Optimizations on Compiler Design: Constant Folding Precomputations and Reduced Type Checking

Erick Alanis
University of Colorado Boulder
Boulder, USA
erick.alanis@colorado.edu

Praneeth Brungi
University of Colorado Boulder
Boulder, USA
praneeth.brungi@colorado.edu

Abstract

Compilers are essential for converting high-level languages like Python into low-level languages such as x86 assembly. Throughout the pipeline of this translation process, many optimizations to the user provided code can be done to speed up execution of the assembly and subsequently, binary. This paper discusses two types of optimizations experimented with to further reduce the size of the binary and speed up execution.

The first optimization was reducing the amount of type checking involved during runtime using dictionaries to associate variable names to their last known variable type instead of needing to type check them whenever an operation involved them. This results in a smaller abstract syntax tree (AST) and subsequently, smaller binary.

The second optimization was applying common optimizations directly to the source code instead of any intermediate representation (IR). The common optimizations applied was a mixture of constant folding and elimination of dead stores. Once directly applied, pre-computations involving operations such as a summation between two constants are possible. The transformed code requires less type checking due to fewer variable operations.

The resulting code from both of these optimizations resulted in significant size reduction and faster executions compared to the original explication and optimization process used.

Keywords: Compilers,Optimizations,Local Value Numbering,Explication, Deadstore Elimination, Precomputations

ACM Reference Format:
Erick Alanis and Praneeth Brungi. 2024. Optimizations on Compiler Design: Constant Folding Precomputations and Reduced Type Checking. In *Proceedings of* . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In the rapidly evolving world of software development, compilers serve as a crucial bridge between high-level programming languages and low-level machine code. High-level languages such as Python offer programmers abstraction and flexibility, enabling the creation of complex applications without requiring a deep understanding of the underlying architecture. However, the cost of abstraction often leads to inefficiencies, which can result in suboptimal runtime performance. This is where the role of compilers becomes paramount. They not only convert high-level code into a format that machines can understand but also play a critical role in optimizing that code for efficient execution.

Compilers are more than mere translators; they are sophisticated tools that analyze and refine code to ensure optimal performance. The translation process involves converting the high-level source code into an intermediate representation (IR), followed by further translation into target machine code, such as x86 assembly. This multi-stage process offers opportunities for optimizations at each step. By transforming code and eliminating inefficiencies, compilers help achieve smaller binaries and reduced execution times.

Optimization techniques aim to improve different aspects of compiled code, including execution speed, memory usage, binary size, and overall efficiency. Common optimizations include dead code elimination, constant folding, loop unrolling, and inlining. However, some challenges remain due to the need for runtime type checking, which adds overhead and may slow down execution, particularly in dynamically typed languages like Python.

In dynamically typed languages, variable types are not explicitly declared, allowing variables to hold values of any type. This flexibility requires runtime type checking, which ensures the compatibility of variable types during operations. However, frequent type checking incurs significant performance costs due to the overhead involved in verifying types before every operation.

Moreover, dynamically typed languages often have verbose syntax trees, which lead to larger intermediate representations and final binaries. As a result, reducing the overhead of type checking and minimizing the size of the abstract syntax tree (AST) become critical goals for improving overall performance.

The compiler we built as part of the CSCI 4555-5525/ECEN 4553-5523 Compiler Construction takes a python file and outputs an assembly file. The compiler was bootstrapped from a smaller subset of python called P0. The subset we are using as the base for optimizations is called P1 which adds support for types. Along with types, it also supports dictionaries and lists.

The figure 1 shows the flow we have in place for P1. All of

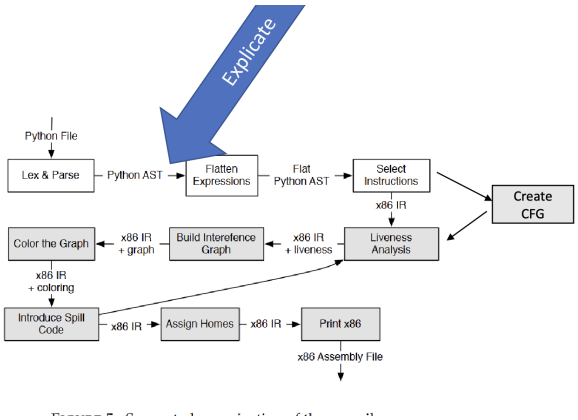


Figure 1. Flow of the compiler for P1 .

the optimizations we are planning on doing are done before the flattening at the beginning of the flow. This project introduces two optimization strategies aimed at reducing runtime overhead and binary size in dynamically typed languages like Python.

- **Optimized Type Checking Using Dictionaries**: Instead of relying solely on runtime checks, this approach maintains a dictionary that maps variable names to their most recent types. Each time an operation is performed, the compiler consults this dictionary to determine if a type check is necessary. If the variable type has not changed since the last operation, the check can be skipped, resulting in a more compact AST and reduced binary size.
- **Source-Level Optimizations**: Applying optimizations such as constant folding and dead store elimination directly to the source code enables pre-computation of expressions involving constants and removes unnecessary operations. Constant folding replaces expressions with known values (e.g., replacing 3 + 5 with 8), while dead store elimination removes variables or assignments that do not impact the final output. By addressing these issues at the source level, the resulting code has fewer operations and requires less type checking, leading to smaller binaries and faster execution.

The results

2 Motivation

Modern compilers are tasked with producing efficient binaries not only for improved runtime performance but also for reduced code size. These goals become even more pertinent when considering dynamically typed languages like Python, where abstraction introduces additional layers that can inflate both binary size and runtime overhead. The optimizations that were practiced in class, including Local Value Numbering (LVN) and dead store elimination, were effectively applied to the p0a subset of Python, yielding a noticeable reduction in code size. However, these techniques have their limitations, especially when considering more complex subsets such as P1, which introduces types into the language subset.

In the case of P1, the inclusion of explicit type information leads to a significant increase in the size of the compiled binaries. This is largely due to the need for explication: the process of converting high-level constructs into lower-level representations that more closely align with the underlying hardware architecture. This transformation inevitably introduces new intermediate variables and requires additional checks to ensure type consistency throughout the program. As a result, the final binary output can expand considerably, often by a measurable margin (provide the specific percentage increase if known).

In light of this issue, there is a pressing need to further refine optimization strategies to maintain binary efficiency despite the added complexity of explicit typing. This motivates the exploration of innovative optimization techniques aimed at reducing unnecessary code bloat and minimizing runtime overhead. Achieving a smaller binary size not only makes the codebase easier to manage and comprehend but also contributes to significant performance improvements.

3 Type Checking Optimization

We have two types of type checking optimizations designed. One with a dictionary in the explicate which we have been talking about and the other with a dictionary in the source code keeping track of the type. The purpose of the second design is to maintain Python's dynamic typing while optimizing, meanwhile the first design leans toward static typing because it determines types prior to runtime.

3.1 Dictionary in Explicate

In figure 1 as we previously observed that explication is happening right after the source code is parsed and converted into an AST. The explication process visits each node and based on the operation or type, it adds the type info required in order for the binary to able to execute them dynamically.

To achieve this we have been using some run time functions provided by the instructor. "Inject" takes a type and adds type info by making the bottom two bits which are not used to store the value when a variable is defined(keeping in mind that we are designing for a 32 bit architecture). This is also referred to as "Boxing". So every type after being boxed will have the same type which we refer to it as "pyobj". Once all the types are converted into pyobj. There are helper functions in the runtime for us make the decision of using the correct function?computation. This is known as "dispatching". Before performing the operation we "Project" back the value from pyobj into type. Since the lists and dictionaries are considered as 'bigs' in this flow. There needs to be 5 condition checks of types as int and bool operations are valid in python. So a simple addition operation is taking around 60 lines of code after flattening the AST, which is a lot.

This repeats for every operation irrespective of the operands involved. To reduce this we maintain a dictionary and update it in an assign node every time me visit with the type it currently has. In the visit BinOp we see any of the variables are already in the cache and if they are we check for what type and do the operation with inject and project. The listing 1 shows the AST after the explicate in our optimized explication. When flattened this will be less than 10 lines.

Listing 1. Example of an AST after reduced explicate

```
Expr(
    value=Call(
    func=Name(id='print_any'
         , ctx=Load()),
    args=[
        InjectInt(
            value=BinOp(
            left=UnboxInt(
            value=InjectInt(
                 value=1)),
            op=Add(),
            right=UnboxBool(
            value=InjectBool
                 (value=True)
                 )))],
    keywords=[]))],
    type_ignores=[])
```

This further required modifications in the flattening to get this working as the flattening was expecting a complicated AST.

3.2 Dictionary in Source Code

In this design we add the dictionary in the source code itself. We do this in the AST itself as it will make it easier to use the same dictionary in the later part of explicate. When we visit the module node we add the dictionary to make sure that it is defined before anything else. Then we add the check to see if the operands are already part of the dictionary before we do any boxing. In the listing 2 below we can see the cache check in action. We see if both nodes are in the dictionary and if not we proceed with our usual operation.

Listing 2. The check before we explicate

```
body=[
    IfExp(
        test=BoolOp(
            op=And(),
            values=[
                Compare(
                    left=Constant(
                        value=Name(id='x', ctx=Load())),
                    ops=[
                        In()],
                    comparators=[
                        Name(id='cache', ctx=Load())]),
                Compare(
                    left=Constant(
                        value=Name(id='d', ctx=Load())),
                    ops=[
                        In()],
                    comparators=[
                        Name(id='cache', ctx=Load())])]),
```

Now inside each condition for type checking we update the dictionary. Since P1 only supports ints and bools as types 1,2 and 3 are shorthand notation for int,bool,big. The below listing 3 gives an example of how both operands' type is being updated to int(1).

Listing 3. The cache update

```
body=[
    Assign(
        targets=[
            Subscript(
                value=Name(id='cache', ctx=Load()),
                slice=Name(id='x', ctx=Load()),
                ctx=Store())],
        value=1),
    Assign(
        targets=[
            Subscript(
                value=Name(id='cache', ctx=Load()),
                slice=Name(id='d', ctx=Load()),
                ctx=Store())],
        value=1),
```

However this may optimize the runtime but definitely makes the code size worse. But the advantage of this design is that it is easier to scale for any further subsets of python which we might bootstrap.

4 Source Code Optimizations

Because of the dynamic nature of Python, type checking is required to know if two variables can be used together in an operation. The explication process transforms the source code's abstract syntax tree (AST) to include Python's type checking process in the binary. The explication process of a source code's abstract syntax tree generates a hard to digest AST tree. Paired with the flattening process, where an AST is transformed back into the higher level language while resembling the assembly's structure, the result is oftentimes hard to interpret and debug during compiler construction.

Making matters worse, the resulting intermediate representation (IR) from the flattening process is even longer when its operations are further broken up to match the assembly's syntax.

The resulting complexity makes implementing common optimizations such as deadstores and constant folding a daunting task. In order to avoid the complexities that the explication process introduces, the idea to apply the optimizations to the source code instead becomes a more attractive and viable approach. With the propagation of literals (fixed values) throughout the code, a further optimization becomes possible, pre-computations. Instead of computing every simple expression at runtime, if all the information needed is available, pre-computing these expressions during compilation can shorten the binary and subsequently, execution time. With all these optimizations involved, the results of each process in the pipeline should be easier to look through and debug during the compiler construction process.

4.1 Constant Folding

During the construction of the compiler, two subsets of python were to be supported. The first subset, known as P0, involved constants, variables, negation, addition, booleans, boolean operations, the print and eval functions and basic control flow such as if statements and while loops. The second subset, known as P1, involved lists, and dictionaries. Will refer to the subset of P0 without control flow as P0A.

When working with P0A, the task of implementing constant folding is trivial. Look through the flattened code and associate assigned values (with the exception of eval function calls) to their respective variables while updating the associated value when necessary. If the variables are referenced in any operation, replace the variable's name with their contents.

Listing 4. Original Code

```
tmp0 = 1
tmp1 = 2
tmp3 = tmp0 + tmp1
tmp4 = tmp1 + tmp3
print(tmp3 + tmp4)
```

Listing 5. Optimized Code

```
tmp0 = 1
tmp1 = 2
tmp3 = 1 + 2
tmp4 = 2 + 3
print(3 + 5)
```

One may be deceived into believing that this is not an optimization due to the same code length, however, that is not the case. The central processing unit (CPU) takes the following steps when executing code: fetching, decoding, executing. When the CPU fetches the next instruction, it does it alongside the operand. With the original code, upon needing to get the value stored in, say, tmp0, the CPU has to then fetch the value stored in the main memory (or a register) to successfully execute the instruction. However, if the operand is instead a constant, the constant is fetched alongside the instruction and subsequently, no further fetching is required.

When dealing with P0, the process of constant folding becomes more complex due to control flows. Take for example the following code:

Listing 6. If Statement

```
b1 = eval(input())
tmp0 = 1
if b1:
    tmp1 = 1
else:
    tmp1 = 2
print(tmp0 + tmp1)
```

Because the value of b1 will not be known until runtime, constant folding may not be applied to tmp1. Only tmp0's value is known and therefore can be propagated during compile time. Constant folding may be applied to control flows when it is only applied to the code block within it and not outside it.

While constant folding may speed up performance, it doesn't necessarily reduce code size or even the necessity of explication. Another form of optimization is paired with constant folding to handle useless code.

4.2 Dead Store Elimination

After performing constant folding throughout the source code, we are now left with the code shown previously in listing 3. However, we are now presented with another opportunity for further optimization, dead store elimination. In short, dead store elimination is the process of removing lines of code where variables are assigned values and then not used later on. This process is usually done with an IR and not the source code. Typically the results from liveness analysis are looked at to easily look at which assignments are used in the IR and which ones are not.

4.2.1 Liveness Analysis. When working with P0A, such a system isn't necessary. Just like liveness analysis, you start at the bottom of the code and work your way up. During which, if a variable is detected in a line of code and it is not being assigned a variable, it is added to a list. Once you reach a line where an assignment takes place, you check the list to see if the variable has been placed there prior to the line. If it is not in the list, you delete the line. If it is, the line stays but the variable is popped off the list. Shown next each line in listing 5 below are the sets of live variables.

Listing 7. Original Code

```
tmp0 = 1            # {}
tmp1 = 2            # {}
tmp3 = tmp0 + tmp1  # {}
tmp4 = tmp1 + tmp3  # {}
print(tmp3 + tmp4) # {}
```

Listing 8. Optimized Code

```
print(3 + 5)
```

With P0 in the mixture, the process becomes more complex. Due to the branching capabilities of P0, a more robust system is required. That is when you turn back to liveness analysis. However, the challenge is that the analysis is normally performed on IR and not higher level code. It therefore

has to be refitted to work with the source code. The same algorithm is used; If a write happens (an assignment), it is removed from the set of live variables. All other cases will be reads.

4.2.2 Control Flow Graphs (CFG).

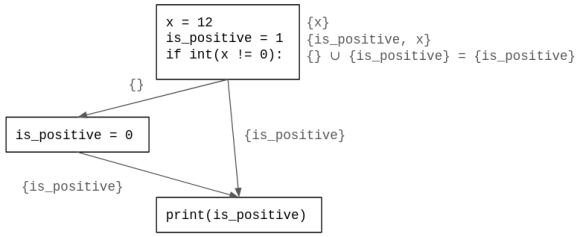
The difference between what was being done for P0A and P0 is the usage of Control Flow Graphs (CFGs). Because not every line of code will be executed, we must consider the liveness of each branch separately. We do this by getting the last live set from every child code block of the current code block and combine them to create the starting live set.

Listing 9. Original Code

```
x = 12                  # {x}
is_positive = 1         # {is_positive, x}
if int(x != 0):         # {is_positive}
    is_positive = 0 # {is_positive}

print(is_positive)  # {}
```

Figure 2. CFG



Applying the process to the source code increases the complexity due to the lack of clear endings to the control flows. In an IR, the process is more straightforward. Anytime you see a jump instruction, you know you've reached the end of the code block. With the source code (and especially with python), you can only base it off of indentation. The jump instructions in an IR also clearly paint the next code block to be executed. This is not the same for the source code where even though the indentation may have changed, you may have encountered the else code block of an if statement which doesn't follow from the body of an if statement. One must keep track of if statements and correctly pair them with both their body and else blocks. Or if no else block is present, then to the code block after the if statement. After applying constant folding to the above source code and then applying dead stores with CFGs, we get the following result shown in Listing 8.

Listing 10. Optimized Code

```
is_positive = 1
if int(12 != 0):
    is_positive = 0

print(is_positive)
```

4.3 Pre-computations

After both constant folding and dead store removal has been applied, the opportunity to apply pre-computations to the source code often presents itself. The easiest approach to do these computations is to traverse the abstract syntax tree (AST) of the source code. Let's consider the simple example of the optimized code first shown in listing 6 and its AST.

Listing 11. Optimized Code

```
print(3 + 5)
```

Listing 12. Abstract Syntax Tree

```
Module(
    body=[
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    BinOp(
                        left=Constant(value=3),
                        op=Add(),
                        right=Constant(value=5))],
                keywords=[]))],
    type_ignores=[])
```

As stated before, these optimizations are meant to take place before any explication can occur. If the AST is traversed, it is possible to look for specific nodes such as BinOps. Because the only operation between two values supported by P0 are boolean operations and addition, we will focus on those. In Listing 10, once traversed to the BinOp, after checking that the op parameter contains a supported operation, we convert that BinOp expression back into Python. In this case: the expression "3 + 5". Once recreated, we simply place the string into Python's eval function to get the expression computed. We now replace the whole BinOp expression with a literal containing the pre-computed value.

Listing 13. Optimized Abstract Syntax Tree

```
Module(
    body=[
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    Constant(value=8)],
                keywords=[]))],
    type_ignores=[])
```

If turned back into Python, the result code would be "print(8)". The same process can be applied with boolean expressions as well to pre-compute them to either True or False. Once this process has been applied, we can go through the explication, type checking, and any other process that normally takes place.

However, the compile time must be taken into account when choosing to handle pre-computations. How many expressions to precompute must be carefully weighed. In theory, an inputted source code with all the information needed

is available could all be pre-computed but this could significantly increase compile time. In these scenarios, the option to change the "aggressiveness" of the pre-computation should be supplied by the compiler's user. A good balance would be to only pre-compute simple expressions such as addition and boolean expressions.

5 Results

Figure 3. Assembly line amounts

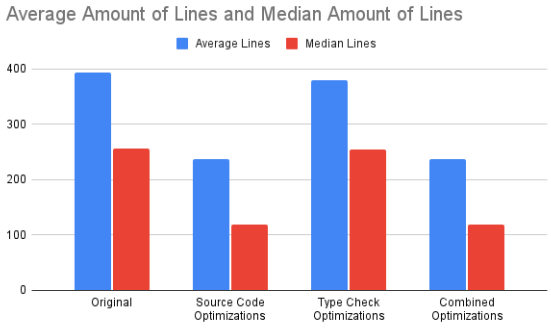


Figure 4. Binary sizes

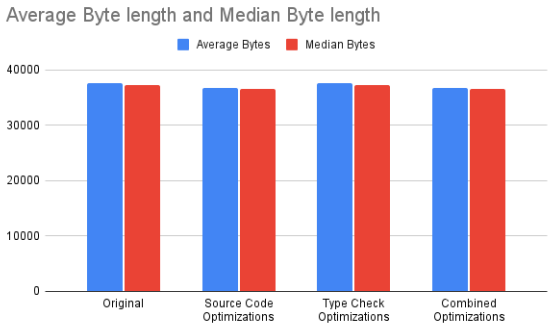
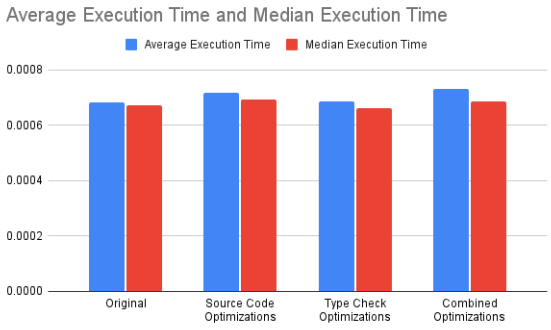


Figure 5. Binary execution times



The results presented in these charts were generated by comparing the effects of different optimization strategies on the set of autograde tests. We are comparing our optimization strategies with the baseline lab 4, individually and then combined. For each comopiler, we are measuring

- **Assembly Line Counts**: The number of lines in the generated assembly code.
- **Binary Sizes**: The size of the compiled binaries in bytes.
- **Execution Times**: The time each binary took to execute

6 Evaluation

- Figure 3 Assembly Line Counts: The original compiler has the highest average line count in the assembly code. Source code optimizations significantly reduce the number of lines by almost 50 percent, while type-check optimizations reduce by 5 percent. Combined optimizations also reduce the line count but not as drastically as source code optimizations alone.
- Figure 4 Binary Sizes: The size of the binaries remains relatively consistent across all optimization types. There's only a slight reduction in the byte size with source code optimizations, but the variations are not substantial. This suggests that binary size is less sensitive to the different optimizations compared to other metrics. This can be attributed to the fact that the runtime functions take significant amount of space in the binaries
- Figure 5 Binary Execution Times: The execution times also remain similar across all optimization strategies. The original and source code optimization strategies have comparable execution times, while the type check optimizations introduce a marginal reduction in execution time. The combined optimizations appear to have an average execution time between the source code and type check optimizations.

Overall, the results suggest that source code optimizations significantly reduce assembly line counts while maintaining comparable binary sizes and execution times. Type check optimizations have less impact on assembly line counts but maintain similar performance metrics to the original programs.

7 Future Work

- Extend the type checking optimization to the comparator operator, which we use a lot in the if and while loops. From the results, we can infer that it would make a lot of impact on the runtime.
- Finish the implementation of the second design type of the type checking optimization.
- Extend both of our optimizations to P2 which adds functions.

- Correctly implement CFG for dead store removal in the source code
- Apply constant folding to other literals such as booleans and bigPyObjs (lists, dicts)
- Apply pre-computations for other literals such as booleans and bigPyObjs (lists, dicts)

8 Conclusion

We were able to successfully implement the optimizations we set out to achieve i.e reduce type checking and constant folding along with pre computing and were able to reduce the number of lines in the code by almost 50 percent for the autograde test cases. We were also able to observe a trend in the runtimes and how extending the optimization would be a good idea.