Haoyi's Programming Blog (..)

Twitter
(https://twitter.com/li_haoyi)

Resume
(https://lihaoyi.github.io/Resume/)

Github
(https://github.com/lihaoyi

Learn Scala
(https://www.handsonscala.com/)

Subscribe
(http://eepurl.com/c3A5Tz)

Talks
(../post/TalksIveGiven.html

How an Optimizing Compiler Works

Posted 2019-11-09 (https://github.com/lihaoyi/blog/commit/508a910612e5c739bcc0a759d5a06af2709be9f5)

← Working with Databases using Scala and Quill
(WorkingwithDatabasesusingScalaandQuill.html)

Beyond Liskov: Type Safe Equality in Scala →
(BeyondLiskovTypeSafeEqualityinScala.html)

Optimizing compilers are a mainstay of modern software: allowing a programmer to write code in a language that makes sense to them, while transforming it into a form that makes sense for the underlying hardware to run efficiently. The optimizing compiler's job is to figure out what your input program does, and create an output program that it knows will do the same thing, just faster.

This post will walk through some of the core inference techniques within an optimizing compiler: how to model a program in a way that is easy to work with, infer facts about your program, and using those facts to make your program smaller and faster.

(https://www.handsonscala.com)
About the Author: *Haoyi is a software engineer, and the author of many open-source Scala tools such as the Ammonite REPL and the Mill Build Tool. If you enjoyed the contents on this blog, you may also enjoy Haoyi's book* ***Hands-on Scala Programming*** *(https://www.handsonscala.com)*

This blog post is a companion to the following conference talk:

- How an Optimizing Compiler Works (https://www.youtube.com/watch?v=EO0u-SQnJRQ)
- Slides (OptimizingCompiler/Slides.pdf) (with speaker notes)

Program optimizers can run in any number of places: as a step within a larger compiler's process (e.g. the Scala Optimizer (https://www.lightbend.com/blog/scala-inliner-optimizer)), as a separate standalone program you run after compiler and before execution (e.g. Proguard (https://www.guardsquare.com/en/products/proguard)) or as part of the runtime that optimizes the program while it is already being executed (e.g. the JVM JIT compiler (https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/)). The constraints imposed on optimizers in each case differ, but their output goal is the same: to take an input program and convert it to an output program that does the same thing, faster.

In this post, we will first walk through a few sample optimizations on a strawman program, to get an intuition of what typical optimizers do and how to identify and perform such optimizations manually. We will then examine a few different ways in which programs can be represented, and finally look at the algorithms and techniques we can use to infer facts about these programs, and then simplify them to be both smaller and faster.

A Strawman Program

This post will use the Java programming language for all our examples. Java is a common language, compiles to a relatively simple "assembly" - Java Bytecode (https://en.wikipedia.org/wiki/Java_bytecode) - and is uniquitous in the industry. This will give us a good canvas on which we can explore optimization compilation techniques on real, runnable examples. The techniques described here apply equally to optimizing almost any other language.

To begin with, let us consider the following strawman program. This program runs a bunch of logic, logging to the standard output in the process, and finally returning a computed result. The program itself isn't meaningful, but will serve as an illustration of what kind of things you can do to optimize it while preserving the existing behavior:

```
static int main(int n){
  int count = 0, total = 0, multiplied = 0;
  Logger logger = new PrintLogger();
  while(count < n){
    count += 1;
    multiplied *= count;
    if (multiplied < 100) logger.log(count);
    total += ackermann(2, 2);
    total += ackermann(multiplied, n);
    int d1 = ackermann(n, 1);
    total += d1 * multiplied;
    int d2 = ackermann(n, count);
    if (count % 2 == 0) total += d2;
  }
  return total;
}

// https://en.wikipedia.org/wiki/Ackermann_function (https://en.wikipedia.org/wiki/Ackermann_function)
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}

interface Logger{
  public Logger log(Object a);
}
static class PrintLogger implements Logger{
  public PrintLogger log(Object a){  System.out.println(a); return this; }
}
static class ErrLogger implements Logger{
  public ErrLogger log(Object a){ System.err.println(a); return this; }
}
```

For now, we will assume that this program is all there is: there are no other pieces of code calling into this, and this standalone program just enters `main` , performs its work, and returns. With that in mind, let us begin looking for optimizations!

Example Optimizations

Type Inference & Inlining

The first thing you may notice is the variable `logger` has a type that's im-precise: while labelled as a `Logger` , we can infer from the code that it is specific subclass: a `PrintLogger` :

```
-  Logger logger = new PrintLogger();
+  PrintLogger logger = new PrintLogger();
```

Now that we know `logger` is a `PrintLogger` , we know that the call to `logger.log` can only have a single implementation, so we can inline it:

```
-    if (multiplied < 100) logger.logcount);
+    if (multiplied < 100) System.out.println(count);
```

This lets us make the program smaller by removing the unnecessary `ErrLogger` class since we never use it, and also removing the various `public Logger log` methods since we've inlined it into our only callsite.

Constant Folding

Separately, we can see that while `count` and `total` change in the course of the program, `multiplied` does not: it starts off `0` , and every time it is multiplied via `multiplied = multiplied * count` it remains `0` . We can thus substitute `0` for it throughout the program:

```
 static int main(int n){
-  int count = 0, total = 0, multiplied = 0;
+  int count = 0, total = 0;
   PrintLogger logger = new PrintLogger();
   while(count < n){
     count += 1;
-     multiplied *= count;
-    if (multiplied < 100) System.out.println(count);
+    if (0 < 100) logger.log(count);
     total += ackermann(2, 2);
-    total += ackermann(multiplied, n);
+    total += ackermann(0, n);
     int d1 = ackermann(n, 1);
-     total += d1 * multiplied;
     int d2 = ackermann(n, count);
     if (count % 2 == 0) total += d2;
   }
   return total;
 }
```

As a consequence, we can see that `d1 * multiplied` is always `0`, and thus `total += d1 * multiplied` does nothing, and can be removed

```
-    total += d1 * multiplied
```

Dead Code Elimination

Now that we've constant folded `multiplied` and see that `total += d1 * multiplied` does nothing, we can follow up and remove the definition of `int d1`:

```
-     int d1 = ackermann(n, 1);
```

This no longer plays a part in the program, and since `ackermann` is a pure function, removing it does not change the output of the program in any way.

Similarly, after inlining `logger.log` earlier, `logger` itself is now un-used and can be eliminated:

```
-   PrintLogger logger = new PrintLogger();
```

Branch Elimination

Since The first conditional in our loop is now contingent on `0 < 100`. Since that is always true, we can simply remove the conditional:

```
- if (0 < 100) System.out.println(count);
+ System.out.println(count);
```

In general, any conditional branch that is always true we can inline the body of the conditional in its place, and for branches that are always false we can remove the conditional together with its body entirely.

Partial Evaluation

Let us take a look at the three remaining calls to `ackermann`:

```
    total += ackermann(2, 2);
    total += ackermann(0, n);
    int d2 = ackermann(n, count);
```

- The first has two constant arguments. We can see that the function is pure, and evaluate the function up-front to find `ackermann(2, 2)` must be equal to `7`
- The second has one constant argument `0`, and one unknown argument `n`. We can feed this into the definition of `ackermann`, and find that when `m` is `0`, the function always returns `n + 1`
- The third has two unknown arguments: `n` and `count`, and so there we leave it in place for now

Thus, given the call to `ackermann`, defined as:

```
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

We can simplify this into:

```
-    total += ackermann(2, 2);
+    total += 7
-    total += ackermann(0, n);
+    total += n + 1
     int d2 = ackermann(n, count);
```

Late Scheduling

Lastly, we can see that the definition of `d2` only gets used in the `if (count % 2 == 0)` conditional. Since the computation of `ackermann` is pure, we can thus move that call into the conditional, so that it doesn't get computed unless it actually gets used:

```
-    int d2 = ackermann(n, count);
-    if (count % 2 == 0) total += d2;
+    if (count % 2 == 0) {
+      int d2 = ackermann(n, count);
+      total += d2;
+    }
```

This lets us avoid half the calls to `ackermann(n, count)`, speeding things up by not calling it when not necessary.

In contrast, the `System.out.println` function is not pure, and thus we cannot move it in or out of conditionals without changing program semantics.

Actual Optimized Output

Putting together all these optimizations, the final output code is as follows:

```
static int main(int n){
  int count = 0, total = 0;
  while(count < n){
    count += 1;
    System.out.println(count);
    total += 7;
    total += n + 1;
    if (count % 2 == 0) {
      total += d2;
      int d2 = ackermann(n, count);
    }
  }
  return total;
}

static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

While all the optimizations we did above were done by hand, it is entirely possible to do them automatically. Below is the de-compiled output of a prototype optimizer I have written for JVM programs:

```
static int main(int var0) {
  new Demo.PrintLogger();
  int var1 = 0;

  int var3;
  for(int var2 = 0; var2 < var0; var2 = var3) {
    System.out.println(var3 = 1 + var2);
    int var10000 = var3 % 2;
    int var7 = var1 + 7 + var0 + 1;
    var1 = var10000 == 0 ? var7 + ackermann(var0, var3) : var7;
  }

  return var1;
}

static int ackermann__I__TI1__I(int var0) {
  if (var0 == 0) return 2;
  else return ackermann(var0 - 1, var0 == 0 ? 1 : ackermann__I__TI1__I(var0 - 1););
}

static int ackermann(int var0, int var1) {
  if (var0 == 0) return var1 + 1;
  else return var1 == 0 ? ackermann__I__TI1__I(var0 - 1) : ackermann(var0 - 1, ackermann(var0, var1 - 1));
}

static class PrintLogger implements Demo.Logger {}

interface Logger {}
```

The de-compiled code looks slightly different from the hand-optimized version. There are some things it didn't quite manage to optimize away (e.g. the un-used call to `new PrintLogger`), and some things it did slightly differently (e.g. split out `ackermann` and `ackermann__I__TI1__I`) but otherwise this automated program optimizer has done almost all the same optimizations that we just did by hand, using our own logic and intelligence.

This begs the question: how?

Intermediate Representations

If you try to build a program optimizer, the first question will encounter is possibly the most important: what, exactly, is "a program"?

You are no doubt used to writing and modifying programs as source code. You have definitely executed them as compiled binaries, and perhaps had to peek into the binaries at some point in order to debug issues. You may have encountered the programs in syntax tree (https://en.wikipedia.org/wiki/Abstract_syntax_tree) form, as three-address codes (https://en.wikipedia.org/wiki/Three-address_code), A-Normal Form (https://en.wikipedia.org/wiki/A-normal_form), Continuation Passing Style (https://en.wikipedia.org/wiki/Continuation-passing_style), or Single Static Assignment (https://en.wikipedia.org/wiki/Static_single_assignment_form) form.

There is a whole zoo of different ways you can represent programs. Here we will discuss a few of the most important ways you can represent "a program" within your program optimizer:

Sourcecode

```
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

Un-compiled source code is, itself, a representation of your program. Source code tends to be relatively compact, and human readable, but has two issues:

- It contains all sorts of naming and formatting details that are important to a user but meaningless to the computer.
- There are many more invalid source programs than there are valid source programs, and during optimization you need to make sure that your program goes from a valid source input to a valid source output.

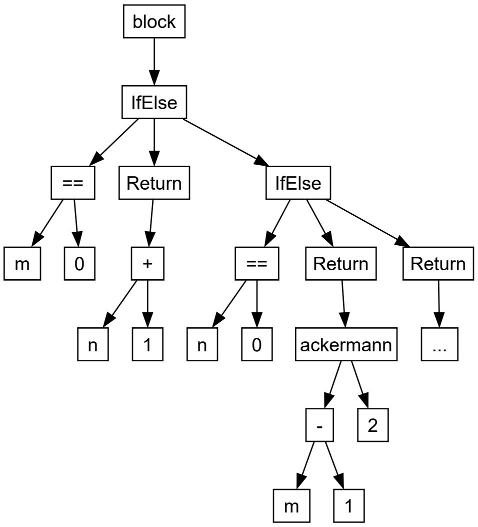
These two factors make source code a difficult form of program for a program optimizer to work with. While you *can* perform program transformations on source, e.g. using regexes (https://en.wikipedia.org/wiki/Regular_expression) to try and identify patterns and replace them, the first point above means it is difficult to reliably identify patterns in the presence of all the irrelevant detail, and the second point makes it very easy to mess up the replacement and be left with a invalid output program.

While these restrictions are acceptable for program transformers which are run with manual oversight, e.g. using Codemod (https://github.com/facebook/codemod) to refactor and transform a codebase, they make it infeasible to use source code as the primary data model of an automated program optimizer.

Abstract Syntax Trees

```
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}

IfElse(
    cond = BinOp(Ident("m"), "=", Literal(0)),
    then = Return(BinOp(Ident("n"), "+", Literal(1)),
    else = IfElse(
        cond = BinOp(Ident("n"), "=", Literal(0)),
        then = Return(Call("ackermann", BinOp(Ident("m"), "-", Literal(1)), Literal(1)),
        else = Return(
            Call(
                "ackermann",
                BinOp(Ident("m"), "-", Literal(1)),
                Call("ackermann", Ident("m"), BinOp(Ident("n"), "-", Literal(1)))
            )
        )
    )
)
```



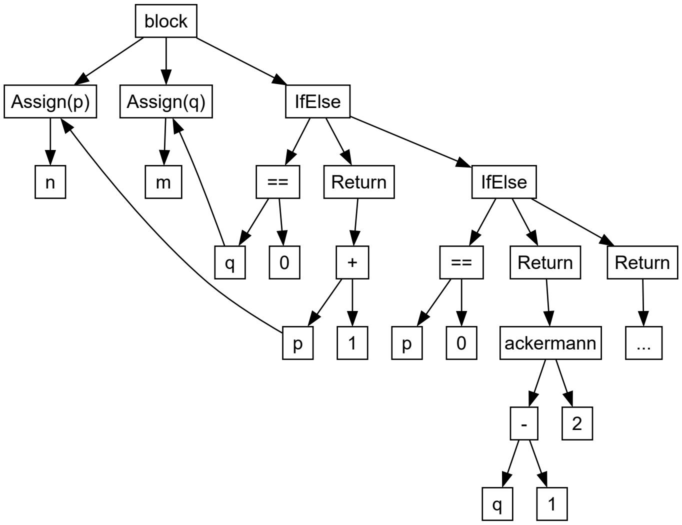
Abstract Syntax Trees (ASTs) are another common intermediate format. One level up the abstraction ladder from source code, ASTs typically discard all formatting details of the sourcecode such as whitespace or comments, but preserve things such as local variable names that get discarded in more abstract representations.

Like source code, ASTs suffer from the ability to encode extraneous information that doesn't affect the actual semantics of your program. For example, the following two programs are semantically identical, differing only by the names of their local variables, but yet have different ASTs

```
static int ackermannA(int m, int n){
  int p = n;
  int q = m;
  if (q == 0) return p + 1;
  else if (p == 0) return ackermannA(q - 1, 1);
  else return ackermannA(q - 1, ackermannA(q, p - 1));
}
Block(
    Assign("p", Ident("n")),
    Assign("q", Ident("m")),
    IfElse(
        cond = BinOp(Ident("q"), "==", Literal(0)),
        then = Return(BinOp(Ident("p"), "+", Literal(1)),
        else = IfElse(
            cond = BinOp(Ident("p"), "==", Literal(0)),
            then = Return(Call("ackermann", BinOp(Ident("q"), "-", Literal(1)), Literal(1)),
            else = Return(
                Call(
                    "ackermann",
                    BinOp(Ident("q"), "-", Literal(1)),
                    Call("ackermann", Ident("q"), BinOp(Ident("p"), "-", Literal(1)))
                )
            )
        )
    )
)
```

```
static int ackermannB(int m, int n){
  int r = n;
  int s = m;
  if (s == 0) return r + 1;
  else if (r == 0) return ackermannB(s - 1, 1);
  else return ackermannB(s - 1, ackermannB(s, r - 1));
}
Block(
    Assign("r", Ident("n")),
    Assign("s", Ident("m")),
    IfElse(
        cond = BinOp(Ident("s"), "==", Literal(0)),
        then = Return(BinOp(Ident("r"), "+", Literal(1)),
        else = IfElse(
            cond = BinOp(Ident("r"), "==", Literal(0)),
            then = Return(Call("ackermann", BinOp(Ident("s"), "-", Literal(1)), Literal(1)),
            else = Return(
                Call(
                    "ackermann",
                    BinOp(Ident("s"), "-", Literal(1)),
                    Call("ackermann", Ident("s"), BinOp(Ident("r"), "-", Literal(1)))
                )
            )
        )
    )
)
```

The key insight here is that while the ASTs are structured as trees, they contain some nodes that semantically do not behave as trees: `Ident("r")` and `Ident("s")` have their meaning defined not by the contents of their subtree, but by the `Assign("r", ...)` and `Assign("s", ...)` nodes seen earlier. In effect, these `Ident` s and `Assign` s have additional semantic edges between them, which are just as important as the edges in the AST's tree structure:



these edges form a generalized graph structure, including cycles if there are recursive function definitions.

Bytecode

Since we are using Java as our base language, compiled programs end up being stored as Java Bytecode in `.class` files.

Recall the `ackermann` function we saw earlier:

```
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
}
```

This compiles to bytecode that looks like this:

```
   0: iload_0
   1: ifne          8
   4: iload_1
   5: iconst_1
   6: iadd
   7: ireturn
   8: iload_1
   9: ifne          20
  12: iload_0
  13: iconst_1
  14: isub
  15: iconst_1
  16: invokestatic ackermann:(II)I
  19: ireturn
  20: iload_0
  21: iconst_1
  22: isub
  23: iload_0
  24: iload_1
  25: iconst_1
  26: isub
  27: invokestatic ackermann:(II)I
  30: invokestatic ackermann:(II)I
  33: ireturn
```

The Java Virtual Machine (JVM) that Java bytecode runs on is a mixed stack/register machine: there is an operand STACK where values can be operated on, and an array of LOCALS where values can be stored. A function begins with its N parameters in the first N local variable slots, and executes by moving things onto the stack, operating on them, and putting them back into locals, eventually calling `return` to return a value on the operand stack to the caller.

If we annotate the above bytecode to represent the values moving between the stack and the local variables table, it looks as follows:

```
BYTECODE                                LOCALS          STACK
                                        |a0|a1|         |
 0: iload_0                             |a0|a1|         |a0|
 1: ifne          8                     |a0|a1|         |
 4: iload_1                             |a0|a1|         |a1|
 5: iconst_1                            |a0|a1|         |a1| 1|
 6: iadd                                |a0|a1|         |v1|
 7: ireturn                             |a0|a1|         |
 8: iload_1                             |a0|a1|         |a1|
 9: ifne          20                    |a0|a1|         |
12: iload_0                             |a0|a1|         |a0|
13: iconst_1                            |a0|a1|         |a0| 1|
14: isub                                |a0|a1|         |v2|
15: iconst_1                            |a0|a1|         |v2| 1|
16: invokestatic ackermann:(II)I        |a0|a1|         |v3|
19: ireturn                             |a0|a1|         |
20: iload_0                             |a0|a1|         |a0|
21: iconst_1                            |a0|a1|         |a0| 1|
22: isub                                |a0|a1|         |v4|
23: iload_0                             |a0|a1|         |v4|a0|
24: iload_1                             |a0|a1|         |v4|a0|a1|
25: iconst_1                            |a0|a1|         |v4|a0|a1| 1|
26: isub                                |a0|a1|         |v4|a0|v5|
27: invokestatic ackermann:(II)I        |a0|a1|         |v4|v6|
30: invokestatic ackermann:(II)I        |a0|a1|         |v7|
33: ireturn                             |a0|a1|         |
```

Here, I have used `a0` and `a1` to represent the functions arguments, which are all stored in the LOCALS table at the start of a function. `1` represents the constants loaded via `iconst_1`, and `v1` to `v7` are the computed intermediate values. We can see that there are three `ireturn` instructions, returning `v1`, `v3`, or `v7`. This function does not define any other local variables, and so the LOCALS array only ever contains the input arguments.

If you recall the two variants `ackermannA` and `ackermannB` we saw earlier, both of them compile to the same bytecode:

```
BYTECODE                                LOCALS          STACK
                                        |a0|a1|         |
 0: iload_1                             |a0|a1|         |a1|
 1: istore_2                            |a0|a1|a1|      |
 2: iload_0                             |a0|a1|a1|      |a0|
 3: istore_3                            |a0|a1|a1|a0|   |
 4: iload_3                             |a0|a1|a1|a0|   |a0|
 5: ifne          12                    |a0|a1|a1|a0|   |
 8: iload_2                             |a0|a1|a1|a0|   |a1|
 9: iconst_1                            |a0|a1|a1|a0|   |a1| 1|
10: iadd                                |a0|a1|a1|a0|   |v1|
11: ireturn                             |a0|a1|a1|a0|   |
12: iload_2                             |a0|a1|a1|a0|   |a1|
13: ifne          24                    |a0|a1|a1|a0|   |
16: iload_3                             |a0|a1|a1|a0|   |a0|
17: iconst_1                            |a0|a1|a1|a0|   |a0| 1|
18: isub                                |a0|a1|a1|a0|   |v2|
19: iconst_1                            |a0|a1|a1|a0|   |v2| 1|
20: invokestatic ackermannA:(II)I       |a0|a1|a1|a0|   |v3|
23: ireturn                             |a0|a1|a1|a0|   |
24: iload_3                             |a0|a1|a1|a0|   |a0|
25: iconst_1                            |a0|a1|a1|a0|   |a0| 1|
26: isub                                |a0|a1|a1|a0|   |v4|
27: iload_3                             |a0|a1|a1|a0|   |v4|a0|
28: iload_2                             |a0|a1|a1|a0|   |v4|a0|a1|
29: iconst_1                            |a0|a1|a1|a0|   |v4|a0|a1| 1|
30: isub                                |a0|a1|a1|a0|   |v4|a0|v5|
31: invokestatic ackermannA:(II)I       |a0|a1|a1|a0|   |v4|v6|
34: invokestatic ackermannA:(II)I       |a0|a1|a1|a0|   |v7|
37: ireturn                             |a0|a1|a1|a0|   |
```

Here, we can see that because the source code takes the two arguments and puts them in local variables, the bytecode has corresponding instructures to load the argument values from LOCALs 0 and 1 and store them back in index 2 and 3. However, bytecode does not care about the names of your local variables: they're treated purely as indices within the LOCALS array, meaning both `ackermannA` and `ackermannB` have identical bytecode. This makes sense, given that they are semantically equivalent!

However, `ackermannA` and `ackermannB` do not compile to the same bytecode as the original `ackermann` : while bytecode abstracts away the names of your local variables, it still does not fully abstract away the loads and stores to/from locals. We still find ourselves having to deal with the details of how values are moved around the LOCALS and STACK, even though they do not affect how the program actually behaves.

Apart from the lack of load/store abstractions, bytecode has another problem: like most linear assemblies, it is very much optimized for compactness, and can be quite difficult to modify when the time comes to perform optimizations.

To understand why, let us revisit the original bytecode for `ackermann` :

```
BYTECODE                                LOCALS          STACK
                                        |a0|a1|         |
 0: iload_0                             |a0|a1|         |a0|
 1: ifne          8                     |a0|a1|         |
 4: iload_1                             |a0|a1|         |a1|
 5: iconst_1                            |a0|a1|         |a1| 1|
 6: iadd                                |a0|a1|         |v1|
 7: ireturn                             |a0|a1|         |
 8: iload_1                             |a0|a1|         |a1|
 9: ifne          20                    |a0|a1|         |
12: iload_0                             |a0|a1|         |a0|
13: iconst_1                            |a0|a1|         |a0| 1|
14: isub                                |a0|a1|         |v2|
15: iconst_1                            |a0|a1|         |v2| 1|
16: invokestatic ackermann:(II)I        |a0|a1|         |v3|
19: ireturn                             |a0|a1|         |
20: iload_0                             |a0|a1|         |a0|
21: iconst_1                            |a0|a1|         |a0| 1|
22: isub                                |a0|a1|         |v4|
23: iload_0                             |a0|a1|         |v4|a0|
24: iload_1                             |a0|a1|         |v4|a0|a1|
25: iconst_1                            |a0|a1|         |v4|a0|a1| 1|
26: isub                                |a0|a1|         |v4|a0|v5|
27: invokestatic ackermann:(II)I        |a0|a1|         |v4|v6|
30: invokestatic ackermann:(II)I        |a0|a1|         |v7|
33: ireturn                             |a0|a1|         |
```

And from here, make a strawman change: we have decided that the function call `30: invokestatic ackermann:(II)I` does not make use of its first argument, and thus we can replace it with an equivalent function call `30: invokestatic ackermann2:(I)I` that only takes 1 argument. This is a common optimization, and can allow any code that was used to compute the first argument of `30: invokestatic ackermann:(II)I` to be eliminated by Dead Code Elimination we saw earlier.

To do so, we need to not only replace instruction `30` , we need to look up the list of instructions to figure out where the first argument (`v4` on the `STACK`) gets computed and remove those as well. We end up walking backwards, from instruction `30` to `22` , and `22` to `21` and `20` . The final change would look something like this:

```
  BYTECODE                                LOCALS          STACK
                                          |a0|a1|         |
   0: iload_0                             |a0|a1|         |a0|
   1: ifne          8                     |a0|a1|         |
   4: iload_1                             |a0|a1|         |a1|
   5: iconst_1                            |a0|a1|         |a1| 1|
   6: iadd                                |a0|a1|         |v1|
   7: ireturn                             |a0|a1|         |
   8: iload_1                             |a0|a1|         |a1|
   9: ifne          20                    |a0|a1|         |
  12: iload_0                             |a0|a1|         |a0|
  13: iconst_1                            |a0|a1|         |a0| 1|
  14: isub                                |a0|a1|         |v2|
  15: iconst_1                            |a0|a1|         |v2| 1|
  16: invokestatic ackermann:(II)I        |a0|a1|         |v3|
  19: ireturn                             |a0|a1|         |
- 20: iload_0                             |a0|a1|         |
- 21: iconst_1                            |a0|a1|         |
- 22: isub                                |a0|a1|         |
  23: iload_0                             |a0|a1|         |a0|
  24: iload_1                             |a0|a1|         |a0|a1|
  25: iconst_1                            |a0|a1|         |a0|a1| 1|
  26: isub                                |a0|a1|         |a0|v5|
  27: invokestatic ackermann:(II)I        |a0|a1|         |v6|
- 30: invokestatic ackermann:(II)I        |a0|a1|         |v7|
+ 30: invokestatic ackermann2:(I)I        |a0|a1|         |v7|
  33: ireturn                             |a0|a1|         |
```

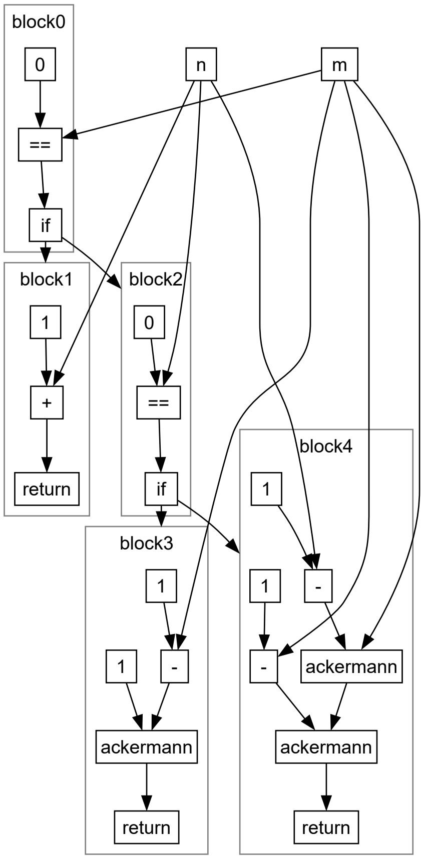
While doing such a trivial change in the simple `ackermann` function is still possible, it gets much more messy with more involved changes in larger, real-world functions. In general, every small semantic change to your program may require in a large number of changes scattered throughout the bytecode.

You may have noticed that the way we perform this change is to look at the LOCALS and STACK values we have simulated on the right: seeing `v4` arrives at instruction `30` from instruction `22`, and instruction `22` takes in `a0` and `1` which arrive from instructions `21` and `20`. These values being passed around between LOCALS and STACK for a graph: from the instruction that computed each value, to the places where it is used.

Like the `Ident` / `Assign` pairs in our ASTs, the values being moved around the LOCALS and STACK form a graph between the computation of a value, and its usages. Why not work with the graph directly?

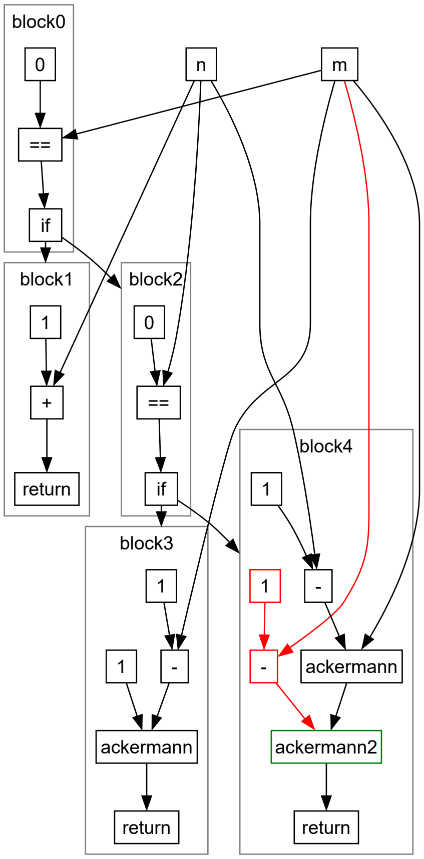
Dataflow Graphs

Dataflow Graphs are one level up the abstraction ladder from Bytecode or ASTs. If we extend our syntax tree with the `Ident` / `Assign` edges, or if we follow how the bytecode is moving values between LOCALS and STACK, we will end up forming the same graph. For the `ackermann` function, it looks like this:



Unlike ASTs or Java's Stack/Register-based bytecode, Dataflow Graphs do not have the concept of a "local variable": instead the graph contains directed edges between each value and its usages. While analyzing bytecode often involves doing abstract interpretation of the LOCALS and STACK to figure out how values are being moved around, and analyzing ASTs involves walking both the tree and managing a symbol table containing `Assign` / `Ident` edges, analyzing Dataflow Graphs is often just straightforward graph traversals: the whole idea of "moving a value" removed from the program representation.

Dataflow graphs are also somewhat easier to manipulate than linear bytecode: replacing a function call node `ackermann` with an `ackermann2` call, and discarding one of its arguments, is a matter of mutating a graph node (below in green) and removing one input edge, along with its transitive upstream edges/nodes, from the graph (below in red):



Thus, we can see how a small change to the program (replacing `ackermann(int n, int m)` with `ackermann2(int m)`) ends up being a relatively localized change in the dataflow graph.

In general, dataflow graphs are much easier to work with than linear bytecodes or ASTs: both graph analyses and graph modifications are straightforward.

This description of dataflow graphs leaves out a lot of detail: apart from the actual physical representation of the graph, there are also many different ways of modeling state, control flow, whose handling would be a bit more involved and not fit within this blog post. We have also left out the details of graph transformations, e.g. adding/removing edges, forward/backwards breadth/depth-first traversals, etc. which should be familiar to anyone who has taken an introductory algorithms course.

Lastly, we have skipped over the algorithms for converting from a linear bytecode to the dataflow graph, and then from the dataflow graph back to a linear bytecode. That is itself an interesting problem, but for now will have to be left as an exercise for the reader.

Inference

Once you have a representation of your program, the next step is inference: finding out facts about your program so you can know how to transform it without changing its behavior. Indeed, many of the optimizations we saw earlier are based on inferring facts about our program:

- Constant Folding: is the output of this expression a known constant value? Is the computation of the expression Pure?
- Type Inference + Inlining: is the type of the method call a type with a single implementation of the called method?
- Branch Elimination: is the boolean conditional expression a constant?
- Dead Code Elimination: is the result of this computation "live", i.e. it ends up contributing to the output of the program? Is the computation Pure?
- Late Scheduling: is this computation Pure, and thus re-schedulable?

The more facts we can infer about a program, and the more specific those facts can be, the more aggressively we can transform the program to optimize it without changing its semantics. We will discuss the inference of types, constants, liveness and reachability in this post, and leave analysis of Purity as an exercise to the reader.

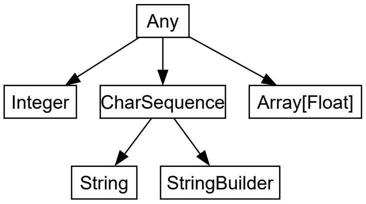
Types, constants, purity, liveness are just some of the most common things an optimizer might want to infer about your program, and the way inference works in the optimizer is a very similar topic to inference of types in a compiler's frontend typechecker.

The Inference Lattice

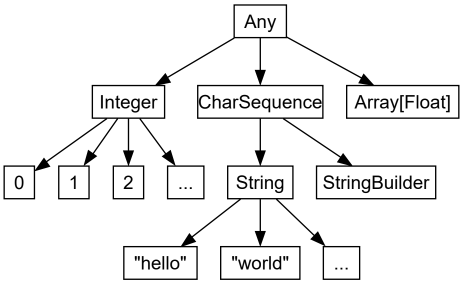
To begin with, let us consider how inferring types and constants would work. At its core, a "type" represents something we know about a value:

- Is it an `Integer` ? A `String` ? An `Array[Float]` s? A `PrintLogger` ?
- Is it a `CharSequence` , meaning it could be a `String` , but could also be something else like a `StringBuilder` ?
- Is it `Any` , meaning, we don't know what it is?

The potential types a value in your program could take form a lattice:

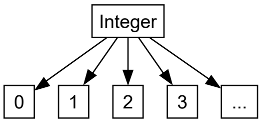


Inferring constant value is very similar to inferring types: in a way, a constant string `"hello"` is a sub-type of `String`, just like `String` is a subtype of `CharSequence`. Since there is only one string `"hello"`, these are often called "Singleton Types". Extending our lattice to include these is straightforward.

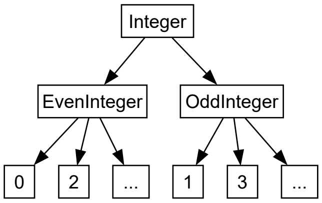


The further up the lattice the type we assign to a value, the less we know about it. The further down the lattice, the more we know. If we know a value has one of two types, e.g. it is either a `String` OR a `StringBuilder` then we can conservatively model it as the least-upper-bound type of the two: a `CharSequence`. If we have a value that is either `0` or `1` or `2`, we infer it to be an `Integer`.

Note that exactly how fine-grained you want to make your lattice is a design decision. In this case we decided the parent of `0`, `1`, `2` and `3` are all `Integer`:



But we could come up with an more fine-grained lattice, e.g. separating even and odd numbers:



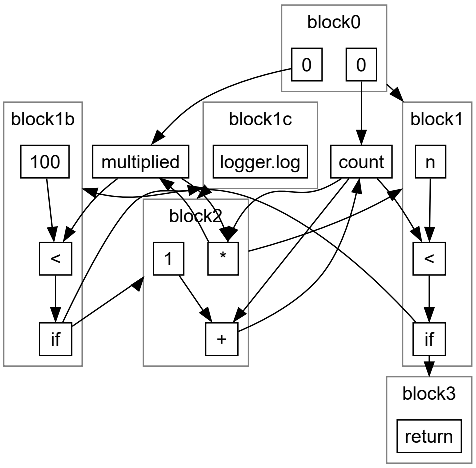
The more fine-grained your lattice, the more precise your inferences, but the longer your analysis will take to run. Exactly how fine-grained you want it to be ends up being a judgement call.

Inferring Count

Let us look at a simplified version of the `main` program we saw earlier:

```
static int main(int n){
  int count = 0, multiplied = 0;
  while(count < n){
    if (multiplied < 100) logger.log(count);
    count += 1;
    multiplied *= count;
  }
  return ...;
}
```

For simplicity have removed the various calls to `ackermann`, to only leave the code using `count`, `multiplied`, and `logger`. First, let's look at the dataflow graph:



We see that `count` isinitialized to `0` in `block0`. Next, control flow shifts to `block1`, where we check if `count < n`: if it is we go to `block3` and `return`, otherwise we go to `block2` which increments `count` by `1` and stores it back in `count`, returning control to `block1` to repeat the check. This loops until the conditional `<` returns `false`, at which point it goes to `block3` and exits.

How do we analyze this?

- Starting in `block0`, we know `count = 0`
- Going into `block1`, we do not know what `n` is (it's an input parameter, and could be any `Integer`), and thus do not know where the `if` will go. We must therefore consider both `block2` and `block3`
- Ignoring `block3` since it's trivial, we go into `block1b`, which transitions into `block2` either directly, or indirectly after logging stuff in `block1c`: in `block2`, we see that it is taking `count` incrementing it by `1`, and putting the value back into `count`
- We now know `count` could either be `0` or `1`: looking at the Lattice we defined earlier, we now infer `count` as `Integer`
- We make another round: going through `block1`, we now see both `n` and `count` are both inferred as `Integer`.
- Going into `block2` again, we see `count` is being changed to `Integer + 1 -> Integer`. Since we already know `count` is an `Integer`, we can stop the analysis.

Inferring Multiplied

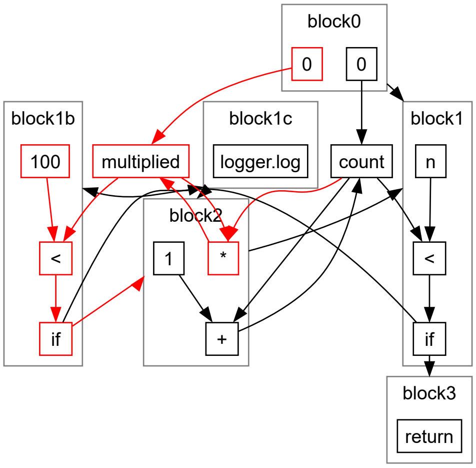
Let us consider the other dataflow graph, regarding `multiplied`, block by block:

- Starting in `block0`, we know `multiplied` is being assigned to `0`
- We transition into `block1`, which has a conditional we couldn't eliminate earlier, bringing us into `block2` and `block3` (which is trivial)
- In `block2`, we find that we are multiplying `block2` (which is `0`) with `count` (which we had earlier determined to be `Integer`). Since we know that `0 * Integer -> 0`, we can leave `multiplied` inferred as `0`
- Going through `block1` and `block2` again, `multiplied` is still inferred as `0`, and thus we can end the analysis.

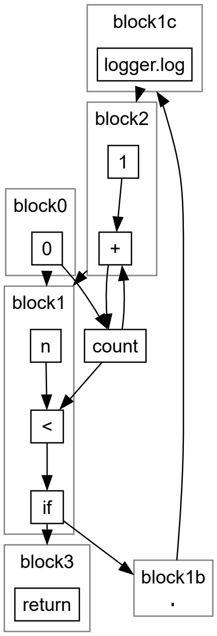
Since `multiplied` is inferred to be `0`, we now know:

- `multiplied < 100` can be inferred to be `true`
- `if (multiplied < 100) logger.log(count);` can be simplified to `logger.log(count)`.
- We can simply remove all computations that end up in `multiplied`, since we already know that the end result is always going to be `0`.

These modifications are marked in red below:



This gives us the following remaining dataflow graph:



Serializing this back to bytecode gives us the following program:

```
static int main(int n){
  int count = 0;
  while(count < n){
    logger.log(count);
    count += 1;
  }
  return ...;
}
```

Thus, we can see that by modeling the program with an appropriate data structure, a simple set of graph traversals is enough to simplify the messy, inefficient code we saw earlier to the tight, efficient loop we see here.

The inference that `multiplied -> 0` can also feed into other optimizations, e.g. the Partial Evaluation (file:///Users/lihaoyi/Dropbox/Workspace/Scaladays/presentation.md#partial-evaluation) step we saw earlier. In general, inferences allow further inferences, and optimizations allow further optimizations. Much of the difficulty in designing a program optimizer is to ensure you can leverage this sort of downstream opportunities for inference and optimization in an effective and efficient manner.

In this section, I have shown how basic inference works on a dataflow graph. we have seen:

- How we can define a Lattice of inferences to represent what we know about each value in the program
- What the dataflow graphs look like for these synthetic programs
- How we can walk the dataflow graph to make inferences about what each value in the program can be
- How we can use those inferences to perform optimizations: branch elimination or partial evaluation, where possible

In this case, we could perform the analysis for `count` first and `multiplied` separately after. This is only possible because `multiplied` depends on `count` , but `count does not depend on` multiplied`. If there is a mutual dependency, we can still do the analysis, but must analyze both variables together in one traversal of the dataflow graph.

Note that this inference is an iterative process: you keep going around loops until your inferences stop changing. In general, program dataflow graphs without cycles (whether loops or recursion) can always be analyzed in one pass, whereas programs with loops or recursion will need iteration for the inferences to converge.

While in this case we only needed to make two rounds around the `while` loop, it is possible to come up with programs that take O(height-of-lattice) iterations to analyze. Thus, while a more fine-grained lattice (e.g. the one separating `EvenInteger` s and `OddInteger` s we saw earlier) could improve the precision of analysis, it also increases the time it takes to run, and exactly where you make that trade-off depends on the use case.

A similar technique can be also used to infer properties of arbitrary recursive functions, as we will see in a moment.

Inter-procedural Function Inference

Above, we performed inference of values within a single function body, ignoring for simplicity calls to other functions or methods. However, most programs are made of many functions calling each other, so working with real programs forces you to infer properties not just about single expressions within a function, but also about the functions themselves and how they relate to each other in a call graph.

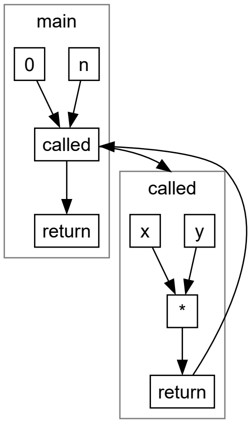
Simple Non-Recursive Functions

In most cases, handling other functions is easy. consider this tiny example program:

```
static int main(int n){
    return called(n, 0);
}

static int called(int x, int y){
    return x * y;
}
```

The datflow graph for these two functions looks something like this:

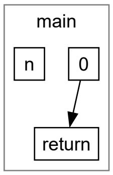


Inferring the return value of `main` proceeds as follows:

- Infer `main(n)`
    - Infer `called(n, 0)`
        - Infer `x * y` for `x = n` and `y = 0`
        - `n * 0` is `0`
    - `called(n, 0)` is `0`
- `main(n)` is `0`

Essentially, when you reach a function call during inference, you simply perform inference on the callee before returning to continue inference on the caller. This stack of inferences can go arbitrarily deep, and mirrors the call stack that will be present when your program is running.

Since we now know `called(n, 0)` is `0`, we can use that knowledge to simplify the dataflow graph:



Which can be serialized back to the following code:

```
static int main(int n){
    return 0;
}
```

This works well as long as your functions are not recursive: if function A calls B calls C calls D, then D returns its inference to C, B, D and A. However, if function A calls B and B calls A, or even function A just calls A recursively, this falls apart as the calls will never return!

A Recursive Factorial Function

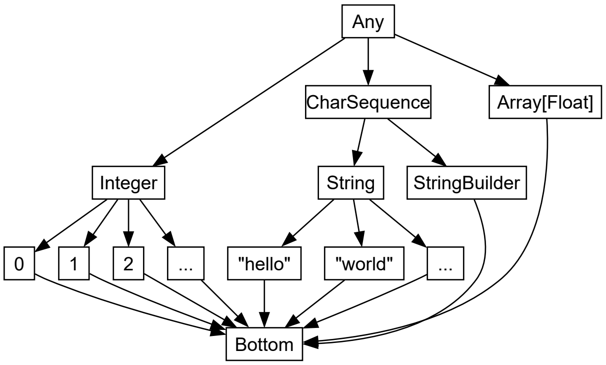
Let's consider a simple recursive function, written in pseudo-Java:

```
public static Any factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

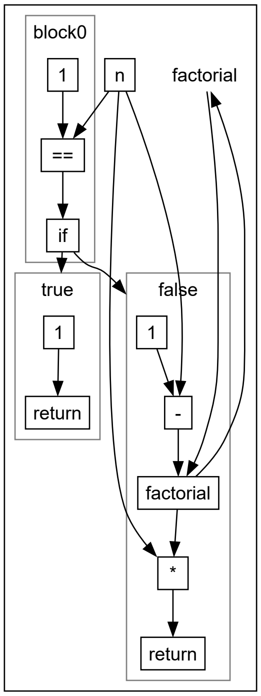
It takes and `int n`, but its return type is labeled `Any`: the declaration does not state what it returns. Visually, we can clearly see that `factorial` returns an `int` (or `Integer` in our lattice), but if we try to fully analyze `factorial`'s body before returning, we will try to analyze the recursive call to `factorial` before our own analysis of `factorial` completes, leaving us in an infinite recursion! How can our inference engine determine that on our behalf?

Inference with Bottom

The solution to this is to extend our inference lattice with the special value `Bottom`:



The purpose of `Bottom` is to say "we don't know what this is yet, but we will come back later and fill it out". We can use `Bottom` on the `factorial` dataflow graph as follows:



- Starting from `block0`, `n` is an unknown `Integer`, `1` is `1`, `n == 1` is thus unknown, so we must analyze both `true` and `false` branches
- The true branch is trivial: `return` returns `1`
- In the `false` branch, `n - 1` is an unknown `Integer` because of `n`
- `factorial` is a recursive call, so we will make it as `Bottom` for now
- `*` of `n: Integer` and `factorial: Bottom` is also `Bottom`
- `return` returns `Bottom`
- The whole factorial function returns either `1` or `Bottom`, and the least-upper-bound of those two inferences in the lattice is `1`
- We plug `1` back in as the inference of the recursive `factorial` call we labeled as `Bottom` earlier
- `Integer * 1` is now `Integer`
- `return` now returns `Integer`
- `factorial` now returns `Integer` or `1`, and the least-upper-bound of the two is `Integer`
- Re-inferring the recursive `factorial` call *again*, this time as `Integer`, we can see that the `*` expression taking `n: Integer` and `factorial: Integer` remains unchanged as `Integer`, and so inference is complete.

Thus, we can infer that `factorial` returns an `Integer`, despite it being a recursive function without a declared return type.

While these examples are a bit contrived, it serves to illustrate the way inference can proceed in the presence of function calls. Even for recursive functions, you stub out the recursive calls with `Bottom` when you first see them, and when the first analysis pass completes, you replace the stub with the first pass inference and propagate the change through the dataflow graph.

In this case, the `*` expression is analyzed three times:

- First, as `(n: Integer) * (factorial: Bottom)`
- Second, as `(n: Integer) * (factorial: 1)`
- Third, as `(n: Integer) * (factorial: Integer)`

As is the case for inferring the loop constant `multiplied` earlier, this might happen up to `O(height-of-lattice)` times before the inference converges. This approach generalizes to other styles of recursive functions, and to inferring other properties such as purity.

Inferring Liveness and Reachability

Liveness and Reachability analyses are a pair of optimizations that generally fall under the category "dead code elimination". These analyses find all the code whose values contribute to the final returned result ("Live") and all code which the the control-flow of the program can reach ("Reachable"): code that fails either test is a safe candidate for removal.

To illustrate these two analyses, let us consider another simplified version of our initial `main` function:

```
static int main(int n){
  int count = 0, total = 0, multiplied = 0;
  while(count < n){
    if (multiplied > 100) count += 1;
    count += 1;
    multiplied *= 2;
    total += 1;
  }
  return count;
}
```

Unlike our earlier versions, here we flipped the conditional around in `if (multiplied > 100)`, and have changed `multiplied *= count` to `multiplied *= 2` to simplify the program graphs without loss of generality.

Traversing Backwards for Liveness

There are two issues we can identify with the program as written above:

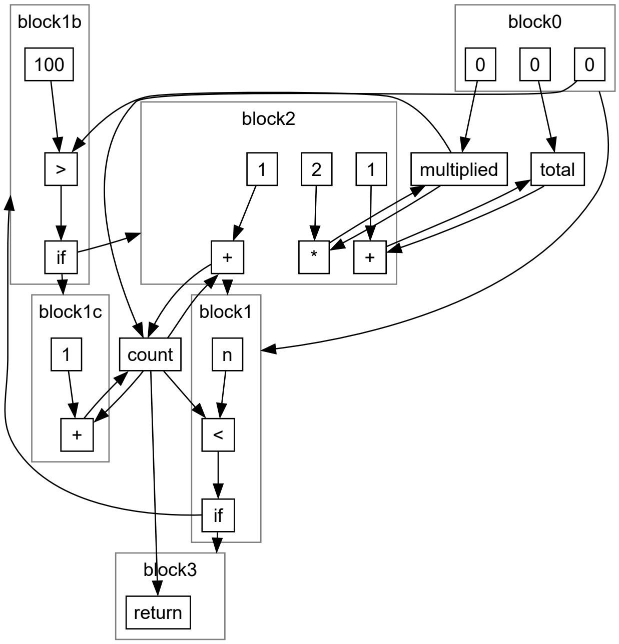
- `multipled > 100` is never true, so `count += 1` will never execute ("unreachable")
- `total` stores some computed values, but the stored value is never used ("not live")

Ideally, this should be written as:

```
static int main(int n){
  int count = 0;
  while(count < n){
    count += 1;
  }
  return count;
}
```

Let us now see how a program analyzer can perform this rewrite automatically.

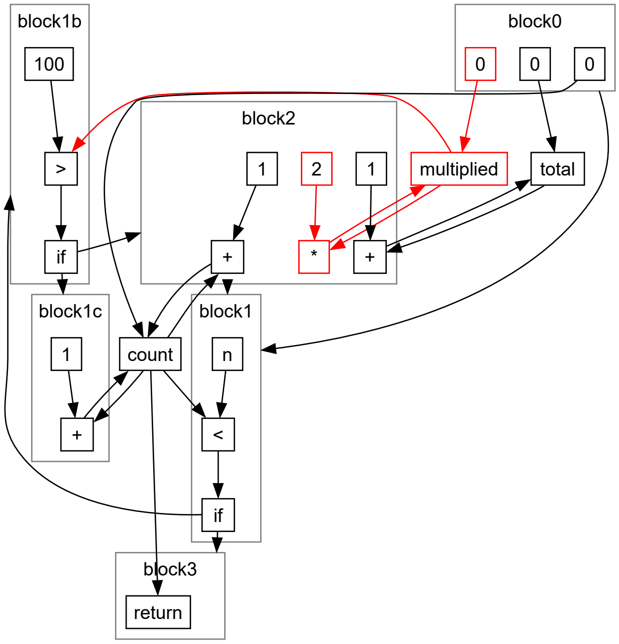
First, let's take a look at the dataflow graph:



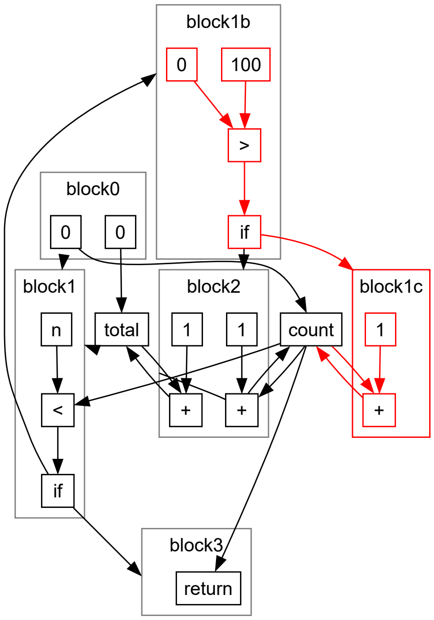
This is a bit messier than the graphs we have seen earlier, but should still be readable: you start off at `block0`, go to `block1`, if the conditional is true `block1b`, if *that* conditional is true `block1c`, and so on, eventually exiting at `block3`'s `return` node.

Removing the Dead and Unreachable

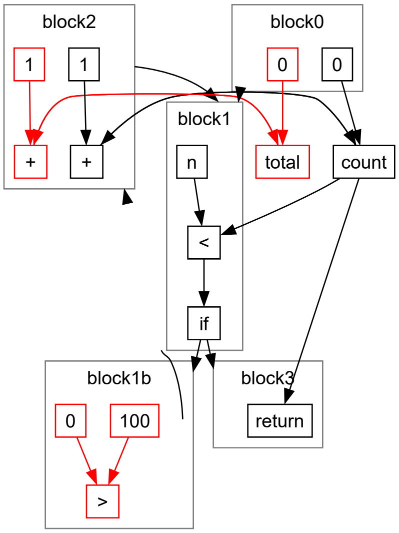
We can run the same type/constant inference we did earlier to find `multiplied -> 0`, and modify the graph accordingly:



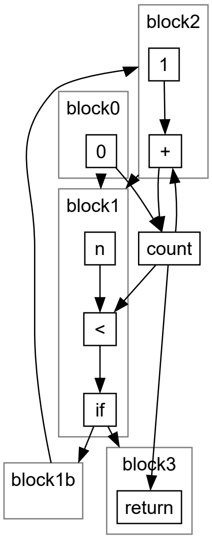
This results in the following dataflow graph:



Now, we can clearly see that the conditional in `block1b` (`0 > 100`) will never evaluate to `true` . That means the `false` branch of that conditional `block1c` is *unreachable*, and can be discarded (and the `if` -conditional eliminated):



Lastly, we can see that we have a number of "dead-end nodes": nodes like `total` and `>` which compute some value, but are not themselves used in any downstream conditional, return or computation. These can be discovered by traversing the dataflow graph backwards starting from the `return` node, collecting all the *live* nodes and eliminating the rest: the `>` , `100` , `0` in `block1b` , `total` , the `0` and `+ 1` feeding into `total` from `block0` and `block2` . This leaves you with the following:



Which when serialized back to bytecode, would produce the "ideal" program we saw earlier:

```
static int main(int n){
  int count = 0;
  while(count < n){
    count += 1;
  }
  return count;
}
```

Conclusion

In this post, we have gone through how program inference can be performed inside an optimizing compiler:

- We have walked through some manual optimizations on a strawman code snippet
- We saw that these optimizations can be done automatically by a program optimizer
- Explored various ways an optimizing compiler may choose to model your program in memory as an Intermediate Representation, settled on the Dataflow Graph representation for the rest of the post
- Walked through the mechanics of how inference can take place, both "intraprocedurally" within a single function as well as "interprocedurally" between multiple, possibly recursive functions
- An example of doing Liveness and Reachability analysis: finding parts of the program that do not contribute to the output.
- Seen how these inferences and analyses can be used to simplify our Intermediate Representation, that can be serialized into simpler, smaller output programs

Through this, we have seen how it is possible to turn a manual process of program optimization performed by a human into an automated process, running simple algorithms over simple datastructures.

This is only a small taste of the kinds of work that an optimizing compiler does to make your code fast. There is a lot this post cannot cover. For further reading, consider the following papers and books as good starting points:

- Engineering a Compiler by Keith D Cooper & Linda Torczon (https://www.amazon.com/Engineering-Compiler-Keith-Cooper/dp/012088478X)
- Combining Analyses, Combining Optimizations by Cliff Noel Click Jr. (https://www.researchgate.net/profile/Cliff_Click/publication/2394127_Combining_Analyses_Combining_Optimizations/links/0a85e537233956f6dd000000.pdf)
- Advanced Compiler Design and Implementation by Steven Muchnick (https://www.amazon.com/Advanced-Compiler-Design-Implementation-Muchnick/dp/1558603204)

(https://www.handsonscala.com)
About the Author: *Haoyi is a software engineer, and the author of many open-source Scala tools such as the Ammonite REPL and the Mill Build Tool. If you enjoyed the contents on this blog, you may also enjoy Haoyi's book* ***Hands-on Scala Programming*** *(https://www.handsonscala.com)*

← Working with Databases using Scala and Quill (WorkingwithDatabasesusingScalaandQuill.html)

Beyond Liskov: Type Safe Equality in Scala → (BeyondLiskovTypeSafeEqualityinScala.html)

Updated 2019-11-09 (https://github.com/lihaoyi/blog/commit/508a910612e5c739bcc0a759d5a06af2709be9f5)

ALSO ON **LIHAOYI.COM**

uJson: fast, flexible and intuitive JSON …
6 years ago · 11 comments
uJson: fast, flexible and intuitive JSON for Scala uJson is a new JSON …

Compact, Streaming Pretty-Printing of …
5 years ago · 1 comment
Compact, Streaming Pretty-Printing of Hierarchical Data Pretty-printing …

Automatic Binary Serialization in …
5 years ago · 2 comments
Automatic Binary Serialization in uPickle 0.7 The latest version 0.7 of …

How I Self-Published My First Technical …
2 years ago · 3 comments
How I Self-Published My First Technical Book Last year I wrote and …

**E
P**
4
E
w
S