Announcements

• Homework 1 grades returned
  • Style
  • Testing

• Homework 2: X86lite
  • Due Tuesday Sept 24

• Homework 3: LLVMlite
  • Will be released Tuesday Sept 24
Today

- Continue Intermediate Representation
- Intro to LLVM
Low-Level Virtual Machine (LLVM)

- Open-Source Compiler Infrastructure
  - see llvm.org for full documentation

- Created by Chris Lattner (advised by Vikram Adve) at UIUC
  - LLVM: An infrastructure for Multi-stage Optimization, 2002
  - LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, 2004

- 2005: Adopted by Apple for XCode 3.1

- Front ends:
  - llvm-gcc (drop-in replacement for gcc)
  - Clang: C, objective C, C++ compiler supported by Apple
  - various languages: Swift, ADA, Scala, Haskell, …

- Back ends:
  - x86 / Arm / PowerPC / etc.

- Used in many academic/research projects
LLVM Compiler Infrastructure

LLVM

frontends like 'clang'

Typed SSA IR

Optimizations/Transformations

Analysis

Ilc backend code gen jit

[Lattner et al.]
Example LLVM Code

- LLVM offers a textual representation of its IR
- Files ending in .ll

factorial64.c

```c
#include <stdio.h>
#include <stdint.h>

int64_t factorial(int64_t n) {
    int64_t acc = 1;
    while (n > 0) {
        acc = acc * n;
        n = n - 1;
    }
    return acc;
}
```

factorial-pretty.ll

```llvm
define @factorial(%n) {
  %1 = alloca
  %acc = alloca
  store %n, %1
  store 1, %acc
  br label %start

  %3 = load %1
  %4 = icmp sgt %3, 0
  br %4, label %then, label %else

  then:
  %6 = load %acc
  %7 = load %1
  %8 = mul %6, %7
  store %8, %acc
  %9 = load %1
  %10 = sub %9, 1
  store %10, %1
  br label %start

  else:
  %12 = load %acc
  ret %12
}
```
• Decorates values with type information
  i64
  i64*
  i1
• Permits numeric identifiers
• Has alignment annotations
• Keeps track of entry edges for each block:
  preds = %5, %0

factorial.ll

; Function Attrs: nounwind ssp
define i64 @factorial(i64 %n) #0 {
  %1 = alloca i64, align 8
  %acc = alloca i64, align 8
  store i64 %n, i64* %1, align 8
  store i64 1, i64* %acc, align 8
  br label %2
}

; <label>:2 ; preds = %5, %0
%3 = load i64* %1, align 8
%4 = icmp sgt i64 %3, 0
br i1 %4, label %5, label %11

; <label>:5 ; preds = %2
%6 = load i64* %acc, align 8
%7 = load i64* %1, align 8
%8 = mul nsw i64 %6, %7
store i64 %8, i64* %acc, align 8
%9 = load i64* %1, align 8
%10 = sub nsw i64 %9, 1
store i64 %10, i64* %1, align 8
br label %2

; <label>:11 ; preds = %2
%12 = load i64* %acc, align 8
ret i64 %12
}
define @factorial(%n) {
  entry:
  %1 = alloca
  %acc = alloca
  store %n, %1
  store 1, %acc
  br label %start

  loop:
  %3 = load %1
  %4 = icmp sgt %3, 0
  br %4, label %then, label %else

  body:
  %6 = load %acc
  %7 = load %1
  %8 = mul %6, %7
  store %8, %acc
  %9 = load %1
  %10 = sub %9, 1
  store %10, %1
  br label %start

  post:
  %12 = load %acc
  ret %12
}
LL Basic Blocks and Control-Flow Graphs

• LLVM enforces (some of) the basic block invariants syntactically.
• Representation in OCaml:

```ocaml
type block = {
  insns : (uid * insn) list;
  term : (uid * terminator)
}

type cfg = block * (lbl * block) list
```

• A **control flow graph** is represented as a list of labeled basic blocks with these invariants:
  • No two blocks have the same label
  • All terminators mention only labels that are defined among the set of basic blocks
  • There is a distinguished, unlabeled, entry block:
LL Storage Model: Locals

• Several kinds of storage:
  • Local variables (or temporaries): %uid
  • Global declarations (e.g. for string constants): @gid
  • Abstract locations: references to (stack-allocated) storage created by the alloca instruction
  • Heap-allocated structures created by external calls (e.g. to malloc)

• Local variables:
  • Defined by the instructions of the form %uid = …
  • Must satisfy the single static assignment invariant
    • Each %uid appears on the left-hand side of an assignment only once in the entire control flow graph.
    • The value of a %uid remains unchanged throughout its lifetime
  • Analogous to “let %uid = e in …” in OCaml
• Intended to be an abstract version of machine registers.
• We’ll see later how to extend SSA to allow richer use of local variables
  • phi nodes
LL Storage Model: alloca

• The alloca instruction allocates stack space and returns a reference to it.
  • The returned reference is stored in local:
    \%ptr = alloca typ
  • The amount of space allocated is determined by the type

• The contents of the slot are accessed via the load and store instructions:
  \%acc = alloca i64 ; allocate a storage slot
  store i64 341, i64* \%acc ; store the integer value 341
  \%x = load i64, i64* \%acc ; load the value 341 into \%x

• Gives an abstract version of stack slots
Structured Data
• Consider C-style structures like those below.
• How do we represent `Point` and `Rect` values?

```c
struct Point { int x; int y; };

struct Rect  { struct Point ll, lr, ul, ur };

struct Rect mk_square(struct Point ll, int len) {
    struct Rect square;
    square.ll = square.lr = square.ul = square.ur = ll;
    square.lr.x += len;
    square.ur.x += len;
    square.ur.y += len;
    square.ul.y += len;
    return square;
}
```
Representing Structs

- Store the data using two contiguous words of memory.
- Represent a `Point` value `p` as the address of the first word.

\[ p \rightarrow x \quad y \]

- Store the data using 8 contiguous words of memory.

\[ \text{square} \rightarrow \begin{array}{cccccccc}
  \text{ll.x} & \text{ll.y} & \text{lr.x} & \text{lr.y} & \text{ul.x} & \text{ul.y} & \text{ur.x} & \text{ur.y}
\end{array} \]

- Compiler needs to know the **size** of the struct at compile time to allocate the needed storage space.
- Compiler needs to know the **shape** of the struct at compile time to index into the structure.
Assembly-level Member Access

Consider: \[ \text{square.ul.y} = (\text{x86.insns, x86.operand}) \]

Assume that \%rcx holds the base address of \text{square}

Calculate the offset relative to the base pointer of the data:
- ul = sizeof(struct Point) + sizeof(struct Point)
- y = sizeof(int)

So: \[ \text{[square.ul.y]} = (\text{Movq 20(\%rcx) ans, ans}) \]
Padding & Alignment

• How to lay out non-homogeneous structured data?

```
struct Example {
    int x;
    char a;
    char b;
    int y;
};
```

32-bit boundaries

Not 32-bit aligned

Padding
Copy-in/Copy-out

• When we do an assignment in C as in:

```c
struct Rect mk_square(struct Point ll, int elen) {
    struct Square res;
    res.lr = ll;
    ...
```

we copy all elements from source and put in the target.

• Same as doing word-level operations:

```c
struct Rect mk_square(struct Point ll, int elen) {
    struct Square res;
    res.lr.x = ll.x;
    res.lr.y = ll.x;
    ...
```

• For really large copies, the compiler uses something like `memcpy` (which is implemented using a loop in assembly).
C Procedure Calls

• Similarly, when we call a procedure, we copy arguments in, and copy results out
  • Caller sets aside extra space in its frame to store results that are bigger than will fit in \%rax
  • We do the same with scalar values such as integers or doubles.

• Sometimes, this is termed "call-by-value".
  • This is bad terminology
  • Copy-in/copy-out is more accurate

• Benefit: locality

• Problem: expensive for large records…

• In C: can opt to pass pointers to structs: “call-by-reference”
  • Languages like Java and OCaml always pass non-word-sized objects by reference.
Call-by-Reference:

- The caller passes in the address of the point and the address of the result (1 word each).

```c
void mkSquare(struct Point *ll, int elen,
              struct Rect *res) {
    res->lr = res->ul = res->ur = res->ll = *ll;
    res->lr.x += elen;
    res->ur.x += elen;
    res->ur.y += elen;
    res->ul.y += elen;
}
void foo() {
    struct Point origin = {0,0};
    struct Square unit_sq;
    mkSquare(&origin, 1, &unit_sq);
}
```
Note that returning references to stack-allocated data can cause problems...

See unsafestack.c

For data that persists across a function call, we need to allocate storage in the heap...

in C, use the malloc library