UNIVERSITY OF CALIFORNIA, SAN DIEGO
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE 231 Project Report

Zhenchao Gan
Antonella Wilby
Tao Li
{zhgan, awilby, taoli} @eng.ucsd.edu

December 10, 2015

1 OVERVIEW

In this project we implemented an intra-procedural dataflow analysis and optimization framework in LLVM. Our framework first builds a control flow graph using LLVM's intermediate representation (IR) code. The framework then runs the worklist algorithm on the IR code, applying flow functions until the analysis reaches a fixed point. This framework takes a lattice bottom node and corresponding flow function to start with, and iterates repeatedly until the entire dataflow analysis is built.

We implemented four analyses using our framework: (1) constant propagation, (2) available expressions analysis, (3) range analysis and (4) intra-procedural pointer analysis. The first three analyses are based on the output of LLVM's mem2reg pass. The mem2reg pass promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses. This pass is very useful such that we will not be disturbed by complicated memory problem. However, this pass has helped us finish pointer analysis, so before applying intra-procedure pointer analysis we have to close this pass.

Using these analyses, we implemented a series of optimizations. We used constant propagation to implement constant folding and branch folding optimizations. We used the available expression analysis to apply common subexpression elimination and dead code elimination. Because these analysis are based on mem2reg pass, our own intra-procedure analysis is just to present information we gathered.

2 ANALYSIS FRAMEWORK

In this section, we present our data analysis framework interface design and introduce important APIs that are used in the worklist algorithm.

2.1 INTERFACE DESIGN

We will discuss two base classes: `LatticeNode`, which represents a node in a lattice, and `FlowFunction`, which is used to define flow functions for each separate analysis.

`LatticeNode.h` is the abstract base class of the four concrete lattice nodes. The common attributes of LatticeNode include:

1. `istop`: represents whether this lattice node is in top.
2. `isbottom`: represents whether this lattice node is in bottom.
3. `kind`: represents the kind of this lattice node (e.g. Constant Prop lattice, Available Expressions lattice, etc.).

Additionally, it provides the following common APIs:

1. `join`: a function that provides a way to join two lattice nodes.
2. `PrintInfo`: a function that dumps information about this lattice node.

We don't need `meet` method, because the lattice is built from bottom to top, only upward. It is impossible to go downwards.

`FlowFunction.h` is the base class of the four concrete flow functions. We overrided the `()` operator, making it much more like a function. `()` operator defines what we will get after going through an instruction when given input.

We implement these interfaces for each of our four analyses. Each analysis is implemented in a subclass which inherits the base class in order to implement the interface, as follows:

1. Constant Propagation is implemented in `CPLatticeNode` and `CPFlowFunction`.
2. Available Expressions analysis is implemented in `AELatticeNode` and `AEFlowFunction`.
3. Range Analysis is implemented in `RALatticeNode` and `RAFlowFunction`.
4. Intra-procedural Pointer Analysis is implemented in `PALatticeNode` and `PAFlowFunction`.

All of these analyses are implemented as subclasses which inherit the base classes of `LatticeNode` and `FlowFunction` in order to implement the interface.

2.2 Worklist

Our worklist algorithm takes three parameters, a procedure `F`, a starting lattice node initialized at bottom and a flow function. Then we will run a forward optimistic iterative dataflow analysis that computes a piece of information at the beginning of each instruction.

The control flow graph in LLVM is based on basic blocks and in LLVM there is no explicit representation of edges. Because of this, the best way to store information while doing your fixed point computation is to store one piece of information for the beginning of each basic block. To represent the edges, we use pre_block–post_block pair as key and lattice node as value to store information. It is easy with LLVM to build the pre_block and post_block relationship. Then we apply block-level flow function on the control flow graph.

When we reached a fixed point, we get the information at the beginning of each basic block. Next we should propagate the information to each instruction. So, in each instruction within a block, we run instruction-level flow function and finally get the information at the beginning of each instruction.

2.3 Optimization Passes

For each analysis, we create a corresponding pass to make use of the information produced by the analysis. The optimizations are as follows:

1. Constant Propagation is used for the Constant Folding and Branch Folding optimizations, implemented in `CFpass.cpp` and `BFpass.cpp`, respectively.
2. Available Expressions Analysis: used for Common Sub-expression Elimination optimization, implemented in `CSEpass.cpp`.
3. Range Analysis: used for a bug-finder to warn a programmer if an array access can't be shown within bounds, implemented in `RApass.cpp`.
4. Pointer Analysis: combines all other analyses for an improved optimization, implemented in `PApass.cpp`.

3 CONSTANT PROPAGATION

In this section, we present our Constant Propagation analysis, and the two optimizations implemented using this analysis: Constant Folding and Branch Folding.

3.1 Assumptions

We run this analysis on the output of `mem2reg`.

3.2 Lattice and Flow Function Definition

We define the lattice for constant propagation analysis as follows:

$$(D, \top, \bot, \sqcup, \sqcap, \sqsubseteq) = (2^{\{x \to N | x \in Vars \land N \in \mathbb{Z}\}}, \emptyset, \{x \to N | x \in Vars \land N \in \mathbb{Z}\}, \cap, \cup, \subseteq) \tag{3.1}$$

The domain is a mapping from variables to constant integer values. $\top$ is the most conservative lattice element, where every variable is non-constant. $\bot$ is the least conservative, where every variable can be a constant. For example, $\{a \to 8\}$ implies a is a constant with value 8.

The flow functions for assignment, binary operators, and PHI are defined as follows:

$$F_{X:=N}(in) = in - \{X \to *\} \cup \{X \to N\} \tag{3.2}$$

$$\begin{aligned} F_{X:=YopZ}(in) = in - \{X \to *\} \cup \{X \to N | (Y \to N_1) \in in \land \\ (Z \to N_2) \in in \land \\ N = N_1 op N_2\} \end{aligned} \tag{3.3}$$

$$\begin{aligned} F_{X:=\Phi(Y_1,Y_2)}(in_1, in_2) = in_1 \cap in_2 \cup \\ \{x \to y | x \to y \in in_1 \land \\ x \to y \in in_2\} \end{aligned} \tag{3.4}$$

3.3 Implementation

In our implementation we represent lattice nodes with a map from `llvm::Value*` to `llvm::Constant*`. For simplicity of implementation, we only handle integer constants, but the implementation can be extended to handle other data types.

To check if

3.4 Constant Folding Optimization

We use our constant propagation analysis to implement a constant folding optimization pass. The constant folding pass finds constant expressions, like $1+2$, and folds them to eliminate the add instruction. For example:

```
L1 : %add = add nsw i32 3, 5
L2 : %add1 = add nsw i32 2, 4
L3 : ret i32 %add1
```

In the above example, we can optimize out the two `add` instructions and replace them with a `ret` instruction. Then, the reference to the second add instruction can be deleted and the last line will just return the constant computed in that add, as follows:

```
L1 : ret i32 8
L2 : ret i32 3
```

3.5 Branch Folding Optimization

We use our constant propagation analysis to implement a branch folding optimization pass. In this optimization, if the conditional expression causing a branch can be evaluated at compile time (i.e., the value of the comparison is known to be true or false). We can use that result of constant propagation to eliminate branches that are never taken.

3.6 Benchmark

We provide two benchmarks to demonstrate optimizations using Constant Propagation: `CPbasic` and `CPbranch1`.

3.7 Discussion

4 AVALIABLE EXPRESSIONS ANALYSIS

In this section, we present available expressions analysis, and the Common Sub-Expression Elimination optimization that uses the analysis.

4.1 Assumptions

We run this analysis on the basis of mem2reg pass, SSA form makes this task much simpler. It is a must analysis.

4.2 Lattice and Flow Function Definition

$$(D, \top, \bot, \sqcup, \sqcap, \sqsubseteq) = (2^{x \to y | x \in Vars \wedge y \in Exprs}, \emptyset, x \to y | x \in Vars \wedge y \in Exprs, \cap, \cup, \subseteq) \tag{4.1}$$

In SSA, we don't care if we will overwrite an existing variable. Each assignment will produce a new variable in SSA.

$$F_{X:=YopZ}(in) = in \cup \{X \to YopZ\} \tag{4.2}$$

In LLVM SSA, there can't be three branches in a phi node.

$$F_{X:=\Phi(Y_1,Y_2)}(in_1, in_2) = in_1 \cap in_2 \cup \{x \to y | x \to y \in in_1 \wedge x \to y \in in_2\} \tag{4.3}$$

We have defined our lattice and flow functions above.

4.3 Implementation

We use a map to represent information of a lattice node. The key of map is `llvm::Value*`, and the value of map is `llvm::Instruction*`. We map a variable to the instruction where it is first defined.

To check if an instruction is already defined is very simple in LLVM. For example, if we have following codes :

```
L1 : %add = add nsw i32 1, 2
L2 : %add1 = add nsw i32 1, 2
```

Then by call API *isIdenticalToWhenDefined*, we can check L1 and L2 are the same. As a result, we map %add and %add2 to the same instruction.

When join, we perform intersection on left side input and right side input.

4.4 Common Sub-Expression Elimination Optimization

In this optimization, our pass searches the LLVM IR for instances of identical expressions, where all the expressions evaluate to the same value, and uses the available expressions analysis to determine whether the expressions should be replaced.

Let's review the above code. Generally speaking, we want to transform out code like this:

```
L1 : %add = add nsw i32 1, 2
L2 : %add1 = %add
```

However, when we apply `mem2reg` pass, there is no single assignment instruction and we can't transform code like this. So I apply a general optimization using our available expressions analysis. Looking at the following code:

```
L1 : %add = add nsw i32 1, 2
L2 : %add1 = add nsw i32 1, 2
L3 : ret i32 %add1
```

We know that %add1 is pointing to L1, so we use %add to replace %add1 in ret instruction, then L2 is dead code, we just simply remove it. By this way, we transform original code like this:

```
L1 : %add = add nsw i32 1, 2
L3 : ret i32 %add
```

It is a useful optimization.

4.5 Benchmark

We provide 4 benchmarks for available expression analysis, including CSEbasictest, CSEbranch, CSE-branch2, and CSEphi.

Here I show an interesting test case named CSEbranch2 :

```
int f1 = 1;
int f2 = 2;
int a = f1 + f2;
if(a > 0) {
    int b = f1 + f2;
} else {
    int c = f2 + f1;
}
return 1;
```

In this case, c = f2 + f1 is the common expression of a = f1 + f2. But the API isIdenticalToWhenDefined can't recognize this. Because this method checks operands one by one, so it return false. But it is not the case because they are definitely same. So When dealing with add instruction, we add one more check : we swap the operands and call isIdenticalToWhenDefined one more time to ensure we can handle this case.

4.6 Discussion

Consider the following code

```
entry:
    %cmp = icmp sgt i32 3, 0
    br i1 %cmp, label %if.then, label %if.else

if.then:                                         ; preds = %entry
    %add = add nsw i32 1, 2
    br label %if.end

if.else:                                         ; preds = %entry
    %add1 = add nsw i32 1, 2
    br label %if.end

if.end:                                          ; preds = %if.else, %if.then
    %add2 = add nsw i32 1, 2
    ret i32 %add2
```

For %add2, we know add new i32 1, 2 is an available expression, and we could transform code like this:

```
if.end:
    %add2 = phi i32 [ %add, %if.then ], [ %add1, %if.else ]
```

But I didn't apply this transformation. If you have many branches, then this method will create many basic blocks and phi nodes, which will bring unnecessary transformations. Because calculating 1+2 is so cheap, we didn't do optimizations for it.

5 RANGE ANALYSIS

In this section, we discuss range analysis. We use mem2reg pass for range analysis.

5.1 Lattice and Flow Function

We define complete lattice:

$$Z = \mathbb{Z} \cup \{-\infty, +\infty\} \tag{5.1}$$

for arithmetic operations. From Z, we define product lattice

$$Z^2 = \{\emptyset\} \cup \{[z_1, z_2] | z_1, z_2 \in Z, z_1 \leq z_2, -\infty < z_2\} \tag{5.2}$$

$\sqsubseteq$ is ordered by subset relation.

Range intersection $\sqcap$:

$$[a_1, a_2] \sqcap [b_1, b_2] = \begin{cases} [max(a_1, b_1), min(a_2, b_2)], b_1 \in [a_1, a_2] \ or \ a_1 \in [b_1, b_2] \\ \emptyset, \ else \end{cases} \tag{5.3}$$

Range union $\sqcup$:

$$[a_1, a_2] \sqcup [b_1, b_2] = [min(a_1, b_1), max(a_2, b_2)] \tag{5.4}$$

V is the set of constraint variables.

$$I : V -> Z^2 \tag{5.5}$$

mapps from constraint variables to intervals in Z^2. Our lattices and flow functions are given above.

5.2 Implementation

5.3 Benchmark

We provide three brenchmarks for range analysis, including RAbasic, RAbranch1 and RAbranch2. Source C++ code (RAbasic) and LLVM IR are given below:

```
// basic test for range analysis
#include <stdio.h>

int RA() {
    int i = 0;
    while (++i < 100) {
        int j = i << 1;
        int k = 0;
        // while (++k < j) {}
    }
    return i;
}

int main()
{
    printf("%d\n", RA());
    return 0;
}
```

5.4 Discussion

6 INTRA-PROCEDURE POINTER ANALYSIS

In this section, we present intra-procedure analysis. We worked on a must-point-to analysis.

6.1 Assumptions

If we run mem2reg pass, we can't apply our own intra-procedure analysis because this pass has already done this task. So before working on it, we closed this pass. If X and Y are variables and the memory of X points to the memory of Y, then we build a mapping from X to Y. If Y is a constant, we build a map from X to constant C, meaning that X is a constant.

6.2 Lattice and Flow Function

$$(D, \top, \bot, \sqcup, \sqcap, \sqsubseteq) = (2^{x \to y | x \in Vars \wedge y \in Vars}, \emptyset, x \to y | x \in Vars \wedge y \in Vars, \cap, \cup, \subseteq) \tag{6.1}$$

When we analyze pointers, there are two important instructions In LLVM, load and store. Store is a memory store and load is a memory load as well.

$$\begin{aligned} F_{store(X,Y)}(in) &= in - \{Y \to *\} \cup \{Y \to X\} \\ F_{X:=load(Y)}(in) &= in - \{X \to *\} \cup \{X \to \{whatY points - to\}\} \end{aligned} \tag{6.2}$$

$$F_{other}(in) = in \tag{6.3}$$

$$F_{merge}(in_1, in_2) = in_1 \cap in_2\} \tag{6.4}$$

6.3 Implementation

We use a map to represent information of a lattice node. the key of map is Value*, and the value of map is a Value* as well. We map a variable to the instruction where it is first defined.

For instruction like this,

```
store i32 1, i32* %a, align 4
```

We map %a to the constant 1 where 1 is stored. Although %a is just a variable, in LLVM IR it is also used as a pointer.

```
store i32* %a, i32** %c, align 8
```

For this instruction, we simply build $\%c \to \%a$ mapping.

```
store i32** %c, i32*** %d, align 8
%1 = load i32*** %d, align 8
```

For these instruction, we first build $\%d \to \%c$ mapping, then we encounter a load instruction. In LLVM, if we meet a load instruction, LLVM will create a temporary register. In this example, we know that $\%1 \to \%c$.

Combined with these examples, we know that temporary registers make things complicated. Because they occur just once and never appeared. So we must eliminate temporary registers. If $\%a \to \%1$ and $\%1 \to \%b$, then we just produce $\%a \to \%b$ and eliminate the above two mappings. This is how our algorithm works.

6.4 Benchmark

We provide 2 benchmarks for available expression analysis, including PAbasictest and PAbranch. Let's look at PAbasictest benchmark. Following is the C++ source code and LLVM IR.

```
int a = 1;
int b = 2;
int o = b;
int *c = &a;
int **d = &c;
int *f = &o;
f = *d;
return 0;
```

```
L1:  store i32 0, i32* %retval
L2:  store i32 1, i32* %a, align 4
L3:  store i32 2, i32* %b, align 4
L4:  %0 = load i32* %b, align 4
L5:  store i32 %0, i32* %o, align 4
L6:  store i32* %a, i32** %c, align 8
L7:  store i32** %c, i32*** %d, align 8
L8:  store i32* %o, i32** %f, align 8
L9:  %1 = load i32*** %d, align 8
L10: %2 = load i32** %1, align 8
L11: store i32* %2, i32** %f, align 8
L12: ret i32 0
```

When processing L5, we find $\%o \rightarrow \%0$ and $\%0 \rightarrow 2(address)$, so we will produce $\%o \rightarrow 2(address)$.

After L9, we have $\%1 \rightarrow \%c$.

When processing L10, we load from register, so we map %2 to the content where %1 points to, that is %a. So we have $\%2 \rightarrow \%a$.

When processing L11, we build $\%f \rightarrow \%2$. At the same time, we find $\%2 \rightarrow \%a$, so we will produce $\%f \rightarrow \%a$.

6.5 Discussion

Pointer analysis is complicated and registers tend to be easier to reason about than memory. So the mem2reg pass is very useful. Without mem2reg pass, we have many alloca, load and store instructions and make other analysis very difficult. With this pass, the above code can be transform to just one instruction.

```
L1:  ret i32 0
```

mem2reg pass converts non-SSA form of LLVM IR into SSA form, raising loads and stores to stack-allocated values to "registers" (SSA values). Many of LLVM optimization passes operate on the code in SSA form and thus most probably will be no-op seeing IR in non-SSA form.

7 CONCLUSION

There are many tricks when using LLVM and we managed to conquer many difficulties when dealing with problem with LLVM such as Instruction visiting. With our interface, we create several sub classes and they perform similarly from the perspective of our design.

One thing that is frustrating is after running mem2reg pass, it is hard to do some optimizations because mem2reg pass has done a lot for us like I mentioned above.

Anyway, LLVM IR is very useful and convenient. Qualcomm compiler team is also choosing LLVM as their developing tool. That's the charm of LLVM.

CONTRIBUTIONS : Zhenchao Gan works on framework, CSE and intra-procedural pointer analysis. Antonella Wilby works on CP. TaoLi works on RA.