Improved Intermediate Code

Problems with the Previous Intermediate Language

The previous set of slides compilation.pdf had some problems that I want to solve in this new set of slides. Problems of the intermediate language were:

1. It was too low level. This makes it not suitable for describing optimizations, makes examples long and unpleasant, and makes the description of code generation harder than it needs to be.
2. (This has nothing to do with the course, but it is important for research.) The intermediate representation was not suitable for proving properties of the program.
3. The intermediate language could not represent SSA-normal form.

Problems with the Previous Translation Algorithm

I was also not happy anymore with the translation algorithm. It had a distinction between **inline** and **outline** functions. Inline functions are functions that are substituted away, while outline functions are functions that are called as subroutines. Outline functions had their parameters always passed in memory, in a complicated way, which was too low level. Inline functions passed had their parameters passed in memory or in local variables. The choice whether a parameter was passed in memory or locally was made by the type system. A type $\mathbf{reg}(T)$ indicates that the parameter was passed in a local variable. A type that is not of form $\mathbf{ref}(T)$ was passed in memory. In order to convert between T and $\mathbf{reg}(T)$, a **read** and a **write** function were used. This was unnecessarily complicated, and it confused the students.

Changes in the Translation Algorithm

In the new version, the following changes are made:

1. There is no reason why the inline/outline distinction should be connected to the way parameters are passed. All functions can pass all parameters of simple types in local variables or in memory.

2. The way a parameter is passed should be not expressed in the language type system, but separately in the function definition. This means that the ref type constructor will disappear, together with the read/write functions.

3. There is no reason why inlining should be performed during compilation. Substituting functions away is a simple operation on intermediate code that can be performed at any time.

Types

The slides in compilation.pdf were rather sloppy about the type system of the intermediate language. It seemed to use a strict subset of the types of C/C^{++}. More precisely, its type system was obtained by removing const, and by replacing reg by pointer everywhere in the source language.

In the new intermediate language, we will only five types: B for **boolean**, C for **char**, I for **integer**, R for **real** and P for **pointer**.

Expressions and Memory

The previous intermediate language used instructions of form `@I = @I + 1; @J = [ $P ];`, etc., in which the right hand side could contain a single operator at most.

In the new intermediate language, we allow the use of arbitrary expressions. This is much easier for algebraic simplification, and it makes the programs shorter and easier to read. Expressions are simply typed by the five types on the previous slide. (A simple type system is a type system with a finite set of types, without subtypes.)

All access to memory will be through pointers. Since the pointer type does not specify what it points to, the type has to be specified when something is loaded from memory.

Constants

- We assume that we have constants for each type $T \in \{B, C, I, F\}$ of the intermediate language. Constants have form $'a'_C, 3_I, 4_I, 4_F$, etc.

We also use some additional constants that act as interface with the source language:

- For each type S of the source language, we assume constants $\text{sizeof}_I(S)$ and $\text{sizeof}_C(S)$. (They differ only in their type.)
- For each **struct**, **class** or **record** type S occuring in the source language, we assume constants $\text{field}_I(S, f)$, and $\text{field}_C(S, f)$, which denote the offset of field f in struct S. (The use of field functions in the previous language was not a good idea.)

Both $\text{field}(S, f)$ and $\text{sizeof}(S)$ are just convenient names for constants. There is no further interaction with the source language.

Pointer Constants

For pointers, we have two types of constants:

- The null pointer nil_P.
- Pointers pointing to global variables. They have form $\text{global}(s)$, where s is the name of the global variable.

  We will not do much with global variables, but we define the mechanism.

Functions

- For each T, we assume basic arithmetic functions $+, -, *, /$, etc. of type $T \times T \to T$.
- We assume boolean operators $\wedge, \vee, \neg$ of type $B \times B \to B$. These operators are not lazy!
- For each T, we assume comparison operators $<, \leq, =, >, \geq, \neq$ of type $T \times T \to B$.
- For types T_1, T_2 we assume conversion functions of form $\text{conv}_{T_1,T_2}(t)$ with type $T_1 \to T_2$. Some of the conversions are lossy, some of them are not. For example, $\text{conv}_{I,F}(i)$ converts i to a float.
- For the pointer type P, and the other types T, we assume addition functions $+\colon P \times T \to P$ and $-\colon P \times T \to P$. There is also a subtraction function $-$ with type $P \times P \to I$. These functions do not take the size of $\star P$ into account!

Reading from Memory

For each type T, we assume a function $\mathrm{mem}_T(p)$ of type $P \to T$ that reads the contents of pointer p from memory.

Reading from memory is not a function (because it uses the memory), but it is still safe to treat it as a function because it has no side effects, and will have the same value when it occurs twice in the same expression.

Local Variables

It is possible to use local variables in expressions. The type of a local variable is determined by its declaration.

Since operators are always typed, every expression has a type, which can be easily determined.

If one wants, one can mark each subexpression with its type, but the type system is so simple, that we will not do this most of the time.

Examples of Expressions

We can now write the translations of simple expressions. Assume that **p** is declared as a **struct** with a field **f** of type **int**. Assume that i is of type **int**. Then `p[ i + 1 ].f` can be represented by the following expression:

$$\text{mem}_I(\ p + \text{sizeof}_I(S) * (i + 1_I) + \text{field}_I(S, f)\ ).$$

We didn't write the types in the expresssion, but they can be easily determined.

Function Definitions

The next step is to define functions. A function declaration has form $F{:}T_1 \times \cdots \times T_n \to U_1 \times \cdots \times U_M :\ \lambda v_1, \ldots, v_n\ G$, where F is the function that is being declared, $v_1, \ldots, v_n$ are the parameters of the function, and G is a flow graph.

The flow graph consists of statements of the following forms:

1. Assignment/Write/Copy Statements.
2. Goto/Conditional statements.
3. Allocation/Deallocation statements.
4. Call/Return statements.
5. Merging statements.

Assignment/Write/Copy Statement

An assignment statement stores the result of an expression into a local variable. It has form $v := e{:}T$, where $T \in \{B, C, I, F, P\}$ is the type of e. If the type of e is easily visible, we may omit T. Assignment can either create variable v, or overwrite variable v.

A write statement writes the result of an expression into memory. It has form write $p,\ e{:}T$, where T is a type, p is an expression of type P, and e is an expression of type T. If the type of e is evident, T can be omitted.

A copy statement has form copy $p, q, \text{sizeof}(S)$. Both p and q must be expressions of type P. If everything goes well, the contents of pointer p in memory is copied into pointer q. This can not be done by a combination of read and write, because S does not have to be a primitive type.

Goto/Conditional Statements

A goto statement has form: goto $L_1, \ldots, L_n$;

It is possible that $n > 1$, because we allow non-determinism.

A conditional statement has form $\{e\}$ where e is an expression of type B.

If e evaluates to true, then evaluation continues. Otherwise, evaluation fails.

The combination of branching gotos and conditional statements can be used to simulate **if** and **switch** statements.

Goto/Conditional Statements (2)

Conditional statements are expressed by a combination of non-deterministic gotos, and conditional statements { }. The statement **`if B then S1 else S2`** is represented by:

```
   goto L1,L2;

L1: { B }; S1; goto L3;

L2: { !B }; S2; goto L3;

L3:
```

This looks a bit strange at first, but it is easier to compute abstract interpretations in this way.

Allocation/Deallocation Statements

Allocation/deallocation is always assumed to take place on the stack. If one wants to allocated something on the heap, one must use **new** and **delete**, which I assume are library functions.

- allocate p, c;

  Allocate a space of size c and assign its address to p.

- deallocate p, c;

  p must evaluate to a pointer that was allocated before with the same number c.

Call/Return

- call f; $e_1{:}T_1, \ldots, e_n{:}T_n$; $w_1{:}U_1, \ldots, w_m{:}U_m$;

  Call function f. The types $T_1, \ldots, T_n, U_1, \ldots, U_m$ must fit to the declaration of f. The expressions $e_1, \ldots, e_n$ are evaluated and written into the parameters $v_1, \ldots, v_n$ of f.

  The $w_1, \ldots, w_k$ must be variables, into which the returned values are copied.

- return $e'_1, \ldots, e'_m$;

  Exit the current function and return to the point from where it was called. The types of $e'_1, \ldots, e'_m$ must fit to the $U_1, \ldots, U_m$ in the declaration of f. The values $e'_1, \ldots, e'_k$ are copied into the variables $w_1, \ldots, w_k$ that were used in the call of f. Return does not work (not return) when there were allocations in the body of the function that were not deallocated.

Call/Return (2)

If the types are clear, we will omit them.

Aim of the call and return mechanism is to be able to model standard ways of parameter passing:

- Simple values can be passed and returned in registers.
- Types that are not simple can be passed and returned through pointers pointing to the position in memory where the parameter can be read from or written to.

Merging Statements

Merging Statements are used in Static Single Assignment form. We will not always use SSA, but we define the merging statements for the case we need them.

A merging statement consists of a set of assignments of form $v := \Phi(v_1, \ldots, v_n){:}T$.

All variables $v, v_1, \ldots, v_n$ must have type T.

It is always assumed that the mergings in the statement take place in parallel.

Example with Arrays

Consider the program

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

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

Arrays (2)

Function fill has type $P \times I \rightarrow$. It is defined as $\lambda p, n :$

```
      $i := 0_I{:}I;$
$L_0 :$  goto $L_1,\ L_2;$
$L_1 :$  $\{i < n\};$
      write $p + i * \mathrm{sizeof}_I(\mathrm{int}),\ i * i;$
      $i := i + 1_I;$
      goto $L_0;$
$L_2 :$  $\{\neg(i < n)\};$
      return;
```

Arrays (3)

Function main has type $\rightarrow I$. It is defined as $\lambda :$

$$\begin{aligned}
&\text{allocate squares},\ 5_I * \text{sizeof}_I(\text{int});\\
&\text{call fill};\ \text{squares},\ 5_I;\\
&\text{deallocate squares},\ 5_I * \text{sizeof}_I(\text{int});\\
&\text{return } 0_I;
\end{aligned}$$

Adding an Offset Type?

It may be sensible to add an offset type O to the intermediate language. O would be used for address calculations. It would be the only type that can be added to a pointer. sizeof and field would always construct O.

We will not use O in the slides, but it probably should be added if one wants to use the intermediate language seriously.

Field Functions

I deleted the field functions because they were not a good idea.

They added an offset to a pointer and changed its type. Since we have identified all pointer types, the second aspect is no longer needed. For adding the offset, one can use field(S, f).

Example with Strcpy

This is the **strcpy** function that was used as an example of optimization:

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

Strcpy in Intermediate Representation

strcpy has type $P \times P \rightarrow$ and can be implemented as $\lambda p, q :$

```
     $i := 0_I;$
$L_0$  $b := (\mathrm{mem}_C(q + i * \ \mathrm{sizeof}_I(\mathrm{char})) \neq 0_C) : B;$
     goto $L_1, L_2;$
$L_1$  $\{b\};$
     write $p + i * \mathrm{sizeof}_I(\mathrm{char}),\ \mathrm{mem}_C(q + i * \ \mathrm{sizeof}_I(\mathrm{char}));$
     $i := i + 1_I;$
     goto $L_0;$
$L_2$  $\{\neg b\};$
     write $p + i * \mathrm{sizeof}_I(\mathrm{char}),\ 0_C;$
     return;
```

Strcpy in SSA Normal Form

$\lambda p, q :$

```
         $i_0 := 0_I;$
   $L_0$   $i_1 := \Phi(i_0, i_2);$
         $b := (\mathrm{mem}_C(q + i_1 * \ \mathrm{sizeof}_I(\mathrm{char})) \neq 0_C\ ) \colon B;$
         goto $L_1, L_2;$
   $L_1$   $\{b\};$
         write $p + i_1 * \ \mathrm{sizeof}_I(\mathrm{char}),\ \mathrm{mem}_C(q + i_1 * \ \mathrm{sizeof}_I(\mathrm{char}));$
         $i_2 := i_1 + 1_I;$
         goto $L_0;$
   $L_2$   $\{\neg b\};$
         write $p + i_1 * \mathrm{sizeof}_I(\mathrm{char}),\ 0_C;$
         return;
```

Observations

The new examples are a lot shorter, and a lot more readable than the same example in optimization.pdf, which used the old intermediate language. It fits on a single slide instead of 6.

Even the SSA fits on a single slide, although we cheated a little bit by omitting the return statement.

It is immediately possible to see the redundant expressions, and the possible optimizations.

Another Example: Recursive Factorial Function

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

Translation of fact

fact has type $I \to I$. Its definition is:

```
λn :
        goto $L_1, L_2$;
  $L_1$   $\{n = 0_I\}$;
        return $1_I$;
  $L_2$   $\{n \neq 0_I\}$;
        call $(n - 1_I); m$;
        return $n * m$;
```

Translation into Stack Machine

(This should be kept, but moved to another, earlier point in the course.)

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.

Type System of C/C^{++}

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

- We assume the primitive types **bool**, **char**, **int**, **float**. (I know that there are more primitive types, but I ignore them.)
- We assume the primitive type **void**.
- We assume the primitive type **pointer**.
- We assume that every **struct** or **class** occurring in the program has an associated type.

  We will not distinguish between **class** and **struct**.

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, then **rvalref**(T) is a type.
- If T is a type, and n is a an integer ≥ 0, then **array**(n, T) is a type.

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

ref(T) denotes `T&`. **rvalref**(T) denotes `T&&`.

pointer denotes a pointer to an unknown type. (This would be called **pointer**(**void**) in C/C^{++}.

Const

I assume that every (sub)type except **void**, **ref** and **rvalref** can be marked with a C, which makes it **const**.

The type **const int**$\star$ is represented as $\mathbf{pointer}(\mathbf{int}_C)$.

The type **int** $\star$ **const** is represented as $\mathbf{pointer}_C(\mathbf{int})$.

C/C^{++}-Function Types

A function has type $T_1 \times \cdots \times T_n \Rightarrow U_m$, with $n \geq 0$, and $m \in \{0, 1\}$.

$m = 0$ when the function returns **void**, and $m = 1$ when the function returns a result.

If the function is a non-static member function of a class X, then $n > 0$ and $T_1 = \mathbf{ref}(X)$ or $\mathbf{ref}(X_C)$.

In the previous version, I used the notation $\mathrm{func}(T_1, \ldots, T_n;\ U)$. This was harder to read, and added nothing useful.

Restrictions on Types

There are couple of restrictions on the possible types:

- One cannot have $T_1(T_2(\ ))$, where T_1 is $\mathbf{ref}, \mathbf{rvalref}, \mathbf{pointer}$ or **array**, and T_2 is $\mathbf{ref}, \mathbf{rvalref}$ or **void**.

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( int_C )

const int& f( const int& , int, int* );
   f : ref( int_C ) * int * pointer(int) -> ref( int_C ).
```

Examples of C^{++}-Types (2)

```
struct X
{
   X( int x );
      int -> X
   X( const X& );
      ref(X_C) -> X
         // Constructors are always static!
   void operator = ( const X& );
      ref(X) * ref(X_C) -> void
         // Non-static, non-const member.
   ~X( );
      ref(X) -> void; // Non-static, non-const member.
   std::ostream& print( std::ostream& ) const;
      ref(X_C) * ref(std::ostream) -> ref(std::ostream);
         // Non-static, non-const member.
```

Inline/Outline Functions

In the previous version of these slides, we had inline and outline functions. Outline functions were called, while inline functions were substituted away.

In addition, inline functions had their parameters passed in local variables, while outline functions had their parameters passed in memory.

Since there is no reason why these two choices should be connected, we will completely separate the way parameters are passed from the question whether the function is substituted or called.

We few inlining as an operation on intermediate code that can be performed one every function whose definition is available after the translation process.

Passing of Parameters and the Return Value

We assume that a function has type $T_1 \times \cdots \times T_n \to U_m$ with $n \geq 0$, and $m \in \{0, 1\}$.

If some T_i has a type that corresponds to a type $\{B, C, I, F, P\}$ of the intermediate language, it can be passed as a parameter in the call statement.

Similarly, if the type of U_0 corresponds to a type of the intermediate language, it can be returned through the return statement.

We call this way of passing values **direct**.

Indirect Parameter Passing

If some T_i is not simple, e.g. a struct or a class, it has to be passed indirectly. The calling environment stores the object somewhere in memory, and passes a pointer to the object to the called function.

The same has to done with U_0 if it is a struct. The calling environment creates a position in memory where the struct should be written and passes a pointer to the function. When the called function meets a return statement, it writes the result to the indicated position and returns.

The reasons for this were explained in the example of the stack machine. Copying user defined objects may be inefficient and/or impossible.

The translation of the fact(n) function used direct parameter passing. With indirect parameter passing, the translation would be $\lambda p, q : P \times P \to$:

```
         $n := \mathrm{mem}_I(p)$;
         goto $L_1, L_2$;
$L_1$    $\{n = 0_I\}$;   write $q{:}I,\ 1_I$;   return;
$L_2$    $\{n \neq 0_I\}$;
         allocate $p_1,$ sizeof$(I)$;   allocate $q_1,$ sizeof$(I)$;
         write $p_1{:}I,\ n - 1_1$;
         call $p_1, q_1$;
         write $q{:}I,\ n * \mathrm{mem}_I(q_1)$;
         deallocate $q_1,$ sizeof$(I)$;   deallocate $p_1,$ sizeof$(I)$;
         return;
```

Parameter Passing (4)

We assume that for each function, each of its parameters T_i and U_0 (if it is there) are marked either as direct or indirect. A parameter T_i, U_0 can be marked as direct only when it is of type **bool**, **char**, **int** or **float**.

We do not view the direct/indirect distinction as part of the signature of the function, but as an additional marker. The first implementation of factorial would be marked as $\mathbf{int}^D \rightarrow \mathbf{int}^D$. The second version would be marked as $\mathbf{int}^I \rightarrow \mathbf{int}^I$.

In the previous version of these slides, I used **reg** for direct parameters, and treated the direct/indirect distinction as part of the source language. This was not a great idea, because it made type checking harder, gave the resulting conversions a cost in the source language, and made the resulting trees more complicated.

Associated Struct, General Format of Function Calls

Let F be a function with type $T_1 \times \cdots \times T_n \rightarrow U_m$.

If one of the T_i is indirect, we associate to the function a **struct** which has a field f_i of type T_i for every T_i that is indirect.

The translation f of F in the intermediate language has the following parameters:

1. For every direct parameter v, the translation has a direct parameter of corresponding type.
2. If there are indirect parameters, then there is one parameter p of type P, which points to the associated struct of F.
3. If U_0 exists and is indirect, then the translation has a parameter q of type P which points to the return value.

(If U_0 exists and is direct, it will be returned through the return statement.)

Typechecking and Adding Conversions

Before an expression tree can be translated into intermediate language (lowered), it has to be typechecked by the algorithm in **typechecking.pdf**. During type checking, conversions are added, in order to make sure that references are properly converted to values, arrays are converted to pointers, and values are properly converted into references. The following needs to be specified:

1. How the leafs of an expression tree (constants and variables) are typed.
2. Overloadings of the standard operators and user defined functions. (User defined functions and built-in operators can be treated in the same way.)
3. Typing rules for field selection.
4. Available implicit conversion functions, and their types.

Typing Rules for Constants

We start at the leafs:

Constants have simple type **bool**, **char**, **int**, **float**, or generic pointer type **pointer**. (The null pointer).

Typing Rules for Variables

We give the typing rules for variables, and function parameters inside the function:

1. Variables and parameters that are declared as `X v` receive type $\textbf{ref}(X)$.
2. Variables and parameters that are declared as `const X v` receive type $\textbf{ref}(X_C)$.
3. Variables and parameters that are declared as `X& v` receive type $\textbf{ref}(X)$.
4. Variables and parameters that are declared as `const X& v` receive type $\textbf{ref}(X_c)$.
5. Variables and parameters that are declared as `X&& v;` receive type $\textbf{ref}(X)$.

Typing Rules for Standard Operators

The standard operators have their standard, totally non-surprising types.

- $+, -, *, /$ have type $T \times T \rightarrow T$ for $T \in \textbf{char}, \textbf{int}, \textbf{double}$.
- $+$ and $-$ also have types $\textbf{pointer}(X) \times T \rightarrow \textbf{pointer}(X)$ and $\textbf{pointer}(X_C) \times T \rightarrow \textbf{pointer}(X_C)$, where X can be every type, and $T \in \textbf{char}, \textbf{int}$.
- In addition, $-$ has can have type $\textbf{pointer}(X_C) \times \textbf{pointer}(X_C) \rightarrow \textbf{int}$, where X can be every type.
- $x++$ and $x--$ have type $\textbf{ref}(X) \rightarrow X$.
- $++x$ and $--x$ have type $\textbf{ref}(X) \rightarrow \textbf{ref}(X)$.

Standard Operators (2)

- The comparison operators $<, >, =, \neq, \leq, \geq$ have type $T \times T \rightarrow \mathbf{bool}$ for $T \in \mathbf{bool}, \mathbf{char}, \mathbf{int}, \mathbf{double}, \mathbf{pointer}(X), \mathbf{pointer}(X_C)$.
- The $\star$ operator (pointer dereference) has type $\mathbf{pointer}(T) \rightarrow \mathbf{ref}(T)$ and $\mathbf{pointer}(T_C) \rightarrow \mathbf{ref}(T_C)$, for every type T that can be used in pointers or references.
- The & operator (address of) has types $\mathbf{ref}(T) \rightarrow \mathbf{pointer}(T)$ and $\mathbf{ref}(T_C) \rightarrow \mathbf{pointer}(T_C)$ for every type T that can be used in pointers or references.
- (Overwriting) Assignment $=$ has type $\mathbf{ref}(T) \times T \rightarrow \mathbf{ref}(T)$. for $T \in \mathbf{bool}, \mathbf{char}, \mathbf{int}, \mathbf{float}, \mathbf{pointer}(X)$. Other assignment operators can be defined by the user.

Standard Operators (3)

- The operator $B?\ E_1 : E_2$ has type $\mathbf{bool} \times T \times T \rightarrow T$ for every type T.
- We assume that $A||B$ is replaced by $A?\ 1 : B$ and that $A\&\&B$ is replaced by $A?\ B : 0$. Both have type $\mathbf{bool} \times \mathbf{bool} \rightarrow \mathbf{bool}$.
- We assume that $!A$ is replaced by $A?\ 0 : 1$. It has type $\mathbf{bool} \rightarrow \mathbf{bool}$.
- We assume that $p[i]$ is replaced by $\star(p+i)$, and that $p \rightarrow f$ is replaced by $(\star p).f$.

(There are some more operators, like $+=$, $-=$, whose types should be clear by now.)

Typing Rules for Field Selection

The typing rules for field selection terms of form $t.f$ are very very complicated. They must be approximately like this: Term t must have a type of form $\textbf{ref}(X)$, $\textbf{ref}(X_C)$ or $\textbf{rvalref}(X)$, and X must be a struct type. The resulting type depends on the form of the declaration of field f in struct X.

- If the declaration is just `Y f,` then the result type will be $\textbf{ref}(Y), \textbf{ref}(Y_C), \textbf{rvalref}(Y)$, where the qualifier is inherited from the type of X.
- If the declaration has form `const Y f`, then the result type will as above, but Y will always be const. I have no idea what happens if X has type $\textbf{rvalref}(X)$.
- If the declaration of f has one of the forms `Y& f`, `const Y& f`, or `Y&& f,`, then the result will be of type $\textbf{ref}(Y)$, $\textbf{ref}(Y_C)$, or $\textbf{rvalref}(Y)$, where the qualifier is taken from the declaration.

Conversions: Const and Rvalref

We assume that

$$\begin{aligned} X &\Rightarrow X_C \\ \mathbf{ref}(X) &\Rightarrow \mathbf{ref}(X_C) \\ \mathbf{pointer}(X) &\Rightarrow \mathbf{pointer}(X_C) \\ \mathbf{rvalref}(X) &\Rightarrow \mathbf{ref}(X_C) \end{aligned}$$

without need to insert a conversion operator.

(It may be that the rules are recursive in reality. I have no wish to check it now.)

Conversions: Copy Constructors

Copy constructors are inserted when a reference is available, but a value is needed.

Copy constructors have type $\mathbf{ref}(T) \rightarrow T$, $\quad \mathbf{ref}(T_C) \rightarrow T$, or $\mathbf{rvalref}(T) \rightarrow T$.

T can be every type, but not an array type.

If i, j are of type **int**, then $i = j$; requires a copy constructor, because j has type $\mathbf{ref}(\mathbf{int})$, and $=$ requires **int** as right argument.

Conversions: The Box Operator

The box operator is inserted when a value is given, but a reference is needed. Consider

```
X g( );
int f1( X& );
int f2( const X& );
int f3( X&& );
```

The expressions `f2(g( ))` and `f3(g( ))` are possible. In order to properly evaluate them, the X constructed by `g( )` needs to be stored in a local variable, and kept until the expression is completely evaluated. We say that the value of `g( )` is boxed. The box operator has type $T \rightarrow \textbf{rvalref}(T)$.

Inserting the conversion in the expressions above gives `f2(box(g( )))` and `f3(box(g( ))).`

Conversions: From Arrays to Pointers

Arrays don't have copy constructors, but they have something much better:

$\textbf{ref}(T) \rightarrow \textbf{pointer}(T)$, and $\textbf{ref}(T_C) \rightarrow \textbf{pointer}(T_C)$.

In earlier versions of C, the same conversion was applied to structs.

Conversions: Upcasting and User Conversions

There all kinds of additional conversions.

$$\textbf{bool} \Rightarrow \textbf{char} \Rightarrow \textbf{int} \Rightarrow \textbf{float}.$$

The conversions are called $T2U$, where T is the type that we convert from, and U is the type that we convert into.

Translate

We define the function **translate**(t, U, I), where t is an expression tree obtained after typechecking, using the rules on the previous slides. U is the C/C^{++}-type of expression t, and I is either **void**, **direct**(v) with v a variable, or **indirect**(p), with p a pointer expression.

- If $I =$ **void**, then **translate** will create code that does not create any result.
- If $I =$ **direct**(v), then **translate** will create code that assigns the result to v.
- If $I =$ **indirect**(p), then **translate** will create code that writes its result into the memory position indicated by p.

Preparation for Box

If the expression that we are translating contains box functions, we assign to each occurrence of box a boolean b and a pointer p. The boolean b is used for remembering if the box operator was executed. This is required for expressions of form $b?\ f(\text{box}(g(\ ))) : e_2$, in case when the boxed object has a destructor.

Top Level Call

1. In a return statement of a function with direct result, in the condition of an if,while, etc., we call **translate**$(t, U, \textbf{direct}(v))$.
2. If the expression occurs 'free', we call **translate**$(t, U, \textbf{void})$.
3. If the expression occurs in an initialization $X\ x = e$ or $X\ x(e)$, then call **translate**($e, X, \text{indirect}(x)$);

I first have to check how long temporaries are kept alive !!

We use notation $\overline{T}$ for type conversion.

Translate: Handling Conflicts

We define **translate**(t, U, I). We first check for conflicts between I and the way in which expression t constructs its result:

- If I has form **void**, and t has form $f(t_1, \ldots, t_n)$ for a function f that has indirect result $U \neq$ **void**, then let p be a new pointer variable.

  $$\ll \quad \text{allocate } p, \ \text{sizeof}(U);$$

  Call **translate**($t, U,$ **indirect**(p)).

  $$\ll \quad \text{deallocate } p, \ \text{sizeof}(U);$$

(This corresponds to the situation where a function creates a result in memory, which is ignored.)

Translate: Handling Conflicts (2)

- If I has form **void**, $U \neq$ **void**, and t is not a functional term, or it has form $f(t_1, \ldots, t_n)$ with f a function that has a direct result, then let v be a new variable of type $\overline{U}$.

  Call **translate**($t, U,$ **direct**(v)).

(This is the situation, where t creates a result in a local variable, which has to be ignored.)

Translate: Handling Conflicts (3)

- If I has form **direct**(v), and t has form $f(t_1, \ldots, t_n)$ for a function f that has indirect result $U \neq$ **void**, then let p be a new pointer variable.

$$\ll \quad \text{allocate } p,\ \text{sizeof}(U);$$

Call **translate**$(\ t, U,\ $**indirect**$(p)\ )$;

$$\ll \quad v = \text{mem}_{\overline{U}}(p);$$
$$\ll \quad \text{deallocate } p,\ \text{sizeof}(U);$$

(This replaces **read** in the previous compilation algorithm.)

Translate: Handling Conflicts (4)

- If I has form **indirect**(p), while either
  1. t is a constant (with type in **bool**, **char**, **int**, **float**) or the null pointer,
  2. t is a variable or a field selection term, (which implies that U has form **ref**(X), **ref**(X_C), or **rvalref**(X)), or
  3. t has form $f(t_1, \ldots, t_n)$ for a function f that has direct result type $U \neq$ **void**,

  then let v be a new variable of type $\overline{U}$.
  Call **translate**($t, U,$ **direct**(v));

  $$\ll \quad \text{write } p,\ v{:}\overline{U};$$

(This replaces **write** in the previous compilation algorithm.)

Translate: The Box-operator

At this point, we are sure that there is no conflict between the result of the translation and I.

Consider **translate**(Box$(t), U, I$). We know that U has form **rvalref**(T) and is direct, so that I has form **direct**(v).

Let b, p be the variables that were associated to this occurrence of Box. First call **translate**($t, T,$ **direct**(p));

$$\begin{array}{ll} \ll & b := 1_B; \\ \ll & v := p{:}P; \end{array}$$

So we write the result in p, remember that we did this (for the case that T has a destructor), and make p our result.

Translate: Function calls

Consider **translate**($f(t_1, \ldots, t_n), U, I$). If one of the T_i is marked as indirect, then f has an associated struct. Let T_f be its type. Let q be an unused variable.

$$\ll \quad \text{allocate } q,\ \text{sizeof}_I(T_f);$$

Recursive Calls for $f(t_1, \ldots, t_n)$.

For each parameter t_i do the following: (It is usually done in reverse order.) Assume that t_i has type T_i.

- If t_i is a direct parameter of f, then let $v_i{:}\overline{T}_i$ be a new variable. Call

  $$\textbf{translate}(\ t_i, \overline{T}_i, \textbf{direct}(v_i)\ ).$$

  The resulting code writes the value of t_i into v_i.

- If T_i is an indirect parameter of f, then let f_i be the field for t_i in associated struct of f (which had type T_f) Call

  $$\textbf{translate}(\ t_i, T_i, \textbf{indirect}(q + \mathrm{field}_I(T_f, f_i))\ ).$$

  Variable q is the pointer that was allocated on the previous slide. The resulting code writes the value of t_i into $(\star q).f_i$.

Calling the Function f:

Let $t'_1, \ldots, t'_{n'}$ with $n' \geq 0$ be the direct parameters of f. On the previous slide, we have created variables $v'_1, \ldots, v'_{n'}$ with their values.

If two slides ago an associated struct was allocated, then let $\hat{q}$ be the pointer that was allocated. Otherwise, let $\hat{q}$ denote nothing.

- If $I = \textbf{direct}(v)$, then $\quad \ll \quad \text{call } f;\ v'_1, \ldots, v'_{n'}, \hat{q};\ v;$
- If $I = \textbf{indirect}(p)$, then $\quad \ll \quad \text{call } f;\ v'_1, \ldots, v'_{n'}, \hat{q}, p;$
- If $I = \textbf{void}$, then $\quad \ll \quad \text{call } f;\ v'_1, \ldots, v'_{n'}, \hat{q};$

The first case returns the result in v. The second case returns the result in $\star p$. The third case returns no result.

Deallocating the Associated Struct for $f(t_1, \ldots, t_n)$

If $\hat{q}$ is not nothing, then we need deallocate q :

$$\ll \quad \text{deallocate } q, \; \text{sizeof}_I(T_f);$$

That is all for function calls.

Translation of Constants

We define **translate**(c, U, I) for constants c. Constants always have a simple type, so that U has form **bool**, **char**, **int**, **double**, or a pointer type.

We are certain that I has form **direct**(v).

$$\ll \qquad v := c{:}\overline{U};$$

Translation of Variables

We define **translate**(w, U, I) for variables w. The type U is always of form $\mathbf{ref}(X)$ or $\mathbf{ref}(X_C)$.

We are certain that I has form $\mathbf{direct}(v)$ and that v has type P. We assume that the symbol table is able to produce a pointer expression p that points to the place of variable w in memory.

If w was declared as `X w`, then

$$\ll \quad v := p;$$

If w was declared as `X& w`, `const X& w`, or `X&& w,` then

$$\ll \quad v := \mathrm{mem}_P(p);$$

Translation of Field Selection

We define **translate**($t.f, U, I$) for field selection terms. Fields are very similar to variables. Term t passed type checking. This implies that t has a type of form **ref**(X), **ref**(X_c) or **rvalref**(X), and X is a struct type that does have a field f. We also know that U is some reference (or rval reference) to some type Y.

We can be certain that I has form **direct**(v), and that v is of type P.

Let p be a new variable of type P. First call **translate**($t, U, \text{direct}(p)$).

If field f was declared in X as `Y f`, then

$$\ll \quad v := p + \text{field}_I(X, f);$$

If field f was delared in X as `Y& f`, `const Y& f`, or `Y&& f`, then

$$\ll \quad v := \text{mem}_P(p + \text{field}_I(X, f));$$

Translation of Boolean ? Expression1 : Expression2

We define **translate**(b ? $e_1 : e_2, U, I$). There is no restriction on the form of I.

Let $v{:}\,B$ be a new variable, and let L_1, L_2, L_3 be new labels. First call **translate**(b, **bool**, **direct**(v)).

$$\ll \quad \text{goto } L_1, L_2;$$

$$\ll \quad L_1 : \ \{b\};$$

Call translate(e_1, U, I);

$$\ll \quad \text{goto } L_3;$$

$$\ll \quad L_2 : \ \{\neg b\};$$

Call translate(e_2, U, I);

$$\ll \quad L_3 :$$

Some Standard Functions

At this point, it is easy to give implementations of some of the standard functions of C/C^{++}. For each implementation of a function, one needs the following information:

1. Its possible types in C/C^{++}.
2. Information about how the arguments and the return value are passed. (direct or indirect)
3. The type(s) of the argument(s) and result(s) in the intermediate language.
4. Name of the implementation.
5. Parameters and flow graph of the implementation.

Copy Constructors

1. Copy constructors have type
   $\textbf{ref}(T) \rightarrow T, \quad \textbf{ref}(T_C) \rightarrow T, \quad \textbf{rvalref}(T) \rightarrow T$, where T is a type that has a copy constructor.

2. For simple types $\textbf{bool}, \textbf{char}, \textbf{int}, \textbf{float}$ and for pointer types, the argument and the return value are direct.

   For user types, there are no conditions on direct/indirectness.

3. The implementation has type $P \rightarrow \overline{T}$.

4. It is called $CC(\overline{T})$.

5. For primitive types T, $\;CC(\overline{T}))$ is implemented as

   $$\lambda p{:}\, P \text{ return } \text{mem}_{\overline{T}}(p);$$

   For other types, the user decides.

Simple Assignment Operators

I describe only the assignment operators on primitive types. There may be more assignment operators, defined by the user.

1. Assignment operators have type $\mathbf{ref}(T) \times T \rightarrow \mathbf{ref}(T)$, for types in $\mathbf{bool}, \mathbf{char}, \mathbf{int}, \mathbf{float}$ and $\mathbf{pointer}(X)$.
2. The argument and return value are direct.
3. The implementation has type $P \times \overline{T} \rightarrow P$.
4. It is called assign($\overline{T}$).
5. It is implemented as

$$\lambda p, v: \ \text{write } p, v;$$

Simple Arithmetic Operators

Let $\otimes$ be one of the standard binary arithmetic (or comparison) operators with type $T_1 \otimes T_2 \to T_3$.

1. For each of its possible types T_1, T_2, T_3 in **bool**, **char**, **int** or **float**, the operator $\otimes$ has some type of form $T_1 \times T_2 \to T_3$.
2. The parameters and return value are always direct.
3. The implementation has type $\overline{T}_1 \times \overline{T}_2 \to \overline{T}_3$.
4. It has name $\otimes(\overline{T_1}, \overline{T_2})$.
5. Its definition is:

$$\lambda v_1, v_2 : \ \text{return } v_1 \otimes v_2;$$

$+$ and $-$ on Pointers

1. $+$ and $-$ have types of form $\textbf{pointer}(T) \times \textbf{int} \Rightarrow \textbf{pointer}(T)$ and $\textbf{pointer}(T_C) \times \textbf{int} \Rightarrow \textbf{pointer}(T_C)$, where T is any type that is not a reference type.
2. The argument are return value are direct.
3. The implementations in the intermediate language have types $P \times I \to P$ and $P \times I \to P$.
4. Implementations are called $+ :: \mathit{pntr}(T)+$ and $- :: \mathit{pntr}(T)$.
5.

$$\lambda p, i : \text{return } p + \text{sizeof}_I(T) * i;$$

$$\lambda p, i : \text{return } p - \text{sizeof}_I(T) * i;$$

– between Pointers

1. Pointer subtraction has type $\textbf{pointer}(T) \times \textbf{pointer}(T) \Rightarrow \textbf{int}$, for every type T that is not a reference type.
2. Arguments and return value are direct.
3. The implementation has type $P \times P \to I$.
4. It is called $-(\text{pointer}(T))$.
5. It is implemented as

$$\lambda p_1, p_2 : \text{return } (p_1 - p_2)/\text{sizeof}(T);$$

$\star$ and &

1. Let T be an arbitrary type of C/C^{++}, that is not a reference type. The $\star$-operator (which dereferences pointers) has types $\mathbf{pointer}(T) \Rightarrow \mathbf{ref}(T)$ and $\mathbf{pointer}(T_C) \Rightarrow \mathbf{ref}(T_C)$.

   Its converse & (the address operator) has types $\mathbf{ref}(T) \Rightarrow \mathbf{pointer}(T)$ and $\mathbf{ref}(T_C) \Rightarrow \mathbf{pointer}(T_C)$.
2. Both have direct argument and return value.
3. The implementations have type $P \rightarrow P$.
4. the name doesn't matter much.
5. Both functions are implemented by the trivial function

$$\lambda p : \text{return } p;$$

Increment/Decrement Operators

We discuss only increment. Decrement is similar.

1. The prefix operators have type $\mathbf{ref}(T) \to \mathbf{ref}(T)$. The postfix operators have type $\mathbf{ref}(T) \to T$.
2. Argument and return value are direct.
3. The implementations have type $P \to P$ and $P \to \overline{T}$.
4. They are called $x++(\overline{T})$ and $++x(\overline{T})$.
5. When T is not a pointer type, the operators have form

$$\lambda p: \ \text{write } p,\ \text{mem}_{\overline{T}}(p) + 1_{\overline{T}};\ \text{return } p;$$

$$\lambda p: \ i := \text{mem}_{\overline{T}}(p);\ j := i + 1_{\overline{T}};\ \text{write } p, j;\ \text{return } i;$$

For pointers, the constant $1_{\overline{T}}$ must be replaced by $\text{sizeof}(X)$.

Inlining

Inlining is easy with the new intermediate language. Suppose that $\lambda v_1 \cdots v_n\ G$ is a function definition, let call $f;\ e_1, \ldots, e_n;\ w_1, \ldots, w_k;$ be a call of the function. Assume that variables $v_1, \ldots, v_n$ and the variables in G do not occur in the calling function. Otherwise, rename the variables.

1. Create a new label L, just after the function call.
2. In the body of the function, replace every statement of form return $e'_1, \ldots, e'_k$ by $\quad w_1 = e'_1; \cdots\ w_k = e'_k;$ goto $L;$
3. Replace the call by $v_1 = e_1; \cdots\ v_n = e_n;\ G;\quad$ (G is the body of the function.)

This is much simpler than before, and we have complete independence between inlining and the way parameters are passed.

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 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.