实验二：基于 JoeQ 的程序优化

李晨昊 2017011466

2019-11-22

目录

1 基本要求 **1**

1.1 冗余 NULL_CHECK 检测 1

1.2 冗余 NULL_CHECK 消除 2

2 额外优化一：拓展冗余 NULL_CHECK 消除 **2**

3 额外优化二：死代码消除 **2**

1 基本要求

1.1 冗余 NULL_CHECK 检测

冗余 NULL_CHECK 检测是基于数据流分析进行的，数据流的具体定义如下：

- 值域：全体变量 (虚拟寄存器)
- 方向：前向
- 传递函数 (单条语句 s 的)：
  - 若本条语句是 NULL_CHECK，则 $f_s(x) = (x - def_s) \bigcup check_s$，其中 $check_s$ 为被检查的变量
  - 否则，$f_s(x) = x - def_s$
- 交汇函数：$\bigcap$
- 边界条件：$out[ENTRY] = \emptyset$
- 初值：$out[b] = U$

在数据流分析的基础上，如果一条 NULL_CHECK 语句的 $check_s$ 在 $in[s]$ 中，则这条语句是冗余的。

我在 NullCheckOpt.java 中实现了对应的数据流。值得注意的是，如同我在上次报告中说明

的一样，框架中已经写好的几个数据流实现都做了太多的保护性拷贝，在我的实现中我把绝大多数拷贝都修改为了直接返回引用，但是这需要上层代码的配合，否则指针 alias 会导致错误的结果，所以**我把 `FlowSolver.java` 替换成了我上次实验中的实现**。

我并不觉得我这样实现有违背接口的意义，因为接口中本来就没有说明 `this` 和返回值间是否可以具有 ownership 的关系，文档没有定义的都应该可以自由实现，至于上层代码能否随意调用接口，这个本来就没有保证。

1.2 冗余 `NULL_CHECK` 消除

数据流分析结束之后，扫描一遍语句，直接删除那些冗余的 `NULL_CHECK` 语句即可。

2 额外优化一：拓展冗余 `NULL_CHECK` 消除

我做了三个拓展：

1. `IFCMP_A` 中，如果一个操作数是 `null` 常量，另一个操作数是寄存器，则跳转的目标基本块或者 fallback 基本块中有一个可以保证该寄存器非空
2. `this` 永远不为 `null`，这是 Java 的语言模型保证的
3. `new` 或者 `new []` 构造出来的对象永远不为 `null`，这是 Java 的语言模型保证的 (分配内存失败时必须抛出 `OutOfMemoryError` 异常，不允许返回 `null`)

后两个实现起来和 `NULL_CHECK` 差不多，都是在执行转移函数的时候识别语句，并且把额外的寄存器添加到函数值中。优化 2 需要记录本函数是否是 `static` 的，如果不是 `static` 的，则把 `R0` 添加到函数值中，它在 JoeQ 中表示 `this`。

第一个需要一定的预处理：先扫描一遍所有的语句，发现满足条件时，获取所需的基本块的第一条语句，在它的 id 处记录下来这个信息，执行转移函数的时候遇到这个 id 时把额外的寄存器添加到函数值中。

3 额外优化二：死代码消除

我在 `LivenessOpt.java` 中做了基于活跃变量分析的死代码消除。

在完成活跃变量分析之后,扫描每条语句,如果它没有副作用 (通过调用 `getOperator().hasSideEffects()` 来判断)，而且它定值的寄存器都不在对应的 *out* 集合中，证明它是死代码，可以消除。

由于采用了活跃变量分析而非 Faintness Analysis，所以无法像 Faintness Analysis 一样一次性消除很多死代码，不过这可以通过多次运行死代码消除来在一定程度上模拟。在 `Optimize.java` 中，我循环调用了这个优化 5 次，这是一个比较随意的数值，没有什么特殊含义。

我在 test 中添加了 LivenessTest.java，写了一个简单的例子，优化效果如下：

```
static int test(int a, int b, int c, int d) {
  int x = a + b - c * d;
  int y = c / d;
  return a + b;
}
```

```
5   ZERO_CHECK_I            T-1 <g>,    R3 int
8   ADD_I                   T4 int,     R0 int, R1 int
9   RETURN_I                T4 int
```

可见，只有与返回值有关的计算，以及有副作用的 ZERO_CHECK_I 被保留下来了。