CM30171

Rob Willison

November 28, 2016

Contents

1 Introduction

This report is an explanation of the design decisions undertaked while writing a –C compiler and interpreter.

2 Code Interpretation

3 Intermediate Code Generation

3.1 TAC Design

The three address code used in the compiler was designed to be abstract enough as too keep any machine dependent decisions out of the TAC stage. Many of the instructions are obvious such as store and mathematic operations, examples below, and they won't be explained in great detail.

```
r2 := 1
r3 := r1 / r2
```

One thing to note about the store instruction is that in the event that the store operand is defined in the scope above the current scope like x is in the following example.

```
int main()
{
  int x = 4;
  int test()
  {
    return x;
  }

  return test();
}
```

In that scenario the store command in the test function will have an extra piece of information saying that its defined in the scope one level above, so the store command will look like the following.

```
DEFINED IN 1 r2 := x
```

This information is also included when a closure is called from a narrower scope, the TAC would be as follows.

```
CALL _1 FROM SCOPE 1
```

For Functions a label instruction denotes the start and a end label for the end. After the start label the new activation frame instruction which tells the compiler to allocate space for a new activation frame with a given number of arguments, locals and tempories. Before the end there is a return instruction which contains the register with the value to return. Below is an example function in TAC.

```
_1:
NEW FRAME 0 arg 0 loc 1 temp
DEFINED IN 1 r2 := x
RETURN r2
```

FUNCTION END

There is one type of control sequence in the language, the if else, the way this is described in TAC is using a sequence of if instructions and labels denoting the various bodies if the if and else parts. An example in –C and the corresponding TAC are below.

```
if (1 > 4) {
  return 4;
} else if (2 > 1) {
  return 3;
}
```

```
r1 := 1
r2 := 4
r3 := 1 > 4
IF NOT r3 GOTO 1
r4 := 4
RETURN 4
GOTO 2
LABEL 1: r5 := 2
r6 := 1
r7 := 2 > 1
IF NOT r7 GOTO 3
r8 := 3
RETURN 3
LABEL 3: LABEL 2:
```

Finally there is one loop type in the language, the while loop, this is translated into a goto, an if and a label at the top and bottom of the loop body. the while condition is placed after the body. n example in –C and the corresponding TAC are below.

```
int x = 0;
while (x < 5)
{
  x = x + 1;
}
```

```
r1 := 0
x := r1
GOTO 1
LABEL 2: r2 := x
r3 := 1
r4 := r2 + r3
x := r4
LABEL 1: r5 := x
r6 := 5
```

```
r7 := r5 < r6
IF r7 GOTO 2
```

3.2 TAC Generation

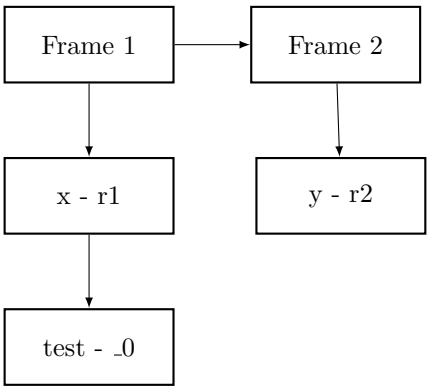
All the TAC complilation is done in the tac_compliler.c file. In order to generate the TAC from source code a tree walk is performed over the parse tree at each node depending on the type a set of TAC instructions are created and added to the current TAC block. If a new function is found a new TAC block is created before that part of the tree is parsed, also if a goto or label TAC instruction are created new blocks are created. When a leaf is reached a store instruction is created for the value at the leaf.

3.2.1 Enviroment Structure

In order to keep track of the location of any local variables in the code and enviroment is created to store the association between token and register location. The enviroment is made up of frames each frame corresponds to a scope in the program and has a linked list of all locals in that scope. The frames also contain a linked list of the functions defined, these are a pair of token and label assigned to the function. For example the environment created while walking the following piece of code is shown below.

```
int main()
{
  int x = 4;
  int test()
  {
    int y = 5;
    return x * y;
  }

  return test();
}
```

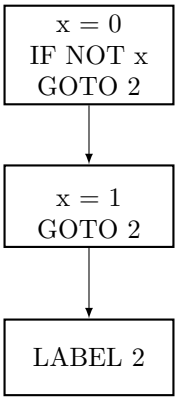


As you can see the environment holds the association between the tokens of the variables and functions with the TAC register location and the function label respectively.

3.2.2 TAC Blocking

The diffrent blocks of TAC code, between labels and jumps, are separated to allow easier optimisation and MIPS translation. As the AST is walked if a instruction which requires a jump (If, While, Function Call) is found a new block is created and set as the current block after the jump. Similarly if a label is created a new block is created and the label is assigned to that new block.

in order to compile the code in the same order as the TAC each block has a link to the block which was created from the code directly after it. A Simple example is given below.



This was done so that the MIPS compilation step can easily follow the sequence of TAC instructions. //TODO write about linking blocks if and when done

3.2.3 Inner functions

In order to implement inner functions, functions defined inside another, in TAC the code for the function must be flattened so that there are no functions in functions in the TAC. To do this when a function definition is reached inside another the current TAC block is stored and another is created for the inner function, once the inner function's AST has finished being walked the previous block is restored as the current block and the AST tree walk continues. This gives the result of placing any inner functions after the function it was defined in.

3.3 TAC Optimisation

After the TAC generation phase has been completed the sequence of blocks is handed over to the optimisation phase. This is in the tac_optimiser.c file. This applies constant folding, copy propagation, dead code elimination, common sub expression elimination and algebraic transformations algorithms to the TAC block repeatedly till now change can be made. Each TAC block is processed from bottom to top for each optimisation technique.

3.3.1 Copy Propagation

Copy propagation looks for a store instruction, once on is found it checks the rest of the code below the store until it finds a write to the store destination. For every read of the destination before replaced the use is replaced with the stores operand. To demonstrate this copy propergation was applied to the same example as above.

```
int x = 4;
int y = 6;

return x + y;
```

```
r1 := 4
x := r1
r2 := 6
y := r2
r3 := x
r4 := y
r5 := r3 + r4
RETURN r5
```

```
r1 := 4
x := 4
r2 := 6
y := 6
r3 := 4
r4 := 6
r5 := 4 + 6
RETURN r5
```

3.3.2 Constant Folding

The constant folding function looks for arithmetic operations and replaces them with the result where possible. This is done by looking for a operation where both operands are tokens with type constant. Below is an example using both constant folding and copy propergation.

```
int x = 4;
int y = 6;

return x + y;
```

```
r1 := 4
x := 4
r2 := 6
y := 6
r3 := 4
r4 := 6
r5 := 4 + 6
RETURN r5
```

```
r1 := 4
x := 4
r2 := 6
y := 6
r3 := 4
r4 := 6
r5 := 10
RETURN 10
```

3.3.3 Dead Code Elimination

Dead code elimination works by analysing each instruction using and useing the next use info removes them if not needed. Firstly this is only done for registers user defined variables are live at the end of the block so an't removed. A instruction is removed if the next use is not live, its another write, or there isn't another next use. Using the same example as above and applying dead code elimination gives the following.

```
int x = 4;
int y = 6;

return x + y;
```

```
r1 := 4
x := 4
r2 := 6
y := 6
r3 := 4
r4 := 6
r5 := 10
RETURN 10
```

```
RETURN 10
```

3.3.4 Common Sub Expression Elimination

If two expression have the same operands and operation the later one can be replaced with a store of the destination of the first. These are found by looking for pairs of arithmatic operations, when one is found if the operation and operands are the same the second is replaced with a store. An exaple is given below usign this method in conjunction with copy propergation.

```
int c = 4 * 6;
int d = 4 * 6;

return c;
```

```
r1 := 4
r2 := 6
r3 := r1 * r2
c := r3
r4 := 4
r5 := 6
r6 := r4 * r5
d := r6
r7 := c
RETURN r7
```

```
r1 := 4
r2 := 6
r3 := 4 * 6
c := r3
r4 := 4
r5 := 6
r6 := r3
d := r3
r7 := r3
RETURN r3
```

3.3.5 Algebraic Transformations

The final optimisation technique is to transform some algebraic expressions which will always give 1 or 0 to those values. In order to do this the optimiser looks for those specific expressions and simply replaces them with a store of either 1 or 0 depending on the expression. An example is given below using copy propagation aswell.

```
                    CALL _1             CALL _1
                    r2 := result        r2 := result
                    x := result         x := result
int x = test();     r3 := 0             r3 := 0
int a = 0 + x;      r4 := result        r4 := result
                    r5 := 0 + result    r5 := result
return a;           a := r5             a := result
                    r6 := r5            r6 := result
                    RETURN r5           RETURN result
```

4 Machine Code Generation