ECEN 5523/ CSCI 5525 Final Project

Chris Bubernak
Eric Grover
Sabarigirish Muralidharanpillai
Xinying Zeng

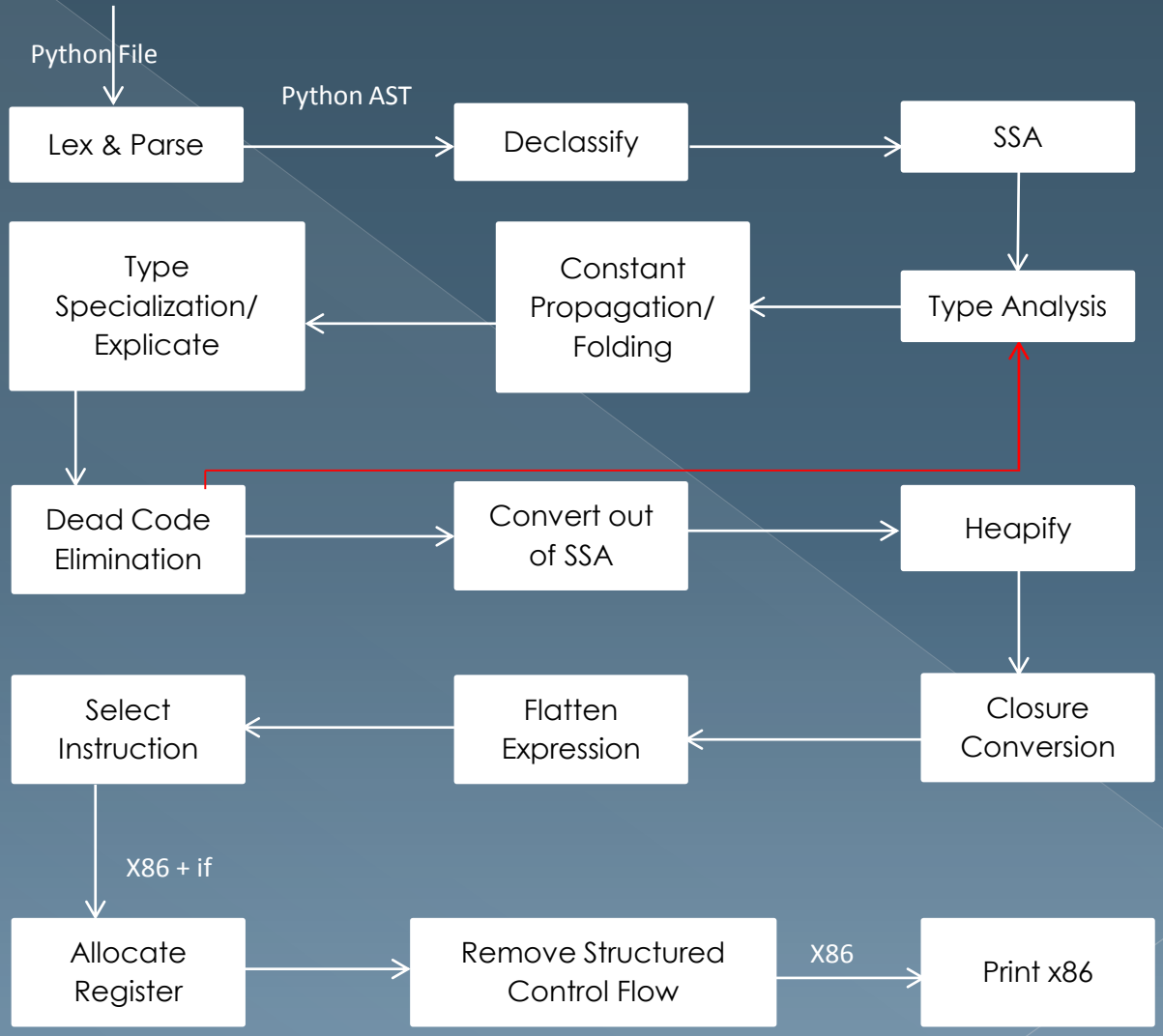
Overview

- Introduction
- Compiler Organization
- Proposed Optimizations
  - Single Static Assignment Form
  - Constant Propagation & Folding
  - Type Analysis & Specialization
  - Dead Code Elimination
- Expected Improvement
- Results

Introduction

- Single Static Assignment
- Constant Propagation & Folding
- Type Analysis & Specialization
- Dead Code Elimination

Compiler Organization



Compiler Organization (contd.)

- We introduce a loop of optimizations.

  After the dead code elimination, compiler reruns type analysis, constant propagation/ folding, type specialization and dead code elimination.

- Why?

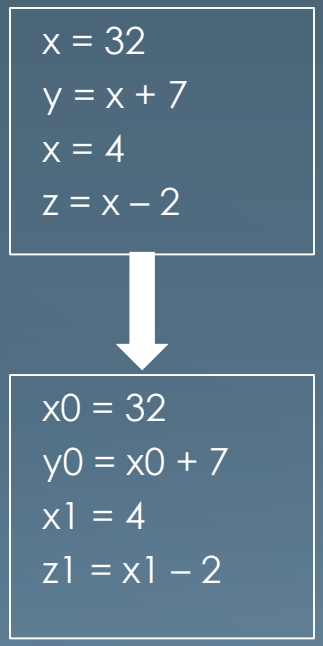
  Example:

```
a_0 = 42                                          a_0 = 42
while (False):
        a_4 = φ(a_0, a_3)
        if (True):
                a_1 = 41.37          ➔
        else:
                a_2 = [1,2,3,4]
        a_3 = φ (a_1, a_2)
print a_4                                         print a_0
```

Single Static Assignment Form

- SSA form dictates that all variables are only assigned to a single time
- General idea of algorithm is create a new version of a variable every time it is assigned to
- Purpose of SSA form is to simplify later optimizations

Convert into SSA



- Φ expression

```
i = 3
m = 0
while (...):
```

→

```
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
```

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

Φ expression return the value of the variable corresponding to the branch that was actually taken at runtime.

Convert out of SSA

- Remove Φ expressions
  - Simple SSA back translation: Pull apart the φ assignments and place parts of them in each of the branches.

SSA

```
x_0 = 1

while(…):
        x_1 = φ(x_0,x_2)
        y_1 = x_1

        x_2 = 2
print y_1
```

Out of SSA

```
x_0 = 1
x_1 = x_0
while(…):
        x_2 = 2
        x_1 = x_2

        y_1 = x_1
print y_1
```

Constant Propagation & Folding

- Constant Propagation is the process of substituting a reference to a constant variable with the variable's value.
- Constant Folding looks for expressions with constant operands and performs the computations.

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
```

Constant Propagation & Folding

- At every statement we classify each variable in one of 3 states:
  - Know constant -> value n
  - potentially taken on two or more variables -> T (top)
  - Value unknown -> ⊥ (bottom).
- At every statement we classify each variable in one of 3 states:
  - Meet function (∩) takes the intersection of two mappings.
  - Join function (U) takes the union of two mappings. The truth table of meet and join functions is as follows.

- Truth table for meet function

∩	T	n∈N	m (!=n)	⊥
T	T	n	m	⊥
n∈N	n	n	⊥	⊥
m (!=n)	m	⊥	m	⊥
⊥	⊥	⊥	⊥	⊥

- Truth table for join function

U	T	n∈N	m (!=n)	⊥
T	T	T	T	T
n∈N	T	n	T	n
m (!=n)	T	T	m	m
⊥	T	n	m	⊥

Constant Propagation & Folding

- Constant propagation & folding algorithm
  - 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.
- SSA simplifies handling the control flow.

  Avoid iteratively traverse the control flow.

Constant Propagation & Folding

- e.g.

Original:

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

SSA form:

```
z_0 = 3
x_0 = 1
while(x_0>0):
        x_1 = φ(x_0, x_3)
        if (x_2=0):
                y_0=7
        else:
                y_1 = z_0+4
        y_2 = φ(y_0, y_1)
        x_3 = 3
        print y_2
```

{x->3, y->7, z->3}

{z_0 -> 3, x_0 ->1, z_2 ->3, x_1 -> Top, y_2->Top}

After dead code elimination, we would know the loop body is executed and the then branch is taken, we could then map x_1 to constant 3 and y_2 to constant 7.

Type Analysis

- Each expression node and variable is annotated with a type
- Depth-first processing of each expression
- Analysis
  - Simple Type Analysis
    - on each leaf node a type will be determined based on static analysis
    - If type cannot be statically determined, uses generic type pyobj
  - Iterative Type Analysis
    - used on loop control-flow nodes
    - iterating over loop to gather all possible types an expression could return
    - if an expression could return more than one type or it cannot be statically determined, then the generic type pyobj is selected

Type Specialization

- Handled in the explicate pass
- If the types of operands in a node are statically known, all type checking and dispatch code can be eliminated
- Example:

```
… elif isinstance(n, Add):
        if(n.left.type == big and n.right.type == big):
            expr = big_processing_expression
        elif(n.left.type == int and n.right.type == int):
            expr = int_processing_expression
        elif(n.left.type == pyobj …)
            expr = type_check_dispatch_expr
        …
        return expr
```

Dead Code Elimination

- Dead Code: Code with no bearing on output of the program

Example:

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

Dead Code Elimination

- Dead Code Elimination - To eliminate all statements that are not marked critical (affecting program output)or definers (modify values used by critical statements)
- Implemented as per algorithm in *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*
- Follows the Static Single Assignment step since SSA makes it easier and more powerful.

Dead Code Elimination

- Implementation in two passes
- First pass : Identify critical statements and mark as live, mark all others as tentatively dead
- Second pass: Identify the definers and mark as live. At the end delete all those which are still marked dead.

Expected Improvement

- Decrease code size
- Decrease runtime
- Decrease compile time (maybe?):
  - The optimizations will reduce the size of the program AST that later compiler passes have to process (reducing compile time)
  - New optimization passes are added and they occur inside a loop (increasing compile time)

Results

- Arith
  - 21% faster runtime
  - 50% less x86
  - 8% faster compile
- Assign LHS Stack
  - 7% faster runtime
  - 12% less x86
  - 15% faster compile
- DER
  - 2% faster runtime
  - 3% less x86
  - 4% slower compile
- Consts All Round
  - 7% faster runtime
  - 25% less x86
  - Same compile

The End

- Questions/Comments?