Part 58: Fixing Pointer Increments/Decrements

In the last part of our compiler writing journey, I mentioned that there was a problem with pointer increments and decrements. Let's see what the problem is and how I fixed it.

We saw with the AST operations A_ADD, A_SUBTRACT, A_ASPLUS and A_ASMINUS where one operand is a pointer and the other is an integer type, we need to scale the integer value by the size of the type that the pointer points at. In `modify_type()` in `types.c` :

```
// We can scale only on add and subtract operations
if (op == A_ADD || op == A_SUBTRACT ||
    op == A_ASPLUS || op == A_ASMINUS) {

  // Left is int type, right is pointer type and the size
  // of the original type is >1: scale the left
  if (inttype(ltype) && ptrtype(rtype)) {
    rsize = genprimsize(value_at(rtype));
    if (rsize > 1)
      return (mkastunary(A_SCALE, rtype, rctype, tree, NULL, rsize));
    else
      return (tree);          // Size 1, no need to scale
  }
}
```

But this scaling doesn't occur when we use `++` or `--`, either as preincrement/decrement or postincrement/decrement operators. Here, we simply strap an A_PREINC, A_PREDEC, A_POSTINC or A_POSTDEC AST node to the AST tree that we are operating on, and then leave it to the code generator to deal with the situation.

Up to now, this got resolved when we call either `cgloadglob()` or `cgloadlocal()` in `cg.c` to load the value of a global or local variable. For example:

```
int cgloadglob(struct symtable *sym, int op) {
  ...
  if (cgprimsize(sym->type) == 8) {
    if (op == A_PREINC)
      fprintf(Outfile, "\tincq\t%s(%%rip)\n", sym->name);
    ...
    fprintf(Outfile, "\tmovq\t%s(%%rip), %s\n", sym->name, reglist[r]);

    if (op == A_POSTINC)
      fprintf(Outfile, "\tincq\t%s(%%rip)\n", sym->name);
  }
  ...
}
```

Note, however, that the `incq` increments by one. That's fine if the variable we are incrementing is of integer type, but it fails to deal with variables that are of pointer type.

As well, the functions `cgloadglob()` and `cgloadlocal()` are very similar. They differ in what instructions we use to access the variable: is it at a fixed location, or a location relative to the stack frame.

Fixing the Problem

For a while I thought I could get the parser to build an AST tree similar to the one that `modify_type()` does, but I gave up on that. Thank goodness. I decided that, as `++` and `--` are already being done in `cgloadglob()`, that I should attack the problem here.

Halfway through, I realised that I could merge `cgloadglob()` and `cgloadlocal()` into a single function. Let's look at the solution in stages.

```
// Load a value from a variable into a register.
// Return the number of the register. If the
// operation is pre- or post-increment/decrement,
// also perform this action.
int cgloadvar(struct symtable *sym, int op) {
  int r, postreg, offset=1;
```

```
  // Get a new register
  r = alloc_register();

  // If the symbol is a pointer, use the size
  // of the type that it points to as any
  // increment or decrement. If not, it's one.
  if (ptrtype(sym->type))
    offset= typesize(value_at(sym->type), sym->ctype);
```

We start by assuming that we will be doing +1 as an increment. However, once we realise that we could be incrementing a pointer, we change this to the the size of the type that it points to.

```
  // Negate the offset for decrements
  if (op==A_PREDEC || op==A_POSTDEC)
    offset= -offset;
```

Now the `offset` is negative if we are going to do a decrement.

```
  // If we have a pre-operation
  if (op==A_PREINC || op==A_PREDEC) {
    // Load the symbol's address
    if (sym->class == C_LOCAL || sym->class == C_PARAM)
      fprintf(Outfile, "\tleaq\t%d(%%rbp), %s\n", sym->st_posn, reglist[r]);
    else
      fprintf(Outfile, "\tleaq\t%s(%%rip), %s\n", sym->name, reglist[r]);
```

This is where our algorithm differs from the old code. The old code used the `incq` instruction, but that limits the variable change to exactly one. Now that we have the variable's address in our register...

```
    // and change the value at that address
    switch (sym->size) {
      case 1: fprintf(Outfile, "\taddb\t$%d,(%s)\n", offset, reglist[r]); brea
      case 4: fprintf(Outfile, "\taddl\t$%d,(%s)\n", offset, reglist[r]); brea
      case 8: fprintf(Outfile, "\taddq\t$%d,(%s)\n", offset, reglist[r]); brea
    }
  }
```

we can add the offset on to the variable, using the register as a pointer to the variable. We have to use different instructions based on the size of the variable.

We've done any pre-increment or pre-decrement operation. Now we can load the variable's value into a register:

```
  // Now load the output register with the value
  if (sym->class == C_LOCAL || sym->class == C_PARAM) {
    switch (sym->size) {
      case 1: fprintf(Outfile, "\tmovzbq\t%d(%%rbp), %s\n", sym->st_posn, regl
      case 4: fprintf(Outfile, "\tmovslq\t%d(%%rbp), %s\n", sym->st_posn, regl
      case 8: fprintf(Outfile, "\tmovq\t%d(%%rbp), %s\n", sym->st_posn, reglis
    }
  } else {
    switch (sym->size) {
      case 1: fprintf(Outfile, "\tmovzbq\t%s(%%rip), %s\n", sym->name, reglist
      case 4: fprintf(Outfile, "\tmovslq\t%s(%%rip), %s\n", sym->name, reglist
      case 8: fprintf(Outfile, "\tmovq\t%s(%%rip), %s\n", sym->name, reglist[r
    }
  }
```

Depending on if the symbol is local, or global, we load from a named location or from an location relative to the frame pointer. We choose an instruction to zero pad the result based on the symbol's size.

The value is safely in register `r`. But now we need to do any post-increment or post-decrement. We can re-use the pre-op code, but we'll need a new register:

```
  // If we have a post-operation, get a new register
  if (op==A_POSTINC || op==A_POSTDEC) {
    postreg = alloc_register();

    // Same code as before, but using postreg

    // and free the register
    free_register(postreg);
  }

  // Return the register with the value
  return(r);
}
```

So the code for `cgloadvar()` is about as complex as the old code, but it now deals with pointer increments. The `tests/input145.c` test program verifies that this new code works:

```
int list[]= {3, 5, 7, 9, 11, 13, 15};
int *lptr;

int main() {
  lptr= list;
  printf("%d\n", *lptr);
  lptr= lptr + 1; printf("%d\n", *lptr);
  lptr += 1; printf("%d\n", *lptr);
  lptr += 1; printf("%d\n", *lptr);
  lptr -= 1; printf("%d\n", *lptr);
  lptr++   ; printf("%d\n", *lptr);
  lptr--   ; printf("%d\n", *lptr);
  ++lptr   ; printf("%d\n", *lptr);
  --lptr   ; printf("%d\n", *lptr);
}
```

How Did I Miss Modulo?

With this fixed, I went back to feeding the compiler source code to itself and found, to my amazement, that the modulo operators `%` and `%=` were missing. I have no idea why I hadn't put them in before.

New Tokens and AST Operators

Adding new operators to the compiler now is tricky because we have to synchronise changes in several places. Let's see where. In `defs.h` we need to add the tokens:

```
// Token types
enum {
  T_EOF,

  // Binary operators
  T_ASSIGN, T_ASPLUS, T_ASMINUS,
  T_ASSTAR, T_ASSLASH, T_ASMOD,
  T_QUESTION, T_LOGOR, T_LOGAND,
  T_OR, T_XOR, T_AMPER,
  T_EQ, T_NE,
  T_LT, T_GT, T_LE, T_GE,
  T_LSHIFT, T_RSHIFT,
  T_PLUS, T_MINUS, T_STAR, T_SLASH, T_MOD,
  ...
};
```

with T_ASMOD and T_MOD the new tokens. Now we need to create AST ops to match:

```
 // AST node types. The first few line up
// with the related tokens
enum {
  A_ASSIGN = 1, A_ASPLUS, A_ASMINUS, A_ASSTAR,                  //  1
  A_ASSLASH, A_ASMOD, A_TERNARY, A_LOGOR,                       //  5
  A_LOGAND, A_OR, A_XOR, A_AND, A_EQ, A_NE, A_LT,               //  9
  A_GT, A_LE, A_GE, A_LSHIFT, A_RSHIFT,                         // 16
  A_ADD, A_SUBTRACT, A_MULTIPLY, A_DIVIDE, A_MOD,               // 21
  ...
};
```

Now we need to add the scanner changes to scan these tokens. I won't show the code, but I will show the change to the table of token strings in `scan.c`:

```
// List of token strings, for debugging purposes
char *Tstring[] = {
  "EOF", "=", "+=", "-=", "*=", "/=", "%=",
  "?", "||", "&&", "|", "^", "&",
  "==", "!=", ",", ">", "<=", ">=", "<<", ">>",
  "+", "-", "*", "/", "%",
  ...
};
```

Operator Precedence

Now we need to set the operators' precedence in `expr.c`. T_SLASH used to be the highest operator but it's been replaced with T_MOD:

```
// Convert a binary operator token into a binary AST operation.
// We rely on a 1:1 mapping from token to AST operation
static int binastop(int tokentype) {
  if (tokentype > T_EOF && tokentype <= T_MOD)
    return (tokentype);
  fatals("Syntax error, token", Tstring[tokentype]);
  return (0);                   // Keep -Wall happy
}

// Operator precedence for each token. Must
// match up with the order of tokens in defs.h
static int OpPrec[] = {
  0, 10, 10,                    // T_EOF, T_ASSIGN, T_ASPLUS,
  10, 10,                       // T_ASMINUS, T_ASSTAR,
  10, 10,                       // T_ASSLASH, T_ASMOD,
  15,                           // T_QUESTION,
  20, 30,                       // T_LOGOR, T_LOGAND
```

```
  40, 50, 60,                   // T_OR, T_XOR, T_AMPER
  70, 70,                       // T_EQ, T_NE
  80, 80, 80, 80,               // T_LT, T_GT, T_LE, T_GE
  90, 90,                       // T_LSHIFT, T_RSHIFT
  100, 100,                     // T_PLUS, T_MINUS
  110, 110, 110                 // T_STAR, T_SLASH, T_MOD
};

// Check that we have a binary operator and
// return its precedence.
static int op_precedence(int tokentype) {
  int prec;
  if (tokentype > T_MOD)
    fatals("Token with no precedence in op_precedence:", Tstring[tokentype]);
  prec = OpPrec[tokentype];
  if (prec == 0)
    fatals("Syntax error, token", Tstring[tokentype]);
  return (prec);
}
```

Code Generation

We already have a `cgdiv()` function to generate the x86-64 instructions to do division. Looking at the manual for the `idiv` instruction:

> idivq S: signed divide `%rdx:%rax` by S. The quotient is stored in `%rax`. The remainder is stored in `%rdx`.

So we can modify `cgdiv()` to take the AST operation being performed, and it can do both division and remainder (modulo). The new function in `cg.c` is:

```
// Divide or modulo the first register by the second and
// return the number of the register with the result
int cgdivmod(int r1, int r2, int op) {
  fprintf(Outfile, "\tmovq\t%s,%%rax\n", reglist[r1]);
  fprintf(Outfile, "\tcqo\n");
  fprintf(Outfile, "\tidivq\t%s\n", reglist[r2]);
  if (op== A_DIVIDE)
    fprintf(Outfile, "\tmovq\t%%rax,%s\n", reglist[r1]);
  else
    fprintf(Outfile, "\tmovq\t%%rdx,%s\n", reglist[r1]);
  free_register(r2);
  return (r1);
}
```

The `tests/input147.c` confirms that the above changes work:

```
#include <stdio.h>

int a;

int main() {
  printf("%d\n", 24 % 9);
  printf("%d\n", 31 % 11);
  a= 24; a %= 9; printf("%d\n",a);
  a= 31; a %= 11; printf("%d\n",a);
  return(0);
}
```

Why Doesn't It Link

We are now at the point where our compiler can parse each and every of its own source code files. But when I try to link them, I get a warning about missing `L0` labels.

After a bit of investigation, it turns out that I wasn't properly propagating the end label for loops and switches in `genIF()` in `gen.c` . The fix is on line 49:

```
// Generate the code for an IF statement
// and an optional ELSE clause.
static int genIF(struct ASTnode *n, int looptoplabel, int loopendlabel) {
  ...
  // Optional ELSE clause: generate the
  // false compound statement and the
  // end label
  if (n->right) {
    genAST(n->right, NOLABEL, NOLABEL, loopendlabel, n->op);
    genfreeregs(NOREG);
    cglabel(Lend);
  }
  ...
}
```

Now that `loopendlabel` is being propagated, I can do this (in a shell script I call `memake`):

```
#!/bin/sh
make install

rm *.s *.o
```

```
for i in cg.c decl.c expr.c gen.c main.c misc.c \
        opt.c scan.c stmt.c sym.c tree.c types.c
do echo "./cwj -c $i"; ./cwj -c $i ; ./cwj -S $i
done

cc -o cwj0 cg.o decl.o expr.o gen.o main.o misc.o \
        opt.o scan.o stmt.o sym.o tree.o types.o
```

We end up with a binary, `cwj0`, which is the result of the compiler compiling itself.

```
$ size cwj0
   text    data     bss     dec     hex filename
 106540    3008      48  109596   1ac1c cwj0

$ file cwj0
cwj0: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
      linked, interpreter /lib64/l, for GNU/Linux 3.2.0, not stripped
```

Conclusion and What's Next

For the pointer increment problem, I definitely had to scratch my head quite a lot and look at several possible alternate solutions. I did get halfway through trying to build a new AST tree with an A_SCALE in it. Then I tossed it all away and went for the change in `cgloadvar()`. That's much nicer.

The modulo operators were simple to add (in theory), but annoyingly difficult to get everything synchronised (in practice). There is probably some scope to refactor here to make the synchronisation much easier.

Then, while trying to link all the object files that our compiler had made from its own source code, I found that we were not propagating loop/switch end labels properly.

We've now reached the point where our compiler can parse every one of its source code files, generate assembly code for them, and we can link them. We have reached the final stage of our journey, one that is probably going to be the most painful, the **WDIW** stage: why doesn't it work?

Here, we don't have a debugger, we are going to have to look at lots of assembly output. We'll have to single-step assembly and look at register values.

In the next part of our compiler writing journey, I will start on the **WDIW** stage. We are going to need some strategies to make our work effective. Next step