Local and Global Optimisation

Michael P. Fourman

October 29, 2006

Assessment

The intermediate deadline, 6.00p.m., 13th **May**, will only apply to stages 1 and 2. Solutions to these parts will only obtain full credit if completed before the deadline.
The **final deadline** for this project is **6.00pm, Friday, 27th May.**
The following marking schedule will apply to the project.

1. 20 marks (15 if late)
2. 20 marks (15 if late)
3. 20 marks
4. 15 marks
5. 15 marks
6. 10 marks

These marks will be divided by 5 to produce a contribution of 20% towards your final mark for the course.
I do not expect everyone to finish all stages (but I do expect that some will). Credit will be given for each working functor (mathcing the specification) that you produce; incomplete code fragments will not score well, (if at all).

Demonstrations

Marking of your project will include a live demonstration of your system. You will be expected to attend a short demonstration, of the code collected for your project, during your scheduled lab session in the final week of term. Updated lab lists will be posted next week. **You should check to see that**

your name appears correctly, and contact Mark Messenger by 20th May, to resolve any problems. Test files, similar to those that will be used for the demonstrations, will be made available by 20th May.

Information

Code for the signatures referred to in the project, and implementations of some useful functors, may be found in the directory
`/home/mulgara/year2/examples/cs201/ml/Proj`
Additions to this and to the directory of test files,
`/home/mulgara/year2/examples/cs201/doc/Project`
may be made from time-to-time, during the course of the project.
Do not copy this code, or the test files; create "soft links" instead. If you want to refer to a code files file `foo.ML` from your project directory, make a soft link by typing
`ln -s /home/mulgara/year2/examples/cs201/ml/Proj/foo.ML .`
(when working in your `Project` directory).

The project: stages 4-6

Do not start work on these stages until you have completed and tested stages 1-3. Further background material is provided in an Appendix to this document, together with some practical hints on implementation.

Stage Four: Optimising simple innermost loops

Produce a functor with the header

```
functor LOOPOPT(F:FlowSig):
sig
   structure OF: FlowSig
   val loopOpt : F.G.Graph -> OF.G.Graph
end
```

The function `loopOpt` should optimise simple innermost loops by moving invariant code outside the loop.
You should: identify simple innermost loops; for each block that is the body of such a loop determine which expressions may safely be computed before entering the loop; split the block accordingly, and make the appropriate alterations to the flowgraph.

Stage Five: Global data-dependency analysis

Produce a functor with the header

```
functor DATAOPT(F : FlowSig):
sig
   structure OF : FlowSig
   val dataOpt  : F.G.Graph -> OF.G.Graph
end
```

The function `dataOpt` should optimise the code by performing a global data-dependency analysis on the flowgraph, and eliminating redundant code from each block.
You should not attempt to analyse data dependencies for general loops. You may choose to analyse data dependencies for simple innermost loops, but you will get partial credit if you take a conservative (that is, pessimistic) approach to all loops.

Stage Six: Optimised code for a basic block

Produce two functors (partial credit will be given for implementation of either one of these).
The first functor should have the header

```
functor DAGOPT(B : BlockSig):
sig
   structure B : BlockSig
   val blockOpt : B.Block -> B.Block
end
```

The function blockOpt should use algebraic transformations to optimise a basic block.
The second functor should have the header

```
functor OPTBLOCKTOCODE( structure C:CodeSig
                        structure B:BlockSig ) : BlockToCodeSig
```

The function `blockToCode` should attempt to minimise the number of temporary variables used in the code it generates.

Appendix

Optimisations

A flow graph representation can be used as the basis for a number of optimisations. Global optimisation involves the analysis of the entire graph; local optimisations can be applied to individual basic blocks, or to small subgraphs of the flowgraph. Optimisations usually involve eliminating unnecessary or duplicated computation, or changing the order of computation.

Loops in flow graphs

The body of a loop is typically executed many times. The benefits of any optimisations we can make to the body of a loop are multiplied accordingly. Optimisations are particularly important in *inner loops,* that is, loops that do not contain other loops, since programs tend to spend most of their time inside such loops.
In general, identifying loops in a flowgraph is a delicate matter; indeed, some flowgraphs cannot be analysed neatly as a collection of loops. However, in code compiled from a structured language, one that uses constructs such as conditional statements and while loops, rather than the undisciplined "goto", loops have a nested hierarchical structure that can be derived from the flowgraph. Even in this case a general treatment is tricky; general algorithms for identifying loops in graphs are beyond the scope of this course. However, it is easy to identify simple inner loops. A *simple inner loop* is a vertex of the flowgraph with an arc leading back to itself (at the structured source code level, these correspond to loops containing no other loops or conditionals).

Transformations on loops

We briefly describe two transformation that can improve the efficiency of loops. First, *code movement*, moving code outside a loop. Computation of an expression that has the same value regardless of how many times the loop is executed, may be removed from the loop, and placed before the re-entry point. This optimisation can be done at the flow-graph level, by splitting a node of the flow graph in two, and adjusting the arcs appropriately.
Stage 5 involves applying this transformation to simple inner loops. Here are some practical hints on implementation details.
Consider the case of splitting block n. If you leave only the invariant code in block n, and place the rest of the code in a new block, then jumps from elsewhere in the flowgraph will still be correct. (You'll still have to ensure

that exits from the new block lead to the right places). If you use a negative number, $-(n+1)$, to identify the new block vertex, then there will be no danger of confusion with existing vertices, or with blocks generated by splitting other loops.
Should any value you choose to compute outside the loop be temporary, you will need to invent an appropriate variable name, to pass it into the new loop body. Any string that is neither a temporary variable, nor one of the normal "normal variables" will do—for example, a string starting with the character "`!`".
Second, *induction-variable elimination*, which spots where two or more variables are being incremented or decremented in step during the loop, and eliminates all but one by replacing occurrences of the others by suitable linear functions of the survivor. This transformation may not reduce the amount of computation within the loop; its main advantage is in decreasing space requirements. It can reduce the number of variables required to hold values during the execution of the loop. (You are not asked to implement induction variable elimination as part of the project.)
As a trivial example of both transformations being applied in unison, the following two programs are equivalent:

```
i = 0                          i = 0
j = 20                         k = x + y
k = x + y                      l = 20 - i
l = k + j                      l = k + l
i = i + 1                      i = i + 1
j = j - 1                      if i < 10 goto 2
if i < 10 goto 2
```

Dataflow analysis

Each basic block *produces* new values on some variables, its *outputs,* and *uses* the previous values of some variables, its *inputs.* If a block, B, produces a value for a variable x, and does not use the previous value of x at all, then the value of x on entry to the block is irrelevant to B; it is not needed as an input. The value of any variable at the end of a block *depends* on the values of some set of variables (a variable changed by the block depends on those variables occurring as leaves of the corresponding expression tree), at the beginning of the block. We can perform a global optimisation by keeping track of which variables are needed as inputs to each block, and using this information to eliminate redundant code.
Which variables are needed as inputs of a block B depends on which variables

are needed as inputs by the blocks adjacent to B in the flow graph, these are needed as outputs of B. Once we know which variables are needed as outputs of B, we can use the DAG representation of the block to determine which variables are needed as inputs to produce these outputs. Note that variables occuring in the condition of any goto at the end of the block, B, are needed as outputs of that block. We write $\mathit{testvars}(B)$ for this set of variables.

To summarize, for each block, B: the final value of each variable, v, *depends* on a set of variables, $\mathit{depends}(B, v)$; the variables in some set, $\mathit{outputs}(B)$, are needed as outputs of the block; the variables in a set, $\mathit{inputs}(B)$, are needed as inputs to the block. These sets are related by the following equations:

$$\begin{aligned} \mathit{outputs}(B) &= \mathit{testvars}(B) \cup \bigcup_{C \in \mathit{adj}(B)} \mathit{inputs}(C) \\ \mathit{inputs}(B) &= \bigcup_{v \in \mathit{outputs}(B)} \mathit{depends}(B, v) \end{aligned}$$

A data dependency analysis is used to determine which variables are neded as outputs for each block. A precise answer is not necessary, but if we err, we should err on the side of caution; producing too large a set is safe, producing too small a set is an error. The safest course is to assume that all variables are needed as outputs of every block. Dataflow analysis can give a better estimate.

For any terminal block, all normal variables are needed as outputs. We can determine which variables are needed as outputs of other blocks using a depth-first search; code for topologically sorting a graph can be adapted to this purpose, to compute the variables needed by each block (both input and output) in the same order as we would add the nodes to a topologically sorted list. Loops need special treatment: in general we must be pessimistic, when we detect a cycle in the flowgraph we don't raise an exception, instead, we presume that all normal variables are needed as inputs at the entry point to the cycle; for simple inner loops it is possible to iterate the dependency calculation for the block to determine a precise dependency relation for the loop.

Once we have performed a global data dependency analysis, we may optimise each block by eliminating code whose only function is to produce a value for a variable not needed as an output of that block.

Local optimisation of basic blocks

Basic blocks may be optimised using *algebraic transformations,* similar to those introduced in Practical 4, to rearrange each expression in a block into

a more efficiently-computable form. The generation of intermediate code for a basic block may also be optimised to reduce the number of temporary variables used.
To reduce the number of temporary variables used it is helpful to generate the intermediate code backwards, generating code that produces the value of an expression *before* generating the code that will produce the values of its subexpressions. However, you have to plan ahead: before generating the code for an expression, we must decide where to store the values of its subexpressions. A dictionary, with expressions as keys, and variables as items, may be used to keep track of these decisions. The point of generating the code backwards is that we can also keep track of which variables are in use (in fact, its easier to keep track of which variables are available).
Suppose that `available` represents a queue of available variables, and `whereis` represents the dictionary described above. To generate code to place the value x `*` y in the variable `z`, for example, we must

- add `z` to the collection of available variables (this is the crux of the optimisation)
- decide where the values of x and y will be generated: for each subexpression, x, lookup to if it is already in the `whereis` dictionary,
  - if so, return the variable (call it, $w(x)$, say),
  - if not remove an available variable, $w(x)$, from the queue, and record the pair $(x, w(x))$ in the dictionary
- generate the code `z :=`$w(x)$`***`$w(y)$

As a trivial example of several transformations being applied in unison, the following two programs are equivalent:

```
a = b + c;
b = a - d;
c = a - d;
d = a + b;
d = c * 1;
b = d ** 2;
```

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