LLVM Passes

Michele Scandale

Politecnico di Milano

04-04-2012

This material is strongly based on Ettore Speziale's material for the previous year course.

Contents

What is Available Inside LLVM?

LLVM provides passes performing basic transformations:

- variables promotion
- loops canonicalization
- ...

They can be used to normalize/canonicalize the input:

- transform into a form analyzable for further passes

Input normalization is essential:

- keep passes implementation manageable

LLVM IR Language
Static Single Assignment

LLVM IR is SSA-based:

- every variable is statically assigned exactly once

Statically means that:

- inside each function
- for each variable `%foo`
- there is only one statement in the form `%foo = ...`

Static is different from dynamic:

- a static assignment can be executed more than once

Static Single Assignment
Examples

Scalar SAXPY

```
float saxpy(float a, float x, float y) {
    return a * x + y;
}
```

Scalar LLVM SAXPY

```
define float @saxpy(float %a, float %x, float %y) {
  %1 = fmul float %a, %x
  %2 = fadd float %1, %y
  ret float %2
}
```

Temporary `%1` not reused! `%2` is used for the second assignment!

Static Single Assignment

Examples

Array SAXPY

```
void saxpy(float a, float x[4], float y[4], float z[4]) {
  for(unsigned i = 0; i < 4; ++i)
    z[i] = a * x[i] + y[i];
}
```

Array LLVM SAXPY

```
for.cond:
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i32 %i.0, 4
  br i1 %cmp, label %for.body, label %for.end

  ...

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

One assignment for loop counter `%i.0`

Static Single Assignment
Handling Multiple Assignments

Max

```
float max(float a, float b) {
  return a > b ? a : b;
}
```

LLVM Max – Bad

```
  %1 = fcmp ogt float %a, %b
  br i1 %1, label %if.then, label %if.else
if.then:
  %2 = %a
  br label %if.end
if.else:
  %2 = %b
  br label %if.end
if.end:
  ret float %2
```

Why is it bad?

Static Single Assignment

Use **phi** to Avoid Troubles

The `%5` variable must be statically set once

LLVM Max

```
  %1 = fcmp ogt float %a, %b
  br i1 %1, label %if.then, label %if.end
if.then:
  br label %if.end
if.else:
  br label %if.end
if.end:
  %2 = phi float [ %a, %if.then ], [ %b, %if.else ]
  ret float %2
```

The **phi** instructions is a *conditional move*:

- it takes $(variable_i, label_i)$ pairs
- if coming from predecessor identified by $label_i$, return $variable_i$

Static Single Assignment
Definition and Uses

Each SSA variable is set only once:

- variable definition

Each SSA variable can be used by multiple instructions:

- variable uses

Algorithms and technical language abuse of these terms:

Let `%foo` be a variable. If `%foo` definition has not side-effects, and no uses, dead-code elimination can be efficiently performed by erasing `%foo` definition from the CFG.

Static Single Assignment
Rationale

Old compilers are not SSA-based:

- putting input into SSA-form is expensive
- cost must be amortized

New compilers are SSA-based:

- SSA easier to work with
- SSA-based analysis/optimizations faster

All modern compilers are SSA-based:

- exception is HotSpot Client compiler

Contents

Canonicalize Pass Input

We will see the following passes:

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

They are normalization passes:

- put data into a canonical form

Variable Promotion

One of the most difficult things in compiler is:

- considering memory accesses

Plain SAXPY

```
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* %a.addr, align 4
  %1 = load float* %x.addr, align 4
  %mul = fmul float %0, %1
  %2 = load float* %y.addr, align 4
  %add = fadd float %mul, %2
  ret float %add
}
```

Variable Promotion

Simplifying Representation

In the SAXPY kernel some `alloca` are generated:

- represent local variables [1]

They are generated due to compiler conservative approach:

- maybe some instruction can take the addresses of such variables, hence a memory location is needed

Complex representations makes hard performing further actions:

- suppose you want to compute `a * x + y` using only one instruction [2]
- hard to detect due to `load` and `store`

[1] Arguments are local variables
[2] e.g. FMA4

Variable Promotion
Using Memory Only When Necessary

To limit the number of instruction accessing memory:

- we need to eliminate **`load`** and **`store`**
- achieved by promoting variables from memory to registers

Inside LLVM SSA-based representation:

- memory: Stack allocations – e.g `%1 = alloca float, align 4`
- register: SSA variables – e.g. `%a`

The **`mem2reg`** pass focus on:

- eliminating **`alloca`** with only **`load`** and **`store`** uses

Also available as utility:

- `llvm::PromoteMemToReg`[3]

[3] see Transforms/Utils/PromoteMemoryToRegister.cpp

Variable Promotion

Example

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 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
```

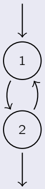
After Copy-propagation

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

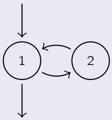
Loops

Different kind of loops:

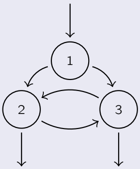
`do-while` Loops



`while` Loops



Irreducible Loops



In LLVM the focus is on one kind of loop:

- natural loops

Natural Loops

A natural loop:

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

Under this definition:

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

Loop Terminology

Loops defined starting from back-edges:

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

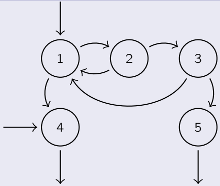
header loop entry node: 1

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 finding is the base pass identify loops, but:

- some features are not analysis/optimization friendly

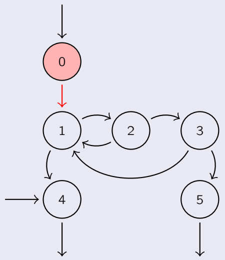
The **`loop-simplify`** pass normalize natural loops:

pre-header the only predecessor of header node

latch the starting node of the only back-edge

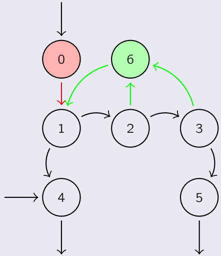
exit-block ensures exits dominated by loop header

Pre-header Insertion

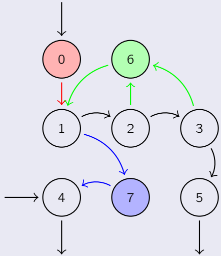


Loop Simplify
Example

Latch Insertion



Exit-block Insertion



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

Loop-closed SSA

Loop representation can be further normalized:

- `loop-simplify` normalize the shape of the loop
- nothing is said about loop definitions

Keeping SSA form is expensive with loops:

- `lcssa` insert **`phi`** instruction at loop boundaries for variables defined inside the loop body and used outside
- this guarantee isolation between optimization performed inside and outside the loop
- faster keeping IR into SSA form – propagation of code changes outside the loop blocked by **`phi`** instructions

Loop-closed SSA

Example

Linear Search

```
unsigned search(float *x, unsigned n, float y) {
  unsigned i, j = 0;
  for(i = 0; i != n; ++i)
    if(x[i] == y)
      j = i;
  return j;
}
```

The example is trivial:

- think about having large loop bodies
- transformation becomes useful

Loop-closed SSA

Example

Before LCSSA

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

  ...

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

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

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

Loop-closed SSA
Example

After LCSSA

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

  ...

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

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

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

Induction Variables

Some loop variables are *special*:

- e.g. counters

Generalization lead to induction variables:

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

  ```
  foo = bar * baz + biz
  ```

  where `bar`, `biz` are loop-invariant [4], and `baz` is an induction variable
- `foo` is a canonical induction variable if it is always incremented by a constant amount:

  ```
  foo = foo + biz
  ```

  where `biz` is loop-invariant

[4] Constants inside the loop

Induction Variable Simplification

Canonical induction variables are used to drive loop execution:

- given a loop, the **`indvars`** pass tries to find its canonical induction variable

With respect to theory, LLVM canonical induction variable is:

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

Normalization

Wrap-up

Normalization passes running order:

1. `mem2reg`: limit use of memory, increasing the effectiveness of subsequent passes
2. `loop-simplify`: canonicalize loop shape, lower burden of writing passes
3. `lcssa`: keep effects of subsequent loop optimizations local, limiting overhead of maintaining SSA form
4. `indvars`: normalize induction variables, highlighting the canonical induction variable

Other normalization passes available:

- try running `opt -help`

Contents

Checking Input Properties

Analysis basically allows to:

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

Keeping analysis information is expensive:

- tuned algorithms updates analysis information when an optimization invalidates them
- incrementally updating analysis is cheaper than recomputing them

Many LLVM analysis supports incremental updates:

- this is an optimization
- forget this feature for the home-work
- focus on information provided by analysis

Useful Analysis

We will see the following passes:

Analysis

Pass	Switch	Transitive
Control flow graph	`none`	No
Dominator tree	`domtree`	No
Post-dominator tree	`postdomtree`	No
Loop information	`loops`	Yes
Scalar evolution	`scalar-evolution`	Yes
Alias analysis	`special`	Yes
Memory dependence	`memdep`	Yes

Requiring analysis by transitivity:

yes `llvm::AnalysisUsage::addRequiredTransitive<T>()`

no `llvm::AnalysisUsage::addRequired<T>()`

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 set of basic blocks
- a basic block is a set of instructions

Functions and basic blocks acts like containers:

- STL-like accessors: `front()`, `back()`, `size()`, ...
- STL-like iterators: `begin()`, `end()`

Each contained element is aware of its container:

- `getParent()`

Control Flow Graph
Walking

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()`

More than one exit blocks can be generated:

- their terminator instructions are **`ret`**s
- they are the leaves/sinks of the graph
- use `llvm::BasicBlock::getTerminator()` to get the terminator ...
- ...then check its real class

Side Note
Casting Framework

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

Meaning	Function
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 `X`?	`bool llvm::isa<X>(Y *)`

Example:

- is `BB` a sink?

```
llvm::isa<llvm::ReturnInst>(BB.getTerminator())
```

Control Flow Graph

Basic Blocks

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)`

Convenience accessors directly available in `llvm::BasicBlock`:

- e.g. `llvm::BasicBlock::getUniquePredecessor()`

Other convenience member functions:

- moving a basic block: `llvm::BasicBlock::moveBefore(llvm::BasicBlock *)` or `llvm::BasicBlock::moveAfter(llvm::BasicBlock *)`
- split a basic block: `llvm::BasicBlock::splitBasicBlock(llvm::BasicBlock::iterator)`
- ...

Control Flow Graph

Instructions

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

- e.g. getting an operand: `llvm::Instruction::getOperand(unsigned)`

Subclasses provide specialized accessors:

- e.g the **`load`** instruction takes an operand that is a pointer:
  `llvm::LoadInst::getPointerOperand()`

The value produced by the instruction is the instruction itself:

Example

Consider:

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

the **`load`** is described by an instance of `llvm::LoadInst`. That instance also models the `%6` variable

Instructions

Creating New Instructions

Instructions built using:

- constructors – e.g. `llvm::LoadInst::LoadInst(...)`
- factory methods – e.g. `llvm::GetElementPtrInst::Create(...)`

Interface is not homogeneous:

- some instructions support both methods
- others support only one

At build-time, instructions can be:

- appended to a basic block
- inserted after/before a given instruction

Insertion point usually specified as builder last argument

Side Note
Definitions and Uses

LLVM class hierarchy is built around two simple concepts:

value something that can be used: `llvm::Value`

user something that can use: `llvm::User`

A value is a definition:

- `llvm::Value::use_begin()`, `llvm::Value::use_end()` to visit uses

An user access definitions:

- `llvm::User::op_begin()`, `llvm::User::op_end()` to visit used values

Functions:

- used by call sites
- uses formal parameters

Instructions:

- define an SSA value
- uses operands

Side Note
Value Typing

Every `llvm::Value` is typed:

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

Since every instructions is/define a value:

- instructions are typed

Example

Consider:

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

the `%6` variable actually is the instruction itself. Its type is the type of **`load`** return value, `i32`

Dominance Trees

Dominance trees answer to control-related queries:

- is this basic block executed before that?
- `llvm::DominatorTree`
- is this basic block executed after that?
- `llvm::PostDominatorTree`

The two trees interface is similar:

- `bool dominates(X *, X *)`
- `bool properlyDominates(X *, X *)`

Where `X` is an `llvm::BasicBlock` or an `llvm::Instruction`

Using `opt` is possible printing them:

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

Loop Information

Loop information are represented using two classes:

- `llvm::LoopInfo` analysis detects natural loops
- `llvm::Loop` represents a single loop

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
Nesting Tree

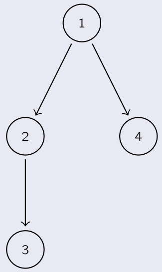
Loops are represented in a nesting tree:

Source

```
while(i < 10) {
  while(j < 10)
    while(k < 10)
      ...

  while(h < 10)
    ...
}
```

Loop Nest



Nest navigation:

- children loops: `llvm::Loop::begin()`, `llvm::Loop::end()`
- parent loop: `llvm::Loop::getParentLoop()`

Loop Information
Query Loops

Accessors for relevant nodes also available:

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

Loop basic blocks accessible via:

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

Scalar Evolution

The SCalar EVolution framework:

- represents scalar expressions
- supports recursive updates
- lower burden of explicitly handling expressions composition
- is designed to support general induction variables

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
Example

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` initial
- `B` operator
- `C` operand
- `D` defining BB

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

More than Induction Variables

The scalar evolution framework manages any scalar expression:

Pointer SCEVs

```
rrayidx = 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}<nw><%for.cond>
```

SCEV is an analysis used for common optimizations:

- induction variable substitution
- strength reduction
- vectorization
- ...

Scalar Evolution
SCEVs Design

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 [5] `llvm::SCEVUnknown`: e.g. `%bar`

SCEV tree explorable through the visitor pattern:

- `llvm::SCEVVisitor`

[5] Not further splittable

Scalar Evolution

Analysis Interface

The `llvm::ScalarEvolution` class:

- analyzes SCEVs for a `llvm::Function`
- builds SCEVs for values:

  `llvm::ScalarEvolution::getSCEV(llvm::Value *)`
- creates new SCEVs:

  `llvm::ScalarEvolution::getConstant(llvm::ConstantInt *)`

  `llvm::ScalarEvolution::getAddExpr(llvm::SCEV *, llvm::SCEV *)`

  ...
- gets important SCEVs:

  `llvm::ScalarEvolution::getBackedgeTakenCount(llvm::Loop *)`

  `llvm::ScalarEvolution::getPointerBase(llvm::SCEV *)`

  ...

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 memory operation scheduling

problem often fails

Different algorithms for alias analysis:

- common interface – `llvm::AliasAnalysis` – for all algorithms
- by default, basic alias analyzer – **`basicaa`** – is used

Requiring Alias Analysis

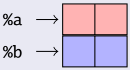
```
AU.addRequiredTransitive<llvm::AliasAnalysis>();
```

Alias Analysis
Memory Representation

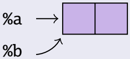
Source

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

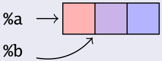
Distinct Locations



Same Location



Overlapping Locations



Basic building block is `llvm::AliasAnalysis::Location`:

- address: e.g. `%a`
- size: e.g. 2 bytes

Alias Analyzer
Basic Interface

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

- `llvm::AliasAnalyzer::NoAlias`: X and Y are different memory locations
- `llvm::AliasAnalyzer::MustAlias`: X and Y are equal – i.e. they points to the same address
- `llvm::AliasAnalyzer::PartialAlias`: X and Y partially overlap – i.e. they points to different addresses, but the pointed memory areas partially overlap
- `llvm::AliasAnalyzer::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::AliasAnalyzer::alias(X, Y)`

Alias Analyzer
Mid-level Interface

Basic alias analyzer interface is low-level – we would like expressing queries about a single pointer X:

- how referenced memory location is accessed?
- which other instructions reference the same location?

What we need is a set, to classify memory locations:

- construct a `llvm::AliasSetTracker` starting from a `llvm::AliasAnalyer *`
- it builds `llvm::AliasSetS`

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

- contains all locations aliasing with X

Alias Analyzer

Alias Set Memory Accesses

Each alias set references the memory:

- `llvm::AliasSet::NoModRef`: no memory reference – i.e. the set is empty
- `llvm::AliasSet::Mod`: memory accessed in write-mode – e.g. a **`store`** is inside the set
- `llvm::AliasSet::Ref`: memory accessed in read-mode – e.g. a **`load`** is inside the set
- `llvm::AliasSet::ModRef`: memory accessed in read-write mode – e.g. a **`load`** and a **`store`** inside the set

Alias Analyzer
Mid-level Interface

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

- `llvm::Value *`: location address
- `uint64_t`: location size
- `llvm::MDNode *`: used for type-based alias analysis [6]
- **`bool`** `*`: whether a new `llvm::AliasSet` has been created to hold the location – location does not alias up to now

Having the `llvm::AliasSet`:

- STL container-like interface: `size()`, `begin()`, `end()`, ...
- check reference type: `llvm::AliasSet::isRef()`, ...
- check aliasing type: `llvm::AliasSet::isMustAlias()`, ...

[6] set to `NULL`

Memory Dependence Analysis

Alias Analyzer High-level Interface

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

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

Reads:

- **`store`**s writing memory locations aliases with the one references by `%foo`

Writes:

- **`load`**s reading memory locations aliased with the one referenced by `%foo`

Memory Dependence Analysis

APIs

Let `%foo` be a `llvm::Instruction` accessing memory:

- call `llvm::MemoryDependenceAnalysis::getDependency(...)`
- you get a `llvm::MemDepResult`

Dependencies are classified:

- `llvm::MemDepResult::isClobber()`: an instruction clobbering – i.e. potentially modifying – location referenced by `%foo` has been found
- `llvm::MemDepResult::isDef()`: an instruction defining – e.g. writing – the exact location referenced by `%foo` has been found
- `llvm::MemDepResult::isNonLocal()`: no dependency found on `%foo` basic block
- `llvm::MemDepResult::isNonFuncLocal()`: no dependency found on `%foo` function

Contents

Conclusions

Inside LLVM there a lot of passes:

normalization put program into a canonical form

analysis get info about program

Please remember that

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

Contents

Bibliography I

LLVM Community.
Autovectorization in LLVM.
http://llvm.org/docs/Vectorizers.html.

LLVM Community.
LLVM Coding Standards.
http://llvm.org/docs/CodingStandards.html.

LLVM Community.
LLVM Passes.
http://llvm.org/docs/Passes.html.

LLVM Community.
LLVM Programmer's Manual.
http://llvm.org/docs/ProgrammersManual.html.

Bibliography II

- Ettore Speziale.
  Compiler Optimization and Transformation Passes.
  https://github.com/speziale-ettore/COTPasses.
- Chris Lattner and Vikram Adve.
  LLVM Language Reference Manual.
  http://llvm.org/docs/LangRef.html.
- Chris Lattner and Jim Laskey.
  Writing an LLVM Pass.
  http://llvm.org/docs/WritingAnLLVMPass.html.
- Michele Scandale (forked from Ettore Speziale).
  Compiler Optimization and Transformation Passes.
  https://github.com/michele-scandale/COTPasses.