Final Report

胥昊天

2025-06-20 01:57:20

一、引言

该报告主要面向：还未学习编译原理的计算机专业学生。在此，我们假设读者已经具备基本的计算机知识，并希望通过这个报告：

1. 讲解编译器的基本实现原理（比如：`Fudan mini Java` 编译器）。
2. 介绍具体的代码实现方案。
3. 以及一些已经实现的编译器优化和可能的优化的方向。

在本报告中，我们都以 `Fudan mini Java`（Java 语言的裁剪版）为例，讨论编译器原理及实现。

实验总共分为 10 个作业和 1 个期末项目。每个作业对应一个具体的编译器实现步骤，最后一个项目是将所有步骤整合成一个完整的编译器。

每个实验的文件目录结构都如下：（每个实验并不相互依赖，可以单独运行）

```
.
  CMakeLists.txt
  Makefile
  README.md
  build
  docs          # 实验报告
  include       # .hh 头文件
  lib           # .cc 源文件
  test          # 供测试用的 fmj 文件
  tools         # main 函数（最终的编译器文件）
  vendor        # 依赖的外部库文件
```

二、编译流程

在现代编译器的设计中，编译器的工作流程大致如下：（这里我们仅简要介绍，后续会展开）

Step1: 词法 + 语法 + 语义分析

- fmj（Fudan Mini Java）源代码文件，包含类定义、方法声明、变量声明和各种语句。这是编译器的输入。
- 我们使用 `Flex` 进行词法分析将源代码转换为词法单元流，`Bison` 进行语法分析构建抽象语法树（AST）。同时进行语义分析，包括类型检查、作用域分析和名称解析，生成带有语义信息的 AST 。

Step2: AST 到 IR

- Tiger IR+（IRP）是基于表达式的树形中间表示，包含临时变量、标签、跳转等低级结构。代码仍保持树形结构，但已经接近汇编语言的抽象级别。
- 我们将 AST 转换为 Tiger IR+ 的中间表示。消除类的概念，将方法转换为函数，处理对象布局和方法调用，生成以表达式和语句为基础的树形中间代码。

Step3: IR 到 QUAD

- 四元式（QUAD）中间代码，采用线性指令序列形式，每条指令最多包含一个操作符和三个操作数。指令包括 MOVE、LOAD、STORE、JUMP、CJUMP 等，为后续 SSA 转换和优化做准备。
- 我们将树形 IR 转换为线性的四元式（QUAD）序列。每个四元式对应一个基本操作，如移动、加载、存储、二元运算等，同时标注每条指令定义或者使用了的变量信息。

Step4: SSA 转换

- SSA 形式的四元式代码，每个变量具有唯一的定义点，Phi 函数处理控制流汇合。这种形式使得数据流分析更加精确，为编译器优化提供了理想的中间表示。
- 将四元式转换为静态单赋值（SSA）形式，确保每个变量只被赋值一次。在控制流汇合点插入 Phi 函数，重命名变量以满足 SSA 性质，便于后续数据流分析和优化。

Step5: 编译优化

- SSA 形式非常便于编译优化，所以我们可以基于 SSA 形式进行各种优化。当然，为了未来可能的优化，我们希望优化后的代码仍然保持 SSA 形式。
- 这里我们实现条件常量传播（CCP）和无用代码消除优化。通过数据流分析识别常量变量并进行传播，删除不可达代码和无效赋值，简化控制流结构，提升程序执行效率。

Step6: 寄存器分配

- 我们为寄存器分配做准备工作，包括构建干扰图、计算活跃变量信息、处理函数调用约定等。将 SSA 形式转换回普通形式，同时保持优化效果。此时，代码已经退出 SSA 形式但保持了优化结果，可以直接进行寄存器着色算法。

Step6: 汇编生成

- 根据寄存器分配结果将四元式转换为 ARM 汇编代码。处理寄存器分配、栈帧管理、函数调用约定，生成符合 ARM 架构规范的汇编指令序列。

Step6: 链接并生成可执行文件

- 使用交叉编译工具链将汇编代码转换为机器码，链接系统库函数，生成可在目标平台执行的二进制文件。这一步由外部工具完成。
- 最终的可执行程序，可以在 ARM 平台上直接运行。通过 QEMU 模拟器可以在 x86 平台上测试执行效果，验证整个编译流程的正确性。

```
Source Code (.fmj)
    |
    | (HW3: 词法+语法+语义分析)
    |
AST (.2-semant.ast)
    |
    | (HW4/5: AST到中间代码)
    |
IR (.3.irp)
    |
    |
    | (HW6: 中间代码到四元式)
Quad (.4.quad)
    |
    | (HW7: SSA转换)
    |
SSA Quad (.4-ssa.quad)
    |
    | (HW10: 代码优化)
    |
Optimized SSA Quad (.4-ssa-opt.quad)
    |
    | (HW8: 寄存器分配准备)
    |
Prepared Quad (.4-prepared.quad)
    |
    | (HW9: 汇编生成)
    |
```

```
Assembly (.s)
    |
    | (汇编器+链接器)
    |
Executable
```

三、语法分析

语法分析是编译过程中的关键阶段，其目标是将词法分析器产生的词法单元流（token stream）转换为抽象语法树（AST）。在本项目中，采用了基于**上下文无关文法（CFG）**的 **LALR(1)** 语法分析方法。

1. 文法定义

FDMJ 语言的语法规则在 `parser.yy` 中定义，采用 `Bison` 语法描述：

```
PROG: MAINMETHOD CLASSDECLLIST
    { $$ = new Program(pos, $1, $2); }

MAINMETHOD: PUBLIC INT MAIN '(' ')' '{' VARDECLLIST STMLIST '}'
    { $$ = new MainMethod(pos, $7, $8); }

STM: '{' STMLIST '}'
    { $$ = new Nested(pos, $2); }
   | IF '(' EXP ')' STM ELSE STM
    { $$ = new If(pos, $3, $5, $7); }
   | WHILE '(' EXP ')' STM
    { $$ = new While(pos, $3, $5); }
```

2. 词法单元定义

词法单元在 lexer.ll 中通过正则表达式定义：

```
num                 [1-9][0-9]*|0
name                [a-z_A-Z][a-z_A-Z0-9]*
punct               [()\[\]{}=,;.]

"public"            { return PUBLIC; }
"class"             { return CLASS; }
"int"               { return INT; }
"if"                { return IF; }
"while"             { return WHILE; }
```

3. 语法分析器实现

使用 `Bison` （Yacc 的 GNU 实现）自动生成 LALR(1) 语法分析器：

- **LALR(1) 表构造**：Bison 自动构造 LALR(1) 分析表，处理移进-归约决策
- **冲突解决**：通过优先级声明（`%left`, `%right`, `%nonassoc`）和结合性规则解决语法歧义
- **错误恢复**：内置错误恢复机制，提供详细的语法错误信息

每个文法规则都对应一个语义动作，负责构建 AST 节点：

```
EXP: EXP '+' EXP
    { $$ = new BinaryOp(pos, $1, "+", $3); }
   | EXP '*' EXP
    { $$ = new BinaryOp(pos, $1, "*", $3); }
   | '(' EXP ')'
    { $$ = $2; }
   | NUM
    { $$ = new NumExp(pos, $1); }
```

4. 错误处理机制

- **位置跟踪**：通过 `ast_location` 类精确跟踪每个 token 的源代码位置
- **详细错误报告**：`%define parse.error detailed` 提供丰富的错误上下文信息
- **错误恢复**：使用错误符号进行局部错误恢复，提高分析器的鲁棒性

5. 模块化设计

- **词法分析器类**：`ASTLexer` 封装 Flex 生成的词法分析器
- **语法分析器类**：`ASTParser` 封装 Bison 生成的语法分析器
- **AST 节点层次**：完整的 AST 类层次结构，支持访问者模式

三、类型检查

类型检查是编译器确保程序语义正确性的关键阶段，主要验证变量使用、表达式计算和方法调用等是否符合语言规范。以下是本编译器类型检查的核心实现：

1. 类型系统设计

类型系统基于 `AST_Semant` 结构体实现，包含三种基本类型：

- INT：整型，用于数值运算和逻辑判断
- ARRAY：整型数组，需记录维度信息
- CLASS：类类型，需记录类名信息

每种类型还标记是否为左值（可被赋值），并支持类型兼容性检查（如类继承关系）。

2. 核心检查机制

类型检查通过 AST_Semant_Visitor 访问者模式实现，主要功能包括：

1. **符号表查询**：
   - 通过 name_maps 查询变量/方法的声明信息
   - 支持多级查找（局部变量-> 参数-> 成员变量-> 方法）
2. **赋值兼容性检查**：

```
bool is_assignable(AST_Semant* left, AST_Semant* right) {
    // 检查基础类型匹配
    if (left->get_type() != right->get_type()) return false;

    // 类类型需检查继承关系
    if (left->get_type() == CLASS) {
        return is_subclass(right_class, left_class);
    }
}
```

3. **表达式类型推导**：
   - 二元运算要求操作数类型一致
   - 数组访问要求索引为 INT 类型
   - 方法调用检查参数数量和类型匹配

3. 关键检查场景

1. **变量声明检查**：
   - 数组初始化需验证维度与初始化列表匹配
   - 类成员变量检查可见性
2. **方法调用检查**：

```
void check_method_call(string class_name, string method_name) {
    // 检查方法是否存在
    if (!name_maps->is_method(class_name, method_name)) {
        error(" 方法不存在");
    }

    // 检查参数类型
    auto formals = name_maps->get_method_formal_list(class_name, method_name);
    for (int i=0; i<formals.size(); i++) {
        if (!is_assignable(formals[i], actuals[i])) {
            error(" 参数类型不匹配");
        }
```

```
    }
}
```

3. **控制流检查**：
   - if/while 条件必须为 INT 类型
   - break/continue 必须在循环体内
   - return 类型需匹配方法声明

4. 错误处理

发现类型错误时，通过以下方式报告：

1. 输出错误位置（源代码行号）
2. 明确错误类型（如”变量未定义”、“类型不匹配”）
3. 终止编译过程（exit(1)）

5. 多态处理

通过动态查找方法实现类实现：

```
string find_real_method(string class_name, string method_name) {
    while (存在父类 && 父类方法签名相同) {
        class_name = 父类名;
    }
    return class_name;
}
```

四、IR 中间代码生成

1. 程序结构类

- **tree::Program** 表示整个程序，包含所有函数声明（FuncDecl）的列表
- **tree::FuncDecl** 表示函数声明，存储函数名、参数列表、基本块（Block）、返回类型等信息，是代码生成的起点
- **tree::Block** 表示基本块（一组顺序执行的语句），包含入口标签、语句列表和出口标签，用于构建控制流图（CFG）

2. 语句类（Stm）

- **tree::Seq** 将多个语句组合成顺序执行的语句列表（类似 { stmt1; stmt2; }）
- **tree::LabelStm** 在代码中插入标签（如 L100:），作为跳转目标
- **tree::Jump** 无条件跳转（goto L100），直接跳转到指定标签

- **tree::Cjump** 条件跳转（if (x > y) goto L1; else goto L2;），根据比较结果选择跳转目标
- **tree::Move** 赋值语句（t1 = t2），将源表达式（src）的值赋给目标（dst）
- **tree::Phi** SSA 形式的 Phi 函数，用于合并不同控制流路径的变量值（如 t1 = (t2@B1, t3@B2)）
- **tree::ExpStm** 将表达式转换为语句，忽略返回值（如 a + b; 仅执行副作用）
- **tree::Return** 函数返回语句（return x），携带返回值表达式

3. 表达式类（Exp）

- **tree::Binop** 二元运算（如 a + b、x < y），包含操作符和左右操作数
- **tree::Mem** 内存访问（如 *p），表示从地址加载值或向地址存储值
- **tree::TempExp** 将临时变量（Temp）转换为表达式（如 t1 可直接参与运算）
- **tree::Eseq** 组合语句和表达式（如 { stmt; return exp; }），先执行语句再求值表达式
- **tree::Name** 将标签（Label）转换为地址表达式（如 &L100），用于函数调用或跳转
- **tree::Const** 常量值（如 42），直接嵌入到表达式中
- **tree::Call** 函数调用（如 obj.f(a, b)），包含调用目标、对象和参数列表
- **tree::ExtCall** 外部函数调用（如 print(“hello”)），类似 Call 但目标为外部函数名

4. 辅助类

- **tree::Label** 表示代码标签（如 L100:），用于控制流跳转（goto、if 等）
- **tree::Temp** 表示临时变量（如 t100），用于寄存器分配或中间值存储
- **tree::Type** 表示变量或表达式的类型（INT 或 PTR）

IR 中间代码翻译

1. 算数运算

算数运算（如 `+`、`-`、`*`、`/`）通过 `tree::Binop` 表示。
每个算数运算节点会递归访问左右操作数，并将它们转换为 `tree::Exp`，然后构造一个 `tree::Binop` 节点。

```
void ASTToTreeVisitor::visit(fdmj::BinaryOp* node) {
    node->left->accept(*this);
    auto left = newExp->unEx(&temp_map)->exp;

    node->right->accept(*this);
    auto right = newExp->unEx(&temp_map)->exp;

    newExp = new Tr_ex(new tree::Binop(tree::Type::INT, node->op->op, left, right));
}
```

2. 比较运算

比较运算（如 <、>、== 等）通过 tree::Cjump 表示。
在翻译时，会生成一个条件跳转语句，包含两个目标标签（true 和 false）。
这些标签会被后续的控制流逻辑填补。

```
void ASTToTreeVisitor::visit(fdmj::BinaryOp* node) {
    if (is_comparison_op(node->op->op)) {
        node->left->accept(*this);
        auto left = newExp->unEx(&temp_map)->exp;

        node->right->accept(*this);
        auto right = newExp->unEx(&temp_map)->exp;

        Label* t = temp_map.newlabel();
        Label* f = temp_map.newlabel();
        newExp = new Tr_cx(new Patch_list(t), new Patch_list(f),
                           new tree::Cjump(node->op->op, left, right, t, f));
    }
}
```

3. 逻辑运算

逻辑运算（如 && 和 ||）通过短路求值实现。
- 对于 &&，先翻译左操作数，如果为 false，则直接跳转到 false 标签；否则继续翻译右操作数。- 对于 ||，先翻译左操作数，如果为 true，则直接跳转到 true 标签；否则继续翻译右操作数。

```
void ASTToTreeVisitor::visit(fdmj::BinaryOp* node) {
    if (node->op->op == "&&") {
        auto left_cx = left_tr->unCx(&temp_map);
        auto right_cx = right_tr->unCx(&temp_map);

        auto L1 = left_cx->true_list;
        auto L2 = left_cx->false_list;
        auto L3 = right_cx->true_list;
        auto L4 = right_cx->false_list;

        L1->patch(temp_map.newlabel());
```

```
        L2->add(L4);

        newExp = new Tr_cx(L3, L2, new tree::Seq({left_cx->stm, right_cx->stm}));
    }
}
```

4. 赋值

赋值语句通过 tree::Move 表示。
左值和右值分别被翻译为 tree::Exp，然后构造一个 tree::Move 节点。

```
void ASTToTreeVisitor::visit(fdmj::Assign* node) {
    node->left->accept(*this);
    auto left = newExp->unEx(&temp_map)->exp;

    node->exp->accept(*this);
    auto right = newExp->unEx(&temp_map)->exp;

    newNode = new tree::Move(left, right);
}
```

5. 条件语句（if）

条件语句通过 tree::Cjump 和标签语句实现。
- 首先翻译条件表达式，生成一个 tree::Cjump。- 然后翻译 if 和 else 分支，
分别跳转到对应的标签。

```
void ASTToTreeVisitor::visit(fdmj::If* node) {
    node->exp->accept(*this);
    auto exp_cx = newExp->unCx(&temp_map);

    auto L_true = temp_map.newlabel();
    auto L_false = temp_map.newlabel();
    auto L_end = temp_map.newlabel();

    exp_cx->true_list->patch(L_true);
    exp_cx->false_list->patch(L_false);

    vector<tree::Stm*>* sl = new vector<tree::Stm*>();
    sl->push_back(exp_cx->stm);
    sl->push_back(new tree::LabelStm(L_true));

    node->stm1->accept(*this);
    sl->push_back(static_cast<tree::Stm*>(newNode));
    sl->push_back(new tree::Jump(L_end));

    sl->push_back(new tree::LabelStm(L_false));
    if (node->stm2) {
        node->stm2->accept(*this);
```

```
        sl->push_back(static_cast<tree::Stm*>(newNode));
    }

    sl->push_back(new tree::LabelStm(L_end));
    newNode = new tree::Seq(sl);
}
```

6. 循环语句（while）

循环语句通过循环标签和条件跳转实现。
- 首先生成循环的入口标签。- 然后翻译条件表达式，生成一个 `tree::Cjump`。- 最后翻译循环体，并跳转回入口标签。- 需要实现嵌套循环标签栈，用于 break 和 continue 跳转

```
void ASTToTreeVisitor::visit(fdmj::While* node) {
    auto L_while = temp_map.newlabel();
    auto L_true = temp_map.newlabel();
    auto L_end = temp_map.newlabel();

    node->exp->accept(*this);
    auto exp_cx = newExp->unCx(&temp_map);

    exp_cx->true_list->patch(L_true);
    exp_cx->false_list->patch(L_end);

    vector<tree::Stm*>* sl = new vector<tree::Stm*>();
    sl->push_back(new tree::LabelStm(L_while));
    sl->push_back(exp_cx->stm);
    sl->push_back(new tree::LabelStm(L_true));

    node->stm->accept(*this);
    sl->push_back(static_cast<tree::Stm*>(newNode));
    sl->push_back(new tree::Jump(L_while));
    sl->push_back(new tree::LabelStm(L_end));

    newNode = new tree::Seq(sl);
}
```

7. 数组初始化 (VarDecl->ARRAY)

数组初始化通过 `tree::ExtCall` 调用 `malloc` 函数分配内存，并设置数组的大小和初始值。
- 首先计算数组的大小（如果维数为空, 则需要考虑初始化长度）。- 调用 `malloc` 分配内存，并将数组的大小存储在数组的第一个位置。- 如果数组有初始值列表，则依次将初始值存储到数组中。

8. 数组存取 (ArrayExp)

数组存取通过计算数组的偏移量并访问对应的内存地址实现。

- 首先检查数组下标是否越界（index >= size），如果越界则调用 exit(-1) 终止程序。- 计算数组元素的地址:*(arr + (index + 1) * int_length)。- 返回对应的内存值。

9. 类的初始化在类的初始化过程中，我们为类的每个实例分配内存，并初始化其成员变量和方法表:

- 为类实例分配内存:

```
auto class_malloc = new tree::ExtCall(tree::Type::PTR, "malloc", class_malloc_args);
```

- 初始化成员变量 (带初始化):

```
auto var_mem = new tree::Mem(tree::Type::PTR, new tree::Binop(tree::Type::PTR, "+", cla
newNodes.push_back(new tree::Move(var_mem, array_init));
```

- 初始化方法表:

```
auto method_mem = new tree::Mem(tree::Type::PTR, new tree::Binop(tree::Type::PTR, "+",
auto method_nameExp = new Name(temp_map.newstringlabel(method_real_class + "^" + method
newNodes.push_back(new tree::Move(method_mem, method_nameExp));
```

10. 访问类变量通过 this 指针和变量的偏移量计算变量的地址，并生成访存代码:

```
auto var_mem = new tree::Mem(var_type, new tree::Binop(tree::Type::PTR, "+", this_temp, new
```

11. 访问类方法通过 this 指针和方法的偏移量查找方法的入口地址，并生成调用代码:

```
auto method_mem = new tree::Mem(tree::Type::PTR, new tree::Binop(tree::Type::PTR, "+", objEx
auto method_call = new tree::Call(return_type, method_name, method_mem, args);
```

难点：多态处理

在处理多态时，需要根据实际调用的对象类型来确定方法的真实类名 - 首先获取当前对象的类名 class_name 和方法名 method_name。- 然后通过 name_maps 不断向上查找父类，直到找到第一个与当前类的返回形参结点地址不同的类 (即实现方法不同) - 最后一个相同的类就是方法的真实类名 method_real_class。

```
// 找到第一个 return_formal 不同的
auto cur_class_name = class_name;
auto par_class_name = name_maps->get_parent(class_name);
while (par_class_name != "") {
    auto cur_f_return = name_maps->get_method_return_formal(cur_class_name, method_name);
    auto par_f_return = name_maps->get_method_return_formal(par_class_name, method_name);
    if (par_f_return != cur_f_return)
        break;

    method_real_class = par_class_name;
    cur_class_name = par_class_name;
```

```
    par_class_name = name_maps->get_parent(cur_class_name);
}
```

示例代码：hw4test01.2.ast -> hw4test01.3.irp

```
<?xml version="1.0" encoding="UTF-8"?>
<Program>
    <MainMethod>
        <VarDeclList>
            <VarDecl>
                <Type typeKind="INT"/>
                <IdExp id="x"/>
            </VarDecl>
        </VarDeclList>
        <StmList>
            <Assign>
                <IdExp s_kind="Value" typeKind="INT" lvalue="true" id="x"/>
                <IntExp s_kind="Value" typeKind="INT" lvalue="false" val="1"/>
            </Assign>
            <Return s_kind="Value" typeKind="INT" lvalue="false">
                <IdExp s_kind="Value" typeKind="INT" lvalue="true" id="x"/>
            </Return>
        </StmList>
    </MainMethod>
    <ClassDeclList/>
</Program>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Program>
    <FunctionDeclaration name="_^main^_^main" return_type="INT" last_temp="100" last_label="
        <Blocks>
            <Block entry_label="100">
                <Sequence>
                    <Label label="100"/>
                    <Move>
                        <Temp type="INT" temp="100"/>
                        <Const value="1"/>
                    </Move>
                    <Return>
                        <Temp type="INT" temp="100"/>
                    </Return>
                </Sequence>
            </Block>
        </Blocks>
    </FunctionDeclaration>
```

</Program>

五、Quad 指令选择

Tree2Quad 转换器主要组件及功能

1. 顶层结构转换

- **程序 (Program) 转换**:
  - 遍历函数声明列表 (funcdecllist)
  - 每个函数声明转换为 QuadFuncDecl
  - 最终构造 QuadProgram 作为输出
- **函数声明 (FuncDecl) 转换**:
  - 初始化 temp_map 管理临时变量
  - 遍历基本块列表 (blocks)
  - 每个基本块转换为 QuadBlock
  - 构造包含函数元信息的 QuadFuncDecl

2. 基本块处理

- **基本块 (Block) 转换**:
  - 遍历语句列表 (sl)
  - 每条语句转换为对应的 QuadStm 子类
  - 构造 QuadBlock，包含:
    - 入口标签 (entry_label)
    - 出口标签列表 (exit_labels)

3. 语句转换

语句类型	转换结果	关键特性
赋值 (Move)	QuadMove..	根据操作数类型选择不同指令
条件跳转 (CJump)	QuadCJump	包含关系运算符和跳转目标
无条件跳转 (Jump)	QuadJump	简单跳转指令
标签 (Label)	QuadLabel	标记基本块入口
返回 (Return)	QuadReturn	包含返回值处理

4. 表达式转换

表达式类型	转换结果	输出形式
二元运算 (Binop)	QuadMoveBinop	包含操作符和操作数
访存 (Mem)	QuadLoad	内存加载操作
临时变量 (TempExp)	QuadTerm	表示寄存器变量
常量 (Const)	QuadTerm	嵌入常量值
方法标签 (Name)	QuadTerm	表示类方法地址

5. 方法调用处理

- **类方法调用 (Call)**:
  - 生成 QuadCall
  - 包含:
    - 方法名
    - 对象指针
    - 参数列表
- **外部调用 (ExtCall)**:
  - 生成 QuadExtCall
  - 包含:
    - 外部函数名
    - 参数列表

6. 辅助函数

辅助函数	功能	使用场景
call_helper	生成类方法调用指令	QuadCall 和 Move 语句
extcall_helper	生成外部调用指令	QuadExtCall 和 Move 语句
binop_helper	生成二元运算指令	QuadMoveBinop 和 Binop 语句
load_helper	生成内存加载指令	QuadLoad 和 Mem 语句

7. 状态管理

- **visit_results**:
  - 存储下层节点的转换结果
  - 用于收集生成的 Quad 指令
- **visit_term**:
  - 存储表达式转换结果
  - 用于传递操作数信息
- **temp_map**:
  - 管理临时变量和标签编号
  -

确保变量唯一性

示例代码: hw4test01.3.irp -> hw4test01.4.quad

```
<?xml version="1.0" encoding="UTF-8"?>
<Program>
    <FunctionDeclaration name="_^main^_^main" return_type="INT" last_temp="100" last_label="
        <Blocks>
            <Block entry_label="100">
                <Sequence>
                    <Label label="100"/>
```

```
                    <Move>
                        <Temp type="INT" temp="100"/>
                        <Const value="1"/>
                    </Move>
                    <Return>
                        <Temp type="INT" temp="100"/>
                    </Return>
                </Sequence>
            </Block>
        </Blocks>
    </FunctionDeclaration>
</Program>
```

```
Function _^main^_^main() last_label=100 last_temp=100:
  Block: Entry Label: L100
    Exit labels:
    LABEL L100; def: use:
    MOVE t100:INT <- Const:1; def: 100 use:
    RETURN t100:INT; def: use: 100
```

六、静态单赋值（SSA）

1. 支配边界插入 Phi 函数（quad/quadssa.cc/placePhi）

1. **构造数据流信息**：
   - 创建 DataFlowInfo 对象，提取函数中的所有变量
   - 计算变量的活跃性信息（liveness），包括每个语句的 livein 和 liveout 集合
2. **初始化 Phi 函数记录**：
   - 为每个基本块初始化一个集合 A_phi，用于记录该块中已插入的 Phi 函数变量
3. **遍历变量**：
   - 对于每个变量 a：
     - 如果变量没有定义（如函数形参），则跳过
     - 初始化工作集 W，包含所有定义变量 a 的语句
4. **插入 Phi 函数**：
   - 遍历工作集 W：
     - 对于每个基本块 n_block，获取其支配边界集合
     - 对于支配边界中的每个块 Y：
       - 如果变量 a 的 Phi 函数尚未插入，并且变量 a 在块 Y 的 liveout 集合中：
         - 创建一个新的 Phi 函数，将其插入到块 Y 的语句列表中
         - 将变量 a 添加到 A_phi[Y] 中

2. 重命名变量（quad/quadssa.cc/renameVariables）

1. **构造数据流信息**：

- 创建 DataFlowInfo 对象，提取函数中的所有变量

2. **初始化计数器和栈**：
   - 为每个变量初始化一个计数器 Count，用于生成唯一的变量名
   - 为每个变量初始化一个栈 Stack，用于记录当前作用域内的变量版本
3. **递归重命名**：
   - 从入口基本块开始，递归处理每个基本块：
     - **替换变量的使用**：
       * 对于每条语句，使用栈顶的变量版本替换 use 集合中的变量
     - **重命名变量的定义**：
       * 对于每条语句，生成新的变量版本，更新 def 集合，并将新版本压入栈
     - **处理后继块**：
       * 对于每个后继块中的 Phi 函数，添加当前块的变量版本作为参数
     - **递归处理子块**：
       * 根据支配树递归处理子块
     - **回溯**：
       *

在回溯时，弹出当前块中定义的变量版本

示例代码：hw10test05.4.quad -> hw10test05.4-ssa.quad

```
Function _^main^_^main() last_label=100 last_temp=105:
  Block: Entry Label: L100
    Exit labels:
    LABEL L100; def: use:
    MOVE_EXTCALL t100:INT <- getint(); def: 100 use:
    MOVE_BINOP t101:INT <- (+, t100:INT, Const:1); def: 101 use: 100
    MOVE_BINOP t102:INT <- (+, t101:INT, Const:1); def: 102 use: 101
    MOVE_BINOP t103:INT <- (+, t102:INT, Const:1); def: 103 use: 102
    MOVE_BINOP t104:INT <- (+, t103:INT, Const:1); def: 104 use: 103
    RETURN t104:INT; def: use: 104
```

```
Function _^main^_^main() last_label=100 last_temp=105:
  Block: Entry Label: L100
    Exit labels:
    LABEL L100; def: use:
    MOVE_EXTCALL t10000:INT <- getint(); def: 10000 use:
    MOVE_BINOP t10100:INT <- (+, t10000:INT, Const:1); def: 10100 use: 10000
    MOVE_BINOP t10200:INT <- (+, t10100:INT, Const:1); def: 10200 use: 10100
    MOVE_BINOP t10300:INT <- (+, t10200:INT, Const:1); def: 10300 use: 10200
    MOVE_BINOP t10400:INT <- (+, t10300:INT, Const:1); def: 10400 use: 10300
    RETURN t10400:INT; def: use: 10400
```

七、SSA 优化

(1) calculateBT 函数实现

1. 函数作用

- calculateBT 用于遍历函数的基本块和四元式指令，分析每个块是否可达（可执行），并推断每个变量的常量传播信息（单值/多值/无值）
- 该过程是常量传播和可达性分析的核心，便于后续优化（如删除无用代码、替换常量等）

---

2. 主要流程

- 外层循环遍历所有基本块，跳过不可达块
- 内层循环遍历块内所有四元式指令，根据不同指令类型进行处理
- 若分析过程中有新信息产生（如某块变为可达、某变量值发生变化），则跳转回头重新分析，直到收敛

---

3. 关键实现细节

3.1 标签指令（LABEL）

- 若当前块可达且只有一个后继块，则将后继块标记为可达

3.2 条件跳转（CJUMP）

- 若左右操作数均为常量或单值变量，则直接计算条件结果，分别将真/假分支标记为可达
- 若任一操作数为多值，则两个分支都标记为可达

3.3 赋值（MOVE）

- 若源操作数为常量，则目标变量获得单值
- 若源操作数为变量，则目标变量获得与源变量相同的值类型

3.4 二元操作赋值（MOVE_BINOP）

- 若两个操作数均为常量或单值变量，则目标变量获得计算结果的单值
- 若任一操作数为多值，则目标变量为多值

3.5 Phi 函数（PHI）

- 若有任一参数为多值且对应前驱块可达，则目标变量为多值
- 若所有可达前驱参数均为单值且值相同，则目标变量为该单值；若值不同，则为多值

3.6 读内存/函数调用

- 读内存、类方法调用、外部函数调用等均将目标变量标记为多值

4. 迭代与收敛

- 只要有新的可达块或变量值发生变化，立即跳回重新分析，保证信息充分传播，直到所有信息收敛

---

(2) `modifyFunc` 函数实现

1. 函数作用

- `modifyFunc` 用于根据常量传播和可达性分析的结果，对函数的四元式中间代码进行优化和重写
- 主要目标是：**删除无用代码、替换常量、消除不可达块、简化控制流**

---

2. 主要流程

2.1 初始化

- 创建 `Temp_map`，用于新建临时变量和标签
- 遍历所有基本块和块内所有语句

2.2 删除不可达块

- 如果某个基本块不可达，直接从块列表中移除

2.3 删除无用赋值语句

- 对于赋值（`MOVE`）、二元操作赋值（`MOVE_BINOP`）、Phi 函数（`PHI`）：
  - 如果目标变量是单值（常量传播结果），则删除该语句

2.4 优化 Phi 函数参数

- 对于 `PHI` 语句的每个参数：
  - 如果参数是单值变量，则在前驱块插入一条常量赋值语句，并将参数重命名为新变量

2.5 替换单值变量为常量

- 对每条语句的所有使用变量：
  - 如果该变量是单值，则将其在语句中替换为常量

2.6 条件跳转优化

- 对于条件跳转（CJUMP）：
  - 如果左右操作数均为常量，则直接计算条件结果，将 CJUMP 替换为无条件跳转（JUMP），并更新出口标签

2.7 更新计数

- 最后，更新函数的最大临时变量编号和最大标签编号

---

3. 关键实现细节

- **循环与 goto**：每次有代码结构发生变化（如删除语句、块），立即跳回重新遍历，保证所有优化都能被及时应用
- **常量传播与替换**：利用 getRtValue 获取变量的传播结果，进行常量替换和死代码删除
- **Phi 参数特殊处理**：单值参数需要在前驱块插入赋值，保证 SSA 形式正确
- **控制流简化**：将恒定条件的 CJUMP 转换为 JUMP，简化控制流

示例代码：hw8test11.4-ssa.quad -> hw8test11.5-ssa-opt.quad

```
Function _^main^_^main() last_label=105 last_temp=104:
  Block: Entry Label: L105
    Exit labels: L102 L103
    LABEL L105; def: use:
    MOVE t10000:INT <- Const:-2; def: 10000 use:
    CJUMP > t10000:INT Const:0? L102 : L103; def: use: 10000
  Block: Entry Label: L102
    Exit labels: L104
    LABEL L102; def: use:
    EXTCALL putint(t10000:INT); def: use: 10000
    JUMP L104; def: use:
  Block: Entry Label: L103
    Exit labels: L104
    LABEL L103; def: use:
    MOVE_BINOP t10200:INT <- (-, Const:0, t10000:INT); def: 10200 use: 10000
    EXTCALL putint(t10200:INT); def: use: 10200
    JUMP L104; def: use:
  Block: Entry Label: L104
    Exit labels:
    LABEL L104; def: use:
    EXTCALL putch(Const:10); def: use:
    RETURN t10000:INT; def: use: 10000

Function _^main^_^main() last_label=105 last_temp=104:
  Block: Entry Label: L105
```

```
    Exit labels: L103
    LABEL L105; def: use:
    JUMP L103; def: use:
  Block: Entry Label: L103
    Exit labels: L104
    LABEL L103; def: use:
    EXTCALL putint(Const:2); def: use:
    JUMP L104; def: use:
  Block: Entry Label: L104
    Exit labels:
    LABEL L104; def: use:
    EXTCALL putch(Const:10); def: use:
    RETURN Const:-2; def: use:
```

八、Quad 程序准备与活跃性分析

Quad 程序准备

为函数级寄存器分配准备中间代码，确保符合 ARM 调用约定，处理特殊指令模式，并优化指令序列以提高寄存器分配效果。

2. 主要处理阶段

2.1 函数入口处理 (`handleEntry`)

- **参数寄存器分配**:
  - 将前 4 个参数分配到 r0-r3 寄存器
  - 生成 MOV 指令将参数从寄存器拷贝到临时变量

2.2 返回指令处理 (`handleReturn`)

- **返回值处理**:
  - 插入 MOV 指令将返回值移动到 r0
  - 修改 RETURN 指令直接使用 r0

2.3 Phi 函数处理 (`handlePhi`)

- **Phi 消除**:
  - 在控制流前驱块插入 MOV 指令
  - 删除原始 Phi 节点

2.4 调用指令处理 (`handleCall`)

- **调用约定适配**:
  - 参数移动到 r0-r3
  - 返回值从 r0 移动到目标

2.5 特殊指令处理 (handleOtherStm)

指令类型	处理规则
STORE	常量/标签源 → 先 MOV 到临时变量
MOVE_BINOP	常量操作数 → 先 MOV 到临时变量
CJUMP	常量比较 → 先 MOV 到临时变量

活跃性分析 (dataflowinfo.cc/computeLiveness)

1. 核心功能

- **目标**：计算函数内每个四元式语句的变量活跃性（live-in 和 live-out 集合）
- **算法**：基于控制流图的反向数据流分析
- **关键数据结构**：

```
map<QuadStm*, set<int>> livein;  // 语句入口活跃变量
map<QuadStm*, set<int>> liveout; // 语句出口活跃变量
```

2. 主要处理流程

1. **初始化阶段**：
   - 构建控制流图（ControlFlowInfo）
   - 建立语句级关系映射：

```
map<QuadStm*, QuadStm*> nextStmtInBlock; // 块内语句顺序
map<QuadBlock*, vector<QuadStm*>> blockSuccStatements; // 块间跳转关系
```

2. **迭代分析阶段**：

```
while (changed) {
    changed = false;
    // 逆序处理基本块和语句
    for (block : reverse(blocks)) {
        for (stmt : reverse(block->stmts)) {
            // 更新 live-in 和 live-out 集合
            updateLiveness(stmt);
        }
    }
}
```

3. **集合更新规则**：
   - **live-in**：(liveout - def) use
   - **live-out**：
     - 块末尾语句：后继块首语句的 live-in 并集

– 其他语句：下条语句的 live-in

3. 关键实现细节

1. **变量处理**：

```
// 移除定义变量
for (temp : stmt->def) liveIn.erase(temp->num);
// 添加使用变量
for (temp : stmt->use) liveIn.insert(temp->num);
```

2. **控制流处理**：

```
if (isLastStmt(stmt)) {
    // 处理跨块跳转
    for (succ : blockSuccStatements[block])
        liveOut.insert(succ->livein);
} else {
    // 处理块内顺序
    liveOut = nextStmt->livein;
}
```

示例代码：hw4test01.4-ssa.quad -> hw4test01.6-ssa-prepared

```
Function _^main^_^main() last_label=100 last_temp=100:
  Block: Entry Label: L100
    Exit labels:
    LABEL L100; def: use:
    MOVE t10000:INT <- Const:1; def: 10000 use:
    RETURN t10000:INT; def: use: 10000
```

```
Function _^main^_^main() last_label=100 last_temp=100:
  Block: Entry Label: L100
    Exit labels:
    LABEL L100; def: use:
    MOVE t0:INT <- Const:1; def: 0 use:
    RETURN t0:INT; def: use: 0
```

九、寄存器分配

1. simplify()

功能：
在干扰图中查找邻居数小于寄存器数 k 的结点，并将其压入简化栈（simplifiedNodes），同时从图中删除该结点 **细节**：
- 遍历所有结点，跳过被保护（即在 movePairs 中）的结点 - 找到满足条件的结点后，压栈并删除，返回 true - 若无可简化结点，返回 false

---

2. coalesce()

功能：
尝试合并属于同一 move 对的两个结点，以减少 move 指令 **细节**：
- 遍历 movePairs，取出一对 (dst, src) - 若两结点间有干扰边，则不能合并 - 使用 Briggs 策略：合并后新结点的高阶邻居数（度数 k）小于 k 才允许合并 - 若 src 是机器寄存器，交换 dst 和 src，保证 dst 为机器寄存器 - 合并后，删除 src 结点，将其邻居与合并集添加到 dst，并更新 movePairs 中所有 src 为 dst - 合并成功返回 true，否则继续尝试，若无可合并的，返回 false

---

3. freeze()

功能：
当无法简化或合并时，解除某一 move 对的保护关系，使相关结点可以被简化
细节：
- 直接移除 movePairs 中的一个 move 对（及其相关的对），返回 true - 若 movePairs 为空，返回 false

---

4. spill()

功能：
选择一个高干扰（度数最大）的结点作为溢出结点，压入简化栈，并从图中删除
细节：
- 记录度数最大的结点，执行压栈和删除操作，返回 true - 若无可溢出结点，返回 false

---

5. select()

功能：
从简化栈中依次弹出结点分配颜色，若无法分配则标记为溢出
细节：
- 先为机器寄存器分配固定颜色 - 对每个弹出的结点，统计其邻居已用颜色，分配未被占用的颜色 - 若无可用颜色，则将该结点及其合并集标记为溢出 - 最后调用 checkColoring() 检查着色合法性

十、ARM 汇编生成

1. **初始化与函数头部生成**：
   - 规范化函数名，生成汇编标签和段声明
   - 输出 `.balign 4`、`.global`、`.section .text` 等指令

- 生成函数入口标签，保存寄存器（`push {r4-r10, fp, lr}`），设置帧指针（`add fp, sp, #32`）
- 若有溢出变量（spilled temps），为其分配栈空间（`sub sp, sp, #N`）

2. **遍历基本块与四元式**：
   - 对每个基本块，遍历其四元式（QuadStm）列表
   - 对每条四元式，根据类型（kind）分情况处理
3. **四元式到汇编的转换**：
   - **LABEL**：输出汇编标签
   - **MOVE**：处理寄存器/内存间的数据移动，必要时加载/存储溢出变量
   - **MOVE_BINOP**：二元操作（如加减乘），支持与 LOAD/STORE 合并优化（如 `ldr/str [base, offset]`）
   - **LOAD/STORE**：内存读写，处理溢出变量的加载与存储
   - **CALL/MOVE_CALL/EXTCALL/MOVE_EXTCALL**：函数调用，处理参数与返回值
   - **JUMP/CJUMP**：无条件/条件跳转，优化多余的连续标签跳转
   - **RETURN**：恢复栈帧，弹出寄存器，返回
4. **溢出变量处理**：
   - 对于分配到栈上的临时变量，使用 `ldr/str` 指令在需要时加载/保存
   -

统一使用 r9/r10 作为溢出变量的临时寄存器

示例代码：hw4test01.6-ssa-prepared -> hw4test01.s

```
Function _^main^_^main() last_label=100 last_temp=100:
  Block: Entry Label: L100
    Exit labels:
    LABEL L100; def: use:
    MOVE t0:INT <- Const:1; def: 0 use:
    RETURN t0:INT; def: use: 0
```

```
.section .note.GNU-stack

@ Here is the RPI code

@ Here's function: _^main^_^main

.balign 4
.global main
.section .text

main:
        push {r4-r10, fp, lr}
        add fp, sp, #32
main$L100:
```

```
        mov r0, #1
        sub sp, fp, #32
        pop {r4-r10, fp, pc}

.global malloc
.global getint
.global putint
.global putch
.global putarray
.global getch
.global getarray
.global starttime
.global stoptime
```

十一、使用指南

3.1 编译环境配置

```
# 安装依赖
sudo apt install flex bison gcc g++ cmake
```

3.2 项目构建

```
make build  # 生成编译器可执行文件 fdmjc
```

3.3 程序编译

```
make compile  # 编译 test/目录下所有.fmj 文件
```

3.4 执行测试

```
make run  # 使用 qemu 模拟执行生成代码
make run-one PROG=filename.fmj
```

3.5 输出文件说明

文件后缀	生成阶段	内容描述
.2.ast	语法分析阶段	带语义标注的抽象语法树 (AST)，包含类型信息和符号表数据
.3.irp	IR 中间代码生成	初始 IR+ 中间表示，保持高级语言结构特征
.3.irp-canon	IR 规范化	经过规范化的 IR+ 代码，已进行基本块划分和控制流简化
.4.quad	Quad 指令选择	初始 Quad 指令序列，尚未进行基本块划分

文件后缀	生成阶段	内容描述
`.4-xml.quad`	Quad XML 表示	Quad 指令的 XML 格式中间表示，用于跨阶段数据交换
`.4-block.quad`	基本块划分	划分基本块后的 Quad 程序，已建立控制流图结构
`.4-ssa.quad`	SSA 转换	静态单赋值形式的 Quad 代码，包含 phi 函数
`.5-ssa-opt.quad`	SSA 优化	经过优化的 SSA 形式 Quad 代码（如常量传播、死代码消除等）
`.6-ssa-prepared`	寄存器分配准备	适配 ARM 调用约定的 Quad 程序，已完成特殊指令处理
`.6-ssa-prepared.clr`	寄存器分配结果	XML 格式的寄存器分配方案，包含变量-寄存器/栈槽映射关系
`.s`	汇编代码生成	最终 ARMv6 汇编代码，可直接通过 QEMU 执行