Decaf PA5 实验说明

任务描述

PA1 到 PA3 完成了 Decaf 编译器的前端, 而 PA4 包含了处于中后端的数据流分析. 数据流分析的结果是, 我们知道了每个基本块, 以及每个 TAC 的 liveOut, liveIn, liveUse, def 集合. 之后在具体的后端中, 我们会根据中端得到的信息, 将 TAC 一步一步变成底层代码. 在具体的后端中我们会将 TAC 中的指令替换成目标 ISA 的具体指令, 并且将 TAC 中的中间变量变成寄存器或立即数等.

在 PA5 中, 我们将完成 Decaf 编译器的寄存器分配功能. 寄存器分配的目的是, 将 TAC 中的临时变量无冲突地, 高效地映射到物理寄存器. 我们给出了一个贪心分配寄存器的方法, 它是 PA4 的后端采用的方案. 同学们需要实现的是基于图着色的寄存器分配方法, 考虑到完整的图着色算法比较复杂, 同学们只需要实现没有 **spill** 的简化版本中的干涉图构建. 当然若你能实现完整的基于干涉图染色的寄存器分配算法, 那是再好不过的. 第十二讲课堂讲稿给出了图着色寄存器分配的算法, 同学们也可自行参考龙书和虎书. 在框架中已经有一个简单的 MIPS 后端, 它会和寄存器分配一起, 将 TAC 变成最终的 MIPS 汇编.

测试的方法和 PA4 相同, 编译之后在 TestCases/S4 中运行 runAll.py. 它会使用我们的后端将 decaf 代码编译成 MIPS 代码, 然后由 SPIM 程序执行. PA5 要求同学们实现的寄存器分配能够正确工作, 生成的代码能够被 SPIM 执行. 简化的算法面对复杂的输入可能失败, 但你的实现至少要保证 TestCases/S4 中所有测试用例通过. **PA5** 和 **PA4** 一样**,** 没有包含本学期新增语言的内容**,** 故请不要将前面阶段的工作移植过来**.**

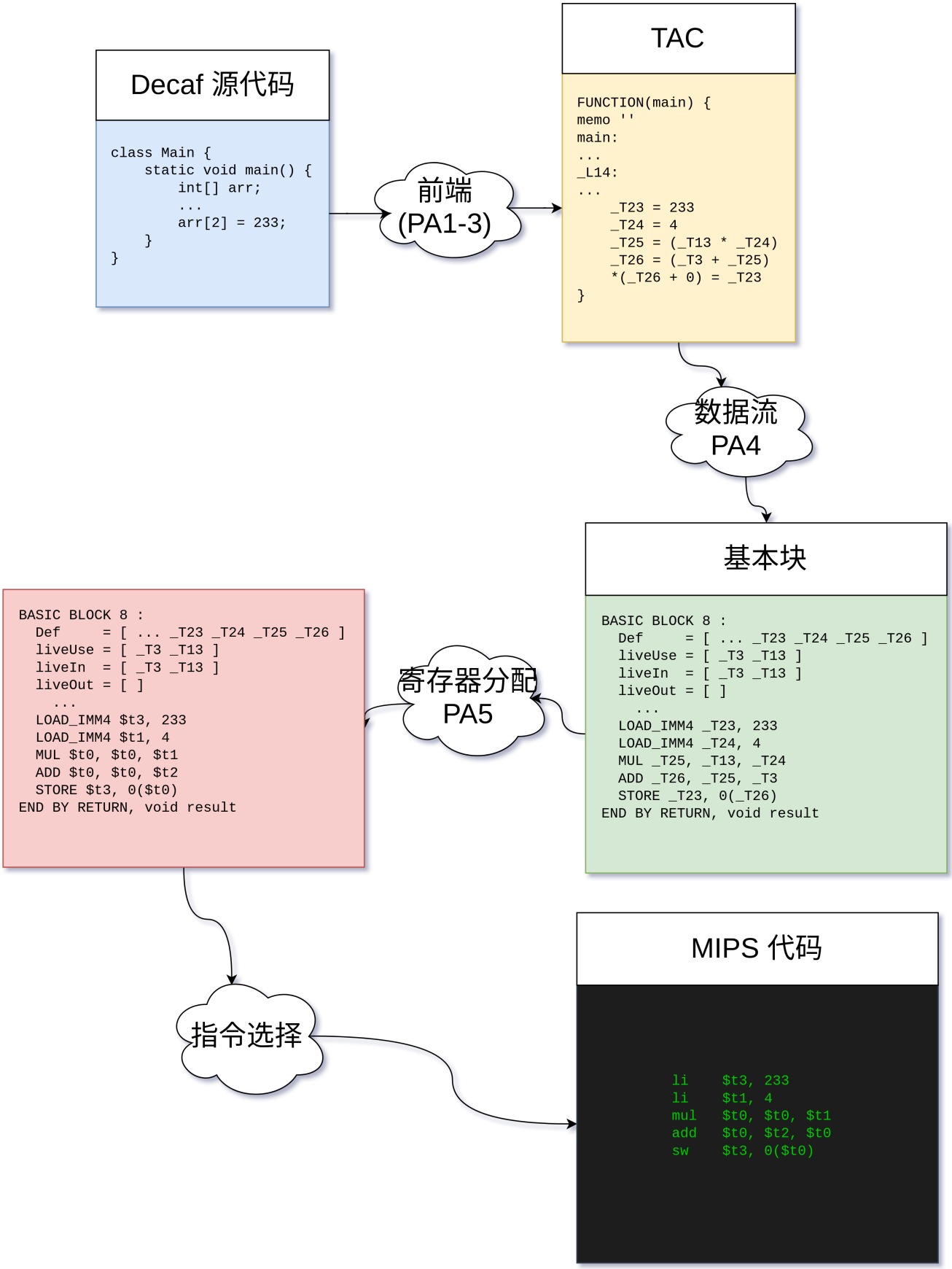
PA5 是新加入的实验, 如果同学们有任何意见或者发现任何错误, 助教欢迎你们随时联系! *#你的意见对我们很重要*

本阶段涉及到的类和工具说明

文件/类	含义	说明
backend/BruteRegisterAllocator.java	一个简单的贪心寄存器分配算法	不需要修改, 可以参考
backend/GraphColorRegisterAllocator.java	基于干涉图染色的寄存器分配算法	需要修改
backend/InferenceGraph.java	干涉图上的染色和寄存器分配	需要实现干涉图的构建
backend/*	后端的目标无关代码	不需要修改
backend/Mips/*	Mips 后端, 包含和架构相关的部分	Mips.java 中你可选择使用哪个 RegisterAllocator
dataflow/*	控制流图部分	不需要修改
machdesc/*	目标 ISA 描述部分	不要修改
tac/*	TAC 语句	不需要修改
translate/*	从 AST 翻译到 TAC 的部分	不要修改
frontend/*	编译器前端, 构建 AST 的部分	不要修改
typecheck/*	语义监察部分	不要修改
scope/*	作用域定义	不要修改
symbol/*	符号定义	不要修改
type/*	类型定义	不要修改
tree/*	AST 的结点定义	不要修改
error/*	错误异常类	不需要修改
Driver	Decaf 编译器入口	不需要修改
Option	编译器选项	不需要修改
utils	辅助工具	不需要修改
build.xml	Ant build file	不要修改

注: 上面 不需要修改 以及 可以修改 的类是指不修改也可完成本次实验, 但是你如果认为确实有必要, 也可以适当修改; 标明 不要修改 的类请不要动, 不恰当的修改会影响得分。

在 Decaf 后端中, 以 Mips 为例, 代码生成的入口在 `MachineDescription` 类的 `emitAsm(List<FlowGraph> gs)` 中. `gs` 中每个 `FlowGraph` 都表示一个函数, 其中包含若干的基本块. 现在的 Decaf 后端实现非常简单, 流程如下图.



编译器后端

编译器后端完成将中间表示, 如 TAC 转换成最终汇编代码的工作. 下面以 MIPS 后端为例说明.

后端翻译的过程中, 代码变得越来越底层 (lowering). 但高层代码 (TAC) 中有一些元素在底层语言 (汇编) 中没有对应, 如

- (*操作数 operand 不对应*) 某些中间表示 (如 LLVM IR) 中, 可能有 byte, bool 等类型, 但汇编中寄存器都是 32/64 位的
- (*操作数不对应*) TAC 中操作数都是 Temp, 但是 MIPS 指令的操作数只能是寄存器和立即数
- (*操作 operation 不对应*) TAC 中只有一个 `add`, 但是 MIPS 有 `ADD`, `ADDU`, `ADDIU`
- (*操作不对应*) TAC 中有 `div` 和 `mod`, 但是 x86 的 DIV 和 IDIV 指令同时给出商和余数
- (*操作不对应*) TAC 中同时有 `leq` (对应 MIPS 的 `SLT`) 和 `beqz`, 但是某些 ISA 没有将比较结果直接保存到 GPR 的指令
- (*概念 construct 不对应*) TAC 中有函数的概念, 但实际汇编中还需要包含进入离开函数的栈帧调整代码
- (*概念不对应*) TAC 中有 `[in]direct_call`, 但实际汇编中除了执行函数跳转, 还需要保存 caller-save 寄存器等.

所以后端需要做的事情, 有如下几个

1. 将操作数变得底层:

- 操作数类型: 变成目标原生支持的类型, 如 i1 提升成 i32, i64 分成两个 i32
  Decaf 框架中这不是一个问题, 因为 TAC 只有两种操作数: Temp 和 Label.
  (实际上还有 vtable 和字符串常量, 但是它们都用标号 (label) 表示了)
  而 Temp 要么是临时变量, 要么是 4 字节立即数.
- 操作数位置: TAC 中操作数是 Temp, 可能是临时变量和 4 字节立即数.
  我们需要决定每个 Temp 被放到那里, 是寄存器还是内存? 是哪个寄存器?
  这是同学们的工作

2. 将操作变得底层:

将 TAC 中的指令变换成 MIPS 的指令. 这对应编译课程上 指令选择 的概念. 在很多框架如 LLVM 中, 指令选择使用的是 tree rewriting 算法 (LLVM 使用类似的 DAG rewriting), 并且在寄存器分配之前进行. 基本思想是将 TAC 指令变成表达式树, 将树分成很多块子树, 每个子树对应一条指令. tree rewriting 和我们实验相关性不大, 所以略过对这个算法的详细介绍, 有兴趣的同学可以参考龙书或虎书.

我们 Decaf 后端采用的是一种类似 macro expansion [1] 的方法, 其实可以说是暴力算法, 把每条 TAC 指令用一系列 MIPS 指令机械地替换. 举几个例子, 它简单地将 TAC 的加翻译成 MIPS 的 `addu` 指令; 遇到立即数的话, 在使用立即数的指令前, 加入将立即数装载入寄存器的指令; 它将 TAC 中的 `CALL` 翻译成一场串指令. 这个算法简单, 但是可想而知优化不会很好.

3. 将概念变得底层:

TAC 中有函数, 字符串常量, vtable 等高级概念, 但汇编中只有标号等底层概念. 所以我们需要将这些高层概念投射到底层概念上.

字符串常量和 vtable 相对简单, 可以直接被标号代替.

但函数更加复杂, 因为涉及到

- callee 调用函数之前, 需要将参数保存到正确的位置, 保存 caller-save 寄存器.

- callee 最开始, 需要执行一段 prologue 汇编, 其中调整 sp/fp 寄存器以分配足够的栈帧, 保存 callee-save 寄存器等工作, 之后再进入函数体.
  这一步和前一步被一起称为 calling sequence.
- callee 返回之前, 需要执行一段 epilogue 汇编, 其中恢复 sp/fp 寄存器以销毁栈帧, 恢复 callee-save 等工作, 之后再返回.
- caller 在 callee 返回之后, 取得返回值. 如果返回值在寄存器 v0 中, caller 可能希望把返回值移动到其他寄存器以便使用 v0,
  如果返回值在栈上, caller 可能需要将其加载到寄存器中.

不过在我们的后端中, TAC 的 `CALL` 对应到汇编使用的是一种很暴力的方式, 我们根本没有考虑这个问题的余地.

基于干涉图染色的寄存器分配

在我们的框架中, 寄存器分配出现在数据流分析和指令选择之间, 针对每个基本块进行寄存器分配. 即, 我们完成的是基本块内的局部寄存器分配 (local register allocation) 而非全局寄存器分配 (global register allocation). 在基本块的入口, 我们假设寄存器中不包含任何信息. 所以需要将所有的 liveUse 值加载到寄存器中. 同样, 在基本块的出口, 假设寄存器不能保留任何信息, 所以将所有的 liveOut 值保存到栈上.

这是一种简化的模型, 事实上更优化的应当是全局寄存器分配, 即一个函数内跨基本块的寄存器分配. 我们考虑的简化模型相对简单的算法并不会有太大提升, 甚至它可能连简单的贪心都比不过. 这里我们只是希望同学们借此熟悉干涉图的含义, 以及其上寄存器分配的实现. 为了方便, 我们之后叙述的是全局的寄存器分配**,** 但是同学们只需要实现基本块内的局部寄存器分配**.** 当然如果你实现了全局寄存器分配, 那是最好的.

前置知识: 数据流分析, 你应当懂得 liveUse, liveIn 等集合的含义, 并且理解下面方程的含义

$$
\begin{aligned}
liveIn(bb) &= liveUse(bb) \cup (liveOut(bb) - def(bb)) \\
liveOut(bb) &= \cup_{bb' \text{ is successor of } bb} liveIn(bb')
\end{aligned}
$$

并且你能将这个方程拓展到 tac 上, 计算出 $def(tac), liveOut(tac)$ 等.

基于干涉图染色寄存器分配的基本步骤是

1. 从数据流分析的结果, 构建函数的干涉图
2. 对干涉图染色
3. 根据染色结果, 确定 Temp 分配到哪个寄存器, 还是 spill 到内存

干涉图是一个无向图 G, 结点是 Temp 变量 (一种理解是将 Temp 认为是无穷多的虚拟寄存器), 结点 u, v 之间有边说明 Temp u 和 Temp v 不能被分配到同一寄存器中. 而连边的准则, 龙书中是

> [f]or each procedure a register-interference graph is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined.

到 PA5 中, 需要将 procedure 改为 basic block, 并且你需要把所谓的 "is live" 和 "defined" 自己定义清楚.

如果一种颜色代表一个寄存器, 那么对于有 K 个通用寄存器的 ISA 来说, 这个图能被 K 着色就说明其中所有 Temp 可以被分配到物理寄存器而不用 spill. 而如果不能 K 着色, 说明其中有些 Temp 需要被 spill 到内存, 删除这些 Temp 对应的结点, 直到删除结点后的新图可以被 K 着色.

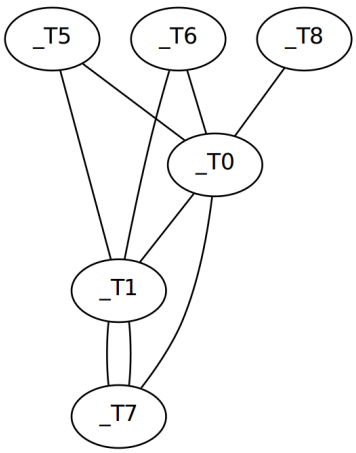
下面给出一个例子. 辗转相除的 decaf 代码是

```
    ...
    static int gcd(int a, int b) {
        while (a != 0) {
            int c;
            c = b % a;
            b = a;
            a = c;
        }
        return b;
    }
    ...
```

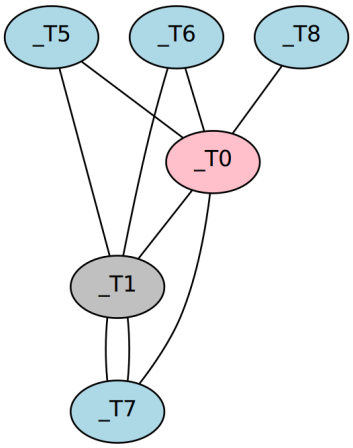
经过数据流分析后, 变成

```
// (liveIn  = [ _T0 _T1 ])

BASIC BLOCK 0 :
    _T5 = 0                                  [ _T0 _T1 _T5 ]
    _T6 = (_T0 != _T5)                       [ _T0 _T1 _T6 ]
    END BY BEQZ, if _T6 = 0 : goto 2; 1 : goto 1 [ _T0 _T1 ]


BASIC BLOCK 1 :
    _T8 = (_T1 % _T0)   [ _T0 _T8 ]
    _T7 = _T8           [ _T0 _T7 ]
    _T1 = _T0           [ _T1 _T7 ]
    _T0 = _T7           [ _T0 _T1 ]
    END BY BRANCH, goto 0 [ _T0 _T1 ]


BASIC BLOCK 2 :
  END BY RETURN, result = _T1 [ _T0 _T1 ]
```

构建干涉图如下



一个染色方案如



淡蓝色对应 t0, 粉色对应 t1, 灰色对应 t2, 则完成寄存器分配如

```
// (liveIn  = [ $t1 $t2 ])

BASIC BLOCK 0 :
    $t0 = 0
    $t0 = ($t1 != $t0)
    END BY BEQZ, if $t0 = 0 : goto 2; 1 : goto 1


BASIC BLOCK 1 :
    $t0 = ($t2 % $t1)
    $t0 = $t0
    $t2 = $t1
    $t1 = $t0
    END BY BRANCH, goto 0


BASIC BLOCK 2 :
  END BY RETURN, result = $t2
```

容易看出, 这个寄存器分配是正确的.

源代码结构

`backend/` 目录下包含了 Decaf 的后端文件, 其中 `backend/mips` 下是针对 MIPS 后端的相关文件, 而 `backend/*.java` 是目标无关的相关文件.

本次实验中, 你需要修改的文件有

- `backend/InferenceGraph.java`:
  你需要理解干涉图的含义, 确定什么时候两个结点应当连边,
  之后实现 `InferenceGraph.makeEdges()`, 其中调用 `addEdge()`.
  建图的发现所有节点过程已经帮你写好了, 在 `makeNodes()` 中.
- `backend/GraphColorRegisterAllocator.java`:
  在 `alloc()` 中调用 `InferenceGraph` 的 `alloc` 函数.
  除此之外, 按照我们的约定, 你还需要在基本块开始处将所有的 liveUse 加载到继承器中.

实验评分

1. 你提交的代码应当能够通过 S4/ 中所有测例. 由于 PA5 只要求实现简化的算法, 因此算法表现不会有明显提升, 所以我们还会检查你的代码是否正确实现了要求. 这部分占 80%.
2. 实验报告占 20%, 其中你应当包含
    - 基本块中两个结点连边的条件, 使用 liveOut, liveUse, liveIn, def 等叙述.
    - PA5 只要求了简化版本的干涉图染色寄存器分配, 试叙述在现有框架中如何实现完整的干涉图染色的寄存器分配算法.
      说明需要修改那些部分, 并且使用伪代码描述关键代码.

参考文献

[1] Gabriel Hjort Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer International Publishing, 2016.