Java中易于溢出的比较(Overflow-prone comparisons in Java)

您可以使用CodeQL在Java代码中检查比较一侧容易溢出的比较。

关于本文（About this article）

在本教程文章中，您将编写一个查询，以查找循环中整数和长整数之间的比较，这可能会导致由于溢出而导致不终止。
首先，请考虑以下代码片段：

```
void foo(long l) {
    for(int i=0; i<l; i++) {
        // do something
    }
}
```

如果l大于2的31次方–1（int类型的最大正值），则此循环将永远不会终止：i将以零开始，一直递增直到2的31次方–1（仍小于l）。 当它再次增加时，会发生算术溢出，并且我变为–2的31次方，也小于l！ 最终，我将再次达到零，并且循环重复。

```
有关溢出的更多信息

所有原始数字类型都有一个最大值，如果超过该最大值，它们将回绕到它们的最低可能值（称为“溢出”）。 对于int，此最大值为2的31次方–1。类型long可以容纳更大的值，最大为2的63次方–1。在此示例中，这意味着l可以采用大于int类型的最大值的值； 我将永远无法达到该值，而是溢出并返回低值。
```

我们将开发一个查询，以查找看起来可能表现出这种行为的代码。 我们将使用几种标准的库类来表示语句和函数。 有关完整列表，请参见“用于Java程序的抽象语法树类”。

初始查询（Initial query）

我们将首先编写一个查询，以查找小于表达式（CodeQL类LTExpr），其中左操作数为int类型，而右操作数为long类型：

```
import java

from LTExpr expr
where expr.getLeftOperand().getType().hasName("int") and
    expr.getRightOperand().getType().hasName("long")
select expr
```

注意，我们使用谓词getType（在Expr的所有子类上都可用）来确定操作数的类型。 类型反过来定义hasName谓词，它使我们能够识别基本类型int和long。 就目前而言，此查询查找所有比较int和long的小于表达式，但实际上，我们只对循环条件中的比较感兴趣。 另外，我们要过滤掉其中两个操作数都恒定的比较，因为这些比较不可能是真正的错误。 修改后的查询如下所示：

```
import java

from LTExpr expr
where expr.getLeftOperand().getType().hasName("int") and
    expr.getRightOperand().getType().hasName("long") and
    exists(LoopStmt l | l.getCondition().getAChildExpr*() = expr)
and
    not expr.getAnOperand().isCompileTimeConstant()
select expr
```

LoopStmt类是所有循环的公共超类，尤其是上面示例中的for循环。 虽然不同类型的循环具有不同的语法，但是它们都有一个循环条件，可以通过谓词getCondition进行访问。 我们使用应用于getAChildExpr谓词的自反传递闭包运算符*来表达将expr嵌套在循环条件内的要求。 特别地，它可以是循环条件本身。 where子句中的最后一个合取词利用了谓词可以返回多个值（它们实际上是关系）的事实。 特别是，getAnOperand可能返回expr的任何一个操作数，因此，如果至少一个操作数是常量，则expr.getAnOperand（）、isCompileTimeConstant（）成立。 否定该条件意味着查询将仅找到两个操作数都不为常量的表达式。

泛化查询（Generalizing the query）

当然，int和long之间的比较不是唯一有问题的情况：窄类型和宽类型之间的任何小于小于的比较都可能被怀疑，小于或等于，大于和大于或大于 等于比较与小于比较一样有问题。
为了比较类型的范围，我们定义一个谓词，该谓词返回给定整数类型的宽度（以位为单位）：

```
int width(PrimitiveType pt) {
    (pt.hasName("byte") and result=8) or
    (pt.hasName("short") and result=16) or
    (pt.hasName("char") and result=16) or
    (pt.hasName("int") and result=32) or
    (pt.hasName("long") and result=64)
}
```

现在，我们希望对查询进行一般化，以将其应用于比较较小端的类型宽度小于较大端的类型的宽度的任何比较。 让我们称这种比较容易发生溢出，并引入一个抽象类对其进行建模：

```
abstract class OverflowProneComparison extends ComparisonExpr {
    Expr getLesserOperand() { none() }
    Expr getGreaterOperand() { none() }
}
```

该类有两个具体的子类：一个用于<=或<比较，而一个用于> =或>比较。 在这两种情况下，我们都以仅与所需表达式匹配的方式实现构造函数：

```
class LTOverflowProneComparison extends OverflowProneComparison {
    LTOverflowProneComparison() {
        (this instanceof LEExpr or this instanceof LTExpr) and
        width(this.getLeftOperand().getType()) < width(this.getRi
ghtOperand().getType())
    }
}

class GTOverflowProneComparison extends OverflowProneComparison {
    GTOverflowProneComparison() {
        (this instanceof GEExpr or this instanceof GTExpr) and
        width(this.getRightOperand().getType()) < width(this.getL
eftOperand().getType())
    }
}
```

现在，我们重写查询以使用以下新类

```
import Java

// Insert the class definitions from above

from OverflowProneComparison expr
where exists(LoopStmt l | l.getCondition().getAChildExpr*() = expr
) and
not expr.getAnOperand().isCompileTimeConstant()
select expr
```