作业四：数据流分析框架（二）

唐亚周 519021910804

1 项目概述

运行程序需要加上命令行参数，即输入的 IR 文件位置。`./ir/p1.txt` 是第 1 题的示例 IR 代码，`./ir/p2.txt` 我用于测试第 2 题而构造的 IR 代码。

整个项目结构如下，其中 `toyparser` 模块是语法分析模块，`toyanalyzer` 模块是活跃变量分析和未定义变量分析模块。

```
.
├── ir
│   ├── p1.txt
│   └── p2.txt
└── src
    └── me
        └── adswt518
            └── dfa
                ├── Main.java
                ├── toyanalyzer
                │   └── ToyAnalyzer.java
                └── toyparser
                    ├── AssignInst.java
                    ├── BinInst.java
                    ├── Block.java
                    ├── BrInst.java
                    ├── CmpInst.java
                    ├── Function.java
                    ├── Instruction.java
                    ├── JmpInst.java
                    ├── RetInst.java
                    ├── ToyParser.java
                    └── Variable.java
```

2 第一题

> 编写程序对龙书图 9.10 所对应的 Toy IR 代码（见附录）进行活跃变量分析，具体要求为:
> 1、首先对该段代码进行语法分析，构建控制流图。Toy IR 语法较为简单，使用基本的IO、字符串函数即可分析，不需要编写复杂的分析算法。附录里给出了 Toy IR 的EBNF 语法规则作为参考，语法分析程序不需要严格遵照此语法，能支持本次作业即可。
> 2、然后编写 Worklist 算法，对控制流图进行分析，打印输出每个基本块的 def、use、IN、OUT 集合。

2.1 第一问

语法分析器的实现我参考了网上的博客[1]。整体思路如下：

1. 整个 String 是一个 `Function`
2. `Function` 由 3 个部分组成，`returnType` 代表返回值类型，`name` 代表函数名，`blocks` 代表基本块的列表
3. 每个 `Block` 由以下几个部分组成
   (a) `label` 代表该基本块的名字
   (b) `instructions` 代表该基本块中指令的列表
   (c) `prev` 代表该基本块前面的基本块，`next` 代表该基本块后面的基本块
   (d) `IN`，`OUT`，`def`，`use` 是活跃变量分析中需要使用的（**均是 BitSet 的数据类型**）
4. 指令为 6 种（`AssignInst`，`BrInst`，`BinInst`，`CmpInst`，`JmpInst`，`RetInst`），再加上单个的变量（`Variable`），一共 7 种，均继承自 `Instruction` 这个抽象类。
   (a) 所有不在最后一行的都是 `AssignInst`（仅限于完成作业，按照 ToyIR 的语法并非如此）
   (b) 最后一行一定是 `BrInst` 或 `JmpInst` 或 `RetInst`（仅限于完成作业，按照 ToyIR 的语法并非如此）
   (c) `AssignInst` 等号左边的是 `Variable`，右边的是 `BinInst` 或 `CmpInst` 或 `Variable`
   (d) `Variable` 还可以出现在 `BrInst` 的跳转条件，以及 `BinInst` 和 `CmpInst` 中

语法分析结果如下

```
Function{returnType=I32, name='@main', blocks=[
    Block ENTRY
        IN: null
        OUT: {}
        def: null
        use: null
        instructions: null
        prev: []
        next: [],
    Block B1
        IN: {}
        OUT: {}
        def: {}
        use: {}
        instructions: [%a = 1, %b = 2, jmp B2]
        prev: []
        next: [],
    Block B2
        IN: {}
        OUT: {}
        def: {}
        use: {}
        instructions: [%c = add %a %b, %d = sub %c %a, jmp B3]
```

```
        prev: []
        next: [],
    Block B3
        IN: {}
        OUT: {}
        def: {}
        use: {}
        instructions: [%d = add %b %d, br 1 B5 B4]
        prev: []
        next: [],
    Block B4
        IN: {}
        OUT: {}
        def: {}
        use: {}
        instructions: [%d = add %a %b, %e = add %e 1, jmp B3]
        prev: []
        next: [],
    Block B5
        IN: {}
        OUT: {}
        def: {}
        use: {}
        instructions: [%b = add %a %b, %e = sub %c %a, br 0 B6 B2]
        prev: []
        next: [],
    Block B6
        IN: {}
        OUT: {}
        def: {}
        use: {}
        instructions: [%a = mul %b %d, %b = sub %a %d, ret 0]
        prev: []
        next: [],
    Block EXIT
        IN: {}
        OUT: null
        def: null
        use: null
        instructions: null
        prev: []
        next: []]}
```

2.2 第二问

主要函数如下:

```
public void analyze() {
    List<Block> revBlocks = new ArrayList<>(function.getBlocks());
    Collections.reverse(revBlocks); // 活跃变量分析是逆向的
    analyzeNext(function.getBlocks());
    analyzePrev(function.getBlocks());
    analyzeDefUse(revBlocks);
    liveVariableAnalysis(revBlocks);
}
```

1. `analyzeNext` 函数和 `analyzePrev` 函数根据 `BrInst` 和 `JmpInst` 来推断出基本块之间的关系
2. `analyzeDefUse` 函数通过基本块中的指令来逆向推断出每个基本块中的 `def` 和 `use`
3. `liveVariableAnalysis` 函数通过活跃变量分析的公式来计算出每个基本块中的 `IN` 和 `OUT`

程序运行结果如下

```
ENTRY
    IN: null
    OUT: {%e}
    DEF: null
    USE: null
B1
    IN: {%e}
    OUT: {%a %b %e}
    DEF: {%a %b}
    USE: {}
B2
    IN: {%a %b %e}
    OUT: {%a %b %c %d %e}
    DEF: {%c %d}
    USE: {%a %b}
B3
    IN: {%a %b %c %d %e}
    OUT: {%a %b %c %d %e}
    DEF: {}
    USE: {%b %d}
B4
    IN: {%a %b %c %e}
    OUT: {%a %b %c %d %e}
    DEF: {%d}
    USE: {%a %b %e}
B5
    IN: {%a %b %c %d}
    OUT: {%a %b %d %e}
    DEF: {%e}
```

```
    USE: {%a %b %c}
B6
    IN: {%b %d}
    OUT: {}
    DEF: {%a}
    USE: {%b %d}
EXIT
    IN: {}
    OUT: null
    DEF: null
    USE: null
```

3 第二题

> 完成龙书习题 9.4.1，具体要求为：
> 1、编写程序，检查 Toy IR 代码中是否存在使用未定义（初始化）变量的情况。为减少不必要的工作量，请在第一题代码的基础上进行扩展来完成此题，尽可能复用已有代码。
> 2、编写存在变量未定义就使用情况的示例 Toy IR 代码，输入所编写的检查程序，打印输出存在未定义就被使用情况的变量名及使用未定义变量的指令位置。

3.1 第一问

这一题我主要基于第一题第二问来修改，主要代码如下

```java
public void UndefVariableCheck() {
    analyze();
    List<Block> blocks = function.getBlocks();
    BitSet undefV = function.getBlock("ENTRY").getOUT(); // 未定义的变量
    function.getBlock("ENTRY").setUndefVarAfter(undefV);
    boolean changed = true;
    while (changed) {
        changed = false;
        for (Block block : blocks.subList(1, blocks.size() - 1)) {
            BitSet undefVBef = new BitSet();
            for (Block prev : block.getPrev()) {
                undefVBef.or(prev.getUndefVarAfter());
            }
            if (!undefVBef.equals(block.getUndefVarBefore())) {
                changed = true;
            }
            block.setUndefVarBefore(undefVBef);
            BitSet undefVAfter = new BitSet();
            undefVAfter.or(undefVBef);
            undefVAfter.andNot(block.getDef());
            if (!undefVAfter.equals(block.getUndefVarAfter())) {
                changed = true;
            }
            block.setUndefVarAfter(undefVAfter);
        }
```

```
    }
    System.out.println("未定义的变量: " + printBitSet(undefV));

    List<Block> usedBlock = new ArrayList<>(); // 使用了未定义的变量的基本块
    for (Block block : blocks.subList(1, blocks.size() - 1)) {
        if (block.getUndefVarBefore().intersects(block.getUse())) {
            usedBlock.add(block);
        }
    }
    List<Instruction> usedInst = new ArrayList<>();
    for (Block block : usedBlock) {
        for (Instruction instruction : block.getInstructions()) {
            if (getUsedInst(instruction, undefV)) {
                usedInst.add(instruction);
            }
        }
    }

    System.out.println("使用了未定义的变量的指令: " + usedInst);
    printUndefVar(blocks);
}
```

整体思路如下:

1. 首先调用活跃变量分析的 `analyze` 函数。与活跃变量分析不同的是，**每个基本块的 `def` 集合不再是 define before use 的变量集合，而是所有 define 的变量集合**。我用一个布尔值来进行了判断（在 `analyzeDefUseInst` 函数中）

```
if (isLVA) {
    // 只有进行活跃变量分析时才会清除 def
    // 在进行未定义变量检测时，保留前面的 def，告诉后面的 block，该变量在这里定义了
    def.clear(index);
}
```

2. 然后我从 `OUT(ENTRY)` 出发，计算出每个基本块的 `undefVarBefore` 和 `undefVarAfter`，也就是进入/离开该基本块时，未定义变量的集合。当遍历了所有基本块之后该集合不发生变化，即可认为它就是最终结果。
3. 然后对使用未定义变量的指令进行定位。首先计算出使用未定义变量的基本块：所有 `undefVarBefore` 不为空的基本块。然后遍历该基本块中的指令，并递归地寻找是否有使用这些未定义变量。

3.2 第二问

构造示例 Toy IR 代码如下

```
define i32 @main() {
B1:
    %a = %f
```

```
    %b = 2
    jmp B2
B2:
    %c = add %a %b
    %d = sub %c %a
    jmp B3
B3:
    %f = add %d %c
    %d = add %b %d
    br %g B5 B4
B4:
    %d = add %a %b
    %e = add %e 1
    jmp B3
B5:
    %b = add %f %b
    %e = sub %c %a
    br 0 B6 B2
B6:
    %a = mul %b %d
    %b = sub %a %d
    ret 0
}
```

输出如下:

```
未定义的变量：{%f %g %e}
使用了未定义的变量的指令：[%a = %f, br %g B5 B4, %e = add %e 1]
ENTRY
    undefVarBefore: null
    undefVarAfter: {%f %g %e}
    DEF: null
    USE: null
B1
    undefVarBefore: {%f %g %e}
    undefVarAfter: {%f %g %e}
    DEF: {%a %b}
    USE: {%f}
B2
    undefVarBefore: {%f %g %e}
    undefVarAfter: {%f %g %e}
    DEF: {%c %d}
    USE: {%a %b}
B3
    undefVarBefore: {%f %g %e}
    undefVarAfter: {%g %e}
    DEF: {%f %d}
```

```
    USE: {%b %c %d %g}
B4
    undefVarBefore: {%g %e}
    undefVarAfter: {%g}
    DEF: {%d %e}
    USE: {%a %b %e}
B5
    undefVarBefore: {%g %e}
    undefVarAfter: {%g}
    DEF: {%b %e}
    USE: {%a %f %b %c}
B6
    undefVarBefore: {%g}
    undefVarAfter: {%g}
    DEF: {%a %b}
    USE: {%b %d}
EXIT
    undefVarBefore: {}
    undefVarAfter: null
    DEF: null
    USE: null
```

4 致谢

感谢我的同学秦健行在本次作业中给我的帮助！在他的帮助下，我成功解决了 Java 语法的一些问题以及第二题的思路问题。

1. https://nettee.github.io/posts/2019/How-to-write-a-simple-parser-in-hand/ ↩