Mapping High Level Constructs to LLVM IR Documentation

Test

Jun 28, 2018

Contents:

Click here to read the book on readthedocs.org

CHAPTER 1

About

This is a gitbook dedicated to providing a description on how LLVM based compilers map high-level language constructs into the LLVM intermediate representation (IR).

This document targets people interested in how modern compilers work and want to learn how high-level language constructs can be implemented. Currently the books focuses on C and C++, but contributions about other languages targeting LLVM are highly welcome. This document should help to make the learning curve less steep for aspiring LLVM users.

For the sake of simplicity, we'll be working with a 32-bit target machine so that pointers and word-sized operands are 32-bits. Also, for the sake of readability we do not mangle (encode) names. Rather, they are given simple, easy-to-read names that reflect their purpose. A production compiler for any language that supports overloading would generally need to mangle the names so as to avoid conflicts between symbols.

CHAPTER 2

Contributing

The repository for this gitbook is hosted on github. All contributions are welcome. If you find an error file an Issue or fork the repository and create a pull-request.

CHAPTER 3

License

Fig. 1: https://creativecommons.org/licenses/by-sa/4.0/

3.1 A Quick Primer

Here are a few things that you should know before reading this document:

- LLVM IR is not machine code, but sort of the step just above assembly. So some things look more like a high-level language (like functions and the strong typing). Other looks more like low-level assembly (e.g. branching, basic-blocks).
- LLVM IR is strongly typed so expect to be told when you do something wrong.
- LLVM IR does not differentiate between signed and unsigned integers.
- LLVM IR assumes two's complement signed integers so that say `trunc` works equally well on signed and unsigned integers.
- Global symbols begin with an at sign (`@`).
- Local symbols begin with a percent symbol (`%`).
- All symbols must be declared or defined.
- Don't worry that the LLVM IR at times can seem somewhat lengthy when it comes to expressing something; the optimizer will ensure the output is well optimized and you'll often see two or three LLVM IR instructions be coalesced into a single machine code instruction.

- If in doubt, consult the Language Reference[1]. If there is a conflict between the Language Reference and this document, this document is wrong! Please file an issue on github then.
- All LLVM IR examples are presented without a data layout and without a target triple. You can assume it's usually x86 or x86_64.
- The original version of this document was written a while ago, therefore some of the snippets of LLVM IR might not compile anymore with the most recent LLVM/clang version. Please file a bug report at github if you encounter such a case.

3.1.1 Some Useful LLVM Tools

The most important LLVM tools for use with this article are as follows:

Name	Function	Reads	Writes	Arguments
`clang`	C Compiler	`.c`	`.ll`	`-emit-llvm -S`
`clang++`	C++ Compiler	`.cpp`	`.ll`	`-emit-llvm -S`
`opt`	Optimizer	`.bc/.ll`	`.bc`	
`llvm-dis`	Disassembler	`.bc`	`.ll`	
`llc`	IR Compiler	`.ll`	`.s`	

While you are playing around with generating or writing LLVM IR, you may want to add the option `-fsanitize=undefined` to Clang/Clang++ insofar you use either of those. This option makes Clang/Clang++ insert run-time checks in places where it would normally output an `ud2` instruction. This will likely save you some trouble if you happen to generate undefined LLVM IR. Please notice that this option only works for C and C++ compiles.

Note that you can use `.ll` or `.bc` files as input files for `clang(++)` and compile full executables from bitcode files.

3.2 Basic Constructs

In this chapter, we'll look at the most basic and simple constructs that are part of nearly all imperative/OOP languages out there.

3.2.1 Global Variables

Global varibles are trivial to implement in LLVM IR:

```
int variable = 21;

int main()
{
    variable = variable * 2;
    return variable;
}
```

Becomes:

[1] http://llvm.org/docs/LangRef.html

```
@variable = global i32 21

define i32 @main() {
    %1 = load i32, i32* @variable  ; load the global variable
    %2 = mul i32 %1, 2
    store i32 %2, i32* @variable   ; store instruction to write to global variable
    ret i32 %2
}
```

Globals are prefixed with the `@` character. You can see that also functions, such as `main`, are also global variables in LLVM. Please notice that LLVM views global variables as pointers; so you must explicitly dereference the global variable using the `load` instruction when accessing its value, likewise you must explicitly store the value of a global variable using the `store` instruction. In that regard LLVM IR is closer to Assembly than C.

3.2.2 Local Variables

There are two kinds of local variables in LLVM:

- Temporary variables/Registers
- Stack-allocated local variables.

The former is created by introducing a new symbol for the variable:

```
%reg = add i32 4, 2
```

The latter is created by allocating the variable on the stack:

```
%stack = alloca i32
```

Nearly every instruction returns a value, that is usually assigned to a temporary variable. Because of the SSA form of the LLVM IR, a temporary variable can only be assigned once. The following code snippet would produce an error:

```
%tmp = add i32 4, 2
%tmp = add i32 4, 1  ; Error here
```

To conform to SSA you will often see something like this:

```
%tmp.0 = add i32 4, 2
%tmp.1 = add i32 4, 1  ; fine now
```

Which can be further shortened to:

```
%0 = add i32 4, 2
%1 = add i32 4, 1
```

The number of such local variables is basically unbounded. Because a real machine does have a rather limited number of registers the compiler backend might need to put some of these temporaries on the stack.

Please notice that `alloca` yields a pointer to the allocated type. As is generally the case in LLVM, you must explicitly use a `load` or `store` instruction to read or write the value respectively.

The use of `alloca` allows for a neat trick that can simplify your code generator in some cases. The trick is to explicitly allocate all mutable variables, including arguments, on the stack, initialize them with the appropriate initial value and then operate on the stack as if that was your end goal. The trick is to run the "memory to register promotion" pass on your code as part of the optimization phase. This will make LLVM store as many of the stack variables in

registers as it possibly can. That way you don't have to ensure that the generated program is in SSA form but can generate code without having to worry about this aspect of the code generation.

This trick is also described in chapter 7.4, Mutable Variables in Kaleidoscope, in the OCaml tutorial on the LLVM website.

3.2.3 Constants

There are two different kinds of constants:

- Constants that do *not* occupy allocated memory.
- Constants that *do* occupy allocated memory.

The former are always expanded inline by the compiler as there is no LLVM IR equivalent of those. In other words, the compiler simply inserts the constant value wherever it is being used in a computation:

```
%1 = add i32 %0, 17     ; 17 is an inlined constant
```

Constants that do occupy memory are defined using the `constant` keyword:

```
@hello = internal constant [6 x i8] c"hello\00"
%struct = type { i32, i8 }
@struct_constant = internal constant %struct { i32 16, i8 4 }
```

Such a constant is really a global variable whose visibility can be limited with `private` or `internal` so that it is invisible outside the current module.

Constant Expressions

An example for constant expressions are `sizeof`-style computations. Even though the compiler ought to know the exact size of everything in use (for statically checked languages), it can at times be convenient to ask LLVM to figure out the size of a structure for you. This is done with the following little snippet of code:

```
%Struct = type { i8, i32, i8* }
@Struct_size = constant i32 ptrtoint (%Struct* getelementptr (%Struct* null, i32 1))␣
↪to i32
```

`@Struct_size` will now contain the size of the structure `%Struct`. The trick is to compute the offset of the second element

in the zero-based array starting at `null` and that way get the size of the structure.

3.2.4 Structures

LLVM IR already includes the concept of structures so there isn't much to do:

```
struct Foo
{
  size_t x;
  double y;
};
```

It is only a matter of discarding the actual field names and then index with numerals starting from zero:

```
%Foo = type {
    i64,        ; index 0 = x
    double      ; index 1 = y
}
```

Nested Structures

Nested structures are also straightforward. They compose in exactly the same way as a C/C++ `struct`.

```
struct FooBar
{
    Foo x;
    char* c;
    Foo* y;
}
```

```
%FooBar = type {
    %Foo,           ; index 0 = x
    i8*,            ; index 1 = c
    %Foo*           ; index 2 = y
}
```

Incomplete Structure Types

Incomplete types are very useful for hiding the details of what fields a given structure has. A well-designed C interface can be made so that no details of the structure are revealed to the client, so that the client cannot inspect or modify private members inside the structure:

```
void Bar(struct Foo *);
```

Becomes:

```
%Foo = type opaque
declare void @Bar(%Foo)
```

Accessing a Structure Member

As already told, structure members are referenced by index rather than by name in LLVM IR. And at no point do you need to, or should you, compute the offset of a given structure member yourself. The `getelementptr` (short GEP) instruction is available to compute a pointer to any structure member with no overhead (the `getelementptr` instruction is typically coascaled into the actual `load` or `store` instruction). The `getelementptr` instruction even has it's own article over at the docs[1]. You can also find more information in the language reference manual[2].

So let's assume we have the following C++ struct:

```
struct Foo
{
    int a;
    char *b;
```

(continues on next page)

[1] The Often Misunderstood GEP Instruction
[2] LangRef: getelementptr Instruction

(continued from previous page)

```
    double c;
};
```

This maps pretty straight forward to the following LLVM type. The GEP indices are in the comments beside the subtypes.

```
%Foo = type {
    i32,        ; 0: a
    i8*,        ; 1: b
    double      ; 2: c
}
```

Now we allocate the object on the stack and access the member `b`, which is at index 1 and has type `char*` in C++.

```
Foo foo;
char **bptr = &foo.b;
```

First the object is allocated with the `alloca` instruction on the stack. To access the `b` member, the GEP instruction is used to compute a pointer to the memory location.

```
%foo = alloca %Foo
; char **bptr = &foo.b
%1 = getelementptr %Foo, %Foo* %foo, i32 0, i32 1
```

Now let's see what happens if we create an array of `Foo` objects. Consider the following C++ snippet:

```
Foo bar[100];
bar[17].c = 0.0;
```

It will translate to roughly something like the following LLVM IR. First a pointer to 100 `Foo` objects is allocated. Then the GEP instruction is used to retrieve the second element of the 17th entry in the array. This is done within one GEP instruction:

```
; Foo bar[100]
%bar = alloca %Foo, i32 100
; bar[17].c = 0.0
%2 = getelementptr %Foo, %Foo* %bar, i32 17, i32 2
store double 0.0, double* %2
```

Note that newer versions of `clang` will produce code that directly uses the built-in support for Array types[3]. This explicitly associates the length of an array with the allocated object. GEP instructions can also have more than two indices to compute addresses deep inside nested objects.

```
%bar = alloca [100 x %Foo]
%p = getelementptr [100 x %Foo], [100 x %Foo]* %bar, i64 0, i64 17, i32 2
store double 0.000000e+00, double* %p, align 8
```

It is highly recommended to read the LLVM docs about the GEP instruction very thouroughly (see[12]).

3.2.5 Casts

There are nine different types of casts:

- Bitwise casts (type casts).

[3] LangRef: Array type

- Zero-extending casts (unsigned upcasts).
- Sign-extending casts (signed upcasts).
- Truncating casts (signed and unsigned downcasts).
- Floating-point extending casts (float upcasts).
- Floating-point truncating casts (float downcasts).
- Pointer-to-integer casts.
- Integer-to-pointer casts.
- Address-space casts (pointer casts).

Bitwise Casts

A bitwise cast (`bitcast`) reinterprets a given bit pattern without changing any bits in the operand. For instance, you could make a bitcast of a pointer to byte into a pointer to some structure as follows:

```
typedef struct
{
    int a;
} Foo;

extern void *malloc(size_t size);
extern void free(void *value);

void allocate()
{
    Foo *foo = (Foo *) malloc(sizeof(Foo));
    foo.a = 12;
    free(foo);
}
```

Becomes:

```
%Foo = type { i32 }

declare i8* @malloc(i32)
declare void @free(i8*)

define void @allocate() nounwind {
    %1 = call i8* @malloc(i32 4)
    %foo = bitcast i8* %1 to %Foo*
    %2 = getelementptr %Foo* %foo, i32 0, i32 0
    store i32 12, i32* %2
    call void @free(i8* %1)
    ret void
}
```

Zero-Extending Casts (Unsigned Upcasts)

To upcast an unsigned value like in the example below:

```
uint8 byte = 117;
uint32 word;

void main()
{
    /* The compiler automatically upcasts the byte to a word. */
    word = byte;
}
```

You use the `zext` instruction:

```
@byte = global i8 117
@word = global i32 0

define void @main() nounwind {
    %1 = load i8* @byte
    %2 = zext i8 %1 to i32
    store i32 %2, i32* @word
    ret void
}
```

Sign-Extending Casts (Signed Upcasts)

To upcast a signed value, you replace the `zext` instruction with the `sext` instruction and everything else works just like in the previous section:

```
@char = global i8 -17
@int  = global i32 0

define void @main() nounwind {
    %1 = load i8* @char
    %2 = sext i8 %1 to i32
    store i32 %2, i32* @int
    ret void
}
```

Truncating Casts (Signed and Unsigned Downcasts)

Both signed and unsigned integers use the same instruction, `trunc`, to reduce the size of the number in question. This is because LLVM IR assumes that all signed integer values are in two's complement format for which reason `trunc` is sufficient to handle both cases:

```
@int = global i32 -1
@char = global i8 0

define void @main() nounwind {
    %1 = load i32* @int
    %2 = trunc i32 %1 to i8
    store i8 %2, i8* @char
    ret void
}
```

Floating-Point Extending Casts (Float Upcasts)

Floating points numbers can be extended using the `fpext` instruction:

```
float small = 1.25;
double large;

void main()
{
    /* The compiler inserts an implicit float upcast. */
    large = small;
}
```

Becomes:

```
@small = global float 1.25
@large = global double 0.0

define void @main() nounwind {
    %1 = load float* @small
    %2 = fpext float %1 to double
    store double %2, double* @large
    ret void
}
```

Floating-Point Truncating Casts (Float Downcasts)

Likewise, a floating point number can be truncated to a smaller size:

```
@large = global double 1.25
@small = global float 0.0

define void @main() nounwind {
    %1 = load double* @large
    %2 = fptrunc double %1 to float
    store float %2, float* @small
    ret void
}
```

Pointer-to-Integer Casts

Pointers do not support arithmetic, which is sometimes needed when doing systems programming. LLVM has support for casting pointer types to integer types using the `ptrtoint` instruction (reference)

Integer-to-Pointer Casts

The `inttoptr` instruction is used to cast an integer back to a pointer (reference).

Address-Space Casts (Pointer Casts)

3.2.6 Function Definitions and Declarations

The translation of function definitions depends on a range of factors, ranging from the calling convention in use, whether the function is exception-aware or not, and if the function is to be publicly available outside the module.

Simple Public Functions

The most basic model is:

```
int Bar(void)
{
    return 17;
}
```

Becomes:

```
define i32 @Bar() nounwind {
    ret i32 17
}
```

Simple Private Functions

A static function is a function private to a module that cannot be referenced from outside of the defining module:

```
define private i32 @Foo() nounwind {
    ret i32 17
}
```

Note that this does not directly map to public/private in the context of `C++`. Two `C++` classes in side one LLVM module can call each other private methods, because they're simply module-level private functions for LLVM.

Function Prototypes

A function prototype, aka a profile, is translated into an equivalent `declare` declaration in LLVM IR:

```
int Bar(int value);
```

Becomes:

```
declare i32 @Bar(i32 %value)
```

Or you can leave out the descriptive parameter name:

```
declare i32 @Bar(i32)
```

Functions with a Variable Number of Parameters

To call a so-called vararg function, you first need to define or declare it using the elipsis (...) and then you need to make use of a special syntax for function calls that allows you to explictly list the types of the parameters of the function that is being called. This "hack" exists to allow overriding a call to a function such as a function with variable

parameters. Please notice that you only need to specify the return type once, not twice as you'd have to do if it was a true cast:

```
declare i32 @printf(i8*, ...) nounwind

@.textstr = internal constant [20 x i8] c"Argument count: %d\0A\00"

define i32 @main(i32 %argc, i8** %argv) nounwind {
    ; printf("Argument count: %d\n", argc)
    %1 = call i32 (i8*, ...) @printf(i8* getelementptr([20 x i8], [20 x i8]* @.
↪textstr, i32 0, i32 0), i32 %argc)
    ret i32 0
}
```

Function Overloading

Function overloading is actually not dealt with on the level of LLVM IR, but on the source language. Function names are mangled, so that they encode the types they take as parameter and return in their function name. For a C++ example:

```
int function(int a, int b) {
    return a + b;
}

double function(double a, double b, double x) {
    return a*b + x;
}
```

For LLVM these two are completely different functions, with different names etc.

```
define i32 @_Z8functionii(i32 %a, i32 %b) #0 {
; [...]
  ret i32 %5
}

define double @_Z8functionddd(double %a, double %b, double %x) #0 {
; [...]
  ret double %8
}
```

Struct by Value as Parameter or Return Value

Classes or structs are often passed around by value, implicitly cloning the objects when they are passed. But they are not

```
struct Point {
    double x;
    double y;
    double z;
};

Point add_points(Point a, Point b) {
  Point p;
  p.x = a.x + b.x;
  p.y = a.y + b.y;
```

(continues on next page)

(continued from previous page)

```
  p.z = a.z + b.z;
  return p;
}
```

This simple example is in turn compiled to

```
%struct.Point = type { double, double, double }

define void @add_points(%struct.Point* noalias sret %agg.result,
                        %struct.Point* byval align 8 %a,
                        %struct.Point* byval align 8 %b) #0 {
; there is no alloca here for Point p;
; p.x = a.x + b.x;
  %1 = getelementptr inbounds %struct.Point, %struct.Point* %a, i32 0, i32 0
  %2 = load double, double* %1, align 8
  %3 = getelementptr inbounds %struct.Point, %struct.Point* %b, i32 0, i32 0
  %4 = load double, double* %3, align 8
  %5 = fadd double %2, %4
  %6 = getelementptr inbounds %struct.Point, %struct.Point* %agg.result, i32 0, i32 0
  store double %5, double* %6, align 8
; p.y = a.y + b.y;
  %7 = getelementptr inbounds %struct.Point, %struct.Point* %a, i32 0, i32 1
  %8 = load double, double* %7, align 8
  %9 = getelementptr inbounds %struct.Point, %struct.Point* %b, i32 0, i32 1
  %10 = load double, double* %9, align 8
  %11 = fadd double %8, %10
  %12 = getelementptr inbounds %struct.Point, %struct.Point* %agg.result, i32 0, i32 1
  store double %11, double* %12, align 8
; p.z = a.z + b.z;
  %13 = getelementptr inbounds %struct.Point, %struct.Point* %a, i32 0, i32 2
  %14 = load double, double* %13, align 8
  %15 = getelementptr inbounds %struct.Point, %struct.Point* %b, i32 0, i32 2
  %16 = load double, double* %15, align 8
  %17 = fadd double %14, %16
  %18 = getelementptr inbounds %struct.Point, %struct.Point* %agg.result, i32 0, i32 2
  store double %17, double* %18, align 8
; there is no real returned value, because the previous stores directly wrote
; to the caller allocated value via %agg.result
  ret void
}
```

We can see that the funtion now actually returns `void` and another parameter was added. The first parameter is a pointer to the result, which is allocated by the caller. The pointer has the attirbute `noalias` because there is no way that one of the parameters might point to the same location. The `sret` attribute indicates that this is the return value.

The parameters have the `byval` attribute, which indicates that they are structs that are passed by value.

Let's see how this function would be called.

```
int main() {
  Point a = {1.0, 3.0, 4.0};
  Point b = {2.0, 8.0, 5.0};
  Point c = add_points(a, b);
  return 0;
}
```

is compiled to:

```
define i32 @main() #1 {
; these are the a, b, c in the scope of main
  %a = alloca %struct.Point, align 8
  %b = alloca %struct.Point, align 8
  %c = alloca %struct.Point, align 8
; these are copies, which are passed as arguments
  %1 = alloca %struct.Point, align 8
  %2 = alloca %struct.Point, align 8
; copy the global initializer main::a to %a
  %3 = bitcast %struct.Point* %a to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %3, i8* bitcast (%struct.Point* @main.a to␣
↪i8*), i64 24, i32 8, i1 false)
; copy the global initializer main::b to %b
  %4 = bitcast %struct.Point* %b to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %4, i8* bitcast (%struct.Point* @main.b to␣
↪i8*), i64 24, i32 8, i1 false)
; clone a to %1
  %5 = bitcast %struct.Point* %1 to i8*
  %6 = bitcast %struct.Point* %a to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %5, i8* %6, i64 24, i32 8, i1 false)
; clone b to %1
  %7 = bitcast %struct.Point* %2 to i8*
  %8 = bitcast %struct.Point* %b to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %7, i8* %8, i64 24, i32 8, i1 false)
; call add_points with the cloned values
  call void @add_points(%struct.Point* sret %c, %struct.Point* byval align 8 %1,
↪%struct.Point* byval align 8 %2)
  ; [...]
}
```

We can see that the caller, in our case `main`, allocates space for the return value `%c` and also makes sure to clone the parameters `a` and `b` before actually passing them by reference.

Exception-Aware Functions

A function that is aware of being part of a larger scheme of exception-handling is called an exception-aware function. Depending upon the type of exception handling being employed, the function may either return a pointer to an exception instance, create a `setjmp`/`longjmp` frame, or simply specify the `uwtable` (for UnWind Table) attribute. These cases will all be covered in great detail in the chapter on *Exception Handling* below.

Function Pointers

Function pointers are expressed almost like in C and C++:

```
int (*Function)(char *buffer);
```

Becomes:

```
@Function = global i32(i8*)* null
```

3.2.7 Unions

Unions are getting more and more rare as the years have shown that they are quite dangerous to use; especially the C variant that does not have a selector field to indicate which of the union's variants are valid. Some may still have a

legacy reason to use unions. In fact, LLVM does not support unions at all:

```
union Foo
{
    int a;
    char *b;
    double c;
};

Foo Union;
```

Becomes this when run through clang++:

```
%union.Foo = type { double }
@Union = %union.Foo { 0.0 }
```

What happened here? Where did the other union members go? The answer is that in LLVM there are no unions; there are only structs that can be cast into whichever type the front-end want to cast the struct into. So to access the above union from LLVM IR, you'd use the `bitcast` instruction to cast a pointer to the "union" into whatever pointer you'd want it to be:

```
%1 = bitcast %union.Foo* @Union to i32*
store i32 1, i32* %1
%2 = bitcast %union.Foo* @Union to i8**
store i8* null, i8** %2
```

This may seem strange, but the truth is that a union is nothing more than a piece of memory that is being accessed using different implicit pointer casts. There is no type-safety when dealing with unions.

If you want to support unions in your front-end language, you should simply allocate the total size of the union (i.e. the size of the largest member) and then generate code to reinterpret the allocated memory as needed.

The cleanest approach might be to simply allocate a range of bytes (`i8`), possibly with alignment padding at the end, and then cast whenever you access the structure. That way you'd be sure you did everything properly all the time.

Tagged Unions

When dealing with unions in C, one typically adds another field that signals the content of the union, since accidently interpreting the bytes of a `double` as a `char*`, can have disastrous consequences.

Many modern programming languages feature type-safe tagged unions. Rust has `enum` types, that can optionally contain values. C++ has the `variant` type since C++17.

Consider the following short rust program, that defines an `enum` type that can hold three different primitive types.

```
enum Foo {
    ABool(bool),
    AInteger(i32),
    ADouble(f64),
}

fn main() {
    let x = Foo::AInteger(42);
    let y = Foo::ADouble(1337.0);
    let z = Foo::ABool(true);

    if let Foo::ABool(b) = x {
```

(continues on next page)

(continued from previous page)

```
        println!("A boolean! {}", b)
    }
    if let Foo::ABool(b) = y {
        println!("A boolean! {}", b)
    }
    if let Foo::ABool(b) = z {
        println!("A boolean! {}", b)
    }
}
```

`rustc` generates something similar to the following LLVM IR to initialize the `Foo` variables.

```
; basic type definition
%Foo = type { i8, [8 x i8] }
; Variants of Foo
%Foo_ABool = type { i8, i8 }        ; tagged with 0
%Foo_AInteger = type { i8, i32 }    ; tagged with 1
%Foo_ADouble = type { i8, double } ; tagged with 2

; allocate the first Foo
%z = alloca %Foo
; pointer to the first element of type i8 (the tag)
%0 = getelementptr inbounds %Foo, %Foo* %x, i32 0, i32 0
; set tag to '1'
store i8 1, i8* %0
; bitcast Foo to the right Foo variant
%1 = bitcast %Foo* %x to %Foo_AInteger*
; store the constant '42'
%2 = getelementptr inbounds %Foo_AInteger, %Foo_AInteger* %1, i32 0, i32 1
store i32 42, i32* %2

; allocate and initialize the second Foo
%y = alloca %Foo
%3 = getelementptr inbounds %Foo, %Foo* %y, i32 0, i32 0
; this time the tag is '2'
store i8 2, i8* %3
; cast to variant and store double constant
%4 = bitcast %Foo* %y to %Foo_ADouble*
%5 = getelementptr inbounds %Foo_ADouble, %Foo_ADouble* %4, i32 0, i32 1
store double 1.337000e+03, double* %5
```

To check whether the given `Foo` object is a certain variant, the tag must be retrieved and compared to the desired value.

```
%9 = getelementptr inbounds %Foo, %Foo* %x, i32 0, i32 0
%10 = load i8, i8* %9
; check if tag is '0', which identifies the variant Foo_ABool
%11 = icmp i8 %10, 0
br i1 %11, label %bb1, label %bb2

bb1:
  ; cast to variant
  %12 = bitcast %Foo* %x to %Foo_ABool*
  ; retrieve boolean
  %13 = getelementptr inbounds %Foo_ABool, %Foo_ABool* %12, i32 0, i32 1
  %14 = load i8, i8* %13,
```

(continues on next page)

(continued from previous page)

```
  %15 = trunc i8 %14 to i1
  ; <...>
```

3.3 Control-Flow Constructs

Similar to low-level assembly languages, LLVM's IR consists of sequences of instructions, that are executed sequentially. The instructions are grouped together to form *basic blocks*. Each basic block terminates with an instruction that changes the control flow of the program.

3.3.1 Simple "if-then-else" Branching

First let's take a look at a very simple function, that computes the maximum of two integers. This is implemented using a single if control statement.

```
int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Remember that in LLVM IR control-flow is implemented by jumping between *basic blocks*, which contain instruction sequences that do not change control flow. Each basic block ends with an instruction that changes the control flow. The most common branching instruction is `br`[2]. `br` can be used conditionally, then it implements a simple if-then-else, taking a boolean condition flag and two basic block labels as parameter.

```
br i1 %cond, label %iftrue, label %iffalse
```

`br` can also be used to unconditinally jump to a certain destination:

```
br label %dest
```

```
define i32 @max(i32 %a, i32 %b) {
entry:
  %retval = alloca i32, align 4
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse

btrue:                                            ; preds = %2
  store i32 %a, i32* %retval, align 4
  br label %end

bfalse:                                           ; preds = %2
  store i32 %b, i32* %retval, align 4
  br label %end

end:                                              ; preds = %btrue, %bfalse
  %1 = load i32, i32* %retval, align 4
  ret i32 %1
}
```

[2] LangRef: br

In the example above, there are 4 basic blocks. The first one is the function entry block. There space is allocated on the stack with `alloca`[1], which acts as a temporary storage for the bigger value. Then the two paramter `%a` and `%b` are compared using the `icmp` instruction[3]. The result is a boolean (`i1`) flag, which is then used as condition for the `br` instruction. Then depending on the taken branch, either `%a` or `%b` is stored into the temporary `%retval` variable. Each of the branches then end with a unconditional branch to the last basic block `%end`. There the value from `%retval` is loaded and returned.

You can get a graphical representation of the control-flow in the form of a control-flow graph (CFG). This can be generated by using `opt -dot-cfg input.ll`.

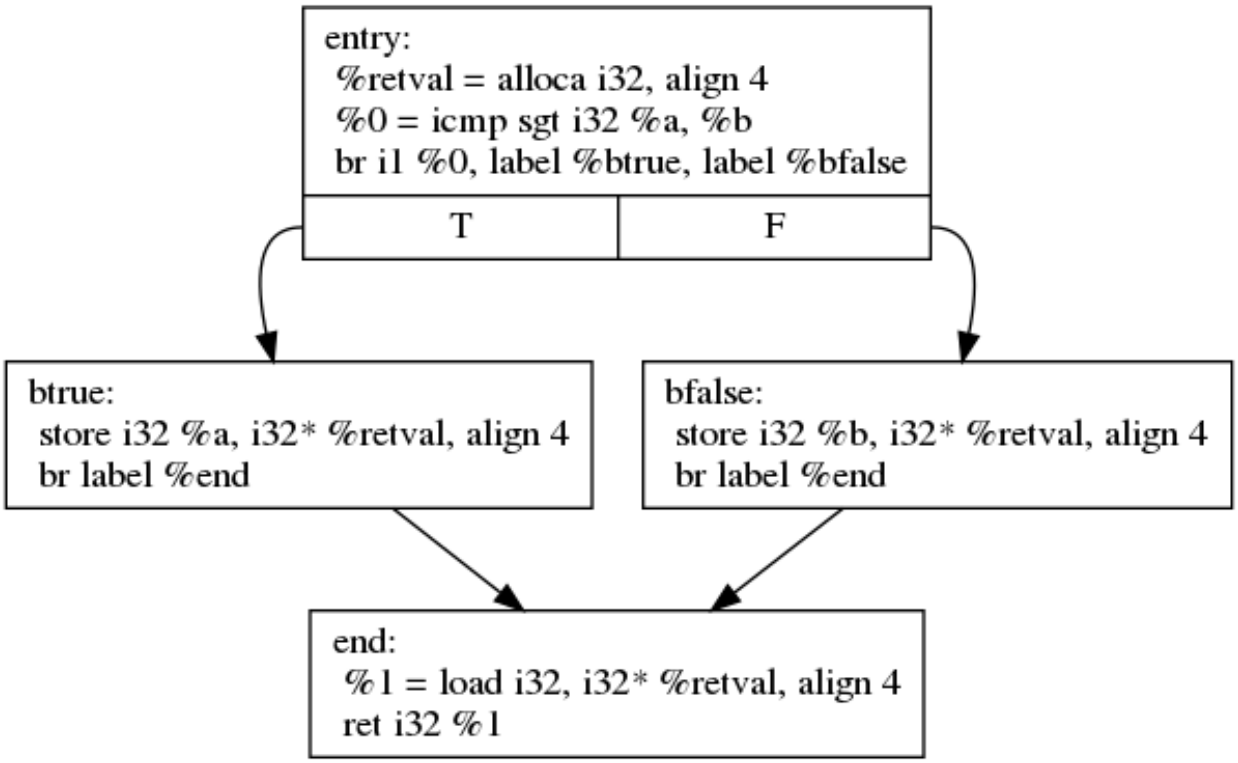

Fig. 2: Control-Flow Graph of the max function

LLVM IR is a rather rich intermediate code format. So when compiling the above snippet with higher optimization levels, LLVM will optimize the code to use the `select` instruction[4] instead of generating branches. The `select` instruction simply chooses between two values, based on a boolean condition. This shortens the code significantly.

```
define i32 @max(i32 %a, i32 %b) {
  %1 = icmp sgt i32 %a, %b
  %2 = select i1 %1, i32 %a, i32 %b
  ret i32 %2
}
```

3.3.2 Single-Static Assignment Form and PHI

We'll take a look at the same very simple `max` function, as in the previous section.

[1] 'LangRef: alloca <http://llvm.org/docs/LangRef.html#alloca-instruction> '__
[3] LangRef: icmp
[4] LangRef: select

```
int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Translated to LLVM IR:

```
define i32 @max(i32 %a, i32 %b) {
entry:
  %retval = alloca i32, align 4
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse

btrue:                                            ; preds = %2
  store i32 %a, i32* %retval, align 4
  br label %end

bfalse:                                           ; preds = %2
  store i32 %b, i32* %retval, align 4
  br label %end

end:                                              ; preds = %btrue, %bfalse
  %1 = load i32, i32* %retval, align 4
  ret i32 %1
}
```

We can see that the function allocates space on the stack with `alloca`[2], where the bigger value is stored. In one branch `%a` is stored, while in the other branch `%b` is stored to the stack allocated memory. However, we want to avoid using memory load/store operation and use registers instead, whenever possible. So we would like to write something like this:

```
define i32 @max(i32 %a, i32 %b) {
entry:
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse

btrue:
  %retval = %a
  br label %end

bfalse:
  %retval = %b
  br label %end

end:
  ret i32 %retval
}
```

This is not valid LLVM IR, because it violates the static single assignment form (SSA,[1]) of the LLVM IR. SSA form requires that every variable is assigned only exactly once. SSA form enables and simplifies a vast number of compiler optimizations, and is the de-facto standard for intermediate representations in compilers of imperative programming languages.

[2] LangRef: alloca
[1] Wikipedia: Static single assignment form

Now how would one implement the above code in proper SSA form LLVM IR? The answer is the magic `phi` instruction. The `phi` instruction is named after the ϕ function used in the theory of SSA. This functions magically chooses the right value, depending on the control flow. In LLVM you have to manually specify the name of the value and the previous basic block.

```
end:
  %retval = phi i32 [%a, %btrue], [%b, %bfalse]
```

Here we instruct the `phi` instruction to choose `%a` if the previous basic block was `%btrue`. If the previous basic block was `%bfalse`, then `%b` will be used. The value is then assigned to a new variable `%retval`. Here you can see the full code listing:

```
define i32 @max(i32 %a, i32 %b) {
entry:
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse

btrue:                                            ; preds = %2
  br label %end

bfalse:                                           ; preds = %2
  br label %end

end:                                              ; preds = %btrue, %bfalse
  %retval = phi i32 [%a, %btrue], [%b, %bfalse]
  ret i32 %retval
}
```

PHI in the Back End

Let's have a look how the `@max` function now maps to actual machine code. We'll have a look what kind of assembly code is generated by the compiler back end. In this case we'll look at the code generated for x86 64-bit, compiled with different optimization levels. We'll start with a non-optimizing backend (`llc -O0 -filetype=asm`). We will get something like this assembly:

```
max:                                    # @max
# %bb.0:                                # %entry
    cmpl    %esi, %edi                  # %edi = %a, %esi = %b
    jle     .LBB0_2
# %bb.1:                                # %btrue
    movl    %edi, -4(%rsp)              # mov src, dst
    jmp     .LBB0_3
.LBB0_2:                                # %bfalse
    movl    %esi, -4(%rsp)              # mov src, dst
    jmp     .LBB0_3
.LBB0_3:                                # %end
    movl    -4(%rsp), %eax              # return value in eax
    retq
```

The parameters `%a` and `%b` are passed in `%edi` and `%esi` respectively. We can see that the compiler back end generated code that uses the stack to store the bigger value. So the code generated by the compiler back end is not what we had in mind, when we wrote the LLVM IR. The reason for this is that the compiler back end needs to implement the `phi` instruction with real machine instructions. Usually that is done by assigning to one register or storing to one common stack memory location. Usually the compiler back end will use the stack for implementing the `phi` instruction. However, if we use a little more optimization in the back end (i.e., `llc -O1`), we can get a more optimized version:

```
max:                                    # @max
# %bb.0:                                # %entry
    cmpl    %esi, %edi
    jg      .LBB0_2
# %bb.1:                                # %bfalse
    movl    %esi, %edi
.LBB0_2:                                # %end
    movl    %edi, %eax
    retq
```

Here the `phi` function is implemented by using the `%edi` register. In one branch `%edi` already contains the desired value, so nothing happens. In the other branch `%esi` is copied to `%edi`. At the `%end` basic block, `%edi` contains the desired value from both branches. This is more like what we had in mind. We can see that optimization is something that needs to be applied through the whole compilation pipeline.

3.4 Object-Oriented Constructs

In this chapter we'll look at various object-oriented constructs and see how they can be mapped to LLVM IR.

3.4.1 Classes

A class is nothing more than a structure with an associated set of functions that take an implicit first parameter, namely a pointer to the structure. Therefore, is is very trivial to map a class to LLVM IR:

```
#include <stddef.h>

class Foo
{
public:
    Foo()
    {
        _length = 0;
    }

    size_t GetLength() const
    {
        return _length;
    }

    void SetLength(size_t value)
    {
        _length = value;
    }

private:
    size_t _length;
};
```

We first transform this code into two separate pieces:

- The structure definition.
- The list of methods, including the constructor.

```
; The structure definition for class Foo.
%Foo = type { i32 }

; The default constructor for class Foo.
define void @Foo_Create_Default(%Foo* %this) nounwind {
    %1 = getelementptr %Foo* %this, i32 0, i32 0
    store i32 0, i32* %1
    ret void
}

; The Foo::GetLength() method.
define i32 @Foo_GetLength(%Foo* %this) nounwind {
    %1 = getelementptr %Foo* %this, i32 0, i32 0
    %2 = load i32* %this
    ret i32 %2
}

; The Foo::SetLength() method.
define void @Foo_SetLength(%Foo* %this, i32 %value) nounwind {
    %1 = getelementptr %Foo* %this, i32 0, i32 0
    store i32 %value, i32* %1
    ret void
}
```

Then we make sure that the constructor (`Foo_Create_Default`) is invoked

whenever an instance of the structure is created:

```
Foo foo;
```

```
%foo = alloca %Foo
call void @Foo_Create_Default(%Foo* %foo)
```

3.4.2 Virtual Methods

A virtual method is no more than a compiler-controlled function pointer. Each virtual method is recorded in the `vtable`, which is a structure of all the function pointers needed by a given class:

```
class Foo
{
public:
    virtual int GetLengthTimesTwo() const
    {
        return _length * 2;
    }

    void SetLength(size_t value)
    {
        _length = value;
    }

private:
    int _length;
};
```

(continues on next page)

(continued from previous page)

```
int main()
{
    Foo foo;
    foo.SetLength(4);
    return foo.GetLengthTimesTwo();
}
```

This becomes:

```
%Foo_vtable_type = type { i32(%Foo*)* }

%Foo = type { %Foo_vtable_type*, i32 }

define i32 @Foo_GetLengthTimesTwo(%Foo* %this) nounwind {
    %1 = getelementptr %Foo* %this, i32 0, i32 1
    %2 = load i32* %1
    %3 = mul i32 %2, 2
    ret i32 %3
}

@Foo_vtable_data = global %Foo_vtable_type {
    i32(%Foo*)* @Foo_GetLengthTimesTwo
}

define void @Foo_Create_Default(%Foo* %this) nounwind {
    %1 = getelementptr %Foo* %this, i32 0, i32 0
    store %Foo_vtable_type* @Foo_vtable_data, %Foo_vtable_type** %1
    %2 = getelementptr %Foo* %this, i32 0, i32 1
    store i32 0, i32* %2
    ret void
}

define void @Foo_SetLength(%Foo* %this, i32 %value) nounwind {
    %1 = getelementptr %Foo* %this, i32 0, i32 1
    store i32 %value, i32* %1
    ret void
}

define i32 @main(i32 %argc, i8** %argv) nounwind {
    %foo = alloca %Foo
    call void @Foo_Create_Default(%Foo* %foo)
    call void @Foo_SetLength(%Foo* %foo, i32 4)
    %1 = getelementptr %Foo* %foo, i32 0, i32 0
    %2 = load %Foo_vtable_type** %1
    %3 = getelementptr %Foo_vtable_type* %2, i32 0, i32 0
    %4 = load i32(%Foo*)** %3
    %5 = call i32 %4(%Foo* %foo)
    ret i32 %5
}
```

Please notice that some C++ compilers store `_vtable` at a negative offset into the structure so that things like `memset(this, 0, sizeof(*this))` work, even though such commands should always be avoided in an OOP context.

3.4.3 Single Inheritance

Single inheritance is very straightforward: Each "structure" (class) is laid out in memory after one another in declaration order.

```
class Base
{
public:
    void SetA(int value)
    {
        _a = value;
    }

private:
    int _a;
};

class Derived: public Base
{
public:
    void SetB(int value)
    {
        SetA(value);
        _b = value;
    }

protected:
    int _b;
}
```

Here, `a` and `b` will be laid out to follow one another in memory so that inheriting from a class is simply a matter of declaring a the base class as a first member in the inheriting class:

```
%Base = type {
    i32         ; '_a' in class Base
}

define void @Base_SetA(%Base* %this, i32 %value) nounwind {
    %1 = getelementptr %Base* %this, i32 0, i32 0
    store i32 %value, i32* %1
    ret void
}

%Derived = type {
    i32,        ; '_a' from class Base
    i32         ; '_b' from class Derived
}

define void @Derived_SetB(%Derived* %this, i32 %value) nounwind {
    %1 = bitcast %Derived* %this to %Base*
    call void @Base_SetA(%Base* %1, i32 %value)
    %2 = getelementptr %Derived* %this, i32 0, i32 1
    store i32 %value, i32* %2
    ret void
}
```

So the base class simply becomes plain members of the type declaration for the derived class.

And then the compiler must insert appropriate type casts whenever the derived class is being referenced as its base class as shown above with the `bitcast` operator.

3.4.4 Multiple Inheritance

Multiple inheritance is not that difficult, either, it is merely a question of laying out the multiply inherited "structures" in order inside each derived class.

```
class BaseA
{
public:
    void SetA(int value)
    {
        _a = value;
    }

private:
    int _a;
};

class BaseB: public BaseA
{
public:
    void SetB(int value)
    {
        SetA(value);
        _b = value;
    }

private:
    int _b;
};

class Derived:
    public BaseA,
    public BaseB
{
public:
    void SetC(int value)
    {
        SetB(value);
        _c = value;
    }

private:
    int _c;
};
```

This is equivalent to the following LLVM IR:

```
%BaseA = type {
    i32         ; '_a' from BaseA
}

define void @BaseA_SetA(%BaseA* %this, i32 %value) nounwind {
    %1 = getelementptr %BaseA* %this, i32 0, i32 0
    store i32 %value, i32* %1
```

(continues on next page)

(continued from previous page)

```
    ret void
}

%BaseB = type {
    i32,        ; '_a' from BaseA
    i32         ; '_b' from BaseB
}

define void @BaseB_SetB(%BaseB* %this, i32 %value) nounwind {
    %1 = bitcast %BaseB* %this to %BaseA*
    call void @BaseA_SetA(%BaseA* %1, i32 %value)
    %2 = getelementptr %BaseB* %this, i32 0, i32 1
    store i32 %value, i32* %2
    ret void
}

%Derived = type {
    i32,        ; '_a' from BaseA
    i32,        ; '_b' from BaseB
    i32         ; '_c' from Derived
}

define void @Derived_SetC(%Derived* %this, i32 %value) nounwind {
    %1 = bitcast %Derived* %this to %BaseB*
    call void @BaseB_SetB(%BaseB* %1, i32 %value)
    %2 = getelementptr %Derived* %this, i32 0, i32 2
    store i32 %value, i32* %2
    ret void
}
```

And the compiler then supplies the needed type casts and pointer arithmentic whenever `baseB` is being referenced as an instance

of `BaseB`. Please notice that all it takes is a `bitcast` from one class to another as well as an adjustment of the last argument to `getelementptr`.

3.4.5 Virtual Inheritance

Virtual inheritance is actually quite simple as it dictates that identical base classes are to be merged into a single occurence. For instance, given this:

```
class BaseA
{
public:
    int a;
};

class BaseB: public BaseA
{
public:
    int b;
};

class BaseC: public BaseA
{
```

(continues on next page)

(continued from previous page)

```
public:
    int c;
};

class Derived:
    public virtual BaseB,
    public virtual BaseC
{
    int d;
};
```

`Derived` will only contain a single instance of `BaseA` even if its inheritance graph dictates that it should have two instances. The result looks something like this:

```
class Derived
{
public:
    int a;
    int b;
    int c;
    int d;
};
```

So the second instance of `a` is silently ignored because it would cause multiple instances of `BaseA` to exist in `Derived`,

which clearly would cause lots of confusion and ambiguities.

3.4.6 Interfaces

An interface is nothing more than a base class with no data members, where all the methods are pure virtual (i.e. has no body).

As such, we've already described how to convert an interface to LLVM IR - it is done precisely the same way that you convert a virtual member function to LLVM IR.

3.4.7 Boxing and Unboxing

Boxing is the process of converting a non-object primitive value into an object. It is as easy as it sounds. You create a wrapper class which you instantiate and initialize with the non-object value:

Unboxing is the reverse of boxing: You downgrade a full object to a mere scalar value by retrieving the boxed value from the box object.

It is important to notice that changes to the boxed value does not affect the original value and vice verse. The code below illustrates both steps:

```
@Boxee = global i32 17

%Integer = type { i32 }

define void @Integer_Create(%Integer* %this, i32 %value) nounwind {
    ; you might set up a vtable and associated virtual methods here
    %1 = getelementptr %Integer* %this, i32 0, i32 0
```

(continues on next page)

(continued from previous page)

```
    store i32 %value, i32* %1
    ret void
}

define i32 @Integer_GetValue(%Integer* %this) nounwind {
    %1 = getelementptr %Integer* %this, i32 0, i32 0
    %2 = load i32* %1
    ret i32 %2
}

define i32 @main() nounwind {
    ; box @Boxee in an instance of %Integer
    %1 = load i32* @Boxee
    %2 = alloca %Integer
    call void @Integer_Create(%Integer* %2, i32 %1)

    ; unbox @Boxee from an instance of %Integer
    %3 = call i32 @Integer_GetValue(%Integer* %2)

    ret i32 0
}
```

3.4.8 Class Equivalence Test

There are two ways of doing this:

- If you can guarantee that each class a unique vtable, you can simply compare the pointers to the vtable.
- If you cannot guarantee that each class has a unique vtable (because different vtables may have been merged by the linker), you need to add a unique field to the vtable so that you can compare that instead.

The first variant goes roughly as follows (assuming identical strings aren't merged by the compiler, something that they are most of the time):

```
bool equal = (typeid(first) == typeid(other));
```

As far as I know, RTTI is simply done by adding two fields to the _vtable structure: `parent` and `signature`. The former is a pointer to the vtable of the parent class and the latter is the mangled (encoded) name of the class. To see if a given class is another class, you simply compare the `signature` fields. To see if a given class is a derived class of some other class, you simply walk the chain of `parent` fields, while checking if you have found a matching signature.

3.4.9 Class Inheritance Test

A class inheritance test is a question of the form: *Is class X identical to or derived from class Y?*

To answer that question, we can use one of two methods:

- The naive implementation where we search upwards in the chain of parents.
- The faster implementation where we search a preallocated list of parents.

The naive implementation is documented in the first two exception handling examples as the `Object_IsA` function.

3.4.10 The New Operator

The `new` operator is generally nothing more than a type-safe version of the C `malloc` function - in some implementations of C++, they may even be called interchangeably without causing unseen or unwanted side-effects.

The Instance New Operator

All calls of the form `new X` are mapped into:

```
declare i8* @malloc(i32) nounwind

%X = type { i8 }

define void @X_Create_Default(%X* %this) nounwind {
    %1 = getelementptr %X* %this, i32 0, i32 0
    store i8 0, i8* %1
    ret void
}

define void @main() nounwind {
    %1 = call i8* @malloc(i32 1)
    %2 = bitcast i8* %1 to %X*
    call void @X_Create_Default(%X* %2)
    ret void
}
```

Calls of the form `new X(Y, Z)` are the same, except `Y` and `Z` are passed into the constructor as arguments.

The Array New Operator

New operations involving arrays are equally simple. The code `new X[100]` is mapped into a loop that initializes each array element in turn:

```
declare i8* @malloc(i32) nounwind

%X = type { i32 }

define void @X_Create_Default(%X* %this) nounwind {
    %1 = getelementptr %X* %this, i32 0, i32 0
    store i32 0, i32* %1
    ret void
}

define void @main() nounwind {
    %n = alloca i32                 ; %n = ptr to the number of elements in the array
    store i32 100, i32* %n

    %i = alloca i32                 ; %i = ptr to the loop index into the array
    store i32 0, i32* %i

    %1 = load i32* %n               ; %1 = *%n
    %2 = mul i32 %1, 4              ; %2 = %1 * sizeof(X)
    %3 = call i8* @malloc(i32 %2)   ; %3 = malloc(100 * sizeof(X))
    %4 = bitcast i8* %3 to %X*      ; %4 = (X*) %3
    br label %.loop_head
```

(continues on next page)

(continued from previous page)

```
.loop_head:                               ; for (; %i < %n; %i++)
    %5 = load i32* %i
    %6 = load i32* %n
    %7 = icmp slt i32 %5, %6
    br i1 %7, label %.loop_body, label %.loop_tail

.loop_body:
    %8 = getelementptr %X* %4, i32 %5
    call void @X_Create_Default(%X* %8)

    %9 = add i32 %5, 1
    store i32 %9 i32* %i

    br label %.loop_head

.loop_tail:
    ret void
}
```

3.5 Exception Handling

Exceptions can be implemented in one of three ways:

- The simple way, by using a propagated return value.
- The bulky way, by using `setjmp` and `longjmp`.
- The efficient way, by using a zero-cost exception ABI.

Please notice that many compiler developers with respect for themselves won't accept the first method as a proper way of handling exceptions. However, it is unbeatable in terms of simplicity and can likely help people to understand that implementing exceptions does not need to be very difficult.

The second method is used by some production compilers, but it has large overhead both in terms of code bloat and the cost of a `try-catch` statement (because all CPU registers are saved using `setjmp` whenever a `try` statement is encountered).

The third method is very advanced but in return does not add any cost to execution paths where no exceptions are being thrown. This method is the de-facto "right" way of implementing exceptions, whether you like it or not. LLVM directly supports this kind of exception handling.

In the three sections below, we'll be using this sample and transform it:

```
#include <stdio.h>
#include <stdlib.h>

class Object {
  public:
  virtual ~Object() {}
};

class Exception : public Object {
  public:
  Exception(const char* text)
      : _text(text)
```

(continues on next page)

(continued from previous page)

```
  {
  }

  const char* GetText(const) { return _text; }

  private:
  const char* _text;
}

class Foo {
  public:
  int GetLength() const { return _length; }

  void SetLength(int value) { _length = value; }

  private:
  int _length;
};

int Bar(bool fail)
{
  Foo foo;
  foo.SetLength(17);
  if (fail)
    throw new Exception("Exception requested by caller");
  foo.SetLength(24);
  return foo.GetLength();
}

int main(int argc, const char* argv[])
{
  int result;

  try {
    /* The program throws an exception if an argument is specified. */
    bool fail = (argc >= 2);

    /* Let callee decide if an exception is thrown. */
    int value = Bar(fail);

    result = EXIT_SUCCESS;
  } catch (Exception* that) {
    printf("Error: %s\n", that->GetText());
    result = EXIT_FAILURE;
  } catch (...) {
    puts("Internal error: Unhandled exception detected");
    result = EXIT_FAILURE;
  }

  return result;
}
```

3.5.1 Exception Handling by Propagated Return Value

This method is a compiler-generated way of implicitly checking each function's return value. Its main advantage is that it is simple - at the cost of many mostly unproductive checks of return values. The great thing about this method

is that it readily interfaces with a host of languages and environments - it is all a matter of returning a pointer to an exception.

The C++ example from the beginning of the section maps to the following code:

```
;******************** External and Utility functions ********************

declare i8* @malloc(i32) nounwind
declare void @free(i8*) nounwind
declare i32 @printf(i8* noalias nocapture, ...) nounwind
declare i32 @puts(i8* noalias nocapture) nounwind

;***************************** Object class *****************************

%Object_vtable_type = type {
        %Object_vtable_type*,          ; 0: above: parent class vtable pointer
        i8*                            ; 1: class: class name (usually mangled)
        ; virtual methods would follow here
}

@.Object_class_name = private constant [7 x i8] c"Object\00"

@.Object_vtable = private constant %Object_vtable_type {
        %Object_vtable_type* null,     ; This is the root object of the object␣
↪hierarchy
        i8* getelementptr([7 x i8]* @.Object_class_name, i32 0, i32 0)
}

%Object = type {
        %Object_vtable_type*           ; 0: vtable: class vtable pointer (always non-
↪null)
        ; class data members would follow here
}

; returns true if the specified object is identical to or derived from the
; class with the specified name.
define i1 @Object_IsA(%Object* %object, i8* %name) nounwind {
.init:
        ; if (object == null) return false
        %0 = icmp ne %Object* %object, null
        br i1 %0, label %.once, label %.exit_false

.once:
        %1 = getelementptr %Object* %object, i32 0, i32 0
        br label %.body

.body:
        ; if (vtable->class == name)
        %2 = phi %Object_vtable_type** [ %1, %.once ], [ %7, %.next]
        %3 = load %Object_vtable_type** %2
        %4 = getelementptr %Object_vtable_type* %3, i32 0, i32 1
        %5 = load i8** %4
        %6 = icmp eq i8* %5, %name
        br i1 %6, label %.exit_true, label %.next

.next:
        ; object = object->above
        %7 = getelementptr %Object_vtable_type* %3, i32 0, i32 0
```

(continues on next page)

(continued from previous page)

```
        ; while (object != null)
        %8 = icmp ne %Object_vtable_type* %3, null
        br i1 %8, label %.body, label %.exit_false

.exit_true:
        ret i1 true

.exit_false:
        ret i1 false
}

;*************************** Exception class ******************************

%Exception_vtable_type = type {
        %Object_vtable_type*,                       ; 0: parent class vtable pointer
        i8*                                         ; 1: class name
        ; virtual methods would follow here.
}

@.Exception_class_name = private constant [10 x i8] c"Exception\00"

@.Exception_vtable = private constant %Exception_vtable_type {
        %Object_vtable_type* @.Object_vtable,       ; the parent of this class is␣
↪the Object class
        i8* getelementptr([10 x i8]* @.Exception_class_name, i32 0, i32 0)
}

%Exception = type {
        %Exception_vtable_type*,                    ; 0: the vtable pointer
        i8*                                         ; 1: the _text member
}

define void @Exception_Create_String(%Exception* %this, i8* %text) nounwind {
        ; set up vtable
        %1 = getelementptr %Exception* %this, i32 0, i32 0
        store %Exception_vtable_type* @.Exception_vtable, %Exception_vtable_type** %1

        ; save input text string into _text
        %2 = getelementptr %Exception* %this, i32 0, i32 1
        store i8* %text, i8** %2

        ret void
}

define i8* @Exception_GetText(%Exception* %this) nounwind {
        %1 = getelementptr %Exception* %this, i32 0, i32 1
        %2 = load i8** %1
        ret i8* %2
}

;****************************** Foo class *********************************

%Foo = type { i32 }

define void @Foo_Create_Default(%Foo* %this) nounwind {
        %1 = getelementptr %Foo* %this, i32 0, i32 0
```

(continues on next page)

(continued from previous page)

```
        store i32 0, i32* %1
        ret void
}

define i32 @Foo_GetLength(%Foo* %this) nounwind {
        %1 = getelementptr %Foo* %this, i32 0, i32 0
        %2 = load i32* %1
        ret i32 %2
}

define void @Foo_SetLength(%Foo* %this, i32 %value) nounwind {
        %1 = getelementptr %Foo* %this, i32 0, i32 0
        store i32 %value, i32* %1
        ret void
}

;******************************* Foo function ************************

@.message1 = internal constant [30 x i8] c"Exception requested by caller\00"

define %Exception* @Bar(i1 %fail, i32* %result) nounwind {
        ; Allocate Foo instance
        %foo = alloca %Foo
        call void @Foo_Create_Default(%Foo* %foo)

        call void @Foo_SetLength(%Foo* %foo, i32 17)

        ; if (fail)
        %1 = icmp eq i1 %fail, true
        br i1 %1, label %.if_begin, label %.if_close

.if_begin:
        ; throw new Exception(...)
        %2 = call i8* @malloc(i32 8)
        %3 = bitcast i8* %2 to %Exception*
        %4 = getelementptr [30 x i8]* @.message1, i32 0, i32 0
        call void @Exception_Create_String(%Exception* %3, i8* %4)
        ret %Exception* %3

.if_close:
        ; foo.SetLength(24)
        call void @Foo_SetLength(%Foo* %foo, i32 24)
        %5 = call i32 @Foo_GetLength(%Foo* %foo)
        store i32 %5, i32* %result
        ret %Exception* null
}

;******************************* Main program ************************

@.message2 = internal constant [11 x i8] c"Error: %s\0A\00"
@.message3 = internal constant [44 x i8] c"Internal error: Unhandled exception␣
↪detectd\00"

define i32 @main(i32 %argc, i8** %argv) nounwind {
        ; "try" keyword expands to nothing.

        ; Body of try block.
```

(continues on next page)

(continued from previous page)

```
        ; fail = (argc >= 2)
        %fail = icmp uge i32 %argc, 2

        ; Function call.
        %1 = alloca i32
        %2 = call %Exception* @Bar(i1 %fail, i32* %1)
        %3 = icmp ne %Exception* %2, null
        br i1 %3, label %.catch_block, label %.exit

.catch_block:
        %4 = bitcast %Exception* %2 to %Object*
        %5 = getelementptr [10 x i8]* @.Exception_class_name, i32 0, i32 0
        %6 = call i1 @Object_IsA(%Object* %4, i8* %5)
        br i1 %6, label %.catch_exception, label %.catch_all

.catch_exception:
        %7 = getelementptr [11 x i8]* @.message2, i32 0, i32 0
        %8 = call i8* @Exception_GetText(%Exception* %2)
        %9 = call i32 (i8*, ...)* @printf(i8* %7, i8* %8)
        br label %.exit

.catch_all:
        %10 = getelementptr [44 x i8]* @.message3, i32 0, i32 0
        %11 = call i32 @puts(i8* %10)
        br label %.exit

.exit:
        %result = phi i32 [ 0, %0 ], [ 1, %.catch_exception ], [ 1, %.catch_all ]
        ret i32 %result
}
```

3.5.2 Setjmp/Longjmp Exception Handling

The basic idea behind the `setjmp` and `longjmp` exception handling scheme is that you save the CPU state whenever you encounter a `try` keyword and then do a `longjmp` whenever you throw an exception. If there are few `try` blocks in the program, as is typically the case, the cost of this method is not as high as it might seem. However, often there are implicit exception handlers due to the need to release local resources such as class instances allocated on the stack and then the cost can become quite high.

`setjmp`/`longjmp` exception handling is often abbreviated `SjLj` for `SetJmp`/`LongJmp`.

The sample translates into something like this:

```
; jmp_buf is very platform specific, this is for illustration only...
%jmp_buf = type { i32 }
declare i32 @setjmp(%jmp_buf* %env)
declare void @longjmp(%jmp_buf* %env, i32 %val)

;******************** External and Utility functions ********************

declare i8* @malloc(i32) nounwind
declare void @free(i8*) nounwind
declare i32 @printf(i8* noalias nocapture, ...) nounwind
declare i32 @puts(i8* noalias nocapture) nounwind
```

(continues on next page)

(continued from previous page)

```
;***************************** Object class ***************************

%Object_vtable_type = type {
        %Object_vtable_type*,          ; 0: above: parent class vtable pointer
        i8*                            ; 1: class: class name (usually mangled)
        ; virtual methods would follow here
}

@.Object_class_name = private constant [7 x i8] c"Object\00"

@.Object_vtable = private constant %Object_vtable_type {
        %Object_vtable_type* null,     ; This is the root object of the object␣
↪hierarchy
        i8* getelementptr([7 x i8]* @.Object_class_name, i32 0, i32 0)
}

%Object = type {
        %Object_vtable_type*           ; 0: vtable: class vtable pointer (always non-
↪null)
        ; class data members would follow here
}

; returns true if the specified object is identical to or derived from the
; class with the specified name.
define i1 @Object_IsA(%Object* %object, i8* %name) nounwind {
.init:
        ; if (object == null) return false
        %0 = icmp ne %Object* %object, null
        br i1 %0, label %.once, label %.exit_false

.once:
        %1 = getelementptr %Object* %object, i32 0, i32 0
        br label %.body

.body:
        ; if (vtable->class == name)
        %2 = phi %Object_vtable_type** [ %1, %.once ], [ %7, %.next]
        %3 = load %Object_vtable_type** %2
        %4 = getelementptr %Object_vtable_type* %3, i32 0, i32 1
        %5 = load i8** %4
        %6 = icmp eq i8* %5, %name
        br i1 %6, label %.exit_true, label %.next

.next:
        ; object = object->above
        %7 = getelementptr %Object_vtable_type* %3, i32 0, i32 0

        ; while (object != null)
        %8 = icmp ne %Object_vtable_type* %3, null
        br i1 %8, label %.body, label %.exit_false

.exit_true:
        ret i1 true

.exit_false:
        ret i1 false
```

(continues on next page)

(continued from previous page)

```
}

;************************** Exception class ******************************

%Exception_vtable_type = type {
        %Object_vtable_type*,                           ; 0: parent class vtable pointer
        i8*                                             ; 1: class name
        ; virtual methods would follow here.
}

@.Exception_class_name = private constant [10 x i8] c"Exception\00"

@.Exception_vtable = private constant %Exception_vtable_type {
        %Object_vtable_type* @.Object_vtable,           ; the parent of this class is␣
↪the Object class
        i8* getelementptr([10 x i8]* @.Exception_class_name, i32 0, i32 0)
}

%Exception = type {
        %Exception_vtable_type*,                        ; 0: the vtable pointer
        i8*                                             ; 1: the _text member
}

define void @Exception_Create_String(%Exception* %this, i8* %text) nounwind {
        ; set up vtable
        %1 = getelementptr %Exception* %this, i32 0, i32 0
        store %Exception_vtable_type* @.Exception_vtable, %Exception_vtable_type** %1

        ; save input text string into _text
        %2 = getelementptr %Exception* %this, i32 0, i32 1
        store i8* %text, i8** %2

        ret void
}

define i8* @Exception_GetText(%Exception* %this) nounwind {
        %1 = getelementptr %Exception* %this, i32 0, i32 1
        %2 = load i8** %1
        ret i8* %2
}

;****************************** Foo class ********************************

%Foo = type { i32 }

define void @Foo_Create_Default(%Foo* %this) nounwind {
        %1 = getelementptr %Foo* %this, i32 0, i32 0
        store i32 0, i32* %1
        ret void
}

define i32 @Foo_GetLength(%Foo* %this) nounwind {
        %1 = getelementptr %Foo* %this, i32 0, i32 0
        %2 = load i32* %1
        ret i32 %2
}
```

(continues on next page)

(continued from previous page)

```
define void @Foo_SetLength(%Foo* %this, i32 %value) nounwind {
        %1 = getelementptr %Foo* %this, i32 0, i32 0
        store i32 %value, i32* %1
        ret void
}

;****************************** Foo function ************************

@.message1 = internal constant [30 x i8] c"Exception requested by caller\00"

define i32 @Bar(%jmp_buf* %throw, i1 %fail) nounwind {
        ; Allocate Foo instance
        %foo = alloca %Foo
        call void @Foo_Create_Default(%Foo* %foo)

        call void @Foo_SetLength(%Foo* %foo, i32 17)

        ; if (fail)
        %1 = icmp eq i1 %fail, true
        br i1 %1, label %.if_begin, label %.if_close

.if_begin:
        ; throw new Exception(...)
        %2 = call i8* @malloc(i32 8)
        %3 = bitcast i8* %2 to %Exception*
        %4 = getelementptr [30 x i8]* @.message1, i32 0, i32 0
        call void @Exception_Create_String(%Exception* %3, i8* %4)
        %5 = ptrtoint %Exception* %3 to i32
        call void @longjmp(%jmp_buf* %throw, i32 %5)
        ; we never get here
        br label %.if_close

.if_close:
        ; foo.SetLength(24)
        call void @Foo_SetLength(%Foo* %foo, i32 24)
        %6 = call i32 @Foo_GetLength(%Foo* %foo)
        ret i32 %6
}

;****************************** Main program ************************

@.message2 = internal constant [11 x i8] c"Error: %s\0A\00"
@.message3 = internal constant [44 x i8] c"Internal error: Unhandled exception␣
↪detectd\00"

define i32 @main(i32 %argc, i8** %argv) nounwind {
        ; "try" keyword expands to a call to @setjmp
        %env = alloca %jmp_buf
        %status = call i32 @setjmp(%jmp_buf* %env)
        %1 = icmp eq i32 %status, 0
        br i1 %1, label %.body, label %.catch_block

.body:
        ; Body of try block.
        ; fail = (argc >= 2)
        %fail = icmp uge i32 %argc, 2
```

(continues on next page)

(continued from previous page)

```
        ; Function call.
        %2 = call i32 @Bar(%jmp_buf* %env, i1 %fail)
        br label %.exit

.catch_block:
        %3 = inttoptr i32 %status to %Object*
        %4 = getelementptr [10 x i8]* @.Exception_class_name, i32 0, i32 0
        %5 = call i1 @Object_IsA(%Object* %3, i8* %4)
        br i1 %5, label %.catch_exception, label %.catch_all

.catch_exception:
        %6 = inttoptr i32 %status to %Exception*
        %7 = getelementptr [11 x i8]* @.message2, i32 0, i32 0
        %8 = call i8* @Exception_GetText(%Exception* %6)
        %9 = call i32 (i8*, ...)* @printf(i8* %7, i8* %8)
        br label %.exit

.catch_all:
        %10 = getelementptr [44 x i8]* @.message3, i32 0, i32 0
        %11 = call i32 @puts(i8* %10)
        br label %.exit

.exit:
        %result = phi i32 [ 0, %.body ], [ 1, %.catch_exception ], [ 1, %.catch_all ]
        ret i32 %result
}
```

3.5.3 Zero Cost Exception Handling

3.5.4 Resources

- Compiler Internals - Exception Handling.
- Exception Handling in C without C++.
- How a C++ Compiler Implements Exception Handling.
- DWARF standard - Exception Handling.
- Itanium C++ ABI.

3.6 Advanced/Functional Constructs

In this chapter, we'll look at various non-OOP constructs that are highly useful and are becoming more and more widespread in use.

3.6.1 Lambda Functions

A lambda function is an anonymous function with the added spice that it may freely refer to the local variables (including argument variables) in the containing function. Lambdas are implemented just like Pascal's nested functions, except the compiler is responsible for generating an internal name for the lambda function. There are a few different ways of implementing lambda functions (see Wikipedia on Nested Functions for more information).

```
int foo(int a)
{
  auto function = [a](int x) { return x + a; };
  return function(10);
}
```

Here the "problem" is that the lambda function references a local variable of the caller, namely `a`, even though the lambda function is a function of its own. This can be solved easily by passing the local variable in as an implicit argument to the lambda function:

```
define internal i32 @lambda(i32 %a, i32 %x) {
        %1 = add i32 %a, %x
        ret i32 %1
}

define i32 @foo(i32 %a) {
        %1 = call i32 @lambda(i32 %a, i32 10)
        ret i32 %1
}
```

Alternatively, if the lambda function uses more than a few variables, you can wrap them up in a structure which you pass in a pointer to the lambda function:

```
extern int integer_parse();

int foo(int a, int b)
{
  int c = integer_parse();
  auto function = [a, b, c](int x) { return (a + b - c) * x; };
  return function(10);
}
```

Becomes:

```
; ModuleID = 'lambda_func_1_cleaned.ll'
source_filename = "lambda_func_1_cleaned.ll"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

%lambda_args = type { i32, i32, i32 }

declare i32 @integer_parse()

define i32 @lambda(%lambda_args* %args, i32 %x) {
  %1 = getelementptr %lambda_args, %lambda_args* %args, i32 0, i32 0
  %a = load i32, i32* %1
  %2 = getelementptr %lambda_args, %lambda_args* %args, i32 0, i32 1
  %b = load i32, i32* %2
  %3 = getelementptr %lambda_args, %lambda_args* %args, i32 0, i32 2
  %c = load i32, i32* %3
  %4 = add i32 %a, %b
  %5 = sub i32 %4, %c
  %6 = mul i32 %5, %x
  ret i32 %6
}

define i32 @foo(i32 %a, i32 %b) {
```

(continues on next page)

(continued from previous page)

```
  %args = alloca %lambda_args
  %1 = getelementptr %lambda_args, %lambda_args* %args, i32 0, i32 0
  store i32 %a, i32* %1
  %2 = getelementptr %lambda_args, %lambda_args* %args, i32 0, i32 1
  store i32 %b, i32* %2
  %c = call i32 @integer_parse()
  %3 = getelementptr %lambda_args, %lambda_args* %args, i32 0, i32 2
  store i32 %c, i32* %3
  %4 = call i32 @lambda(%lambda_args* %args, i32 10)
  ret i32 %4
}
```

Obviously there are some possible variations over this theme:

- You could pass all implicit as explicit arguments as arguments.
- You could pass all implicit as explicit arguments in the structure.
- You could pass in a pointer to the frame of the caller and let the lambda function extract the arguments and locals from the input frame.

3.6.2 Generators

A generator is a function that repeatedly yields a value in such a way that the function's state is preserved across the repeated calls of the function; this includes the function's local offset at the point it yielded a value.

The most straigthforward way to implement a generator is by wrapping all of its state variables (arguments, local variables, and return values) up into an ad-hoc structure and then pass the address of that structure to the generator.

Somehow, you need to keep track of which block of the generator you are doing on each call. This can be done in various ways; the way we show here is by using LLVM's `blockaddress` instruction to save the address of the next local block of code that should be executed. Other implementations use a simple state variable and then do a `switch`-like dispatch according to the value of the state variable. In both cases, the end result is the same: A different block of code is executed for each local block in the generator.

The important thing is to think of iterators as a sort of micro-thread that is resumed whenever the iterator is called again. In other words, we need to save the address of how far the iterator got on each pass through so that it can resume as if a microscopic thread switch had occured. So we save the address of the instruction after the return instruction so that we can resume running as if we never had returned in the first place.

I resort to pseudo-C++ because C++ does not directly support generators. First we look at a very simple case then we advance on to a slightly more complex case:

```
#include <stdio.h>

generator int foo()
{
    yield 1;
    yield 2;
    yield 3;
}

int main()
{
    foreach (int i in foo())
        printf("Value: %d\n", i);
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

This becomes:

; Compiled and run successfully against LLVM v3.4 on 2013.12.06.

```
%foo_context = type {
    i8*,      ; 0: block (state)
    i32       ; 1: value (result)
}

define void @foo_setup(%foo_context* %context) nounwind {
    ; set up 'block'
    %1 = getelementptr %foo_context* %context, i32 0, i32 0
    store i8* blockaddress(@foo_yield, %.yield1), i8** %1

    ret void
}

; The boolean returned indicates if a result was available or not.
; Once no more results are available, the caller is expected to not call
; the iterator again.
define i1 @foo_yield(%foo_context* %context) nounwind {
    ; dispatch to the active generator block
    %1 = getelementptr %foo_context* %context, i32 0, i32 0
    %2 = load i8** %1
    indirectbr i8* %2, [ label %.yield1, label %.yield2, label %.yield3, label %.done␣
↪]

.yield1:
    ; store the result value (1)
    %3 = getelementptr %foo_context* %context, i32 0, i32 1
    store i32 1, i32* %3

    ; make 'block' point to next block to execute
    %4 = getelementptr %foo_context* %context, i32 0, i32 0
    store i8* blockaddress(@foo_yield, %.yield2), i8** %4

    ret i1 1

.yield2:
    ; store the result value (2)
    %5 = getelementptr %foo_context* %context, i32 0, i32 1
    store i32 2, i32* %5

    ; make 'block' point to next block to execute
    %6 = getelementptr %foo_context* %context, i32 0, i32 0
    store i8* blockaddress(@foo_yield, %.yield3), i8** %6

    ret i1 1

.yield3:
    ; store the result value (3)
    %7 = getelementptr %foo_context* %context, i32 0, i32 1
    store i32 3, i32* %7
```

(continues on next page)

(continued from previous page)

```
    ; make 'block' point to next block to execute
    %8 = getelementptr %foo_context* %context, i32 0, i32 0
    store i8* blockaddress(@foo_yield, %.done), i8** %8

    ret i1 1

.done:
    ret i1 0
}

declare i32 @printf(i8*, ...) nounwind

@.string = internal constant [11 x i8] c"Value: %d\0A\00"

define void @main() nounwind {
    ; allocate and initialize generator context structure
    %context = alloca %foo_context
    call void @foo_setup(%foo_context* %context)
    br label %.head

.head:
    ; foreach (int i in foo())
    %1 = call i1 @foo_yield(%foo_context* %context)
    br i1 %1, label %.body, label %.tail

.body:
    %2 = getelementptr %foo_context* %context, i32 0, i32 1
    %3 = load i32* %2
    %4 = call i32 (i8*, ...)* @printf(i8* getelementptr([11 x i8]* @.string, i32 0,␣
↪i32 0), i32 %3)
    br label %.head

.tail:
    ret void
}
```

And now for a slightly more complex example that involves local variables:

```
#include <stdio.h>

generator int foo(int start, int after)
{
    for (int index = start; index < after; index++)
    {
        if (index % 2 == 0)
            yield index + 1;
        else
            yield index - 1;
    }
}

int main(void)
{
    foreach (int i in foo(0, 5))
        printf("Value: %d\n", i);

    return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

This becomes something like this:

; Compiled and run successfully against LLVM v3.4 on 2013.12.06.

```
%foo_context = type {
    i8*,       ; 0: block (state)
    i32,       ; 1: start (argument)
    i32,       ; 2: after (argument)
    i32,       ; 3: index (local)
    i32        ; 4: value (result)
}

define void @foo_setup(%foo_context* %context, i32 %start, i32 %after) nounwind {
    ; set up 'block'
    %1 = getelementptr %foo_context* %context, i32 0, i32 0
    store i8* blockaddress(@foo_yield, %.init), i8** %1

    ; set up 'start'
    %2 = getelementptr %foo_context* %context, i32 0, i32 1
    store i32 %start, i32* %2

    ; set up 'after'
    %3 = getelementptr %foo_context* %context, i32 0, i32 2
    store i32 %after, i32* %3

    ret void
}

define i1 @foo_yield(%foo_context* %context) nounwind {
    ; dispatch to the active generator block
    %1 = getelementptr %foo_context* %context, i32 0, i32 0
    %2 = load i8** %1
   indirectbr i8* %2, [ label %.init, label %.loop_close, label %.end ]

.init:
    ; copy argument 'start' to the local variable 'index'
    %3 = getelementptr %foo_context* %context, i32 0, i32 1
    %start = load i32* %3
    %4 = getelementptr %foo_context* %context, i32 0, i32 3
    store i32 %start, i32* %4
    br label %.head

.head:
    ; for (; index < after; )
    %5 = getelementptr %foo_context* %context, i32 0, i32 3
    %index = load i32* %5
    %6 = getelementptr %foo_context* %context, i32 0, i32 2
    %after = load i32* %6
    %again = icmp slt i32 %index, %after
    br i1 %again, label %.loop_begin, label %.exit

.loop_begin:
    %7 = srem i32 %index, 2
    %8 = icmp eq i32 %7, 0
    br i1 %8, label %.even, label %.odd
```

(continues on next page)

(continued from previous page)

```
.even:
    ; store 'index + 1' in 'value'
    %9 = add i32 %index, 1
    %10 = getelementptr %foo_context* %context, i32 0, i32 4
    store i32 %9, i32* %10

    ; make 'block' point to the end of the loop (after the yield)
    %11 = getelementptr %foo_context* %context, i32 0, i32 0
    store i8* blockaddress(@foo_yield, %.loop_close), i8** %11

    ret i1 1

.odd:
    ; store 'index - 1' in value
    %12 = sub i32 %index, 1
    %13 = getelementptr %foo_context* %context, i32 0, i32 4
    store i32 %12, i32* %13

    ; make 'block' point to the end of the loop (after the yield)
    %14 = getelementptr %foo_context* %context, i32 0, i32 0
    store i8* blockaddress(@foo_yield, %.loop_close), i8** %14

    ret i1 1

.loop_close:
    ; increment 'index'
    %15 = getelementptr %foo_context* %context, i32 0, i32 3
    %16 = load i32* %15
    %17 = add i32 %16, 1
    store i32 %17, i32* %15
    br label %.head

.exit:
    ; make 'block' point to the %.end label
    %x = getelementptr %foo_context* %context, i32 0, i32 0
    store i8* blockaddress(@foo_yield, %.end), i8** %x
    br label %.end

.end:
    ret i1 0
}

declare i32 @printf(i8*, ...) nounwind

@.string = internal constant [11 x i8] c"Value: %d\0A\00"

define i32 @main() nounwind {
    ; allocate and initialize generator context structure
    %context = alloca %foo_context
    call void @foo_setup(%foo_context* %context, i32 0, i32 5)
    br label %.head

.head:
    ; foreach (int i in foo(0, 5))
    %1 = call i1 @foo_yield(%foo_context* %context)
    br i1 %1, label %.body, label %.tail
```

(continues on next page)

(continued from previous page)

```
.body:
    %2 = getelementptr %foo_context* %context, i32 0, i32 4
    %3 = load i32* %2
    %4 = call i32 (i8*, ...)* @printf(i8* getelementptr([11 x i8]* @.string, i32 0,␣
↪i32 0), i32 %3)
    br label %.head

.tail:
    ret i32 0
}
```

Another possible way of doing the above would be to generate an LLVM IR function for each state and then store a function pointer

in the context structure, which is updated whenever a new state/function needs to be invoked.

3.7 Interoperating with a Runtime Library

It is common to provide a set of run-time support functions that are written in another language than LLVM IR and it is trivially easy to interface to such a run-time library. The use of `malloc` and `free` in the examples in this document are examples of such use of externally defined run-time functions.

The advantages of a custom, non-IR run-time library function is that it can be optimized by hand to provide the best possible performance under certain criteria. Also a custom non-IR run-time library function can make explicit use of native instructions that are foreign to the LLVM infrastructure.

The advantages of IR run-time library functions is that they can be run through the optimizer and thereby also be inlined automatically.

3.8 Interfacing to Operating Systems

I'll divide this chapter into two sections:

- How to Interface to POSIX Operating Systems.
- How to Interface to the Windows Operating System.

3.8.1 Interface to POSIX Operating Systems

On POSIX, the presence of the C run-time library is an unavoidable fact for which reason it makes a lot of sense to directly call such C run-time functions.

Sample POSIX "Hello World" Application

On POSIX, it is really very easy to create the `Hello world` program:

```
declare i32 @puts(i8* nocapture) nounwind

@.hello = private unnamed_addr constant [13 x i8] c"hello world\0A\00"
```

(continues on next page)

(continued from previous page)

```
define i32 @main(i32 %argc, i8** %argv) {
    %1 = getelementptr [13 x i8]* @.hello, i32 0, i32 0
    call i32 @puts(i8* %1)
    ret i32 0
}
```

3.8.2 How to Interface to the Windows Operating System

On Windows, the C run-time library is mostly considered of relevance to the C and C++ languages only, so you have a plethora (thousands) of standard system interfaces that any client application may use.

Sample Windows "Hello World" Application

`Hello world` on Windows is nowhere as straightforward as on POSIX:

```
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
↪f32:32:32-f64:64:64-f80:128:128-v64:64:64-v128:128:128-a0:0:64-f80:32:32-n8:16:32-
↪S32"
target triple = "i686-pc-win32"

%struct._OVERLAPPED = type { i32, i32, %union.anon, i8* }
%union.anon = type { %struct.anon }
%struct.anon = type { i32, i32 }

declare dllimport x86_stdcallcc i8* @"\01_GetStdHandle@4"(i32) #1

declare dllimport x86_stdcallcc i32 @"\01_WriteFile@20"(i8*, i8*, i32, i32*, %struct._
↪OVERLAPPED*) #1

@hello = internal constant [13 x i8] c"Hello world\0A\00"

define i32 @main(i32 %argc, i8** %argv) nounwind {
    %1 = call i8* @"\01_GetStdHandle@4"(i32 -11)     ; -11 = STD_OUTPUT_HANDLE
    %2 = getelementptr [13 x i8]* @hello, i32 0, i32 0
    %3 = call i32 @"\01_WriteFile@20"(i8* %1, i8* %2, i32 12, i32* null, %struct._
↪OVERLAPPED* null)
    ; todo: Check that %4 is not equal to -1 (INVALID_HANDLE_VALUE)
    ret i32 0
}

attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-
↪frame-pointer-elim-non-leaf"
"no-infs-fp-math"="fa lse" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8
↪" "unsafe-fp-math"="false"
"use-soft-float"="false"
}
```

3.9 Epilogue

Remember that you can learn a lot by using the `-emit-llvm` option to the `clang`/`clang++` compiler. This gives you a chance to see a live production compiler in action and how it precisely does things.

If you discover something new this way, or any errors in this document or you need more information than given here, please create an issue at Github

3.9.1 Further Reading

This chapter lists some resources that may be of interest to the reader:

- LLVM documentation http://llvm.org/docs/
- Modern Compiler Implementation in Java, 2nd Edition.
- Eli Bendersky's collection of code examples for using LLVM/clang, https://github.com/eliben/llvm-clang-samples
- "How Clang Compiles a Function" by John Regehr, June 2018, https://blog.regehr.org/archives/1605

3.9.2 Contribution

If you want to contribute to this document you can:

- Create an issue on Github
- Fork and fix it yourself. We're happy to merge any pull requests or accept patches.

Thank You

A brief list of all people who've contributed to the document (thank you all very much!):

- Michael Rodler (current maintainer and back-porter from MeWeb markup to GitHub markdown).
- Mikael Egevig (original author of the document itself - under the name of Mikael Lyngvig).
- Dirkjan Ochtman (basically all the generator-related stuff by giving me some crucial samples).
- Eli Bendersky (for small grammar fixes and mention of `opt`'s `.bc` output).
- Sean Silva (for using proper C++11 lambda syntax in lambda samples).
- Isaac Dupree (for correction the name of '@': It is "at-sign", not "ampersand").
- Wilfred Hughes (`i` became `index` and addition of separator between generator C++ and LLVM IR code).
- Kenneth Ho (correction to C++11 lambda example and C++ exception handling sample).

Should your name be here? If so, contribute :-)

3.10 How to Implement a String Type in LLVM

There are two ways to implement a string type in LLVM:

- To write the implementation in LLVM IR.
- To write the implementation in a higher-level language that generates IR.

I'd personally much prefer to use the second method, but for the sake of the example, I'll go ahead and illustrate a simple but useful string type in LLVM IR. It assumes a 32-bit architecture, so please replace all occurences of `i32` with `i64` if you are targetting a 64-bit architecture.

We'll be making a dynamic, mutable string type that can be appended to and could also be inserted into, converted to lower case, and so on, depending on which support functions are defined to operate on the string type.

It all boils down to making a suitable type definition for the class and then define a rich set of functions to operate on the type definition:

```
; The actual type definition for our 'String' type.
%String = type {
    i8*,      ; 0: buffer; pointer to the character buffer
    i32,      ; 1: length; the number of chars in the buffer
    i32,      ; 2: maxlen; the maximum number of chars in the buffer
    i32       ; 3: factor; the number of chars to preallocate when growing
}

define fastcc void @String_Create_Default(%String* %this) nounwind {
    ; Initialize 'buffer'.
    %1 = getelementptr %String* %this, i32 0, i32 0
    store i8* null, i8** %1

    ; Initialize 'length'.
    %2 = getelementptr %String* %this, i32 0, i32 1
    store i32 0, i32* %2

    ; Initialize 'maxlen'.
    %3 = getelementptr %String* %this, i32 0, i32 2
    store i32 0, i32* %3

    ; Initialize 'factor'.
    %4 = getelementptr %String* %this, i32 0, i32 3
    store i32 16, i32* %4

    ret void
}

declare i8* @malloc(i32)
declare void @free(i8*)
declare i8* @memcpy(i8*, i8*, i32)

define fastcc void @String_Delete(%String* %this) nounwind {
    ; Check if we need to call 'free'.
    %1 = getelementptr %String* %this, i32 0, i32 0
    %2 = load i8** %1
    %3 = icmp ne i8* %2, null
    br i1 %3, label %free_begin, label %free_close

free_begin:
    call void @free(i8* %2)
    br label %free_close

free_close:
    ret void
}

define fastcc void @String_Resize(%String* %this, i32 %value) {
    ; %output = malloc(%value)
    %output = call i8* @malloc(i32 %value)

    ; todo: check return value
```

(continues on next page)

(continued from previous page)

```
    ; %buffer = this->buffer
    %1 = getelementptr %String* %this, i32 0, i32 0
    %buffer = load i8** %1

    ; %length = this->length
    %2 = getelementptr %String* %this, i32 0, i32 1
    %length = load i32* %2

    ; memcpy(%output, %buffer, %length)
    %3 = call i8* @memcpy(i8* %output, i8* %buffer, i32 %length)

    ; free(%buffer)
    call void @free(i8* %buffer)

    ; this->buffer = %output
    store i8* %output, i8** %1

    ret void
}

define fastcc void @String_Add_Char(%String* %this, i8 %value) {
    ; Check if we need to grow the string.
    %1 = getelementptr %String* %this, i32 0, i32 1
    %length = load i32* %1
    %2 = getelementptr %String* %this, i32 0, i32 2
    %maxlen = load i32* %2
    ; if length == maxlen:
    %3 = icmp eq i32 %length, %maxlen
    br i1 %3, label %grow_begin, label %grow_close

grow_begin:
    %4 = getelementptr %String* %this, i32 0, i32 3
    %factor = load i32* %4
    %5 = add i32 %maxlen, %factor
    call void @String_Resize(%String* %this, i32 %5)
    br label %grow_close

grow_close:
    %6 = getelementptr %String* %this, i32 0, i32 0
    %buffer = load i8** %6
    %7 = getelementptr i8* %buffer, i32 %length
    store i8 %value, i8* %7
    %8 = add i32 %length, 1
    store i32 %8, i32* %1

    ret void
}
```