Lecture 8

COMPILER DESIGN
Announcements

- HW3: LLVM lite
  - Available from next week
  - Due: Monday, Oct. 29th

START EARLY!!
TAGGED DATATYPES
C-style Enums / ML-style datatypes

• In C:

```c
enum Day {sun, mon, tue, wed, thu, fri, sat} today;
```

• In ML:

```ml
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

• Associate an integer tag with each case: sun = 0, mon = 1, ...
  – C lets programmers choose the tags

• ML datatypes can also carry data:

```ml
type foo = Bar of int | Baz of int * foo
```

• Representation: a foo value is a pointer to a pair: (tag, data)

• Example: tag(Bar) = 0, tag(Baz) = 1

```ml
[let f = Bar(3)] =

0 3
```

```ml
[let g = Baz(4, f)] =

1 4 f
```
Switch Compilation

• Consider the C statement:

```c
switch (e) {
    case sun: s1; break;
    case mon: s2; break;
    ...
    case sat: s3; break;
}
```

• How to compile this?
  – What happens if some of the break statements are omitted? (Control falls through to the next branch.)
Cascading ifs and Jumps

\[
\text{[switch(e) \{case tag1: s1; case tag2 s2; \ldots\}]} =
\]

- Each $\text{tag1} \ldots \text{tagN}$ is just a constant int tag value.

- Note: \[\text{break;}\] (within the switch branches) is:
  \[\text{br \%merge}\]

\[
\begin{align*}
\text{%tag} &= [e]; \\
\text{br label \%l1} \\
\text{l1: \%cmp1} &= \text{icmp eq \%tag, \$tag1} \\
\text{br \%cmp1 label \%b1, label \%merge} \\
\text{b1: [s1]} \\
\text{br label \%l2} \\
\text{l2: \%cmp2} &= \text{icmp eq \%tag, \$tag2} \\
\text{br \%cmp2 label \%b2, label \%merge} \\
\text{b2: [s2]} \\
\text{br label \%l3} \\
\text{...} \\
\text{lN: \%cmpN} &= \text{icmp eq \%tag, \$tagN} \\
\text{br \%cmpN label \%bN, label \%merge} \\
\text{bN: [sN]} \\
\text{br label \%merge}
\end{align*}
\]

merge:
Alternatives for Switch Compilation

• Nested if-then-else works OK in practice if # of branches is small
  – (e.g. < 16 or so).
• For more branches, use better data structures to organize the jumps:
  – Create a table of pairs (v1, branch_label) and loop through
  – Or, do binary search rather than linear search
  – Or, use a hash table rather than binary search
• One common case: the tags are dense in some range [min…max]
  – Let N = max – min
  – Create a branch table Branches[N] where Branches[i] = branch_label for tag i.
  – Compute tag = ⟦e⟧ and then do an \textit{indirect jump}: J Branches[tag]
• Common to use heuristics to combine these techniques.
ML-style Pattern Matching

- ML-style match statements are like C’s switch statements except:
  - Patterns can bind variables
  - Patterns can nest

- Compilation strategy:
  - “Flatten” nested patterns into matches against one constructor at a time.
  - Compile the match against the tags of the datatype as for C-style switches.
  - Code for each branch additionally must copy data from \[[e]\] to the variables bound in the patterns.
- There are many opportunities for optimization, many papers about “pattern-match compilation”
  - Many of these transformations can be done at the AST level

```
match e with
| Bar(z)  -> e1
| Baz(y, Bar(w)) -> e2
| _      -> e3

match e with
| Bar(z)  -> e1
| Baz(y, tmp) ->
  (match tmp with
   | Bar(w)  -> e2
   | Baz(_, _) -> e3)
```
DATATYPES IN THE LLVM IR
Structured Data in LLVM

- LLVM’s IR uses types to describe the structure of data.

\[
t ::=\begin{align*}
\text{void} & \quad \text{N-bit integers} \\
i1 & \mid i8 \mid i64 & \quad \text{arrays} \\
[<\#\text{elts}> \times t] & \\
fty & \quad \text{function types} \\
\{t_1, t_2, \ldots, t_n\} & \quad \text{structures} \\
t^* & \quad \text{pointers} \\
\%Tident & \quad \text{named (identified) type}
\end{align*}
\]

- `<#elts>` is an integer constant \(\geq 0\)
- Structure types can be named at the top level:

\[
\%T1 = \text{type} \{t_1, t_2, \ldots, t_n\}
\]

- Such structure types can be recursive
Example LL Types

- An array of 341 integers: \([ 341 \times i64 ]\)

- A two-dimensional array of integers: \([ 3 \times [ 4 \times i64 ] ]\)

- Structure for representing arrays with their length:
  \(\{ i64 , [0 \times i64] \}\)
  - There is no array-bounds check; the static type information is only used for calculating pointer offsets.

- C-style linked lists (declared at the top level):
  \(\%\text{Node} = \text{type} \{ i64, \%\text{Node}*\}\)

- Structs from the C program shown earlier:
  \(\%\text{Rect} = \{ \%\text{Point}, \%\text{Point}, \%\text{Point}, \%\text{Point} \}\)
  \(\%\text{Point} = \{ i64, i64 \}\)
LLVM provides the `getelementptr` instruction to compute pointer values:

- Given a pointer and a “path” through the structured data pointed to by that pointer, `getelementptr` computes an address.
- This is the abstract analog of the X86 LEA (load effective address). It does not access memory.
- It is a “type indexed” operation, since the size computations depend on the type.

Example: access the x component of the first point of a rectangle:

```plaintext
%tmp1 = getelementptr %Rect* %square, i32 0, i32 0
%tmp2 = getelementptr %Point* %tmp1, i32 0, i32 0
```
GEP Example*

```c
struct RT {
    int A;
    int B[10][20];
    int C;
}
struct ST {
    struct RT X;
    int Y;
    struct RT Z;
}
int *foo(struct ST *s) {
    return &s[1].Z.B[5][13];
}
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
    entry:
        %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
        ret i32* %arrayidx
}
```

1. %s is a pointer to an (array of) %ST structs, suppose the pointer value is ADDR
2. Compute the index of the 1st element by adding size_ty(%ST).
3. Compute the index of the Z field by adding size_ty(%RT) + size_ty(i32) to skip past X and Y.
4. Compute the index of the B field by adding size_ty(i32) to skip past A.
5. Index into the 2d array.

Final answer: ADDR + size_ty(%ST) + size_ty(%RT) + size_ty(i32) + size_ty(i32) + 5*20*size_ty(i32) + 13*size_ty(i32)

Zhendong Su    Compiler Design  *adapted from the LLVM documentation: see http://llvm.org/docs/LangRef.html#getelementptr-instruction
getelementptr

• GEP *never* dereferences the address it’s calculating:
  – GEP only produces pointers by doing arithmetic
  – It doesn’t actually traverse the links of a datastructure

• To index into a deeply nested structure, need to “follow the pointer” by loading from the computed pointer
  – See list.ll from HW3
1. Translate high level language types into an LLVM representation type.
   – For some languages (e.g. C) this process is straightforward
     • The translation simply uses platform-specific alignment and padding
   – For other languages, (e.g. OO languages) there might be a fairly complex elaboration.
     • e.g. for Ocaml, arrays types might be translated to pointers to length-indexed structs.

\[
\text{[int array]} = \{ \text{i32, } [0 x \text{i32}]\}^* 
\]

2. Translate accesses of the data into getelementptr operations:
   – e.g. for Ocaml array size access:

\[
\text{[length a]} = \%
\text{1} = \text{getelementptr } \{\text{i32, } [0\text{x i32}]\}^* \%a, \text{i32 }0, \text{i32 }0
\]
Bitcast

• What if the LLVM IR’s type system isn’t expressive enough?
  – e.g. if the source language has subtyping, perhaps due to inheritance
  – e.g. if the source language has polymorphic/generic types

• LLVM IR provides a **bitcast** instruction
  – This is a form of (potentially) unsafe cast. Misuse can cause serious bugs
    (segmentation faults, or silent memory corruption)

```assembly
%rect2 = type { i64, i64 }   ; two-field record
%rect3 = type { i64, i64, i64 }   ; three-field record

define @foo() {
  %1 = alloca %rect3   ; allocate a three-field record
  %2 = bitcast %rect3* %1 to %rect2*   ; safe cast
  %3 = getelementptr %rect2* %2, i32 0, i32 1   ; allowed
  ...
}
```
see HW3

LLVMLITE SPECIFICATION
Discussion: Defining a Language

• Premise: programming languages are purely ‘formal’ objects
  – We (as language designers) get to determine the meaning of the language constructs

• Question: How do we specify that meaning?

• Question: What are the properties of a good specification?

• Examples?
Approaches to Language Specification

• Implementation
  – It does what it does!

• Social
  – Authority figure says: “it means X”
  – English prose

• Technological
  – Multiple implementations
  – Reference interpreter
  – Test cases / Examples

• Translation
  – Semantics given in terms of (hopefully better specified) target

• Mathematical
  – “Informal” specifications
  – “Formal” specifications

Less “formal”: Techniques may miss problems in programs

This isn’t a tradeoff… all of these methods should be used.

Even the most “formal” can still have holes:
  • Did you prove the right thing?
  • Do your assumptions match reality?
  • Knuth. “Beware of bugs in the above code; I have only proved it correct, not tried it.”

More “formal”: eliminate with certainty as many problems as possible.
LLVMlite notes

• Real LLVM requires that constants appearing in getelementptr indexing into structs be declared with type i32:

```llvm
%struct = type { i64, [5 x i64], i64}
@gbl = global %struct {i64 1,
   [5 x i64] [i64 2, i64 3, i64 4, i64 5, i64 6], i64 7}

define void @foo() {
   %1 = getelementptr %struct* @gbl, i32 0, i32 0
   ...
}
```

• LLVMlite ignores the i32 annotation and treats these as i64 values
  – we keep the i32 annotation in the syntax to retain compatibility with the clang compiler
COMPILING LLVM LITE TO X86
Compiling LLVMlite Types to X86

- $\llbracket i1 \rrbracket, \llbracket i64 \rrbracket, \llbracket t* \rrbracket =$ quad word (8 bytes, 8-byte aligned)
- raw $i8$ values are not allowed (they must be manipulated via $i8*$)
- array and struct types are laid out sequentially in memory

- getelementptr computations must be relative to the LLVMlite size definitions
  - i.e. $\llbracket i1 \rrbracket =$ quad
Compiling LLVM locals

- How do we manage storage for each %uid defined by an LLVM instruction?

- Option 1:
  - Map each %uid to a x86 register
  - Efficient!
  - Difficult to do effectively: many %uid values, only 16 registers
  - We will see how to do this later in the semester

- Option 2:
  - Map each %uid to a stack-allocated space
  - Less efficient!
  - Simple to implement

- For HW3 we will follow Option 2
Other LLVMlite Features

• Globals
  – must use %rip relative addressing

• Calls
  – Follow x64 AMD ABI calling conventions
  – Should interoperate with C programs

• getelementptr
  – trickiest part
see HW3 and README

ll.ml, using main.native, clang, etc.

TOUR OF HW 3
// Extract LLVM-IR from C code – with optimizations
clang -S -emit-llvm -O3 -o file.ll file.c

// Extract LLVM-IR from C code – no optimization
clang -S -emit-llvm -O3 -o file.ll cile.c -Xclang -disable-llvm-passes

// View the CFG of a file
opt -view-cfg file.ll

// Name all instructions
opt -instnamer file.ll

// Compile .ll file to .o file
clang file.ll -c -o file.o

// Compile .ll file to executable
clang file.ll -o file.exe
Compiler Social

When: Thursday (today), 19:00

Where: CAB, E72

What: Socialize + Research

Topics:
- Compiler Support for Time Predictable Execution
  Bjoern Forsberg
- LLHD: An Intermediate Representation for Digital Hardware
  Fabian Schuiki

Sign up: meetup.com/llvm-compiler-and-code-generation-socials-zurich