Stage 4

2023010730 杨凯森

实现细节

1. 词法语法分析

AST 节点设计

根据文档要求，我们实现了以下 AST 节点：

- **Function**: 表示函数定义或声明
  - `ret_t`：返回类型
  - `ident`：函数名
  - `params`：参数列表（`Declaration` 列表）
  - `body`：函数体（`Block`）
- **Call**: 表示函数调用表达式
  - `func`：被调用的函数标识符
  - `args`：参数表达式列表

语法规则修改

在 `frontend/parser/ply_parser.py` 中：

1. 修改 `program` 规则为 `program : function_list`
2. 添加 `function_list` 规则支持多个函数
3. 添加 `function` 规则支持函数定义和声明
4. 添加 `parameter_list` 规则处理参数列表
5. 修改 `postfix` 规则支持函数调用语法

2. 语义分析

符号表管理

在 `frontend/typecheck/namer.py` 中：

1. **函数符号收集**：遍历 `program.children` 收集所有函数节点

2. **重复定义检查**：使用 `seen_names` 集合检测重复的函数定义
3. **函数声明处理**：先声明所有函数符号，再处理函数体
4. **参数作用域**：在函数体作用域中添加参数符号，支持参数与局部变量的作用域检查

关键实现：

- 使用 `ctx.current_scope().lookup()` 检查当前作用域的声明冲突，正确处理变量遮蔽参数的情况
- 迭代 `block.children` 而非直接迭代 `block`，避免潜在的迭代问题

类型检查

在 `frontend/typecheck/typer.py` 中：

1. **函数类型检查**：检查函数返回类型与 return 语句的一致性
2. **函数调用类型检查**：检查参数个数和类型匹配
3. **参数类型处理**：在函数作用域中为参数设置类型

3. 中间代码生成

TAC 指令设计

采用文档推荐的"一整条函数调用"设计：

- **CALL 指令**：`_Tdst = CALL func(_Targ1, _Targ2, ...)`
  - 优点：语义清晰，与高级语言对应关系明确
  - 参数传递和返回值处理集中在一个指令中

参数分配

在 `frontend/tacgen/tacgen.py` 中：

- 参数使用临时变量索引从 0 开始（`_T0`，`_T1`，...）
- 局部变量从 `len(func.params)` 开始分配，避免冲突

4. 指令选择

在 `backend/riscv/riscvasmemitter.py` 的 `RiscvInstrSelector.visitCall` 中：

1. **参数传递**：将前 8 个参数移动到 `a0` - `a7` 寄存器
2. **函数调用**：生成 `call func` 指令
3. **返回值处理**：将 `a0`（返回值）移动到目标临时变量

当前实现限制：不支持超过 8 个参数（抛出 `NotImplementedError`）

5. 寄存器分配

调用者（Caller）处理

在 `backend/reg/bruteregalloc.py` 的 `allocForLoc` 中：

调用前保存：

1. **保存 `ra` 寄存器**：检测到 `Riscv.Call` 指令时，首先分配栈空间并保存 `ra`（因为 `call` 指令会覆盖 `ra`）
2. 遍历 `loc.liveOut` 中所有活跃的临时变量
3. 对于绑定到 caller-saved 寄存器的临时变量，保存到栈
4. 记录保存的 `(temp_index, reg)` 对

调用后恢复：

1. **恢复 `ra` 寄存器**：从栈恢复 `ra`
2. 检查临时变量是否仍在 `loc.liveOut` 中
3. 无论 `fillRegs` 是否改变了绑定，都恢复到原始寄存器
4. 从栈加载保存的值

关键实现细节：

- `ra` 作为 caller-saved 寄存器，在每次函数调用前后都会被保存/恢复
- 参数如果活跃且绑定到 caller-saved 寄存器，也会被保存（虽然参数在参数寄存器中，但可能被重新绑定到其他 caller-saved 寄存器）
- 恢复时优先检查是否有栈偏移，如果有则从栈加载（支持参数被保存的情况）

被调用者（Callee）处理

参数绑定：

- 参数在首次使用时通过 `allocRegFor` 分配寄存器
- `emitLoadFromStack` 检查如果是参数且未保存到栈，则从参数寄存器（`a0 - a7`）加载
- 通过 `bind` 将参数临时变量绑定到分配的寄存器

返回地址处理：
`ra` 寄存器按照 caller-saved 约定处理：

1. **调用者保存**：在 `backend/reg/bruteregalloc.py` 的 `allocForLoc` 中，检测到 `Riscv.Call` 指令时：
   - 调用前：将 `ra` 保存到栈（分配栈空间并生成 `sw ra, offset(sp)`）
   - 调用后：从栈恢复 `ra`（生成 `lw ra, offset(sp)`）

2. **被调用者不处理**：在 `backend/riscv/riscvasmemitter.py` 的 `emitFunc` 中，prologue 和 epilogue 不再保存/恢复 `ra`
3. **栈对齐**：确保栈指针 16 字节对齐（RISC-V 调用约定要求）

6. 栈管理

栈布局

```
高地址
  ...
  参数（如果超过8个，在调用者栈帧中）
  ...
  CalleeSaved[0]          <- sp + 0
  CalleeSaved[1]          <- sp + 4
  ...
  局部变量和溢出临时变量
  ra (caller-saved)       <- 在调用前后临时保存
低地址                    <- sp (16字节对齐)
```

注意： `ra` 不在被调用者的栈帧中，而是在调用者需要时临时保存到栈上。

参数处理

- **参数传递**：前 8 个参数通过 `a0` - `a7` 传递
- **参数存储**：参数在函数开始时从参数寄存器加载到分配的寄存器
- **参数保存**：如果参数在调用后仍然活跃且绑定到 caller-saved 寄存器，会被保存到栈

实现中的关键问题与解决方案

1. 参数在递归调用中的保存和恢复

问题：在递归函数（如 `fib` ）中，参数 `n` 在第一次调用后仍然需要用于计算 `n-2` ，但参数寄存器可能被覆盖。

解决方案：

- 在调用前，如果参数绑定到 caller-saved 寄存器且仍然活跃，保存到栈
- 在调用后，从栈恢复参数值
- 修改 `emitLoadFromStack` ，优先检查是否有栈偏移，支持从栈恢复保存的参数

2. fillRegs 改变寄存器绑定后的恢复

问题： `fillRegs` 可能将返回值移动到已保存的寄存器，导致恢复逻辑判断错误。

解决方案：

- 恢复逻辑无条件恢复所有保存的寄存器，无论当前绑定状态
- 如果临时变量绑定到不同寄存器，先解绑再恢复到原始寄存器

3. 栈对齐

问题：RISC-V 调用约定要求栈指针 16 字节对齐。

解决方案：

- 在 `emitFunc` 中，将 `nextLocalOffset` 向上舍入到 16 的倍数
- 在序言和尾声中使用相同的对齐偏移

测试结果

所有 step9 的测试用例均通过，包括：

- 基本函数调用和返回值
- 递归函数（如 `fib`）
- 参数传递
- 函数声明和定义
- 作用域检查（变量遮蔽参数等）

思考题

1. 中间表示中的函数调用指令设计

问题：你更倾向采纳哪一种中间表示中的函数调用指令的设计（一整条函数调用 vs 传参和调用分离）？写一些你认为两种设计方案各自的优劣之处。

回答：

我采用了"一整条函数调用"的设计，即：

```
_T3 = CALL foo(_T2, _T1, _T0)
```

优点：

1. **语义清晰**：与高级语言的函数调用语义直接对应，易于理解和维护
2. **数据流分析简单**：参数和返回值在一个指令中，数据流关系明确
3. **优化友好**：可以整体优化函数调用，如内联、尾调用优化等
4. **实现简洁**：不需要额外的 PARAM 指令，减少中间表示的大小

缺点：

1. **灵活性较低**：参数传递和调用耦合，难以单独优化参数传递
2. **扩展性**：如果未来需要支持更复杂的调用约定，可能需要修改指令设计

"传参和调用分离"设计的优缺点：

优点：

1. **灵活性高**：可以单独优化参数传递，支持部分求值等优化
2. **扩展性好**：易于支持不同的调用约定和参数传递方式
3. **调试友好**：可以单独查看参数传递过程

缺点：

1. **数据流分析复杂**：需要追踪 PARAM 指令和 CALL 指令的关系
2. **中间表示冗余**：需要额外的 PARAM 指令
3. **实现复杂**：需要维护参数传递的状态

2. RISC-V 调用约定中的寄存器分类

问题：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？

回答：

为何需要两类寄存器：

1. **性能优化**：
   - **Callee-saved 寄存器**：适合保存长期使用的值（如循环变量、函数参数），避免频繁保存/恢复
   - **Caller-saved 寄存器**：适合保存临时值，调用者可以选择性保存，减少不必要的保存操作
2. **代码生成灵活性**：
   - 编译器可以根据值的生命周期选择合适的寄存器类型
   - 短期值用 caller-saved，长期值用 callee-saved，减少栈操作
3. **平衡保存开销**：

- 如果全部由 caller 保存：调用频繁时开销大
- 如果全部由 callee 保存：被调用函数可能不需要保存某些寄存器，造成浪费
- 两类寄存器平衡了这两种情况

为何 ra 是 caller-saved：

1. **调用语义**： `ra` 只在函数调用时被修改（ `call` 指令自动设置），函数返回后不再需要
2. **叶子函数优化**：叶子函数（不调用其他函数）不需要保存 `ra` ，减少开销
3. **调用频率**：大多数函数都会调用其他函数， `ra` 会被覆盖，由 caller 保存更合理
4. **实现灵活性**：只有调用其他函数的函数才需要保存 `ra` ，由调用者决定是否需要保存

实际实现中的处理：

在我们的实现中， `ra` 严格按照 caller-saved 约定处理：

- **调用者负责**：在 `backend/reg/bruteregalloc.py` 的 `allocForLoc` 中，检测到 `Riscv.Call` 指令时：
  - 调用前：分配栈空间并保存 `ra` （ `sw ra, offset(sp)` ）
  - 调用后：从栈恢复 `ra` （ `lw ra, offset(sp)` ）
- **被调用者不处理**：在 `backend/riscv/riscvasmemitter.py` 的 `emitFunc` 中，prologue 和 epilogue 不再保存/恢复 `ra`
- **栈空间**： `ra` 的保存空间在调用者的栈帧中临时分配，不在被调用者的固定栈布局中

这样实现符合 RISC-V 调用约定，并且只有实际调用其他函数的代码路径才会保存/恢复 `ra` ，对叶子函数没有额外开销。