CM30171

Rob Willison

November 28, 2016

Contents

1 Introduction

This report is an explanation of the design decisions undertaken 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
```

2

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
```
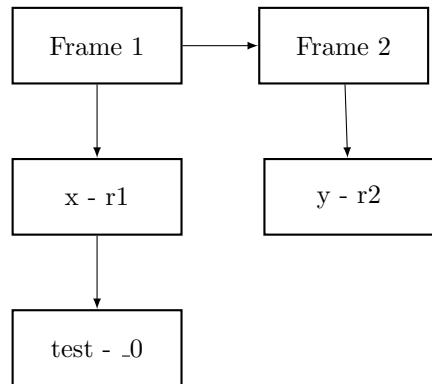
3.2 TAC Generation

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 loca-
tion. 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();
}
```

```
┌──────────┐        ┌──────────┐
│ Frame 1  │───────▶│ Frame 2  │
└──────────┘        └──────────┘
     │                   │
     ▼                   ▼
┌──────────┐        ┌──────────┐
│  x - r1  │        │  y - r2  │
└──────────┘        └──────────┘
     │
     ▼
┌──────────┐
│ test - _0│
└──────────┘
```
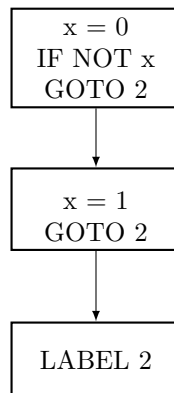
As you can see the environment holds the association between the tokens of the variables and functions with the TAC register location and the function label respectively.

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.

```
┌──────────┐
│  x = 0   │
│ IF NOT x │
│ GOTO 2   │
└──────────┘
     │
     ▼
┌──────────┐
│  x = 1   │
│ GOTO 2   │
└──────────┘
     │
     ▼
┌──────────┐
│ LABEL 2  │
└──────────┘
```

3.3   TAC Optimisation

4   Machine Code Generation