How to Compile C and C^{++}.

December 21, 2009

The aim of this text is to understand in detail how a C^{++}-compiler works. For the moment, we do not cover the complete language. In particular, we will ignore inheritance and enumeration types. If we have time left, we will say something about it later.

1 Type System for C^{++}/C

Ideally, types are defined by recursive type construction rules. These are rules of the form: If A, B are types, then $\Phi(A, B)$ is also a type. The function space constructor $A \to B$ in λ-calculus has this form. Another example is the product constructor $A \times B$.

Most types in C^{++} are recursive (array of, pointer to), but `struct` is a problem. The problem is that struct definitions can be circular, as in the following:

```
struct list
{
   int element;
   list* nextinlist;
};

list( );
~list( );
void mergewith( const list& l );
```

The member `nextlinlist` is of type `list`. The member function `mergewith` has an argument of type `const list&`.

Circularities can also extend over more than one struct:

```
struct aa;

struct bb
{
   aa* a;

   aa getthea( ) const;
```

```
};

struct aa
{
   bb* b;
   bb gettheb( ) const;
};

aa bb::getthea( ) const
{
   return *a;
}

bb aa::gettheb( ) const
{
   return *b;
}
```

In order to solve this problem, C^{++} distinguishes *complete* and *incomplete* definitions. The first definition `struct aa;` is incomplete. Incomplete definitions can only be used in declarations of pointers, references, and as arguments of incomplete function definitions.

Definition 1.1 *We define the types of C^{++} in two steps. First, we give a recursive definition of the possible types: Assume an infinite set of struct variables, which we will denote as $s_1, \ldots, s_n$.*

- *The primitive types* $\text{bool}, \text{char}, \text{int}, \text{unsigned int}$ *and* double *are primitive types.*
- *Each struct variable s_i is a type.*
- *If T is a type, then* $\text{pointerto}(T)$ *is a type.*
- *If T is a type, then* $\text{refto}(T)$ *is a type. This is the translation for the usual reference type.*
- *If T is a type, then* $\text{refpointerto}(T)$ *is a type. A reference pointer is a pointer that has been a reference before.*
- *If T is a type, and N is an unsigned integer, then* $\text{array}(N, T)$ *and* $\text{array}(T)$ *are types.*
- *Each enumeration type is a type.*
- *If $T_1, \ldots, T_n, U$ are types, then $U(T_1, \ldots, T_n)$ is a type. (The type of functions with arguments $T_1, \ldots, T_n$ and result U.)*

Each type (except $\text{refpointerto}(X)$ *and* $\text{pointerto}(X)$*) has two versions, namely* const *and* $\text{non} - \text{const}$.

Incomplete types will be used later for uninitialized memory. This is memory that is allocated, which will eventually contain an object, but for which the constructor has not yet been called.
pointerto is the standard pointer type. refto is used for references. After checking, references are replaced by pointers. (Because that's what they are, after all.) We use a special kind of pointer refpointerto in order to remember that the pointer originates from a reference. refpointerto is more restricted than pointerto because on refpointerto, no pointer arithmetic is allowed.
The relation between pointers and arrays in C is subtle. It seems that the behaviour of arrays can be explained as follows: By default, a variable represents a reference to its type. If one has `X x`, then by default, x is of type `X&`. In situations, where an `X` is required, the copy constructor of `X` is used to transform `X&` into `X`. Types of form arrayof(N, X) have no copy constructor. Instead, they can be converted to pointerto(X).

Definition 1.2 *A struct definition S is a finite list of triples of form (I, T, c), where*

- *I is an identifier,*
- *T is a C^{++} type,*
- *c is a storage class, which is either non-static or static.*

A type system is a finite list of struct declarations $(S_1, \ldots, S_n)$. Each S_i defines the struct variable s_i. The list $(S_1, \ldots, S_n)$ must satisfy the following condition:

- *If variable s_j occurs in declaration S_i and $j \geq i$, then s_j is in the scope of at least one* pointerto, refto, *or* explicitrefto. *(This is the forward reference condition)*

A static member of a struct that is not a function can be viewed as a global variable. As far as I can see, its only relation with its struct is of syntactical nature. It can be viewed as an independent global variable that happens to be in the same namespace as the struct.

For functions, static meens something different: (All member functions are static in the sense that they are not part of any object. Different objects do not have different member functions.) For a member function, static means that the function can only adress static variables of the the struct.

2 Intermediate Representation

In the intermediate representation, we try to make explicit as much as possible, without losing algebraic semantics. If one gives up algebraic semantics, analysis (needed for optimization) becomes much harder. In the intermediate representation, the following things are made explicit:

- **References, constructor calls and implicit `this`-arguments.** Consider the assignment in the following program fragment:

```
struct X
{
   X operator = ( X );               // Def 1.
   const X& operator = ( const X& ); // Def 2.
};

main( )
{
   X x1;
   X x2;

   x1 = x2;
}
```

  By default, the variable `x2` is of type `X&`. If Def2 is used, the reference `x2` can be passed to the assignment operator without further conversion. If Def1 is used, the reference `x2` has to be changed into an object of type `X`, which is done by the copy constructor. The compiler has to decide whether a copy constructor is necessary and to insert it, if necessary.

  In order to modify `x1`, `operator =` needs to have access to it. This is possible, because `operator =` is a method of `struct X`. Inside the definition of `operator =`, the user can use a variable `this` of type `X*` to modify the assigned variable.

  As a consequence, assignment must be a binary function with argument types pointerto(X) and refpointerto(X). It would be more natural to have the first argument of type refpointerto(X) as well, but it is a pointer in C^{++} for historical reasons, and we will stick with that.

- **Initializations and unitialized memory.**

  The best solution would be if memory could be always initialized at the same time when it is allocated, but unfortunately this will not be possible.

  As a general rule, a constructor of some `struct X` cannot allocate by itself the memory to which it will write its `X`. It is possible that the memory is allocated on the heap, and the constructor cannot do the heap allocation. If the members of `struct X` have their own constructors, then if the constructor for `struct X` allocates its memory, the constructors for the members cannot do their own allocation.

  If one wants to compile a function call `f(t)` with signature `X f( Y y )` one has two choices: Either one first allocates space for the result `X`, then computes t, and after that calls f. In this case the `X`-constructor cannot

allocate by itself. The alternative is that one first computes t, then calls `f`, which allocates and constructs its result `X`. When that is done, the X has to be moved when t is cleaned up, which involves calling the copy constructor for `X`.

Therefore, we need unitialized memory as separate type. We will use an array of characters as unitialized memory. A constructor for type X is a function that has (at least) one argument of type **refpointerto**(array(sizeof(X), char).

- **Resolving Overloading**

  + on integers is another operator than + on double. Adition between pointers and integers is yet another operator. When an integer I is added to a pointer of type `X*`, the new value of the pointer will be $p+I$.sizeof(X), and the + that we used in this last expression is yet another +. In the intermediate representation, all these +-ses will have different names, so that there will be no remaining ambiguity.

One of the design aims of the intermediate representation is that it should be suitable for optimization. In addition, there should exist an easy transformation into the final code.

In order to resolve ambiguity, we will allow typenames as defined in Definition 1.2 to be part of function names. This makes it easy to give names to instances of polymorphic methods, like for + or = . So a function name will have form

$$U_1 :: U_2 :: \cdots :: U_n,$$

where each U_i is either an identifier, or a type name put between $\langle\ \rangle$. We give a few examples:

- The sizeof operator for type T has name sizeof :: $\langle T \rangle$. For the type `const char*`, it has name sizeof :: $\langle$pointerto(const char)$\rangle$.
- A member function x of struct X can have name $\langle$structvar $N\rangle :: x$, where N is the number of struct variable that represents X.
- The new function for type X has name new :: $\langle X \rangle$. It returns an object of type pointerto(array(sizeof :: $\langle X \rangle$, char)).

Figure 1 gives a list of the elementary operators. These are operators that deal with references, copy constructors, assignment and initialization. We will give some explanations:

- If f is a function of type int f(int), then it is not possible to directly apply f on a variable of type x, because a variable represents a reference, i.e. it has type refpointerto(int). In order to transform refpointerto(int) to int, the operator copy :: $\langle$int$\rangle$ (refpointerto(int) has to be applied. Then the complete expression is

  $$f(\text{copy} :: \langle \text{int} \rangle\ (x)),$$

  which is of type int.

- Arrays are somewhat special in C/C^{++} because they have no copy constructor. Instead, they can be transformed into a pointer. Suppose that the function strcpy has type void strcpy(pointerto(char), pointerto(char)), and that we have two variables p, q of type array(char). The expression `strcpy(p,q)` has to be translated as follows:

  $$\text{strcpy}(\text{array2pointer} :: \langle \text{char} \rangle\ (p), \text{array2pointer} :: \langle \text{char} \rangle\ (q)\ ).$$

  The array2pointer function does nothing, except changing the type of a reference to an array of char into a pointer to char.

- We assume that for primitive data types, assignment has form T assign :: $\langle T \rangle$ (refpointer(T), T). The first argument is the variable being assigned to, so that it must be a reference pointer. For the second argument, one has the choice of using a reference to T, or T itself. Since primitive types are small, we choose T.

  The default assignment for user defined types has form

  $$T \text{ assign} :: \langle T \rangle\ (\text{refpointer}(T), \text{refpointer}(\text{const } T)).$$

  The second argument is changed into a const reference, so that the overhead of an additional copy constructor is avoided.

  For user defined assignment operators, the type of the first argument is changed into pointerto(T), because it is a method of T which has a `this`-variable. The type of the second argument is chosen by the user.

- The purpose of the init-operator is to overwrite raw memory with its first value. (The init-function will not appear in the final code, because every function will obtain an additional argument that tells it where to write its result.)

  As example, consider the following code fragment

  ```
  {
     int x = 3;
     x = f(x);
  }
  ```

  First the variable x is created:

  $$\text{int } x;$$

  After its creation, x is not yet an integer. It has type array(sizeof :: $\langle \text{int} \rangle()$, char). Now one can do

  $$\text{init} :: \langle \text{int} \rangle(x, 3),$$

  after which x is of type int. The assigmnent is translated as

  $$\text{assign} :: \langle \text{int} \rangle(x, f(\text{copy} :: \langle \text{int} \rangle(x)).$$

Figure 1: Elementary Copying/Assignment Operations

- If T is not an array type, then T has a copy constructor:

  $$T \text{ copy} :: \langle T \rangle \text{ (refpointerto}(T)),$$

  and

  $$T \text{ copy} :: \langle T \rangle \text{ (refpointerto(const } T).$$

  The copy constructor is either default, or defined by the user (for struct variables)

- For every type T, there are operators

  $$\text{pointerto}(T) \text{ array2pointer} :: \langle T \rangle \text{ (refpointerto(array}(N, T))),$$

  and

  $$\text{pointerto}(T) \text{ array2pointer} :: \langle T \rangle \text{ (refpointerto(array}(T))).$$

- If T is one of $\text{char}, \text{bool}, \text{unsigned int}, \text{int}, \text{double}$, of form pointerto(U), then T has a built-in assignment operator

  $$T \text{ assign} :: \langle T \rangle (\text{refpointerto}(T), T).$$

- If T is a struct variable, for which the user defined assignment operators, they must have one of the following forms:

  $$U \text{ assign} :: \langle T \rangle \text{ (pointerto}(T), V),$$

  $$U \text{ assign} :: \langle T \rangle \text{ (pointerto}(T), \text{const } V),$$

  $$U \text{ assign} :: \langle T \rangle \text{ (pointerto}(T), \text{refpointerto}(V)),$$

  $$U \text{ assign} :: \langle T \rangle \text{ (pointerto}(T), \text{refpointerto(const } V)).$$

- If for a struct variable T, the user did not define an assignment operator, then the default assignment operator has the following form:

  $$\text{refpointer(const } T) \text{ assign} :: \langle T \rangle \text{ (refpointerto}(T), \text{refpointerto(const } T)).$$

- Every type T has a sizeof operator

  $$\text{unsigned int sizeof} :: \langle T \rangle (\ ).$$

- Every type T, possibly const, has an initialization operator

  $$\text{init} :: \langle T \rangle (\text{refpointerto(array(sizeof} :: \langle T \rangle (), \text{char}), T).$$

Figure 2 contains operators that are related to adressing. The first operator adressoff takes the adress of an object and returns it as pointer. It is essentially the translation of `&` in C^{++}. In its normal use, `&` is applied on a reference, like in `int* p = &q;` with `q` an int variable. Since a reference is already an adress, the operator actually does nothing, except for changing the type.
The complement of adressof is deref. deref is the translation of `*` in C^{++}. If `p` is of type `char*`, then `*p` is of type `char&`, so that `*` also does nothing, except for changing the type.
In C^{++}, it is forbidden to take the adress of a constant, (like in `const int*p = &4;`), but it is allowed to make a reference from a constant. This gives the strange situation that the following code is legal:

```
inline const int* cheat( const int& x ) { return &x; }

const int*p = cheat(4); // Legal.
const int*p = &4;       // Illegal, but with the same result.
```

The third operator in Figure 2 takes an object, and makes a reference from it. (This is implicitly used by the `cheat` function above.) It is part of the standard of C^{++} that it is possible to supply an object of type `X`, where in fact an reference of type `const X&` is needed. It is guaranteed that the `X` is kept alive until the expression is completely evaluated.

```
const X& operator = ( const X& );
X f(int);

X x1, x2, x3, x4;
x1 = ( x2 = ( x3 = ( x4 = f(2))));
   // Legel C++, f(2) is kept and its reference
   // is passed through all the assignments.
```

The pointer addition functions add do not involve any multiplication. If one wants to add `i` to a pointer `p`, one first has to multiply i by the size of `*p`. If `p` is of type `int*`, then the expression `p[i]` is translated into the following intimidating expression:

$$\text{deref} :: \langle \text{int} \rangle \ (\ \text{add} :: \langle \text{pointerto(int)},\ \text{int},\ \text{pointerto(int)}\ \rangle$$

$$(\text{copy} :: \langle \text{pointerto(int)} \rangle\ (p)),\ \text{times} :: \langle \text{int} \rangle (\text{copy} :: \langle \text{int} \rangle (i),\ \text{sizeof} :: \langle \text{int} \rangle ())).$$

If p would be declared as `int p[100]`, then

$$\text{copy} :: \langle \text{pointerto(int)} \rangle\ (p)$$

would have to be replaced by

$$\text{array2pointer} :: \langle \text{int} \rangle\ (p).$$

Very probably, one also has to insert a cast from unsigned int to int around sizeof $:: \langle \text{int} \rangle\ ()$.

Figure 2: Adressing and Referencing Operators

- For every type T, possibly const, there is an adress operator

$$\text{pointerto}(T)\ \text{adressof} :: \langle T \rangle\ (\text{refpointerto}(T)).$$

- For every type T, possibly const, there is a dereferencing operator

$$\text{refpointerto}(T)\ \text{deref} :: \langle T \rangle\ (\text{pointerto}(T)).$$

- For every type T, there is a reference operator

$$\text{refpointerto}(\text{const } T)\ \text{makeref} :: \langle T \rangle\ (T).$$

- Pointers and reference pointers have addition operators of the following form

$$Z\ \text{add} :: \langle X, Y, Z \rangle\ (X, Y).$$

  Here X is the input (pointer or reference), Y is the value that is added to it (int or unsigned int), and Z is the result of the addition (again pointer or reference). If X refers or points to something that is const, then the type that Z refers or points to must be also const.

Reference addition is used for accessing the fields of a struct. If `x` is of type `struct X`, and `f` is a field of `X`, then `x.f` denotes reference addition. The reference to `x` is increased by the position of field `f` in `X` and cast into a reference to the type of `f`. The intermediate representation is as follows:

$$\text{add} :: \langle \text{refpointer}(X), \text{unsigned int}, \text{refpointer}(F) \rangle (x, N),$$

where F is the type of field `f`, and N is some number derived from the position of field `f` in `struct X`.

We made the types of add very liberal, so that the optimizer can combine different additions, like for example in the expression `p. x[i]. y [j]`, into a single addition.

3 Intermediate Result of Compilation

- We assume we have goto, jumpfalse refpointerto(bool), and labels.
- create v array(n, char). Create unitialized variable v with size n on stack.
- release v. Drop variable v from stack.
- Function calls are always on the last allocated variables. The first argument is a refpointerto the place where the result must be written.

Figure 3: Arithmetic Operators

- The types bool, char, int, unsigned int, double, and also pointerto(const char) have constants. When the type matters, we write

  $$\text{const} :: \langle X \rangle (n),$$

  otherwise just n.

- For each of the types bool, char, int, unsigned int, double, there are the arithmetic operators add, sub, mult, div which have form:

  $$X \text{ add} :: \langle X \rangle (X, X).$$

- For each of the types bool, char, int, unsigned int, double, there are comparison operators lessthan, greaterthan, lessequalthan, greaterequalthan, equal, notequal with form

  $$\text{bool lessthan} :: \langle X \rangle \ (X, X).$$

- There are cast operations of form

  $$Y \text{ cast} :: \langle X, Y \rangle (X),$$

  which take an X and transform it into a Y.

We will invent other primitive operations when we need them. (Bit operations, unary operations, etc.)

4 Unfolding

In the unfolded version, functions are used only in initializations.

Expressions are unfolded as follows:

1. If the expression contains makeref-operators, (which transform an object into a reference), then create local a local variable for the object.

2. As long as we have nested expressions of form, $init(v, f(t_1, \ldots, t_n))$, proceed as follows: If not void, create local variables for the t_i. Replace expression by

```
create v1
init( v1, t1 ),
...
create vn
init( vn, tn ),

init( v, f( v1, ..., vn ).
```

   In case one of the t_i was a makeref, we initialize the variable created in step 1, and then store the adress of it in v_i. For subterms of type void, there is no init.

This is also the point on which inline functions can be expanded. They are instantiated away. The return statement is replaced by

1. initialization of the result.

2. Cleanup of local variables.

3. A goto out of the function.