# CONSTANT FOLDING AND TYPE CHECKING

## OUR COMPILER PROJECT

Created by Evan Roncevich and Irakli Zhuzhunashvili

# THE PROBLEMS

Too many temp variables when adding

Too many conditionals

Slow coloring

Overall, too many lines of code to debug

# CONSTANT FOLDING

## The problem

```
def f():
        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
        return 1 + 2 + 3 + 4 + x
print f()
```

## Lines without folding

```
3231 tests/big_no_fold.s
```

## Lines with folding

```
387 tests/bigfold.s
```

# INTERESTING EXAMPLE 1

```
def f():
        y = 12
        x = 1 + 2 + input() + 12 + y + input() + 5
        return x

print f()
```

Converts into

```
def f():
        y = 12
        x = 20 + input() + y + input()
        return x

print f()
```

# INTERESTING EXAMPLE 2

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

print f()
```

Converts into

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

print f()
```

# IMPLEMENTATION

Recurse untill Add node

If child nodes include Const, add it to a list

If it includes another Add node, recurse on it

For anything else, recurse on it and add to another list

Return the two lists

Constant folding is the first step in the compiler

# CONSTANT PROPAGATION

## (NOT FULLY IMPLEMENTED)

Idea:

```
def f():
        x = 14
        y = 7 - x / 2
        return y * (28 / x + 2)
```

Converts into

```
def f():
        x = 14
        y = 0
        return 0
```

# IMPLEMENTATION

Similar to type checking

Create a dictionary mapping variables to values

Recurse on the AST

At every node, update the dictionary

If variable's value is known, store it, otherwise set it to 'unknown'

Do the folding in second recursion using the dictionary

# TYPE CHECKING

## THE PROBLEM

Too many conditionals

Slow compilation

Hard to debug

# WHAT IT IS

Find the type of variable during compile time

Don't explicate if the type is known

# IMPLEMENTATION

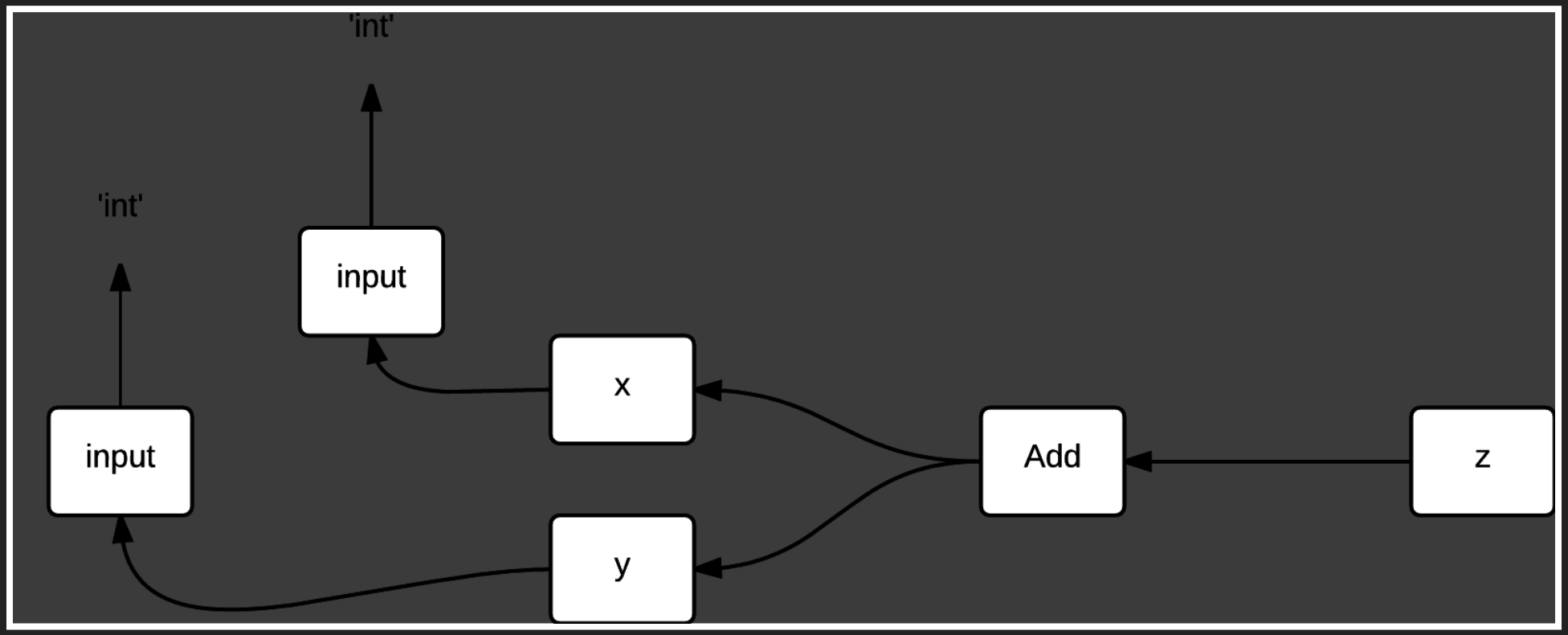
Do extra pass before explicate

At every statement, store known types in a dictionary

For if statements, if then/else types aren't same, set to 'unknown'

Set function args to 'unknown'

At explicate pass, use the dictionary to skip some steps


'int'
'int'
input
input
x
y
Add
z

# WHAT THE SKIPPING STEP LOOKS LIKE

```
#adding l and r:
        if (lType == 'int' or lType =='bool')
        and (rType == 'int' or rType=='bool'):
                return InjectFrom('int', Add(
                (ProjectTo('int',l),ProjectTo('int',r))))
        if lType == "big" and rType == "big":
                return InjectFrom('big',CallFunc(Name("add"),
                [ProjectTo('big',l),ProjectTo('big',r)]))
```

# EXAMPLE

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

The contents of the dictionary at each point

```
Assign([AssName('True a0', 'OP_ASSIGN')], Const(1.25))
--> {'True a0': 'bool', 'fvs': 'unknown'}
Assign([AssName('False a0', 'OP_ASSIGN')], Const(0.25))
--> {'False a0': 'bool', 'fvs': 'unknown', 'True a0': 'bool'}
Assign([AssName('x a0', 'OP_ASSIGN')], CallFunc(Name('input'),
        [], None, None))
--> {'False a0': 'bool', 'True a0': 'bool', 'fvs': 'unknown',
        'x a0': 'int'}
Assign([AssName('y a0', 'OP_ASSIGN')], CallFunc(Name('input'),
        [], None, None))
--> {'False a0': 'bool', 'fvs': 'unknown', 'y a0': 'int', 'True a0':
        'bool', 'x a0': 'int'}
```

# Continued

```
Assign([AssName('z a0', 'OP_ASSIGN')], Add((Name('x a0'),
        UnarySub(Name('y a0')))))
--> {'False a0': 'bool', 'fvs': 'unknown', 'y a0': 'int', 'z a0':
        'int', 'True a0': 'bool', 'x a0': 'int'}
Printnl([Name('z a0')], None)
--> {'False a0': 'bool', 'y a0': 'int', 'z a0': 'int', 'True a0':
        'bool', 'fvs': 'unknown', 'x a0': 'int'}
--> {'False a0': 'bool', 'y a0': 'int', 'z a0': 'int', 'True a0':
        'bool', 'fvs': 'unknown', 'x a0': 'int'}
```

# WHY THIS IS IMPORTANT

Run time of infamous simplex test without our type checking

```
59038 lines, 1min 10 secs compile
```

Run time with our type checking

```
10065 lines, 5 secs compile
```

Certain programs can take forever to run

Enormous condition statements for simple things like addition and comparisons

Easier to run and debug assembly

Removes the overhead of determining obj type at runtime

# FUTURE USAGE AND DEVELOPMENT

Include Constant Propagation

Perform analysis on specific heap-based values for function calling

Perform analysis on Lists

Include type-error checking and warnings at compile-time

Removed unneeded Inject-from Project-to when type is know