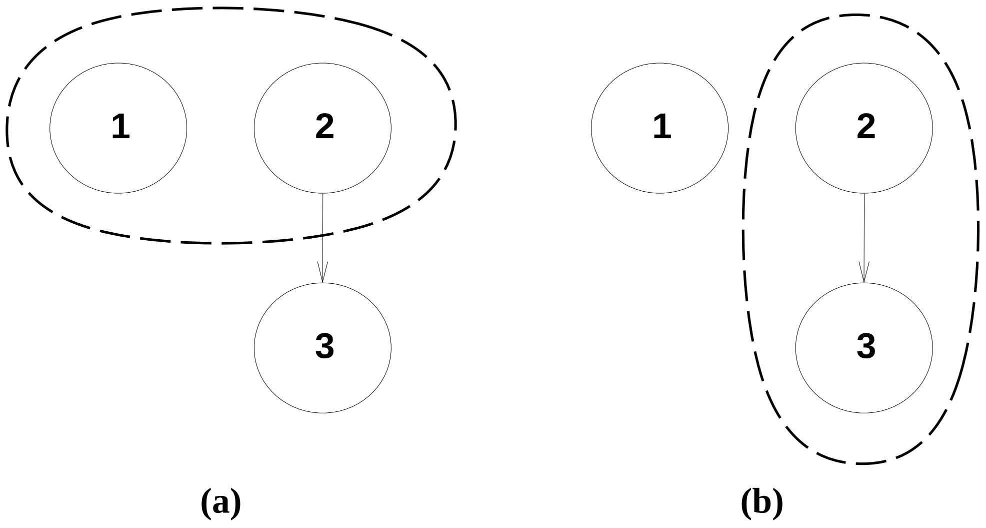

Figure 6: Greedy and optimal partitions

g_l and there is no fusion-preventing edge (n, r) such that $r \in g_h$ or $r \in g_k$ where g_k must precede g_h.

Profitability. Compute a sum of the edges (m, n), $m \in g_l$, and a sum for each g_h of edges (n, p), such that $p \in g_h$ and n may be safely moved into g_h. Pick the partition g with the biggest sum. If $g \neq g_l$ move n down into g.

(See [14] page 312.)

Their algorithm steps backwards through all the partition, and for each partition steps backwards through the nodes, and moves each node to the partition of greatest profit that it is safe to do. The reason for working backwards is that a node cannot be moved down to a later partition if any of its successors are in its current partition; it makes sense to see if these successors can be moved down first.

The description of the safety properties seems to me contain a small error. Suppose one were considering moving a node n from a partition g_l to a partition g_h, but r – a successor of n – was in a partition g_k that had to precede g_h. Moving n to g_h would produce an illegal partitioning (g_h and g_k would have to precede each other, which is not possible) irrespective of whether the edge (n, r) was fusion preventing or not.

There is also a claim that this algorithm is linear. It is not said what the algorithm is linear *in*, but I assume it is meant to be either the number of nodes or edges in the graph. I do not believe the algorithm is linear in either of these. When the safety of moving a node down to a different partition is being checked, part of this involves seeing if the node has any successors in partitions that must precede the target partition. A partition must precede another partition the first partition contains a node that must precede a node contained in the second; so potentially the relation between each node in the first partition and each node in the second must be checked. Assuming the 'must-precede' relation on a pair of nodes can be determined in constant time, this would mean the time complexity of determining the 'must-precede' relation on a pair of partitions would be $O(N^2)$, where N is the number of nodes. The relation between a pair of partitions depends on the relations between the nodes they contain. But during the course of the algorithm's run nodes will be moved from partition to partition. So the relation between a pair of partitions must be re-determined every time it is needed, which is to say every time the safety of a move is being checked. The number of moves that might be considered is bound by the number of directed edges in the graph (call this E), so the overall complexity resulting from just one of the safety properties is $O(EN^2)$.

This safety property also causes another complication. If it is safe to move a node n from from a partition g_l to a partition g_h then there must be no successors of n in partitions that must precede g_h. This is to say that there is a legal order in which to execute the partitions, and in this order none of the partitions containing successors of n precede g_h. However, such an order has to be calculated after the partitioning has been finalized. In the first implementation of the algorithm I missed the need for this step, and simply put the partitions in the same order they had been originally created. Subsequently I changed the safety check so that a node n cannot be moved to a partition g_h if it has a successor in a partition that actually does precede g_h (with the partitions being placed in the order they were created during the greedy algorithm).

Once the partitions are produced the nodes in the flow chart have to be reordered to reflect this, and the nodes that will be involved in loop fusion have to marked in recognition of this. The nodes affected by the partitioning (which is to say the `rest-loop` basic block) are ordered primarily by partition. Amongst the nodes in a partition, then ones that are not actually fuseable (for the reasons given in Section 4.4.1) are placed first, and the others come afterwards are are marked to indicate that they should be fused in the code generation stage.

4.5 Code Generation

I have not finished implementing this stage of the system, so in this section I will have to indicate how I envisage it working, and how I did performed this last stage in the process by hand for two benchmark programs: Conjugate Gradient Descent and Quasi-Minimal Residual.

This stage is a matter of taking the flowchart, and generating C code from it; the task of generating object code can be delegated to a standard C compiler.

As the overall organization of the of source programs (and their corresponding flowcharts) follow a rigid pattern in my system, so the overall structure of the generated code can be similarly stereotyped. The code has to be emitted in the form of a procedure, so that a larger program can use it as it will. This procedure could take problem size and the number of iterations to be performed as parameters. As well as code from the basic blocks of the flowchart, there would also have to be declarations for all variables used and allocation of memory used for vectors and matrices. An outline for a routine like this is shown in Figure 7. When I coded up the benchmark programs I found it just as easy to have the iterations controlled by a test for convergence (both programs featured a scalar measure of the error in their solutions) as having a fixed number.

When code for a block is to be generated then code for the nodes in that block must be generated in sequence. Nodes that are not marked for fusion can

```
void run(int SZ, int maxiter)
{
  int ct = 0;
  double* a_vector, /* etc */;
  double* a_matrix, /* etc */;
  double a_scalar, /* etc */;

  a_vector = malloc(SZ * sizeof(double));
  a_matrix = malloc(SZ * SZ * sizeof(double));
  /* etc */

  /* Code from "setup" block */

  if (ct++ < maxiter)
  {
    /* Code from "first-loop" block */
  }

  while (ct++ < maxiter)
  {
    /* Code from "rest-loop" block */
  }

  /* Code from "finish" block */
}
```

Figure 7: Ouline for the generated code

be processed individually, whereas those that are to be fused must be handled in their groups.

The easiest flowchart instructions to generate code for would be `initialise` and `results`, as these would just produce calls to similarly named C procedures, with the expectation that they would be defined elsewhere in a user's program.

Assignment instructions are also straightforward; these result in an assignment in the generated code. The assigned variable is given in in instruction's argument list, as is the right-hand side expression, although this must be 'unparsed' to create a valid C expression.

With a sub-BLAS instruction in node that is not to be fused the code to implement the operation is known independently of the particular program being processed – the only variation being provided by the arguments. So optimized code can be created up-front. This could be put in a library, so the code generated for the node would just be a call to a library routine. Or it could be inlined in the generated code, with the instruction arguments substituted in. When I created the code for the benchmark programs I put the code inline, but used naive implementations as I was interested in the effect that loop fusion would have rather than absolute performance.

In my system in its present form the only nodes that can be fused are ones that would be implemented in a single loop. Furthermore, each of the instructions in such nodes could be implemented with just a single line of code in the body of its loop. For instance, the node:

```
(33 dot (tmp13 r z))
```

could be implemented as

```
for (i = 0; i < SZ; i++)
  tmp13 += r[i] * z[i];
```

The format of the line in the body depends on the instruction and the variables (or scalar literals) used depend on the arguments. Knowing this is would be possible to create a fused loop where its body would simply be made up of a sequence of these lines corresponding to the nodes to be fused. Doing things this way would have two drawbacks. Firstly, it could result in unnecessary loads from memory: if a node reads a vector that is referenced in an earlier node in the sequence then it is possible that the value to be read in an iteration would already be in a register, but the C compiler may not know this and require the value to me loaded from main memory.[14] Secondly, it could result in unnecessary writes to memory: a vector written to by a node may turn out to be dead at the end of the sequence of nodes being fused.

The approach I used was to explicitly put values from the vectors involved in the fused loop into scalar variables local to the loop, and to divide the body of the loop into three parts: the first part would read values from the vectors into these local variables, the second part would consist of the lines essential to the implementation of each node's operation (but replacing references to vectors with the corresponding local variables), and the third to store values back into the vectors. To find the vectors that have to be read in the first part, it is necessary to go through each node in the sequence and find which vector

[14]The C compiler may be overly conservative when analyzing how memory references might alias each other.

variables it reads that had not be read or written by any nodes earlier in the sequence. To find the vectors that have to be written in the last part, it is necessary to go through all the nodes in the sequence and find the vectors that are written to, then compare this to the variables that are known to be live at the exit of the last node in the sequence. As an example of the kind of code created, consider this sequence of fuseable nodes:

```
(35 * copy (r z))
(36 | dot (tmp13 r z))
(37 | zero (q))
```

where z and q are both live at the exit of node 37. This produces the following code:

```
  for (i = 0; i < SZ; i++)
  {
    double r_i, z_i, q_i;

    /* First part */
    r_i = r[i];

    /* Second part */
    z_i = r_i;
    tmp13 += r_i * z_i;
    q_i = 0.0;

    /* Third part */
    z[i] = z_i; q[i] = q_i;
}
```

The information about which variables the nodes read and write would have been recorded when the flowchart was first created. The live variable analysis is performed after the loop fusion analysis (which may reorder the nodes in the flowchart). There is no need to analyze the fused loop itself whilst it is being created.

While there is not point having my compiler optimize the code generated of unfused nodes (as any such optimization could have been done in advance), it does make sense for it to apply further optimizations to the code generated for fused nodes; however I have not actually explored this practically. On machines where instruction-level parallelism can be exploited it may be advantageous to modify a fused loop so as to execute more that one iteration of the body simultaneously – through loop unrolling or software pipelining – which would be facilitated by having more that one set of local scalar variables for the different iterations proceeding simultaneously.

5 Results

As I have already said, I did not complete the compiler to the extent of having it produce runnable code. However it was not too much trouble to translate benchmark programs by hand from flowchart form into C code.[15]

The two programs I chose were Conjugate Gradient Descent (`CG`) and Quasi-Minimal Residual (`QMR`). As I said in Section 3.3.1, the language handled by my system lacks a feature that is used by the `QMR`: the ability to exit the program on certain error conditions (particular scalar values becoming too small). It was possible for me to ignore this issue until it came to generating C code. This was because any error-checking on a scalar variable would have to come between a definition and a use of that variable, and hence the nodes in question would have a dependency relation that would always prevent them being fused; all the analysis and optimization performed by my system could be done in ignorance of this error-checking, and it would always be possible to insert the checks in the C code where they were needed. This convinced me that adding error-checking features to the language itself would not affect the basic workings of my system. Also, I had the programs run until convergence (indicated by a suitably small value of calculated error) rather than running for a fixed number of iterations; as this was also easy to do I think it would be unproblematic for the system to be extended to accommodate this.

The system I tested on was a Linux-based PC with a *Get spec* processor, a ??? cache and ??? of physical memory. The C code was compiled with `gcc` with full optimization. As I primarily wanted to see the effects of loop fusion, I created three different C programs for each benchmark: one where the loop fusion stage had been omitted, one where it had been done but using the partitioning produced by the greedy algorithm, and one where the partitioning had been processed by the improving algorithm. For a benchmark, it would only be the code from the `rest-loop` block that would differ in the three cases, as this was the only block where loop fusion was performed. Execution time was measured by having code in the C programs to find the time immediately before and after the execution of the `rest-loop` block; by keeping count of the number of iterations required for a program to reach convergence on a particular problem it was possible to determine the average amount of time required for a single iteration of the main loop.

The initial results showed that loop fusion did decrease the execution time, but only very marginally. I suspected that this was due to the fact that the optimization only effected operations that were linear in the problem size, but that much of the execution time was being spent in matrix-vector multiplications that were quadratic in the problem size. I altered the C programs to include code immediately before and after each matrix-vector multiplications so as to exclude them from the measured time. The results are shown in Figures 8 and 9. When the programs with loop fusion performed using only the greedy algorithm were compared with those where no loop fusion was performed, the reduction in the execution time of the `rest-loop` block (excluding the matrix-vector multiplications) averaged 6% for `CG` and 17% for `QMR`. When programs with loop fusion performed using the improving algorithm were compared with programs with no loop fusion, then the reduction averages 10% for `CG` and 33% for `QMR`.

The fact that the performance improvements were so much greater in `QMR` is probably due to the fact that nodes tended to be fused in larger groups in `QMR` and `CG`: in the fastest version of `CG` the largest fusion partition contained

[15]Doing this was very tedious, and hence automateable. The re-writing of flowchart nodes as C code was made much easier by the heavy use of the regular expression replacement facility in the `emacs` editor.

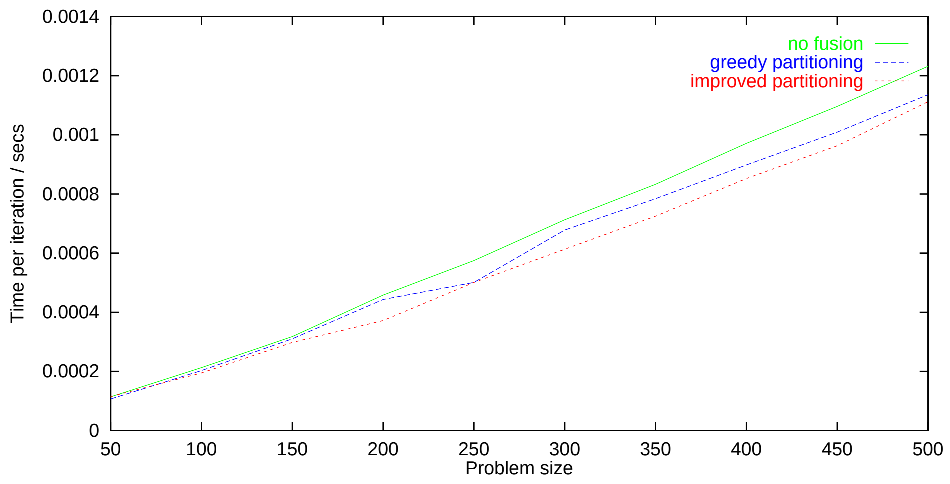

Figure 8: Three version of Conjugate Gradient Descent

3 nodes, whereas with `QMR` it the largest partition contained 10 nodes. I had thought that larger partitions might have an adverse effect on performance, as it could result in loops that accessed a large number of different vectors, and hence cause cache contention during execution. However, this does not seem to be the case; in these tests, at least.

6 Discussion

In broad terms, what have I achieved by writing my system? I have created a compiler for a very specific class of program, and in it I have focused on one sort of optimisation. As is always the case, the extent of the improvement in a program's peformance due to a particular optimisation is dependent on the proportion of the program to which the optimtimisation can be applied. My system performs fusion amongst single loops, and in regions of a program where execution time is dominated by such loops then the optimisation performed by my system produces a significant effect.

6.1 Optimisation

The question arises: how does this differ from what could be achieved with an optimising compiler for a general-purpose language? An answer comes from the fact that in my system loops are fused that relate to separate BLAS operations; optimisation takes place across operation boundaries. In a sense, there is nothing going on that could not happen in a normal compiler: if a program was written without the use a library such as BLAS, or if the compiler could inline all subroutine calls, then it would be perfectly possible for a compiler to perform the same sort of cross-operation optimisation. And, in a sense, the way I have envisaged my system's code generation phase working is not very different from inlining subroutine calls.

What is different in a regular compiler and in my system is how the analysis necessary to optimisation is performed. If a program makes use of BLAS

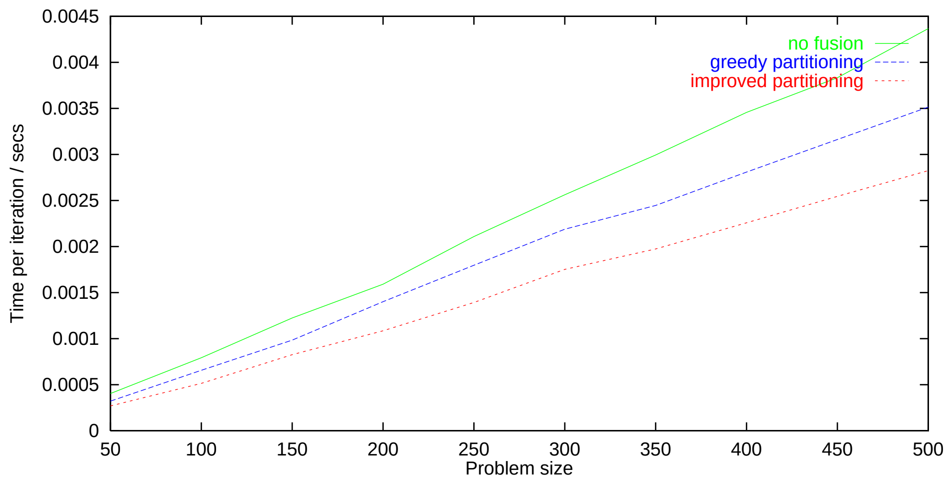

Figure 9: Three version of Quasi-Minimal Residual

routines, and a regular compiler was to do any cross-operation optimisation, then it will have to inline the subroutine calls, then analyse the resulting code to see what oportunities it contains. In my system – after BLAS operations have been decomposed in sub-BLAS operations – the analysis for loop fusion can be performed without having to look 'inside' the code that implements the computations; such information as is required (such as where different variables are written to and read from) is placed in node objects that represent a program at a higher level of abstraction[16] than would be present in a compiler for a normal programming language. Because the language processed by my system is very limited in comparision to a general-purpose langauge it can use a representation that is tailored to its particular needs, such as making loop fusion analysis as easy as possible. I might also say that highly optimising compilers for general-purpose programming languages tend to be written for specific – high performance – hardware platforms (often by the vendors themselves) and so may not be available for many systems.

6.2 Domain specific languages

The sort of mini-language that is handled by my system is commonly refered to as a *domain specific language* (DSL) – in the case of my system the 'domain' covered is that of iterative system solvers. In [6] such languages are divided into three categories: fixed, separate DSLs, such as TeX or Awk; embedded DSLs, often in the form of prodecure or class libraries, that can be used from general purpose programing languages; modularly composable DSLs, which require the use of a specially designed development system (such as Microsoft's Intentional Programming system [18] where programs can be constructed from parts written using a variety little languages.

My system falls into the first of these categories. The way it avoids being an isolated "technology island" is by its code generation strategy being to produce C that can be compiled and linked into a larger program. This is not unusual,

[16]Or – equivalently – a lower level of detail.

the UNIX utilities `lex` and `yacc` also define little languages and use this method of integration. As my system was not finished as a whole – and in particular lacked the code generation phase – I cannot really judge the success of this choice with regard to the pragmatics of using the system. However it is not difficult to imagine the inconvenience of separating parts of a program that can be accomodated within the language of my system from parts that can (even though they might logically belong together).

Working within the third of the categories is not a realistic possibility at the moment for all but a few people, since it depends on the use of advanced program development environments that are not generally available outside the research labs of a few companies. Whether this remains to be the case is not a question that can yet be answered.

To follow on from my argument of Section 1.2, I would say that most of the work of linguistic creativity that goes on in software engineering is channeled into the second of the three categories. I suspect that the authors of [6] might not go as far as me in saying that any library interface constitutes a minor programming language. In any case, there is a difference between saying this and making use of it in a self-conscious manner: if a library interface can be thought of as at least being language-like, then it is possible its implementation could be compiler-like. This is the step taken with `active libraries` [7], which are ones where the library takes on an active role in the code generation process rather than just being a repository of compiled code pre-existing the program that uses it. Research on active libraries is on-going, but I will compare it to my work in two areas: the relation between implementation strategy and effectiveness, and the intellectual analysis required prior to design.

6.2.1 Active library *vs* little compiler

There are currently active libraries whose problem domain is in the same general region as that of my system: numerical computations involving matrices and vectors. These systems (Blitz++, the Generative Matrix Computation Library, see [7] for details) use C++. C++ is a good language to use for this sort of thing in that its operator overloading facility makes it easy to construct readable expressions. For instance, one might be able to add three vectors together with a statement like this:

```
a = b + c + d;
```

whereas in my system this would be done like this:

```
(dcopy b a)
(daxpy 1.0 c a)
(daxpy 1.0 d a)
```

Older linear algebra class libraries would suffer in their handling of the example given above bacause they would too often use temporary objects to hold intermediate values. In the example above the addition of `b` and `c` would be executed first with its result being put in a temporary vector object, which would then be added with `d`. In a library like Blitz++, some rather peculiar features of the C++ template system are used (misued, some would say) to prevent this happening. The overloaded addition operators do not yield the results of vector additions, but objects that represent the expression to be evaluated.

Expression objects can themselves be part of larger expressions, and it is not until an expression is the right-hand side of an assignment that the 'active' part of the library will generate code (at compile time) to evaluate the expression. For the above example this might produce a result equivalent to the following hand-written C code:

```
for (i = 0; i < SZ; i++)
  a[i] =  b[i] + c[i] + d[i];
```

In my system the same basic outcome could be produced. I have shown the three statements needed to perform this summation. These would result in three flowchart nodes. During loop fusion analysis it would be likely that these nodes would end up in the same partition, and so the loop finaly generated to do the summation would be much the same as the loop given above.

What is different between the two approaches is that a C++ active library is constrained to work at the level of statements. Continuing with the example, what if in the context of a larger program it turned out that the vector a was itself just a temporary holding an intermediate result – one created by the programmer to make the program clearer – and that it would be possible to fuse the summation with the operation where its result was used and never store anything to a at all? My system could potentially detect this situation: the form of the language does not restrict which loops can be fused, and if a vector variable turns out to be dead on the exit of a fused loop then it need not be written to. An active library could not do this, because its actions are essentially guided by syntactic form of a program.

It could be argued that the problem of program source code containing named variables to contain intermediate results is one that is exagerated by the lack of expressiveness in a BLAS-like interface, and the ability of my system to look at a program in terms of data flow simply compensates for this. However, I think the matter goes deeper than this. It is true that I followed the BLAS interface when defining the statements in my language, but I need not have done. But I see no way for an active library to get past its limitations. An active library is really not that different from a system of macros, but whereas macros manipulate a program its concrete form, an active library works at the level of a program's 'metaobjects': classes, type definitions, methods, *etc.*[16] This meta-level for handling programs still seems to me to be working near the 'skin' of a program; to delve deeper, to make optimisation decisions based on how data flows through a program rather than the form the program takes, would require the sort of analysis only a real compiler can provide. This is not to say that there is anything wrong with being a smart macro system; it remains to be seen whether the limitations of active libraries constitute flaws or not.

6.2.2 Domain Analysis

When my system is compared to the Generative Matrix Computation Library (GMCL, see [6] Chapter 14), one area where the limitations of my approach is apparent is in the prior analysis of the problem domain. I cannot really say that I performed an analysis as such at all; I took BLAS to be a good model for an interface on account of its widespread use, I looked at some benchmark programs, noticed some striking regularities of form, and took it from there. Having no great experience of numerical computation I found it difficult to evaluate the merits of particular focus I chose. With GMCL it is the fact that there is so much to know about linear algebra computations that makes it interesting: “Matrix computations is a domain with plenty of documented domain knowledge and thus a perfect candidate for Domain Engineering. Given some 30 years of development, it is a complex domain with a lot of variablity.”[6]

It is not simply a question of having a wider or narrower scope, although that comes into it. When I wanted a focus for my project I looked at the common properties of a number of benchmark programs. I did not consider *variability* as a feature of the problem space in itself; variation was simply everything that remained after the limits of the system had been set. In reality this meant that when operations could be put in sequences, there were no restrictions on how *that* could be done.

The analysis done for GMCL – on the other hand – seems to be a way of looking at the problem space in terms of a system of variations: real or complex numbers, single or double precision, sparse or dense matrix representation, different types of subscripting, *etc.* The system is generative in the sense that – were the system to reach its ideal point – it would be to generate from within itself any approach to matrix computation anyone might have used over the last 30 years, just by varying its mix of features. Such an approach must enevitably be encyclodepic both in its scope and in the particular way it handles accumulated knowledge.

References

[1] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*. ACM Press, 1987.

[2] J. L. Autin. *How to do things with words*. Clarendon Press, 1962.

[3] Olav Beckmann. A lazy, self-optimising parallel matrix library. Master's thesis, Department of Computing, Imperial College, London SW7 2AZ, UK, 1996.

[4] Andrew Berlin. Partial evaluation applied to numerical computation. In *1990 ACM Conference on Lisp and Functional Programming*. ACM Press, 1990.

[5] Frederick P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, anniversary edition, 1995.

[6] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools and applications*. Addison-Wesley, 2000.

[7] K. Czarnecki, U. Eisenecker, R. Gluck, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries, 1998.

[8] Alain Darte. On the complexity of loop fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*. IEEE Computer Society Press, 1999.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 1988.

[10] Umberto Eco. *The Aesthetics of Thomas Aquinas*. Radius, 1988. Translated by Hugh Bredin.

[11] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Langauges*. MIT Press, 1992. Forward by Harold Abelson.

[12] David Gelernter. *The Aesthetics of Computing*. Phoenix, paperback edition, 1999.

[13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[14] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving dara locality via loop fusion and distribution. In *The Sixth Annual Languages and Compilers for Parallelism Workshop*. Springer-Verlag, 1993.

[15] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[16] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[17] Donald E. Knuth. Computer programming as an art. In *ACM Turing Award Lectures*, pages 33–46. ACM Press, 1987.

[18] The intentional programming project. Homepage at http://www.research.miscrosoft.com/ip.

[19] Reference blas implementation. Available at http://www.netlib.org/blas/.

[20] Felmming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[21] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

A Scheme source code

A.1 parsing.scm

```
(require 'struct)

(define-record prgm (decls setup loop finish))
(define-record decls (matrices vectors scalars))

(define-record assign (var rhs))
(define-record loop-cond (first-body rest-body))
(define-record sub-call (name args))

(define-record fun-call (name args))
(define-record var-ref (name))
(define-record lit (value))
(define-record aexpr (rator rands))

(define parse-prgm
  (letrec
      ((parse-decls
(lambda (text)
  (begin
    (assert (eq? (car text) 'declarations) "Missing declarations")
    (make-decls (cadr text) (caddr text) (cadddr text)))))
       (parse-setup
(lambda (text)
  (begin
    (assert (eq? (car text) 'setup) "Missing setup")
    (map (lambda (x) (parse-stmt x #f)) (cdr text)))))
       (parse-loop
(lambda (text)
  (begin
    (assert (eq? (car text) 'loop) "Missing loop")
    (map (lambda (x) (parse-stmt x #t)) (cdr text)))))
       (parse-finish
(lambda (text)
  (begin
    (assert (eq? (car text) 'finish) "Missing finish")
    (map (lambda (x) (parse-stmt x #f)) (cdr text)))))
       (parse-stmt
(lambda (s in-loop)
  (let ((command (car s)))
    (cond
     ((eq? command ':=)
      (make-assign (cadr s) (parse-expr (caddr s))))
     ((eq? command 'first-time)
      (begin
(assert in-loop "firsttime only allowed in loop")
(make-loop-cond
 (map (lambda (x) (parse-stmt x #f)) (cadr s))
 (map (lambda (x) (parse-stmt x #f)) (caddr s)))))
     (else
      (make-sub-call command (map parse-expr (cdr s)))))))
       (parse-expr
(lambda (s)
  (cond
   ((string? s) s) ; ie a "T" or "N" from a dgemv call
   ((number? s) (make-lit s))
   ((symbol? s) (make-var-ref s))
   ((pair? s)
    (let ((op (car s))
  (l (length (cdr s))))
      (cond
       ((and (= l 2) (memv op '(+ - * /)))
(make-aexpr op (map parse-expr (cdr s))))
       ((and (= l 1) (eq? op '-))
(make-aexpr 'neg (map parse-expr (cdr s))))
       (else
(make-fun-call op (map parse-expr (cdr s)))))))
   (else
    (error "Bad expression" s))))))
    (lambda (text)
      (make-prgm
       (parse-decls (car text))
       (parse-setup (cadr text))
       (parse-loop (caddr text))
       (parse-finish (cadddr text))))))
```

A.2 analysing.scm

```
(require 'struct)
(require 'format)

(define-record node (instr args fusion reads writes))
(define-record flowchart (decls nodes num-nodes rel labels))

; An instruction has a number of arguments. An argument could be the name of a
; variable that is read, written or both, or an expression (in which case all its
; free variables are read).
; The instructions "initialise" and "results" are not dealt with here, as they
; are special (they don't have a fixed number of arguments for a start).

(define instr-arg-pattern
  (lambda (instr)
    (case instr
      ((dummy) '())
      ((assign) '(w e))
      ((mvmul-n mvmul-t) '(e r r rw))
      ((copy) '(r w))
      ((dot) '(rw r r))
      ((axpy) '(e r rw))
      ((scal) '(e rw))
      ((add-assign sub-assign) '(r rw))
      ((neg) '(rw))
      ((zero) '(w))
      (else (error "Unknown instruction" instr)))))

(define fusion-info
  (lambda (instr)
    (case instr
      ((dummy initialise results assign mvmul-t mvmul-n)
       'nonfuseable)
      ((copy axpy add-assign sub-assign scal neg zero)
       'parloop)
      ((dot)
       'seqloop)
      (else (error "unknown instruction" instr)))))

(define new-flowchart
  (lambda (decls)
    (make-flowchart decls (make-vector 1) 0 '() '())))

(define expand-flowchart
  (lambda (fl)
    (let* ((vec (flowchart->nodes fl))
   (vec-len (vector-length vec))
   (new-vec (make-vector (* 2 vec-len))))
      (begin
(do ((n 0 (1+ n)))
    ((= n vec-len))
  (vector-set! new-vec n (vector-ref vec n)))
(set-flowchart-nodes! fl new-vec)))))

(define add-flowchart-label
  (lambda (fl label-name)
    (begin
      (add-node+ fl 'dummy '())
      (let* ((posn (1- (flowchart->num-nodes fl)))
     (new-label (cons label-name posn))
     (old-label-list (flowchart->labels fl)))
(set-flowchart-labels! fl (cons new-label old-label-list))))))

(define flowchart-label->posn
  (lambda (fl label-name)
    (letrec
((find-posn (lambda (alist)
      (cond
       ((null? alist)
(error "Unknown label" label-name))
       ((eq? label-name (caar alist))
(cdar alist))
       (else
(find-posn (cdr alist)))))))
      (find-posn (flowchart->labels fl)))))

(define flowchart-posn->label
  (lambda (fl posn)
    (letrec
((find-label (lambda (alist)
       (cond
((null? alist)
 '<unlabelled>)
((eq? posn (cdar alist))
 (caar alist))
(else
 (find-label (cdr alist)))))))
      (find-label (flowchart->labels fl)))))

(define flowchart->node
  (lambda (fl posn)
    (vector-ref (flowchart->nodes fl) posn)))

(define print-flowchart
  (lambda (fl)
    (let* ((vec (flowchart->nodes fl))
   (num-nodes (flowchart->num-nodes fl)))
      (do ((n 0  (1+ n)))
  ((= n num-nodes))
(let ((node (vector-ref vec n)))
  (cond
   ((eq? 'dummy (node->instr node))
    (print (list n 'LABEL (flowchart-posn->label fl n))))
   ((eq? 'fusion-begin (node->fusion node))
    (print (list n '* (node->instr node) (node->args node))))
   ((eq? 'fusion-continue (node->fusion node))
    (print (list n '| (node->instr node) (node->args node))))
   (else
    (print (list n (node->instr node) (node->args node))))))))))

(define set-reads-writes!
  (lambda (node)
    (let ((instr (node->instr node))
  (args (node->args node)))
      (case instr
((initialise) (set-node-writes! node args))
((results) (set-node-reads! node args))
(else
 (let* ((reads '())
(writes '())
(add-read  (lambda (x) (set! reads  (adjoin x reads))))
(add-write (lambda (x) (set! writes (adjoin x writes))))
(examine-args
 (lambda (arg pattern)
   (case pattern
     ((e) (set! reads (free-vars arg)))
     ((r) (begin
    (assert (symbol? arg) "should be var")
    (add-read arg)))
     ((w) (begin
    (assert (symbol? arg) "should be var")
    (add-write arg)))
     ((rw wr) (begin
(assert (symbol? arg) "should be var")
(add-read arg)
(add-write arg)))
     (else (error "unrecognised pattern"))))))
   (begin
     (map2 examine-args args (instr-arg-pattern instr))
     (set-node-reads! node reads)
     (set-node-writes! node writes))))))))

(define free-vars
  (lambda (expr)
    (letrec
((vars '())
 (fv (lambda (expr)
      (variant-case expr
 (var-ref (name)
   (if (not (memv name vars))
      (set! vars (cons name vars))))
 (lit (value) nothing)
 (fun-call (name args)
   (map fv args))
 (aexpr (rator rands)
   (map fv rands))
 (else (error "Illegal expression" expr))))))
      (begin
(fv expr)
vars))))

(define add-node
  (lambda (fl instr args)
    (begin
      (if (= (vector-length (flowchart->nodes fl))
     (flowchart->num-nodes fl))
  ; node vector full
  (expand-flowchart fl))
```

```
      (let ((vec (flowchart->nodes fl))
    (posn (flowchart->num-nodes fl))
    (node (make-node instr args
     (fusion-info instr)
     '() '())))
(begin
  (set-reads-writes! node)
  (vector-set! vec posn node)
  (set-flowchart-num-nodes! fl (+ posn 1)))))))

(define add-node+
  (lambda (fl instr args)
    (let ((posn (flowchart->num-nodes fl)))
      (begin
(add-node fl instr args)
(if (> posn 0)
    (add-flow fl (1- posn) posn))))))

(define add-flow
  (lambda (fl src dst)
    (let* ((posn (lambda (x) (if (symbol? x) (flowchart-label->posn fl x) x)))
   (f (cons (posn src) (posn dst)))
   (rel (flowchart->rel fl)))
      (set-flowchart-rel! fl (adjoin f rel)))))

(define new-scalar
  (lambda (fl)
    (let* ((decls (flowchart->decls fl))
   (num-scalars (length (decls->scalars decls)))
   (name (string->symbol (format "tmp~a" num-scalars))))
      (begin
(set-decls-scalars! decls (cons name (decls->scalars decls)))
name))))

(define unfuseable
  (lambda (fl n)
    (let* ((nodes (flowchart->nodes fl))
   (node (vector-ref nodes n)))
      (equal? (node->fusion node) 'nonfuseable))))

;-----------------------------------------------

(define analyse-prgm
  (lambda (p)
    (let* ((decls (variant-case (prgm->decls p)
    (decls (matrices vectors scalars)
      (make-decls matrices vectors scalars))
    (else (error "Case match failed"))))
   (setup (prgm->setup p))
   (loop (prgm->loop p))
   (finish (prgm->finish p))
   (fl (new-flowchart decls)))
      (begin
(add-flowchart-label fl 'setup-begin)
(analyse-statements  fl 'setup setup)
(add-flowchart-label fl 'setup-end)

(add-flowchart-label fl 'first-loop-begin)
(analyse-statements  fl 'first-loop loop)
(add-flowchart-label fl 'first-loop-end)

(add-flowchart-label fl 'rest-loop-begin )
(analyse-statements  fl 'rest-loop loop)
(add-flowchart-label fl 'rest-loop-end )

(add-flowchart-label fl 'finish-begin )
(analyse-statements  fl 'finish finish)
(add-flowchart-label fl 'finish-end )

(add-flow fl 'setup-end 'finish-begin) ; loop body might never execute
(add-flow fl 'first-loop-end 'finish-begin) ; or execute just once
(add-flow fl 'rest-loop-end 'rest-loop-begin)

fl))))

(define analyse-statements
  (lambda (fl section statements)
    (map (lambda (stmnt)
   (begin
     (analyse-statement fl section stmnt)))
 statements)))

(define analyse-statement
  (lambda (fl section stmnt)
    (variant-case stmnt
      (assign (var rhs)
        (analyse-assign fl var rhs))
      (sub-call (name args)
(case name
  ((initialise results)
   (let ((vars (map var-ref->name args)))
     (add-node+ fl name vars)))
  (else
   (analyse-sub-call fl name args))))
      (loop-cond (first-body rest-body)
        (cond
 ((eq? section 'first-loop)
  (analyse-statements fl section first-body))
 ((eq? section 'rest-loop)
  (analyse-statements fl section rest-body))
 (else
  (error "Illegal loop-cond in section" section))))
      (else (error "Unknown statement type" stmnt)))))

(define analyse-assign
  (lambda (fl var rhs)
    (letrec
((remove-blas-fun-calls
  (lambda (expr)
    (variant-case expr
              (var-ref (name) expr)
      (lit (value) expr)
      (fun-call (name args)
(if (blas-fun? name)
    (let ((ref (make-var-ref (new-scalar fl)))
  (sub (sub-of-fun name)))
      (begin
(analyse-sub-call fl sub (cons ref args))
ref))
    (make-fun-call name (map remove-blas-fun-calls args))))
      (aexpr (rator rands)
(make-aexpr rator (map remove-blas-fun-calls rands)))
      (else expr)))))
      (add-node+ fl 'assign (list var (remove-blas-fun-calls rhs))))))
```

```
(define analyse-sub-call
  (lambda (fl name args)
    (let*
((remove-complex-exprs
  (lambda (expr)
    (if (not (or (fun-call? expr) (aexpr? expr)))
expr
(let ((tmp (new-scalar fl)))
  (begin
    (analyse-assign fl tmp expr)
    (make-var-ref tmp))))))
 (coerce
  (lambda (type arg)
    (if (eq? type 'var)
(begin
  (assert (var-ref? arg) "Incorrect argument type")
  (var-ref->name arg))
arg)))
 (new-args (map remove-complex-exprs args))
 (analysis-func (analysis-of-sub name))
 (arg-types (arg-types-of-sub name)))
      (apply analysis-func (cons fl (map2 coerce arg-types new-args))))))

;---------------------------------------------------------------------------

(define analyse-dnrm2-s
  (lambda (fl res v)
    (let* ((tmp (new-scalar fl))
   (sqrt-call (make-fun-call 'sqrt (list (make-var-ref tmp)))))
      (begin
(add-node+ fl 'assign (list tmp (make-lit 0.0)))
(add-node+ fl 'dot (list tmp v v))
(add-node+ fl 'assign (list res sqrt-call))))))

(define analyse-ddot-s
  (lambda (fl res v1 v2)
    (begin
      (add-node+ fl 'assign (list res (make-lit 0.0)))
      (add-node+ fl 'dot (list res v1 v2)))))

(define analyse-dcopy
  (lambda (fl v1 v2)
    (add-node+ fl 'copy (list v1 v2))))

(define analyse-daxpy
  (lambda (fl alpha x y)
    (variant-case alpha
      (var-ref (name)
(add-node+ fl 'axpy (list alpha x y)))
      (lit (value)
(cond
 ((= value 0.0) (nothing))
 ((= value 1.0)
  (add-node+ fl 'add-assign (list x y)))
 ((= value -1.0)
  (add-node+ fl 'sub-assign (list x y)))
 (else
  (add-node+ fl 'axpy (list alpha x y)))))
      (else
       (error "daxpy: alpha expression of wrong sort" alpha)))))
```

```
(define analyse-dgemv
  (lambda (fl trans alpha a x beta y)
    (let ((tr (or (equal? trans "T") (equal? trans "t"))))
      (begin
(variant-case beta
  (var-ref (name)
    (add-node+ fl 'scal (list beta y)))
  (lit (value)
    (cond
     ((= value 0.0)
      (add-node+ fl 'zero (list y)))
     ((= value 1.0) (nothing))
     ((= value -1.0)
      (add-node+ fl 'neg (list y)))
     (else
      (add-node+ fl 'scal (list alpha y)))))
    (else
     (error "dgemv: beta expression of wrong sort" beta)))

(add-node+ fl (if tr 'mvmul-t 'mvmul-n) (list alpha a x y))))))

(define analyse-dscal
  (lambda (fl alpha x)
    (add-node+ fl 'scal (list alpha x))))

;-----------------------------------------------------------------

(define flowchart-var->type
  (lambda (fl var)
    (variant-case (flowchart->decls fl)
      (decls (matrices vectors scalars)
        (cond
 ((member? var matrices) 'matrix)
 ((member? var vectors) 'vector)
 ((member? var scalars) 'scalar)
 (else (error "Unknown variable" var))))
      (else (error "Case match failed")))))

;-----------------------------------------------------------------

(define blas-fun?
  (lambda (name)
    (memv name '(dnrm2 ddot))))

(define sub-of-fun
  (lambda (name)
    (case name
     ((dnrm2) 'dnrm2-s)
     ((ddot) 'ddot-s)
     (else (error "Unknown blas function" name)))))

(define analysis-of-sub
  (lambda (name)
    (case name
     ((daxpy) analyse-daxpy)
     ((dcopy) analyse-dcopy)
     ((dnrm2-s) analyse-dnrm2-s)
     ((ddot-s) analyse-ddot-s)
     ((dgemv) analyse-dgemv)
     ((dscal) analyse-dscal)
     (else (error "Unknown blas subroutine" name)))))
```

```
(define arg-types-of-sub
  (lambda (name)
    (case name
     ((daxpy) '(* var var))
     ((dcopy) '(var var))
     ((dnrm2-s) '(var var))
     ((ddot-s) '(var var var))
     ((dgemv) '(* * var var * var))
     ((dscal) '(* var))
     (else (error "Unknown blas subroutine" name)))))
```

A.3 data-flow.scm

```
(define mfp
  (lambda (bottom lte? join flow init? init-val tf num-nodes)
    (let ((w flow)
  (analysis (make-vector num-nodes))
  (tf-analysis (make-vector num-nodes)))
      (begin
;Step 1 - Initialisation
(do ((l 0 (1+ l)))
    ((= l num-nodes))
  (let* ((v (if (init? l) init-val bottom))
 (tf-v (tf l v)))
    (begin
      (vector-set! analysis l v)
      (vector-set! tf-analysis l tf-v))))
;Step 2 - Iteration
(do ()
    ((null? w))
  (let ((l (caar w))
(l- (cdar w)))
    (if (lte? (vector-ref tf-analysis l)
      (vector-ref analysis l-))
(set! w (cdr w))
(let*
    ((v (join (vector-ref analysis l-)
      (vector-ref tf-analysis l)))
     (tf-v (tf l- v)))
  (begin
    (vector-set! analysis l- v)
    (vector-set! tf-analysis l- tf-v)
    (set! w (append (filter (lambda (p) (eq? l- (car p))) flow)
    (cdr w))))))))

;Step 3 - Presentation
analysis))))

(define rd-entry
  (lambda (fl)
    (let*
((vars (variant-case (flowchart->decls fl)
 (decls (matrices vectors scalars)
   (append matrices (append vectors scalars)))
 (else (error "Case match failed"))))
 (nodes (flowchart->nodes fl))
 (num-nodes (flowchart->num-nodes fl))
 (bottom '())
 (lte? subset?)
 (join union)
 (flow (reverse (flowchart->rel fl)))
 (init? (lambda (l) (eq? l 0)))
 (init-val (map (lambda (var) (cons var '?)) vars))
 (tf (lambda (l v)
       (let* ((node (vector-ref nodes l))
      (writes (node->writes node)))
 (union (filter (lambda (x) (not (member? (car x) writes))) v)
(map (lambda (var) (cons var l)) writes))))))
      (mfp bottom lte? join flow init? init-val tf num-nodes))))

(define lv-exit
  (lambda (fl)
    (let*
((vars (variant-case (flowchart->decls fl)
 (decls (matrices vectors scalars)
   (append matrices (append vectors scalars)))
 (else (error "Case match failed"))))
 (nodes (flowchart->nodes fl))
 (num-nodes (flowchart->num-nodes fl))
 (bottom '())
 (lte? subset?)
 (join union)
 (flow (map (lambda (x) (cons (cdr x) (car x))) (flowchart->rel fl)))
 (init? (lambda (l) (eq? l (1- num-nodes))))
 (init-val '())
 (tf (lambda (l v)
       (let* ((node (vector-ref nodes l))
      (reads (node->reads node))
      (writes (node->writes node)))
 (union (filter (lambda (x) (not (member? x writes))) v)
reads)))))
      (mfp bottom lte? join flow init? init-val tf num-nodes))))

; This routine prints for each node the definitions that reach the node entry
; of variables that the node reads
(define print-rd-e
  (lambda (fl)
    (let ((rd-e (rd-entry fl))
  (nodes (flowchart->nodes fl)))
      (do ((i 0 (1+ i)))
  ((= i (vector-length rd-e)))
(let* ((node (vector-ref nodes i))
       (reads (node->reads node))
       (relevant (lambda (v) (member? (car v) reads))))
  (print (list i (filter relevant (vector-ref rd-e i)))))))))

; This routine prints for each node the variables that are written to
; by the node but that are dead on exit
(define print-lv-e
  (lambda (fl)
    (let ((lv-e (lv-exit fl))
  (nodes (flowchart->nodes fl)))
      (do ((i 0 (1+ i)))
  ((= i (vector-length lv-e)))
(let* ((node (vector-ref nodes i))
       (live (vector-ref lv-e i))
       (relevant (lambda (v) (not (member? v live)))))
  (print (list i (filter relevant (node->writes node)))))))))

(define loop-rd-entry
  (lambda (fl)
    (let*
((vars (variant-case (flowchart->decls fl)
 (decls (matrices vectors scalars)
   (append matrices (append vectors scalars)))
 (else (error "Case match failed"))))
 (nodes (flowchart->nodes fl))
 (num-nodes (flowchart->num-nodes fl))
 (bottom '())
 (lte? subset?)
 (join union)
 (loop-begin (flowchart-label->posn fl 'rest-loop-begin))
 (loop-end (flowchart-label->posn fl 'rest-loop-end))
 (in-loop? (lambda (x) (and (>= x loop-begin) (<= x loop-end))))
 (flow (reverse (filter (lambda (r)
  (let ((src (car r))
(dst (cdr r)))
    (and (in-loop? src) (in-loop? dst)
 (> dst src))))
(flowchart->rel fl))))
 (init? (lambda (l) (eq? l loop-begin)))
 (init-val (map (lambda (var) (cons var '?)) vars))
 (tf (lambda (l v)
       (let* ((node (vector-ref nodes l))
      (writes (node->writes node)))
 (union (filter (lambda (x) (not (member? (car x) writes))) v)
(map (lambda (var) (cons var l)) writes))))))
      (mfp bottom lte? join flow init? init-val tf num-nodes))))

(define all-defs
  (lambda (rd-e) (fold union '() (vector->list rd-e))))

(define chains
  (lambda (fl rd-e defs)
    (let* ((num-defs (length defs))
   (def-use (make-vector num-defs '()))
   (def-def (make-vector num-defs '()))
   (add-entry (lambda (v i el)
(vector-set! v i (adjoin el (vector-ref v i))))))
      (do ((n 0 (1+ n)))
  ((= n (flowchart->num-nodes fl)) (cons def-use def-def))
(let* ((node (vector-ref (flowchart->nodes fl) n))
       (reads (node->reads node))
       (writes (node->writes node))
       (reaching-defs (vector-ref rd-e n)))
  (map (lambda (reaching-def)
 (let ((var (car reaching-def))
       (i (index defs reaching-def)))
   (begin
     (if (member? var reads)
 (add-entry def-use i n))
     (if (member? var writes)
 (add-entry def-def i n)))))
       reaching-defs))))))

(define-record proto-dep (src dst kind))

(define proto-deps
  (lambda (def uses defs)
    (let* ((var (car def))
   (label (cdr def))
   (flow-deps
    (if (equal? label '?)
'()
(map (lambda (use) (make-proto-dep label use 'flow)) uses)))
```

```
   (input-deps
    (map-pairs (lambda (src dst) (make-proto-dep src dst 'input))
       uses))
   (anti-deps
    (filter (lambda (pd)
      (not (equal? (proto-dep->src pd) (proto-dep->dst pd))))
    (map-product (lambda (src dst) (make-proto-dep src dst 'anti))
 uses defs)))
   (output-deps
    (if (equal? label '?)
'()
(map (lambda (def-) (make-proto-dep label def- 'output)) defs))))
      (append flow-deps input-deps anti-deps output-deps))))

(define new-dep
  (lambda (src dst)
    (make-dep src dst #f 0 #f)))

(define dependencies
  (lambda (fl defs def-use def-def)
    (let* ((deps '())
   (edges '())
   (add-dep
    (lambda (src dst fp weight dir)
      (let* ((edge (cons src dst))
     (dep (if (member? edge edges)
      (nth deps (index edges edge))
      (begin
(set! deps (cons (new-dep src dst) deps))
(set! edges (cons edge edges))
(car deps)))))
(update-dep dep fp weight dir)))))
      (do ((i 0 (1+ i))) ((= i (length defs)) deps)
(let* ((def (nth defs i))
       (var (car def))
       (var-is-vector (equal? (flowchart-var->type fl var) 'vector))
         (new-deps (proto-deps
  def (vector-ref def-use i) (vector-ref def-def i)))
       (weight (if var-is-vector 100 1)))
  (map (lambda (pd)
 (let* ((src (proto-dep->src pd))
(dst (proto-dep->dst pd))
(kind (proto-dep->kind pd))
(nodes-unfuseable (or (unfuseable fl src)
      (unfuseable fl dst)))
(weight- (case kind
   ((input flow) weight)
   ((anti output) 0)))
(fp (case kind
      ((input)
       ;nodes-unfuseable
       #f)
      ((flow anti output)
       (or nodes-unfuseable
   (not var-is-vector)))))
(dir (case kind
       ((input) #f)
       ((flow anti output) #t))))
   (add-dep src dst fp weight- dir)))
       new-deps))))))

(define update-dep
  (lambda (dep fp weight dir)
    (begin
      (set-dep-fp! dep (or (dep-fp? dep) fp))
      (set-dep-dir! dep (or (dep-dir? dep) dir))
      (set-dep-weight! dep (+ (dep->weight dep) weight))
      (if (dep-fp? dep) (set-dep-weight! dep epsilon)))))
```

A.4 fusion.scm

```
(require 'struct)

(define-record dep (src dst fp weight dir))

(define epsilon 0.0001)

(define dep-dst?
  (lambda (dst)
    (lambda (dep)
      (equal? dst (dep->dst dep)))))

(define dep-src?
  (lambda (src)
    (lambda (dep)
      (equal? src (dep->src dep)))))

(define dep-fp? dep->fp)

(define dep-dir? dep->dir)

(define greedy
  (lambda (nodes deps)
    (let* ((deps- (filter dep-dir? deps))
   (num-nodes (length nodes))
   (last-node (1- num-nodes))
   (preds (lambda (node) (map dep->src (filter (dep-dst? node) deps-))))
   (no-preds (filter (lambda (node) (null? (preds node))) nodes))
   (mapping (make-vector num-nodes))
   (visited '())
   (num-unvisited num-nodes))
      (do ((partition 0 (1+ partition))) ((= num-unvisited 0) mapping)
(if (equal? num-unvisited 1)
    (begin
      ; The last node (being a dummy) always goes in a partition
      ; by itself at the end
      (vector-set! mapping last-node partition)
      (set! num-unvisited 0))
    (let* ((not-ok '())
   (process (lambda (w fp)
      (begin
(if (subset? (preds w) visited)
    (set! no-preds (adjoin w no-preds)))
(if fp
    (set! not-ok (adjoin w not-ok))))))
      (do ((i 0 (1+ i))) ((>= i last-node))
(let* ((v (nth nodes i))
       (deps-- (filter (dep-src? v) deps-)))
  (if (and (member? v no-preds)
   (not (member? v not-ok)))
      (begin
(vector-set! mapping i partition)
(set! visited (adjoin v visited))
(set! num-unvisited (1- num-unvisited))
(set! no-preds (remove v no-preds))
(map (lambda (dep)
       (process (dep->dst dep) (dep->fp dep)))
     deps--))))))))))

(define mapping->partitions
  (lambda (nodes mapping)
    (let* ((num-nodes (vector-length mapping))
   (last-node (1- num-nodes))
   (num-partitions (1+ (vector-max mapping)))
   (partitions (make-vector num-partitions '())))
      (do ((i last-node (1- i))) ((< i 0) partitions)
(let ((node (nth nodes i))
      (p (vector-ref mapping i)))
  (vector-set! partitions p
       (adjoin node (vector-ref partitions p))))))))

(define transitive-successors
  (lambda (nodes deps)
    (let* ((num-nodes (length nodes))
   (succ-array (make-vector num-nodes '())))
      (do ((i (- num-nodes 1) (1- i))) ((< i 0) succ-array)
(let ((node (nth nodes i)))
  (map (lambda (dep)
 (if (and (equal? node (dep->src dep))
  (dep-dir? dep))
     (let* ((dst (dep->dst dep))
    (j (index nodes dst)))
       (vector-set!
succ-array i
(union (vector-ref succ-array i)
       (adjoin dst (vector-ref succ-array j)))))))
       deps))))))

(define improve!
  (lambda (nodes deps mapping)
    (let* ((partitions (mapping->partitions nodes mapping))
   (num-nodes (vector-length mapping))
   (num-partitions (vector-length partitions))
   (node->index
    (lambda (n) (index nodes n)))
   (node->partition
    (lambda (n) (vector-ref mapping (index nodes n))))
   (trans-succs (transitive-successors nodes deps))
   (must-preceed?
    (lambda (p1 p2)
      (and (not (equal? p1 p2))
   (let ((par1 (vector-ref partitions p1))
 (par2 (vector-ref partitions p2)))
     (exists?
      (lambda (n1)
(let ((tr-succs (vector-ref trans-succs (index nodes n1))))
  (exists?
   (lambda (n2)
     (member? n2 tr-succs))
   par2)))
      par1)))))
   (profit
    (lambda (n p)
      (let* ((partition (vector-ref partitions p))
     (src-in-p? (lambda (dep) (member? (dep->src dep) partition)))
     (dst-in-p? (lambda (dep) (member? (dep->dst dep) partition))))
```

```
(sum (filtermap (or-? (and-? (dep-src? n) dst-in-p?)
      (and-? (dep-dst? n) src-in-p?))
dep->weight deps)))))
   (move-node
    (lambda (n p1 p2)
      (let ((i (index nodes n)))
(begin
  (print "moving node" n "from" p1 "to" p2)
  (vector-set! mapping i p2)
  (vector-set! partitions p1
       (remove n (vector-ref partitions p1)))
  (vector-set! partitions p2
       (adjoin n (vector-ref partitions p2))))))))
      ; Step backwards through the partions
      (do ((p (vector-max mapping) (1- p))) ((< p 0))
;      (do ((p 7 (1- p))) ((= p 6))
; Step backwards through the nodes in the current partition
; The first node (being a dummy) is never moved.
(do ((i (1- num-nodes) (1- i))) ((< i 1))
  (if (equal? p (vector-ref mapping i))
      (let* ((n (nth nodes i))
     (succs (filtermap (and-? (dep-src? n) dep-dir?)
       dep->dst deps))
     (succ-p (map-s node->partition succs))
     ; FIXME: fp-succs expression doesn't use dep-dir?
     ; actually, all fp edges are directed
     (fp-succs (filtermap (and-? (dep-src? n) dep-fp?)
  dep->dst deps))
     (fp-succ-p (map-s node->partition fp-succs)))
(if (not (intersect? succs (vector-ref partitions p)))
    (let* ((best-p p)
   (best-weight (profit n p)))
      (begin
(do ((p- (vector-max mapping) (1- p-))) ((<= p- p))
  (if (not (or (member? p- fp-succ-p) ; if safe
       (exists? (lambda (p--)
  (< p-- p-))
;(must-preceed? p-- p-)
succ-p)))
      (let ((weight (profit n p-)))
(if (>= weight best-weight) ; test if profitable
    (begin
      (set! best-p p-)
      (set! best-weight weight))))))
(if (not (equal? p best-p))
    (move-node n p best-p)))))))))))

(define schedule!
  (lambda (fl posns mapping)
    (let* ((partitions (mapping->partitions posns mapping))
   (num-partitions (vector-length partitions))
   (num-nodes (length posns))
   (nodes (flowchart->nodes fl))
   (node-vec (make-vector num-nodes))
   (i 0)
   (add-node (lambda (node)
       (begin
 (vector-set! node-vec i node)
 (set! i (1+ i))))))
      (begin
; Put the node records into the correct order, and tag
; them as which partition they will go into.
; For each partition, first have the nodes that cannot be
; fused (ie not single loops) then the ones that can
(do ((p 0 (1+ p))) ((>= p num-partitions))
  (let* ((partition-nodes (vector-ref partitions p))
 (posn->node (lambda (posn) (vector-ref nodes posn)))
 (not-fuse? (lambda (n) (unfuseable fl n)))
 (not-fuse (filtermap not-fuse? posn->node partition-nodes))
 (fuse (filtermap (not-? not-fuse?) posn->node partition-nodes)))
    (begin
      (map add-node not-fuse))
      (if (> (length fuse) 1)
  ; it only fuse loops if there's more than one of them
  (begin
    (set-node-fusion! (car fuse) 'fusion-begin)
    (add-node (car fuse))
    (map (lambda (node)
   (begin
     (set-node-fusion! node 'fusion-continue)
     (add-node node)))
 (cdr fuse)))
  (map add-node fuse))))
(assert (= i num-nodes) "schedule!: internal error")
; Copy the nodes in the correct order back into the flowchart
(do ((i 0 (1+ i))) ((>= i num-nodes))
  (let ((node (vector-ref node-vec i))
(posn (nth posns i)))
    (vector-set! nodes posn node)))))))
```

A.5 utils.scm

```
(define assert
  (lambda (c msg)
    (if (not c) (error msg))))


(define nothing
  (lambda x
    (if #f (error "False is true!!!"))))

(define multicons
  (letrec ((mc
    (lambda (list)
      (if (null? (cdr list))
  (car list)
  (cons (car list) (mc (cdr list)))))))
  (lambda x
    (if (pair? x)
(mc x)
(error "multicons: wrong number of arguments")))))

(define map2
  (lambda (f l1 l2)
    (letrec
((m2
  (lambda (l1 l2)
    (if (null? l1)
'()
(cons (f (car l1) (car l2)) (m2 (cdr l1) (cdr l2)))))))
      (m2 l1 l2))))

(define filter
  (lambda (pred? l)
    (letrec
((f
  (lambda (l)
    (cond
     ((null? l) '())
     ((pred? (car l))
      (cons (car l) (f (cdr l))))
     (else (f (cdr l)))))))
      (begin
(if util-debug (print pred? l))
(f l)))))

(define filtermap
  (lambda (pred? func l)
    (map func (filter pred? l))))

(define mapfilter
  (lambda (func pred? l)
    (filter pred? (map func l))))

(define fold
  (lambda (func unit l)
    (letrec ((f
      (lambda (l res)
(cond
 ((null? l) res)
 (else (f (cdr l) (func (car l) res)))))))
      (f l unit))))


(define map-s
  (lambda (func l)
    (setify (map func l))))

(define forall?
  (lambda (pred? l)
    (cond
     ((null? l) #t)
     (else (and (pred? (car l)) (forall? pred? (cdr l)))))))

(define exists?
  (lambda (pred? l)
    (cond
     ((null? l) #f)
     (else (or (pred? (car l)) (exists? pred? (cdr l)))))))

(define member?
  (lambda (el l)
    (letrec
((m?
  (lambda (l)
    (cond
     ((null? l) #f)
     ((equal? (car l) el) #t)
     (else (m? (cdr l)))))))
      (m? l))))

(define adjoin
  (lambda (el l)
    (if (not (member? el l))
(cons el l)
```

```
l)))

(define setify
  (lambda (l)
    (cond
     ((null? l) l)
     (else (adjoin (car l) (setify (cdr l)))))))

(define union
  (lambda (l1 l2)
    (append l1 (filter (lambda (x) (not (member? x l1))) l2))))

(define sub
  (lambda (l1 l2)
    (filter (lambda (x) (not (member? x l2))) l1)))

(define intersect?
  (lambda (l1 l2)
    (cond
     ((null? l1) #f)
     ((member? (car l1) l2) #t)
     (else (intersect? (cdr l1) l2)))))

(define subset?
  (lambda (l1 l2)
    (cond
     ((null? l1) #t)
     ((member? (car l1) l2) (subset? (cdr l1) l2))
     (else #f))))

(define remove
  (lambda (el l)
    (cond
     ((null? l) l)
     ((equal? (car l) el) (cdr l)) ; assume at most one occurrence
     (else (cons (car l) (remove el (cdr l)))))))

(define *undefined* '*undefined*)

(define range
  (lambda (lo hi)
    (if (>= lo hi)
'()
(cons lo (range (1+ lo) hi)))))

(define nth
  (lambda (l n)
    (if (= n 0)
(car l)
(nth (cdr l) (1- n)))))

(define index
  (lambda (l el)
    (letrec ((index-iter
      (lambda (l- i)
(cond
 ((null? l-)
  (error "index: item not found" l el))
 ((equal? (car l-) el)
  i)
 (else
  (index-iter (cdr l-) (1+ i)))))))
```

```
      (index-iter l 0))))

(define sum
  (letrec ((sum-iter
    (lambda (l acc)
      (cond
       ((null? l) acc)
       (else (sum-iter (cdr l) (+ (car l) acc)))))))
    (lambda (l) (sum-iter l 0))))

(define list-max
  (lambda (l)
    (cond
     ((null? l) 0)
     (else (max (car l) (list-max (cdr l)))))))

(define vector-max
  (lambda (v)
    (list-max (vector->list v))))

(define vector-copy
  (lambda (v1)
    (let ((v2 (make-vector (vector-length v1))))
      (do ((i 0 (1+ i))) ((>= i (vector-length v1)) v2)
(vector-set! v2 i (vector-ref v1 i))))))

(define and-?
  (lambda (p1 p2)
    (lambda (x)
      (and (p1 x) (p2 x)))))

(define or-?
  (lambda (p1 p2)
    (lambda (x)
      (or (p1 x) (p2 x)))))

(define not-?
  (lambda (p)
    (lambda (x)
      (not (p x)))))

(define map-pairs
  (lambda (func l)
    (letrec ((cross-prod-iter
      (lambda (el l ac)
(cond
 ((null? l) ac)
 (else
  (cross-prod-iter el (cdr l) (cons (func el (car l)) ac))))))
   (pairs-iter
    (lambda (l ac)
      (cond
       ((null? l) ac)
       (else
(pairs-iter (cdr l) (cross-prod-iter (car l) (cdr l) ac)))))))
      (reverse (pairs-iter l '())))))

(define map-product
  (lambda (func l1 l2)
    (letrec ((inner
      (lambda (el l ac)
(cond
 ((null? l) ac)
 (else
  (inner el (cdr l) (cons (func el (car l)) ac))))))
     (outer
      (lambda (l1 l2 acc)
(cond
 ((null? l1) acc)
 (else
  (outer (cdr l1) l2 (inner (car l1) l2 acc)))))))
      (reverse (outer l1 l2 '())))))
```

A.6 all.scm

```
(begin
  (load "utils.scm")
  (load "parsing.scm")
  (load "analysing.scm")
  (load "data-flow.scm")
  (load "fusion.scm")
;  (load "qmr.scm")
  (load "cg.scm")
  )

(define fl
  (analyse-prgm (parse-prgm cg-code)))
;  (analyse-prgm (parse-prgm qmr-code)))
(define rde (loop-rd-entry fl))
(define defs (all-defs rde))
(define ch (chains fl rde defs))
(define def-use (car ch))
(define def-def (cdr ch))
(define posns (range (flowchart-label->posn fl 'rest-loop-begin)
     (1+ (flowchart-label->posn fl 'rest-loop-end))))
(define deps (dependencies fl defs def-use def-def))
(define mapping (greedy posns deps))

(define do-stuff1
  (lambda ()
    (schedule! fl posns mapping)))

(define do-stuff2
  (lambda ()
    (begin
      (improve! posns deps mapping)
      (schedule! fl posns mapping))))

(define show-reads-writes
  ; Give this routine a list of numbers refering to nodes that are to be
  ; fused and it will say which variables have to be read and written by
  ; the resulting loop.
  (lambda (fl nodes lve)
    (letrec ((show (lambda (nodes lv reads writes)
     (if (null? nodes)
 (begin
   (print "reads" reads)
   (print "writes" writes)
   (print "live writes" (intersection writes lv))
   '())
 (let* ((posn (car nodes))
(rest (cdr nodes))
(node (flowchart->node fl posn)))
   (show rest (vector-ref lve posn)
 (union reads (sub (node->reads node) writes))
 (union writes (node->writes node))))))))
      (show nodes '() '() '()))))
```

B Benchmark sources and flowcharts

In the printed form of flowcharts, when a node has a * directly after its node number this means that it is the first node in a fusion partition, and if it has a | then it belongs in an on-going fusion partition.

The the flowcharts showing programs after loop fusion has been performed, only the effected block (`rest-loop`) has been shown.

B.1 Conjugate gradient descent

B.1.1 Source

```
(define cg-code
  '((declarations (A) (b x p q r res z) (alpha beta bnrm2 err rho rho_o))
    (setup (initialise A x b)
           (:= rho_o 1.0)
           (dcopy b r)
           (dgemv "T" -1.0 A x 1.0 r)
           (:= bnrm2 (dnrm2 b))
           (:= bnrm2 (nonzero bnrm2)))
    (loop (dcopy r z)
          (:= rho (ddot r z))
          (first-time
           ((dcopy z p))
           ((:= beta (/ rho rho_o))
            (daxpy beta p z)
            (dcopy z p)))
          (dgemv "T" 1.0 A p 0.0 q)
          (:= alpha (/ rho (ddot q p)))
          (daxpy alpha p x)
          (daxpy (- alpha) q r)
          (:= rho_o rho)
          (:= err (/ (dnrm2 r) bnrm2)))
    (finish (dcopy b res)
            (dgemv "T" 1.0 A x -1.0 res)
            (:= err (dnrm2 res))
            (results b x))))
```

B.1.2 Flowchart — no fusion

```
(0 label setup-begin)
(1 initialise (a x b))
(2 assign (rho_o #(lit 1.0)))
(3 copy (b r))
(4 mvmul-t (#(lit -1.0) a x r))
(5 assign (tmp7 #(lit 0.0)))
(6 dot (tmp7 b b))
(7 assign (tmp6 #(fun-call sqrt (#(var-ref tmp7)))))
(8 assign (bnrm2 #(var-ref tmp6)))
(9 assign (bnrm2 #(fun-call nonzero (#(var-ref bnrm2)))))
(10 label setup-end)
(11 label first-loop-begin)
(12 copy (r z))
(13 assign (tmp8 #(lit 0.0)))
(14 dot (tmp8 r z))
(15 assign (rho #(var-ref tmp8)))
(16 copy (z p))
(17 zero (q))
(18 mvmul-t (#(lit 1.0) a p q))
(19 assign (tmp9 #(lit 0.0)))
```

```
(20 dot (tmp9 q p))
(21 assign (alpha #(aexpr / (#(var-ref rho) #(var-ref tmp9)))))
(22 axpy (#(var-ref alpha) p x))
(23 assign (tmp10 #(aexpr neg (#(var-ref alpha)))))
(24 axpy (#(var-ref tmp10) q r))
(25 assign (rho_o #(var-ref rho)))
(26 assign (tmp12 #(lit 0.0)))
(27 dot (tmp12 r r))
(28 assign (tmp11 #(fun-call sqrt (#(var-ref tmp12)))))
(29 assign (err #(aexpr / (#(var-ref tmp11) #(var-ref bnrm2)))))
(30 label first-loop-end)
(31 label rest-loop-begin)
(32 copy (r z))
(33 assign (tmp13 #(lit 0.0)))
(34 dot (tmp13 r z))
(35 assign (rho #(var-ref tmp13)))
(36 assign (beta #(aexpr / (#(var-ref rho) #(var-ref rho_o)))))
(37 axpy (#(var-ref beta) p z))
(38 copy (z p))
(39 zero (q))
(40 mvmul-t (#(lit 1.0) a p q))
(41 assign (tmp14 #(lit 0.0)))
(42 dot (tmp14 q p))
(43 assign (alpha #(aexpr / (#(var-ref rho) #(var-ref tmp14)))))
(44 axpy (#(var-ref alpha) p x))
(45 assign (tmp15 #(aexpr neg (#(var-ref alpha)))))
(46 axpy (#(var-ref tmp15) q r))
(47 assign (rho_o #(var-ref rho)))
(48 assign (tmp17 #(lit 0.0)))
(49 dot (tmp17 r r))
(50 assign (tmp16 #(fun-call sqrt (#(var-ref tmp17)))))
(51 assign (err #(aexpr / (#(var-ref tmp16) #(var-ref bnrm2)))))
(52 label rest-loop-end)
(53 label finish-begin)
(54 copy (b res))
(55 neg (res))
(56 mvmul-t (#(lit 1.0) a x res))
(57 assign (tmp19 #(lit 0.0)))
(58 dot (tmp19 res res))
(59 assign (tmp18 #(fun-call sqrt (#(var-ref tmp19)))))
(60 assign (err #(var-ref tmp18)))
(61 results (b x))
(62 label finish-end)
```

B.1.3 Flowchart — greedy partitioning

```
(31 label rest-loop-begin)
(32 assign (tmp13 #(lit 0.0)))
(33 assign (tmp14 #(lit 0.0)))
(34 assign (tmp17 #(lit 0.0)))
(35 * copy (r z))
(36 | zero (q))
(37 dot (tmp13 r z))
(38 assign (rho #(var-ref tmp13)))
(39 assign (beta #(aexpr / (#(var-ref rho) #(var-ref rho_o)))))
(40 assign (rho_o #(var-ref rho)))
(41 * axpy (#(var-ref beta) p z))
(42 | copy (z p))
(43 mvmul-t (#(lit 1.0) a p q))
(44 dot (tmp14 q p))
(45 assign (alpha #(aexpr / (#(var-ref rho) #(var-ref tmp14)))))
(46 assign (tmp15 #(aexpr neg (#(var-ref alpha)))))
```

```
(47 axpy (#(var-ref alpha) p x))
(48 * axpy (#(var-ref tmp15) q r))
(49 | dot (tmp17 r r))
(50 assign (tmp16 #(fun-call sqrt (#(var-ref tmp17)))))
(51 assign (err #(aexpr / (#(var-ref tmp16) #(var-ref bnrm2)))))
(52 label rest-loop-end)
```

B.1.4 Flowchart — improved partitioning

```
(31 label rest-loop-begin)
(32 assign (tmp13 #(lit 0.0)))
(33 assign (tmp14 #(lit 0.0)))
(34 assign (tmp17 #(lit 0.0)))
(35 * copy (r z))
(36 | dot (tmp13 r z))
(37 | zero (q))
(38 assign (rho #(var-ref tmp13)))
(39 assign (beta #(aexpr / (#(var-ref rho) #(var-ref rho_o)))))
(40 * axpy (#(var-ref beta) p z))
(41 | copy (z p))
(42 mvmul-t (#(lit 1.0) a p q))
(43 dot (tmp14 q p))
(44 assign (alpha #(aexpr / (#(var-ref rho) #(var-ref tmp14)))))
(45 assign (rho_o #(var-ref rho)))
(46 assign (tmp15 #(aexpr neg (#(var-ref alpha)))))
(47 axpy (#(var-ref alpha) p x))
(48 * axpy (#(var-ref tmp15) q r))
(49 | dot (tmp17 r r))
(50 assign (tmp16 #(fun-call sqrt (#(var-ref tmp17)))))
(51 assign (err #(aexpr / (#(var-ref tmp16) #(var-ref bnrm2)))))
(52 label rest-loop-end)
```

B.2 Quasi-Minimal Residual

B.2.1 Source

```
(define qmr-code
  '((declarations
     (A)
     (b d x p ptld q qtld r rtld res s v vtld w wtld y ytld z ztld)
     (beta bnrm2 delta epsilon err eta gamma gamma_o rho rhorecip
           scalds scalp scalq theta theta_o xi xirecip rho_o))
    (setup
     (initialise A b x)
     (:= rho_o 1.0)
     (dcopy b r)
     (dgemv "T" -1.0 a x 1.0 r)
     (:= bnrm2 (dnrm2 b))
     (:= bnrm2 (nonzero bnrm2))
     (dcopy r vtld)
     (dcopy vtld y)
     (:= rho (dnrm2 y))
     (dcopy r wtld)
     (dcopy wtld z)
     (:= xi (dnrm2 z))
     (:= gamma 1.0)
     (:= eta -1.0)
     (:= epsilon 0.0)
     (:= theta 0.0))
    (loop
     (:= rhorecip (/ 1.0 rho))
     (dcopy vtld v)
```

```
(dscal rhorecip v)
(dscal rhorecip y)

(:= xirecip (/ 1.0 xi))
(dcopy wtld w)
(dscal xirecip w)
(dscal xirecip z)

(:= delta (ddot z y))

(dcopy y ytld)
(dcopy z ztld)

(first-time
 ((dcopy ytld p)
  (dcopy ztld q))

 ((:= scalp (- (* xi (/ delta epsilon))))
  (daxpy scalp p ytld)
  (dcopy ytld p)

  (:= scalq (- (* rho (/ delta epsilon))))
  (daxpy scalq q ztld)
  (dcopy ztld q)))

(dgemv "T" 1.0 A p 0.0 ptld)

(:= epsilon (ddot ptld q))
(:= beta (/ epsilon delta))

(dcopy ptld vtld)
(daxpy (- beta) v vtld)

(dcopy vtld y)
(:= rho_o rho)
(:= rho (dnrm2 y))

(dcopy w wtld)
(dgemv "N" 1.0 A q (- beta) wtld)

(dcopy wtld z)
(:= xi (dnrm2 z))
(:= theta_o theta)
(:= theta (/ rho (* gamma (fabs beta))))
(:= gamma_o gamma)
(:= gamma (/ 1.0 (sqrt (+ 1.0 (* theta theta)))))

(:= eta (/ (* (* (- eta) rho_o) (* gamma gamma))
           (* beta (* gamma_o gamma_o))))

(first-time
 ((dcopy p d)
  (dscal eta d)
  (dcopy ptld s)
  (dscal eta s))
 ((:= scalds (* theta_o gamma))
  (:= scalds (* scalds scalds))
  (dscal scalds d)
  (daxpy eta p d)
  (dscal scalds s)
  (daxpy eta ptld s)))
```

```
     (daxpy 1.0 d x)
     (daxpy -1.0 s r)

     (:= err (/ (dnrm2 r) bnrm2)))

    (finish
     (dcopy b res)
     (dgemv "T" 1.0 A x -1.0 res)
     (:= err (dnrm2 res))
     (results err))))
```

B.2.2 Flowchart — no fusion

```
(0 label setup-begin)
(1 initialise (a b x))
(2 assign (rho_o #(lit 1.0)))
(3 copy (b r))
(4 mvmul-t (#(lit -1.0) a x r))
(5 assign (tmp19 #(lit 0.0)))
(6 dot (tmp19 b b))
(7 assign (tmp18 #(fun-call sqrt (#(var-ref tmp19)))))
(8 assign (bnrm2 #(var-ref tmp18)))
(9 assign (bnrm2 #(fun-call nonzero (#(var-ref bnrm2)))))
(10 copy (r vtld))
(11 copy (vtld y))
(12 assign (tmp21 #(lit 0.0)))
(13 dot (tmp21 y y))
(14 assign (tmp20 #(fun-call sqrt (#(var-ref tmp21)))))
(15 assign (rho #(var-ref tmp20)))
(16 copy (r wtld))
(17 copy (wtld z))
(18 assign (tmp23 #(lit 0.0)))
(19 dot (tmp23 z z))
(20 assign (tmp22 #(fun-call sqrt (#(var-ref tmp23)))))
(21 assign (xi #(var-ref tmp22)))
(22 assign (gamma #(lit 1.0)))
(23 assign (eta #(lit -1.0)))
(24 assign (epsilon #(lit 0.0)))
(25 assign (theta #(lit 0.0)))
(26 label setup-end)
(27 label first-loop-begin)
(28 assign (rhorecip #(aexpr / (#(lit 1.0) #(var-ref rho)))))
(29 copy (vtld v))
(30 scal (#(var-ref rhorecip) v))
(31 scal (#(var-ref rhorecip) y))
(32 assign (xirecip #(aexpr / (#(lit 1.0) #(var-ref xi)))))
(33 copy (wtld w))
(34 scal (#(var-ref xirecip) w))
(35 scal (#(var-ref xirecip) z))
(36 assign (tmp24 #(lit 0.0)))
(37 dot (tmp24 z y))
(38 assign (delta #(var-ref tmp24)))
(39 copy (y ytld))
(40 copy (z ztld))
(41 copy (ytld p))
(42 copy (ztld q))
(43 zero (ptld))
(44 mvmul-t (#(lit 1.0) a p ptld))
(45 assign (tmp25 #(lit 0.0)))
(46 dot (tmp25 ptld q))
(47 assign (epsilon #(var-ref tmp25)))
(48 assign (beta #(aexpr / (#(var-ref epsilon) #(var-ref delta)))))
```

```
(49 copy (ptld vtld))
(50 assign (tmp26 #(aexpr neg (#(var-ref beta)))))
(51 axpy (#(var-ref tmp26) v vtld))
(52 copy (vtld y))
(53 assign (rho_o #(var-ref rho)))
(54 assign (tmp28 #(lit 0.0)))
(55 dot (tmp28 y y))
(56 assign (tmp27 #(fun-call sqrt (#(var-ref tmp28)))))
(57 assign (rho #(var-ref tmp27)))
(58 copy (w wtld))
(59 assign (tmp29 #(aexpr neg (#(var-ref beta)))))
(60 scal (#(var-ref tmp29) wtld))
(61 mvmul-n (#(lit 1.0) a q wtld))
(62 copy (wtld z))
(63 assign (tmp31 #(lit 0.0)))
(64 dot (tmp31 z z))
(65 assign (tmp30 #(fun-call sqrt (#(var-ref tmp31)))))
(66 assign (xi #(var-ref tmp30)))
(67 assign (theta_o #(var-ref theta)))
(68 assign (theta #(aexpr / (#(var-ref rho) #(aexpr * (#(var-ref gamma) #(fun-call fabs (#(var-ref beta)))))))
(69 assign (gamma_o #(var-ref gamma)))
(70 assign (gamma #(aexpr / (#(lit 1.0) #(fun-call sqrt (#(aexpr + (#(lit 1.0) #(aexpr * (#(var-ref theta) #(va
(71 assign (eta #(aexpr / (#(aexpr * (#(aexpr * (#(aexpr neg (#(var-ref eta))) #(var-ref rho_o))) #(aexpr * (#(
(72 copy (p d))
(73 scal (#(var-ref eta) d))
(74 copy (ptld s))
(75 scal (#(var-ref eta) s))
(76 add-assign (d x))
(77 sub-assign (s r))
(78 assign (tmp33 #(lit 0.0)))
(79 dot (tmp33 r r))
(80 assign (tmp32 #(fun-call sqrt (#(var-ref tmp33)))))
(81 assign (err #(aexpr / (#(var-ref tmp32) #(var-ref bnrm2)))))
(82 label first-loop-end)
(83 label rest-loop-begin)
(84 assign (rhorecip #(aexpr / (#(lit 1.0) #(var-ref rho)))))
(85 copy (vtld v))
(86 scal (#(var-ref rhorecip) v))
(87 scal (#(var-ref rhorecip) y))
(88 assign (xirecip #(aexpr / (#(lit 1.0) #(var-ref xi)))))
(89 copy (wtld w))
(90 scal (#(var-ref xirecip) w))
(91 scal (#(var-ref xirecip) z))
(92 assign (tmp34 #(lit 0.0)))
(93 dot (tmp34 z y))
(94 assign (delta #(var-ref tmp34)))
(95 copy (y ytld))
(96 copy (z ztld))
(97 assign (scalp #(aexpr neg (#(aexpr * (#(var-ref xi) #(aexpr / (#(var-ref delta) #(var-ref epsilon))))))))
(98 axpy (#(var-ref scalp) p ytld))
(99 copy (ytld p))
(100 assign (scalq #(aexpr neg (#(aexpr * (#(var-ref rho) #(aexpr / (#(var-ref delta) #(var-ref epsilon))))))))
(101 axpy (#(var-ref scalq) q ztld))
(102 copy (ztld q))
(103 zero (ptld))
(104 mvmul-t (#(lit 1.0) a p ptld))
(105 assign (tmp35 #(lit 0.0)))
(106 dot (tmp35 ptld q))
(107 assign (epsilon #(var-ref tmp35)))
(108 assign (beta #(aexpr / (#(var-ref epsilon) #(var-ref delta)))))
(109 copy (ptld vtld))
(110 assign (tmp36 #(aexpr neg (#(var-ref beta)))))
```

```
(111 axpy (#(var-ref tmp36) v vtld))
(112 copy (vtld y))
(113 assign (rho_o #(var-ref rho)))
(114 assign (tmp38 #(lit 0.0)))
(115 dot (tmp38 y y))
(116 assign (tmp37 #(fun-call sqrt (#(var-ref tmp38)))))
(117 assign (rho #(var-ref tmp37)))
(118 copy (w wtld))
(119 assign (tmp39 #(aexpr neg (#(var-ref beta)))))
(120 scal (#(var-ref tmp39) wtld))
(121 mvmul-n (#(lit 1.0) a q wtld))
(122 copy (wtld z))
(123 assign (tmp41 #(lit 0.0)))
(124 dot (tmp41 z z))
(125 assign (tmp40 #(fun-call sqrt (#(var-ref tmp41)))))
(126 assign (xi #(var-ref tmp40)))
(127 assign (theta_o #(var-ref theta)))
(128 assign (theta #(aexpr / (#(var-ref rho) #(aexpr * (#(var-ref gamma) #(fun-call fabs (#(var-ref beta))))))
(129 assign (gamma_o #(var-ref gamma)))
(130 assign (gamma #(aexpr / (#(lit 1.0)
                              #(fun-call sqrt (#(aexpr + (#(lit 1.0)
                                                          #(aexpr * (#(var-ref theta) #(var-ref theta)))))))))
(131 assign (eta #(aexpr / (#(aexpr * (#(aexpr * (#(aexpr neg (#(var-ref eta))) #(var-ref rho_o)))
                                       #(aexpr * (#(var-ref gamma) #(var-ref gamma)))))
                            #(aexpr * (#(var-ref beta)
                                       #(aexpr * (#(var-ref gamma_o) #(var-ref gamma_o)))))))))
(132 assign (scalds #(aexpr * (#(var-ref theta_o) #(var-ref gamma)))))
(133 assign (scalds #(aexpr * (#(var-ref scalds) #(var-ref scalds)))))
(134 scal (#(var-ref scalds) d))
(135 axpy (#(var-ref eta) p d))
(136 scal (#(var-ref scalds) s))
(137 axpy (#(var-ref eta) ptld s))
(138 add-assign (d x))
(139 sub-assign (s r))
(140 assign (tmp43 #(lit 0.0)))
(141 dot (tmp43 r r))
(142 assign (tmp42 #(fun-call sqrt (#(var-ref tmp43)))))
(143 assign (err #(aexpr / (#(var-ref tmp42) #(var-ref bnrm2)))))
(144 label rest-loop-end)
(145 label finish-begin)
(146 copy (b res))
(147 neg (res))
(148 mvmul-t (#(lit 1.0) a x res))
(149 assign (tmp45 #(lit 0.0)))
(150 dot (tmp45 res res))
(151 assign (tmp44 #(fun-call sqrt (#(var-ref tmp45)))))
(152 assign (err #(var-ref tmp44)))
(153 results (err))
(154 label finish-end)
```

B.2.3 Flowchart — greedy partitioning

```
(83 label rest-loop-begin)
(84 assign (rhorecip #(aexpr / (#(lit 1.0) #(var-ref rho)))))
(85 assign (xirecip #(aexpr / (#(lit 1.0) #(var-ref xi)))))
(86 assign (tmp34 #(lit 0.0)))
(87 assign (tmp35 #(lit 0.0)))
(88 assign (rho_o #(var-ref rho)))
(89 assign (tmp38 #(lit 0.0)))
(90 assign (tmp41 #(lit 0.0)))
(91 assign (theta_o #(var-ref theta)))
(92 assign (gamma_o #(var-ref gamma)))
```

```
(93 assign (tmp43 #(lit 0.0)))
(94 * copy (vtld v))
(95 | copy (wtld w))
(96 | zero (ptld))
(97 * scal (#(var-ref rhorecip) v))
(98 | scal (#(var-ref rhorecip) y))
(99 | scal (#(var-ref xirecip) w))
(100 | scal (#(var-ref xirecip) z))
(101 | dot (tmp34 z y))
(102 | copy (y ytld))
(103 | copy (z ztld))
(104 | copy (w wtld))
(105 assign (delta #(var-ref tmp34)))
(106 assign (scalp #(aexpr neg (#(aexpr * (#(var-ref xi) #(aexpr / (#(var-ref delta) #(var-ref epsilon))))))))
(107 assign (scalq #(aexpr neg (#(aexpr * (#(var-ref rho) #(aexpr / (#(var-ref delta) #(var-ref epsilon))))))))
(108 * axpy (#(var-ref scalp) p ytld))
(109 | copy (ytld p))
(110 | axpy (#(var-ref scalq) q ztld))
(111 | copy (ztld q))
(112 mvmul-t (#(lit 1.0) a p ptld))
(113 * dot (tmp35 ptld q))
(114 | copy (ptld vtld))
(115 assign (epsilon #(var-ref tmp35)))
(116 assign (beta #(aexpr / (#(var-ref epsilon) #(var-ref delta)))))
(117 assign (tmp36 #(aexpr neg (#(var-ref beta)))))
(118 assign (tmp39 #(aexpr neg (#(var-ref beta)))))
(119 * axpy (#(var-ref tmp36) v vtld))
(120 | copy (vtld y))
(121 | dot (tmp38 y y))
(122 | scal (#(var-ref tmp39) wtld))
(123 assign (tmp37 #(fun-call sqrt (#(var-ref tmp38)))))
(124 mvmul-n (#(lit 1.0) a q wtld))
(125 assign (rho #(var-ref tmp37)))
(126 * copy (wtld z))
(127 | dot (tmp41 z z))
(128 assign (tmp40 #(fun-call sqrt (#(var-ref tmp41)))))
(129 assign (theta #(aexpr / (#(var-ref rho) #(aexpr * (#(var-ref gamma) #(fun-call fabs (#(var-ref beta))))))))
(130 assign (xi #(var-ref tmp40)))
(131 assign (gamma #(aexpr / (#(lit 1.0)
                              #(fun-call sqrt (#(aexpr + (#(lit 1.0)
                                                          #(aexpr * (#(var-ref theta) #(var-ref theta))))))))))
(132 assign (eta #(aexpr / (#(aexpr * (#(aexpr * (#(aexpr neg (#(var-ref eta))) #(var-ref rho_o)))
                                       #(aexpr * (#(var-ref gamma) #(var-ref gamma)))))
                            #(aexpr * (#(var-ref beta)
                                       #(aexpr * (#(var-ref gamma_o) #(var-ref gamma_o)))))))))
(133 assign (scalds #(aexpr * (#(var-ref theta_o) #(var-ref gamma)))))
(134 assign (scalds #(aexpr * (#(var-ref scalds) #(var-ref scalds)))))
(135 * scal (#(var-ref scalds) d))
(136 | axpy (#(var-ref eta) p d))
(137 | scal (#(var-ref scalds) s))
(138 | axpy (#(var-ref eta) ptld s))
(139 | add-assign (d x))
(140 | sub-assign (s r))
(141 | dot (tmp43 r r))
(142 assign (tmp42 #(fun-call sqrt (#(var-ref tmp43)))))
(143 assign (err #(aexpr / (#(var-ref tmp42) #(var-ref bnrm2)))))
(144 label rest-loop-end)
```

B.2.4 Flowchart — improved partitioning

```
(83 label rest-loop-begin)
(84 assign (rhorecip #(aexpr / (#(lit 1.0) #(var-ref rho)))))
```

```
(85 assign (xirecip #(aexpr / (#(lit 1.0) #(var-ref xi)))))
(86 assign (tmp34 #(lit 0.0)))
(87 assign (tmp35 #(lit 0.0)))
(88 assign (tmp38 #(lit 0.0)))
(89 assign (tmp41 #(lit 0.0)))
(90 assign (theta_o #(var-ref theta)))
(91 assign (tmp43 #(lit 0.0)))
(92 * scal (#(var-ref rhorecip) y))
(93 | scal (#(var-ref xirecip) z))
(94 | dot (tmp34 z y))
(95 | copy (y ytld))
(96 | copy (z ztld))
(97 | zero (ptld))
(98 assign (delta #(var-ref tmp34)))
(99 assign (scalp #(aexpr neg (#(aexpr * (#(var-ref xi) #(aexpr / (#(var-ref delta) #(var-ref epsilon)))))))))
(100 assign (scalq #(aexpr neg (#(aexpr * (#(var-ref rho) #(aexpr / (#(var-ref delta) #(var-ref epsilon)))))))))
(101 assign (rho_o #(var-ref rho)))
(102 * axpy (#(var-ref scalp) p ytld))
(103 | copy (ytld p))
(104 mvmul-t (#(lit 1.0) a p ptld))
(105 * axpy (#(var-ref scalq) q ztld))
(106 | copy (ztld q))
(107 | dot (tmp35 ptld q))
(108 assign (epsilon #(var-ref tmp35)))
(109 assign (beta #(aexpr / (#(var-ref epsilon) #(var-ref delta)))))
(110 assign (tmp36 #(aexpr neg (#(var-ref beta)))))
(111 assign (tmp39 #(aexpr neg (#(var-ref beta)))))
(112 * copy (vtld v))
(113 | scal (#(var-ref rhorecip) v))
(114 | copy (wtld w))
(115 | scal (#(var-ref xirecip) w))
(116 | copy (ptld vtld))
(117 | axpy (#(var-ref tmp36) v vtld))
(118 | copy (vtld y))
(119 | dot (tmp38 y y))
(120 | copy (w wtld))
(121 | scal (#(var-ref tmp39) wtld))
(122 assign (tmp37 #(fun-call sqrt (#(var-ref tmp38)))))
(123 mvmul-n (#(lit 1.0) a q wtld))
(124 assign (rho #(var-ref tmp37)))
(125 * copy (wtld z))
(126 | dot (tmp41 z z))
(127 assign (theta #(aexpr / (#(var-ref rho) #(aexpr * (#(var-ref gamma) #(fun-call fabs (#(var-ref beta)))))))
(128 assign (gamma_o #(var-ref gamma)))
(129 assign (tmp40 #(fun-call sqrt (#(var-ref tmp41)))))
(130 assign (gamma #(aexpr / (#(lit 1.0)
                              #(fun-call sqrt (#(aexpr + (#(lit 1.0)
                                                          #(aexpr * (#(var-ref theta) #(var-ref theta)))))))))))
(131 assign (xi #(var-ref tmp40)))
(132 assign (eta #(aexpr / (#(aexpr * (#(aexpr * (#(aexpr neg (#(var-ref eta))) #(var-ref rho_o)))
                                       #(aexpr * (#(var-ref gamma) #(var-ref gamma)))))
                            #(aexpr * (#(var-ref beta)
                                       #(aexpr * (#(var-ref gamma_o) #(var-ref gamma_o)))))))))
(133 assign (scalds #(aexpr * (#(var-ref theta_o) #(var-ref gamma)))))
(134 assign (scalds #(aexpr * (#(var-ref scalds) #(var-ref scalds)))))
(135 * scal (#(var-ref scalds) d))
(136 | axpy (#(var-ref eta) p d))
(137 | scal (#(var-ref scalds) s))
(138 | axpy (#(var-ref eta) ptld s))
(139 | add-assign (d x))
(140 | sub-assign (s r))
(141 | dot (tmp43 r r))
```

```
(142 assign (tmp42 #(fun-call sqrt (#(var-ref tmp43)))))
(143 assign (err #(aexpr / (#(var-ref tmp42) #(var-ref bnrm2)))))
(144 label rest-loop-end)
```