编译原理

MiniDecaf 编译器实验报告 -- STAGE 4

2021010706 岳章乔

一、 思考题

step 9:

1.

```
int foo(int x, int y) {
    return x + y;
}
int main() {
    int x = 3;
    int y = 2;
    return foo(x = y, y = x);
}
```

这样其返回值既有可能是 4， 也有可能是 6 。至于实际上是哪种情况，取决于编译器的实现。

2.

如果只有 **被调用者** 保存的寄存器，那么 **被调用者** 每次执行前都需要记录寄存器内容，然后开始执行函数指令，返回前也需要把寄存器恢复，这会制约 **被调用** 函数的执行效率，尤其是叶节点（不含调用的函数）的执行效率；

如果只有 **调用者** 保存的寄存器，那么 **调用者** 在调用别的函数前，要把所有的寄存器保存在栈帧里；返回后（那就是 `call <foo>` 后）把之前用过的寄存器恢复，这会制约 **调用者** 的执行效率。

综上所述，寄存器的分配和使用，除了要考虑语义规范、以中间代码规范的控制流图以外，还要考虑调用规范，应尽量把函数调用后活跃变量放在被调用者保存寄存器；把调用后不活跃的放在调用者保存寄存器 -- 最小化 prologue 和 epilogue 段。

至于 返回地址 寄存器，调用者需要把 call 的下一条指令的地址告知被调用者，因此这就跟函数传参一样，如果没有在调用前记录返回地址寄存器的值，在函数调用以后，那个寄存器就可能被修改，不指向调用者的返回地址，例如指向该函数的调用后指令的地址。

按 RISC-V ISA 的规范，

`ret` ⇔ `jr x1` ⇔ `jalr x0, x1, 0`

`call foo` ⇔

```
auipc x1, foo[31:12]
jalr  x1, x1, foo[11:0]
```

例如

```
foo:
...
2004 ret
...
1008 call foo
1012 ret
...
```

这种情况下，对于第 2004 行而言，如果是从 1008 行调用 foo，那里的 `ra` 就会记录 1012， 跳转回 1012 行的时候，`ra` 仍然是 1012， 那就导致死循环。

因此，在调用函数前，除了要把前 8 个参数的寄存器原有的值保存，然后把调用值转移，对返回地址也应该同样如此 -- 转移 call 之后的地址。

step 10:

1.

这个取决于在编译的时候有没有指定生成位置无关目标代码。

一般情况下，生成位置有关码。

先规定符号 `a` 在某一条指令处的 delta 值为

$$\mathrm{delta} = \&\mathrm{a} - \mathrm{pc}$$

pc 是下一条指令的地址。

由于 记录 a 的区段的地址，跟其下一条指令地址之差是固定的，因此 delta 也是固定的。

那么直观来说，如果 `aipc rd, imm` 代表的是把立即数 `imm` 和 `pc` 相加，把结果保存在寄存器 `rd` 里。

但是，由于 RISC-V 的微架构设计的限制， RISC-V 指令都是 32 位长，因此如果直接提取 32 位立即数，就没有地方存放指令代号和目标寄存器等信息了。

RISC-V 架构以七位表示指令格式，有时也表示指令代号，五位表示寄存器（共有 31 个通用寄存器，加上 “零” “寄存器” 共 32 个），那么剩下的 20 位就可以表示立即数了。

对于以低七位不能表示指令代号的情况， RISC-V 会指定另外三位 (funct3) 将之表示，例如表示

```
addi rd, rs1, imm
```

的时候，指令类型用了七位，两个寄存器共用了十位， funct3 用了三位，剩下的十二位表示立即数。

综上所述， 可以在`auipc` 和 `addi` 指令的共同作用下 ，把 32位地址加载到某个寄存器里。

auipc 的 u 表示立即数的高二十位，auipc 就是把这高二十位和 pc 的高二十位相加，然后把结果记录在 rd 里。

综上所述，在这种情况下，

```
la rd, a
```

等价于

```
auipc rd, (delta[31:12] + delta[11])
addi  rd, rd, delta[11:0]
```

相应的有

```
<delta[31:12]> < rd 的代号 > 0010111
<delta[11: 0]> < rd 的代号 > 000 < rd 的代号 > 0010011
```

由于这种情况下，无论是 delta 还是 pc 都是确定的，因此也可以直接用 `lui` 表示符号地址的高 20 位，把 auipc 和 delta 分别换成 lui 和 a 的地址即可。

如果在编译的时候指定 `-fPIC` ，那么生成的低级语言关于寻址的部分是位置无关的。**一般用来生成共享库，在执行期间动态链接。**

如果符号 `a` 在共享库，那么 `a` 的寻址方式就会有不同。

同样也可以定义 delta，为

```
delta = GOT[a] - pc
```

这里的 GOT 是所谓的 Global Offset Table，记录 `a` 在虚拟内存的地址。

类似的，这样的 delta 也是固定的。

如果还是用上述的 `auipc+addi` 组合，得到的将是 `a` 的 GOT 的地址，该地址记录的才是 `a` 的真正的地址，因此对于这种情况，应该生成

```
auipc rd, (delta[31:12] + delta[11])
lw  rd, rd, delta[11:0]
```

`lw` 与 `addi` 的区别在于， `addi` 是把 `rs1` 和 `imm` 相加后直接记录在 `rd` ，而 `lw` 是把 `rs1` 加上 `imm` ，然后把那个值在内存对应的地方取值，再把它存在 `rd` 。

lw 指令也可以写成

```
lw rd, delta[11:0](rd)
```

那么对应的二进制指令就是

```
<delta[31:12]> < rd 的代号 > 0010111
<delta[11: 0]> < rd 的代号 > 010 < rd 的代号 > 0000011
```

二、实验内容

2.0. 约定

只有声明，没有定义函数体的函数声明语句 叫 弱定义。

既有声明，也有定义函数体的叫强定义。

符号有强定义，指的是遍历函数定义节点的时候，已经有其他函数定义节点，其代码位置在当下节点之前，而且有定义函数体。

2.1. 实验需求

1. 实现函数声明、定义和调用。
2. 实现全局变量的定义和读写。

2.2. 具体实现

2.2.1. 词法分析

补全 `frontend/scanner.l` 的单词表。

step-9:

仅需要识别逗号，作为参数声明和取用的分隔符。

step-10:

无。

2.2.2. 语法分析

step-9:

原有的 非终结符 `FormalList` 展开空串。现在需要其识别形式参数列表，在原有的基础上，非终结符也可以展开成 形式如下 的符号：

```
(Type IDENTIFIER COMMA) * Type IDENTIFIER
```

这里用 `FormalListPrefix` 表示前面的 `(Type IDENTIFIER ,)*` 。

至于函数调用，由于函数有返回值（ `void` 的函数返回值是未定义，而且可以在类型检查阶段排除），令 `Expr` 展开成函数调用，就在其尾部加上下列产生式

```
            | IDENTIFIER LPAREN ExprList RPAREN
                { $$ = new ast::FuncRef($1, $3, POS(@1)); }
```

这里 `FuncRef` 表示函数调用节点，记录函数名和函数实参。

这里的 `ExprList` 类似于 `FormalList` 的非终结符，记录表达式列表，类似也定义了 `ExprListPrefix` ，而其后缀则只要把 Type IDENTIFIER 替换成 expr 即可。

由于新定义了 `ast_func_call.cpp` 表示函数调用，因此需要在 `makefile` 上增加 `ast_func_call.o` 以便最终链接。

step-10:

`FoDList` 现在除了可以展开为 函数声明、定义列表，也可以展开为全局变量定义 -- 新增

`FoDList --> FODList VarDecl | VarDecl` 产生式即可。

2.2.3. 语义分析

构造符号表：

修改 `translation/build_sym.cpp`。

step-9:

需要处理函数声明、函数定义。

在读取某个函数声明节点 (`FuncDefn`)，如果符号表不存在当下节点，则按正常情况处理。

如果存在，则对定义的强弱分类讨论：

对于弱定义，如果其与之前定义的返回类型相异，或形式参数列表类型不完全一致，则报错，叫函数签名 (signature) 检查。如果检查通过，则处理完毕。

对于强定义，如果签名检查通过，则判断该符号是否存在强定义。

如果有强定义，则报错。

为此，需要修改函数符号的定义，在 `symb/symbol.hpp` 和 `symb/function.cpp` 两处，增加 `bool hasDefined` 和 `void Define(void);` 以及 `bool hasDefinition() const;` 表示该符号当下是否存在强定义。

如果没有强定义，则把那个符号指向的函数内符号表里面记录的形式参数列表重命名，把形式参数的名称，更新为当下形式参数的名称。

为此，需要在 `symb/variable.cpp` 增加函数 `rename(std::string new_name, int order)` 将之修改。

接下来把函数域内的变量声明。

最后对于强定义，无论之前有没有定义过，都一律调用 上面提过的 `Define` ，将符号标记为有强定义。

step-10：

只需要处理定义的初始值。如果有，而且是常数，则用 `var->setGlobalInit` 将之记录，如果有，但不是常数则报错。

类型检查：

step-9:

对上文提到的 `FuncRef` 检查即可。

先确保符号表上存在，而且为函数符号。

接着检查实参列表长度与类型和形式参数列表是否一致，如果一致则把返回类型指定为函数的返回类型；把函数符号挂在节点上。

step-10:

无。

2.2.4. 中间代码生成

这里需要考虑目标代码的生成情况。

step-9:

分函数定义和函数调用两部分讨论。

对于函数定义，只需要翻译强定义，不需要翻译弱定义。

对于各个形式参数，需要从被调用者取值，因此定义三地址码节点：

```
Tac::FETCHARG
```

用 mark 来表示形式参数的下标，第一个为零，第二个为一，以此类推。

用 `op0.var` 表示形式参数的临时变量，临时变量的生成已在本来的实现提供。

现在需要对每个形式参数，生成取参的三地址码，这就要对 `TransHelper` 扩展，加入对生成这样的三地址码的函数，因此定义 `genFetchArg` ，生成该三地址码同时接到当前三地址码链表的尾部。

对于函数调用，考虑到对于记录在内存的参数的取参方式是取被调用者栈底指针下方的值，而且是越往后的参数，放的就越接近底部，因此实参需要反过来压栈。因此生成保存实参指令的时候，应当对实参列表反向遍历，反向生成记录实参的指令。

一方面，需要用 `rbegin` ，用反向迭代器进行遍历，另一方面，需要定义三地址节点，表示记录实参和函数调用，分别叫 `Tac::SAVEARG` 和 `Tac::CALL` 。

这里用 `op0.var` 记录待保存参数的临时变量，mark 表示下标； 用 `op0.var` 表示函数调用的返回值，`op1.label` 表示函数的入口标记，同样也构造这两个节点的生成函数。

与此同时，对上述三个提过的三地址节点，都构造其输出部分，以便后续的调试。

step-10 :

分定义，读和写三个部分。

对于定义， 参照函数定义，每一条函数都有其对应的 Piece，因此也可以构造全局变量的 Piece，记录其初始值，处理手法类似于函数。函数的 Piece 是 Functy，其挂接三地址码链表。这里用 `Globl` 代表全局变量，用 `PayLoad` 表示全局变量的初始化情况。

用 WORD 表示全局变量的初始值，为方便起见，统一用 `.word` 表示全局变量初始值。

用 PADDING 表示全局变量大小，初始化为零，对应 `.zero` 。

对读和写分别修改对 `LvalueExpr` 和 `AssignExpr` 的部分，类似于 step-9 提到的方法构造 `LoadGSym` 和 `SaveGSym` 的三地址节点 (这里在结合 stage 5 后其实应该分为取全局变量地址，然后对那个地址进行读写，但在实现的时候没有考虑后续阶段，因此这里直接代表全局变量的读写) 。

2.2.5. 中间代码优化

编译器对整个数据流图和每一个基本块内的每一条语句，都分别生成其活跃变量集合。

这样的活跃变量分析是后向分析，转移函数为

$$\mathrm{In}[B] = \mathrm{use}[B] \cup (\mathrm{Out}[B] - \mathrm{def}[B])$$

变量$x \in \mathrm{def}$ ，当且仅当 x 被赋值后才被使用， $x \in \mathrm{use}$ 当且仅当存在对 x 取值的表达式，该表达式在所有对 x 的赋值之前。

对于每一个基本块，其活跃变量分析就是把语句分拆，然后用基本块的方法对每条语句处理其活跃变量。

step-9:

修改 `tac/dataflow.cpp` 的 `computeDefAndLifeUse` ，生成上文提到的 use 和 def 集合。

对于被调用者取参，参数从调用者取值，为定值点，更新 DEF 集合；函数返回值也是定值点，同样更新 DEF 集合。

函数传参是使用某个变量，因此把它加入到 LiveUse 集合里。

接着对于基本块内每条语句，从后往前计算其 LiveOut 集合，具体是从其后方语句转移：删掉后方被赋值的变量，加入后方的被使用的变量： FETCHARG 和 CALL 都只有被赋值变量，而 SAVEARG 只有被使用变量。

具体而言，就是修改 `tac/dataflow.cpp` 的 `BasicBlock::analyzeLiveness` 函数。

step-10:

全局变量的定义无需作数据流分析。

对于全局变量的存取，对 SaveGSym (SAVEGLOBAL) 采用上文提到的被使用的方法处理；LoadGsym (FETCHGLOBAL) 采用被赋值方法。

2.2.6. 目标代码生成

修改 `asm/riscv_md.cpp`

在框架里，先生成汇编指令的链表，然后对其作窥孔优化，最后对其输出。 窥孔优化只处理了 `mv rd, rd` 的情况。

step-9:

对于函数调用，直接将之对应为 `call` 指令即可。与此同时，调用 `spillDirtyRegs`，将所有调用者保存的寄存器的值，转移到内存记录。 调用完毕返回以后，先还原调用者保存寄存器，然后把返回值转移到 `CALL` 对应的临时变量里。

对于函数传参，如果参数下标大于等于 8， 则把它压在调用栈里，具体是模仿 8086 指令的 `push`，将之压栈；而压栈顺序已经在中间代码生成阶段有所规范。

如果小于 8， 就用提供的 `passParamReg` 进行传参。

类似的可以定义 `fetchParamArg` 取参，由于 RISC-V 架构存放参数的寄存器是连续的，因此其对应

`mv <t0.var 所分配的寄存器>, <a0 + 下标>`

对于下标大于等于 8 的参数，则是

`lw <t0.var 所分配的寄存器>, <下标 - 8>(fp)`

分别用 `AddInstr` 和 `emitInstr` 生成和输出汇编指令。

由于需要模拟压栈，因此需要 `addi` 指令，在 `emitInstr` 部分处理即可。

step-10:

对于全局变量的取值，先用

`la <分配的寄存器> , <符号名>` 取得符号的地址，然后用 `lw <分配的寄存器>, 0(<分配的寄存器>)` 将地址的值转移。

对于存值，则是

`la <分配的寄存器> , <符号名>`

`sw <被存变量的寄存器>, 0(<分配的寄存器>)`

对于全局变量的定义和内存分配，则是扩充 `emitPiece` 对函数的处理方法，遇到 `Globl` 的时候，生成

```
  .data
  .globl <var>
<var>:
```

表示全局变量，然后再按 payload 的类型生成 `.word` 和 `.zero` 段。