Writing real Pointer Analysis algorithm for LLVM. Part 1: Introduction or a first date with program analysis universe

Lenar Safin Follow

Oct 4, 2017 · 10 min read

This post introduces a short series on my pointer analysis posts. Pointer analysis algorithms are used to identify, with a given accuracy, those memory areas which variables or expressions may point to. Without information about pointers, it is almost

impossible to analyze pointer-intensive programs (i.e. those written in any modern language, such as C, C++, C#, Java, Python, etc.). That is why, both a simple optimizing compiler and powerful static code analyzer use pointer analysis to obtain accurate results.

In this series of posts, we'll focus on how to write an efficient interprocedural pointer analysis algorithm, then discuss basic approaches used today and, of course, write our own cool pointer analysis algorithm for LLVM.

. . .

Program optimization and analysis algorithms

Imagine for a while that you are writing a compiler for your favorite programming language. You have already written lexical and syntax analyzers, built a translation module syntax tree, and finally translated an original program to some intermediate representation (for example, JVM bytecode or LLVM bitcode). Well, what's next? You may interpret this IR on a virtual machine or further translate it into machine code. Alternatively, you may first try to optimize the IR and then proceed with that boring translation, right? After all, your program will work faster!

Well, what can we optimize?

Consider, for example, the following code fragment.

```
k = 2;
if (complexComputationsOMG()) {
    a = k + 2;
} else {
    a = k * 2;
}
if (1 < a) {
    callUrEx();
}
k = complexComputationsAgain();
print(a);
exit();
```

Note that variable a equals 4, irrespective of what value is returned by the `complexComputationsOMG` function, meaning that you may harmlessly exclude this function call from the program IR (assuming that all our functions are pure and, in particular, free of side effects). In addition since, at the program point, where variable a is compared to 1, it always equals 4, you may execute `callUrEx` unconditionally and exclude the unnecessary branching.

Moreover, a value assigned to variable k in line `k = complexComputationsAgain()` is never used, so there is no need to call this function! Here is the result of all our transformations:

```
callUrEx();
print(4);
exit();
```

Pretty good, isn't it? Now, we just have to make our optimizer perform such code transformations automatically. Here, the wide

variety of dataflow analysis algorithms comes into play together with a universal dataflow framework introduced by notable scientist Gary Kildall in his great paper 'A Unified Approach to Global Program Optimization' and used to analyze programs or, to be exact, so-called dataflow problems.

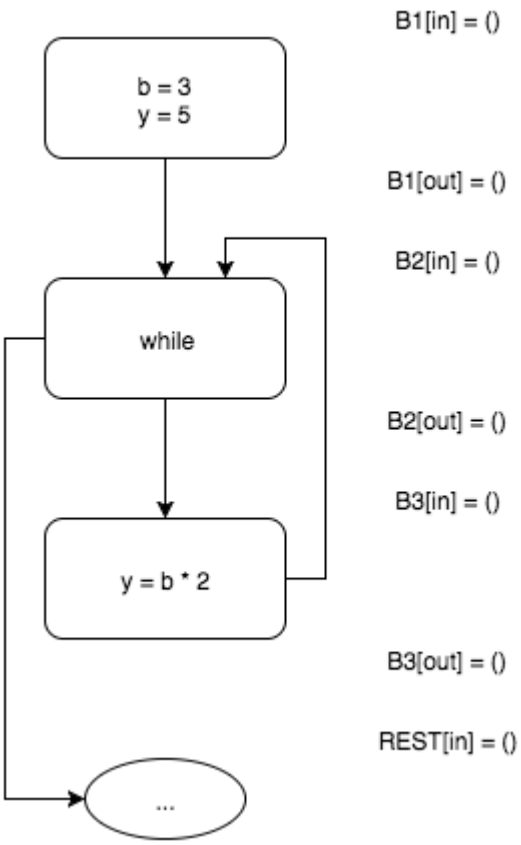
An iterative dataflow problem solution algorithm can be described in very simple terms. All we need to do is define a set of variables' properties we want to track during our analysis (for example, possible values of variables), then set interpretation functions for each basic block, and rules for propagating these properties among basic blocks (for example, intersection of sets). During the iterative algorithm execution, we calculate the values of these variables' properties in different points of the Control Flow Graph (CFG), usually at the beginning and end of each basic block. By iteratively propagating these properties, we must finally reach a fixpoint where the algorithm terminates.

They say a picture is worth a thousand words, so let's review the following example. Here is a code fragment. Let's try to identify possible values of variables in different points of the program.

```
b = 3;
y = 5;
while (...) {
    y = b * 2;
}
if (b < 4) {
    println("Here we go!");
}
```

Below you can see a solution to a classical program analysis problem, namely, an iterative constant propagation algorithm for the taken code fragment.

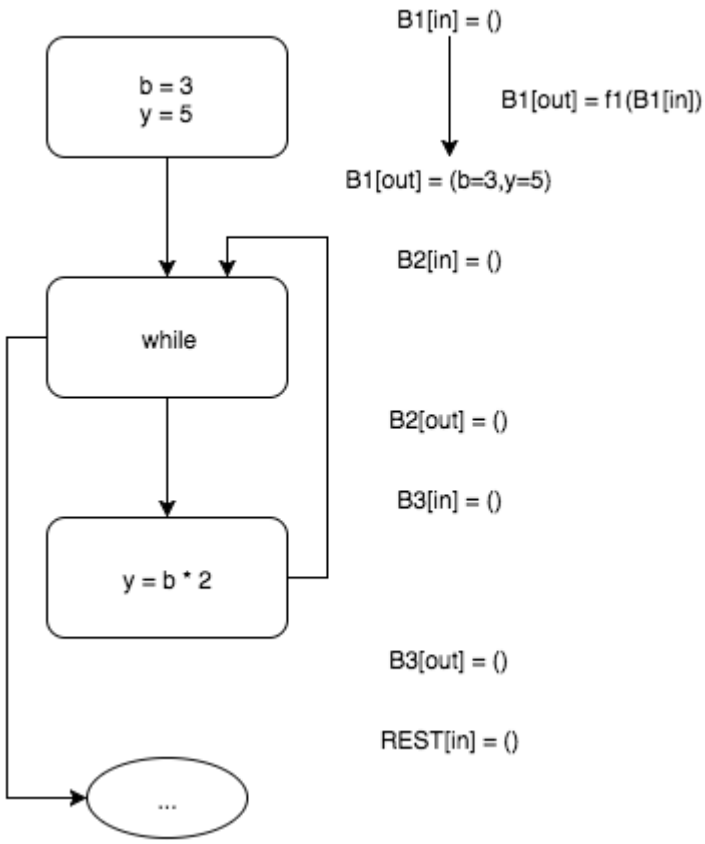
Iterative constant propagation algorithm



At the initial moment, all possible variable value sets are empty.

Interpretation of input block B1 results in b=3 and y=5 at its output.

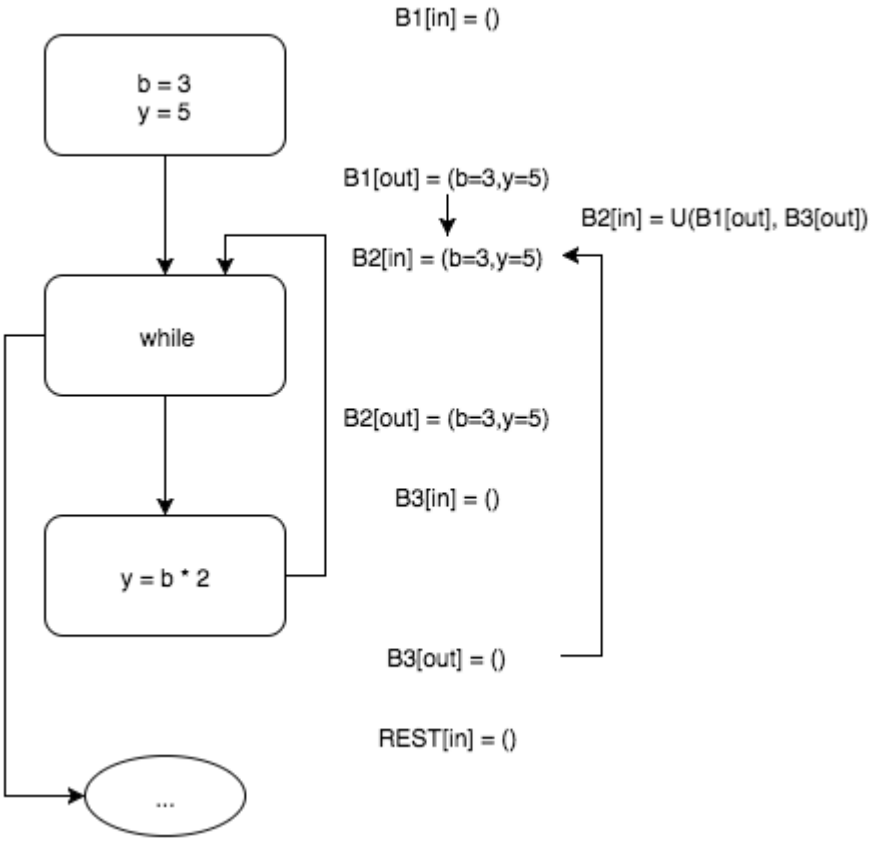
Function f1(NB: similar functions are to be defined for remaining blocks) is a block interpretation function.



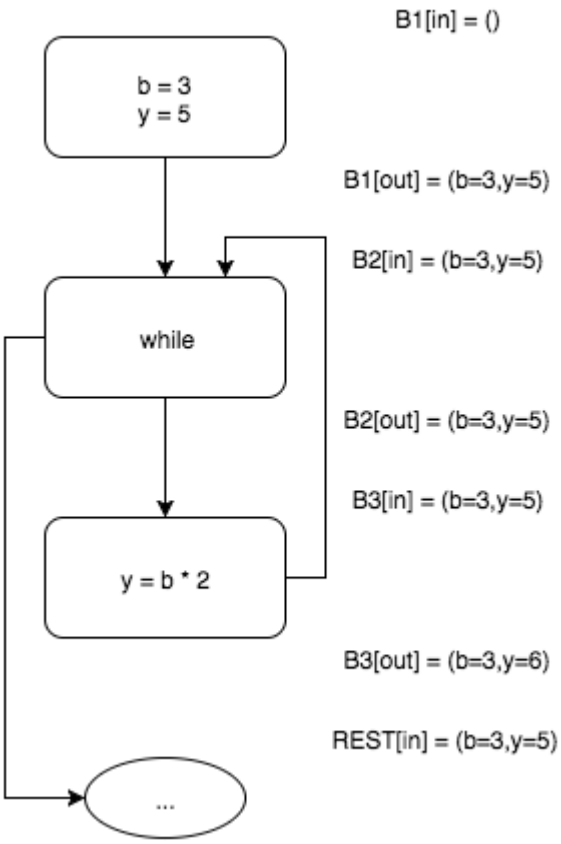
The while loop input block B2 has two predecessors: input block B1 and block B3.

Since B3 does not contain possible variable values yet, we may now assume that b=3 and y=5 are at both input and output of block B2.

Function U is a propagation rule for variable property sets (usually, infimum of a partially ordered set or, to be more precise, of a complete lattice).

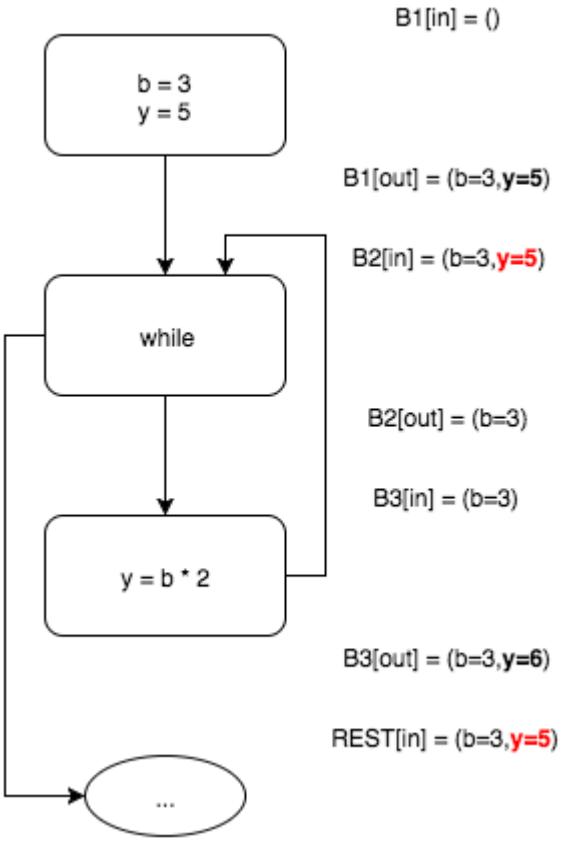


We have b=3 and y=6 at the output of basic block B3.



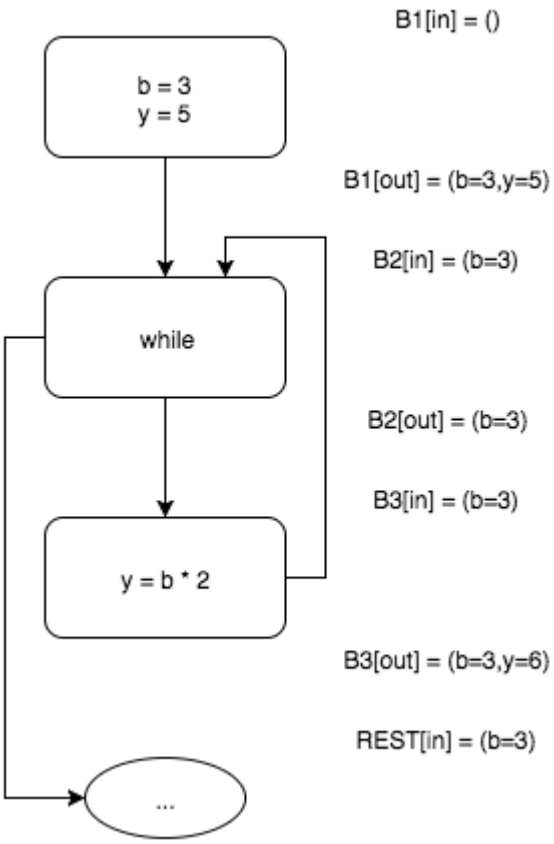
Since information about possible variable values has been changed (as compared to the initial state, i.e. 0th algorithm iteration), we can start the next iteration.

This next iteration repeats the previous one except for the step where an input set for block B2 is calculated.



As we see, this time we have to “intersect” output sets of blocks B1 and B3.

These sets have common part (b=3) which we’ll keep, and differing parts (y=5 and y=6) which we have to discard.



Since further calculations do not give us new values, algorithm execution may be deemed finished. This means that we have reached a fixpoint.

In his paper, Gary Kildall showed that such iterative algorithms will always terminate and, moreover, give the MOP solution (the meet over all paths solution) if the following conditions are met:

1. domain of monitored variable properties is a complete lattice;
2. block interpretation function possesses distributive property on the lattice;
3. infimum operator is used (i.e. meet function of a partially ordered set) to meet preceding basic blocks.

Here goes a dad's joke about the world of Big Science.

It's funny that the example used by Kildall in his paper (constant propagation) does not meet requirements that he specifies for a dataflow problem by himself: interpretation functions for constant propagation do not possess distributive property on a lattice. They are merely monotonous.

Therefore, for program optimization, we can use the full power of dataflow analysis algorithms, such as, for example, an iterative algorithm. Getting back to our first example, we used constant propagation and liveness analysis (live variable analysis) for dead code elimination.

Moreover, dataflow analysis algorithms can be used for static code analysis in the information security context. For example, when searching for SQL injection vulnerabilities, we can flag all variables that can be affected by an attacker (HTTP request parameters, etc.). If a flagged variable is used in SQL query and is not properly sanitized, then we are probably facing severe app vulnerability! All we can do then is show a potential vulnerability warning ~~and sail beyond the sunset~~ leave the user with troubleshooting recommendations.

. . .

In much wisdom is much grief: and he that increaseth knowledge increaseth sorrow

Buck-passing

To summarize, dataflow analysis algorithms are daily bread (and butter!) for any compiler hacker. So why bother with pointer analysis at all?

I'm terribly sorry to kill your mood with the following example:

```
x = 1;
*p = 3;
if (x < 3) {
   killTheCat();
}
```

It's quite obvious that being unaware of where variable `p` points to, we cannot know the value of `x < 3` expression in if operator. We only can answer this question when we know the context for this code fragment. For example, `p` may be a global variable from another module (which in C family languages may point to anywhere and anytime) or a local variable pointing somewhere to a heap. However, even knowing the context, we still need to know a set of locations (abstract memory cells), which this variable may point to. For example, if, before the above code fragment, variable p was initialized as `p = new int`, then we should exclude the conditional branch from the optimized program and call `killTheCat` method unconditionally.

`3` expression in if operator. We only can answer this question when we know the context for this code fragment. For example, `p` may

be a global variable from another module (which in C family languages may point to anywhere and anytime) or a local variable pointing somewhere to a heap. However, even knowing the context, we still need to know a set of locations (abstract memory cells), which this variable may point to. For example, if, before the above code fragment, variable p was initialized as `p = new int`, then we should exclude the conditional branch from the optimized program and call `killTheCat` method unconditionally.

So, we cannot optimize this code until we find a way to learn where variables in the analyzed program may point to!

I believe it's now clear that we cannot avoid using pointer analysis algorithms and why we need to solve this intricate (or, to be precise, algorithmically insoluble) problem. Pointer analysis is a static code analysis method used to identify values of pointers or pointer-type expressions. Depending on problems to be solved, pointer analysis can identify information either for each program point or the entire program (flow-sensitivity) or depending on function call context (context-sensitivity). In a future post, I'll talk about pointer analysis types in more detail.

Analysis results are usually represented as a mapping of a set of pointers to a set of locations, which these pointers may point to. In other words, each pointer p is mapped to a set of objects, which it may point to. Therefore, for example, in a below code fragment, the result of the analysis will be p — {a, b}, q — {p} mapping.

```
int a, b, c, *p, **q;
p = a;
q = p;
*q = b;
```

Remarkably, a mapping arrived at by means of pointer analysis must meet safety criterion, i.e. be as conservative as possible. Otherwise, our optimizations may simply spoil the semantics of an original program. Therefore, for the above code fragment, a safe result approximation is p — {a, b, c}, q — {unknown}. Value unknown is used to show that a pointer may point to all accessible objects in a program.

Consequently, in the below code fragment, dereferencing of variable p may potentially change a value of any program object!

```
extern int *offensiveFoo();
p = offensiveFoo();
*p = 77;
```

We know nothing about offensiveFoo function since it was imported from another translation module, and we therefore have to suppose that p may point absolutely anywhere!

Furthermore, we will assume that all discussed functions and global variables belong to a translation module being analyzed, unless otherwise expressly stated.

. . .

Poor man's pointer analysis

When I first faced a pointer aliasing problem, I without hesitation applied the already known iterative algorithm on a lattice (then, I had no idea that I was solving the same problem as pointer analysis algorithms do). In fact, why can't we monitor objects that may be referenced by pointers, as a set of properties of these pointers? Consider a simple example to understand how the algorithm works. Let object set propagation rules correspond to the "natural" semantics of a program (for example, if p = a, then p — {a}), and let us propagate these sets among basic blocks using a simple union of sets (for example, if q — {a, b} and q — {c} are inputs of a certain basic block, then input set for this block will be q — {a, b, c}).

So, let's go on.

```
x = a;
a = z;
if (...) {
    x = b;
} else {
    c = x;
}
```

Let's wait till the iterative algorithm terminates and view the results.

It works! Despite the algorithm's simplicity, this approach is quite viable. Moreover, until Andersen's 'Program Analysis and Specialization for the C Programming Language' thesis, this exact method (greatly improved, of course) was used to solve pointer aliasing problems. Besides, we'll also discuss his thesis in the next post!

Key shortcomings of this approach are poor scalability and overly conservative results because, when real-life programs are analyzed, calls to other functions must be taken into consideration (i.e. analysis must be of interprocedural nature), often together with the call site context. On the other hand, its major advantage is that pointer information is available for each program point (i.e. it is a flow-sensitive algorithm), while algorithms offered by Andersen and his followers provide results for the entire program (i.e. these are flow-insensitive algorithms).

. . .

In conclusion

I'm now completing the first of my posts on pointer analysis algorithms. In the next one, we'll write a simple and efficient interprocedural pointer analysis algorithm for LLVM.

> *Lenar Safin is lead software engineer in SmartDec.*
>
> *This article was first published on https://appscreener.org/blog/*

Llvm Static Code Analysis Binary C Tech Blog