The LLVM compiler framework

Exploring LLVM

Daniele Cattaneo

Politecnico di Milano

2020-04-25

These slides were originally written by Michele Scandale, Ettore Speziale and Stefano Cherubin for the “Code Transformation and Optimization” course.

Contents

LLVM official documentation

llvm.org/docs

A lot of documentation...

llvm.org/docs links to:

- 4 references about *Design & Overview*
- 6 references about *Getting Started / Tutorials*
- 35 references about *User Guides*
- 15 references about *Programming Documentation*
- 40 references about *Reference Documentation*
- 7 references about *Development Process Documentation*
- 5 Mailing Lists
- 4 IRC bots

Most of the above references are **outdated**!

You probably need documentation *about the documentation*.

Essential documentation

Intro to LLVM [1]	Quick and clear introduction to the compiler infrastructure. Mostly up-to-date.*
Writing an LLVM pass [2]	Explains step by step how to implement a Pass for those who never did anything like that. (We will see this tutorial later in the course)
Doxygen [3]	*The best code documentation is the code itself.* Sometimes the generated doxygen documentation is enough. Updated to the latest development branch, refer to github branches for documentation about the stable versions.
llvm-dev [3]	Mailing List. Last resource: ask other developers. Warning: It has very high traffic.

*At the time I am writing!

Contents

Canonicalizing Pass Input

We will see the following passes:

Pass	Switch
Variable promotion	`mem2reg`
Loop simplification	`loop-simplify`
Loop-closed SSA	`lcssa`
Induction variable simplification	`indvars`

They are **normalization** passes:

- they convert the code into a canonical form

Contents

Variable Promotion

One of the most difficult things in compilers is
handling memory accesses.

Plain SAXPY (Scalar $ax + y$)

```
define float @saxpy(float %a, float %x, float %y) {
entry:
  %a.addr = alloca float, align 4
  %x.addr = alloca float, align 4
  %y.addr = alloca float, align 4
  store float %a, float* %a.addr, align 4
  store float %x, float* %x.addr, align 4
  store float %y, float* %y.addr, align 4
  %0 = load float, float* %a.addr, align 4
  %1 = load float, float* %x.addr, align 4
  %mul = fmul float %0, %1
  %2 = load float, float* %y.addr, align 4
  %add = fadd float %mul, %2
  ret float %add
}
```

Variable Promotion

In the SAXPY kernel all the variables are **`alloca`**ted on the stack!

- Function arguments included!

They are allocated like that because the compiler follows a **conservative** approach:

- an instruction could take the address of one of the variables...

However, complex representations make optimizations more difficult:

- suppose you want to compute the `a*x+y` expression using only **one** instruction (aka FMA4)
- hard to detect due to **`load`** and **`store`**

Variable Promotion

To limit the number of instruction accessing memory we need to eliminate **`load`** and **`store`**

- achieved by **promoting** variables from memory to registers

Inside the LLVM-IR:

memory Stack allocations
`%1 = alloca float, align 4`

register SSA variables
`%a`

The `mem2reg` pass focus on:

- eliminating **`alloca`** used only by **`load`** and **`store`** instructions

Also available as a utility function:

- `llvm::PromoteMemToReg`
  - see `llvm/Transforms/Utils/PromoteMemToReg.h`

Variable Promotion

Starting Point

```
%1 = alloca float
%2 = alloca float
%3 = alloca float
store %a, %1
store %x, %2
store %y, %3
%4 = load %1
%5 = load %2
%6 = fmul %4, %5
%7 = load %3
%8 = fadd %6, %7
ret %8
```

(copy propagation is performed transparently by the compiler)

Promoting `alloca`

```
%1 = %a
%2 = %x
%3 = %y
%4 = %1
%5 = %2
%6 = fmul %4, %5
%7 = %3
%8 = fadd %6, %7
ret %8
```

After Copy-propagation

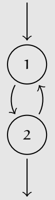
```
%1 = fmul %a, %x
%2 = fadd %1, %y
ret %2
```

Contents

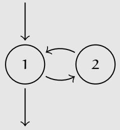
Loops

There are several kind of loops:

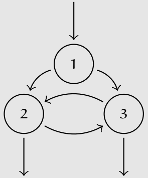
`do-while` Loops



`while` Loops



Irreducible Loops



LLVM focuses on one class of loop:
Natural Loops

Natural Loops

A natural loop:

- has only one entry node – the *header*
- there is a back edge that enters the loop header

Under this definition:

- the irreducible loop example is not a natural loop
- since LLVM consider only natural loops, the irreducible loop example **is not recognized** as a loop

Loop Terminology

Loops are defined starting from the back-edges:

back-edge edge entering loop header: $(3, 1)$

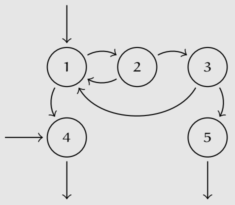
header loop entry node: 1

body nodes that can reach back-edge source node (3) without passing from back-edge target node (1) plus back-edge target node: $\{1, 2, 3\}$

exiting nodes with a successor outside the loop: $\{1, 3\}$

exit nodes with a predecessor inside the loop: $\{4, 5\}$

A loop



Loop Simplify

Natural loops are

- easy to **identify**
- not really analysis/optimization friendly!

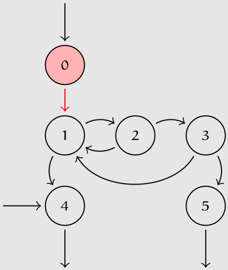
The `loop-simplify` pass normalizes natural loops:

pre-header ensures the **loop header** has **a single entry edge**

latch ensures the loop has **a single back-edge**

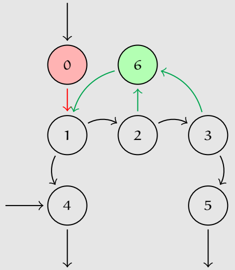
exit-block ensures **exits dominated** by loop **header**

Pre-header Insertion

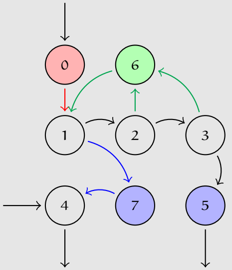


Loop Simplify

Latch Insertion



Exit-block Insertion



- pre-header always executed before entering the loop
- latch always executed before starting a new iteration
- exit-blocks executed only after exiting the loop

Contents

Loop-closed SSA

Loop representation can be further normalized:

- `loop-simplify` normalizes the **shape** of the loop *(control flow)*
- it does not involve the instructions in the loop *(data flow)*

Keeping SSA form is expensive with loops:

- Any optimization involving an SSA variable **defined inside the loop**, and **used outside the loop**, causes a ripple effect!

The `lcssa` transformation is the solution:

- inserts **`phi`** instructions at loop boundaries
- now, optimizations performed inside the loop do not affect the code outside of it

Loop-closed SSA

Linear Search

```
int *search(int *x, int n, int y)
{
  int j = -1;
  for (int i = 0; i < n; i++)
    if (x[i] == y)
      j = i;
  return j;
}
```

The example is trivial, this transformation is mostly useful for *large loop bodies*.

Loop-closed SSA

Before LCSSA

```
for.cond:
  %j.0 = phi i32 [ -1, %entry ], [ %j.1, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end

for.body:
  [...]

if.end:
  %j.1 = phi i32 [ %i.0, %if.then ], [ %j.0, %for.body ]
  br label %for.inc

for.inc:
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:
  ret i32 %j.0
```

Loop-closed SSA

After LCSSA

```
for.cond:
  %j.0 = phi i32 [ -1, %entry ], [ %j.1, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end

for.body:
  [...]

if.end:
  %j.1 = phi i32 [ %i.0, %if.then ], [ %j.0, %for.body ]
  br label %for.inc

for.inc:
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:
  %j.0.lcssa = phi i32 [ %j.0, %for.cond ]
  ret i32 %j.0.lcssa
```

Contents

Induction Variables

Some loop variables are *special*:

- e.g. counters

The generalization of this intuition are **induction variables**:

- `foo` is a **loop induction variable**
  if its successive values form an arithmetic progression:

  $$\texttt{foo} = \texttt{bar} * \texttt{baz} + \texttt{biz}$$

  where: `bar`, `biz` are loop-invariant [*],
  `baz` is an induction variable
- `foo` is a **canonical induction variable**
  if it is always incremented by a constant amount:

  $$\texttt{foo} = \texttt{foo} + \texttt{biz}$$

  where `biz` is loop-invariant

---

[*] Constants inside the loop

Induction Variable Simplification

Canonical induction variables are often used to **drive** loop execution.

Given a loop, the `indvars` pass tries to transform its induction variables into **canonical** induction variables.

- It also transforms loop exit conditions in simple inequalities
- Definition of other variables derived from the induction variables are moved outside the loop if used there

LLVM defines canonical induction variables as:

- initialized to `0`
- incremented by `1` at each loop iteration

Contents

Normalization

"Standard" running order:

1. `mem2reg`: limits use of memory
2. `loop-simplify`: canonicalizes loops
   - Improved detection of a lot of standard patterns!
3. `lcssa`: keeps effects of subsequent loop optimizations local
   limits overhead of maintaining SSA form
4. `indvars`: normalizes induction variables
   simplifies and highlightsthe loop condition

For more normalization passes:

- try running `opt -help`

Contents

Checking Input Properties

Analyses basically allow to:

- **derive** information and properties of the input
- **verify** properties of input

Keeping analyzed information updated is expensive:

- tuned algorithms update information when an optimization invalidates it
- incrementally updating analyses are cheaper than recomputing them

As an **optimization**, many LLVM analysis supports incremental updates.

Useful Analyses

We will see the following passes:

Pass	Switch	Transitive
Control flow graph	—	No
Dominator tree	`domtree`	No
Post-dominator tree	`postdomtree`	No
Loop information	`loops`	Yes
Scalar evolution	`scalar-evolution`	Yes
Alias analysis	—	Yes
Memory SSA	`memoryssa`	Yes

Requesting an Analysis

Your pass needs to tell the pass manager which analyses it needs!

Transitive analyses:
`llvm::AnalysisUsage::addRequiredTransitive<T>()`

Non-transitive analyses:
`llvm::AnalysisUsage::addRequired<T>()`

For **chained analyses***, the `addRequiredTransitive` method should be used instead of the `addRequired` method.

This informs the PassManager that the transitively required pass should be alive as long as the requiring pass is.

---

*Analyses that use the result of another analysis

Contents

Control Flow Graph

The Control Flow Graph is implicitly maintained by LLVM:

- no specific pass to build it

Recap:

- CFG for a function is a graph of basic blocks
- a basic block is a list of instructions

Functions and basic blocks act like containers:

- STL-like accessors: `front()`, `back()`, `size()`, ...
- STL-like iterators: `begin()`, `end()`
  - Warning for BBs: order of iteration $\neq$ order of execution!

Each contained element is aware of its container:

- `getParent()`

Control Flow Graph

Every CFG has an **entry** basic block:

- the **first** executed basic block
- it is the **root/source** of the graph
- get it with `llvm::Function::getEntryBlock()`

At the end of a basic blocks there's always a **terminator** instruction:

- **`ret`**, **`br`**, **`switch`**, **`unreachable`**, …

More than one **exit** block can be present in a function:

- they are the **leaves/sinks** of the graph
- their terminator instructions are always **`ret`**s
  1. `llvm::BasicBlock::getTerminator()`
  2. check the opcode of the terminator

Side Note

For performance reasons, a custom casting framework is used:

- you cannot use **`static_cast`** and **`dynamic_cast`** with types/classes provided by LLVM

LLVM Casting Functions

Static cast of `Y*` to `X`	`X *llvm::cast<X>(Y *)`
Dynamic cast of `Y*` to `X`	`X *llvm::dyn_cast<X>(Y *)`
Is `Y*` an instance of `X`?	`bool llvm::isa<X>(Y *)`

Example:

- is `BB` a sink?
  `llvm::isa<llvm::ReturnInst>(BB.getTerminator())`

Control Flow Graph

Every basic block `BB` has one or more*:

predecessors from `pred_begin(BB)` to `pred_end(BB)`

successors from `succ_begin(BB)` to `succ_end(BB)`

Other convenience methods available in `llvm::BasicBlock`:

- useful getters
  - `BasicBlock *getUniquePredecessor()`
  - …
- moving a basic block
  - `moveBefore(llvm::BasicBlock *)`
  - `moveAfter(llvm::BasicBlock *)`
- split a basic block:
  - `splitBasicBlock(llvm::BasicBlock::iterator)`
- …

*see include/llvm/IR/CFG.h

Control Flow Graph

The `llvm::Instruction` class defines common operations:

- getting an operand
  - `getOperand(unsigned)`

Subclasses provide specialized accessors:

- the **`load`** instruction takes as operand the pointer to the memory to be loaded:
  - `llvm::LoadInst::getPointerOperand()`

Instructions

Instructions are created using:

- constructors
  - `llvm::LoadInst::LoadInst(...)`
- factory methods
  - `llvm::GetElementPtrInst::Create(...)`
- the helper class `llvm::IRBuilder`
  - `llvm::IRBuilder<> builder(insPoint);`
    `builder.CreateAdd(...);`

Interface is not homogeneous!
Some instructions support all methods, others support only one.

Instructions

Instructions can be inserted:

- automatically by `IRBuilder`
  - insertion point is given at `IRBuilder` instantiation
- manually by appending to a basic block
- manually by inserting after/before another instruction

From Control Flow to Data Flow

In LLVM, the data flow generated by the various instructions is represented by a simple hierarchy:

value	something that can be used: `llvm::Value`
user	something that can use: `llvm::User`
use	the link between the **value** and the **user**: `llvm::Use`

A value is a **definition**:

- Visiting where a definition is used:
  - `llvm::Value::use_begin()`, `llvm::Value::use_end()`

An user accesses **definitions**:

- Visiting the definitions that are used:
  - `llvm::User::op_begin()`, `llvm::User::op_end()`

From Control Flow to Data Flow

- `llvm::Value` inherits from `llvm::User`
- `llvm::Instruction` inherits from `llvm::Value`
  - ⇒ The value produced by the instruction is the **instruction itself**!

Example

```
%6 = load i32, i32* %1, align 4
```

The **`load`** is described by an instance of `llvm::Instruction`. That instance also represents the `%6` variable.

Not all instances of `llvm::Value` are also `llvm::Instruction`s!

i.e. function arguments

From Control Flow to Data Flow

Every `llvm::Value` is typed:

- use `llvm::Value::getType()` to get the type

Since every instruction is a value:

- instructions are typed

Example

```
%6 = load i32, i32* %1, align 4
```

The type of the `%6` variable is the type of the return value of the **`load`** instruction, `i32`

Contents

Dominance Trees

Dominance trees answer to control-related queries:

is `A` executed **before** `B`?	is `A` executed **after** `B`?
$\Downarrow$	$\Downarrow$
`llvm::DominatorTree`	`llvm::PostDominatorTree`

The interfaces of these two trees is mostly the same:

- `bool dominates(A, B)`
- `bool properlyDominates(A, B)`

`A` and `B` are either `llvm::BasicBlocks` or `llvm::Instructions`

By using `opt`, it is possible to show the trees:

- `-view-dom`, `-dot-dom`
- `-view-postdom`, `-dot-postdom`

Contents

Loop Information

Loop information is represented using two classes:

`llvm::LoopInfo` The result of `llvm::LoopAnalysis`, performed on a given function.

`llvm::Loop` Represents a single loop in a function. Contained inside a `llvm::LoopInfo`.

Using `llvm::LoopInfo` it is possible:

- navigate through top-level loops:
  - `llvm::LoopInfo::begin()`
  - `llvm::LoopInfo::end()`
- get the loop for a given basic block:
  - `llvm::LoopInfo::operator[](llvm::BasicBlock *)`

Loop Information

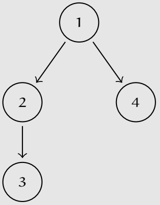
Loops are represented as a **tree**:

Source

```
while(i < 10) {       // loop 1
  while(j < 10)       // loop 2
    while(k < 10)     // loop 3
      ...

  while(h < 10)       // loop 4
    ...
}
```

Loop Hierarchy



children loops `llvm::Loop::begin()`, `end()`

parent loop `llvm::Loop::getParentLoop()`

Loop Information

Accessors for important nodes:

pre-header	`llvm::Loop::getLoopPreheader()`
header	`llvm::Loop::getHeader()`
latch	`llvm::Loop::getLoopLatch()`
exiting	`llvm::Loop::getExitingBlock()`, `llvm::Loop::getExitingBlocks(...)`
exit	`llvm::Loop::getExitBlock()` `llvm::Loop::getExitBlocks(...)`

The list of all BBs in the loop is accessible via:

iterators	`llvm::Loop::block_begin()`, `llvm::Loop::block_end()`
vector	`std::vector<BasicBlock *> &Loop::getBlocks()`

Contents

Scalar Evolution

The **SC**alar **EV**olution pass analyzes scalar expressions inside loops.

- all expressions are categorized and represented uniformly
- is capable of handling **general induction variables**
- also useful outside of loops
- `opt` flags: `-analyze -scalar-evolution`

Example

```
for.cond:
  %i.0 = phi [ 0, %entry ], [ %i.inc, %for.inc ]
  %cond = icmp ne %i.0, 10
  br %cond, label %for.body, label %for.end
for.inc:
  %i.inc = add nsw %i.0, 1
  br label %for.cond
for.end:
  ...
```

SCEV for `%i.0`:

- initial value 0
- incremented by 1 at each iteration
- final value 10

Scalar Evolution

Source

```
void foo() {
  int bar[10][20];

  for(int i = 0; i < 10; ++i)
    for(int j = 0; j < 20; ++j)
      bar[i][j] = 0;
}
```

SCEV `{A,B,C}<%D>`:

- `A` starting value
- `B` operator
- `C` stride
- `D` loop head BB

`{0,+,1}=0+1+1+1+…`

Induction Variables

```
%i.0 = phi i32 [ 0, %entry ], [ %inc6, %for.inc5 ]
-->  {0,+,1}<nuw><nsw><%for.cond>      Exits: 10
%j.0 = phi i32 [ 0, %for.body ], [ %inc, %for.inc ]
-->  {0,+,1}<nuw><nsw><%for.cond1>     Exits: 20
```

Scalar Evolution

The scalar evolution framework manages **any scalar expression**:

Pointer SCEVs in two nested loops

```
  %arrayidx = getelementptr {...} %bar, i32 0, i32 %i.0
  -->  {%bar,+,80}<nsw><%for.cond>
Exits: {%bar,+,80}<nsw><%for.cond>

  %arrayidx4 = getelementptr {...} %arrayidx, i32 0, i32 %j.0
  -->  {{%bar,+,80}<nsw><%for.cond>,+,4}<nsw><%for.cond1>
Exits: {(80 + %bar),+,80}<nsw><%for.cond>
```

SCEV is an analysis used by many common optimizations

- induction variable substitution
- strength reduction
- vectorization
- …

Scalar Evolution

SCEVs are modeled by the `llvm::SCEV` class:

- a subclass for each kind of SCEV: e.g. `llvm::SCEVAddExpr`
- instantiation disabled

A SCEV actually is a tree of SCEVs:

- `{(80 + %bar),+,80}` =
  - `{%1,+,80}`
  - `%1 = 80 + %bar`

Tree leaves:

constant `llvm::SCEVConstant`: e.g. `80`

unknown* `llvm::SCEVUnknown`: e.g. `%bar`

SCEV tree explorable through the visitor pattern:

- `llvm::SCEVVisitor`

---

*Not further splittable

Scalar Evolution

The `llvm::ScalarEvolutionAnalysis` pass computes all the SCEVs for a given `llvm::Function`.

The `llvm::ScalarEvolution` instance produced by the pass provides the following services:

- get the SCEV representing a value:
  - `getSCEV(llvm::Value *)`
- get important SCEVs from other structures or SCEVs:
  - `getBackedgeTakenCount(llvm::Loop *)`
  - `getPointerBase(llvm::SCEV *)`
  - …
- create new SCEVs explicitly:
  - `getConstant(llvm::ConstantInt *)`
  - `getAddExpr(llvm::SCEV *, llvm::SCEV *)`
  - …

Contents

Alias Analysis

Let `X` be an instruction accessing a memory location:

- is there another instruction accessing the same location?

Alias analysis tries to answer the question:

application optimization of memory operations

problem often fails

Different algorithms are available for alias analysis:

- common interface: `llvm::AAResults`
- base implementation: basic alias analysis (`basicaa`)

Requiring Alias Analysis

```
AU.addRequiredTransitive<AAResultsWrapperPass>();
```

Alias Analysis

Source

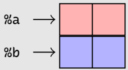
```
%1 = load i16, i16* %a
%2 = load i16, i16* %b
store i16 %2, i32* %a
store i16 %1, i32* %b
```

Basic building block:
`llvm::MemoryLocation`

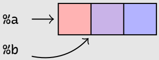
Encapsulates a tuple:
(**address**, **size**)

Can be computed from a
`llvm::Value`

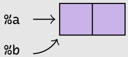
Distinct Locations



Overlapping Locations



Same Location



Alias Analysis

Given two memory locations X, Y, the alias analyzer classifies them:

- **`llvm::AliasResult::NoAlias`**
  X and Y **are different** memory locations
- **`llvm::AliasResult::MustAlias`**
  X and Y **are equal** – i.e. they points to the same address
- **`llvm::AliasResult::PartialAlias`**
  X and Y **partially overlap** – i.e. they points to different addresses, but the pointed memory areas partially overlap
- **`llvm::AliasResult::MayAlias`**
  unable to compute aliasing information – i.e. X and Y can be different locations, or X can be a complete/partial alias of Y

Queries performed using:

- `llvm::AAResults::alias(X, Y)`

Alias Analysis

A different categorization involves whether an instruction I **reads and/or modifies** a memory location X:

- **`llvm::ModRefInfo::NoModRef`**
  The access neither references nor modifies the value stored in X
- **`llvm::ModRefInfo::Ref`**
  The access may reference the value stored in X
- **`llvm::ModRefInfo::Mod`**
  The access may modify the value stored in X
- **`llvm::ModRefInfo::ModRef`**
  The access may reference and may modify the value stored in X

Queries performed using:

- `llvm::AAResults::getModRefInfo(I, X)`

Alias Analysis

This interface is very low-level!

What if we wanted to compute all aliases of a single value X?

To do this, LLVM provides the `llvm::AliasSet` class:

1. instantiate a new `llvm::AliasSetTracker` starting from `llvm::AAResults`*
2. it builds (one or more) `llvm::AliasSet`

For a given location X, a `llvm::AliasSet`:

- contains all locations aliasing with X

*`using llvm::AliasAnalysis = llvm::AAResults;`

Alias Analysis

Alias sets return memory reference and aliasing information just like the low-level interface.

Warning: This information is **less precise**, as it is derived by **conservatively aggregating** more detailed data!

- **`bool llvm::AliasSet::isRef()`**
  memory accessed in read-mode – e.g. a **`load`** is inside the set
- **`bool llvm::AliasSet::isMod()`**
  memory accessed in write-mode – e.g. a **`store`** is inside the set

- **`bool llvm::AliasSet::isMustAlias()`**
  all pointers in the set `MustAlias` with each other
- **`bool llvm::AliasSet::isMayAlias()`**
  at least one pair of pointer is not a `MustAlias` pair

Alias Analysis

Entry point is
`llvm::AliasSetTracker::getAliasSetFor(...)`

Only argument is a reference to
`llvm::MemoryLocation`

Once you have the `llvm::AliasSet` you can inspect the list of memory locations in it with the standard C++ iterator pattern:

`size()`, `begin()`, `end()`

Contents

Memory SSA

The `llvm::MemorySSAAnalysis` pass wraps alias analysis to answer queries in the following form:

- let `%foo` be an instruction accessing memory. Which preceding instructions does `%foo` depends on?

This is done by representing all memory accesses in a **SSA-like form**:

- **`store`**-like instructions become **definitions** (`MemoryDef`)
- **`load`**-like instructions become **uses** (`MemoryUse`)
- **`store`**s to the same location in parallel CFG branches become **phi**s (`MemoryPhi`)

Memory Dependence Analysis

MemorySSA “instructions” are owned by `llvm::MemorySSA` objects.

They are **overlaid** on top of the normal CFG.

- `AccessList *getBlockAccesses(BasicBlock *)`
- `DefsList *getBlockDefs(BasicBlock *)`

This basic interface is very **hard to use**:

- `llvm::MemorySSAWalker` provides support for the most common query
- `MemoryAccess *getClobberingMemoryAccess(...)`
  - Returns the nearest dominating memory access that clobbers the same memory location given.

Contents

Conclusions

Inside LLVM there a lot of passes:

normalization put program into a canonical form

analysis get info about the program

Please remember that:

- a good compiler writer **re-uses** code
- check LLVM sources before re-implementing a pass

Contents

Bibliography I

- Chris Lattner.
  Intro to LLVM.
  http://www.aosabook.org/en/llvm.html.
- Chris Lattner and Jim Laskey.
  Writing an LLVM Pass.
  http://llvm.org/docs/WritingAnLLVMPass.html.
- LLVM Community.
  Doxygen annotations.
  http://llvm.org/doxygen/annotated.html.
- Chris Lattner and Vikram Adve.
  LLVM Language Reference Manual.
  http://llvm.org/docs/LangRef.html.

Bibliography II

- LLVM Community.
  LLVM Coding Standards.
  http://llvm.org/docs/CodingStandards.html.
- LLVM Community.
  LLVM Passes.
  http://llvm.org/docs/Passes.html.
- LLVM Community.
  Autovectorization in LLVM.
  http://llvm.org/docs/Vectorizers.html.
- LLVM Community.
  LLVM Programmer's Manual.
  http://llvm.org/docs/ProgrammersManual.html.