Lecture 7

COMPILER DESIGN
Scanning our face
Privacy in the public sphere

Thursday, 17 October 2019, 12.15 – 1.45 p.m., ETH Zurich, Main Building, HG F30 (Audimax)

Welcome by
Detlef Günther, Vice President for Research and Corporate Relations,
ETH Zurich

Input by
Karim Nemr, Founder and CBO, PXL Vision

Moderated by
Anna Jobin, Health Ethics and Policy Lab, ETH Zurich

Panel discussion with
Othmar Hilliges, Professor of Computer Science, ETH Zurich
Fabian Kühhne, Chief Airport Special Division, Cantonal Police of Zurich
Karim Nemr, Founder and CBO, PXL Vision
Joanna Sleigh, Chair of Bioethics, ETH Zurich
Jean-Daniel Strub, Co-founder, ethix
Compiler Social

When: Thursday (tomorrow), 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
Intermediate Representations

- IR1: Expressions
  - simple arithmetic expressions, immutable global variables

- IR2: Commands
  - global *mutable* variables
  - commands for update and sequencing

- IR3: Local control flow
  - conditional commands & while loops
  - basic blocks

- IR4: Procedures (top-level functions)
  - local state
  - call stack
Basic Blocks

• A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  – Starts with a label that names the entry point of the basic block.
  – Ends with a control-flow instruction (e.g. branch or return) the “link”
  – Contains no other control-flow instructions
  – Contains no interior label used as a jump target

• Basic blocks can be arranged into a control-flow graph
  – Nodes are basic blocks
  – There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.
See llvm.org
LLVM Compiler Infrastructure

[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

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
}
```
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
}
STRUCTURED DATA
Compiling 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.ul.y += len;
    square.ur.x += len;
    square.ur.y += len;
    return square;
}
```
struct Point { int x; int y;};

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

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

\( p \) \[ \rightarrow \] \[ x \] \[ y \]

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

- Store the data using 8 contiguous words of memory.

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

square \[ \rightarrow \] \[ ll.x ll.y lr.x lr.y ul.x ul.y ur.x ur.y \]

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

• Assume that \(\%rcx\) holds the base address of \text{square}\n• Calculate the offset relative to the base pointer of the data:
  – \(\text{ul} = \text{sizeof(struct Point) + sizeof(struct Point)}\)
  – \(\text{y} = \text{sizeof(int)}\)

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

• How to lay out non-homogeneous structured data?

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

32-bit boundaries

Padding

Not 32-bit aligned
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;
    ...
}
```

then we copy all of the elements out of the source and put them 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:

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);
}

• The caller passes in the address of the point and the address of the result (1 word each).
• Note that returning references to stack-allocated data can cause problems.
  – Need to allocate storage in the heap...
Stack Pointers Can Escape

• Note that returning references to stack-allocated data can cause problems...

```c
int* bad() {
    int x = 341;
    int *ptr = &x;
    return ptr;
}
```

– 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
ARRAYS
Arrays

- Space is allocated on the stack for `buf`.
  - Note, without the ability to allocated stack space dynamically (C’s `alloca` function) need to know size of `buf` at compile time...

- `buf[i]` is really just: `(base_of_array) + i * elt_size`
Multi-Dimensional Arrays

• In C, \texttt{int M[4][3]} yields an array with 4 rows and 3 columns.
  • Laid out in \textit{row-major} order:

|---|---|---|---|---|---|---|---|

• \(M[i][j]\) compiles to?

• In Fortran, arrays are laid out in \textit{column major order}.

|---|---|---|---|---|---|---|---|

• In ML and Java, there are no multi-dimensional arrays:
  – \((\text{int array})\text{ array}\) is represented as an array of pointers to arrays of ints.
• Why is knowing these memory layout strategies important?
Array Bounds Checks

- Safe languages (e.g. Java, C#, ML but not C, C++) check array indices to ensure that they’re in bounds.
  - Compiler generates code to test that the computed offset is legal
- Needs to know the size of the array… where to store it?
  - One answer: Store the size before the array contents.

```
arr
```

|--------|------|------|------|------|------|------|------|

- Other possibilities:
  - Pascal: only permit statically known array sizes (very unwieldy in practice)
  - What about multi-dimensional arrays?
Array Bounds Checks (Implementation)

- Example: Assume `%rax` holds the base pointer (`arr`) and `%ecx` holds
  the array index `i`. To read a value from the array `arr[i]`:

  ```
  movq -8(%rax) %rdx // load size into rdx
  cmpq %rdx %rcx // compare index to bound
  j l __ok // jump if 0 <= i < size
  callq __err_oob // test failed, call the error handler
  __ok:
  movq (%rax, %rcx, 8) dest // do the load from the array access
  ```

- Clearly more expensive: adds move, comparison & jump
  - More memory traffic
  - Hardware can improve performance: executing instructions in parallel,
    branch prediction

- These overheads are particularly bad in an inner loop

- Compiler optimizations can help remove the overhead
  - e.g. In a for loop, if bound on index is known, only do the test once
A string constant "foo" is represented as global data:

```
_string42: 102 111 111 0
```

C uses null-terminated strings
- Strings are usually placed in the text segment so they are read only.
  - allows all copies of the same string to be shared.

Rookie mistake (in C): write to a string constant.

```c
char *p = "foo";
p[0] = 'b';
```

Instead, must allocate space on the heap:

```c
char *p = (char *)malloc(4 * sizeof(char));
strncpy(p, "foo", 4); /* include the null byte */
p[0] = 'b';
```