Karlsruhe Institute of Technology

Bounded Model Checking of Software for Real-World Applications

Parts 1-3

UniGR Summer School on Verification Technology, Systems & Applications
VTSA 2018
Nancy, France

Carsten Sinz
Institute for Theoretical Informatics (ITI)
Karlsruhe Institute of Technology (KIT)

29.08.2018

The Bounded Model Checker LLBMC

- **LLBMC**
  - Bounded model checker for C programs
  - Developed at KIT
  - Successful in SV-COMP competitions
- **Functionality**
  - Integer overflow, division by zero, invalid bit shift
  - Illegal memory access (array index out of bound, illegal pointer access, etc.)
  - Invalid free, double free
  - User-customizable checks (via __llbmc_assume / __llbmc_assert)
- **Employed techniques**
  - Loop unrolling, function inlining; LLVM as intermediate language
  - SMT solvers, various optimizations (e.g. for handling array-lambda-expressions)

```
struct list_node {
    int data;
    struct list_node *tail;
};
typedef struct list_node list;

list *reverse(list *l) {
    list *r = l, *p = NULL;
    while (r != NULL) {
        list *q = r;
        r = r->tail;
        q->tail = p;
        p = q;
    }
    return p;
}
```

Overview

Wednesday, August 29:

Part 1: Introduction to LLVM

Part 2: Run-time errors in C (and C++)

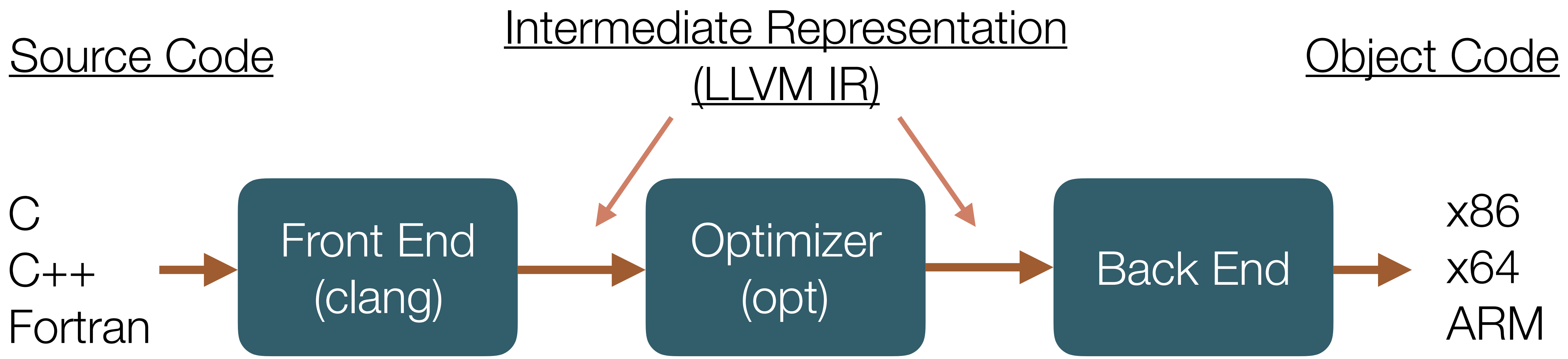
Part 3: Decision procedures for program arithmetic

Working in groups on exercises

Part 1:
Introduction to LLVM

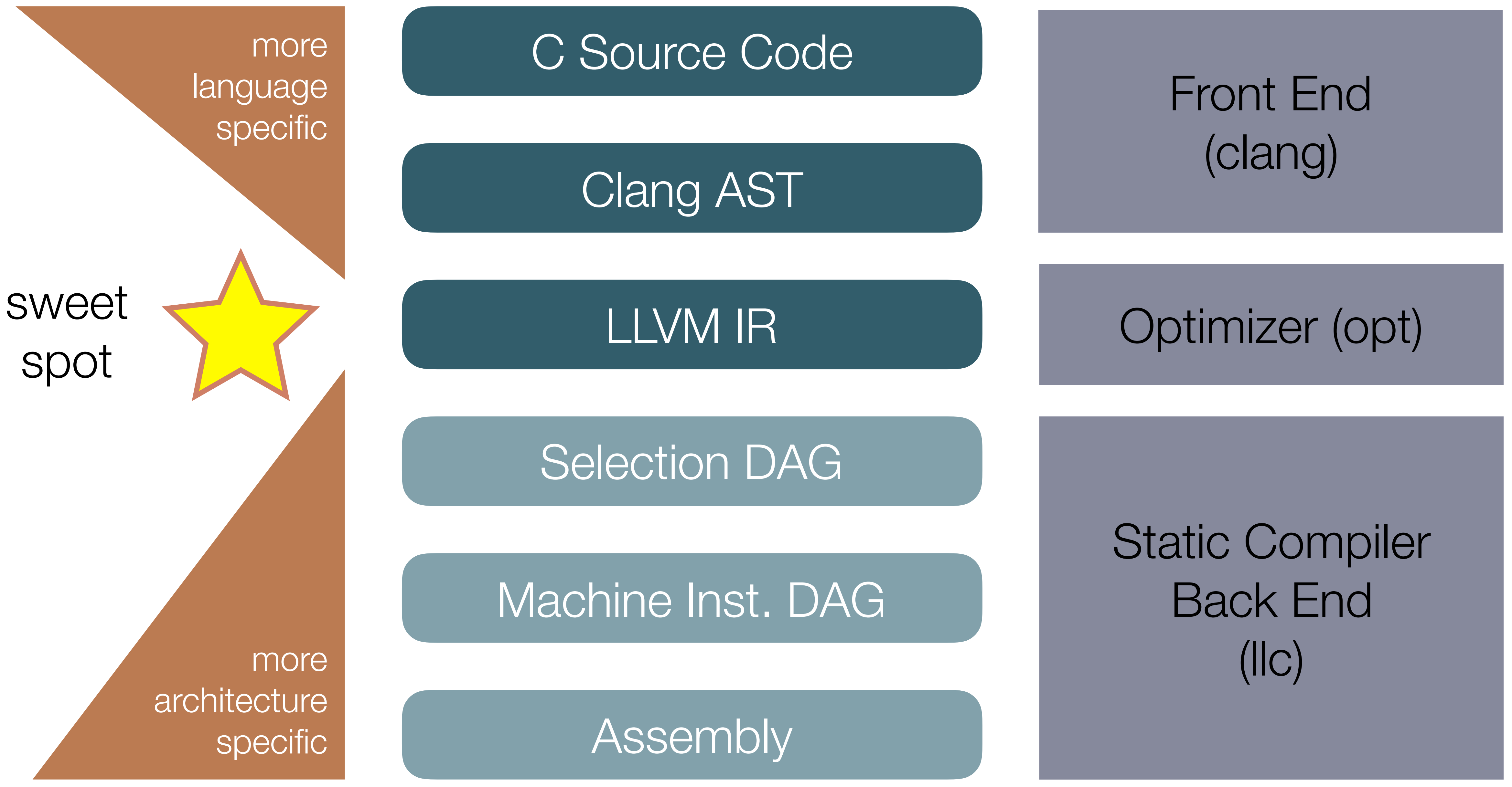
Slides adapted from Jonathan Burket, CMU

The LLVM Compiler Framework



- LLVM is a toolbox for constructing compilers and programming tools
- LLVM IR is a **virtual instruction set**, similar to an assembler language
  - Source code and object code independent (mostly)
  - Always in *Static Single Assignment* (SSA) form (facilitates analysis)
  - Used in many software analysis tools nowadays

LLVM: From Source to Binary



LLVM IR

$$\begin{tikzcd}
 & \text{In-memory data structure} \arrow[dl] \arrow[dr] & \\
\text{Bitcode (.bc files)} \arrow[rr, leftrightarrow, "\texttt{llvm-as}", "\texttt{llvm-dis}"'] & & \text{Text format (.ll files)}
\end{tikzcd}$$

```
4243C0DE 06103239
0A324424 18000000
E6C6211D 210C0000
9201840C 480A9021
98000000 E6A11CDA
```

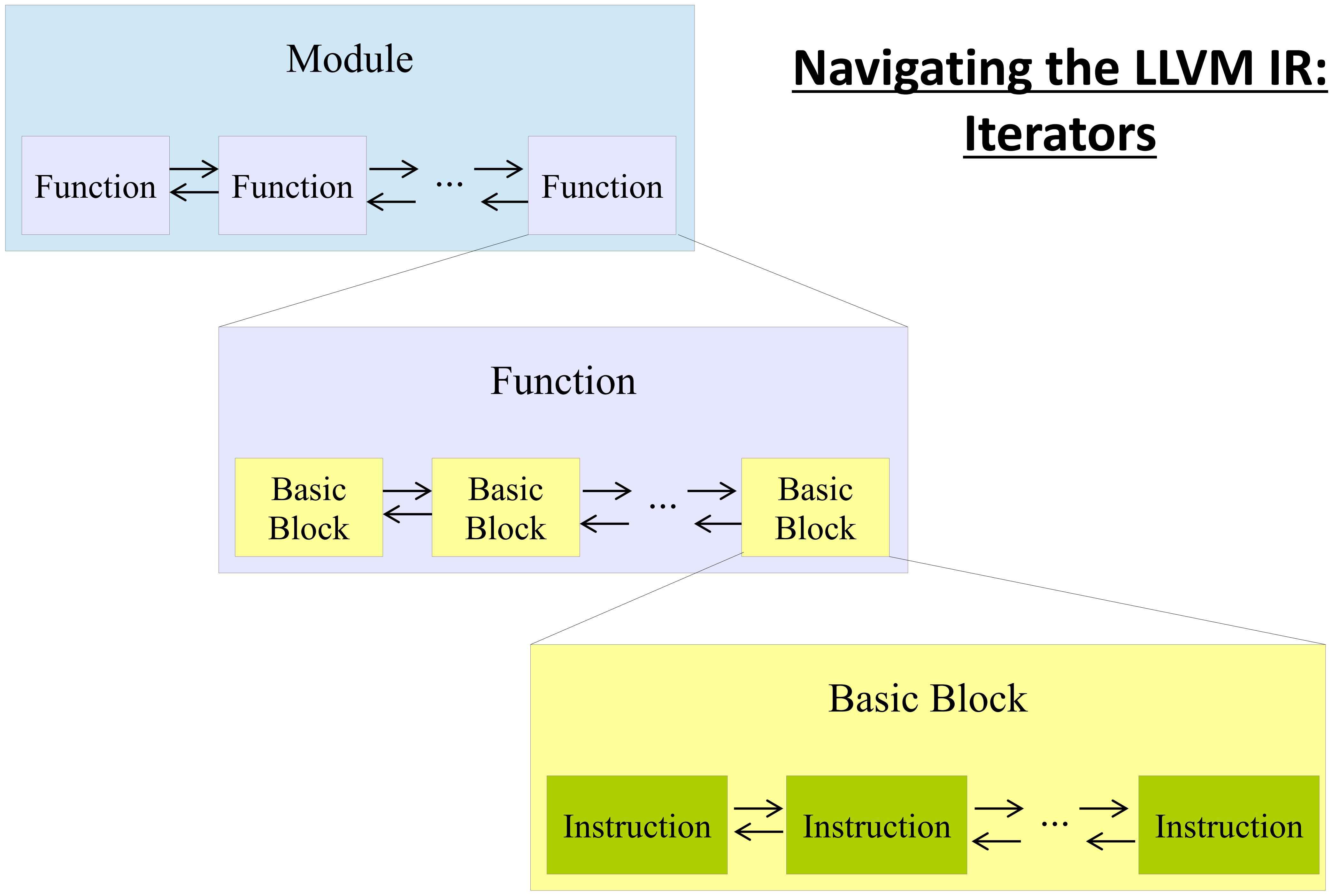
`llvm-as`

`llvm-dis`

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  …
```

- Bitcode files and LLVM IR text files are lossless serialization formats

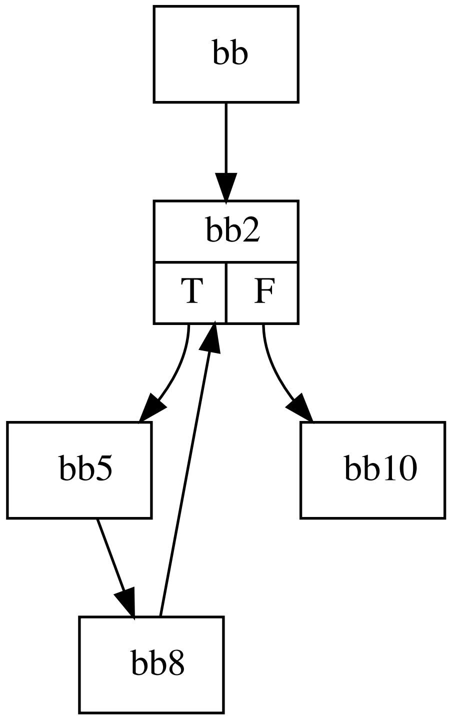
Structure of a Bitcode File (Module)



Navigating the LLVM IR: Iterators

Bitcode Example

```c
int next_power_of_two(int x)
{
    unsigned int i;
    x--;
    for(i=1; i < sizeof(int)*8; i *= 2)
        x = x | (x >> i);
    return x+1;
}
```



```
; ModuleID = 'next_power_of_two-opt.bc'
source_filename = "next_power_of_two.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

; Function Attrs: noinline nounwind ssp uwtable
define i32 @next_power_of_two(i32 %x) #0 {
bb:
  %x1 = add nsw i32 %x, -1
  br label %bb2

bb2:                                              ; preds = %bb8, %bb
  %i = phi i32 [ 1, %bb ], [ %i2, %bb8 ]
  %x2 = phi i32 [ %x1, %bb ], [ %x3, %bb8 ]
  %i1 = zext i32 %i to i64
  %cmp = icmp ult i64 %i1, 32
  br i1 %cmp, label %bb5, label %bb10

bb5:                                              ; preds = %bb2
  %sh = ashr i32 %x2, %i
  %x3 = or i32 %x2, %sh
  br label %bb8

bb8:                                              ; preds = %bb5
  %i2 = mul i32 %i, 2
  br label %bb2

bb10:                                             ; preds = %bb2
  %res = add nsw i32 %x2, 1
  ret i32 %res
}
```

LLVM Data Structures

- LLVM provides many optimized data structures:
  - `BitVector`, `DenseMap`, `DenseSet`, `ImmutableList`, `ImmutableMap`, `ImmutableSet`, `IntervalMap`, `IndexedMap`, `MapVector`, `PriorityQueue`, `SetVector`, `ScopedHashTable`, `SmallBitVector`, `SmallPtrSet`, `SmallSet`, `SmallString`, `SmallVector`, `SparseBitVector`, `SparseSet`, `StringMap`, `StringRef`, `StringSet`, `Triple`, `TinyPtrVector`, `PackedVector`, `FoldingSet`, `UniqueVector`, `ValueMap`
- STL works well in combination with LLVM data structures

LLVM Instructions and Values

```
int main()
{
  int x;
  int y = 2;
  int z = 3;
  x = y + z;
  y = x + z;
  z = x+y;
}
```

clang + mem2reg

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %add = add nsw i32 2, 3
  %add1 = add nsw i32 %add, 3
  %add2 = add nsw i32 %add, %add1
  ret i32 0
}
```

Instruction I: `%add1 = add nsw i32 %add, 3`

You can't "get" `%add1` from Instruction I.
Instruction is identified with the value `%add1`.

Operand (and result) type

Operand 1

Operand 2

LLVM Instructions and Values

```
int main()
{
  int x;
  int y = 2;
  int z = 3;
  x = y + z;
  y = x + z;
  z = x+y;
}
```

clang + mem2reg

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %add = add nsw i32 2, 3
  %add1 = add nsw i32 %add, 3
  %add2 = add nsw i32 %add, %add1
  ret i32 0
}
```

Instruction I: `%add1 = add nsw i32 %add, 3`

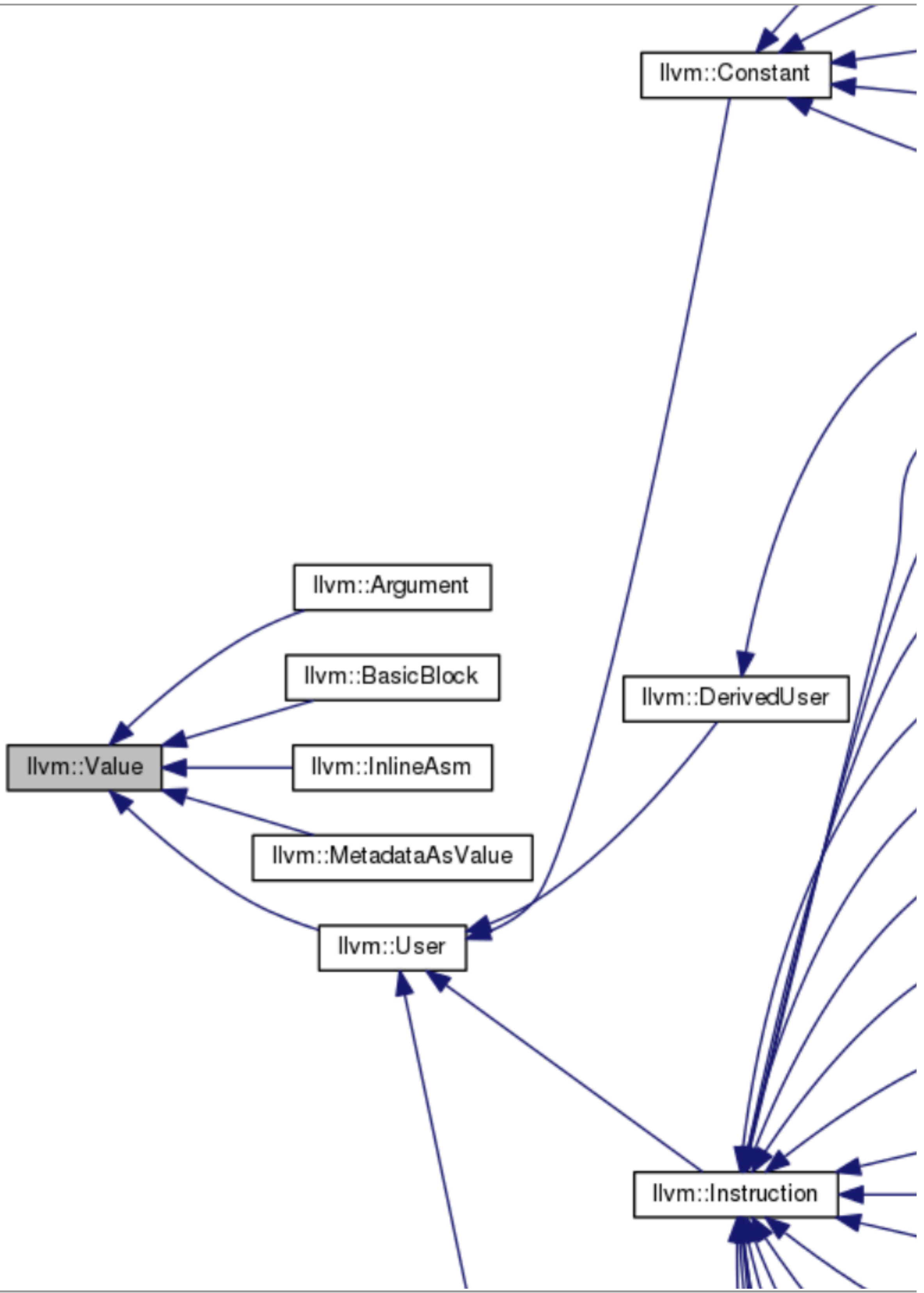
```
outs() << *I.getOperand(0);  ➞   "%add = add new i32 2, 3"
outs() << *I.getOperand(0)->getOperand(0);  ➞   "2"
```

This only makes sense for SSA form!

Casting and Type Introspection

Given a Value `*v`, what kind of Value is it?

- `isa<Argument>(v)`
  - Is `v` an instance of the `Argument` class?
- `Argument *v = cast<Argument>(v`
  - I know v is an Argument, perform the cast. Causes assertion failure if you are wrong.
- `Argument *v = dyn_cast<Argument>(v)`
  - Cast `v` to an Argument if it is an argument, otherwise return `nullptr`. Combines both `isa` and `cast` in one command.
- `dyn_cast` is not to be confused with the C++ `dynamic_cast` operator!



Casting and Type Introspection

```
void analyzeInstruction(Instruction * I)
{
    if (CallInst *CI = dyn_cast<CallInst>(I)) {
        outs() << “I’m a Call Instruction!\n”;
    }
    if (UnaryInstruction *UI = dyn_cast<UnaryInstruction>(I)) {
        outs() << “I’m a Unary Instruction!\n”;
    }
    if (CastInstruction *CI = dyn_cast<CastInstruction>(I)) {
        outs() << “I’m a Cast Instruction!\n”;

    }
    ...
}
```

Navigating the LLVM IR: Iterators

- `Module::iterator`
  - Modules are “program units”
  - Iterates through the functions of a module
- `Function::iterator`
  - Iterates through a function’s basic blocks
- `BasicBlock::iterator`
  - Iterates through the instructions in a basic block
- `Value::use_iterator`
  - Iterates through uses of a value
    (recall that instructions are treated as values)
- `User::op_iterator`
  - Iterates over the operands of an instruction (the “user” is the instruction)
  - Prefer to use convenient accessors defined in many instruction classes

Navigating the LLVM IR: Iterators

- **Iterate through every instruction in a function:**

```
for (Function::iterator FI = func->begin(), FE = func->end();
     FI != FE;
     ++FI) {
    for (BasicBlock::iterator BBI = FI->begin(), BBE = FI->end();
         BBI != BBE;
         ++BBI) {
        outs() << “Instruction: “ << *BBI << “\n”;
    }
}
```

- **Using `InstIterator` (Provided by “`llvm/IR/InstIterator.h`“):**

```
for (inst_iterator I = inst_begin(F), E = inst_end(F);
     I != E;
     ++I) {
    outs() << *I << "\n";
}
```

Navigating the LLVM IR: Iterators

- **Iterate through a basic block's predecessors:**

```
#include "llvm/Support/CFG.h"

BasicBlock *BB = ...;

for (pred_iterator PI = pred_begin(BB), E = pred_end(BB);
     PI != E;
     ++PI) {
    BasicBlock *Pred = *PI;
    // ...
}
```

Many further useful iterators are defined outside of Function, BasicBlock, etc.

Navigating the LLVM IR: Casting and Iterators

```
for (Function::iterator FI = func->begin(), FE = func->end();
     FI != FE;
     ++FI) {
    for (BasicBlock::iterator BBI = FI->begin(), BBE = FI->end();
         BBI != BBE;
         ++BBI) {
        Instruction *I = BBI;
        if (CallInst *CI = dyn_cast<CallInst>(I)) {
            outs() << “I’m a Call Instruction!\n”;
        }
        if (UnaryInstruction *UI = dyn_cast<UnaryInstruction>(I)) {
            outs() << “I’m a Unary Instruction!\n”;
        }
        if (CastInstruction * CI = dyn_cast<CastInstruction>(I)) {
            outs() << “I’m a Cast Instruction!\n”;
        }
        ...
    }
}
```

Very common code pattern

Navigating the LLVM IR: Visitor Pattern

```
struct MyVisitor : public InstVisitor<MyVisitor> {
    void visitCallInst(CallInst &CI) {
        outs() << “I’m a Call Instruction!\n”;
    }
    void visitUnaryInstruction(UnaryInstruction &UI) {
        outs() << “I’m a Unary Instruction!\n”;
    }
    void visitCastInst(CastInst &CI) {
        outs() << “I’m a Cast Instruction!\n”;
    }
    void visitMul(BinaryOperator &I) {
        outs() << “I’m a multiplication Instruction!\n”;
    }
}

MyVisitor MV;
MV.visit(F);
```

No need for iterators or casting

A given instruction only triggers one method: a `CastInst` will not call `visitUnaryInstruction` if `visitCastInst` is defined.

You can opt out on operators too, (even if there isn’t a specific class for them)

LLVM Pass Manager

- **Compiler is organized as a series of “passes”:**
  - Each pass is one analysis or transformation
- **Seven types of passes:**
  - `ImmutablePass`: doesn’t do much
  - `LoopPass`: process loops
  - `RegionPass`: process single-entry, single-exit portions of code
  - `ModulePass`: general inter-procedural pass
  - `CallGraphSCCPass`: bottom-up on the call graph
  - `FunctionPass`: process a function at a time
  - `BasicBlockPass`: process a basic block at a time
- **Constraints imposed (e.g. FunctionPass):**
  - FunctionPass can only look at “current function”
  - Cannot maintain state across functions

Useful LLVM Passes: `mem2reg`

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

```
define i32 @main() #0 {
entry:
  %sub = sub nsw i32 5, 3
  ret i32 %sub
}
```

Not always possible: Sometimes stack operations are too complex

What mem2reg Cannot Handle

```
int main(int argc, char *argv[])
{
    int vals[4] = {2,4,8,16};
    int x = 0;
    vals[1] = 3;
    x += vals[0];
    x += vals[1];
    x += vals[2];
    return x;
}
```

What `mem2reg` Cannot Handle

```
@main.vals = private unnamed_addr constant [4 x i32]
                        [i32 2, i32 4, i32 8, i32 16], align 4

define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %vals = alloca [4 x i32], align 4
  %0 = bitcast [4 x i32]* %vals to i8*
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* %0,
    i8* bitcast ([4 x i32]* @main.vals to i8*), i32 16, i32 4, i1 false)
  %arrayidx = getelementptr inbounds [4 x i32]* %vals, i32 0, i32 1
  store i32 3, i32* %arrayidx, align 4
  %arrayidx1 = getelementptr inbounds [4 x i32]* %vals, i32 0, i32 0
  %1 = load i32* %arrayidx1, align 4
  %add = add nsw i32 0, %1
  %arrayidx2 = getelementptr inbounds [4 x i32]* %vals, i32 0, i32 1
  %2 = load i32* %arrayidx2, align 4
  %add3 = add nsw i32 %add, %2
  %arrayidx4 = getelementptr inbounds [4 x i32]* %vals, i32 0, i32 2
  %3 = load i32* %arrayidx4, align 4
  %add5 = add nsw i32 %add3, %3
  ret i32 %add5
}
```

Other Useful Passes

- **Simplify CFG (`-simplifycfg`)**
  - Removes unnecessary basic blocks by merging unconditional branches if the second block has only one predecessor
  - Removes basic blocks with no predecessors
  - Eliminates `phi` nodes for basic blocks with a single predecessor, removes unreachable blocks
- Loop Information (`-loops`)
  - Reveals the basic blocks in a loop; headers and pre-headers; exiting blocks; back edges; “canonical induction variable”; loop count
- **Scalar Evolution (`-scalar-evolution`)**
  - Tracks changes to variables through nested loops
- **Alias Analyses**
  - If you know that different names refer to different locations, you have more freedom to reorder code, etc. Also helps a lot in making code analysis more scalable
- **Naming of values (`-instnamer`)**

Useful LLVM Documentation

- **LLVM Programmer's Manual**

  `http://llvm.org/docs/ProgrammersManual.html`
- **LLVM Language Reference Manual**

  `http://llvm.org/docs/LangRef.html`
- **Writing an LLVM Pass**

  `http://llvm.org/docs/WritingAnLLVMPass.html`
- **LLVM's Analysis and Transform Passes**

  `http://llvm.org/docs/Passes.html`
- **LLVM Internal Documentation**

  `http://llvm.org/doxygen`

Useful LLVM Command Lines

- Generating bitcode from a C program:

```
> clang -c -g -emit-llvm prog.c
```

- Run optimizer passes `mem2reg` and `instnamer` on bitcode file:

```
> opt -mem2reg -instnamer prog.bc -o prog-opt.bc
```

- Viewing a bitcode file (converting it to `.ll` format)

```
> llvm-dis -o - prog.bc | less
```

- Viewing the AST of a C program:

```
> clang -cc1 -ast-dump prog.c
```

- Viewing the CFG / call graph of a bitcode file:

```
> opt -dot-cfg[-only] prog.bc          > opt -dot-callgraph prog.bc
```

- Building a program based on LLVM:

```
> clang++ -g myprog.cpp `llvm-config --cxxflags --ldflags --system-libs \
   --libs core` -O3 -o myprog
```

LLVM IR – Instruction Groups

Instruction Group	Members
Terminator instructions	**ret**, **br**, **switch**, **indirectbr**, invoke, resume, catchswitch, catchret, cleanupret, **unreachable**
Binary operations	**add**, **fadd**, **sub**, **fsub**, **mul**, **fmul**, **udiv**, **sdiv**, **fdiv**, **urem**, **srem**, **frem**
Bitwise binary operations	**shl**, **lshl**, **ashr**, **and**, **or**, **xor**
Vector operations	extractelement, insertelement, shufflevector
Aggregate operations	**extractvalue**, **insertvalue**
Memory access and addressing operations	**alloca**, **load**, **store**, fence, cmpxchg, atomicrmw, **getelementptr**
Conversion operations	**trunc**, **zext**, **sext**, **fptrunc**, **fpext**, **fptoui**, **fptosi**, **uitofp**, **sitofp**, **ptrtoint**, **inttoptr**, **bitcast**, addrspacecast
Other instructions	**icmp**, **fcmp**, **phi**, **select**, **call**, va_arg, landingpad, catchpad, cleanuppad

LLVM IR – Intrinsic Functions

Group	Intrinsics (`llvm.*`)
Variable argument handling	**`va_start`**, **`va_end`**, **`va_copy`**
Garbage collection	`gcroot`, `gcread`, `gcwrite`
Code generator	`returnaddress`, `addressofreturnaddress`, `frameaddress`, `localescape`, `localrecover`, `read_register`, `write_register`, `stacksave`, `stackrestore`, `get.dynamic.area.offset`, `prefetch`, `pcmarker`, `readycyclecounter`, `clear_cache`, `instrprof.increment`, `instrprof.value.profile`, `llvm.thread.pointer`
Standard C library	**`memcpy`**, **`memmove`**, **`memset`**, **`sqrt`**, **`powi`**, **`sin`**, **`cos`**, **`pow`**, **`exp`**, **`exp2`**, **`log`**, **`log10`**, **`log2`**, **`fma`**, **`fabs`**, `minnum`, `maxnum`, `copysign`, `floor`, `ceil`, `trunc`, `rint`, `nearbyint`, `round`
Bit manipulation	`bitreverse`, `bswap`, `ctpop`, `ctlz`, `cttz`, `fshl`, `fshr`
Arithmetic with overflow	**`sadd.with.overflow`**, **`uadd.with.overflow`**, **`ssub.with.overflow`**, **`usub.with.overflow`**, **`smul.with.overflow`**, **`umul.with.overflow`**
Misc	many more…

Part 2:
Run-Time Errors in C (and C++)

What is an Error?

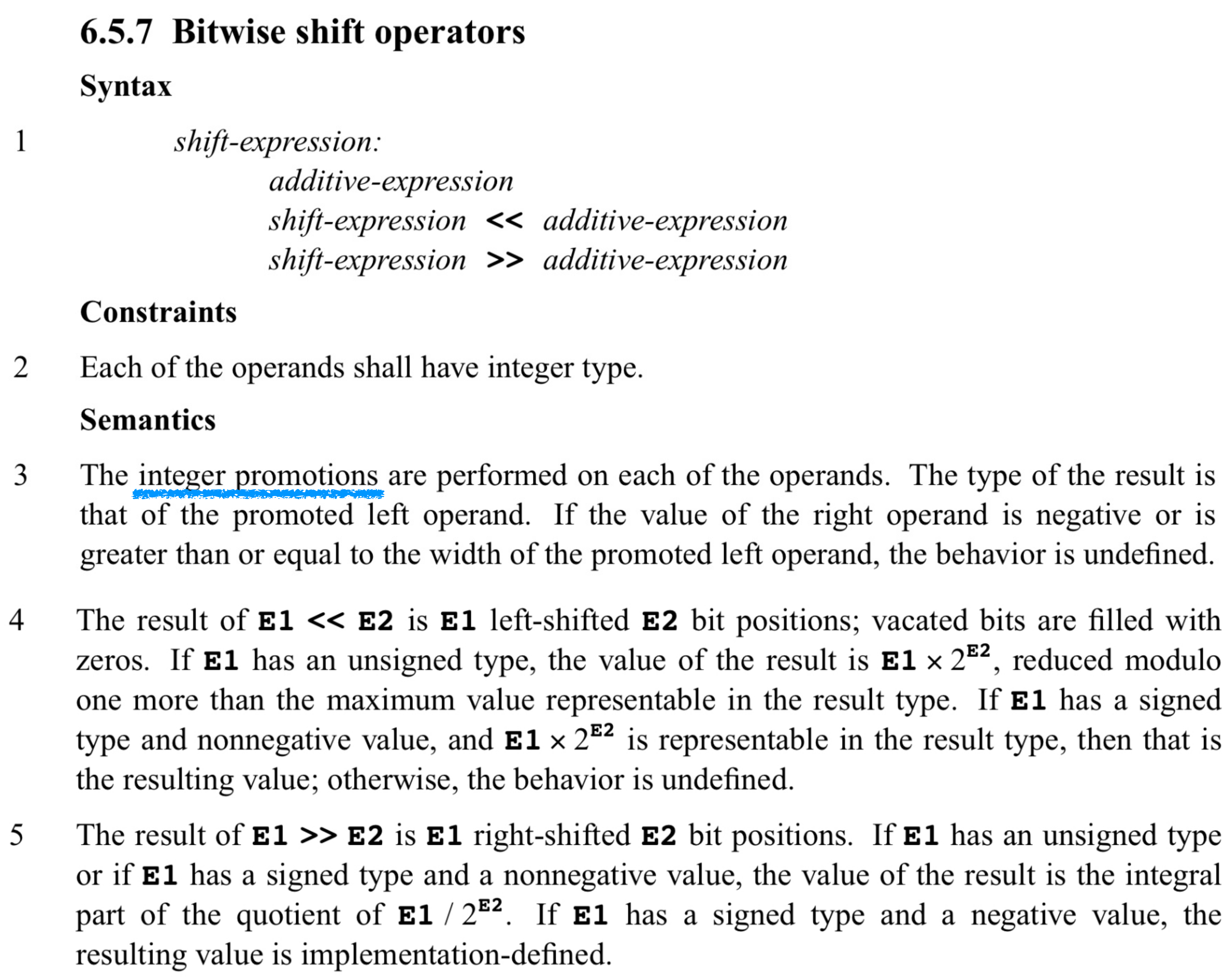
- **C Standard distinguishes:**
  - Unspecified: "standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance"
  - Implementation-defined: "semantics is defined by the implementation at hand"
  - Undefined: "anything might happen"

Property	Behavior
Arithmetic overflow (unsigned)	Ok (wrap-around)
Arithmetic ove	efined
Type cast: U ->	fined if V is signed, vise ok
Shift (2nd arg. n	efined
Shift (1st ar	ned if >>, undefined <<

```
unsigned int x = 0;
int y = -1;
if (y > x) {
  printf("surprise!");
}
```

- **May add: unexpected behavior**

C Standard (C99)



6.5.7 Bitwise shift operators

Syntax

1 *shift-expression:*
  additive-expression
  shift-expression **<<** *additive-expression*
  shift-expression **>>** *additive-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 The result of **`E1 << E2`** is **`E1`** left-shifted **`E2`** bit positions; vacated bits are filled with zeros. If **`E1`** has an unsigned type, the value of the result is $\mathbf{E1} \times 2^{\mathbf{E2}}$, reduced modulo one more than the maximum value representable in the result type. If **`E1`** has a signed type and nonnegative value, and $\mathbf{E1} \times 2^{\mathbf{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 The result of **`E1 >> E2`** is **`E1`** right-shifted **`E2`** bit positions. If **`E1`** has an unsigned type or if **`E1`** has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $\mathbf{E1} / 2^{\mathbf{E2}}$. If **`E1`** has a signed type and a negative value, the resulting value is implementation-defined.

C Standard: Integer Promotions

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

Every integer type has an *integer conversion rank* defined as follows:

(…)

If an **`int`** can represent all values of the original type, the value is converted to an **`int`**; otherwise, it is converted to an **`unsigned int`**. These are called the *integer promotions*.[48] All other types are unchanged by the integer promotions.

48) The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary **`+`**, **`-`**, and **`~`** operators, and to both operands of the shift operators, as specified by their respective subclauses.

- …

C Standard: Usual Arithmetic Conversions

6.3.1.8 Usual arithmetic conversions

1 Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a *common real type* for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the *usual arithmetic conversions*:

(…)

Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:

If both operands have the same type, then no further conversion is needed.

Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

$$\begin{tikzcd}
\texttt{long long} \arrow[r] & \texttt{unsigned long long} \\
\texttt{long} \arrow[u] \arrow[r] & \texttt{unsigned long} \arrow[u] \arrow[ul] \\
\texttt{int} \arrow[u] \arrow[r] & \texttt{unsigned int} \arrow[u] \arrow[ul] \\
\texttt{short} \arrow[u] \arrow[r] & \texttt{unsigned short} \arrow[u] \arrow[ul] \\
\texttt{char} \arrow[u] \arrow[r] & \texttt{unsigned char} \arrow[u] \arrow[ul]
\end{tikzcd}$$

Why Undefined Behavior?

- Allows the compiler to assume that some circumstances will never occur in a "conforming program"
- Gives the compiler more information about code
- Can lead to more optimization opportunities
- Example:

```
int foo(unsigned char x)
{
    int value = 2147483600;
    value += x;
    if (value < 2147483600) {
        bar();
    return value;
}
```

"Program Arithmetic"

```
unsigned int
square_check(unsigned int x)
{
    unsigned int y = x * x;
    if (y == 33) { error(); }
    return y;
}
```

Is `error()` reachable?

Has $x^2 \equiv 33 \mod 2^{32}$ a solution?

Yes!
4 Solutions, e.g. 663169809

Part 3:
Decision Procedures for Program Arithmetic

Algebraic Properties

Mathematical Integers vs. Signed vs. Unsigned

Addition

Property	ℤ	signed int (if defined)	unsigned int
Closure	yes	yes	yes
Associativity `a+(b+c) = (a+b)+c`	yes	yes	yes
Commutativity `a+b = b+a`	yes	yes	yes
Ex. of identity `a+0 = a`	yes	yes	yes
Ex. of inverse `a+(-a) = 0`	yes	yes	**no**

Multiplication

Property	ℤ	signed int (if defined)	unsigned int
Closure	yes	yes	yes
Associativity `a*(b*c) = (a*b)*c`	yes	yes	yes
Commutativity `a*b = b*a`	yes	yes	yes
Ex. of identity `a*1 = a`	yes	yes	yes
Ex. of inverse $a*(a^{-1}) = 1$	only 1 and -1	only 1 and -1	**all odd numbers**

- ℤ: commutative ring with unity; integral domain (no zero divisors); Euclidian domain (division with remainder)
- $\mathbb{Z}/2^k\mathbb{Z}$: also commutative ring with unity, but no integral domain (for k>1)

Arithmetic in $\mathbb{Z}/2^k\mathbb{Z}$

- Definition:

  $\mathbb{Z}/n\mathbb{Z} = \{\bar{a}_n \mid a \in \mathbb{Z}\}$ with $\bar{a} = \{\ldots, a-n, a, a+n, \ldots\}$
- As usual, we identify $\bar{a}$ with a, where $0 \leq a < n$, thus

  $\mathbb{Z}/2^k\mathbb{Z} = \{0,\ldots,2^k - 1\}$
- Examples of arithmetic in $\mathbb{Z}/2^k\mathbb{Z}$:
  - When has the equation $a \cdot x = b$ a solution? Is it unique?
  - Has the equation $x^2 = 33$ a solution in $\mathbb{Z}/2^8\mathbb{Z}$? Is it unique?
- Basic facts:
  - $\sum_{i=1}^{n} a_i x_i \equiv b \pmod{m}$ is solvable for the unknowns x_i, iff the greatest common divisor of $\{a_1, \ldots, a_n, m\}$ divides b.
  - a has a multiplicative inverse $\bmod\ m$, iff $\gcd(a, m) = 1$.
  - a^{-1} can be computed using the extended Euclidian algorithm or using Euler's theorem, $a^{-1} \equiv a^{\phi(m)-1} \pmod{m}$. For $m = 2^k$, $\phi(m) = \phi(2^k) = 2^{k-1}$, and thus $a^{-1} \equiv a^{2^{k-1}-1} \pmod{2^k}$.

Solving Equations in $\mathbb{Z}/2^k\mathbb{Z}$

- **Given:** Polynomial $p(x)$
- **Goal:** Solutions of $p(x) \equiv 0 \mod 2^k$
- First, consider the linear case: $p(x) = a \cdot x - b$, i.e. solving the equation $a \cdot x = b$ modulo $m = 2^k$.
- If a is invertible, then $x = b \cdot a^{-1}$ is the (unique) solution. (This is the case, if a is odd.)
- Otherwise, $a \cdot x = b$ has solutions, iff $\gcd(a, 2^k) \mid b$. The solution is not unique, but a particular solution is given by $x = b/a$.
- **Theorem:** *The congruence ax ≡ b (mod m) is soluble in integers if, and only if, gcd(a, m) | b. The number of incongruent solutions modulo m is gcd(a, m).*
- How can we find all solutions?
- For all solutions x, the following holds: $\exists t\,.\,ax + tm = b$. Having a first solution x$_0$, all solutions are given by $x_k = x_0 + k \cdot (m/\gcd(a,m))$ for $0 \leq k < \gcd(a,m)$.

Solving Systems of Linear Congruences

- Given a system $S = \{E_j\}$ of linear congruences (mod m = 2^k) over n variables, with

$$E_j : \sum_{i=1}^{n} a_{ji}x_i \equiv b_j \mod 2^k \quad ,$$

find its solution set.

- Algorithm [Ganesh, 2007]:
  - If there is an odd coefficient a_{ji}, solve equation E_j for x_i and substitute x_i in all other equations. If E_j cannot be solved for x_i, i.e. if $\gcd\{a_{j1}, \ldots, a_{jn}, m\} \nmid b_j$, then there is no solution to S.
  - If all coefficients a_{ji} are even, divide all a_{ji}, b_j by two and decrease k by one.
  - Repeat the algorithm with the resulting system of congruences and stop with "success" if there is only one solved equation left.
- Properties:
  - The algorithm is a sound and complete decision procedure for linear congruences.

Solving Systems of Linear Congruences

- Example: Solve the following system of congruences modulo 8:

$$\begin{aligned} 3x + 4y + 2z &= 0 \\ 2x + 2y &= 6 \\ 4y + 2x + 2z &= 0 \end{aligned}$$

- Note:
  - Ganesh considers the unknowns as bit-vectors of length k; when the system is divided by 2, the highest bit in each bit-vector is dropped (i.e. left unconstrained)
- Question:
  - How can the set of all solutions of S be determined after the algorithm finished?

Solving Non-Linear Congruences

- **Task:** Given a polynomial p(x), find all solutions of $p(x) \equiv 0 \mod 2^k$.
- **Hensel lifting algorithm** (special case for m = 2^k):
  1. [k=1] Check, whether $p(x) \equiv 0 \mod 2$ has a solution. If not, exit with "no solution".
  2. [k→k+1] Let {x_i} be the set of solutions for $p(x) \equiv 0 \mod 2^k$. We distinguish two cases to lift each x_i from k to k+1:
     A. If $p'(x_i) \equiv 0 \mod 2$: [0 or 2 lifted solutions]
        1. If $p(x_i) \not\equiv 0 \mod 2^{k+1}$, x_i cannot be lifted
        2. Otherwise there are two lifted solutions $x_i^* = x_i + t \cdot 2^k, \ t \in \{0,1\}$
     B. If $p'(x_i) \not\equiv 0 \mod 2$: [unique lifting]
        $x_i^* = x_i - p(x_i)$

- **Note:** Hensel-lifting also works for multivariate polynomials. However, already the base case (k=1) is NP-complete. (Why?)

Solving Non-Linear Congruences

- **Example:** $x^2 \equiv 33 \mod 2^4$
- $p(x) = x^2 - 33, \quad p'(x) = 2x$
- **[k=1, mod 2]:** $x^2=1$ mod 2 has solution x*=1
- **[k=2, mod 4]:** Try to lift x*=1: p'(x*)=0 mod 2, thus 0 or 2 lifted solutions p(x*)=0 mod 4, thus 2 liftings: x*'= x*+2t = {1, 3}
- **[k=3, mod 8]:**
  - Lifting x*=1: 0 or 2 lifted solutions, p(x*)=0 mod 8, x*' = { 1, 5 }
  - Lifting x*=3: 0 or 2 lifted solutions, p(x*)=0 mod 8, x*' = { 3, 7 }
- **[k=4, mod 16]:**
  - Lifting x*=1: p(x*)=0 mod 16, x*' = { 1, 9 }
  - Lifting x*=3: p(x*)=8 mod 16, no lifting
  - Lifting x*=5: p(x*)=8 mod 16, no lifting
  - Lifting x*=7: p(x*)=0 mod 16, x*' = { 7, 15 }

Summary

- **LLVM:**
  - SSA, iterators, passes
- **Undefined behavior:**
  - Allows for optimization
  - Conversion rules error prone
- **Modular arithmetic:**
  - Decision procedures for
    - multivariate linear congruences
    - univariate polynomial congruences

References

- **Chris Lattner: What Every C Programmer Should Know About Undefined Behavior**
  - http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
- **Juneyoung Lee et al.: Taming Undefined Behavior in LLVM (PLDI 2017)**
- **SEI CERT C Coding Standard (CMU)**
  - https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard
- **LLVM UndefinedBehaviorSanitizer**
  - Run-time analysis tool
  - https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- **Vijay Ganesh: Decision Procedures for Bit-Vectors, Arrays and Integers (PhD Thesis, 2007)**