Part 60: Passing the Triple Test

In this part of our compiler writing journey, we will get the compiler to pass the triple test! How do I know? I've just got it to pass the triple test by changing a few source code lines in the compiler. But I don't yet know why the original lines are not working.

So, this part will be a investigation where we gather the clues, deduce the problem, fix it and finally get the compiler to pass the triple test properly.

Or, so I hope!

The First Piece of Evidence

We now have three compiler binaries:

1. `cwj` , built with the Gnu C compiler,
2. `cwj0` , built with the `cwj` compiler, and
3. `cwj1` , built with the `cwj0` compiler

The last two should be identical but they are not. Thus, `cwj0` isn't generating the right assembly output, and this is because of a flaw in the compiler's source code.

How can we narrow the problem down? Well, we have a pile of test programs in the `tests/` directory. Let's run `cwj` and `cwj0` over all these tests and see if there's a difference.

Yes there is, with `tests/input002.c` :

```
$ ./cwj -o z tests/input002.c ; ./z
17
$ ./cwj0 -o z tests/input002.c ; ./z
24
```

What's The Problem?

So, `cwj0` is producing incorrect assembly output. Let's start with the test source code:

```
void main()
{
  int fred;
  int jim;
  fred= 5;
  jim= 12;
  printf("%d\n", fred + jim);
}
```

We have two local variables, `fred` and `jim` . The two compilers produce assembly code with these differences:

```
42c42
<         movl    %r10d, -4(%rbp)
---
>         movl    %r10d, -8(%rbp)
51c51
<         movslq  -4(%rbp), %r10
---
>         movslq  -8(%rbp), %r10
```

Hmm, the second compiler is calculating the offset of `fred` incorrectly. The first compiler is correctly calculating the offset as `-4` below the frame pointer. The second compiler is calculating the offset as `-8` below the frame pointer.

What's Causing the Problem?

These offsets are being calculated by the function `newlocaloffset()` in `cg.c` :

```
// Create the position of a new local variable.
static int localOffset;
static int newlocaloffset(int size) {
  // Decrement the offset by a minimum of 4 bytes
  // and allocate on the stack
  localOffset += (size > 4) ? size : 4;
  return (-localOffset);
}
```

At the start of each function, `localOffset` is set to zero. As we create local variables, we get the size of each one, pass it to `newlocaloffset()` and get back the offset.

Both `fred` and `jim` local variables are `int` s, which are size 4. Therefore, their offsets should be `-4` and `-8` .

More Evidence, Please

Let's abstract `newlocaloffset()` into a separate source file, `z.c` (my "go to" temporary file name) and compile it. The source file is:

```
static int localOffset=0;
static int newlocaloffset(int size) {
  localOffset += (size > 4) ? size : 4;
  return (-localOffset);
}
```

And here is the output assembly with my comments:

```
        .data
localOffset:
        .long   0

        .text
newlocaloffset:
        pushq   %rbp
        movq    %rsp, %rbp              # Set up the stack and
        movl    %edi, -4(%rbp)          # frame pointers
        addq    $-16,%rsp
        movslq  localOffset(%rip), %r10  # Get localOffset into %r10
                                        # in preparation for the +=
        movslq  -4(%rbp), %r11          # Get size into %r11
        movq    $4, %r12                # Get  4   into %r12
        cmpl    %r12d, %r11d            # Compare them
        jle     L2                      # Jump if size < 4
```

```
        movslq  -4(%rbp), %r11
        movq    %r11, %r10              # Get size into %r10
        jmp     L3                      # and jump to L3
L2:
        movq    $4, %r11                # Otherwise get 4
        movq    %r11, %r10              # into %r10
L3:
        addq    %r10, %r10              # Add the += exression to the
                                        # cached copy of localOffset
        movl    %r10d, localOffset(%rip) # Save %r10 into localOffset
        movslq  localOffset(%rip), %r10
        negq    %r10                    # Negate localOffset
        movl    %r10d, %eax             # Set up the return value
        jmp     L1
L1:
        addq    $16,%rsp                # Restore the stack and
        popq    %rbp                    # frame pointers
        ret                             # and return
```

Hmm, the code is trying to do `localOffset += expression`, and we have a copy of `localOffset` cached in `%r10`. However, the expression itself also uses `%r10`, thus destroying the cached version of `localOffset`.

The `addq %r10, %r10`, in particular, is just wrong: it should be adding two different registers.

Passing the Triple Test by Cheating

We can pass the triple test by rewriting the source code to `newlocaloffset()`:

```
static int newlocaloffset(int size) {
  if (size > 4)
    localOffset= localOffset + size;
  else
    localOffset= localOffset + 4;
  return (-localOffset);
}
```

When we now do:

```
$ make triple
cc -Wall -o cwj  cg.c decl.c expr.c gen.c main.c misc.c opt.c scan.c stmt.c
sym.c tree.c types.c
./cwj    -o cwj0 cg.c decl.c expr.c gen.c main.c misc.c opt.c scan.c stmt.c
```

```
sym.c tree.c types.c
./cwj0   -o cwj1 cg.c decl.c expr.c gen.c main.c misc.c opt.c scan.c stmt.c
sym.c tree.c types.c
size cwj[01]
   text    data     bss     dec     hex filename
 109652    3028      48  112728   1b858 cwj0
 109652    3028      48  112728   1b858 cwj1
```

the last two compiler binaries are 100% identical. But this hides the fact that the original `newlocaloffset()` source code should work but it doesn't.

Why are we reallocating `%r10` when we know that it is allocated?

A Possible Culprit

I added back in to `cg.c` the `printf()` lines to see when registers were being allocated and freed. I noticed that, after these assembly lines:

```
        movslq  -4(%rbp), %r11          # Get size into %r11
        movq    $4, %r12                # Get   4   into %r12
        cmpl    %r12d, %r11d            # Compare them
        jle     L2                      # Jump if size < 4
```

all the registers are freed, even though `%r10` holds the cached copy of `localOffset`. Which function is generating these lines and freeing the registers? The answer is:

```
// Compare two registers and jump if false.
int cgcompare_and_jump(int ASTop, int r1, int r2, int label, int type) {
  int size = cgprimsize(type);

  // Check the range of the AST operation
  if (ASTop < A_EQ || ASTop > A_GE)
    fatal("Bad ASTop in cgcompare_and_set()");

  switch (size) {
  case 1:
    fprintf(Outfile, "\tcmpb\t%s, %s\n", breglist[r2], breglist[r1]);
    break;
  case 4:
    fprintf(Outfile, "\tcmpl\t%s, %s\n", dreglist[r2], dreglist[r1]);
    break;
  default:
    fprintf(Outfile, "\tcmpq\t%s, %s\n", reglist[r2], reglist[r1]);
  }
```

```
    fprintf(Outfile, "\t%s\tL%d\n", invcmplist[ASTop - A_EQ], label);
    freeall_registers(NOREG);
    return (NOREG);
  }
```

Looking at the code, we can definitely free `r1` and `r2`, so let's try that instead of freeing all the registers.

Yes, that helps, and all our regression tests still pass. However, another function is also freeing all the registers. It's time to use `gdb` and follow the execution.

The Real Culprit

It looks like the real culprit is that I forgot that many operations can be part of an expression, and I can't free all registers until the expression's result is either used or discarded.

As I looked at the execution with `gdb`, I saw that the code that deals with ternary operators is freeing registers, even though this may only be part of a bigger expression with registers already allocated (in `gen.c`):

```
static int gen_ternary(struct ASTnode *n) {
  ...
  // Generate the condition code
  genAST(n->left, Lfalse, NOLABEL, NOLABEL, n->op);
  genfreeregs(NOREG);           // HERE

  // Get a register to hold the result of the two expressions
  reg = alloc_register();

  // Generate the true expression and the false label.
  // Move the expression result into the known register.
  // Don't free the register holding the result, though!
  expreg = genAST(n->mid, NOLABEL, NOLABEL, NOLABEL, n->op);
  cgmove(expreg, reg);
  genfreeregs(reg);             // HERE

  // Generate the false expression and the end label.
  // Move the expression result into the known register.
  // Don't free the register holding the result, though!
  expreg = genAST(n->right, NOLABEL, NOLABEL, NOLABEL, n->op);
  cgmove(expreg, reg);
  genfreeregs(reg);             // HERE
```

```
  ...
}
```

Looking through `cg.c`, all the functions in there free registers that are no longer used, so I think that we can lose the `genfreeregs()` straight after the generation of the condition code.

Next up, once we move the true expression's value in the register reserved for the ternary result, we can free `expreg`. Ditto for the false expression's value.

To make this happen, I've made a previously-static function in `cg.c` global and renamed it:

```
// Return a register to the list of available registers.
// Check to see if it's not already there.
void cgfreereg(int reg) { ... }
```

We can now rewrite the ternary handling code in `gen.c`:

```
static int gen_ternary(struct ASTnode *n) {
  ...
    // Generate the condition code followed
  // by a jump to the false label.
  genAST(n->left, Lfalse, NOLABEL, NOLABEL, n->op);

  // Get a register to hold the result of the two expressions
  reg = alloc_register();

  // Generate the true expression and the false label.
  // Move the expression result into the known register.
  expreg = genAST(n->mid, NOLABEL, NOLABEL, NOLABEL, n->op);
  cgmove(expreg, reg);
  cgfreereg(expreg);
  ...
  // Generate the false expression and the end label.
  // Move the expression result into the known register.
  expreg = genAST(n->right, NOLABEL, NOLABEL, NOLABEL, n->op);
  cgmove(expreg, reg);
  cgfreereg(expreg);
  ...
}
```

With this change, the compiler now passes several tests:

- the triple test: `$ make triple`

- a quadruple test where we do one more compiler compilation:

```
$ make quad
...
./cwj  -o cwj0 cg.c decl.c expr.c gen.c main.c misc.c opt.c scan.c stmt.c
sym.c tree.c types.c
./cwj0 -o cwj1 cg.c decl.c expr.c gen.c main.c misc.c opt.c scan.c stmt.c
sym.c tree.c types.c
./cwj1 -o cwj2 cg.c decl.c expr.c gen.c main.c misc.c opt.c scan.c stmt.c
sym.c tree.c types.c
size cwj[012]
   text    data     bss     dec     hex filename
 109636    3028      48  112712   1b848 cwj0
 109636    3028      48  112712   1b848 cwj1
 109636    3028      48  112712   1b848 cwj2
```

- the regression tests with the Gnu C compiled compiler: `$ make test`
- the regression tests with our compiler compiled with itself: `$ make test0`

That feels very satisfying.

Conclusion and What's Next

I've reached the original goal of this journey: to write a self-compiling compiler. It's taken 60 parts, 5,700 lines of code, 149 regression tests and 108,000 words in the *Readme* files.

That said, this doesn't have to be the end of the journey. There is still a lot of work that could be done to the compiler to make it more production ready. However, I've been working sporadically on this for about two months now, so I feel like I can (at least) have a small break.

In the next part of our compiler writing journey, I will outline what more can be done with our compiler. Perhaps I'll do some of these things; perhaps you will. Next step