CS453 Automated Software Testing

Engineering LLVM IR and Measuring Condition Coverage of C Programs

Prof. Moonzoo Kim

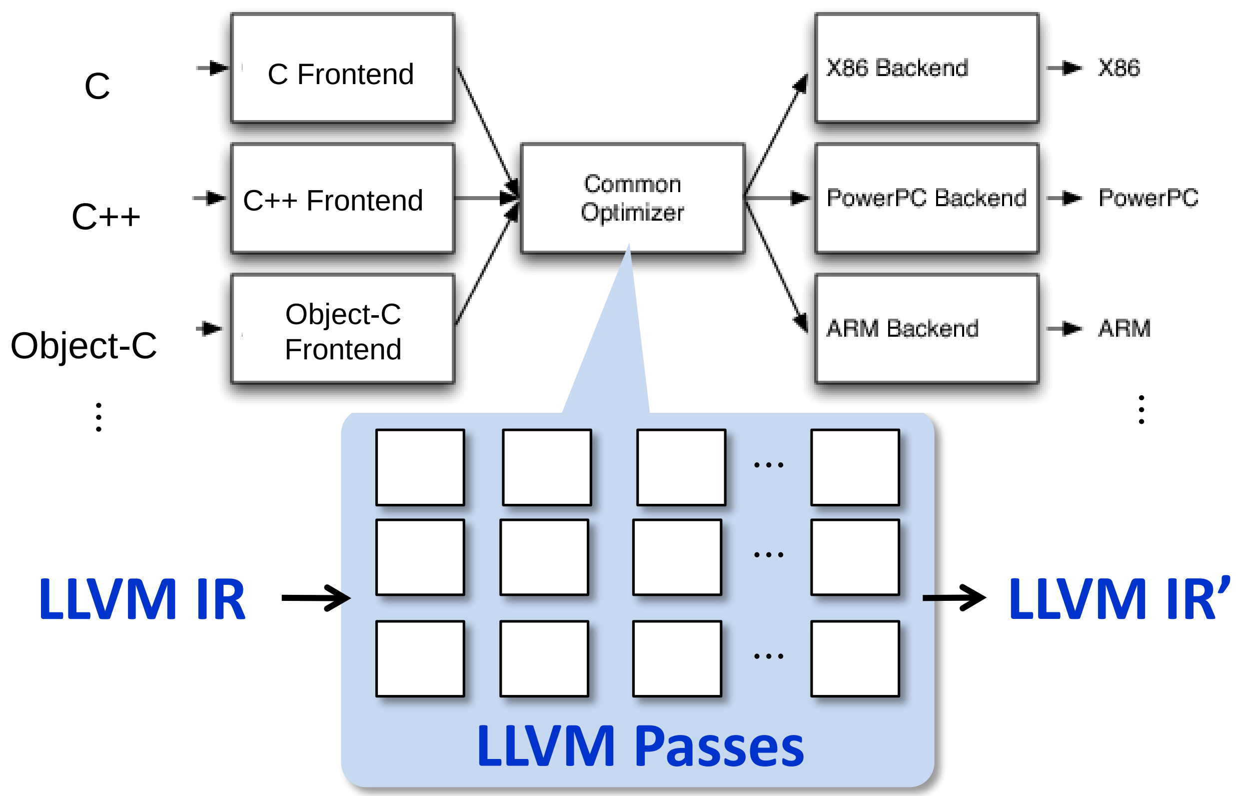
CS Dept., KAIST

Overview

- Build an LLVM IR level code instrumentor (transformer) that automatically inserts the conditional coverage measurement feature to a test target program
  - LLVM IR: simple and informative intermediate representation of C/C++ programs
  - Condition coverage: check if every conditional expression was ever evaluated as both false and true in a test

LLVM Compiler Infrastructure

- A collection of modular and reusable compilers and analyzers
- Being widely used in academy, open source, and commercials



LLVM is Professional Compiler

- Clang, the LLVM C/C++ front-end supports the full-features of C/C++ and compatible with GCC
  - The dominating compiler in Apple Inc. (used for *Swift* by Apple)
- The executable compiled by Clang/LLVM is as fast as the executable by GCC
- LLVM provides 108^+ Passes http://llvm.org/docs/Passes.html
  - Analyzers (41): alias analysis, call graph constructions, dependence analysis, etc.
  - Transformers (57): dead code elimination, function inlining, constant propagation, loop unrolling, etc.
  - Utilities (10): CFG viewer, basic block extractor, etc.

LLVM IR As Analysis Target

- The LLVM IR of a program is a *better target for analysis and engineering* than the program source code.
  - Language-independent
    - Able to represent C/C++/Object-C programs
  - Simple
    - register machine
    - static single assignment (SSA)
    - composed as basic blocks
  - Informative
    - typed language
    - control-flow
- LLVM IR is also called as LLVM language, assembly, bitcode, bytecode, code representation

LLVM IR At a Glance

C program language	*LLVM IR*
• Scope: *file, function*	*module, function*
• Type: *bool, char, int, struct{int, char}*	*i1, i8, i32, {i32, i8}*
• A statement with multiple expressions	A sequence of instructions each of which is in a form of “x = y *op* z”.
• Data-flow: a sequence of reads/writes on variables	1. load the values of memory addresses (variables) to registers; 2. compute the values in registers; 3. store the values of registers to memory addresses * each register must be assigned exactly once (SSA)
• Control-flow in a function: if, for, while, do while, switch-case,…	A set of basic blocks each of which is ended with a conditional jump (or return)

Example

simple.c

```c
1  #include <stdio.h>
2  int x, y ;
3
4  int main() {
5    int t ;
6    scanf(“%d %d”,&x,&y);
7    t = x – y ;
8    if (t > 0)
9     printf(“x > y”) ;
10   return 0 ;
11 }
```

```
$ clang –S –emit-llvm simple.c
```

simple.ll (simplified)

```llvm
   …
2   6 @x = common global i32 0, align 4
    7 @y = common global i32 0, align 4

4  11 define i32 @main() #0 {
   12 entry:
   …
5  14 %t = alloca i32, align 4
   …
6  16 %call = call i32 (i8*, ...)*
        @__isoc99_scanf(…i32* @x,i32* @y)
7  17 %0 = load i32* @x, align 4
   18 %1 = load i32* @y, align 4
   19 %sub = sub nsw i32 %0 %1
   20 store i32 %sub, i32* %t, align 4
8  21 %2 = load i32* %t, align 4
   22 %cmp = icmp sgt i32 %2, 0
   23 br i1 %cmp, label %if.then,
                  label %if.end
9  24 if.then:
   25   %call1 = call i32 … @printf(…
   26   br label %if.end
10 27 if.end:
   28   ret i32 0
```

Contents

- LLVM IR Instruction
  - architecture, static single assignment

- Data representation
  - types, constants, registers, variables
  - load/store instructions, cast instructions
  - computational instructions

- Control representation
  - control flow (basic block)
  - control instructions

- How to instrument LLVM IR

* *LLVM Language Reference Manual* http://llvm.org/docs/LangRef.html
* *Mapping High-Level Constructs to LLVM IR*
http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR

LLVM IR Architecture

- RISC-like instruction set
  - Only 31 op-codes (types of instructions) exist
  - Most instructions (e.g. computational instructions) are in three-address form: one or two operands, and one result
- Load/store architecture
  - Memory can be accessed via load/store instruction
  - Computational instructions operate on registers
- Infinite and typed *virtual registers*
  - It is possible to declare a new register any point
    (the backend maps virtual registers to physical ones).
  - A register is declared with a primitive type (boolean, int, float, pointer)

Static Single Assignment (1/2)

- In SSA, each variable is assigned exactly once, and every variable is defined before its uses.
- Conversion
  - For each definition, create a new version of the target variable (left-hand side) and replace the target variable with the new variable.
  - For each use, replace the original referred variable with the versioned variable reaching the use point.

```
x = y + x ;
y = x + y ;
if (y > 0)
  x = y ;
else
  x = y + 1 ;
```

```
x1 = y0 + x0 ;
y1 = x1 + y0 ;
if (y1 > 0)
  x2 = y1 ;
else
  x3 = y1 + 1 ;
```

Static Single Assignment (2/2)

- Use ϕ function if two versions of a variable are reaching one use point at a joining basic block
  - $\phi(x_1, x_2)$ returns a either x_1 or x_2 depending on which block was executed

```
x = y + x ;
y = x + y ;
if (y > 0)
  x = y ;
else
  x = y + 1 ;
y = x - y ;
```

```
x1 = y0 + x0 ;
y1 = x1 + y0 ;
if (y1 > 0)
  x2 = y1 ;
else
  x3 = y1 + 1 ;
x4 = ϕ(x2, x3);
y2 = x4 - y1 ;
```

Data Representations

- Primitive types
- Constants
- Registers (virtual registers)
- Variables
  - local variables, heap variables, global variables
- Load and store instructions
- Aggregated types

Primitive Types

- Language independent primitive types with predefined sizes
  - void: **`void`**
  - bool: **`i1`**
  - integers: **`i[N]`** where N is 1 to 2^{23}-1
    e.g. **`i8`**, **`i16`**, **`i32`**, **`i1942652`**
  - floating-point types:
    `half` (16-bit floating point value)
    `float` (32-bit floating point value)
    `double` (64-bit floating point value)

- Pointer type is a form of **`<type>*`** (e.g. `i32*`, `(i32*)*`)

Constants

- Boolean (`i1`): **`true`** and **`false`**
- Integer: standard integers including negative numbers
- Floating point: decimal notation, exponential notation, or hexadecimal notation (IEEE754 Std.)
- Pointer: **`null`** is treated as a special value

Registers

- Identifier syntax
  - Named registers: `[%][a-zA-Z$._][a-zA-Z$._0-9]*`
  - Unnamed registers: `[%][0-9][0-9]*`
- A register has a function-level scope.
  - Two registers in different functions may have the same identifier
- A register is assigned for a particular type and a value at its first (and the only) definition

Variables

- In LLVM, all addressable objects ("lvalues") are explicitly allocated.
- Global variables
  - Each variable has a global scope symbol that points to the memory address of the object
  - Variable identifier: `[@][a-zA-Z$._][a-zA-Z$._0-9]*`
- Local variables
  - The **`alloca`** instruction allocates memory in the stack frame.
  - Deallocated automatically if the function returns.
- Heap variables
  - The **`malloc`** function call allocates memory on the heap.
  - The **`free`** function call frees the memory allocated by **`malloc`**.

Load and Store Instructions

- Load

```
<result> = load <type>* <ptr>[, align <n>]
```

  - result: the target register
  - type: the type of the data (a pointer type)
  - ptr: the register that has the address of the data
  - n: the alignment of the memory address (optional)

- Store

```
store <type> <value>, <type>* <ptr>[, align <n>]
```

  - type: the type of the value
  - value: either a constant or a register that holds the value
  - ptr: the register that has the address where the data should be stored
  - n: the alignment of the memory address (optional)

Variable Example

```c
#include <stdlib.h>

int g = 0 ;

int main() {
 int t = 0;
 int * p;
 p=malloc(sizeof(int));
 free(p);
}
```

```llvm
@g = global i32 0, align 4
…
define i32 @main() #0 {
…
%t = alloca i32, align 4
store i32 0, i32* %t, align 4
%p = alloca i32*, align 8
%call = call noalias i8*
 @malloc(i64 4) #2
%0 = bitcast i8* %call to i32*
store i32* %0, i32** %p,
align 8
%1 = load i32** %p, align 8
…
```

Aggregate Types and Function Type

- Array: **[<# of elements> x <type>]**
  - Single dimensional array ex: [40 x i32], [4 x i8]
  - Multi dimensional array ex: [3 x [4 x i8]], [12 x [10 x float]]

- Structure: **type {<a list of types>}**
  - E.g. type{ i32, i32, i32 }, type{ i8, i32 }

- Function: **<return type> (a list of parameter types)**
  - E.g. i32 (i32), float (i16, i32*)*

Getelementptr Instruction

- A memory in an aggregate type variable can be accessed by **`load`**/**`store`** instruction and **`getelementptr`** instruction that obtains the pointer to the element.

- Syntax:

```
<res> = getelementptr <pty>* <ptrval>{,<t> <idx>}*
```

  - res: the target register
  - pty: the register that defines the aggregate type
  - ptrval: the register that points to the data variable
  - t: the type of index
  - idx: the index value

Aggregate Type Example

```c
struct pair {
    int first;
    int second;
};

int main() {
  int arr[10];
  struct pair a;

  a.first = arr[1];
  …
```

```llvm
%struct.pair = type{ i32, i32 }

define i32 @main() {
entry:
  %arr = alloca [10 x i32]
  %a = alloca %struct.pair

  %arrayidx = getelementptr
       [10 x 32]* %arr,i32 0,i64 1

  %0 = load i32* %arrayidx

  %first = getelementptr
   %struct.pair* %a,i32 0,i32 0

  %store i32 %0, i32* %first
```

Integer Conversion (1/2)

- Truncate
  - Syntax: **<res> = trunc <iN1> <value> to <iN2>**
    where **iN1** and **iN2** are of integer type, and **N1** > **N2**
  - Examples

```
%X = trunc i32 257 to i8 ;%X becomes i8:1
%Y = trunc i32 123 to i1 ;%Y becomes i1:true
%Z = trunc i32 122 to i1 ;%Z becomes i1:false
```

Integer Conversion (2/2)

- Zero extension
  - **<res> = zext <iN1> <value> to <iN2>** where **iN1** and **iN2** are of integer type, and **N1** < **N2**
  - Fill the remaining bits with zero
  - Examples
    - %X = **zext** i32 257 **to** i64 ;%X becomes i64:257
    - %Y = **zext** i1 true **to** i32 ;%Y becomes i32:1
- Sign extension
  - **<res> = sext <iN1> <value> to <iN2>** where **iN1** and **iN2** are of integer type, and **N1** < **N2**
  - Fill the remaining bits with the sign bit (the highest order bit) of value
  - Examples
    - %X = **sext** i8 -1 **to** i16 ;%X becomes i16:65535
    - %Y = **sext** i1 true **to** i32 ;%Y becomes i32:-1

Other Conversions

- Float-to-float
  - fptrunc .. to, fpext .. to
- Float-to-integer (vice versa)
  - fptoui .. to, tptosi .. to, uitofp .. to, sitofp .. to
- Pointer-to-integer
  - ptrtoint .. to, inttoptr .. to
- Bitcast
  - <res> = bitcast <t1> <value> to <t2>
    where t1 and t2 should be different types and have the same size

Computational Instructions

- Binary operations:
  - Add: `add`, `sub` , `fsub`
  - Multiplication: `mul` , `fmul`
  - Division: `udiv` , `sdiv` , `fdiv`
  - Remainder: `urem` , `srem` , `frem`

- Bitwise binary operations
  - shift operations: `shl` , `lshl` , `ashr`
  - logical operations: `and` , `or` , `xor`

Add Instruction

- `<res> = add [nuw][nsw] <iN> <op1>, <op2>`
  - nuw (no unsigned wrap): if unsigned overflow occurs, the result value becomes a poison value (undefined)
    - E.g: `add nuw i8 255, i8 1`
  - nsw (no signed wrap): if signed overflow occurs, the result value becomes a poison value
    - E.g. `add nsw i8 127, i8 1`

Control Representation

- The LLVM front-end constructs the control flow graph (CFG) of every function explicitly in LLVM IR
  - A function has a set of basic blocks each of which is a sequence of instructions
  - A function has exactly one entry basic block
  - Every basic block is ended with exactly one *terminator* instruction which explicitly specifies its successor basic blocks if there exist.
    - Terminator instructions: branches (conditional, unconditional), return, unwind, invoke

- As the instructions are represented as a form of CFG, it is very convenient to analyze, transform the target program in LLVM IR

Label, Return, and Unconditional Branch

- A label is located at the start of a basic block
  - Each basic block is addressed as the start label
  - A label `x` is referenced as register `%x` whose type is label
  - The label of the entry block of a function is "`entry`"
- Return **`ret <type> <value> | ret void`**
- Unconditional branch **`br label <dest>`**
  - At the end of a basic block, this instruction makes a transition to the basic block starting with label **`<dest>`**
  - E.g: `br label %entry`

Conditional Branch

- <res> = icmp <cmp> <ty> <op1>, <op2>
  - Returns either true or false (i1) based on comparison of two variables (op1 and op2) of the same type (ty)
  - cmp: comparison option

    eq (equal), ne (not equal), ugt (unsigned greater than),
    uge (unsigned greater or equal), ult (unsigned less than),
    ule (unsigned less or equal), sgt (signed greater than),
    sge (signed greater or equal), slt (signed less than), sle (signed less or equal)
- br i1 <cond>, label <thenbb>, label <elsebb>
  - Causes the current execution to transfer to the basic block <thenbb> if the value of <cond> is true; to the basic block <elsebb> otherwise.

- Example:

```
if (x > y)
  return 1 ;
return 0 ;
```

```
%0 = load i32* %x
%1 = load i32* %y
%cmp = icmp sgt i32 %0, %1
br i1 %cmp, label %if.then, label %if.end

if.then:
…
```

Switch

-
```
switch <iN> <value>, label <defaultdest>
                     [<iN> <val>, label <dest> …]
```
  - Transfer control flow to one of many possible destinations
  - If the value is found (`val`), control flow is transferred to the corresponding destination (`dest`); or to the default destination (`defaultdest`)
  - Examples:

```
switch(x) {
   case 1:
      break ;
   case 2:
      break ;
   default:
     break ;
}
```

```
%0 = load i32* %x
switch i32 %0, label %sw.default [
  i32 1, label %sw.bb
  i32 2, label %sw.bb1]

sw.bb:
  br label %sw.epilog

sw.bb1:
  br label %sw.epilog

sw.default:
  br label %sw.epilog

sw.epilog:
  …
```

PHI (Φ) instruction

- ```
  <res> = phi <t> [ <val_0>, <label_0>],
                  [ <val_1>, <label_1>], …
  ```
  - Return a value `val_i` of type `t` such that the basic block executed right before the current one is of `label_i`

- Example

```
y = (x > 0) ? x : 0 ;
```

```
%0 = load i32* %x
%c = icmp sgt i32 %0
br i1 %c, label %c.t, %c.f
c.t:
 %1 = load i32* %x
 br label %c.end
c.f:
 br label %c.end
c.end:
 %cond = phi i32 [%1, %c.t], [0, %c.f]
 store i32 %cond, i32* %y
```
```

Function Call

- **<res> = call <t> [<fnty>*] <fnptrval>(<fn args>)**
  - t: the type of the call return value
  - fnty: the signature of the pointer to the target function (optional)
  - fnptrval: an LLVM value containing a pointer to a target function
  - fn args: argument list whose types match the function signature

- Examples:

```
printf("%d", abs(x));
```

```
@.str = [3 x i8] c"%d\00"

%0 = load i32* %x
%call  = call i32 @abs(i32 %0)

%call1 = call i32 (i8*, ...)*
    @printf(i8*
     getelementptr ([3 x i8]* @.str,
      i32 0, i32 0),
    i32 %call)
```

Unaddressed Issues

- Many options/attributes of instructions
- Vector data type (SIMD style)
- Exception handling
- Object-oriented programming specific features
- Concurrency issues
  - Memory model, synchronization, atomic instructions

* *http://llvm.org/docs/LangRef.html*