Java 中易溢出的比较

您可以使用 CodeQL 检查 Java 代码中的比较，其中比较的一侧容易溢出。

关于这篇文章

在本教程文章中，您将编写一个查询，用于查找循环中整数和长整数之间的比较，这些循环可能会导致溢出导致不终止。
首先，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 231-1（int 类型的最大正值），那么这个循环将永远不会终止：我将从零开始，一直递增到 231-1，这仍然小于 l。当它再次递增时，发生算术溢出，我变成-231，它也小于 l！最终，我会再次达到零，循环重复。
有关溢出的详细信息
所有基元数值类型都有一个最大值，超过这个值，它们将返回到其可能的最小值（称为“溢出”）。对于 int，这个最大值是 231-1。long 类型可以容纳更大的值，最大值为 263-1。在这个例子中，这意味着 l 可以接受一个大于 int 类型的最大值的值；我永远无法达到这个值，而是溢出并返回到一个较低的值。
我们将开发一个查询，它可以找到看起来可能表现出这种行为的代码。我们将使用几个标准库类来表示语句和函数。有关完整列表，请参阅使用 Java 程序的抽象语法树类。

初始查询

我们将首先编写一个查询，该查询查找小于表达式（CodeQL class LTExpr），其中左操作数的类型为 int，右操作数的类型为 long：

```
import java



from LTExpr expr

where expr.getLeftOperand().getType().hasName("int") and

    expr.getRightOperand().getType().hasName("long")
```

```
select expr
```

➤请在 LGTM.com 网站. 此查询通常查找大多数项目的结果。

注意，我们使用谓词 getType（可用于 Expr 的所有子类）来确定操作数的类型。反过来，类型定义 hasName 谓词，它允许我们标识 int 和 long 基元类型。目前，这个查询会查找比较 int 和 long 的所有小于表达式，但实际上我们只对循环条件中的比较感兴趣。此外，我们还希望过滤掉任何一个操作数都是常量的比较，因为它们不太可能是真正的 bug。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
```

➤请在 LGTM.com 网站. 请注意，找到的结果较少。

类 LoopStmt 是所有循环的一个公共超类，尤其包括上面例子中的 for 循环。虽然不同类型的循环有不同的语法，但它们都有一个循环条件，可以通过谓词 getCondition 访问。我们使用自反传递闭包运算符*来表示 expr 应该嵌套在循环条件中的要求。特别是，它可以是循环条件本身。
where 子句中的最后一个连接词利用了这样一个事实：谓词可以返回多个值（它们实际上是关系）。特别是，getAnOperand 可能返回 expr 的任意一个操作数，因此 expr. getAnOperand 公司（）. isCompileTimeConstant（）在至少一个操作数为常量时保持。否定此条件意味着查询将只查找两个操作数都不是常量的表达式。

泛化查询

当然，int 和 long 之间的比较并不是唯一有问题的情况：窄类型和宽类型之间的任何小于比较都可能是可疑的，小于或等于、大于或大于或等于的比较与小于比较一样有问题。
为了比较类型的范围，我们定义了一个谓词，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，我们希望将查询推广到任何比较中，其中比较小端的类型宽度小于较大一端类型的宽度。让我们称这种比较容易溢出，并引入一个抽象类来建模：

```
abstract class OverflowProneComparison extends ComparisonExpr {

    Expr getLesserOperand() { none() }

    Expr getGreaterOperand() { none() }

}
```

这个类有两个具体的子类：一个用于<=或<比较，另一个用于>=或>比较。在这两种情况下，我们都以这样一种方式实现构造函数：它只匹配我们想要的表达式：

```
class LTOverflowProneComparison extends OverflowProneComparison {

    LTOverflowProneComparison() {

        (this instanceof LEExpr or this instanceof LTExpr) and

        width(this.getLeftOperand().getType()) <
width(this.getRightOperand().getType())

    }

}


class GTOverflowProneComparison extends OverflowProneComparison {

    GTOverflowProneComparison() {

        (this instanceof GEExpr or this instanceof GTExpr) and
```

```
        width(this.getRightOperand().getType()) <
width(this.getLeftOperand().getType())

    }

}
```

现在我们重写查询以使用这些新类：

```
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
```