编绎原理 · stage-4 · 实验报告

计01　容逸朗　2020010869

实验内容

实验目标

本阶段需要支持 MiniDecaf 语言函数的声明和定义、函数调用和全局变量。

实验思路

函数

为了在函数调用 (call) 时可以检查对应的参数类型是否正确，我们需要在创建符号表时遍历函数的参数列，此时我们需要考虑函数声明 / 定义的三种情况：

```
/* Case 1: 只有声明 */
int f(int a, int b);
int main() {
  return f(1, 2);
}
```

```
/* Case 2: 只有定义 */
int f(int c, int d) {
  return c + d;
}
int main() {
  return f(1, 2);
}
```

```
/* Case 3: 有声明和定义 */
int f(int a, int b);
int main() {
  return f(1, 2);
}
int f(int c, int d) {
  return c + d;
}
```

如果是第一或第二种情况，那么沿用以往的函数处理方法并不会导致函数重定义 / 冲突。然而对于第三种情况，由于声明和定义语句都会指向名为 f 的函数，这会导致全局符号表冲突，此时不难想到如下做法：

- 当遇到第一个函数语句 (无论是声明还是定义) 时，在全局作用域内加入函数 f 的符号，同时遍历参数表把对应的参数符号加入函数 f 的符号表中。对于后续的声明仅作类型检查，不做创建符号表操作，对于定义则只创建语句 (stmts) 内的符号。

此方法会导致上面的情况 3 在翻译时出错，原因在于第 6 行的函数定义在符号表创建阶段时没有为参数创建相应的符号。

为了解决这一问题，我做了如下的操作：

- 在创建符号表阶段时为**函数声明**和**函数定义**分别创建了两个不同的符号，其中：
  - 前者以 `{函数名}__##DEF##__` 为名创建新的符号（由于改变了名字，因此在生成 Label 时需要去掉多余的符号）；
  - 后者沿用以往的符号；
- 同时，在函数声明时出现冲突时（对应多个声明的情况），我们只检查参数类型是否一致，**不会**把参数加入函数作用域内。
  - 由于函数声明只参与**语义分析**而不参与翻译，这样做是没有问题的；
- 函数定义出现冲突时，由于本次实现不要求实现函数重载，因此直接沿用旧方法报错即可。

函数调用

由于需要实现 c 的标准传参约定（即前八个参数依次存入 a0-a7 寄存器之中，后面的参数由右至左压栈），可以对两种类型的参数分别做如下操作：

- 对于以寄存器传参的参数

  在 `Translation::FuncDefn` 阶段中为参数对应的 `Temp` 生成 `POP` 指令，使得后端在 `emitPopTac` 时可以把对应位置的参数赋给 `Temp`。

- 对于需要压栈的参数

  我们需要在 `Translation::FuncDefn` 为需要压栈的寄存器分配栈空间，具体来说，就是调用语法糖 `NEXT_OFFSET` 取得压栈位置，然后在函数开始时（即调用 `TransHelper::startFunc`）遍历函数符号表，利用 `Symbol::getOrder()` 找到是参数且编号大于 7 的符号，然后把对应的 `is_offset_fixed` 设为 `true` 并传入偏移量信息。

在后端需要实现 `emitCallTac` 的翻译，具体可以分为五步：

- 先把要压栈的内容压栈（先分配空间 `sp ← sp-4`，然后 `sw`）
- 然后调用 `RiscvDesc::passParamReg` 实现寄存器传参
- 用 `RiscvDesc::spillDirtyRegs` 保存脏的寄存器
- 生成 `call` 指令调用函数
- 使用 `mv XX, a0` 的方式保存返回值

全局变量

在创建符号表阶段遍历到 `ast::VarDecl` 节点时，需要调用 `Variable::setGlobalInit` 的方法为全局变量赋值，同时在翻译阶段时增加判断，防止程序在中端翻译全局变量。(尽管在最终生成汇编代码时不会有任何影响)

在后端的 `RiscvDesc::emitPieces` 函数中遍历全局符号表，把全局变量用如下格式翻译即可：

```
1          .data
2          .globl a
3  a:
4          .word 2022
```

思考题

1. MiniDecaf 的函数调用时参数求值的顺序是未定义行为。试写出一段 MiniDecaf 代码，使得不同的参数求值顺序会导致不同的返回结果。

- 例子如下：

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

此时若先求第一个参数的值，则返回值为 4；若先求第二个参数的值，则返回值可以为 7.

2. 为何 RISC-V 标准调用约定中要引入 callee-saved 和 caller-saved 两类寄存器，而不是要求所有寄存器完全由 caller/callee 中的一方保存？为何保存返回地址的 ra 寄存器是 caller-saved 寄存器？

- 这是出于性能方面的考量：
  - 如果每次调用函数都需要遍历一次寄存器堆然后储存值，那么对内存操作的部分会很大程度上影响了程序的性能；
  - 其次是，大部分程序 / 函数阶段在执行时都不会利用到全部的寄存器。因此任意一方保存所有寄存器是没有必要的；
  - 因此，引入 callee-saved 和 caller-saved 寄存器可以使得程序运行变快，原因在于 caller 和 callee 都被分配到一段寄存器空间（两者对应使用 callee-saved 和 caller-saved 寄存器），这样可以使用函数调用时需要保存的寄存器数目大大减少了。
- 因为在进入（被调用的）函数前 ra 的值已经被新的返回值覆盖，故 ra 是 caller-saved 寄存器。

3. 写出 la v0, a 这一 RiscV 伪指令可能会被转换成哪些 RiscV 指令的组合（说出两种可能即可）。

- 可能的指令组合如下：

```
auipc v0, %hi(a)
addi v0, v0, %lo(a)
```

```
auipc v0, %hi(a)
lw v0, %lo(a)(v0)
```

注：上面的指令组合没有考虑由于立即数符号位扩展所带来的误差，若要修正错误则需要为第一条指令的 `a` 增加偏移量 `0x800` 来较正结果。

参考

实现代码的过程中参考了如下资料：

1. 实验思路指导与问答墙.
2. Nora Sandler, C Compiler, Part 9: Functions.
3. lackluster, 调用约定（Calling Convention）浅析.
4. RISC-V Assembly Programmer's Manual.