Intermediate Code Generation

Memory Allocation

For a type T, **sizeof**(T) is the space that is required to store an object of type T in memory.

For primitive types, typical values are

sizeof(**bool**) $= 1$, **sizeof**(**char**) $= 1$,
sizeof(**int**) $= 4$, **sizeof**(**unsigned int**) $= 4$,
sizeof(**float**) $= 4$, **sizeof**(**double**) $= 8$.

Alignment

In some architectures, not every object can start on every address in memory, or at least not with the same efficiency.

In 32-bit architectures, a 32-bit word consists of four bytes, each of which has a separate address. If an **int** is positioned at an address that is a multiple of four, then it can be loaded in one read cycle. If its address is not a multiple of four, then two cycles are required.

In the same way, reading/writing a **double** can take 2 or 3 cycles.

Alignment

One can put every object on the largest possible alignment but that is inefficient. For example the array `char p [10]` would then take 40 bytes instead of 10.

Sometimes, it is possible to save by space rearranging variables, but I am not sure if there exist compilers that do that:

```
char c1;   int i1;   char c2;   int i2;
   // 16 bytes because i1 and i2 must be aligned.
char c1;   char c2;  int i1;    int i2;
   // 12 bytes.
```

Alignment seems pretty complicated, and I do not know a general method. I will ignore alignment in the rest of the slides.

Arrays

Consider the code fragment:

```
#define SIZE 100
int p [ SIZE ];

for( unsigned int i = 0; i < SIZE; ++ i )
   p [i] = i * i;
```

Arrays (2)

Dealing with arrays is easy:

```
T p[ N ];
p [ i ];
```

Define

$$\text{sizeof}(p[N]) = N.\text{sizeof}(T).$$

If p is of type `T[]&`, then `p[i]` is of type `T&`. It can be computed by

$$p[i] = p + i \times \text{sizeof}(T).$$

(Remember that p has a reference type.)

Structs

Consider:

```
struct S { T1 t1; ... ; Tn tn };
```

We have seen before, that there are good reasons to treat S like a primitive type. If a struct is used one time in the code, without name, one should generate a name for it.

Define:

$$\text{sizeof}(S) = \text{sizeof}(T_1) + \cdots + \text{sizeof}(T_n).$$

This calculation needs to be made only once when S is declared.

Field Functions

In order to access the fields of a struct, we will assume that each **struct** type has field functions.

Intuitively, a field function is a function that adds the offset of the field inside the struct to a reference to the struct (its address), and changes its type.

For example

```
struct S
{
   int node;
   char c;
   S* next;
};
```

would have the field functions shown on the next slide.

Field Functions (2)

```
int& S_node( S& s ) { return s; }
const int& S_node( const S& s ) { return s; }

char& S_c( S& s ) { return s + 4; }
const char& S_c( const S& s ) { return s + 4 };

S* & S_next( S& s ) { return s + 5; }
S* const& S_next( const S& s ) { return s + 5; }
```

The parser should replace expression of form $f.a$ by field_a(f), and the type checker should replace this by S_a(f) for the proper **struct** type S.

Field Functions (3)

In case the **struct** has **static** fields, the field functions have to be modified to return a global variable.

```
struct S
{
   static int x;
}

int& S_x( S& s ) { return &stat_x }
const int& S_x( const S& s ) { return &stat_x; }
```

We assume **stat_x** is a global variable in which field S::x is stored. In the rest of these slides, we assume that every **struct** type has the associated field functions defined, and that the parser always replaces field selection by a generic field function.

Field Functions (4)

There are some subtleties involving references: If a field already is a reference, then it is not made into a double reference:

```
struct S
{
   int& x;
};

S.x = 3; // shouldn't change x itself, but the
         // int that x refers to.

int& x S_x( S& s ) { return s; }
int& x S_x( const S& s ) { return s; }
```

Static Member Functions

For functions, static means something different than for member fields.

All member functions are static in the sense that they do not belong to any object. Different objects of the same type have the same member functions.

For a member function, static means that the function can only adress static variables of the struct.

Usage of 'this'

```
class A
{
   f( B b, C c );
   g( B b, C c ) const;
}
```

can be replaced by

```
A::f( A* this, B b, C c );
B::g( const A* this, B b, C c );
```

In the call, `a.f(b,c)` and `a.g(b,c)` can be replaced by `f(&a,b,c)` and `g(&a,b,c)`.

The C-language

The *C*-language has evolved from an almost unchecked language into a strongly typed language.

Type System of C/C^{++}

We recursively define the type system of C/C^{++}.

- We assume the primitive types given before: **bool**, **char**, **int**, **unsigned int**, **float**, **double**.
- If T is a type, then **const**(T) is a type.
- If T is a type, then **raw**(T) is a type.

The aim of **raw**(T) is to represent the type of unitialized memory. Unitialized memory is the space occupied by an object of type T before a constructor is called, or after the destructor was called.

Type System of C/C^{++}

- If T is a type, then **pointer**(T) is a type.
- If T is a type, then **ref**(T) is a type.
- If T is a type, and n is a natural number, then **array**(n, T) is a type.

The aim of **ref**(T) is to represent reference types of form $T\&$. The type **const** $T\&$ is represented by **ref**(**const**(T)).

We do not distinguish arrays with unknown size from pointers. (Like `char* p` and `char p[ ]`) As far as I know, they are the same in C/C^{++}. It they turn out different, then another type constructor for arrays with unknown size needs to be added.

Type System of C/C^{++}

- If T is a type, then $\mathbf{reg}(T)$ is a type.

The aim of $\mathbf{reg}(T)$ is to represent the fact that T is stored in a register variable.

These are not the registers of the processor, but local variables in the intermediate representation.

Type System of C/C^{++}

- If v is an identifier that represents the name of a **struct** or a **class** type, then v is a type. We treat v like a primitive type!
- If v is an identifier, that is the name of an **enum** type, then v is a type. We treat v like a primitive type!
- If U and $T_1, \ldots, T_n$ are types, then $\mathbf{func}(U; T_1, \ldots, T_n)$ is also a type.

We assume that for each **struct**, the field functions are defined, (see a few slides back what these are). In case a **struct** is introduced without name, then we assume that it has a random name.

$\mathbf{func}(U; T_1, \ldots, T_n)$ is the type of functions from $T_1 \times \cdots \times T_n$ to U.

Restrictions on Types

There are couple of restrictions on the possible types:

Types of form **const**(**const**(T)) are not allowed.

Types of form **const**(**ref**(T)) are not allowed.

Types of form **reg**(T) cannot be used to construct further types. T must be a primitive type. (Not struct.)

Examples of C-Types

```
char p[100], *q, **r;
   p : array(100,char),
   q : pointer(char),
   r : pointer(pointer(char))

char r[5][100];
   r : array( 5, array( 100, char ))

const int& x;
   x : ref( const( int ))

const int& f( const int& , int );
   f : func( ref( const( int ));
             ref( const( int )), int )
```

Translation into Stack Machine

I first explain how translation into a stack machine works, then I explain that a stack machine is not sufficient for compilation of C^{++}.

Expressions can be transformed into reverse Polish notation.

A function with arity n can be replaced by a code fragment that takes n-objects from the stack, and puts one object back on the stack.

Stack Machine (2)

For example, the expression $x := x + 4 * y$ can be replaced by $x\ 4\ y\ *\ +\ (:= x)$.

```
push( x ); // Push value of x on the stack.
push( 4 ); // Push 4 on the stack.
push( y ); // Push value of y on the stack.
* // Take two numbers from top of stack,
  // multiply them, and push result back.
+ // Take two numbers from top of stack,
  // multiply them, and push result back.
write(x);  // Take number from top of stack,
           // and write it into variable x.
```

Stack Machine (3)

```
unsigned int fact( unsigned int x )
{
   if( x == 0 )
      return 1;
   else
   {
      f = fact( x - 1 );
      return x * f;
   }
}
```

Stack Machine (4)

Local variables (x and f on the previous slide) can be easily maintained on the stack.

When a local variable goes out of scope, it can be easily removed by increasing the stack pointer.

The defined function **fact** can have the same interface as built-in functions. It removes top of stack, and replaces it by the result. In this way, defined functions can be mixed with built-in functions in expressions.

We use **stack**[i] (with $i \geq 0$) to refer to the object that occurs at depth i on the stack. Top of stack is **stack**[0].

```
// Initially, we assume that local variable x is
// on top of the stack.

push stack[0] // We evaluate (x==0).
push 0
== // Takes two numbers from top of stack, compares
   // them, and puts result back on stack.
iffalse goto L1;
   // Remove one boolean from top of stack, and jump
   // if boolean is true.

push 1;   // We prepare to return 1.
stack[1] = stack[0];
       // We overwrite local variable x with result,
pop;   // and correct the size of the stack.
return;
```

```
L1:
   push stack[0];
   push 1;
   - ;          // We calculated (x-1).
   call fact; // Replaces x-1 by fact(x-1);

   // We create variable f, and initialize with fact(x-1).
   // Since fact(x-1) is already on the top of the stack,
   // we need to do nothing.

   push stack[1];  // Variable x;
   push stack[1];  // Variable f;
   *               // Now we have x*f on top of stack.
   stack[2] = stack[0];
   pop; pop; return;
      // Remove local variables, and return.
```

Problems with Stack Machines and C^{++}

Unfortunately, it is not possible to evaluate C^{++} with a simple stack model, because of the following two reasons:

1. Nearly every return from a function involves copying back the result. (Look at the previous slide.) This is inefficient for big objects. Moreover, objects are not required to have a copy or move constructor in C^{++}. Such objects cannot be copied, but still it should be possible to write a function that returns such an object.

2. In an expression of form $f(g(h(\ )))$, the function h can construct an object, which is passed as reference to g. Function g can decide to further pass the reference to f. This implies that the result of h has to be preserved all through the evaluation of $f(g(h(\ )))$, so that an evaluator based on a pure stack machine, is not possible.

Problems with Stack Machines and C^{++} (2)

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

A sum( const A& a1, const A& a2 )
{
   A res = a1;
   res. add( a2 );
   return res;
}
```

Problems with Stack Machines and C^{++} (3)

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";
```

It is clear that we need a more sophisticated evaluation model for expressions.

The good news is that local variables still can be kept in a stack-like fashion.

I will assume that the stack always grows downwards, because it more natural to look forward, than to look backwards. When arguments of a function are pushed onto the stack in reverse order, they will be in the right order on the stack.

Intermediate Representation

I will introduce an intermediate representation, on which it is still possible to do code optimization, but from which it is also easy to generate executable code.

- We assume a stack based memory model. Local variables are stored on the stack. Parameters to unknown functions (that are not inline) are also stored in memory.
- We don't store intermediate results in memory. Storing in memory causes an aliasing problem, which makes optimization difficult.

Intermediate Representation (2)

We assume an infinite set of register variables. The register variables have either one of the primitive types **bool**, **char**, **int**, **unsigned int**, **float**, **double**, or a **pointer/reference** type. **struct** types are not stored in register variables.

The name of a register variables starts with `@`, when it is not of **pointer** or **reference** type.

The name of a register variable with **pointer** or **reference** variable starts with `$`.

In principle, one should include the primitive type of a register in the name also, but we assume that the primitive type will be clear from the context.

Intermediate Representation (3)

We assume that for every primitive type, there are instructions of the form below, assigning a constant to a register variable:

```
@i = 4;
@pi = 3.1415926535;
@c = 'h';

@i = sizeof(T);   // It's also a constant.
```

Intermediate Representation (4)

We assume that for every primitive operation $\star$ on primitive types, there is an instruction of form

```
@i1 = @i2 * @i3;
```

I also assume that there are conversion functions between the primitive types.

```
@f1 = float(@i3);    // Convert integer into float.
```

I will not list all instructions, but it is clear that such a list can be constructed, and that it is not too big.

Usage of Pointer Registers

Pointers can be used for writing and reading from memory:

```
[ $P ] = @i;  // Write @i into address determined by
              // $P.
@f = [ $Q ];  // Write content of address, determined
              // by $Q into @f.
$P = [ $Q ];  // Pointers by themselves can also be
              // written and read.
```

And of course, there is also pointer arithmetic:

```
$P = $Q + @i; // Pointers to different types can
              // be freely mixed.
$Q = $P;
```

Creation and Deletion of Variables

Local variables are created in memory, in stack like fashion. The stack grows downwards.

```
pushvar $P, T // Create a local variable of type
              // T on stack, and make $P point
              // to it.

popvar T      // Remove the last pushed local variable
              // which must have type T, from stack.

staticvar $P, T, name.
              // Create or refer to a static variable
              // of type T with the given name.
```

The functions `new` and `delete` are library functions, that have no instructions in intermediate representation.

Creation and Deletion of Variables (2)

Actually, it works a bit different: There are three special register variables, `$SP` (Stack Pointer), `$SBP` (Static Base Pointer) and `$SEP` (Static End Pointer).

Static memory is in the interval `[ $SBP .. $SEP )`. Dynamic memory is above `$SP`.

`pushvar $P, T` is an abbreviation for

```
$P = $SP; $SP = $SP - sizeof(T);
```

Similarly, `popvar T` is an abbreviation for
`$SP = $SP + sizeof(T)`.

Linking replaces a command of form `staticvar $P,T, name` by `$P = $SBP + X`, where `X` is the position of `name` in static memory.

Control Instructions

```
goto L;

iftrue @B goto L;
iffalse @B goto L;
   // Jump if boolean is true or false.

call Fname;   call [ $P ];
   // Call a function. The return address is pushed
   // on the stack. (The same stack stack that is
   // used for local variables.)

return;
   // Return from a function.
   // This means that we pull an address from the
   // top of the stack, and jump to it.
```

Inline and Outline Functions

There are two types of functions, inline and outline functions, which are compiled in different ways.

Inline functions are not called. Instead, the definition of the function is inserted whenever the function is called. This is more efficient, but it makes the resulting code longer. In addition, an inline function can never be recursive. One needs a couple of inline functions as building blocks for the translation.

Outline functions are functions that are really called. 'Outline' is not a proper English term, but I find it convenient to use it.

Outline functions have to be treated like a black box. The parameters are put in memory, together with a pointer where the result should be put. After that the function is called, and it assumed that it writes its result at the indicated place.

Calling Convention for Outline Functions

Suppose that we want to call function f of type **func**$(T;\ T_1, \ldots, T_n)$ with arguments $t_1, \ldots, t_n$.

1. First decide where the result will be written. If necessary, create a raw(T) on the stack.
2. For each t_i, create code that pushes the result (which will have type T_i) on the stack.
3. Push a pointer to the result (the memory allocated in step 1) on the stack.
4. Call f. We assume that the code of f writes the result and cleans up all local variables that were used inside the function body.
5. Clean up the parameters (created in step 2).

Calling Convention for Outline Functions (2)

The reasons for passing the pointer to the result (instead of letting the function decide where it puts its result, or using a purely stack-based order) are as follows:

1. Constructors cannot choose the location, into which the object has to be constructed, by themselves. For example, in `X* p = new X(t)`, the memory manager determines the address and the constructor has to put the `X` where `new` decided.

2. If somewhere, in a block with local variables, a statement of form `return X(t)` is encountered, the function cannot create the `X`, and after that clean up the local variables.

   This would force us to either keep the local variables, or to move the `X`. Keeping local variables is inefficient. Moving is also inefficient, or impossible when `X` has no copy constructor.

How an Outline Function is Called

A				
B				t_n
C				t_n
D				$t_1, \ldots, t_n$
E				$t_1, \ldots, t_n$
F			& result	$t_1, \ldots, t_n$
G		return address	& result	$t_1, \ldots, t_n$
H	local variables	return address	& result	$t_1, \ldots, t_n$
I	local variables	return address	& result	$t_1, \ldots, t_n$
J		return address	& result	$t_1, \ldots, t_n$
K			& result	$t_1, \ldots, t_n$
L				

How an Outline Function is Called (2)

The figure on the previous slide explains what happens when $f(t_1, \ldots, t_n)$ is called.

A This is the situation before we start preparing the call. We have to decide where in memory the result will be written. (In reality, it is already decided by the context.)

B We create a space for t_n on the stack. The color red means that the space is created, but no meaninful value has been written. (Raw data.)

C We evaluate t_n, specifying that the result should be written on the position of t_n. Since t_n has a value now, it is not red anymore.

D,E We do this for all of the parameters.

How an Outline Function is Called (3)

The figure on the previous slide explains what happens when $f(t_1, \ldots, t_n)$ is called.

F In/Before **A**, it was decided on which memory location, the result of $f(t_1, \ldots, t_n)$ will be written. We now push the address of this memory location on the stack. Since the result is not written yet, it is the address of something red.

G We call function f, which pushes the return address on the stack.

H We are now inside function f. The code of f may create local variables and other data, which can further extend the stack downwards.

I At point I, the code of f reaches a **return** statement. Since we know & result, we know where it has to be written. After writing, result becomes black.

J If there are local variables, we clean them up.

K We leave the code of f, by taking the return address from the stack, and jumping to it.

L We are not inside f anymore. We clean up the pointer to the result and the parameters. The result itself is stored somewhere else, and it is not deleted of course.

Inline Functions

Inline functions are functions that are not called, but substituted away. Whenever the compiler encounters a call $f(t_1, \ldots, t_n)$ of an inline function f, it looks up the code of the definition of f, and replaces the call $f(t_1, \ldots, t_n)$ by its definition.

Inline functions are more efficient than outline functions, because

1. Passing of parameters can take places in register variables, which can be accessed more efficiently than memory.
2. There is no need to jump, to store a return address, and to return.
3. Because the compiler has access to the resulting code, and code using registers can be more easily analyzed than code involving memory, the compiler has a better chance of optimizing it.

Inline Functions (2)

The body of an inline function f of type func$(T;\ T_1, \ldots, T_n)$ is a block of code of form $C[R; S_1, \ldots, S_n; U_1, \ldots, U_m]$, where

- R is the return register. If T has form $\mathbf{reg}(T')$, then the type of register R equals T'. Otherwise, the type of R equals $\mathbf{ref}(T)$.
- $S_1, \ldots, S_n$ are the input registers. For each of the input registers holds: If T_i has form $\mathbf{reg}(T_i')$, then the type of S_i equals T_i'. Otherwise, the type of S_i equals $\mathbf{ref}(T_i)$.
- $U_1, \ldots, U_m$ are the remaining registers. They are the local variables of the function. They can be of any type.

(C is a block of code, with parameters $R, S_1, \ldots, S_n, U_1, \ldots, U_m$ that will instantiated at the moment of compilation.)

Compilation of Inline Functions

Suppose that we want to substitute an inline function call $f(t_1, \ldots, t_n)$, and that f has type $\mathbf{func}(T; T_1, \ldots, T_n)$.

1. If T is not a register type, then decide where in memory the result will be written. Reserve a pointer variable for the result and make it point to a $\mathbf{raw}(T)$. If T is a register type, then reserve a register for the result itself.
2. For each argument t_i do: If T_i has a register type, then reserve a register for t_i and create code that writes t_i into the assigned register. If T_i does not have a register type, then create code that pushes the result of t_i onto the stack.
3. Substitute the definition of f by replacing the local registers in the body of f by ehe register variables that were selected in step 1 and 2. The body of f should clean up all local variables.
4. Clean up the parameters that are in memory.

Compilation of Inline Functions (2)

Assume that the inline function has type func$(U; T_1, \ldots, T_n)$. Let R be its output register. Let $S_1, \ldots, S_n$ be the input registers.

If type U has form reg(U'), then R has type U'. Otherwise, U has type ref(U).

The same applies to $T_1, \ldots, T_n$ and $S_1, \ldots, S_n$: If type T_i has form reg(T_i'), then S_i has type T_i'. Otherwise, S_i has type ref(T_i).

Compilation of Outline Functions

The first task of the outline functions is to assign registers to the parameters, so that they will be available in the body as variables. If the function has n parameters with types $T_1, \ldots, T_n$, then it starts with a sequence:

```
$T1 = $SP + d1; // d1 is position of parameter T1.
$T2 = $SP + d2; // d2 is position of parameter T2.
...
$Tn = $SP + dn; // dn is position of parameter Tn.
```

Each `$Ti` has type **ref**(T_i). In addition, we have

```
$R' = [ $SP ] + d; // d is position of return variable.
$R = [ $R' ];      // Result can be written into [R].
```

Compilation of Constants and Variables

A constant of primitive type T is always considered to have type $\mathbf{reg}(T)$. Non-primitive types have no constants.

A variable of type T has type $\mathbf{reg}(\mathbf{ref}(T))$, when it occurs in an expression. In the code, the variable will be represented by a register of type $\mathbf{ref}(T)$. Local variables are created by
`pushvar $P, T`.

Static variables are loaded into a pointer register by
`staticvar $P, T, name`, when they are needed.

In all cases, the resulting pointer register has type $\mathbf{ref}(T)$, where T is the type of the variable.

String constants are considered static variables of type $\mathbf{array}(n, \mathbf{const}(\mathbf{char}))$.

Conversions

Conversions are functions that change the type of a term, and which the user doesn't have to insert in the term. Sometimes the user has to provide definitions. Conversions can be both inline and outline.

- A copy constructor of type T is a function of type **func**(T; **ref**(**const**(T))) or **func**(T; **ref**(T)).

  User defined types can have default copy constructors, or copy constructors written by the user.

- All primitive types, all pointer, and all reference types have inline copy constructors, which always use registers. They have type **func**(**reg**(T); **reg**(**ref**(**const**(T)))).

Conversions (2)

- A box function of type T is a function that changes a value into a reference. The type of a box function has form **func**(**reg**(**ref**(**const**(T))); T). Every type has a box function.

  Box functions have no actual definition, but they are used as an intermediate step.

- A read function is an inline function that changes a memory variable into a register variable. Read functions are defined for primitive types, and for reference and pointer types. The type has form: **func**(**reg**(T); T).

- A write function is an inline function that changes a register variable into a memory variable. Store functions are defined only for primitive types, and for reference and pointer types. The type of a write function has form: **func**(T; **reg**(T)).

Conversions (3)

- Arrays have no copy constructors, but instead are converted into pointers. For this we have array2pointer functions, which are inline, and which have type

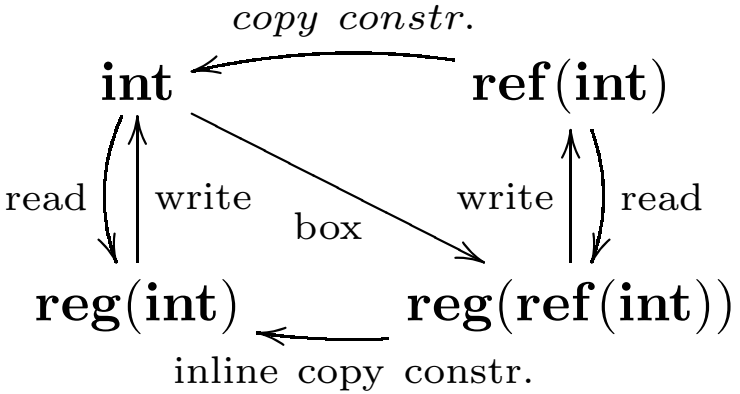
$$\mathbf{func}(\ \mathbf{reg}(\mathbf{pointer}(T));\ \mathbf{reg}(\mathbf{ref}(\mathbf{array}(n, T)))\ )$$

- There are the usual conversions between primitive types: They have (for example) types $\mathbf{func}(\mathbf{reg}(\mathbf{float});\ \mathbf{reg}(\mathbf{int}))$, $\mathbf{func}(\mathbf{reg}(\mathbf{int});\ \mathbf{reg}(\mathbf{char}))$.

- In addition, there can be user defined conversions between user defined types.

Graph of Basic Conversions for Primitive Types

The graph shows the possible conversions for primitive type **int**. Changing a reference into an object is done by a **copy constructor**. Changing register to non-register is done by a **write function**. The **write function** on the left is the **write function** of **int**. The **write function** on the right belongs to **ref(int)**. Changing non-register to register is done by a **read function**. The **read function** to the left belongs to **int**. The **read function** to the right belongs to **ref(int)**.



Translation Function

We now define the translation function **translate**(V, t). V is the name of a register variable, and t is the term that will be compiled. The term t should not contain box functions. We will deal with those later.

- If the type of t equals T and T is not a register type, then V must be a register of type **ref**(T). The call of **translate**(V, t) will generate code, that writes the result of t into the memory position specified by V.
- If the type of t has form **reg**(T), then V must be a register variable of type T. The call of **translate**(V, t) will generate code that writes the result of t into register V.

Translation Function

We use `<<` for emitting statements. So `<< call f` means that we generate the statement `call f`.

In the algorithm, we use the notation `t[i]` instead of t_i for the i-th subterm.

We also write `T[i]` instead of T_i for its type.

Translation Function

Suppose that we call **translate**$(V, a + 1)$. If the type of $+$ equals **func**(**int**; **int**, **int**), then V must have type **ref**(**int**), and the generated code will have form:

```
....
[ V ] = the sum of a and 1.
```

This implies that `V` must be initialized before with a proper address.

If the type of $+$ equals **func**(**reg**(**int**); **int**, **int**)), then V must have type **int**, and the generated code will have form:

```
...
V = the sum of and 1.
```

If f is an outline function, then **translate**$(V,\ f(t_1, \ldots, t_n)\ )$ is defined as follows:

```
for( i = n; i != 0; -- i )
{
   Let T[i] be type of t[i].
   Let V[i] be a new pointer register of type
      ref( T[i] ).

   << pushvar V[i], T[i];
      // The notation << means that we write the
      // instruction into the output.

   translate( V[i], t[i] );
}
```

```
Let T be the type of f( t[1], ..., t[n] ).
Let W be a new pointer register of type
   ref( ref( T )).
<< pushvar V, ref(T);
<< [ V ] = W;
<< call f;
<< popvar ref(T);
for( i = n; i != 0; i -- )
   << popvar T[i];
```

(For the moment, we ignore destructor calls. We come back to this.)

For the case where f is an inline function,
translate$(V, f(t_1, \ldots, t_n))$ is defined on the next slide.

Inline functions are bit more complicated, because of the mixing of register and non-register parameters.

```
for( i = n; i != 0; -- i )
{
   Let T[i] be the type of t[i].
   If T[i] has form reg( T'[i] ) then
   {
      let V[i] be a new register variable of type T'[i]
      translate( V[i], t[i] );
   }
   else
   {
      let V[i] be a new pointer variable
                              of type ref(T[i]);
      << pushvar V[i], T[i];
      translate( V[i], T[i] );
   }
}
```

```
Remember that the definition of F has form
that P( R, S[1], ..., S[n], U[1], ..., U[m] ).

Let F[1], ..., F[m] be a sequence of unused register
                    variables.

<< P [ R := V; S[1] := V[1], ..., S[n] := V[n];
               U[1] := F[1], ..., U[m] := F[m] );

   // If type of f( t[1], ..., t[n] ) is a
   // register type, then P [ ... ] has now put
   // its result in V. Otherwise, it has
   // written its result into [V].
```

```
// It remains to up parameters that were in memory:

for( i = n; i != 0; -- i )
{
   If T[i] is not of form reg( ... ), then
   << popvar T[i];
}
```

Compiling Constants and Variables

For a constant c, the call of **translate**(V, c) emits the code `V = c`. This is possible because constants are always of type **reg**(T) with T primitive.

For a local variable v of type T, there always exists a pointer variable with type reg(ref(T)) that represents v. Let P be this pointer variable. Then **translate**(V, v) emits the code `V = P`.

For a static variable v, **translate**(V, v) emits the code

```
staticvar V, T, v;
```

Some Examples with **int**

First we try a few inline functions: Between `[` and `]` are the parameters, which will be substituted.

```
int_plus : func( reg( int ); reg( int ), reg( int ))
[ @R0, @R1; @R2; ]  { @R2 = @R0 + @R1; }

int_minus : func( reg( int ); reg( int ), reg( int ))
[ @R0, @R1; @R2 ] { @R2 = @R0 - @R1; }

int_assign: func( reg(ref(int)); reg(ref(int)), reg(int) )
[ $P, @R; $Q; ] { [ $P ] = @R; $Q = $P; }

int_inline_copy: func( reg(int); reg( ref(const(int))) )
[ @R; $P; ]  { @R = [ $P ] }
```

We try to compile the expression `i = j + k`. Assume that `i,j,k` are local variables, represented by pointer registers `$I,$J,$K`.

Result of type checking and adding conversions is:

```
int_assign( i, int_plus( int_inline_copy( j ),
                         int_inline_copy( k ) )).
```

Type checking `i = ( j + 4 )` results in

```
int_assign( i, int_plus( int_inline_copy( j ), 4 )).
```

Example with Outline Function

```
fact : func( int; int, int )
   (Implementation is not visible)

int_read: func( reg(int); int )
[ @R; $P ] { @R = [ $P ]; }

int_write: func( int; reg(int) )
[ $P; @R ] { [ $P ] = @R; }

i = fact( i + 1 )   ==>
int_assign( i,
   int_read(
      fact(
         int_write(
            int_plus( int_inline_copy( i ), 1 )))));
```

Example with Arrays and Field Functions

Consider:

```
struct tt
{
   int a;
   int b;
}
   table [ 100 ];

i = 53;
table [i]. a = table [ i + 1 ]. b;
```

Example with Arrays and Field Functions (2)

We need the field functions:

```
tt_a : func( reg(ref(int));  reg(ref(tt)) )
[ $P; $Q ] { $P = $Q; }

tt_b : func( reg(ref(int));  reg(ref(tt)) )
[ $P; $Q ] { $P = $Q + sizeof(int); }
```

and the functions on the next slide:

```
tt_array2pntr : func( reg(pointer(tt));
                      reg(ref(array(tt))) )
[ $P; $Q ] { $P = $Q; }
      // Because it is only a type conversion.

tt_pntr_plus : func( reg(pointer(tt));
                     reg(pointer(tt)), reg(int))
[ $P; $Q, @I; @J ]
   { @J = sizeof(tt) * @I;
     @P = $Q + @J;
   }
      // Finally an example with a local variable!

tt_star : func( reg(ref(tt)); reg(pointer(tt)) )
[ $P; $Q ] { $P = $Q; }
      // The * operator. It is only a type conversion.
```

```
table [i]. a = table [ i + 1 ]. b;

// This expression is syntactic sugar for:

field_a( * ( table + i ) ) =
   field_b( * ( table + ( i + 1 )) )
```

The result of type checking and overload resolution is on the next slide.

```
int_assign(
   tt_a( tt_star(
      tt_pntr_plus( tt_array2pntr( table ),
                    int_inline_copy( i ))),
   int_inline_copy( tt_b( tt_star(
      tt_pntr_plus( tt_array2pntr( table ),
                    int_plus( int_inline_copy(i), 1 ))))))
```

We leave the rest of the compilation of this expression as a trivial exercise to the to the interested reader!

Boolean ? Statement1 : Statement2

We explain how to compile expressions like:

```
abs = ( x < 0 ) ? -x : x;
```

```
int fact(int i) { return i==0 ? 1 : i*fact(i-1); }
```

We assume that for primitive type t, **if** has type

$$\mathbf{func}(\mathbf{reg}(t);\ \mathbf{reg}(\mathbf{bool}), \mathbf{reg}(t), \mathbf{reg}(t)).$$

For a non-primitive type t, it has type

$$\mathbf{func}(t;\ \mathbf{reg}(\mathbf{bool}), t, t).$$

translate(V, **if**(b, t_1, t_2)) is defined as follows:

```
Let B be a new register of type bool.

translate( B, b );

let L1,L2 be two new labels.

<< iffalse B goto L1;

translate( V, t1 );
<< goto L2:
<< L1:
translate( V, t2 );
<< L2:
```

Complete Compilation of Expressions

There a couple of topics left:

- After evaluation of an expression, the result is **(1)** either thrown away, **(2)** used in a **return** statement, or **(3)** used in an initialization. In the first case, we way need to create a temporary variable for the result, when it is not a register type.

  Case 2 and 3 are very similar.

- An expression may contain **box**-operators. They need to be replaced by temporary variables.

Compilation of the Box Operator

Suppose that we have a **max** function which selects the greater of two integers. Both of its arguments are reference types:

func(**reg**(**ref**(**int**)); **reg**(**ref**(**int**)), **reg**(**ref**(**int**)).

Assume that we want to compile the expression

```
std::cout << max( 10, i + 1 );
```

In this expression, **max** is applied on **int**s, but it requires **reg**(**ref**(**int**))s. Inserting the conversions results in:

```
max( int_box( int_write( 10 )),
     int_box( int_write(
                   int_plus( int_inline_copy(i), 1 ))));
```

Since it is impossible to determine how long the boxed **int**s are needed, the semantics of C^{++} guarantees that objects in boxes exist until the expression is completely evaluated.

Compilation of the Box Operator (2)

We 'unwind' the expression, and introduce local variables for the boxed variables, until no box operators are left. After that, we compile as before:

```
{
   int b1 = int_write(10);
   int b2 = int_write( int_plus( int_inline_copy(i), 1 ));

   std::cout << max( b1, b2 );
}
```

Complete Compilation

Compilation proceeds in two stages: We first construct a sequence of `pushvars`, `translates` and `popvars`. When this sequence is complete, we expand the `translates` as described before.

The sequence is obtained by iteration:

If we want to compile **return** t, then let R be the register representing the return variable of the function. Start with:

```
fulltranslate( R, t );
   // Will write result into [R].

(clean up local variables of the function)
return;
```

Complete Compilation (2)

If we want to compile t and the result is thrown away, then let T be the type of t. If T is not of form $\mathbf{reg}(T')$, then start with:

```
<< pushvar V, T;
fulltranslate( V, t );
<< popvar V;
```

otherwise, start with

```
fulltranslate( V, t );
```

Complete Compilation (3)

If we want to compile $T\ v = t$, (initialization of a variable v with t, and t has type T, then start with:

```
<< pushvar V, T;
   // Now register V represents variable v.
fulltranslate( V, t );
```

T cannot be of form $\mathbf{reg}(T')$.

We now have a sequence of `pushvars`, `popvars`, and `fulltranslates`. On the next slides, we will replace the Box-operators by local variables.

Complete Compilation (4)

As long as there are occurrences of `fulltranslate(V,t)`, proceed as follows:

If t contains no box operators, then replace `fulltranslate(V,t)` by `translate(V,t)`.

Otherwise, write t in the form $t[\ \mathrm{box}(t')\ ]$, where box is an outermost occurrence of a box operator. Let T be the type of t'. Type T is certainly not a register type. Replace `fulltranslate(V,t)` by

```
<< pushvar W, T' ;      // W is a new register.
<< fulltranslate( W, t' );
<< translate( V, t'[ box(t') replaced by W ] );
...
<< popvar W;    // At the end of the sequence,
                // in front of the other popvars.
```

Rvalue References

Using the compliation algorithm, it is easy to explain how Rvalue references are used:

- For every type T, the box operator constructs an R-value reference, so that it has type **func**(**reg**(**rvalref**(T)); T).
- When a return statement has form `return v;`, with `v` a local variable of the function, **translate** is not used at all. Instead, an an R-value copy constructor is used, if there is one.

  As soon as `v` is used in an expression (even as simple as a field selection), R-value references are not used.
- By explicit cast using `std::move( )`.

Destructor Calls

In order to handle destructors properly, every occurrence of `popvar T;` that corresponds to `pushvar V, T`, has to to be replaced by the following sequence of statements.

```
if type T has a destructor then
{
   Let W be new pointer variable of type pntr(ref(T)).
   << pushvar W, ref(T);
   [W] = V;
   << (the destructor call for type T);
   << popvar W, ref(T);
}
<< popvar V, T;
```

Box operators in the context of lazy evaluation

Special attention needs to be given to Box operators in the scope of the lazy evaluating operators `? :`, `&&`, and `||`. Example:

```
X& f( X& );
X operator - ( const X&, const X& );
bool operator < ( const X&, const X& );

x3 = ( x1 < x2 ) ? f( x2 - x1 ) : f( x1 - x2 );
```

For each Box operator in the scope of a lazy operator, one should create a boolean that remembers whether its object has been constructed.

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.
- One could consider adding a second **bool** type, call it **branching_bool**, intended for the efficient handling of nested occurrences of `||` and `&&`. In that case, it must be possible to have calls to **translate** of form `translate( (L1,L2), t)`. It will generate code that jumps either to `L1` or to `L2`.
- Of course, it would be nice to verify the correctness of the translation algorithm, but how? The 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.