SLIM: Simplified LLVM IR Modelling

- **Introduction**

  SLIM stands for Simplified LLVM IR Modelling, a conceptual framework that provides an abstraction over LLVM IR that not only models the IR instructions in a clean and simplified manner by hiding low level details, but also reduces the number of instructions that need to be considered. SLIM differs from LLVM IR in its view of memory abstraction. LLVM IR views memory in terms of load-store instructions whereas memory view of SLIM consists of the source variable names. SLIM is closer to the source program and thus, paves a way for an easy analysis on the IR. SLIM is being used actively for several analyses.

- **Background**

  LLVM IR is a low-level IR each instruction of which contains a lot of information resulting in an increase in the number of instructions. LLVM instructions are based on a load-store architecture so operands are accessed via a pointer and not directly. This adds an extra level of indirection to the operands thereby conflicting with their original definitions making analysis of the programs difficult. Program analysis is much easier in terms of the names of variables rather than in terms of their memory locations. Also, the load and store instructions are closer to the machine that executes the instructions thus magnifying the complexities of interpretation during analysis of the program. This is demonstrated in the following example.

  Example 1: Consider simple source statements where x and y are pointers of type integer and its corresponding LLVM IR.

Cases	Source statement	LLVM IR	SLIM
1	int *x, *y; x = y;	%i = load i32*, i32** @y, align 8	x = y
		store i32* %i, i32** @x, align 8	
2	int **y, *x; x = *y;	%i = load i32**, i32*** @y,align8	i_main = y
		%i1 = load i32*, i32** %i, align 8	i1_main = *i_main
		store i32* %i1, i32** @x, align 8	x = i1_main

3	int ***y, x; x = **y;	%i2=load i32**, i32*** @y,align8	i2_main = y
		%i3 = load i32*, i32** %i2,align 8	i3_main = *i2_main
		%i4 = load i32, i32* %i3, align 4	i4_main = *i3_main
		store i32 %i4, i32* @x, align 4	x = i4_main

Table 1: Source statements with corresponding LLVM IR and SLIM.

It can be observed that:

- LLVM may generate multiple IR instructions for a simple assignment statement in the source program.
- LLVM adds an extra level of indirection to the pointers x and y. For example in case 1, the source defines x and y as pointer-to an integer but LLVM IR represents them as pointer-to a pointer-to an integer. For case 3, **y is translated into two statements using a temporary such that each statement involves a single dereference.
- Apart from the original operands an LLVM IR instruction may also contain compiler-generated temporaries, type information, and alignment specification.

- **Motivation**

  Program Analysts writing machine independent passes may face the following challenges when performing some analysis on the LLVM IR code in example 1,
  - Separating and extracting out the relevant information from the irrelevant details in the LLVM IR.
  - All machine independent analyses view program state as a collection of variables instead of memory requiring load-store instructions.
  - Ignoring the extra indirection attached to the operands in order to match them to their original types.

  The key difference between LLVM IR and SLIM is in their view of memory abstraction. LLVM IR views memory in terms of load-store instructions containing variables which are either registers or memory locations accessed using pointers. SLIM views memory in terms of variables names that are allocated, accessed, and modified during program execution.

  Data flow analysis is designed for three address code, closer to source code. Since LLVM IR represents the instructions in the form of load-store instructions, SLIM provides an abstraction lifting it to three address code so that the implementation truthfully models the design of analysis.

SLIM extracts the relevant information from LLVM IR instructions irrespective of the type of analyses to be performed while ignoring the irrelevant information listed in Table 2.

Relevant Information	Irrelevant Information
operand names	alignment information
operand attributes (global/temporary/alloca/GEPoperator/ array/function name/formal argument)	instruction mnemonics
field indices	addressing modes
type of instruction	size of the data types
operand data type	LLVM IR keywords and comments

Table 2: Relevant information extracted by SLIM and irrelevant information ignored by SLIM from LLVM IR.

- **Example C program and its corresponding SLIM**

  An example C source program and its corresponding SLIM is shown in the following Figure 1.

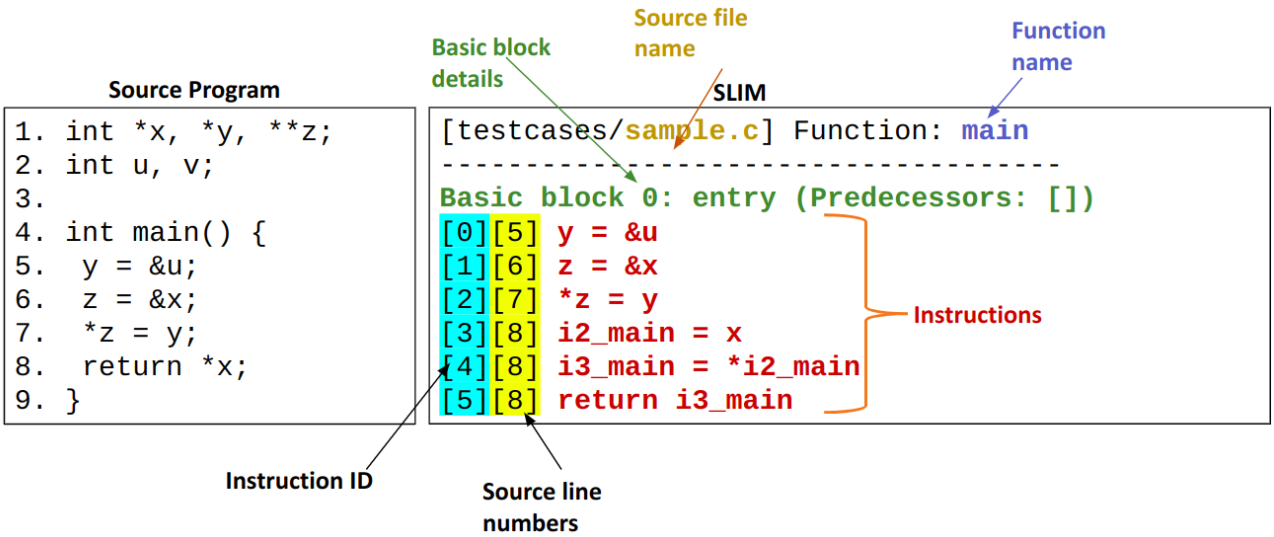

Figure 1: Example C program and its corresponding SLIM

- **SLIM Modelling**

  The key idea behind SLIM modeling is to translate an LLVM IR instruction into the following form:

  <Lhs> = <Rhs_1 {[$index_1$] ...[$index_n$]}> ,..., <Rhs_n {[$index_1$] ...[$index_n$]}>

  where Lhs and Rhs are the operands of the corresponding LLVM IR instruction and index (optional) represents the field index in case of aggregate data structures defined in the source. Some instructions such as return and compare which indicate the use of the operands with respective to program analysis are modelled as Rhs only and Lhs is ignored.

  SLIM attaches a level of indirection with each operand for better modeling as shown in Table 3. The indirection level indicates the number of edges traversed to reach the operand in the memory graph.

Variable	Indirection Level
&x	0
x	1
*x	2

Table 3: Levels of indirection

- **Features of SLIM**

  The features of SLIM are the following:

  - SLIM generates a modelling that is clean, simple, closer to the source program and yet in Three Address Code (TAC)
  - Eliminates extra level of indirection of source variables added due to LLVM IR's view of memory in terms of memory locations rather than variables.
  - Explicates the source variables used in the instructions.
  - Provides an abstraction to eliminate temporaries and obtain source variables. Using source variables instead of temporaries increases the scalability of context-sensitive interprocedural analysis significantly.
  - Performs formal to actual argument mapping for a call by adding assignment statements before the call instruction.

- Performs return argument mapping by adding assignment statements after the call instruction.
- Extracts relevant information from LLVM IR as listed in Table 2.

- **Tutorial Problem**

  Consider the C program shown in Figure 2. Table 4 lists the LLVM IR instructions computed for the program in Figure 2 and also presents the corresponding SLIM modelling of these instructions.

```
int a, b, c, *x,*y;
struct S {
     int *m;
     int n;
}*p, s1;

int main() {
  int *q;
  int r, s;
  y = &r;
  r = 10;
  if (a == b)
    q = &a;
  else
    q = &b;
  x = q;
  p = &s1;
  p->m = &c;
  x = p->m;
  s1.n = a + b;
  c = *y;
  return 0;
}
```

Figure 2: Example C program

LLVM IR	SLIM
%r = alloca i32	Alloca instructions are ignored
store i32* %r, i32** @y	y = &r_main
store i32 10, i32* %r	r_main = &10
%i = load i32, i32* @a	i_main = a
%i1 = load i32, i32* @b	i1_main = b
%cmp = icmp eq i32 %i, %i1	cmp_main = i_main == i1_main
br i1 %cmp, label %if.then, label %if.else	branch (cmp_main) if.then_main, if.else_main
if.then: br label %if.end	branch if.end_main
if.else: br label %if.end	branch if.end_main
%q.0 = phi i32* [@a, %if.then], [@b, %if.else]	q.0_main = phi(a, b)
store i32* %q.0, i32** @x	x = q.0_main
store %struct.S* @s1, %struct.S** @p	p = &s1
%i3 = load %struct.S*, %struct.S** @p	i3_main = p
%m = getelementptr inbounds %struct.S, %struct.S* %i3	m_main = &i3_main[0][0]
store i32* @c, i32** %m	*m_main = &c
%i4 = load %struct.S*, %struct.S** @p	i4_main = p
%m1 = getelementptr inbounds %struct.S, %struct.S* %i4, i32 0, i32 0	m1_main = &i4_main[0][0]
%i5 = load i32*, i32** %m1	i5_main = *m1_main
store i32* %i5, i32** @x	x = i5_main
%i6 = load i32, i32* @a	i6_main = a
%i7 = load i32, i32* @b	i7_main = b
%add = add nsw i32 %i6, %i7	add_main = i6_main + i7_main
store i32 %add, i32* getelementptr inbounds (%struct.S, %struct.S* @s1, i32 0, i32 1)	s1[0][1] = add_main
%i8 = load i32*, i32** @y	i8_main = y
%i9 = load i32, i32* %i8	i9_main = *i8_main
store i32 %i9, i32* @c	c = i9_main
ret i32 0	return 0

Table 4: LLVM IR and SLIM for example C program in Figure 2.

For simplicity we have omitted the indirection levels from all the instructions in Table 4. SLIM may print the instruction m1_main = &i4_main[0][0] as <m1_main, 1> = <i4_main[0][0], 0> where, 1 and 0 represent the indirection levels.

- **SLIM Interface**
  Following is the list of some SLIM API's that are frequently used for performing any program analysis.

  - Fetching SLIM Operands: *(function definitions in Instructions.cpp)*
    - getLHS : Returns the LHS operand
    - getRHS : Returns list of RHS operands

  - Attributes of an operand: *(function definitions in Operand.cpp)*
    - getOperandType : Returns the operand type
    - getValue : Returns the pointer to the corresponding llvm::Value object
    - getTransitiveIndices : Returns index vector for a GEP operator
    - isArrayElement : Returns true if the operand is of array type
    - isAlloca : Returns true if the operand is a result of an alloca instruction
    - isVariableGlobal : Returns true if the operand is a global variable or an address-taken local
    - isPointerVariable : Returns true if the operand is a pointer variable
    - isPointerInLLVM : Returns true if the operand is a pointer variable in LLVM IR

  - Printing operand: *(function definition in Operand.cpp)*
    - getOnlyName : Returns name of the operand

  - Dumping SLIM : *(function definitions in IR.cpp)*
    - dumpIR : Prints SLIM instructions

- **Installation Guide**

  LLVM-14 must be successfully installed before installing SLIM.
  You can check the version by using the following command:
  $> llvm-config -- version

  Steps to install SLIM library

  - Download SLIM in a directory
  - Create a build folder in that directory
    $> mkdir -p build
  - Change the working directory to build
    $> cd build
  - Run the cmake command to generate the Makefile
    $> cmake -S .. -DDISABLE_IGNORE_EFFECT=ON -B .
    $> cmake --build .
  - Install the library with the command
    $> sudo cmake --install .

On successful SLIM installation, the library will be installed at the location `/usr/local/lib` and the header files will be stored at the location `/usr/local/include/slim`.