Range Analysis

> 编译原理 大作业 简单C++程序的范围分析
>
> 孙培艺 1500012899

1. 实验说明

作业要求：针对静态单赋值（SSA）形式的函数中间代码输入，输出函数返回值的范围。

实现思路： 基本根据 2013年在CGO会议上提出的“三步法”范围分析法加以实现[3]，求得各个变量的范围。

算法优势：空间复杂度和时间复杂度都是$O(n)$，效率高。

算法瓶颈： “三步法”的功能存在较大局限，它只能分析各个变量的最大范围，对活跃变量只做了最简单的考虑，因此最终得到的范围比较不准确，往往只能得到范围的一个界。

2. 项目使用

`python main.py` （ssa文件路径在main.py中设置）

不需要安装任何库。

3. 算法原理

简单概括：采用三步法（2013年在CGO会议上提出）

具体介绍： 我将算法原理和基本步骤详细地整理成了一份PPT，详见`/docs/*.ppt`。在本报告中，我只在2.4部分中简要的概括主要思路。

3.1 构建CFG

代码见：`\src\eSSAConstraintGraph.py; \src\structure.py`

功能：解析SSA，构建CFG。

由于函数之间存在调用关系，因此首先把SSA划分成不同的函数的SSA，再分别构建CFG。CFG中保留了每一个函数的语句、Block之间的关系，为下一步构建Constraint Graph打基础。

CFG的结构如下：

```
#CFG类
class CFG:
    def __init__(self):
        self.name = ''
        self.Blocks = []
        self.Edges = []
        self.Arguments = []
```

3.2 构建Constraint Graph

代码见`\src\eSSAConstraintGraph.py`

三步法的前提是构建Constraint Graph。数据结构如下。在这一步中，我用自己定义的数据类型`MyNode`来表示一条`Constraint`。

```
#Constraint Graph类
class ConstraintGraph:
    def __init__(self, cfg):
        self.MyNodes = []           #基本节点，每一个节点是一个Constraint
        self.MyConditions = []      #用于后面E-SSA Constraint Graph补充条件
        self.cfg = cfg
        self.Arguments = []         #输入参数
        self.returnName = ''        #输出参数
#MyNode : Constraint Graph的节点，也就是保存变量范围的地方
class MyNode:
    def __init__(self, t= "", name = "",  args = [], result = [], fromBlock = 0, Statement =
''):
        self.type = t           #节点类型：leave 叶节点存放范围和值 #op运算符 #var变量名
        self.name = name.strip()  #节点名称：运算名称，或变量名称
        self.args = args    #参数，一个节点是另一个节点的argument，意味着二者之间有边相连
        self.result = result        #被用到哪，一个节点是另一个节点的result，意味着二者之间有边相连
        self.Conditions = []        #约束条件，  在后面E-SSA Constraint Graph中补充条件
        self.fromBlock = fromBlock  #在CFG的哪个Block中定义的
        self.Statement = Statement  #在SSA中的哪条Statement中
        self.Range = Range()        #节点范围
        self.size = ''
        self.input = False
# Range由两个Bound组成
class Range:
    def __init__(self ):
        self.lowBound = Bound()
        self.highBound = Bound()
#Bound由值和类型组成
class Bound:
    def __init__(self):
        self.value = 'None'      # inf 最大值 ；  -inf 最小值; None 未设置;  Not Exists 不存在
        self.size = 'None'       #边界是 int or float
```

需要注意的是，在解决两个函数之间的调用关系时，将被调用的函数**内联进原函数**。我将被调用的函数的所有变量名都加入相应的后缀，比如`foo`调用`bar`函数，那么`bar`中的变量`i_1`将被更名保存为`i_1#bar$1`，其中#是变量原名和后缀分割符，$是函数名和一个随机数的分割符，$的作用是为了区分多次调用同一个函数的情况。

3.3 构建E-SSA Constraint Graph

代码见`\src\eSSAConstraintGraph.py`

这一步用于解决条件的添加。诸如`if (i_2 < j_3)`这样的条件。在MyNode节点类型中，我设置了Conditions结构用于保存条件。Condition的数据结构如下：

```
#Class Description : Constraint Graph中的条件，附加在MyNode中
class MyCondition:
    def __init__(self, condition, index):
        self.condition = condition
        self.arg1 = re.sub("\(.*\)", "",condition.split()[0].strip())
        self.arg2 = re.sub("\(.*\)", "",condition.split()[2].strip())
        self.op = condition.split()[1].strip()
        self.index = index
```

其中，`arg1`和`arg2`分别表示条件的两个参数，`op`表示条件的比较运算符。在`Future Resolution`这一步会进行比较，进行范围的约束。

以`t7.ssa`为例，得到的E-SSA Constraint Graph如下：

```
call bar$1  in 2 : |Arguments: i_2,|Result: |Conditions:
var i_2  in 2 : |Arguments: |Result: bar$1,i#bar$1,i_2#bar$1,|Conditions:
var j_4  in 2 : |Arguments: _1#bar$1,|Result: bar$2,i#bar$2,i_2#bar$2,|Conditions:
ret bar$1  in 2 : |Arguments: |Result: j_4,|Conditions:
call bar$2  in 2 : |Arguments: j_4,|Result: |Conditions:
var k_6  in 2 : |Arguments: _1#bar$2,|Result: _7,|Conditions:
ret bar$2  in 2 : |Arguments: |Result: k_6,|Conditions:
var _7  in 2 : |Arguments: k_6,|Result: |Conditions:
var i_2#bar$1  in 3 : |Arguments: i_2,|Result: +,-,|Conditions: 0#bar$1 0|
leaf 10  in 3 : |Arguments: |Result: +,|Conditions:
op +  in 3 : |Arguments: i_2#bar$1,10,|Result: _3#bar$1,|Conditions: 0#bar$1 0|
var _3#bar$1  in 3 : |Arguments: +,|Result: PHI,|Conditions: 0#bar$1 0|
leaf 5  in 4 : |Arguments: |Result: -,|Conditions:
op -  in 4 : |Arguments: 5,i_2#bar$1,|Result: _4#bar$1,|Conditions: 0#bar$1 1|
var _4#bar$1  in 4 : |Arguments: -,|Result: PHI,|Conditions: 0#bar$1 1|
op PHI  in 4 : |Arguments: _3#bar$1,_4#bar$1,|Result: _1#bar$1,|Conditions: 0#bar$1 1|
var _1#bar$1  in 4 : |Arguments: PHI,|Result: j_4,|Conditions: 0#bar$1 1|
leaf i#bar$1  in  : |Arguments: i_2,|Result: |Conditions:
var i_2#bar$2  in 3 : |Arguments: j_4,|Result: +,-,|Conditions: 0#bar$2 0|
leaf 10  in 3 : |Arguments: |Result: +,|Conditions:
op +  in 3 : |Arguments: i_2#bar$2,10,|Result: _3#bar$2,|Conditions: 0#bar$2 0|
var _3#bar$2  in 3 : |Arguments: +,|Result: PHI,|Conditions: 0#bar$2 0|
leaf 5  in 4 : |Arguments: |Result: -,|Conditions:
op -  in 4 : |Arguments: 5,i_2#bar$2,|Result: _4#bar$2,|Conditions: 0#bar$2 1|
var _4#bar$2  in 4 : |Arguments: -,|Result: PHI,|Conditions: 0#bar$2 1|
op PHI  in 4 : |Arguments: _3#bar$2,_4#bar$2,|Result: _1#bar$2,|Conditions: 0#bar$2 1|
var _1#bar$2  in 4 : |Arguments: PHI,|Result: k_6,|Conditions: 0#bar$2 1|
leaf i#bar$2  in  : |Arguments: j_4,|Result: |Conditions:

Conditions:
i_2(D) >= 0#bar$1 0#bar$1,i_2(D) >= 0#bar$2 0#bar$2,
```

3.4 三步法

3.4.1 Widen

代码见`\src\rangeAnalysis.py`

Widen 步骤用于将 变量范围扩大。此步骤可以在O(n)阶段内完成。基于原理如下：可以形象的理解为：在进行ϕ操作时，如果发现变量范围向上增加，就直接扩大到$\inf$，如果发现变量范围向下减小，就直接减小到$-\inf$。

$$I[Y] = \begin{cases} \text{if } I[Y] = [\bot, \bot] \text{ then } e(Y) \\ \text{elif } e(Y)_\downarrow < I[Y]_\downarrow \text{ and } e(Y)_\uparrow > I[Y]_\uparrow \text{ then } [-\infty, \infty] \\ \text{elif } e(Y)_\downarrow < I[Y]_\downarrow \text{ then } [-\infty, I[Y]_\uparrow] \\ \text{elif } e(Y)_\uparrow > I[Y]_\uparrow \text{ then } [I[Y]_\downarrow, \infty] \end{cases}$$

这样下来后，每一个MyNode的范围都会扩大到最大。

3.4.2 Future Resolution & Narrow

代码见`\src\rangeAnalysis.py`

在Widen步骤中，只能解决每一个变量内部之间的赋值行为，在Future Resolution步骤，可以对变量之间的运算、以及条件进行处理。

我用了复杂的`ConditionHandle()`函数来解决条件变量的Constraint问题。我在每一个MyNode中添加了Conditions结构，用Condition约束来代替变量替换。这样可以大大减少变量替换带来的麻烦。

在`ConditionHandle()`中，我将条件拆分成`arg1` `arg2`和`op`三部分，将他们组合成条件为真的范围，和条件为假的范围。并把相应的范围赋给相应的变量，以及检查此路径是否可以相通。

以`t7.ssa`为例，三步法得到的所有变量的范围如下：

```
Enter Range For i: -10 10
bar$1 None None | Range:  Not Exists Not Exists
i_2 int int | Range:  -10 10
j_4 int int | Range:  0 20
bar$1 None None | Range:  Not Exists Not Exists
bar$2 None None | Range:  Not Exists Not Exists
k_6 int int | Range:  5 30
bar$2 None None | Range:  Not Exists Not Exists
_7 int int | Range:  5 30
i_2#bar$1 int int | Range:  -10 10
10 None None | Range:  10 10
+ int int | Range:  0 20
_3#bar$1 int int | Range:  0 20
5 None None | Range:  5 5
- int int | Range:  Not Exists Not Exists
_4#bar$1 int int | Range:  15 -5
PHI int int | Range:  0 20
_1#bar$1 int int | Range:  0 20
i#bar$1 None None | Range:  Not Exists Not Exists
i_2#bar$2 int int | Range:  0 20
10 None None | Range:  10 10
+ int int | Range:  10 30
_3#bar$2 int int | Range:  10 30
5 None None | Range:  5 5
- int int | Range:  Not Exists Not Exists
```

```
_4#bar$2 int int | Range:  5 -15
PHI int int | Range:  5 30
_1#bar$2 int int | Range:  5 30
i#bar$2 None None | Range:  Not Exists Not Exists
```

可以直接得到结果变量`_7`的范围为：`_7 int int | Range: 5 30`

4. 实验结果

```
#t1.SSA
Reference Range:[100, 100]
Output Range: [100, +inf]
#t2.SSA
Reference Range:[200, 300]
Output Range: [200, +inf]
#t3.SSA
Reference Range:[20, 50]
Output Range: [20, +inf]
#t4.SSA
Reference Range:[0, +inf]
Output Range: [0, +inf]
#t5.SSA
Reference Range:[210, 210]
Output Range: [0, +inf]
#t6.SSA
Reference Range:[-9, 10]
Output Range: [-9, 10]
#t7.SSA
Reference Range:[16, 30]
Output Range: [5, 30]
#t8.SSA
Reference Range:[-3.2192308, 5.94230769]
Output Range: [-0.41923075526423315, 14.700000286102295]
#t9.SSA
Reference Range:[9791, 9791]
Output Range: [-10, +inf]
#t10.SSA
Reference Range:[-10, 40]
Output Range: [1, 1]
```

5. 总结与后记

在本实验中，我采用python语言对SSA形式的C程序进行解析，并采用三步法针对特定输入进行了相应的范围分析。收货了写代码的乐趣，也为最后的效果遗憾。

最后的效果中，10个benchmark的结果中准确结果寥寥无几。尤其是上界，很多都直接到无穷了。这一方面是为了追求时间效率和空间效率，放弃了模拟执行采用三步法的缺陷，另一方面也是因为我没有想到合适的改进方法。

我自知效果平平，亦深为惭愧。希望以后自己更加努力。

6. 参考文献

[1] Aho A V, Ullman J D. Principles of Compiler Design (Addison-Wesley series in computer science and information processing)[M]. Addison-Wesley Longman Publishing Co., Inc., 1977.

[2] Harrison W H. Compiler analysis of the value ranges for variables[J]. IEEE Transactions on software engineering, 1977 (3): 243-250.

[3] Rodrigues R E, Campos V H S, Pereira F M Q. A fast and low-overhead technique to secure programs against integer overflows[C]//Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on. IEEE, 2013: 1-11.