LLVM2GOTO: A translator from LLVM IR to CPROVER IR

Sapate Rasika Avinash

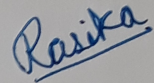
A thesis submitted to
Indian Institute of Technology Hyderabad
in partial fulfillment of the requirements
for the degree of
Master of Technology

Department of Computer Science and Engineering
Indian Institute of Technology Hyderabad

June 2018

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.



(Sapate Rasika Avinash)
CS16MTECH11014

Approval Sheet

This Thesis entitled LLVM2GOTO by Sapate Rasika Avinash is approved for the degree of Master of Technology from IIT Hyderabad

(Dr. M. V. Panduranga Rao) Examiner
Dept. CSE
IITH

(Dr. Ramakrishna Upadrasta) Examiner
Dept. CSE
IITH

(Dr. Saurabh Joshi) Adviser
Dept. of CSE
IITH

(———) Chairman
Dept. of CSE
IITH

Acknowledgements

I take this opportunity to express a deep sense of gratitude towards my guide Dr. Saurabh Joshi, for providing excellent guidance and encouragement throughout the project work. Without his guidance, this work would never have been successful.
I would also like to thank Dr. Ramakrishna Upadrasta for his support and coordination.

Abstract

There are more than 700 programming languages. The number of softwares is astronomical. It is highly important to verify whether the software meets it's specification and it is safe. However, there are very few stable software verification tools. Translating a source program into verification intermediate representation(VIR) is an overhead for software verification community. If we translate compiler intermediate representation into VIR, the overhead of translating source to VIR is reduced and software written in programming languages supported by the compiler can be verified. LLVM2GOTO uses LLVM IR as compiler IR and CPROVER's goto IR as VIR. In the current implementation we support variable declaration, load, store, arithmetic, bitwise, typecast, branch and switch instructions are supported by LLVM2GOTO.

Contents

Declaration . . . ii
Approval Sheet . . . iii
Acknowledgements . . . iv
Abstract . . . v

Nomenclature **vii**

1 Introduction **1**
1.1 LLVM . . . 2
1.1.1 Intermediate representation (IR) . . . 2
1.1.2 Metadata . . . 3
1.2 CPROVER . . . 4
1.2.1 GOTO IR . . . 4
1.3 LLVM2GOTO . . . 5
1.4 Related work . . . 5
1.4.1 SMACK . . . 5

2 Internals of LLVM2GOTO **6**
2.1 Terminator Instructions . . . 6
2.1.1 Ret . . . 6
2.1.2 Br . . . 6
2.1.3 Switch . . . 8
2.1.4 Unreachable . . . 10
2.2 Binary operators . . . 11
2.2.1 Add/Sub/Mul . . . 11
2.2.2 UDiv/SDiv/URem/SRem . . . 12
2.2.3 FAdd/FSub/FMul/FDiv/FRem . . . 13
2.3 Memory access and addressing operations . . . 13
2.3.1 Alloca . . . 13
2.3.2 Load . . . 13
2.3.3 Store . . . 13
2.3.4 GetElementPtr . . . 13
2.4 Conversion operations . . . 14
2.5 Comparison operations . . . 14
2.6 Bitwise operations . . . 14

2.7 Other instructions 15
2.7.1 PHI 15
2.7.2 Call 16
2.8 Data types 17
2.9 Scoping information 18

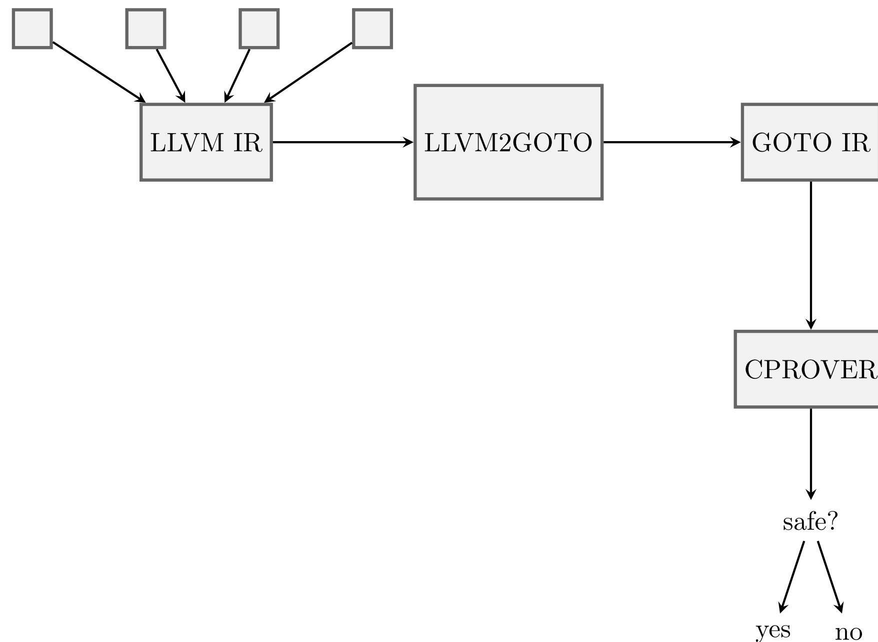
3 Summary and Future work **21**
3.1 Summary 21
3.2 Future work 21

References **22**

Chapter 1

Introduction

Correctness of a software extremely important. Software verification includes verifying whether the software works according to it's specification, verifying it is safe with respect to the language specification and termination of program using formal methods. There are countless softwares/programs but very few stable verification tools available for very few languages. Translating source program to VIR is an overhead. Reasearchers are more interested in verification algorithms rather than this translation. Using LLVM2GOTO may allow researchers to focus on verification techniques as it translates LLVM IR to GOTO VIR. Thus, higher number of languages can be supported by verifiers.

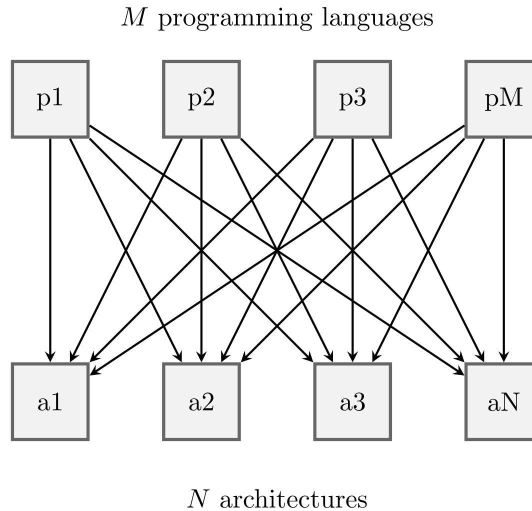


1.1 LLVM

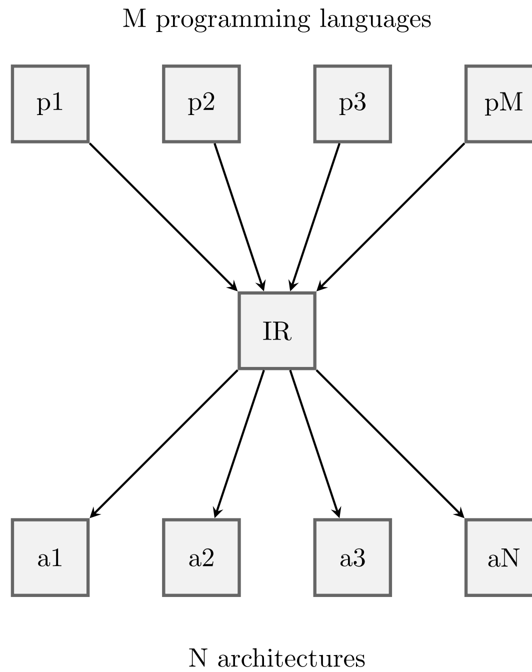
"The LLVM Project is a collection of modular and reusable compiler and $tool - chain$ technologies" [1].
LLVM is well known for it's modularity and extensibility. It supports frontends for languages such as C, C++, CUDA, D, Delphi, Fortran, Halide, Haskell, Julia, R, Ruby, Rust, Scala, Swift, Ada, Go, Lua, Java bytecode, etc. These frontends translate programs in given source languages into LLVM IR. It also supports various backends such as X86, AMD, PowerPC, Mips, Sparc, etc. LLVM IR has language independent instruction set. Also it is in Static Single Assignment(SSA) [2] form where each variable is assigned only once. This makes it easy to perform dataflow analyses.
An LLVM program is a list of functions. Each function consists of list of basic blocks. A basic block is list of instructions to be executed in sequential manner.

1.1.1 Intermediate representation (IR)

Let us say there are M programming languages and N architectures. To compile these programs to execute on these architectures, we need $M * N$ compilers.



To reduce the compiler efforts, we translate program written in any programming language into intermediate representation, and later translate this intermediate representation into machine code of desired architecture. Now $number\ of compilers/translators\ required = M + N$.
This reduces the effort from multiplicative to additive.



Apart from reduction in the number of compilers, intermediate representation also allows machine independent optimization.

1.1.2 Metadata

```
define i32 @main() #0 !dbg !18 {
entry:
  %retval = alloca i32, align 4
  %x = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  %0 = load i32, i32* getelementptr
   inbounds (%struct.student,%struct.student* @S,i32 0,i32 0),align 4,!dbg !21
  call void @llvm.dbg.declare
   (metadata i32* %x,metadata !22,metadata !23),!dbg !24
  store i32 5, i32* %x, align 4, !dbg !24
  ret i32 0, !dbg !25
}
```

Part of metadata for above program is as follows:

```
!0 = !DIGlobalVariableExpression(var: !1)
!1 = distinct !DIGlobalVariable(name: "S", scope: !2, file: !3, line: 5,
     type: !6, isLocal: false, isDefinition: true)
!2 = distinct !DICompileUnit(language: DW_LANG_C99, file: !3, producer:
    "clang version 5.0.0 (trunk 295264)", isOptimized: false,
    runtimeVersion: 0, emissionKind: FullDebug, enums: !4, globals: !5)
!3 = !DIFile(filename: "scope.c", directory: "llvm2goto")
```

```
!6 = distinct !DICompositeType(tag: DW_TAG_structure_type, name: "
   student", file: !3, line: 1, size: 192, elements: !7)
!7 = !{!8, !10}
!8 = !DIDerivedType(tag: DW_TAG_member, name: "roll_no", scope: !6, file
   : !3, line: 3, baseType: !9, size: 32)
!9 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!11 = !DICompositeType(tag: DW_TAG_array_type, baseType: !12, size: 160,
    elements: !13)
!13 = !{!14}
!14 = !DISubrange(count: 20)
!18 = distinct !DISubprogram(name: "main", scope: !3, file: !3, line: 6,
    type: !19, isLocal: false,  isDefinition: true, scopeLine: 7,
   isOptimized: false, unit: !2, variables: !4)
!19 = !DISubroutineType(types: !20)
!20 = !{!9}
!21 = !DILocation(line: 8, column: 4, scope: !18)
!22 = !DILocalVariable(name: "x", scope: !18, file: !3, line: 9, type:
   !9)
```

Metadata provides information about the program such as variable name, scope, location, type in the source program. Though LLVM does not preserve sign of integer types, it can be determined using encoding field in metadata.

1.2 CPROVER

CPROVER [3] is a verification framework. Various tools like CBMC [4], ESBMC [5], 2LS [6], JBMC [7], SATAbs [8], etc., use it's verification IR i.e. GOTO IR.

1.2.1 GOTO IR

As mentioned in earlier section, using an intermediate representation helps reducing the efforts required. It contains a list of GOTO functions. GOTO functions store the information about function such as function name, variable names, etc. and GOTO program which corresponds to function body. Each GOTO program is a list of GOTO instructions. GOTO IR supports goto(branch), assume, assert, start−thread, end−thread, return, assignment, declaration, function call, throw and catch instructions.

1.3 LLVM2GOTO

LLVM2GOTO is a translator that translates LLVM IR into CPROVER's verification IR i.e. GOTO IR.

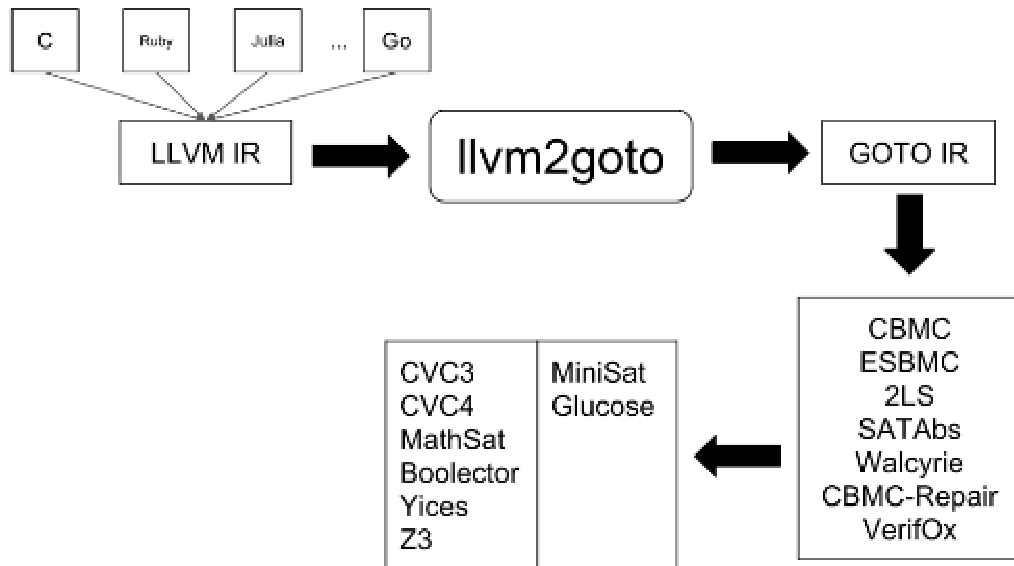


Figure 1.1: LLVM2GOTO toolchain

1.4 Related work

1.4.1 SMACK

SMACK is a translator that translates LLVM's intermediate representation into Boogie's Verification Intermediate Representation(VIR) [9]. SMACK currently supports C. Another limitation of SMACK is that floating point operations.

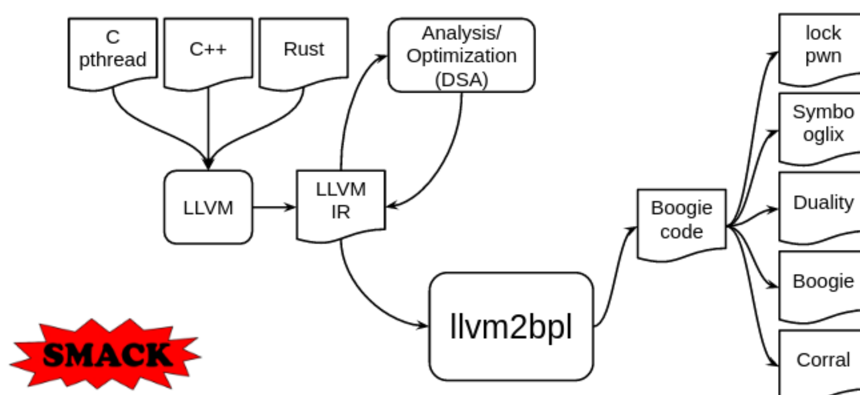


Figure 1.2: "SMACK toolchain". The image is borrowed from [9]

Chapter 2

Internals of LLVM2GOTO

2.1 Terminator Instructions

These instructions appear at the end of basic block.

2.1.1 Ret

We have assumed that **Ret** instruction returns a temporary variable or a load instruction or a constant or void. The type of constant can be integer(boolean and signed or unsigned integers) or float(IEEE 754 standard float and double).

2.1.2 Br

A branch can be conditional or unconditional. As we translate instructions one by one, we do not have the target for **Br** instruction. Thus we use two tables, one that maps LLVM branch instruction to goto instruction and second maps instruction at the beginning of BasicBlock to corresponding goto instruction. After first pass, we can assign the target to given LLVM branch instruction.

```c
int main()
{
  int i, a, b;
  if(i == 0)
  {
    a = b;
  }
  return 0;
}
```

```llvm
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  %0 = load i32, i32* %i, align 4
  %cmp = icmp eq i32 %0, 0
  br i1 %cmp, label %if.then, label %if.end

if.then: ; preds = %entry
  %1 = load i32, i32* %b, align 4
  store i32 %1, i32* %a, align 4
  br label %if.end

if.end: ; preds = %if.then, %entry
  ret i32 0
}
```

Conditional branch

```c
int main()
{
  int i, a, b;
  goto p;
  a = 10;
p:
  i = a+b;
  return 0;
}
```

```llvm
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  br label %p

p: ; preds = %entry
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %i, align 4
  ret i32 0
}
```

Unconditional branch

2.1.3 Switch

The condition in **Switch** instruction can be a variable, load instruction, or constant integer type. For every `case`, the value-target pair is translated into a branch instruction. After all the cases, unconditional branch to default is generated.

```c
int main()
{
  int i, a, b;
  i = 100;
  switch(i)
  {
    case 0:
           a = a + b;
         break;
    case 1:
      a = a - b;
         break;
    default :
             a = 20;
  }
  return 0;
}
```

```llvm
define i32 @main() #0 !dbg !6 {
entry:
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 100, i32* %i, align 4, !dbg !17
  %0 = load i32, i32* %i, align 4, !dbg !18
  switch i32 %0, label %sw.default [
    i32 0, label %sw.bb
    i32 1, label %sw.bb1
  ], !dbg !19

sw.bb: ; preds = %entry
  %1 = load i32, i32* %a, align 4, !dbg !20
  %2 = load i32, i32* %b, align 4, !dbg !22
  %add = add nsw i32 %1, %2, !dbg !23
  store i32 %add, i32* %a, align 4, !dbg !24
  br label %sw.epilog, !dbg !25

sw.bb1: ; preds = %entry
  %3 = load i32, i32* %a, align 4, !dbg !26
  %4 = load i32, i32* %b, align 4, !dbg !27
  %sub = sub nsw i32 %3, %4, !dbg !28
  store i32 %sub, i32* %a, align 4, !dbg !29
  br label %sw.epilog, !dbg !30

sw.default: ; preds = %entry
  store i32 20, i32* %a, align 4, !dbg !31
  br label %sw.epilog, !dbg !32

sw.epilog: ; preds = %sw.default, %sw.bb1, %sw.bb
  ret i32 0, !dbg !33
}
```

Switch with break and default

Every LLVM block has a branch instruction at the end. Thus, fall through and other characteristics of switch are taken care of. The branch targets are assigned after the first pass as mentioned in Br.

```c
int main()
{
  int i, a, b;
  i = 0;
  switch(i)
  {
    case 0:
      a = a + b;
    case 1:
      a = a - b;
    default :
      a = 20;
  }
}
```

```llvm
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 0, i32* %i, align 4
  %0 = load i32, i32* %i, align 4
  switch i32 %0, label %sw.default [
    i32 0, label %sw.bb
    i32 1, label %sw.bb1
  ]

sw.bb: ; preds = %entry
  %1 = load i32, i32* %a, align 4
  %2 = load i32, i32* %b, align 4
  %add = add nsw i32 %1, %2
  store i32 %add, i32* %a, align 4
  br label %sw.bb1

sw.bb1: ; preds = %entry, %sw.bb
  %3 = load i32, i32* %a, align 4
  %4 = load i32, i32* %b, align 4
  %sub = sub nsw i32 %3, %4
  store i32 %sub, i32* %a, align 4
  br label %sw.default

sw.default: ; preds = %entry, %sw.bb1
  store i32 20, i32* %a, align 4
  br label %sw.epilog

sw.epilog: ; preds = %sw.default
  ret i32 0
}
```

Switch with fall through

```c
int main()
{
  int i, a, b;
  i = 0;
  switch(i)
  {
    case 0:
      a = a + b;
    case 1:
      a = a - b;
  }
  return 0;
}
```

```llvm
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 0, i32* %i, align 4
  %0 = load i32, i32* %i, align 4
  switch i32 %0, label %sw.epilog [
    i32 0, label %sw.bb
    i32 1, label %sw.bb1
  ]

sw.bb: ; preds = %entry
  %1 = load i32, i32* %a, align 4
  %2 = load i32, i32* %b, align 4
  %add = add nsw i32 %1, %2
  store i32 %add, i32* %a, align 4
  br label %sw.bb1

sw.bb1: ; preds = %entry, %sw.bb
  %3 = load i32, i32* %a, align 4
  %4 = load i32, i32* %b, align 4
  %sub = sub nsw i32 %3, %4
  store i32 %sub, i32* %a, align 4
  br label %sw.epilog

sw.epilog: ; preds = %sw.bb1, %entry
  ret i32 0
}
```

Switch without default

2.1.4 Unreachable

This instruction is used for compiler optimizations. As we do not want any optimization, we ignore this instruction.

2.2 Binary operators

2.2.1 Add/Sub/Mul

Operands can be temporary variable, load instruction or constant. Type of operand can be signed or unsigned integer. As output for these instructions in bitvector is the same independent of sign of the operands, LLVM does not preserve the sign information for these instructions.

```
int main()
{
  int i, a, b;
  a = 10;
  b = -9;
  i = a + b;
  return 0;
}
```

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 10, i32* %a, align 4
  store i32 -9, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %i, align 4
  ret i32 0
}
```

Addition of signed integers

```
int main()
{
  int i, a, b;
  a = 10;
  b = 9;
  i = a + b;
  return 0;
}
```

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 10, i32* %a, align 4
  store i32 9, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %i, align 4
  ret i32 0
}
```

Addition of unsigned integers

Though output is the same, we need to know the sign of the operands to identify if there is any overflow or underflow.

```
void swap(unsigned &a, unsigned &b)
{
  a = a + b;
  b = a - b;
  a = a - b;
}

Input:
  1
  4294967295

Output:
  4294967295
  1
```

```
void swap(int &a, int &b)
{
  a = a + b;
  b = a - b;
  a = a - b;
}

Input:
  1
  4294967295

Output:
  4294967295
  1
```

Note, there is no way to recognize the sign of an integer constant. Thus sign of the other variable involved in an expression is assigned to the constant.
e.g.

```
int a, b;
a = b + 10;
```

2.2.2 UDiv/SDiv/URem/SRem

Output of division and remainder depends on the sign of their operands. As mentioned earlier, LLVM does not preserve the sign information. Thus, there are separate instructions for signed and unsigned operands.

```
int i, a = 9;
unsigned ui, b = 9;
i = a / -5;
ui = b / 5;
```

```
store i32 9, i32* %a, align 4
store i32 9, i32* %b, align 4
%0 = load i32, i32* %a, align 4
%div = sdiv i32 %0, -5
store i32 %div, i32* %i, align 4
%1 = load i32, i32* %b, align 4
%div1 = udiv i32 %1, 5
store i32 %div1, i32* %ui, align 4
```

udiv and sdiv instructions

2.2.3 FAdd/FSub/FMul/FDiv/FRem

Operands are either float or double type.

```
f = a + b;
```

```
%0 = load float,float* %a,align 4
%1 = load float,float* %b,align 4
%add = fadd float %0,%1
store float %add,float* %f,align 4
```

```
// 6 file float.c line 4 column 8
__CPROVER_floatbv[32][23] add;
// 7 file float.c line 4 column 8
add = a + b;
// 8 file float.c line 4 column 4
f = add;
```

Floating point addition

2.3 Memory access and addressing operations

2.3.1 Alloca

Alloca instruction allocates the memory for program variables. Whenever an alloca instruction is found, an entry is added to the symbol table. No instruction is added to goto program.

2.3.2 Load

Load instruction is handled whenever used as an operand. Thus no corresponding instruction is generated.

2.3.3 Store

Store represents an assignment statement. Types of both value and variable should match. If they do not, we use typecast instruction.

2.3.4 GetElementPtr

GetElementPtr returns the address of a subelement of an composite data type e.g. array, structure.

```
struct st s;
int i;
int arr[3];
i = st.roll;
arr[2];
```

```
%s = alloca %struct.st, align 4
%arr = alloca [3 x i32], align 6
%r=getelementptr inbounds %struct.st,%struct.st* %s,i32 0, i32 0
%0 = load i32, i32* %r, align 4
store i32 %0, i32* %i, align 4
%idx=getelementptr inbounds [3 x i32],[3 x i32]* %arr,i64 0,i64 2
store i32 10, i32* %idx, align 4
```

GetElementPtr example

2.4 Conversion operations

Typecast instruction in goto IR requires the expression to typecast and destination type.

- Trunc : **Trunk** used to reduce bitsize of given integer value or variable. Here, we need to make sure that there is no data loss.
- ZExt/SExt : Two separate instructions **ZExt** and **SEXT** are used to preserve the behaviour of program for signed and unsigned integer extension.
- FPTrunc : **FPTrunk** used to reduce size of given floating point value or variable. We make sure that there is no lossy conversion.
- FPExt : **FPExt** used to extend floating point value to larger floating point value.
- FPToUI : **FPToUI** used to cast floating point value to unsigned integer.
- FPToSI : **FPToSI** used to cast floating point value to signed integer.
- UIToFP : **UIToFP** used to cast unsigned integer to floating point value.
- SIToFP : **SIToFP** used to cast signed integer to floating point value.

2.5 Comparison operations

- ICmp : Used to compare two integers.
- FCmp : Used to compare two floating point values.

2.6 Bitwise operations

- Shl/LShr/Ashr : Bitwise shift operations have their usual meaning.
- And/Or/Xor : Perform bitwise and/or/xor operations. Should not be confused with logical and/or operations. LLVM doesn't have logical operators.

2.7 Other instructions

2.7.1 PHI

PHI instruction is used to implement ϕ node in SSA(static single assignment) [2] graph of the function.

```
int main()        define i32 @main() #0 {
{                 entry:
  int x,y;          %retval = alloca i32, align 4
  x && y;           %x = alloca i32, align 4
  return 0;         %y = alloca i32, align 4
}                   store i32 0, i32* %retval, align 4
                    %0 = load i32, i32* %x, align 4
                    %tobool = icmp ne i32 %0, 0
                    br i1 %tobool, label %land.rhs, label %land.end

                  land.rhs: ; preds = %entry
                    %1 = load i32, i32* %y, align 4
                    %tobool1 = icmp ne i32 %1, 0
                    br label %land.end

                  land.end: ; preds = %land.rhs, %entry
                    %2 = phi i1 [ false, %entry ], [ %tobool1, %land.rhs ]
                    %land.ext = zext i1 %2 to i32
                    ret i32 0
                  }
```

For each **PHI** instruction, a temporary variable is created. It is assigned distinct values in each of the predecessors of it's basic block. The value of phi node is decided based on value of temporary variable.

```
// 1 no location
_phi_0_ = 0;
// 2 file phi.c line 1 column 16
signed __CPROVER_bitvector[32] x;
// 3 file phi.c line 1 column 18
signed __CPROVER_bitvector[32] y;
// 4 no location
_Bool tobool;
// 5 file phi.c line 1 column 20
tobool = x != 0;
// 6 no location
IF !(x != 0) THEN GOTO 1
```

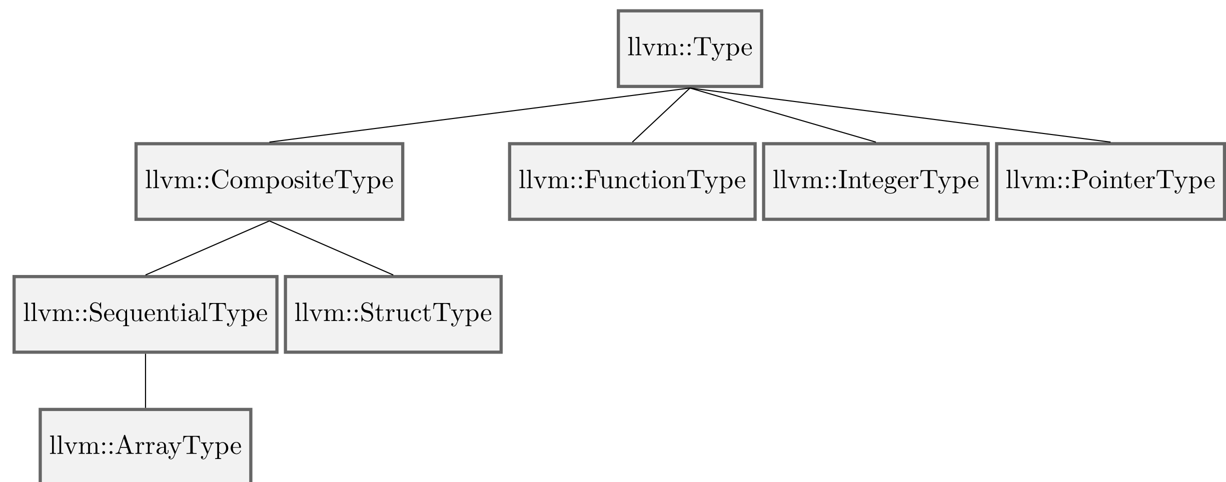
```
   // 7 no location
   _phi_0_ = 1;
   // 8 no location
   _Bool tobool1;
   // 9 file phi.c line 1 column 22
   tobool1 = y != 0;
   // 10 no location
   GOTO 1
   // 11 no location
1: _Bool _phi_0;
   // 12 no location
   IF _phi_0_ != 0 THEN GOTO 2
```

```
    // 13 no location
    _phi_0 = FALSE;
    // 14 no location
 2: IF _phi_0_ != 1 THEN GOTO 3
    // 15 no location
    _phi_0 = tobool1;
    // 16 file phi.c line 1 column 22
 3: signed __CPROVER_bitvector[32] land.ext;
    // 17 file phi.c line 1 column 22
    land.ext =
    (signed __CPROVER_bitvector[32])_phi_0;
    // 18 no location
    END_FUNCTION
```

2.7.2 Call

- Function call : Represents simple function call.
- Assume/Assert : Actual parameters of assume/assert are assumed to variables of boolean type or comparison instruction.
- llvm.dbg.declare : It is an intrinsic function. We use it to retain metadata of given variable.

2.8 Data types



Inheritance diagram of LLVM data types [10].

- Composite type
  - Sequential type
    * Array type : Array type arranges data of same type in sequential order. It requires data type and number of elements.
    * Vector type : Vector data type arranges data of same type in sequential order. The only difference is that vector type is used in Single Instruction Multiple Data(SIMD) instructions.
  - Structure type : A structure is the collection of data of different types.
    e.g.

    ```
    %struct.student = type { i32, [20 x i8] }
    ```

    This structure type consists of integer of size 32 bits and array of 20 elements of integer of size 8 bits.
- Function type : A function type has return type and types of arguments.
- Integer type : Integer type requires size in number of bits.
- Pointer type : Pointer type represents address/memory location of an element. (LLVM does not allow pointer to void).

2.9 Scoping information

Consider the following C program. It has two variables named 'x' in different scopes.

```
int main()
{
    int x = 5 ;
    {
            int x = 10 ;
            assert(x==10);
    }
    assert(x==5);
    return  0;
}
```

If we observe the LLVM IR generated for above program, LLVM has renamed one of variables as 'x1' as it does not preserve the scope information. Whereas scope information is preserved in goto program using name of symbol.

```
Symbol.....: main::1::1::x
Pretty name: main::1::1::x
Module.....: demo
Base name..: x
Mode.......: C
Type.......: signed int
Value......: 10

Symbol.....: main::1::x
Pretty name: main::1::x
Module.....: demo
Base name..: x
Mode.......: C
Type.......: signed int
Value......: 5
```

```
%x = alloca i32, align 4
%x1 = alloca i32, align 4
store i32 0, i32* %retval, align 4
call void @llvm.dbg.declare(metadata i32* %x,
metadata !10, metadata !11), !dbg !12
store i32 5, i32* %x, align 4, !dbg !12
call void @llvm.dbg.declare(metadata i32* %x1,
metadata !13, metadata !11), !dbg !15
store i32 10, i32* %x1, align 4, !dbg !15
%0 = load i32, i32* %x1, align 4, !dbg !16
%cmp = icmp eq i32 %0, 10, !dbg !17
%conv = zext i1 %cmp to i32, !dbg !17
%call = call i32 (i32, ...) bitcast (i32 (...)*
 @assert to i32 (i32, ...)*)(i32 %conv), !dbg !18
%1 = load i32, i32* %x, align 4, !dbg !19
%cmp2 = icmp eq i32 %1, 5, !dbg !20
%conv3 = zext i1 %cmp2 to i32, !dbg !20
%call4 = call i32 (i32, ...) bitcast (i32 (...)*
 @assert to i32 (i32, ...)*)(i32 %conv3), !dbg !21
```

This scope information can be regained using metadata. We can see that scope of the first variable is !6 i.e. main function and scope of second variable is !14 i.e. block at line 4 and column 5.

```
!6 = distinct !DISubprogram(name: "main", scope: !1, file: !1, line: 1,
    type: !7, isLocal: false, isDefinition: true, scopeLine: 2,
    isOptimized: false, unit: !0, variables: !2)
!9 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!10 = !DILocalVariable(name: "x", scope: !6, file: !1, line: 3, type: !9
    )
!11 = !DIExpression()
!12 = !DILocation(line: 3, column: 9, scope: !6)
!13 = !DILocalVariable(name: "x", scope: !14, file: !1, line: 5, type:
    !9)
!14 = distinct !DILexicalBlock(scope: !6, file: !1, line: 4, column: 5)
```

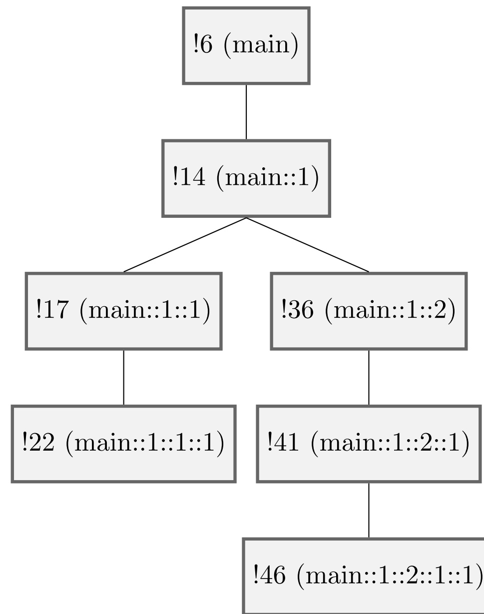
To convert the scope information in LLVM IR into goto format, we do following:

1. Find distinct scopes in a function.

```
!1 = !DIFile(filename: "scope.c", directory: "test")
!6 = distinct !DISubprogram(name: "main", scope: !1, file: !1, line:
     1, type: !7, isLocal: false, isDefinition: true, scopeLine: 2,
    isOptimized: false, unit: !0, variables: !2)
!14 = distinct !DILexicalBlock(scope: !6, file: !1, line: 4, column:
     5)
!17 = distinct !DILexicalBlock(scope: !14, file: !1, line: 6, column
    : 6)
!22 = distinct !DILexicalBlock(scope: !17, file: !1, line: 6, column
    : 6)
!27 = distinct !DILexicalBlock(scope: !22, file: !1, line: 6, column
    : 29)
!36 = distinct !DILexicalBlock(scope: !14, file: !1, line: 9, column
    : 6)
!41 = distinct !DILexicalBlock(scope: !36, file: !1, line: 9, column
    : 6)
!46 = distinct !DILexicalBlock(scope: !41, file: !1, line: 9, column
    : 28)
```

2. Construct tree of these scopes. Each node contains pointer to it's parent, left sibling, right sibling, first child and last child.

3. Fully qualified name for each scope is determined and stored in a map.


!6 (main)
!14 (main::1)
!17 (main::1::1)
!36 (main::1::2)
!22 (main::1::1::1)
!41 (main::1::2::1)
!46 (main::1::2::1::1)

Chapter 3

Summary and Future work

3.1 Summary

Current implementation of the tool has stable support for variable declaration, memory access instructions such as **Load**, **Store** and **GetElementPtr**, bitwise operations, typecast instructions and control flow instructions such as **Br** and **Switch**.
PHI, **BitCast** and function call are unstable.

3.2 Future work

LLVM2GOTO can be extended to support exception handling, parallel program, LLVM intrinsic functionsand Languages other than 'C'.

References

[1] https://llvm.org.

[2] R. Cytron, J. Ferrante, B. K. Rossen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, (1991) 451–490.

[3] http://www.cprover.org/.

[4] D. Kroening and M. Tautschnig. CBMC C Bounded Model Checker. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* 389–391.

[5] J. Morse. Expressive and efficient bounded model checking of concurrent software. Ph.D. thesis, University of Southampton 2015.

[6] M. Brain, S. Joshi, D. Kroening, and P. Schrammel. Safety Verification and Refutation by k-Invariants and k-Induction. *22nd International Static Analysis Symposium (SAS)* .

[7] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In Computer Aided Verification (CAV), LNCS. Springer To appear.

[8] A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-Aware Predicate Abstraction for Shared-Variable Concurrent Programs. In Proceedings of CAV, volume 6806 of *LNCS*. Springer, 2011 356–371.

[9] Z. Rakamari and M. Emmi. SMACK: Decoupling Source Language Details from Verifier Implementations. *International Conference on Computer Aided Verification* 13, (2014) 451–490.

[10] http://llvm.org/doxygen/classllvm_1_1Type.html.