Targeting LLVM IR

LLVM IR, code emission, assignment 4

LLVM Overview

- Common set of tools & optimizations for compiling many languages to many architectures (x86, ARM, PPC, ASM.js).

- Integrates AOT & JIT compilation, VM, lifelong optimization.

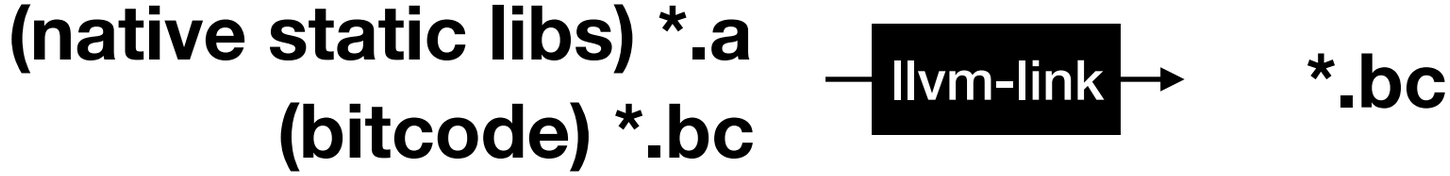
- History: Chris Lattner at UIUC in 2000 (hired by Apple 2005).

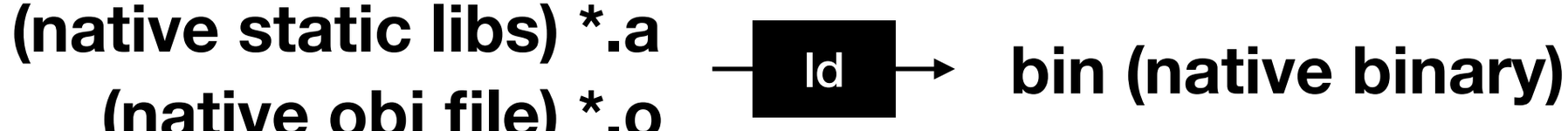
- Three IR formats: Text (.ll), bitcode (.bc), and in-memory representations of programs.

- Infinite register set, programs in SSA form, strongly typed IR.

- 40+ common optimization passes; extensible in C++.

LLVM Overview





Overview of an IR file

```
target datalayout = "e-m:o-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

%struct.A = type { i64, i32 }

@five = global i64 5, align 8
@hello = global [6 x i8] c"hello\00", align 8

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

define i32 @main(i32 %argc, i8** %argv) {
  %a = alloca %struct.A, align 8
  ; …
  ret i32 0
}
```

Overview of an IR file

```
target datalayout = "e-m:o-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"
```

```
%struct.A = type { i64, i32 }

@five = global i64 5, align 8
@hello = global [6 x i8] c"hello\00", align 8

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

define i32 @main(i32 %argc, i8** %argv) {
  %a = alloca %struct.A, align 8
  ; …
  ret i32 0
}
```

Overview of an IR file

```
target datalayout = "e-m:o-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

%struct.A = type { i64, i32 }

@five = global i64 5, align 8
@hello = global [6 x i8] c"hello\00", align 8

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

define i32 @main(i32 %argc, i8** %argv) {
  %a = alloca %struct.A, align 8
  ; …
  ret i32 0
}
```

Overview of an IR file

```
target datalayout = "e-m:o-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

%struct.A = type { i64, i32 }

@five = global i64 5, align 8
@hello = global [6 x i8] c"hello\00", align 8

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

define i32 @main(i32 %argc, i8** %argv) {
  %a = alloca %struct.A, align 8
  ; …
  ret i32 0
}
```

Overview of an IR file

```
target datalayout = "e-m:o-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

%struct.A = type { i64, i32 }

@five = global i64 5, align 8
@hello = global [6 x i8] c"hello\00", align 8

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

define i32 @main(i32 %argc, i8** %argv) {
  %a = alloca %struct.A, align 8
  ; …
  ret i32 0
}
```

Overview of an IR file

```
target datalayout = "e-m:o-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

%struct.A = type { i64, i32 }

@five = global i64 5, align 8
@hello = global [6 x i8] c"hello\00", align 8

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

define i32 @main(i32 %argc, i8** %argv) {
  %a = alloca %struct.A, align 8
  ; …
  ret i32 0
}
```

Types and casts

```
T ::= i1 | i8 | … | i32 | i64 | …
    | half | float | double | fp128
    | void | label
    | T* | T (T,…)*
    | {T, …} | [N x T]
```

- There is no `void*` as there is in C.
- More types we wont cover specifically: address spaces, vector (SIMD) types, opaque types, packed-struct types.
- You may declare named struct/record types at the top-level:

```
%TypeA = type { T, … }
```

Types and casts

```
%r = bitcast T0 %val to T1
%r = bitcast T0* %val to T1*
```

Works as reinterpret_cast<T_1>(%val) does in C++. Must take a first-class, non-aggregate type. Cannot convert a pointer to a non-pointer value.

```
%r = inttoptr i64 %val to T*
%r = ptrtoint T* %val to i64
```

Reinterpret pointers as integers and vice versa.

load, store, getelementptr

```
%val = load T, T* %ptr, align N
```

```
store T %val, T* %ptr, align N
```

load, store, getelementptr

```
%eptr = getelementptr T, T* %ptr, T %index
```

Roughly equivalent to: `eptr = &(ptr[index])`

In C/C++, this is an implicit operation:

```
T* arr;
// …
T v = arr[i];
```

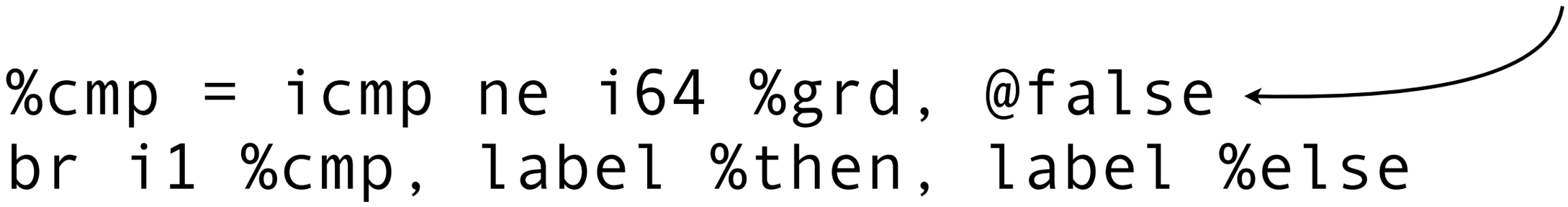
```
T* arr;
// …
T v = *(arr+i);
T v = *((T*)(((char*)arr)
                + i*sizeof(T));
```

Stack allocation

```
define i32 main(i32 %a, i8** %b) {
  %1 = alloca i64, align 8 ; returns i64*
  %2 = alloca i32, align 4 ; returns i32*
  store i32 %a, i32* %2, align 4
  ; ...
}
```

Branching

When translating `(if grd then else)`, compare grd to the value for `#f`



```
%cmp = icmp ne i64 %grd, @false
br i1 %cmp, label %then, label %else

then:
  %r0 = add i64 %x, %y
  ...

else:
  %r1 = sub i64 %w, %z
  ...
```

Phi nodes

Can only occur at the front of basic blocks. Lists some number of values, each paired with the label for its corresponding predecessor block.

(You can allow the LLVM analysis/optimization phase to add phi nodes.)

```
entry:
  ; ...

loop:
  %x = phi i64, [0 %entry], [%r %loop]
  %r = add i64 %x, %r
  ; ...
  br label %loop
```

Function calls / returns

```
%r = call T @fn(T %val, …)
```

```
tail call fastcc void %fn(T %val, …)
```

```
ret i64 %val
```

```
ret void
```

Notes on tail-call optimization

Notes from the language reference:

> Tail call optimization for calls marked `tail` is guaranteed to occur if the following conditions are met:
>
> - Caller and callee both have the calling convention fastcc.
> - The call is in tail position (ret immediately follows call and ret uses value of call or is void).
> - Option -tailcallopt is enabled, or llvm::GuaranteedTailCallOpt is true.
> - Platform-specific constraints are met.

The `musttail` marker means that the call must be tail call optimized in order for the program to be correct. The musttail marker provides these guarantees:

1. The call will not cause unbounded stack growth if it is part of a recursive cycle in the call graph.
2. Arguments with the inalloca attribute are forwarded in place.

Learning IR?

Three of the best ways:

1) Check the reference:

https://llvm.org/docs/LangRef.html

2) Use clang to compile C/C++

```
clang++ main.cpp -S -emit-llvm -o main.ll
```

(Also give godbolt.org a try)

3) Use clang to compile IR

```
clang++ main.ll -o main
```

3) Use clang to compile IR

```
clang++ main.ll -g -o main; gdb ./main
```

Assignment 4

- Two phases: `closure-convert` and `proc->llvm`
    - Closure convert: two helpers with most cases finished (simplify-ae, and remove-varags). Returns a proc-exp? program, a list of first-order procedures.
    - Procedural IR to LLVM IR: return a string encoding IR that may use any functions in header.cpp -> header.ll
- `(eval-llvm ll)` will concatenate `ll` with header.ll and write the result to combined.ll, which is then compiled and run.
- Prim ops require a fixed number of *tagged* i64 values and return a single (tagged) i64 value.
- When producing constants, use const_init_X from header.cpp

Assignment 4 (tagging)

```
u64 const_init_int(s32 a)
{
  return (((u64)((u32)a) << 32) | INT_TAG);
}
// …string, …symbol, …true, …false, …null
```

```
u64 prim__43(u64 a, u64 b)  // (prim-name '+)
{
  // assert that tags are correct
  s32 av = (s32)(((u64)a) >> 32);
  s32 bv = (s32)(((u64)a) >> 32);
  return (((u64)((u32)(av+bv)) << 32) | INT_TAG);
}
```

Assignment 4 (tagging)

```
… (let ([x ‘3])
    (let ([y ‘4])
      (let ([z (prim + x y)])
        (k z))))
```

```
; …
%x = call i64 const_init_int(i32 3);
%y = call i64 const_init_int(i32 4);
%z = call i64 prim__43(i64 %x, i64 %y);
; invoke closure encoded in %k on %k and %z
; …
```