Constant Folding and Type Checking

Irakli Zhuzhunashvili
University of Colorado at Boulder
Department of Computer Science
irakli.zhuzhunashvili@colorado.edu

Evan Roncevich
University of Colorado at Boulder
Department of Computer Science
evan.roncevich@colorado.edu

ABSTRACT

By folding constants of the ast, propagating constants, and removing unnecessary explication through type checking, compilers will generate smaller, faster code. The main issue with compilers before these optimizations is that unnecessary structure in the ast creates more temporary variables, which in turn means more lines of code and a greater difficulty in coloring registers. Our proposed methods speed up the compile-time while producing a faster run-time.

1. INTRODUCTION

Due to Python's dynamic typing, explication is a required yet complex process for compiling. We will present methods of removing large amounts of explication from the compiling process through constant propagation and static type-checking.

The core issue with explication is the enormous amount of code and conditions required for the explication phase. To perform and the most menial tasks such as 1+1 requires at least 6 conditions to determine the type at run-time. This leads to creating a large number of variables which need to be colored. Because the coloring of registers requires the greatest run-time of the compiling, O(n^2logn), removing extra variables will have the greatest compiling improvement.

Constant Propagation and folding will help as well by removing needless constant addition to improve performance.

In addition to the improved compiling time, by not requiring explicating in the code, the actual program will have a great improvement in speed at run-time and due to the fewer lines of assembly and will be easier to debug, which was another problem with the old compiler.

2. EXCESS CONSTANTS

The P1 compiler has many drawbacks when in comes to simple operations such as addition and assign. For a simple addition of a few variables, the compiler generates x86 code with far too many lines and variables. We'll use the following example to show the aforementioned problem:

```
x = 1 + 2 + 3 + 4 +\
5 + 6 + 7 + 8 + 9 +\
10 + 11 + 12 + 13 +\
14 + 15 + 16 + 17 +\
18 + 19 + 20+ 21 +\
22 + 23 + 24 + 25 +\
26 + 27 + 28 + 29 +\
30 + 31 + 32 + 33 +\
34 + 35 + 36 + 37
print 1 + 2 + 3 + 4 + x
```

Listing 1: Many additions

For such a simple piece of code, the P3 compiler generated assembly of 3174 lines. Compilation time is much longer than it should be because all the temporary variables generated must be explicated and colored. Having 3000 instructions means the actual run-time of the x86 code generated is excessively long and difficult to debug.

3. CONSTANT FOLDING

One of the optimization which helps solve the previously defined problem is Constant Folding. This is the process of identifying all the constant expressions and evaluating them during compile time. In other words, if an expression consists only of constants, instead of taking it and evaluating it during run-time, it is computed during the compilation.

This is an important optimization because after just a few adds, many temporary variables are generated which causes the explicate pass to generate many conditionals. This also makes the run-time of the compiler longer due to the coloring phase which will have to deal with more temporary variables than optimal. An example of this is the following:

```
x = 1 + 2 + 3 + 4
y = 5 + 6 + 7 + 8
print x + y
```

Listing 2: Addition chain

Instead of the compiler going through each add and creating a temporary variable (Listing 3: Temporary variables),

```
temp0 = 3 + 4
temp1 = temp0 + 2
temp2 = temp1 + 1
x = temp2
temp3 = 8
temp4 = temp3 + 7
temp5 = temp4 + 6
temp6 = temp5 + 5
y = temp6
print x + y
```

Listing 3: Temporary variables

it will evaluate the addition chain during compile time (Listing 4: Folded) and use the new AST as its new base.

```
x = 10
y = 26
print x + y
```

Listing 4: Folded

After applying this to the P3 compiler, the change was drastic, for example, when the "Listing 1: Many additions" code is compiled, instead of the 3174 lines of x86, only 14 are generated. Also, the compile time is a few milliseconds compared to the old one which took about two seconds.

IMPLEMENTATION

The Constant Folding step of the compiler is done right after the Constant Propagation pass (which for reference is right after the uniquify pass) for reasons discussed later on. The current implementation is fairly straightforward, it's very much like many of the other passes in the compiler, where it recurses over the AST and checks the instance of each node.

The main idea is to recurse down until an Add node is found, once this is complete, the compiler checks both the left child node and the right child node. If either of them is an Add or UnarySub, it keeps recursing on it. If it is a Const, then the value of the Const is added to an array, which will be referred to as the constants list. If it's none of the above, then the node is added to another list, which will be referred to as the expressions list, which is combined with the constants list and returned. Once the recursive call returns up to the parent of the main Add node, the lists are combined to form and Add node with the combined Consts.

INTERESTING EXAMPLES

At first, the compiler only supported folding of expressions which contained only constants, but after adding the expressions and constants lists, the compiler was able to fold the following:

```
def f():
  y = 12
  x = 1 + 2 + 3 + -\
  (input() + 12 + 2 + 1)\
  + 20 + y + - (1 + input())\
  return x

print f()
```

into:

```
def f():
  y = 12
  x = 10 + - input()\
  + y + - input()
  return x

print f()
```

Listing 5: Folded with input

4. CONSTANT PROPAGATION

Constant Propagation goes very well with Constant Folding, reducing the number of temporary variables even further. It is the process of substituting known values of variables into later compilation during compile time. This is another important optimization for the same reasons as Constant Folding, it reduces the number of temporary variables, and also makes it so that there are fewer live variables at a given moment due to them being replaced with constants.
The following is an example of constant propagation: if the following is run through the constant propagation pass.

```
x = 1
y = x + 2
z = y
a = z
print a
```

Listing 6: Variables

Propagating the Constants would convert to the code below.

```
x = 1
y = 3
z = 3
a = 3
print 3
```

Listing 7: Variables

This is the simplest type of constant propagation. The one that most compilers use now is called the sparse conditional constant propagation, which also deals with deadcode elimination, so it would produce something along the lines of just:

```
print 3
```

This is one of the things that could be added to the current implementation of Constant Folding and Propagation. The only concern is to ensure removable code does not need to be heapified.

IMPLEMENTATION

The current implementation relies on folding and propagating until the AST converges, but due to a few complications, it has been left out and only does one iteration. This could be improved by having the fold and the propagation stage in the same pass. Propagation needs to occur before folding because the compiler needs to know what the variable is so that it can fold it with the other constants, this way, the overall optimized code will come out to be shorter. For example, if the following code

```
x = 1
y = 1 + 2 + 3 + x
```

were to be run with folding first, then this would be generated:

```
x = 1
y = 6 + 1
```

versus the following:

```
x = 1
y = 7
```

The implementation of the Constant Propagation pass itself is similar to the Constant Folding, the difference is that the compiler has a dictionary in which it stores the variables and the values that they map to. This gets updated every time there is an assign. When propagation pass is recursing on the AST, as soon as it reaches a Name node, it looks up the name in the dictionary and returns a Const with the corresponding value. If the dictionary doesn't contain the value, it returns None. This way, when the parent receives the return value, it knows whether to replace the Name with the Const node or leave it untouched.

5. TYPE-CHECKING

With the ultimate goal of removing any unnecessary conditionals and intermediate variables required for performing a dynamic type-checking, we can use static type-checking to determine the necessity of explication. With every single addition, unary substitution, and comparison, around 6 conditional statements are used to allow type-checking at run-time. The extra generated assembly makes compiling and debugging large systems a near impossible feat.

By static type-checking before the explication process, we can avoid any needless conditionals, allowing the compile process to run significantly faster. In the following example, we can see the complexity required for simple statements

```
x=input()
y=input()
print x+y
```

Listing 8: input int

This performs an explication for the addition to the following

```
x=input()
y=input()
if (isType('int',x) or isType('bool',x))
and(isType('int',y) or isType('bool',y)):
    print int(x)+int(y)

elif isType('big',x) and isType('big',y):
    print add(big(x),big(y))

else
    error()
```

Every single addition required this explication, even when we know the typing such as in (Listing 8:input int) where our scope dictates input() as integers.

IMPLEMENTATION

We need to determine if the current typing is usable without run-time conditionals. This is accomplish by checking the type of any variable's dependencies at any position of the program. Before the explicate phase starts, an extra phase runs through the AST. At each statement of the AST, we store a list of every known variable and its known type at that position. Storing the mapping at every statement allows the type of variables to change throughout the program without needing to explicate.

The flag of the mapping can be "bool" if assigned from 'True' or 'False'. The flag 'int' is assigned to a variable if set to a constant, or 'input()'. The heapified variables and list variables will be stored as 'list'. For any other dependencies, the pass checks the previous mapping. When the typing of the variable is not known, we need to label the variable as 'unknown'.

```
    x=input()
    ['x':'int']
    y=input()
    ['x':'int','y':'int']
    print x+y
    ['x':'int','y':'int']
```

By propagating the known types, the explication can determine typing at any point in the code. The exceptions to the system are in if-statements, if-expressions, list elements, and function arguments.

The function arguments cannot be known unless out-of-scope analysis occurs because multiple types can be passed into a function. We label all function arguments as 'unknown'.

For list elements, because list elements can be added dynamically, it is difficult to determine the typing of a newly added element at compile-time. Lists element types can also be modified outside functions. The solution is to just treat any element of a list as an 'unknown'.

The main form source of unknown typing we were concerned with resides in if-statements and if-expressions. The implementation compares the end states of both branches. When the types of the variables are the same, there is no conflict and we can combine the sets. If the branch variable types are different, we will set the variable as an 'unknown'.

```
    if input():
        x=1
        ['x':'int']
    else:
        x=[1,2,3]
        ['x':'list']
    ['x':'unknown']
    print x
```

The result of this pass is a mapping at every line of code to the typing if known. We can modify the explicate phase to check this mapping which will determine if we can avoid conditionals.

In the explicate phase, we when receiving an instance of Add and Compare, using the list associated with the current line of code, we remove the explication if the typing of both sides are known.

To perform this, we pass up the current typing of each object. When returning a Const, List/Dictionary, or subscript, the explication stores the current value as a 'int', 'big', 'unknown' respectively. When returning Names, pass up the map typing if present, otherwise 'unknown'. An example of an addition

```
    ['x':'int','y':'int']
    print x + y
```

knowing x and y are ints

```
    print int(x) + int(y)
```

Without needing to check the typing dynamically

Using a test of large numbers of additions

```
print -input() + input() + -input() + input()
    + -input() + input() + -input() + input
    () + -input() + input() + -input() +
    input() + -input() + input() + -input()
    ...
```

Listing 9: Large inputs

When compiling the test originally, the compilation took 1 min 10 secs to compile, outputting 59038 lines of x86 code. After running the type-checking, the compile-time reduced to 5 secs with 10065 lines. With fewer conditionals and arguments, the coloring is faster, meaning faster compilation.

6. CONCLUSION

The goal of the optimizations is to use fewer intermediate variables with the expectation that compilation will occur faster with fewer variables and lines of x86 code.

Through the combination of Constant Folding, Constant Propagation, and static Type-Checking we reduce the number of intermediate variables used in the compilation to achieve a faster coloring, smaller compilation, and faster run-time.

Constant Folding and Propagation were successful in removing code with large numbers of Constant additions. When running sample tests, a program originally producing 3174 lines of code only had 14 lines afterwards. These work for commonly used constant statements.

Type-Checking was successful as well by removing explication with a guarantee of not changing the functionality. The compile-time of large additions and comparisons became significantly faster after the type-checking. Type-checking's utility is broad and will improve compilation of nearly every program, so the results are good.

The optimizations were interesting to design and solve many of the timeout issues caused by previous test in the semester.

7. FUTURE WORK

To improve the propagation and improve the accuracy of the type-checking, we can implement Constant Propagation of If, While, and Functions.

For if statements, if the conditional can be determined at compile-time, remove the conditional all together.

```
if 0:
    print 0
else:
    print 1
```

Can be converted to:

```
print 1
```

While loops can be unraveled to decrease code. The difficulty in this would be determining how far to unravel a loop or if a loop even can be unraveled. Functions can be simplified or in-lined to improve performance, but the next step would be analyzing how functions are used.

Another advanced implementation to extend type-checking would be analysis for heapified variables and lists. The current issue with list is the elements of a list are subject to change. A list or heapified variable can be added to and modified at run-time, but if we can perform type analysis over the known types of the list, especially when related to heapified variables, type-checking will improve performance.

A simple addition to the type-checking would be to add warnings to compile-time if interactions such as additions between incorrect typing occur. Such as in the following

```
print 1+[0]
```

At compile-time a warning would occur because the types are different being added.

A further step of the type-checking is to remove the pyobj aspect of all needless types. Pyobjs were added specifically for the polymorphism, but if the type-checking determines an object's type, the compiler can avoid projectTo and injectFrom to save some compilation and run-time.

8. REFERENCES

[1] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. *ACM IGPLAN-SIGACT symposium on Principles of programming languages*, 1995.

[2] M. N. Wegman. Constant Propagation with Conditional Branches. *TOPLAS*, 13(2), 1991.