Project Write-Up: ECEN 5523/ CSCI 5525

Sabarigirish Muralidharanpillai
Xinying Zeng
Eric Grover
Chris Bubernak

Abstract

This project seeks to reduce the amount of assembly code produced and the runtime of this code by implementing Static Single Assignment Form, Type Analysis and Specialization, Dead Code Elimination, and Constant Propagation & Folding in our compiler. Although we are adding additional complexity and passes to the compiler, we also expect to see improved compile time in some situations. The proposed optimizations reduce code size early on in the compilation process which in turn reduces the amount of work that later passes need to do. The project uses Prof. Siek's compiler as a starting point and builds the aforementioned optimizations on top of it. The write up is laid out broadly as Conception (brief descriptions with examples), Implementation (details of implementation and algorithms), Compiler Layout, and Results.

Conception

Static Single-Assignment Form

Static Single-Assignment Form (SSA form) mandates that every variable can only be assigned to once. The purpose of this form is to put the code into a state that simplifies the later proposed optimizations. Because of this, we convert the AST into SSA form before performing the other optimizations and then convert back out of SSA form after finishing them.

Constant Propagation and Folding

These two optimizations work in tandem to pre-compute expressions whose operands are constant values known at compile time. Constant Propagation is the process of substituting a reference to a constant variable with the variable's value. After applying Constant Propagation we then apply Constant Folding, which looks for expressions with constant operands and performs the computations. By pre-computing some of these expressions at compile time we can reduce the amount of computations that need to be performed at runtime. The performance increase provided by Constant Propagation is particularly noticeable in programs that preform lots of calculations with constant values.

Example:

```
a = 24 + 5    # this could be folded into a = 29
b = 5 - a     # replace a with its value (34) and fold into b = 34
Return b + 37 # fold into Return 71
```

Looking at the example above we can see a situation in which applying Constant Propagation and Folding could provide some benefit. First folding could be used to simplify the assignment to “a”. Then we could propagate that value into the assignment of “b” and preform folding on its assignment. Finally, we could propagate the value of “b” into the return statement and preform folding. These changes would eliminate the need to generate a bunch of extraneous operations.

Type Analysis and Specialization

The compilers built during the course of the semester introduced substantial overhead for handling polymorphism through the creation and inspection of tags, even in cases where the type of the object is known at compile time. Type Analysis and Specialization seeks to cut down on this overhead by avoiding tagging and inspection in cases where the type of the objects being handled is known at compile time, thus improving runtime performance.

Example:

```
a = [1, 2, 3]
b = [2]
c = a + b     # at this point we know that both a & b are lists

d = 42
e = 0
f = d + e     # at this point we know that d & e are both integers
```

Dead Code Elimination

Dead Code is used to refer to program code that has no result on the output of the program. Our compiler, prior to these optimizations, did not attempt to perform any Dead Code analysis and would simply compile all source code into assembly code. By performing Dead Code Elimination we can find code paths that will never be executed, as well as code that does get executed but doesn’t affect the output (such as variable assignments that are overwritten before their value is ever read). Eliminating code that is never run will reduce the size of our assembly code files and removing code that doesn’t affect the output will do the same but also reduce running time of compiled programs.

Example:

```
y = 1
x = y               # dead code
x = input()         # not dead code because of side effect!
x = 2
if True:
       print x
else:
       x = [1,2,3]  # dead code
       print x      # dead code
```

In the above example we can see two instances in which Dead Code Elimination would help to reduce code size. The first assignment to “x” can be removed because “x” is not read from before its next assignment. The second assignment to “x” brings up an interesting situation. While this code does not affect the output of the program it is not considered dead because it has a side effect which is something that needs to be accounted for when doing Dead Code Elimination. The last piece of this that can be removed is the “Else” branch of the “If” expression. Because we know which path will be taken at compile time we can remove the “If” statement and replace it with the body of the “Then” branch.

Implementation

Static Single-Assignment Form

To convert a program into SSA form, we introduce a pass into our compiler that traverses the program from top to bottom and maintains a dictionary mapping variable names to the newest version of the variable. Every time an assignment node is found we check the dictionary to see if an entry exists for the variable. If it does, we increment the counter and replace the entry with the most recent version of the variable. If the variable is not yet in the dictionary we create a new entry for it and start its counter at 0. When we encounter a name node we look in the dictionary and replace the variable reference to the current version of that variable.

The only complicated nodes we have to worry about are if statements and while loops. In if statements variables can be assigned to different values depending on which branch is taken. Similarly, every time a loop body is executed variables can be reassigned to. Consider the following example:

Example:

```
if input():
       x = 42
       y = x +1
else:
       x = 3.14159
       y = x/3
print x + y
```

If the “x” in the then branch gets “x_0” while the else branch “x” gets “x_1” what version should the x in the print statement be? To solve this dilemma a ϕ node assignment is introduced. The semantics of the ϕ expression is to return the value of the variable corresponding to the branch that was actually taken at runtime. Using this, the example above can be translated into:

Example:

```
if input():
       x_0 = 42
       y_0 = x_0 +1
else:
       x_1 = 3.14159
       y_1 = x_1/3
x_2 = φ(x_0, x_1)
y_2 = φ(y_0, y_1)
print x_2 + y_2
```

For while loops we make use of the ϕ node again but in this case it will be inserted just before the test of the loop. This is done because control can enter the test of the loop from the preceding statement or from the end of loop.

Example: (original)

```
i = 3
m = 0
while (…):


      j = 3
      i = i + 1
      l = m + 1
      m = l + 2
      j = i + 2
      k = 2 * j
```

Example: (SSA while loop)

```
i_1 = 3
m_1 = 0
while(…):
      i_2  = φ(i_1,i_3)
      m_2 = = φ(m_1,m_3)
      j_1 = 3
      i_3 = i_2 + 1
      l_1 = m_2 + 1
      m_3 = l_1 + 2
      j_2 = i_3 + 2
      k_1 = 2 * j_2
```

Because ϕ expressions cannot be represented in x86 we have to insert a pass into our compiler to convert out of SSA form and remove them after we finish our optimizations. A simple SSA back translation strategy is to pull apart the ϕ assignments and place parts of them in each of the branches. The following example shows how this is implemented:

Example: (SSA)

```
if input():
      x_0 = 42
      y_0 = x_0 +1
else:
      x_1 = 3.14159
      y_1 = x_1/3
x_2 = φ(x_0, x_1)
y_2 = φ(y_0, y_1)
print x_2 + y_2
```

Example: (out of SSA)

```
if input():
      x_0 = 42
      y_0 = x_0 + 1
      x_2 = x_0
      y_2 = y_0
else:
      x_1 = 3.14159
      y_1 = x_1/3
      x_2 = x_1
      y_2 = y_1
print x_2 + y_2
```

Constant Propagation and Folding

To perform constant propagation and folding, the compiler needs to know, after every statement in the program, which variables contain constant values. At a high level the intuition is that the compiler can determine this by ensuring that at a specific line every possible execution path up to this point would result in the same value being stored in a given variable. To actually implement this we adopt the constant propagation algorithm introduced in “Lecture #4, 15-April-2004: Posets, Lattices and Constant Propagation” [3]. At every statement we classify each variable in one of 3 ways. If it is known to be a constant we assign it its known value. If it could potentially take on two or more values (depending on the code path that gets taken) we assign it T (or “top”). If we don’t know anything about a variable’s value then we assign it ⊥ (or “bottom”).

Example:

```
z = 3
x = 1
while (x > 0):
    if (x == 1):
        y = 7
    else:
        y = z + 4
    x = 3
    print y
```

The execution of the algorithm on the above program would be the following:

1. initial mapping is: [x->⊥, y->⊥, z -> ⊥]
2. after “z = 3”: [x->⊥, y->⊥, z -> 3]
3. after “x = 1”: [x->1, y->⊥, z -> 3]
4. after “y = 7”: [x->1, y->7, z -> 3]
5. after “y = z + 4”: [x->1, y->7, z -> 3]
6. after “x = 3”: [x->T, y->7, z -> 3]
7. The loop will get traversed one more time after which we will exit because none of the values are changing

We use transfer functions to describe the modification that traversing through an edge does to a constant mapping. Two basic binary functions are introduced: meet (⋂) and join (⋃). Meet function takes the intersection of two mappings. Join function takes the union of two mappings. The truth table of meet and join functions is as follows.

$\sqcap$	T	$n \in N$	$m(\neq n)$	$\perp$
T	T	n	M	⊥
$n \in N$	n	n	⊥	⊥
$m(\neq n)$	m	⊥	M	⊥
$\perp$	⊥	⊥	⊥	⊥

$\sqcup$	T	$n \in N$	$m(\neq n)$	$\perp$
T	T	T	T	T
$n \in N$	T	n	T	n
$m(\neq n)$	T	T	m	m
$\perp$	T	n	m	⊥

In general, the process of constant propagation algorithm is described as following stages:

1. Construct a control flow graph (CFG).

2. Associate transfer functions with the edges of the graph. The transfer functions depend on the statement or the program condition itself (the type of statement determines which functions need to be associated to each of its outgoing edges).

3. Iterate until no more changes occur.

Type Analysis and Specialization

Type Analysis

During the type analysis phase, each expression node is annotated with a type as depicted in table 1.

Chambers and Ungar [5] describe several methods for performing type specialization which include simple type analysis, extended message splitting, and iterative type analysis. In our compiler we will be implementing simple type analysis and iterative type analysis. Throughout this process we will keep track of the type of each variable in a single dictionary (thanks to the code being in SSA form).

Chambers describes a "merge node" as the first usage of a variable after control flow that can cause the variable to be assigned from multiple expressions. Since the code at this point is in SSA form, this merge node effectively is the SSA form ϕ node. The rule for type analysis of the ϕ node is described in table 1.

However, when analyzing loop control flow other merge nodes occur. Iterative type analysis is done to merge all possible types that a variable could be assigned. On the first iteration, the loop is analyzed and known type information is annotated on the expressions and variables. On the next iteration of this analysis, any new type information is merged with what was annotated from the last iteration. This continues until no type changes occur in the analysis of the loop. When merging types t_1 and t_2, if t_1 and t_2 are both type a, then the output is type a. If t_1 and t_2 are different types, then the output is type *pyobj*.

For lists and dictionaries, we would only be able to do static type analysis if up to any particular point all list/dictionary references have used constant subscripts and the list or dictionary has not been passed to another function. As soon as we encounter a non-constant subscript or the list/dictionary being passed to a function, we cannot statically determine what could have changed in the list/dictionary nor what location we are accessing. Since non-constant subscripts are quite common, no attempt is made to determine type information for the contents of lists and dictionaries. Only intraprocedural type analysis is being done. Thus all function arguments are of type pyobj.

The expression "typeof(*e*)" is used in table 1 to indicate the statically determined type of the given expression *e*.

Table 1 Type Analysis for Expressions

Expression Node	**Type Analysis Rule**
CallFunc(*e*, *args*)	if user defined function or other expression, then *pyobj* else if runtime library function, then return-type of runtime function
UnarySub(*e*)	*Int*
Const(*c*)	*Int*
Name(*s*)	type previously recorded in variable dictionary
Compare(e_1, [op, e2])	*Bool*
Or([e_1, e_2])	if typeof(e_1) == typeof(e_2) then typeof(e_1) else *pyobj*
And([e_1, e_2])	if typeof(e_1) == typeof(e_2) then typeof(e_1) else *pyobj*
Not(e)	*Bool*
List([e_1, e_2, …, e_n])	*Big*
Dict([e_1, e_2, …, e_n])	*Big*

Subscript(var,key,rhs)	*Pyobj*
IfExp(*test*, e_1, e_2)	if typeof(e_1) == typeof(e_2) then typeof(e_1) else *pyobj*
Lambda(*argnames*, *code*)	*Pyobj*
AssName(*v*)	Variable *v* is recorded in dictionary with the type of right-hand side of assignment expression
Let(*v*, *rhs*, *e*)	Variable *v* is annotated with typeof(*rhs*) typeof(*e*)
Phi(e_1, e_2)*	if typeof(e_1) == typeof(e_2) then typeof(e_1) else *pyobj*

* node introduced at SSA stage

Type Specialization

Type specialization is handled in the Explicate pass. This is done because the general concept is that we use the type analysis information we just gathered to help us avoid a lot of the overhead code introduced by polymorphism in this pass. If we know the types of operands in an expression we should be able to avoid having to create a whole mess of nested “IfExp” nodes (used for tag checking) and instead generate type specific expressions. Below is an explicate/type-specialization example for the “Add” node (in pseudo-Python). Other node types follow the same pattern.

```
...
elif isinstance(n, Add):
       l = explicate(n.left)
       r = explicate(n.right)

       tl = genTemp()
       tr = genTemp()

       if(n.left.type == big and n.right.type == big):
              e = CallFunc("add", tl, tr)
              type = big

       elif(n.left.type in (int,bool) and n.right.type in (int, bool)):
              e = Add((tl, tr))
              type = int

       elif(n.left.type == pyobj and n.right.type != pyobj):
              if(n.right.type == big):
                     bige = InjectFrom(big, CallFunc("add", [ProjectTo(big, tl), tr]))
                     inte = CallFunc("typeError")
              else
                     bige = CallFunc("typeError")
                     inte = InjectFrom(int, Add((ProjectTo(int, tl), tr)))

              e = IfExp(isBig(tl), bige, inte)

              type = pyobj

       elif(n.left.type != pyobj and n.right.type == pyobj):
              if(n.left.type == big):
                     bige = InjectFrom(big, CallFunc("add", [tl, ProjectTo(big, tr)]))
                     inte = CallFunc("typeError")
              else
                     bige = CallFunc("typeError")
                     inte = InjectFrom(int, Add((tl, ProjectTo(int, tr))))
```

```
            e = IfExp(isBig(tr), bige, inte)

            type = pyobj

        elif(n.left.type == pyobj and n.right.type == pyobj):
            e = IfExp(isBothBig(tr,tl), InjectFrom(big,
                    CallFunc("add", [tl, ProjectTo(big, tr)])),
                        InjectFrom(int, Add((tl, ProjectTo(int, tr)))))
            type = pyobj

        else
            raise Exception("type mismatch")

        e = Let(tl, l, Let(tr, r, e))

        if(n.type == pyobj and type != pyobj):
            e = InjectFrom(type, e)
        else
            if(n.type != type):
                raise Exception("type mismatch")

        return e
...
```

Dead Code Elimination

The essence of Dead Code Elimination is that we want to eliminate all statements that are not marked critical or definers. Critical statements are those that directly affect the behavior of the program (I/O statements, function calls, etc.) and definers are statements that modify the values used by the critical statements. Dead Code Elimination is performed after the other optimizations we are implementing because these optimizations may produce dead code.

Our Dead Code Elimination pass is implemented as per the algorithm described in the paper, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph* [4]. An intuitive procedural version of the algorithm is as follows. Initially traverse the code marking all code as dead except for statements that affect program I/O or call routines with side effects, assignments that produce output used by live statements, and conditional branch statements that produce output used by live statements. This is followed by a second pass that will mark the rest of the code as dead or alive. The rationale behind the second pass can be seen in the following example.

```
for each statement S do
        if S ϵ Pre Live
                then Live(S) = true
        else Live(S) = false
        end
Worklist ← PreLive

while (WorkList != 0) do
        take S from WorkList

        for each D ϵ Definers(S) do
                if Live(D) = false
                        then do

                        Live(D)= true

                        WorkList =WorkList U { D }
                end
            end

            for each block B in CD$^{-1}$(Block(S)) do
                If Live(Last(B)) = false
                    then do

                    Live(Last(B)) -=true

                    WorkList = WorkList U { Last(B) }
                end
            end
        end

        for each statement S do

        If Live(S) = false

            then delete S from Block(S)

        end
```

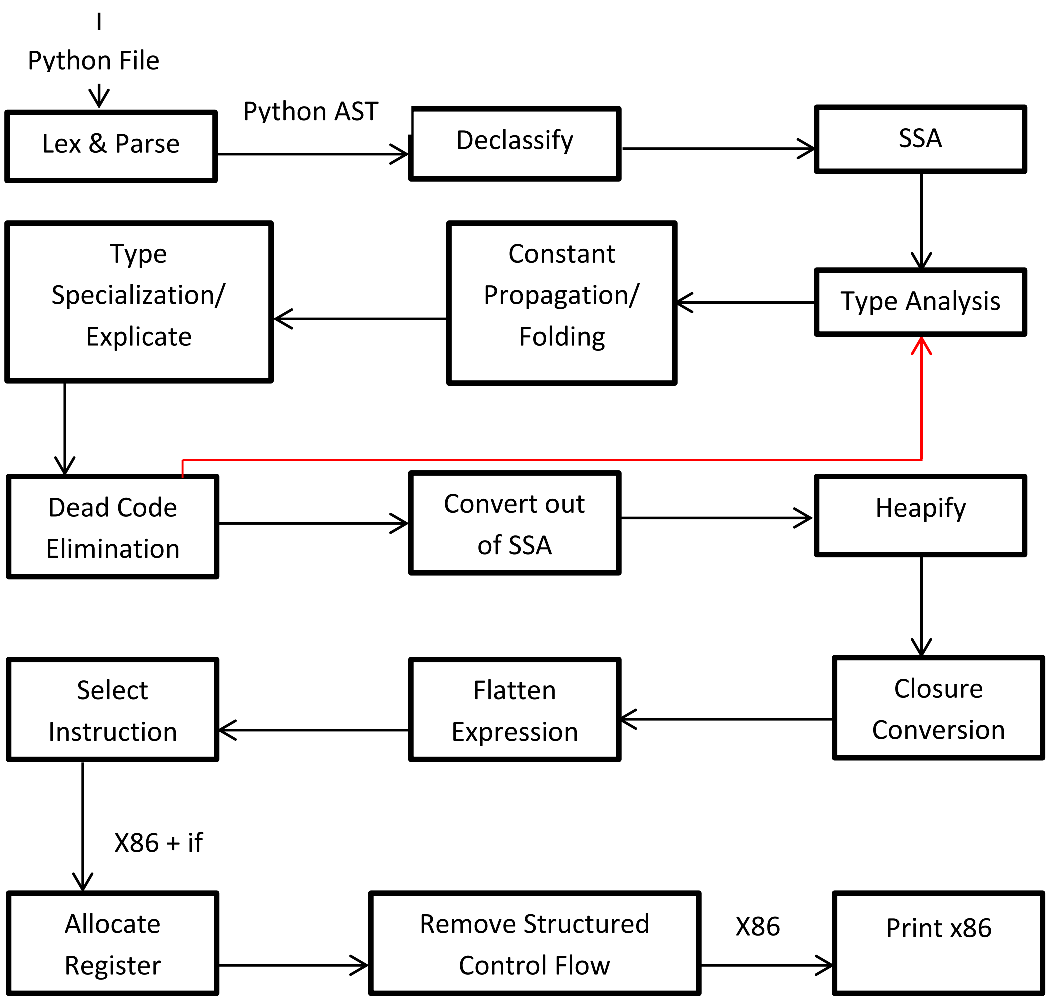
Table 2 Dead Code Elimination Terminology

Term	**Definition**
Live(S)	Indicates that statement S is live.
PreLive	The set of statements whose execution is initially assumed to affect program output. Statements that result in I/O or side effects outside the current procedure scope are typically included
WorkList	List of statements whose liveness has been recently discovered
Definers(*S*)	The set of statements that provide values used by statement S.
Last(B)	The statement that terminates basic block B
Block(S)	The basic block containing statement S
ipdom(B)	The basic block that immediately postdominates block B
CD^{-1}(B)	The set of nodes that are control dependence predecessors of the node corresponding to block B.

After the algorithm discovers the live statements all those still not marked live are deleted. Deleting code here can change the structure of the program and possibly lead to more parts of our code being optimized. Because of this, after dead code elimination it is usually smart to run our other optimization passes again. The details for how this will be done will be discussed in more detail in the compiler organization section.

Compiler Organization

Now that we have discussed the implementation details surrounding the optimizations that will be performed it is important to understand where these changes are implemented in our compiler. The diagram below details the structure of our compiler after implementing the new optimizations.



Obviously, the first pass that needs to be inserted is one to convert the program into SSA form. The reasoning for this, as we've seen, is that SSA form simplifies some of the other optimizations. After this we perform type analysis that is used by the constant propagation/folding pass and the type specialization/explicate pass. The type specialization pass will be used to simplify code in situations where the types of variables are known at compile time. After this we add a pass to handle Dead Code Elimination and then a pass to convert back out of SSA form.

What's interesting to note is the looping behavior indicated in the diagram. After we eliminate dead code, the structure of the program could potentially have changed and new candidates for optimization may have emerged. Because of this we want to loop back until we detect that our AST has stopped changing and we have reached some stable point where no more optimizing is possible. Unfortunately, when we tried to implement this, we ran into some complications and couldn't get it completely working. Because we still wanted to get some benefit out of doing constant propagation and folding a few times we have that in a loop and leave the amount of times the loop will execute as a constant defined in the compiler

Results

As discussed in this report the primary objectives of our optimizations were to decrease runtime and decrease the amount of assembly code produced by our compiler. We also hoped to reduce compile time but, as previously mentioned, the changes to our compiler might actually increase compile time in some changes.

After completing our compiler we ran four benchmark programs to measure the performance changes induced by our optimizations. We kept track of runtime, compile time, and x86 code size (in number of lines of code). The UNIX "time" command was used in the timing tests and the table below shows the difference between our original compiler and our optimized compiler.

	Arith	**Assign LHS Stack**	**DER**	**Consts All Round**
Runtime	-21%	-7%	-4%	-22%
X86 code size	-50%	-12%	-6%	-53%
Compile time	-25%	-15%	+9%	-24%

These results show pretty much what we expected. Runtime and x86 code size decrease across the board and compile time increases in some cases but decreases in others. What's really exciting to see is the rather large improvements in the arithmetic and constants benchmarks. Because two of our optimizations (constant propagation/folding and type specialization) specifically targeted these benchmarks it makes sense that we see improvement across the board for these benchmarks.

Looking at the DER test it makes sense why our compiler didn't do a great job of optimizing this and why we saw an increase in compile time. The problem is this test makes a lot of function calls which our optimizations struggle with. We can't figure out the types of function parameters so we can't do much in type specialization and the parameters don't have constant values so constant propagation doesn't help us much either. As a result all we really do is introduce compiler overhead.

Another interesting thing to note is that our dead code elimination probably isn't providing as much benefit in these results as it could. The reason is, by definition, dead code elimination removes code that has no effect on the output of the program. This means that the programmer has included code that isn't necessary. What this means is that we could see big gains on programs that were written naively and contain useless code but we will see less improvement on programs that are written intelligently.

Reference

[1] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. Software – Practice and Experience, Vol. 28, No. 8, pp. 859–881, July 1998.

[2] Masataka Sassa, Masaki Kohama and Yo Ito. Comparison and Evaluation of Back Translation Algorithms for Static Single Assignment Form. IPSI 2004.

[3] Mooly Sagiv and Noam Rinetzky. Program Analysis Lecture #4, 15-April-2004: Posets, Lattices and Constant Propagation.

[4] Ron Cytron, JeanneFerrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck - Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.

[5] Craig Chamber, David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In LISP AND SYMBOLIC COMPUTATION, 4, 3, 1991.