Intermediate Code Generation

Transforming the abstract syntax tree (AST) into intermediate representation is called lowering.

We assume that intermediate representation is close to LLVM.

We also assume that the AST has been type checked, and that conversions have been inserted.

Lowering control statements **while, if, do** is trivial.

Most of these slides is about lowering expression trees. This is not difficult in essence, but the details irritate.

One recursively walks through the expression tree, and emits code that computes the result of the tree. Dependent on the type of a subtree, the emitted code writes the result in an LLVM-variable, or in a memory location. If the result is written in a memory location, the surrounding context decides about the address and passes a pointer to the subtree.

Lowering If

The statement **if** B **then** $S1$ **else** $S2$ is lowered as:

```
    .... evaluate B into %b
    br i1 %b, label %L1, label %L2

; <label>:L1
    ( Lowering of S1 )
    br label %L3

; <label>:L2
   ( Lowering of S2 )
   br label %L3

; <label>:L3
```

Lowering While

The statement **while** B **do** S is lowered as:

```
; <label>: L1
   evaluate B into %b
   br i1 %b, label %L2, label %L3

; <label>: L2
   ( Lowering of S )
   br label %L1

; <label>: L3
```

Rest of these slides is about lowering of expressions.

The Joy of Temporaries

Temporaries are needed when some function creates an object of type T as argument to a function that requires a **ref**(**const**(T)).

For example:

```
std::ostream& operator << ( std::ostream& , const X& x );

X f( int x );

std::cout << X(3) << "\n";
```

Note: I am ignoring Rvalue references.

Temporaries (2)

```
const A& max( const A& a1, const A& a2 )
{
   if( a1 > a2 )
      return a1;
   else
      return a2;
}

A operator + ( const A& a1, const A& a2 )
{
   A res = a1;
   ... Add a2 to res in some way.
   return res;
}
```

Temporaries

How to evaluate?

```
std::cout << max( max( 1 + 2, 3 + 4 ),
                  max( 5 + 6, 7 + 8 )) +
             max( max( 10 + 11, 12 + 3 ),
                  max( 8 + 9, 12 + 16 )) << "\n";
```

C^{++}-standard: Temporaries have to be kept alive until expression is fully evaluated.

The Joy of Temporaries, Conditionally Created

Type A has destructor, and we use conditional evaluation:

```
std::cout << ( b ? max( a + 1, b + 1 ) :
                   max( c + 1, d + 1 ) ) << "\n";
```

We need to remember which temporaries have have been constructed!

Box-Operators

Every type T has a box operator $\mathbf{box}_T$.

Its input type is T, and its output type is $\mathbf{ref}(\mathbf{const}(T))$.

The type checker has inserted them, and we can attach a temporary variable to every occurrence of **box** during lowering.

Box-Operators (2)

Before an expression t is lowered, do the following:

For every occurrence of $\mathbf{box}_T$ that occurs in it, introduce two LLVM-variables `%b` and `%p` of types `i1` and `T*`.

Also emit the code `%p = alloca T;  %b = i1 0;`

The boolean `%b` is used for remembering if the temporary was constructed.

(If the occurrence of $\mathbf{box}_T$ is not in scope of a conditional operator, or type T has no destructor, every reasonable optimizer will remove the boolean.)

Simple/Non-simple Types

We assume a distinction between simple and non-simple types.

Primitive types are always simple.

Classes that have non-trivial copy-constructor/assignment/destructor are always non-simple.

Non-trivial means: Doing something different than just byte copying.

Other **struct** types can be simple if they are not too big.

In CLANG, a complex number is still simple, while a quaternion is not anymore.

Requirements on Type Checking

We assume that the expression tree has been type checked, and that necessary conversions have been inserted everywhere in the tree.

In addition, we assume that

- in every statement of form **return** t, term t has the return type of the function,
- in every initialization of form $V\ v = t$, term t has type V.

How Functions are Called

A function f must be declared with $n \geq 0$ input types $T_1, \ldots, T_n$, and possibly one output type U.

If the function is a member of some class C, we assume one additional parameter **this** of type **pntr**(C) or **pntr**(**const**(C)).

Passing of Non-Simple Parameters

Simple parameters are passed to the function directly.

Non-simple parameters are passed in memory.

The calling environment creates the object, and passes a pointer to the object to the function.

This implies that whenever a function is declared with a parameter of type T with T non-simple, it will get a $\mathbf{pntr}(T)$ in reality.

Returning a Value from a Function

If the function returns a value U that is simple, it is returned through a `ret` statement.

If U is non-simple, it has to be returned in memory.

The calling environment allocates space for a U and calls the function with a pointer to this U.

This implies that whenever a function is declared with return type U with U non-simple, it has an additional argument **res** of type **pntr**(U), and returns **void** through its `ret` statement.

Note that it is not possible that the function allocates the U, because it does not know what will be done with the U.

More Type Replacements

Functions are not called with their declared types, but their types replaced in a predictable way.

Here are some more replacements.

- Integral types **int**, **char**, **bool**, etc. are replaced by their corresponding LLVM types **i32**, **i8**, **i1**, etc.
- **const** is ignored.
- **ref** is identified with **pntr**.

Examples of LLVM Types vs Declared Types

Function **fact** is declared as `double fact( int i )`. Its LLVM implementation has type `double fact( i32 %i )`.

Assume that **fact** is also defined on **bigint** as `bigint fact( bigint i )`. Its LLVM implementation has type `void fact( bigint* %res, bigint* %i )`.

Suppose we have a printing operator for **bigint**, declared as `std::ostream& operator << ( std::ostream&, const bigint& )` Its LLVM implementation has type `std::ostream* operator<<( std::ostream* %s, bigint* %b )`.

Suppose that **bigint** has a unary minus function `operator-( ) const`, which is a member function. Its LLVM implementation has type `minus( bigint* res, bigint* this )`.

Translate

We define a function **translate**(t, v), where t is an abstract syntax tree that has been type checked, and v is an LLVM variable.

- If $T(t) =$ **void**, then **translate**(t,v) will create code that does not create any result, and v is ignored.
- If $T(t) \neq$ **void** and $T(t)$ is a simple type, then **translate**(t, v) creates code that assigns the result to v.
- If $T(t) \neq$ **void** and $T(t)$ is a non-simple type, then **translate**(t, v) creates code that writes the result into $*v$.

First Call (Free)

A free occurrence of an expression is an occurrence where the result is ignored.

Let T be the type of t.

1. If T is void, call **translate**(t).
2. If T is simple, produce a fresh variable v, and call **translate**(t, v);
3. If T is non-simple, produce a fresh variable p, emit the statement `%p = alloca T`, and call **translate**(t, p). If T has a non-trivial desctructor, emit `destroy-T( T* %p )`.

First Call (Initializer)

1. If we have $T\ x = t$, and X is simple, then produce a fresh variable v, call **translate**(t, v). After that, emit
   `store T v, T* x`
2. If we have $X\ x = t$, and X is non-simple, then call **translate**(t, x).

First Call (Return)

1. If we have **return** t and t has simple type T, then call **translate**(t, v). After that, emit `ret T v`.
2. If we have **return** t and t has non-simple type T, then call **translate**$(t, \mathbf{res})$, where **res** is the parameter through which the function returns its result.

Translate: Function call $f(t_1, \ldots, t_n)$

Consider **translate**$(f(t_1, \ldots, t_n), v)$:

For each parameter t_i, we recursively generate code that computes the value of t_i. (It is usually done in reverse order.)

At the same time, we create variables v_i that hold (or point to) the values of the t_i.

Translate: Argument t_i of Function Call

- if T_i is simple, let v_i be a new variable of type T_i. Call

  $$\textbf{translate}(t_i, v_i).$$

  The resulting code writes the value of t_i into v_i.

- If T_i is non-simple, let v_i be a new variable of type $\textbf{pntr}(T_i)$. Emit `%v_i = alloca T_i`, and call

  $$\textbf{translate}(t_i, v_i).$$

  The resulting code writes the value of t_i into $*v_i$.

Translate: Function call $f(t_1, \ldots, t_n)$

Let T be the type of $f(t_1, \ldots, t_n)$.

For each parameter t_i of type T_i, if T_i is simple, set $T_i' = T_i$, otherwise set $T_i' = \mathbf{pntr}(T_i)$.

- If $T = \mathbf{void}$, then emit

```
call void f( T'_1 %v_1, ... , T'_n %v_n ).
```

- If T is simple, then emit

```
%v = call T f( T'_1 %v_1, ... , T'_n %v_n ).
```

- If T is non-simple, then emit

```
call void f( %v, T'_1 %v_1, ... , T'_n %v_n ).
```

The first case has no result. The second case writes the result into v. The third case writes the result into $*v$.

Translate: Function call $f(t_1, \ldots, t_n)$ (Cleaning Up)

For every t_i whose type is non-simple and has a non-trivial destructor, it is now time to let this destructor do its job:

```
call void destroy-T_i( T_i* %v_i )
```

Translate: $\text{box}_T(t)$

Consider **translate**$(\text{box}_T(t), v)$.

Let T be the type of t. The type of $\mathbf{box}_T(t)$ must be **ref**(**const**(T)). Let p_i (of type **pntr**(T_i)) and b_i be the variables that were assigned to this occurrence of $\mathbf{box}_{T_i}$ during preparation.

- If T is simple, let w be a new variable of type T_i. Call

  translate(t, w).

  The resulting code writes the value of t into w. Emit
  `store Ti %w, T* %p ; %b = 1; %v = %p.`

- If T_i is non-simple, then call

  translate(t, p).

  The resulting code writes the value t into $*p$. Emit
  `%bi = 1; %v = %p.`

Translation of Constants

Consider **translate**(c, v) for constants c. The type of a constant is always simple.

Just `%v = c`.

Translation of Variables and Function Parameters

We define **translate**(w, v) for variable or function parameter w.

- If the variable w was declared with type T, and T is not a reference type, we emit

  ```
  %v = %w
  ```

- If the variable was declared as **ref**(T) or **ref**(**const**(T)), we emit

  ```
  %v = load T** %w
  ```

Translation of Boolean ? Expression1 : Expression2

We define **translate**(B ? t_1 : t_2, v). Let T be the type of the expression.

The translation is easy in principle, but one has to insert a Φ-function when T is simple.

Translation of ? : (simple)

Let w be a new variable of type `i1`, let L_1, L_2, L_3 be new labels, and let v_1, v_2 be new variables of type T. First call **translate**(B, w). Then emit

```
   br i1 %w, label %L1, label %L2
; <label>:L1
```

Call **translate**(t_1, v_1). Emit

```
   br label %L3
; <label>:L2
```

Call **translate**(t_2, v_2). Emit

```
   br label %L3
; <label>:L3
   %v = phi T [ %v1, %L1 ], [ %v2, %L2 ]
```

Translation of ? : (non-simple)

Since everything takes place in memory, there is no need for Φ-functions.

First call **translate**(B, v). Then emit

```
   br i1 %w, label %L1, label %L2
; <label>:L1
```

Call **translate**(t_1, v). Emit

```
   br label %L3
; <label>:L2
```

Call **translate**(t_2, v). Emit

```
   br label %L3
; <label>:L3
```

Cleaning Up Temporaries

In LLVM, allocated variables are deallocated automatically, but the destructors still must be called.

For every occurrence of $\mathbf{box}_T$ whose type T has a destructor, emit

```
   br i1 %b, label %L, label %M
; <label>:L
   call void destroy-T( T* %p )
   br label %M
; <label>:M
```

Here `%p,%b` are the variables that were assigned to $\mathbf{box}_T$.

Some Standard Functions

In order to compile C/C^{++} programs, one needs a library of implementations of the standard functions.

We give the main ones on the following slides.

Load Operators

For a primitive type T, a load operator has a single parameter of type **ref**(**const**(T)).

Its result has type T. Its implementation in LLVM is

```
define T loadT( T* %p ) {
   %v = load T* p
   ret T %v
}
```

Copy Constructors for Simple Types

For a non-primitive, but simple type T, the copy constructor $\mathbf{cc}_T$ has a single parameter of type $\mathbf{ref}(\mathbf{const}(T))$. Its result has type T. The implementation is the similar to the load operator on the previous slide.

Copy Constructors for Non-Simple Types

If the copy constructor is not redefined by the user, the default copy constructor is used, which has a single parameter of type **ref**(**const**(T)) and produces a T.

Otherwise, the parameter has type **ref**(T) or **ref**(**const**(T)), dependent on how the user defined it.

The implementation (in CLANG) seems to be:

```
define void copyT( T* %res, T* %p ) {
   %a1 = getelementpntr T* %p, i32 0, i32 0
   %b1 = getelementpntr T* %res, i32 0, i32 0
   %v1 = load T1* %a1
   store T1 %v1, T1* %b1
   ...
   %an = getelementpntr T* %p, i32 0, i32 n
   %bn = getelementpntr T* %res, i32 0, i32 n
   %vn = load Tn* %an
   store Tn %vn, T1* %bn
   ret void
}
```

(Each field is copied separately.)

Assignment Operators for Simple Types

Simple types have assignments that copy their data and don't care about what they overwrite.

A simple assignment operator has two input parameters with types **ref**(T) and T. The result is of type T.

The implementation is:

```
define T assignT( T* %p, T %v ) {
   store T %v, T* %p
   ret T %v
}
```

Assignment Operators for Non-Simple Types

We assume that the assignment operator is not redefined. It has two input parameters with types **ref**(T) and T. Its output parameter has type T.

Because T is non-simple, the implementation receives its second argument as pointer, and returns its value also through a pointer.

The implementation is similar to the implementation of the copy constructor for non-simple types, (but an additional copy is made.)

It may be better to have assignments return **ref**(**const**(T)) by default.

Simple Arithmetic Operators

Let $\star$ be one of the standard binary arithmetic (or comparison) operators. Let T_1, T_2 be the input types (always simple), and let U be the (simple) output type.

An implementation could have the following form:

```
define iU operatorX( iT1 %v1, iT2 %v2 ) {
   ...
   %v =  ... (code that does the calculation).
   ret %v
}
```

Adding and Subtracting from Pointers

The first parameter has type **pntr**(T) or **pntr**(**const**(T)). The second type has type I for some integral type.

The return type is equal to the first parameter. The implementations in LLVM are:

```
define T* pntr_plus( T* %p, i32 %i ) {
   %p1 = getelementptr T* %p, i32 %i
   ret T* %p1
}

define T* pntr_minus( T* %p, i32 %i ) {
   %i1 = sub i32 0, i32 i
   %p1 = getelementptr T* %p, i32 %i1
   ret T* %p1
}
```

Subtraction Between Pointers

There are two parameters, of type **pntr**(T) or **pntr**(**const**(T)). We assume that the result is of type `i64`.

```
define i64 pntr_minus( T* %p1, T* %p2 ) {
   %q1 = ptrtoint %T* %p1 to i64
   %q2 = ptrtoint %T* %p2 to i64
   %i = isub i64 %q1, q2%
   %res = sdiv exact i64 %i, i64 sizeof(T);
   ret i64 %res
}
```

$*$ and &

Operators $*$ and & may appear difficult to understand, but they are just casts that cast a pointer to a reference, or reverse.

For $*$, the input parameter has type **pntr**(T) and the result has type **ref**(T), or the input parameter has type **pntr**(**const**(T)) and the result has type **ref**(**const**(T)).

```
define T* starT( T* %p ) {
   ret %p
}
```

For &, the input parameter has type **ref**(T) and the result has type **pntr**(T), or the input parameter has type **ref**(**const**(T)) and the result has type **pntr**(**const**(T)).

```
define T* ampersandT( T* %p ) {
   ret %p
}
```

Prefix Increment/Decrement

We discuss only preincrement. Predecrement is similar.

The prefix operators have one input parameter of type **ref**(T) and a result type **ref**(T).

For integral types, the implementation is:

```
define prefpluplus( T* %p ) {
   %v1 = load T* %p;
   %v2 = add T %v1, i32 1
   store T %v2, T* %p
   ret %v1
}
```

If T has form `U*`, then `add T %v1, i32 1` must be replaced by `%v2 = getelementptr U* %v1, i32 1`.

Postfix Increment/Decrement

The postfix operators have input type **ref**(T) and result type T.

For integral types, the implementation is:

```
define postfplusplus( T* %p ) {
   %v = load T* %p
   %v1 = add T v, i32 1
   store T %v1, T* %p
   ret T %v1
}
```

If T has form `U*`, then `add T v, i32 1` must be replaced by `%v1 = getelementptr U* %v, i32 1`.

If there are increment/decrement operators on non-simple types, they are always user defined.

Indexing Operators

We define array indexing operators. One can also view `p[i]` as abbreviation for `*(p+i)`.

Input types are **pntr**(T) and I for some integral type I, output type is **ref**(T).

```
  define T* index( T* %p, i32 %i ) {
  %p1 = getelementptr T* %p, i32 %i
  ret %p1
}
```

Field Selection Functions

Field selection functions have input type **ref**(S) or **ref**(**const**(S)), and return type **ref**(F) or **ref**(**const**(F)).

The implementations are easy:

```
define F* field_f( S* %p ) {
%p1 = getelementptr T* %p, i32 0, i32 %j
   ; j is relative position of field f in struct.
ret F* %p1
}
```

(Fields can also have reference type. In that case, an additional `load` ins necessary.)

Field Selection Functions (2)

For simple types, one could also consider field functions from type S to F. (CLANG seems to use them.)

Implementation would be:

```
define F field_f( { F1, ..., Fn } %s {
%v = extractvalue { F1, ..., Fn } %s, j
   ; j is relative position of field f in struct.
ret F %v
}
```

Example with Non-Simple Functions

Using the declarations:

$$\begin{array}{ll} a & A \\ f & \mathbf{func}(A;\ \mathbf{ref}(\mathbf{const}(A)), A\ ) \\ g & \mathbf{func}(A;\ A) \\ h & \mathbf{func}(\mathbf{ref}(A);\ \mathbf{ref}(A)\ ) \end{array}$$

Overload resolution on $f(g(a), h(a))$ results in

$$f(\ \mathbf{box}_A(g(\ \mathbf{cc}_A(a)\ )),\ h(a)\ ).$$

Example with Non-Simple Functions (2)

```
   %pt = alloc A ; For temporary.
   %bt = i1 1

   %p = alloc A ; For vacuous result.

   { translate( f( ... ), p ) }

   call destroy-A %p ; If there is one.

   br %bt, label %L1, label %L2
; <label>:L1
   call destroy-A %pt; If there is one.
   br label %L2
; <label>:L2
```

Example

This is a possible implementation of **strcpy**:

```
void strcopy( char* p, const char* q )
{
   size_t i = 0;
   while( q[i] )
   {
      p[i] = q[i];
      i ++ ;
   }
   p[i] = 0;
}
```

Length of a C-string

```
size_t length( char* s )
{
   size_t length = 0;
   while( *( s++ ))
      ++ length;
   return length;
}
```

Example with Arrays

Consider the program

```
void fill( int* p, size_t n )
{
   for( size_t i = 0; i < n; ++ i )
      p[i] = i * i;
}

main( )
{
   int squares[5];
   fill( squares, 5 );
   return 0;
}
```

Inlining

If you try a few examples, you quickly see that the result of **translate** is horribly inefficent, and also terrible to look at.

The code consists of a sequence of calls to one/two line functions.

Calls of short functions can be substituted away.

This is called inlining.

Inlining

If the function returns a result, it must be written in such a way that it has only one `ret` statement.

1. Replace the function parameters by the arguments of the call.
2. If the function returns a value, then replace the returned variable, by the variable to which the result is assigned in the calling environment.
3. Rename the remaining variables, and branch labels, to avoid conflicts.
4. Replace the call by the body of the function.

Of course, **translate** can be modified to do these replacements directly, but it makes the algorithm more complicated and less modular.

Remaining Topics

- I did not say much about inheritance. It doesn't have impact on the compilation algorithm, as far as I can see. It makes method definitions more complicated.
- Of course, it would be nice to verify the correctness of the translation algorithm, but againt what? Any specification wouldn't be much simpler than what I wrote, and the whole idea of verification is based on the assumption that specifications are simpler than implementations.