Part 17: Better Type Checking and Pointer Offsets

A couple of parts ago, I introduced pointers and implemented some code to check type compatibility. At the time, I realised that, for code like:

```
int   c;
int  *e;

e= &c + 1;
```

the addition of one to the pointer calculated by `&c` would need to be converted into the size of `c`, to ensure we skip to the next `int` in memory after `c`. In other words, we would have to scale the integer.

We need to do this for pointer, and later on we will need to do this for arrays. Consider the code:

```
int list[10];
int x= list[3];
```

To do this, we need to find the base address of `list[]`, then add on three times the size of `int` to find the element at index position 3.

At the time, I'd written a function in `types.c` called `type_compatible()` to determine if two types were compatible, and to indicate if we needed to "widen" a small integer type so that it was the same size as a larger integer type. This widening, though, was performed elsewhere. In fact, it ended up being done in three places in the compiler.

A Replacement for `type_compatible()`

If `type_compatible()` indicated so, we would A_WIDEN a tree to match a larger integer type. Now we need to A_SCALE a tree so that its value is scaled by the size of a type. And I want to refactor the duplicate widening code.

To this end, I've thrown out `type_compatible()` and replaced it. This took quite a bit of thinking, and I probably will have to tweak or extend it again. Let's look at the design.

The existing `type_compatible()`:

- took two type values as arguments, plus an optional direction,
- returned true if the types were compatible,
- returned A_WIDEN on the left or right if either side needed to be widened,
- didn't actually add the A_WIDEN node to the tree,
- returned false if the types were not compatible, and
- didn't handle pointer types

Now let's look at the use cases for type comparisons:

- when performing a binary operation on two expressions, are their types compatible and do we need to widen or scale one?
- when doing a `print` statement, is the expression an integer and does it need widening?
- when doing an assignment statement, does the expression need widening and does it match the lvalue type?
- when doing a `return` statement, does the expression need widening and does it match the return type of the function?

In only one of these use cases do we have two expressions. Therefore, I've chosen to write a new function that takes one AST tree and the type we want it to become. For the binary operation use case, we will call it twice and see what happens for each call.

Introducing `modify_type()`

`modify_type()` in `types.c` is the replacement code for `type_compatible()`. The API for the function is:

```
// Given an AST tree and a type which we want it to become,
// possibly modify the tree by widening or scaling so that
// it is compatible with this type. Return the original tree
// if no changes occurred, a modified tree, or NULL if the
// tree is not compatible with the given type.
// If this will be part of a binary operation, the AST op is not zero.
struct ASTnode *modify_type(struct ASTnode *tree, int rtype, int op);
```

Question: why do we need whatever binary operation is being performed on the tree and some other tree? The answer is that we can only add to or subtract from pointers. We can't do anything else, e.g.

```
  int x;
  int *ptr;

  x= *ptr;         // OK
  x= *(ptr + 2);   // Two ints up from where ptr is pointing
  x= *(ptr * 4);   // Does not make sense
  x= *(ptr / 13);  // Does not make sense either
```

Here is the code for now. There are lots of specific tests, and at present I can't see a way to rationalise all the possible tests. Also, it will need to be extended later.

```
struct ASTnode *modify_type(struct ASTnode *tree, int rtype, int op) {
  int ltype;
  int lsize, rsize;

  ltype = tree->type;

  // Compare scalar int types
  if (inttype(ltype) && inttype(rtype)) {

    // Both types same, nothing to do
    if (ltype == rtype) return (tree);

    // Get the sizes for each type
    lsize = genprimsize(ltype);
    rsize = genprimsize(rtype);
```

```
    // Tree's size is too big
    if (lsize > rsize) return (NULL);

    // Widen to the right
    if (rsize > lsize) return (mkastunary(A_WIDEN, rtype, tree, 0));
  }

  // For pointers on the left
  if (ptrtype(ltype)) {
    // OK is same type on right and not doing a binary op
    if (op == 0 && ltype == rtype) return (tree);
  }

  // We can scale only on A_ADD or A_SUBTRACT operation
  if (op == A_ADD || op == A_SUBTRACT) {

    // 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, tree, rsize));
    }
  }

  // If we get here, the types are not compatible
  return (NULL);
}
```

The code to add the AST A_WIDEN and A_SCALE operations are now done here in one place only. The A_WIDEN operation converts the child's type to the parent's type. The A_SCALE operation multiplies the child's value by the size which is store in the new `struct ASTnode` union field (in `defs.h`):

```
// Abstract Syntax Tree structure
struct ASTnode {
  ...
  union {
    int size;                   // For A_SCALE, the size to scale by
  } v;
};
```

Using the New `modify_type()` API

With this new API, we can remove the duplicated code to A_WIDEN which is in `stmt.c` and `expr.c`. However, this new function only takes one tree. This is fine when we indeed only have one tree. There are three calls now to `modify_type()` in `stmt.c`. They are all similar, so here is the one from `assignment_statement()`:

```
    // Make the AST node for the assignment lvalue
    right = mkastleaf(A_LVIDENT, Gsym[id].type, id);

    ...
    // Parse the following expression
    left = binexpr(0);

    // Ensure the two types are compatible.
    left = modify_type(left, right->type, 0);
    if (left == NULL) fatal("Incompatible expression in assignment");
```

which is so much cleaner than the code we had before.

And in `binexpr()` ...

But in `binexpr()` in `expr.c`, we now need to combine two AST trees with a binary operations like addition, multiplication etc. Here, we try to `modify_type()` each tree with the other tree's type. Now, one may widen: this also implies that the other will fail and return NULL. Thus, we can't just see if one result from `modify_type()` is NULL: we need to see both be NULL to assume a type incompatibility. Here's the new comparison code in `binexpr()`:

```
  struct ASTnode *binexpr(int ptp) {
    struct ASTnode *left, *right;
    struct ASTnode *ltemp, *rtemp;
    int ASTop;
    int tokentype;

    ...
    // Get the tree on the left.
    // Fetch the next token at the same time.
    left = prefix();
    tokentype = Token.token;

    ...
    // Recursively call binexpr() with the
    // precedence of our token to build a sub-tree
```

```
  right = binexpr(OpPrec[tokentype]);

  // Ensure the two types are compatible by trying
  // to modify each tree to match the other's type.
  ASTop = arithop(tokentype);
  ltemp = modify_type(left, right->type, ASTop);
  rtemp = modify_type(right, left->type, ASTop);
  if (ltemp == NULL && rtemp == NULL)
    fatal("Incompatible types in binary expression");

  // Update any trees that were widened or scaled
  if (ltemp != NULL) left = ltemp;
  if (rtemp != NULL) right = rtemp;
```

The code is a little bit messy but no more that what was previously there, and it now deals with A_SCALE as well as A_WIDEN.

Performing the Scaling

We have added the A_SCALE to the list of AST node operations in `defs.h`. Now we need to implement it.

As I mentioned before, the A_SCALE operation multiplies the child's value by the size which is store in the new `struct ASTnode` union field. For all of our integer types, this will be a multiple of two. Because of this fact, we can multiply the child's value with a shift left of a certain number of bits.

Later on, we will have structs that have a size which isn't a power of two. So we can do a shift optimisation when the scale factor is suitable, but we also need to implement a multiply for a more general scale factor.

Here is the new code that does this in `genAST()` in `gen.c`:

```
  case A_SCALE:
    // Small optimisation: use shift if the
    // scale value is a known power of two
    switch (n->v.size) {
      case 2: return(cgshlconst(leftreg, 1));
      case 4: return(cgshlconst(leftreg, 2));
      case 8: return(cgshlconst(leftreg, 3));
      default:
        // Load a register with the size and
        // multiply the leftreg by this size
```

```
            rightreg= cgloadint(n->v.size, P_INT);
            return (cgmul(leftreg, rightreg));
```

Shifting Left in x86-64 Code

We now need a `cgshlconst()` function to shift a register value left by a constant. When we add the C '<<' operator later, I will write a more general shift left function. For now, we can use the `salq` instruction with an integer literal value:

```
// Shift a register left by a constant
int cgshlconst(int r, int val) {
  fprintf(Outfile, "\tsalq\t$%d, %s\n", val, reglist[r]);
  return(r);
}
```

Our Test Program that Doesn't Work

My test program for the scaling functionality is `tests/input16.c` :

```
int   c;
int   d;
int  *e;
int   f;

int main() {
  c= 12; d=18; printint(c);
  e= &c + 1; f= *e; printint(f);
  return(0);
}
```

I was hoping that `d` would be placed immediately after `c` by the assembler when we generate these assembly directives:

```
        .comm   c,1,1
        .comm   d,4,4
```

But when I compiled the assembly output and checked, they were not adjacent:

```
$ cc -o out out.s lib/printint.c
$ nm -n out | grep 'B '
```

```
0000000000201018 B d
0000000000201020 B b
0000000000201028 B f
0000000000201030 B e
0000000000201038 B c
```

`d` actually comes *before* `c`! I had to work out a way to ensure the adjacency, so I looked at the code that *SubC* generates here, and changed our compiler to now generate this:

```
        .data
        .globl  c
c:      .long   0       # Four byte integer
        .globl  d
d:      .long   0
        .globl  e
e:      .quad   0       # Eight byte pointer
        .globl  f
f:      .long   0
```

Now when we run our `input16.c` test, `e= &c + 1; f= *e;` gets the address of the integer one up from `c` and stores that integer's value in `f`. As we declared:

```
int   c;
int   d;
...
c= 12; d=18; printint(c);
e= &c + 1; f= *e; printint(f);
```

we will print out both numbers:

```
cc -o comp1 -g -Wall cg.c decl.c expr.c gen.c main.c misc.c
      scan.c stmt.c sym.c tree.c types.c
./comp1 tests/input16.c
cc -o out out.s lib/printint.c
./out
12
18
```

Conclusion and What's Next

I feel a lot happier with the code that converts between types. Behind the scenes, I wrote some test code that supplied all possible type values to `modify_type()`, as well as a binary operation and zero for the operation. I eyeballed the output and it seems to be what I want. Only time will tell.

In the next part of our compiler writing journey, I don't know what I will do! Next step