Compiler-related Algorithms

Michael P. Fourman

October 29, 2006

Introduction

This project takes up the theme of Practical 4; it is based around the idea of program optimisation. Background material on optimisation is given as an appendix. Before proceeding any further, you should make sure that you fully understand this material; all of it is directly relevant to the work you have to do.
The SML signatures and functor headers given in the text should be regarded as a partial specification of software that has to be implemented. A direct high-level SML implementation can be obtained by writing appropriate functors. Your SML code should be written in a modular style, making use of lists and ADTs such as sets, dictionaries, and graphs.

What you should do

First read the Appendix, which provides essential background material. The project consists of several, related "stages". We list these briefly. Stages 1-3 are considered in more detail below. The remaining stages will be covered in a subsequent note, which will be issued by 6th April.

1. Construct a DAG representation of a *basic block.*
2. Implement an algorithm that reconstructs an optimised basic block from a DAG.
3. Implement an algorithm that constructs a *flow graph* representation from the abstract syntax of a program.
4. Identify and optimise innermost loops.
5. Perform a global data dependency analysis to eliminate redundant code.

6. Improve the optimisation of a basic block.

Assessment

Credit will be given for all work attempted. As a rough guide, you can get a passing grade for a reasonable implementation of four stages. Incomplete implementations of a stage will not score highly. If time is tight, you should therefore complete as many stages as you can, rather than making incomplete attempts at all stages. Stages 1-3 depend on each other, incrementally. Stages 4-6 depend on stages 1-3, but are otherwise independent; you can tackle them in any order. This project counts for 20% of your final mark for the course.

Deadlines

13.5.94 Intermediate Deadline Your implementations of stages 1-3 will gain extra credit if they are complete by this time.

27.5.94 Final Deadline All code and reports will be frozen at 6.00pm on this day. No further changes will then be allowed.

Housekeeping

You should use a directory named `Project` for your project. Code for functors and signatures should be placed in correspondingly named files. Code, like that described in the section on testing Stage 1, that applies functors to produce structures, should be placed in a single file named `ml_bind.ML`. When you have made changes to your code, you can recompile those parts of the system that may be affected by these changes by typing `PolyML.make "."` to the ML prompt. You can selectively recompile those parts of the system needed by a particular code file by typing `PolyML.make "`*`filename`*`"`.

The intermediate code

This project deals with the optimisation of programs in a simple "intermediate code", that might be produced by a compiler, from a high-level language such as C or Pascal, as a prelude to the generation of machine code for a particular computer. The intermediate code is introduced in the Appendix.

In this section, we describe a type provided to represent intermediate code programs.
The SML signature `CodeSig` describes an abstract syntax for the intermediate code ("abstract syntax" means the syntax of the language, ignoring superficial distinctions of form). The functor `CODE` provides an implementation. Make sure that you understand how the datatype `Code` can be used to represent a program. A function `readCode : string -> Code` is provided to read a file in the format shown in the examples, and produce a value of type `Code`. A function `writeCode : (Code * string) -> unit` is also provided; it writes a textual representation of a value of type`Code` to a file (beware—if you ask it to, it will overwrite the contents of an existing file).

```
infix 6 +++ ---
infix 7 *** ///
infix 4 === <<= >>= <<< >>> !!= ;
signature CodeSig =
sig
datatype Atom = Id of string | Lit of int

datatype Expn = V of Atom
              | +++ of Atom * Atom
              | *** of Atom * Atom
              | /// of Atom * Atom
              | --- of Atom * Atom

datatype Cond = TRUE | FALSE
              | === of Atom * Atom
              | <<= of Atom * Atom
              | >>= of Atom * Atom
              | <<< of Atom * Atom
              | >>> of Atom * Atom
              | !!= of Atom * Atom

datatype Statement =
       := of string * Expn
    | GOTO of Cond * int

datatype Code = Code of Statement list
end
```

Observe that the type `Expn` is *not* recursive; only simple expressions, involving a single operator, are allowed.

Stage One: Constructing a DAG representation of a basic block

Do this in two steps. First build a number of expression trees to represent the overall effect of the assignments in a basic block. The Appendix outlines how this can be done.
You should write a functor

```
functor CODETOBLOCK( structure C:CodeSig
                     structure B:BlockSig ) : CodeToBlockSig
```

The signature `CodeToBlockSig` describes a function that transforms a code segment into a block representation.

```
signature CodeToBlockSig =
sig
    structure C : CodeSig
    structure B : BlockSig
    val codeToBlock :  C.Code -> B.Block
end
```

Your function `codeToBlock` should check that the code consists of a sequence of assignments, and produce the corresponding block.
The signature `BlockSig` describes the type `Block`, used to represent the overall effect of the assignment statements of the block.

```
signature BlockSig =
sig
datatype Expn = Id of string | Lit of int
              | +++ of Expn * Expn
              | *** of Expn * Expn
              | /// of Expn * Expn
              | --- of Expn * Expn

datatype Block = Block of (string * Expn) list
end
```

Notice that the type `Expn` is recursive, unlike the type `Expn` specified in `CodeSig`. The functor `BLOCK` provides an implementation of `BlockSig`
For the second step, recall (from the appendix) that a block can be viewed as a DAG; later optimisations will be based on a topological sort of this DAG. You will use the functor `TOPSORT` to provide this. Since `TOPSORT` takes an argument of signature `GraphSig`, you must write code to present the a block as a graph. To do this, you should implement a functor with header

```
functor BLOCKASDAG( structure B: BlockSig ):
sig structure G : GraphSig
    structure B : BlockSig
    sharing type G.Graph  = B.Block
        and type G.Vertex = B.Expn
end
```

The result signature of this functor provides an alternative view of a block as a DAG; a block can be viewed as a graph whose vertices are expressions. In your implementation, you should cheat slightly by making the function `G.vertices` return a list of those expressions that are the final values of normal variables (rather than all the expressions in the block). A topological sort of the graph starting from this list of vertices will produce a list (in reverse order) of exactly the expressions you need to compute.

Testing

To test the implementation so far, produce a list of the expressions that must be computed for the sample basic block given in
`/Users/mfourman/Documents/Informatics/homepages/mfourman/web/teaching/mlCourse/doc/P`
Place the following code in a file `ml_bind.ML`

```
structure Block       = BLOCK()
structure Code        = CODE()
structure CodeToBlock = CODETOBLOCK ( structure C = Code
                                      structure B = Block )
structure DagBlock    = BLOCKASDAG( structure B = Block )
structure TopSort     = TOPSORT( structure G = DagBlock.G )
```

Type `PolyML.make "."` to the ML prompt; this should compile all your code. You don't need to provide your own implementation of topological sort; the functors `BLOCK`, `CODE`, `TOPSORT` are provided in the `proj` database.
Once the system has compiled sucessfully, you can test it by typing the following to the ML prompt (or by reading it from a file):

```
val testCode = Prac6.readCode
    "/home/mulgara/year2/examples/cs201/doc/Project/test4";
val testBlock= CodeToBlock.codeToBlock testCode;
val toCompute = rev ( TopSort.topSort testBlock ) ;
```

Stage Two: Optimising a basic block

In this stage, you have to implement an algorithm that generates a sequence of statements from a block (the intention is that your sequence is no longer than the original sequence used to generate the block, and that its instructions are no more complicated).
There are two kinds of optimisation that should be done. First, assume that any temporary variables (those which begin with _) have been introduced by the compiler to be used within the particular basic block, and that their values are not required on exit from the block. Thus, you should only generate code to compute the values of normal variables. Second, evaluation of repeated sub-expressions can be eliminated. The value of each common subexpression need only be computed once. These optimisations result automatically from correct use of `topSort` to produce the list of expressions that must be computed.
You have to provide a structure `BlockToCode` matching the signature `BlockToCodeSig`

```
signature BlockToCodeSig =
sig
    structure C : CodeSig
    structure B : BlockSig
    val blockToCode : B.Block -> C.Code
end
```

To do this you should implement and apply a functor whose header looks something like this:

```
functor BLOCKTOCODE(
     structure C : CodeSig
     structure B : BlockSig
     structure S :
        sig type Graph
            type Vertex
            val topSort : Graph -> Vertex list
        end
     ...
     sharing type S.Graph  = B.Block
         and type S.Vertex = B.Expn
     ... ): BlockToCodeSig
```

The ellipses (...) in this header indicate that you should incorporate other arguments and sharing constraints to include any parameters (such as a structure providing a dictionary) needed for your implementation. As usual, the code for this functor should be placed in an appropriately named file, and

the `ml_bind.ML` file should be extended to apply the functor to produce the structure `BlockToCode`.
The idea is to compute the value of the expressions in the sorted list by sucessively producing an instruction for each new expression. You will have to introduce your own temporary variables to store intermediate results, and keep track of the location of each value as it is computed, using a dictionary. For the moment, don't worry about minimising your use of temporary variables. You will do that in stage 6. Just introduce new variables $\_n$ as you need them. The function `stringOfInt : int -> string` is provided, in the structure `Prelude`,to help you construct these variables.

Testing

Test your implementation by applying your optimisation to the block produced from the `test4` example.

Stage Three: Finding the basic blocks

The signature `FlowSig` describes the user-interface to a flowgraph representation of a program.

```
signature FlowSig =
sig
    structure G : GraphSig
    structure B : BlockSig
    structure C : CodeSig
    sharing type G.Vertex = int

    val data : G.Graph -> G.Vertex ->
                    B.Block * C.Cond
end
```

You have to implement a structure
`FlowGraph : sig include FlowSig val codeToFlow : C.Code -> G.Graph end`
The result signature includes the signature `FlowSig`. This means that your implementation must provide all that is specified by `FlowSig`, together with the function `codeToFlow`.
You should implement this by writing and applying a functor

```
functor FLOWGRAPH (structure C2B : CodeToBlockSig
                   ... ):
    sig
        include FlowSig
        val codeToFlow : C.Code -> G.Graph
    end
```

Your implementation should observe a number of conventions.

- The initial vertex of the flowgraph should correspond to the integer 0. (You can use the statement number of the header of each block as the corresponding index.)

- A basic block is represented as a values of type `Block` together with a condition of type `Cond`.

  - Where the basic block ends with a conditional `goto`, the condition should be taken from this statement, and the first block in the adjacency list should correspond to the condition being true.

  - When there is no `goto` at the end of a basic block the condition should be `TRUE`.

(Again using ellipses to indicate that you may wish to add other parameters.) You have to implement a type to represent a flowgraph, and functions to satisfy the signature. A simple representation for a flowgraph is given by the declarations

```
type Vertex = int
datatype Graph = FG of (Vertex                          (* index *)
                        * (Vertex list)                 (* adjacent vertices *)
                        * (C2B.B.Block * C2B.C.Cond) (* block data *)
                       ) list
```

There are many other possibilities.

Testing

A functor with the following header is provided. You should apply it to the structures you have produced in stages 2 and 3, to produce optimised versions of the code in the examples `test1`...`test4`.

```
functor FLOWTOCODE (structure F   : FlowSig
                    structure B2C : BlockToCodeSig
                    sharing B2C.B = F.B and B2C.C = F.C):
    sig
        include FlowSig
        val flowToCode :  G.Graph -> C.Code
    end
```

Appendix: Optimisation of intermediate code

In this note, we introduce a subject area in which there is scope for devising a family of algorithms that all contribute to improving the solution to one problem. The area is one that is central to computer science, albeit at a low level: given a program written in some simple low-level language, transform the statements/expressions of the program so that it will run more quickly. The program may have been written by a human or, more likely, it may have been produced by a compiler that first converts a high-level language program into an equivalent low-level, but machine-independent, language program[1].

Intermediate Code

Intermediate code is a simple, low-level programming language. Its computational statements can perform at most one operation; complex expressions are not allowed, and it has unconditional or conditional branching statements rather than selection or repetition constructs. Our language includes variables, constants, simple operators, assignment, and branching instructions.
An *atomic* expression is either an identifier, which is a string, or an integer constant. Other *expressions* can be formed from two atomic expressions by applying one of the arithmetic operators, `+ * - /`. Complex expressions are not allowed.
A *condition* can be formed by comparing two atomic expressions using one of the operators, `<`, `<=`, `>`, `>=`, `==` or `!=`.
Code for a program consists of a sequence of statements, implicitly numbered $0, 1, 2, 3, \ldots$, in order from beginning to end. There are two different kinds of statement:

assignment: `z :=` *expn*
: assigns the value of *expn* to `z`

conditional goto: `if (`*condition*`) goto n`
: causes a branch to statement `n` if the condition is true.

Two different kinds of variable occur: normal variables are sequences of letters and digits beginning with a letter; *temporary* variables, have names beginning with "_", followed by a sequence of numbers or letters. The significance of temporary variables will be explained later. The files
`/Users/mfourman/Documents/Informatics/homepages/mfourman/web/teaching/mlCours`

[1]The idea is that this program is optimised, and then converted into an equivalent machine-language program for a particular machine.

/Users/mfourman/Documents/Informatics/homepages/mfourman/web/teaching/mlCourse/doc/p
contain examples of programs written in this language.
Before (briefly) considering a few methods for positively transforming programs, we describe an appropriate graph-based data structure for representing programs.

Basic blocks

A *basic block* is a sequence of program statements in which the flow of control enters at the beginning of the sequence and leaves at the end, without halting or branching to any other part of the program. The *leader* of a basic block is its first statement.
A basic block can be represented as a list of assignment statements, in some cases followed by a branch statement. For some purposes this is sufficient. However, it is often convenient to work with a more structured representation of a basic block. We are interested in the overall effect of executing the basic block. We represent the effect of the sequence of assignments as a collection of expressions representing the final values of the variables on leaving the block, in terms of their initial values on entry to the block.
To construct this representation, we step through the statements in the block, one at a time, keeping track of the "current value" of each variable. The algorithm will produce a dictionary that associates each indentifier with the expression that represents the final value of that identifier. We construct these incrementally, by considering the effect of each statement in turn. Initially, the dictionary is empty.
Consider the case of an assignment `z := x` *`op`* `y`, where *op* is one of the operations of the language, and `x` and `y` are variables. If `x` and `y` have entries in the dictionary, giving values $d(\texttt{x})$ and $d(\texttt{y})$, then we should record in the dictionary, the fact that the value of `z` is now $d(\texttt{x})\,op\,d(\texttt{y})$. If either `x` or `y` has no entry in the dictionary, it means that it still has its initial value, which we represent by an appropriate leaf node. Other assignment statements are treated similarly. This construction process is repeated for each statement of the basic block, in turn.
This construction produces a forest of expression trees. The nodes of the tree are the expressions and their sub-expressions. The internal nodes represent the operations to be performed; leaves represent variables or constants used in computing the final values. We can view the forest of expression trees as a directed acyclic graph, or *DAG.* The nodes of the DAG are the trees and their subtrees; a directed edge represents the fact that one operation uses the result of another operation or a variable or constant. When we take this

point of view, we consider two structurally equal trees to be equal, even if they occur as subtrees in different parts of the forest.
Thus, the representation of a program consists of a directed graph (the flow-graph) with vertices that are themselves directed acyclic graphs (the DAGS representing basic blocks).

Producing code from a DAG

To produce code for a basic block from its DAG representation, we introduce temporary variables to store intermediate values, and produce code to compute a value for each node of the DAG, in turn. Before we can compute the value of a node, we have to compute the value of each adjacent node. A reverse topological sort gives an appropriate order for producing the code. A dictionary can be used to keep track of the variable used to store each temporary value.

Flowgraphs

A program can be represented by a *flow graph*: a directed graph whose vertices correspond to basic blocks. A directed edge (B_1, B_2) represents the fact that the basic block B_2 may immediately follow the basic block B_1 in some execution sequence of the program. Therefore, there is an edge joining (B_1, B_2) if the last statement of B_1 is a conditional or unconditional branch to the first statement of B_2, or if B_2 immediately follows B_1 in the program and B_1 does not end with an unconditional branch statement. The *initial vertex* of a flow graph corresponds to the basic block whose leader is the first statement executed in the program.

Finding the flow graph for a program

It is straightforward to construct a flow graph for a program. The graph has a vertex for each basic block leader. The leaders are defined as:

- the first program statement;
- all statements that are destinations of unconditional or conditional branches; and
- all statements that immediately follow unconditional or conditional branch statements.

Each basic block consists of its leader, plus all statements after it until (but not including) the next leader or the end of the program. The edges of the flow graph are as described earlier.