编译 PA4：代码生成 报告

17307130015 王鑫涛

一、背景知识

PA4 的任务是代码生成。

我们要处理的函数的入口为 program_class:cgen，应在这里调用 CgenClassTable::code()函数。

```
132  void program_class::cgen(ostream &os)
133  {
134    // spim wants comments to start with '#'
135    os << "# start of generated code\n";
136
137    initialize_constants();
138    CgenClassTable *codegen_classtable = new CgenClassTable(classes,os);
139
140    os << "\n# end of generated code\n";
141  }
```

code 函数逐步为程序各部分内容生成代码。code_global_data()、code_select_gc()和 code_constants()是已经为我们定义好了的，可以作为参考。

```
void CgenClassTable::code()
{
  if (cgen_debug) cout << "coding global data" << endl;
  code_global_data();

  if (cgen_debug) cout << "choosing gc" << endl;
  code_select_gc();

  if (cgen_debug) cout << "coding constants" << endl;
  code_constants();

//                 Add your code to emit
//                   - prototype objects
//                   - class_nameTab
//                   - dispatch tables
//
```

虽然说，PA4 需要我们生成 MIPS 汇编语言的代码，但实际上，我们不需要具体地搞清楚 MIPS 中每条指令是怎么写的，因为 emit_*函数帮我们定义好了指令是怎么写的。当我们

想要写一条指令 X 时，只需要用相应的参数调用 emit_X 即可。

```
//////////////////////////////////////////////////////////////////////////////
//
//  emit_* procedures
//
//  emit_X  writes code for operation "X" to the output stream.
//  There is an emit_X for each opcode X, as well as emit_ functions
//  for generating names according to the naming conventions (see emit.h)
//  and calls to support functions defined in the trap handler.
//
//  Register names and addresses are passed as strings.  See `emit.h'
//  for symbolic names you can use to refer to the strings.
//
//////////////////////////////////////////////////////////////////////////////

static void emit_load(char *dest_reg, int offset, char *source_reg, ostream& s)
{
  s << LW << dest_reg << " " << offset * WORD_SIZE << "(" << source_reg << ")"
    << endl;
}

static void emit_store(char *source_reg, int offset, char *dest_reg, ostream& s)
{
  s << SW << source_reg << " " << offset * WORD_SIZE << "(" << dest_reg << ")"
      << endl;
}
```

所有的字符串常量都被记录在全局变量 stringtable 中，对应一个 StringEntry 的对象。同理，所有整形常量都被记录在全局变量 inittable 中，对应一个 IntEntry。

```
// All string constants are listed in the global "stringtable" and have
// type StringEntry.  StringEntry methods are defined both for String
// constant definitions and references.
//
// All integer constants are listed in the global "inttable" and have
// type IntEntry.  IntEntry methods are defined for Int
// constant definitions and references.
```

代码整体可以分为两部分，一部分是构建 CgenClassTable、CgenNode 等数据结构，一部分是服务于代码的生成。

二、CgenClassTable 和 CgenNode 上的数据访问

函数 get_class_nodes，该函数第一次调用会将 CgenClassTable 的链表 lds 用以构建一个 CgenNode 的向量和一个从 Symbol 类名到 int (其下标) 的 map。通过 get_class_nodes ()函数，可以得到所有类别的节点。

```
std::vector<CgenNode*> CgenClassTable::get_class_nodes() {
    if (_class_nodes.empty()) {
        for (List<CgenNode> *l = nds; l; l = l->tl()) {
            CgenNode* class_node = l->hd();
            _class_nodes.push_back(class_node);
        }
        std::reverse(_class_nodes.begin(), _class_nodes.end());
        for (int i = 0; i < _class_nodes.size(); ++i) {
            _class_nodes[i]->class_tag = i;
            _class_tags.insert(std::make_pair(_class_nodes[i]->get_name(), i));
        }
    }

    return _class_nodes;
}
```

get_class_tags 函数希望得到 get_class_nodes 中生成的那个从 Symbol 类名到 int（其下标）的 map，为了避免该 map 尚未被创建，所以需要先调用一次 get_class_nodes，如果该 map 已被创建，get_class_nodes 将什么也不会干。

```
std::map<Symbol, int> CgenClassTable::get_class_tags() {
    get_class_nodes();
    return _class_tags;
}
```

节点上定义有函数 get_inheritance，用以获取节点的继承关系。当第一次调用时，会自底向上从父类开始遍历 class_node 的祖先类，将其加入一个向量 inheritance 当中，随后 reverse，获取从第一个祖先开始，到该节点的父类的向量。

```
std::vector<CgenNode*> CgenNode::get_inheritance() {
    if (inheritance.empty()) {
        CgenNode* class_node = this;
        while (class_node->name != No_class) {
            inheritance.push_back(class_node);
            class_node = class_node->get_parentnd();
        }
        std::reverse(inheritance.begin(), inheritance.end());
    }

    return inheritance;
}
```

节点上的函数 get_attributes 用以遍历节点所有 feature，将其中的 attr 加入到_attribs 这个向量中。

```
std::vector<attr_class*> CgenNode::get_attributes() {
    if ( _attribs.empty()) {
        for (int j = features->first(); features->more(j); j = features->next(j)) {
            Feature feature = features->nth(j);
            if (feature->is_attr()) {
                _attribs.push_back((attr_class*)feature);
            }
        }
    }

    return _attribs;
}
```

get_full_attributes 函数则会将 get_inheritance 里得到的所有父类的 attr，加入到 _full_attribs 向量中。同时，创建一个从 attr 的名字到其下标的 map。

```
std::vector<attr_class*> CgenNode::get_full_attributes() {
    if ( _full_attribs.empty()) {
        std::vector<CgenNode*> inheritance = get_inheritance();
        for (CgenNode* class_node : inheritance) {
            Features features = class_node->features;
            for (int j = features->first(); features->more(j); j = features-
                Feature feature = features->nth(j);
                if (feature->is_attr()) {
                    _full_attribs.push_back((attr_class*)feature);
                }
            }
        }
        for (int i = 0; i < _full_attribs.size(); ++i) {
            _attrib_idx_tab[ _full_attribs[i]->name] = i;
        }
    }

    return _full_attribs;
}
```

当想要访问上述 map 时，就先调用一次 get_full_attributes，如果 map 已经存在，该函数会直接返回。然后再返回_attrib_idx_tab。

```
std::map<Symbol, int> CgenNode::get_attribute_idx_table() {
    get_full_attributes();
    return _attrib_idx_tab;
}
```

同理，get_methods 函数用以获得类节点下定义的方法。

```
std::vector<method_class*> CgenNode::get_methods() {
    if ( _methods.empty()) {
        for (int i = features->first(); features->more(i); i = features->next(i)) {
            Feature feature = features->nth(i);
            if (!feature->is_attr()) {
                _methods.push_back((method_class*)feature);
            }
        }
    }
    return _methods;
}
```

而 get_full_methods 函数则会将 get_inheritance 里得到的所有父类的 method，加入到_full_methods 向量中，并构建 method 到对应类的 map 和 method 到其在向量中所在下标的列表。如果父类和子类中重复定义了该 method 时，由于是自顶向下访问 inheritance 数组，后来的（子类的）方法会覆盖父类的方法。

```
std::vector<method_class*> CgenNode::get_full_methods() {
    if ( _full_methods.empty()) {

        std::vector<CgenNode*> inheritance = get_inheritance();
        for (CgenNode* _class_node : inheritance) {
            Symbol _class_name = _class_node->name;
            std::vector<method_class*> _methods = _class_node->get_methods();
            for (method_class* _method : _methods) {
                Symbol _method_name = _method->name;
                if ( _dispatch_idx_tab.find(_method_name) == _dispatch_idx_tab.end()) {
                    _full_methods.push_back(_method);
                    _dispatch_idx_tab[_method_name] = _full_methods.size() - 1;
                    _dispatch_class_tab[_method_name] = _class_name;
                } else {
                    int idx = _dispatch_idx_tab[_method_name];
                    _full_methods[idx] = _method;
                    _dispatch_class_tab[_method_name] = _class_name;
                }
            }
        }
    }
    return _full_methods;
}
```

用 get_dispatch_class_table 和 get_dispatch_idx_table 访问上述 2 个 map。

```
std::map<Symbol, Symbol> CgenNode::get_dispatch_class_table() {
    get_full_methods();
    return _dispatch_class_tab;
}

std::map<Symbol, int> CgenNode::get_dispatch_idx_table() {
    get_full_methods();
    return _dispatch_idx_tab;
}
```

三、表达式的码生成

我们需要为各种表达式定义其代码生成的方式。其中，最简单的整形常量、字符串常量及布尔常量表达式的代码生成已经帮我们定义好了，即调用 emit_load_int、emit_load_string 和 emit_load_bool 即可。

```
931  void int_const_class::code(ostream& s)
932  {
933    //
934    // Need to be sure we have an IntEntry *, not an arbitrary Symbol
935    //
936    emit_load_int(ACC,inttable.lookup_string(token->get_string()),s);
937  }
938
939  void string_const_class::code(ostream& s)
940  {
941    emit_load_string(ACC,stringtable.lookup_string(token->get_string()),s);
942  }
943
944  void bool_const_class::code(ostream& s)
945  {
946    emit_load_bool(ACC, BoolConst(val), s);
947  }
```

以 emit_load_int 为例，它会调用 emit_partial_load_address 函数，然后调用 IntEntry 的 code_ref 函数，生成访问这个整形常量的汇编代码。

```
static void emit_load_int(char *dest, IntEntry *i, ostream& s)
{
  emit_partial_load_address(dest,s);
  i->code_ref(s);
  s << endl;
}
```

```
//
// Ints
//
void IntEntry::code_ref(ostream &s)
{
  s << INTCONST_PREFIX << index;
}
```

当我们要去为一个方法进行代码生成时，首先会标记一个 LABEL，然后构建其上下文。我们把 fp，s0，ra 作为被调用者保存的寄存器。为了描述一个作用域内的环境，定义了一个类 Environment，其属性包含一个类节点_class_node、方法的参数的向量等。

```
class Environment {
public:
    std::vector<int> _scope_lengths;
    std::vector<Symbol> _var_idx_tab;
    std::vector<Symbol> _param_idx_tab;
    CgenNode* _class_node;

    Environment() : _class_node(nullptr) {}
```

method_class::code 用以为一个方法进行代码生成。函数的主体是编码其中的表达式 expr。首先保存被调用者保存的寄存器，创建栈帧，将各 formal 加入到环境中，然后为 expr 进行代码生成，再进行寄存器和栈帧的恢复，然后弹出方法的参数，最后 return。

```
void method_class::code(ostream& s, CgenNode* class_node) {
    emit_method_ref(class_node->name, name, s);
    s << LABEL;
    s << "\t# push fp, s0, ra" << endl;
    emit_addiu(SP, SP, -12, s);
    emit_store(FP, 3, SP, s);
    emit_store(SELF, 2, SP, s);
    emit_store(RA, 1, SP, s);
    s << endl;

    s << "\t# fp now points to the return addr in stack" << endl;
    emit_addiu(FP, SP, 4, s);
    s << endl;

    s << "\t# SELF = a0" << endl;
    emit_move(SELF, ACC, s);
    s << endl;

    s << "\t# evaluating expression and put it to ACC" << endl;
    Environment env;
    env._class_node = class_node;
    for (int i = formals->first(); formals->more(i); i = formals->next(i)) {
        env.AddParam(formals->nth(i)->GetName());
    }
    expr->code(s, env);
    s << endl;

    s << "\t# pop fp, s0, ra" << endl;
    emit_load(FP, 3, SP, s);
    emit_load(SELF, 2, SP, s);
    emit_load(RA, 1, SP, s);
    emit_addiu(SP, SP, 12, s);
    s << endl;

    s << "\t# Pop arguments" << endl;
    emit_addiu(SP, SP, GetArgNum() * 4, s);
    s << endl;

    s << "\t# return" << endl;
    emit_return(s);
    s << endl;
}
```

object_class::code 为 Object Id 进行代码生成，首先要找到这个 ID 是在哪里定义的。可能的出处是 let、formal、attribute 和 self。我们分别在环境中检查这些地方。找到对应的位置后，从该位置进行 load 即可。如果开启了 GC，则需要完成一次参数防止和函数调用。

```
void object_class::code(ostream& s, Environment env) {
    s << "\t# Object:" << endl;
    int idx;

    if ((idx = env.LookUpVar(name)) != -1) {
        s << "\t# It is a let variable." << endl;
        emit_load(ACC, idx + 1, SP, s);
        if (cgen_Memmgr == 1) {
            emit_addiu(A1, SP, 4 * (idx + 1), s);
            emit_jal("_GenGC_Assign", s);
        }
    } else if ((idx = env.LookUpParam(name)) != -1) {
        s << "\t# It is a param." << endl;
        emit_load(ACC, idx + 3, FP, s);
        if (cgen_Memmgr == 1) {
            emit_addiu(A1, FP, 4 * (idx + 3), s);
            emit_jal("_GenGC_Assign", s);
        }
    } else if ((idx = env.LookUpAttrib(name)) != -1) {
        s << "\t# It is an attribute." << endl;
        emit_load(ACC, idx + 3, SELF, s);
        if (cgen_Memmgr == 1) {
            emit_addiu(A1, SELF, 4 * (idx + 3), s);
            emit_jal("_GenGC_Assign", s);
        }
    } else if (name == self) {
        s << "\t# It is self." << endl;
        emit_move(ACC, SELF, s);
    } else {
        s << "Error! object class" << endl;
    }

    s << endl;
}
```

assign_class::code 本质上是一个 store 语句，将其 expr 部分先代码生成，按上述方法依次从 let 变量、参数、属性中寻找其左值的地址，然后向对应地址执行 store。

```
void assign_class::code(ostream& s, Environment env) {
    s << "\t# Assign. First eval the expr." << endl;
    expr->code(s, env);

    s << "\t# Now find the lvalue." << endl;
    int idx;

    if ((idx = env.LookUpVar(name)) != -1) {
        s << "\t# It is a let variable." << endl;
        emit_store(ACC, idx + 1, SP, s);
        if (cgen_Memmgr == 1) {
            emit_addiu(A1, SP, 4 * (idx + 1), s);
            emit_jal("_GenGC_Assign", s);
        }
    } else if ((idx = env.LookUpParam(name)) != -1){
```

static_dispatch 是一次静态的函数调用，我们为其每个表达式生成代码，将结果按次序存储到栈中。

```
void static_dispatch_class::code(ostream& s, Environment env) {
    s << "\t# Static dispatch. First eval and save the params." << endl;

    std::vector<Expression> actuals = GetActuals();
    Environment new_env = env;
    for (Expression expr : actuals) {
        expr->code(s, env);
        emit_push(ACC, s);
        env.EnterScope();
        env.AddObstacle();
    }
```

然后计算其调用的对象，并定义一个 LABEL。如果算出来的结果不对应已知的任何类，会将其赋值为 0。如果不是 0，则 bne 后 jump 到 LABEL 后正常逻辑的执行，否则顺序执行触发 ABORT。

```
s << "\t# eval the obj in dispatch." << endl;
expr->code(s, env);

s << "\t# if obj = void: abort" << endl;
emit_bne(ACC, ZERO, labelnum, s);
s << LA << ACC << " str_const0" << endl;
emit_load_imm(T1, 1, s);
emit_jal("_dispatch_abort", s);

emit_label_def(labelnum, s);
++labelnum;
```

通过 get_class_node，我们可以找到对象的类别对应的节点，然后定位到该方法。我们在该类别对应的 dispatch 表中，找到该 dispatch 的下标，以获得需要调用的函数的地址。然后可以生成一条 jalr 的指令，跳转到该地址。

```
Symbol _class_name = type_name;
CgenNode* _class_node = codegen_classtable->get_class_node(type_name);
s << "\t# Now we locate the method in the dispatch table." << endl;
s << "\t# t1 = " << type_name << ".dispTab" << endl;

std::string addr = type_name->get_string();
addr += DISPTAB_SUFFIX;
emit_load_address(T1, addr.c_str(), s);

s << endl;

int idx = _class_node->get_dispatch_idx_table()[name];
s << "\t# t1 = dispTab[offset]" << endl;
emit_load(T1, idx, T1, s);
s << endl;

s << "\t# jumpto " << name << endl;
emit_jalr(T1, s);
s << endl;
```

而 dispatch_class::code()与 static 的差别仅在于，其_class_name 为表达式 expr 的类别，而不是静态指定的类别。

```
if (expr->get_type() != SELF_TYPE) {
    _class_name = expr->get_type();
}
```

cond_class::code()要生成条件语句。首先，为计算条件的值生成代码。根据上计算机体系结构时的经验，我选择使用一条 beq 的语句，使用 false 和 finish 两个 label。当条件满足时，顺序执行（then 后的语句 then_exp），然后 jump 到 finish；当条件不满足时，jump 到 false（else 后的语句 else_exp），然后执行到 finish。

```
void cond_class::code(ostream& s, Environment env) {
    s << "\t# If statement. First eval condition." << endl;
    pred->code(s, env);

    s << "\t# extract the bool content from acc to t1" << endl;
    emit_fetch_int(T1, ACC, s);
    s << endl;

    int labelnum_false = labelnum++;
    int labelnum_finish = labelnum++;
    // labelnum : false.
    // labelnum + 1: finish
    s << "\t# if t1 == 0 goto false" << endl;
    emit_beq(T1, ZERO, labelnum_false, s);
    s << endl;

    then_exp->code(s, env);

    s << "\t# jumpt finish" << endl;
    emit_branch(labelnum_finish, s);
    s << endl;

    s << "# False:" << endl;
    emit_label_def(labelnum_false, s);

    else_exp->code(s, env);

    s << "# Finish:" << endl;
    emit_label_def(labelnum_finish, s);

}
```

循环语句的代码生成 loop_class::code()和条件语句比较类似。第一个 label 为 start，第二个 label 为 finish。start 后为条件语句 pred，为其生成代码。当条件成立时，进行循环；当条件不成立时，跳转到 finish，退出循环。每当运行完循环体，就会跳转到 start，重新进行判断。

```
void loop_class::code(ostream& s, Environment env) {
    int start = labelnum;
    int finish = labelnum + 1;
    labelnum += 2;

    s << "\t# While loop" << endl;
    s << "\t# start:" << endl;
    emit_label_def(start, s);

    s << "\t# ACC = pred" << endl;
    pred->code(s, env);

    s << "\t# extract int inside bool" << endl;
    emit_fetch_int(T1, ACC, s);
    s << endl;

    s << "\t# if pred == false jumpto finish" << endl;
    emit_beq(T1, ZERO, finish, s);
    s << endl;

    body->code(s, env);

    s << "\t# Jumpto start" << endl;
    emit_branch(start, s);

    s << "\t# Finish:" << endl;
    emit_label_def(finish, s);

    s << "\t# ACC = void" << endl;
    emit_move(ACC, ZERO, s);

}
```

typcase_class::node()比较复杂。首先，我们获得所有类的节点，然后计算表达式 expr。

如果表达式的类型为 void，就跳转到 abort。

```
void typcase_class::code(ostream& s, Environment env) {
    std::map<Symbol, int> _class_tags = codegen_classtable->get_class_tags();
    std::vector<CgenNode*> _class_nodes = codegen_classtable->get_class_nodes();

    s << "\t# case expr" << endl;
    s << "\t# First eval e0" << endl;
    expr->code(s, env);

    s << "\t# If e0 = void, abort" << endl;
    emit_bne(ACC, ZERO, labelnum, s);
    emit_load_address(ACC, "str_const0", s);
    emit_load_imm(T1, 1, s);
    emit_jal("_case_abort2", s);

    emit_label_def(labelnum, s);
    ++labelnum;
```

将 finish 的标签定义在最后。

```
    s << "#finish:" << endl;
    emit_label_def(finish, s);
    s << endl;
```

为每个标签生成代码，在执行完该标签的代码后，jump 到 finish。

```
for (branch_class* _case : _cases) {
    Symbol _name = _case->name;
    Symbol _type_decl = _case->type_decl;
    Expression _expr = _case->expr;

    s << "# eval expr " << caseidx << endl;
    emit_label_def(labelbeg + caseidx, s);
    env.EnterScope();
    env.AddVar(_name);
    emit_push(ACC, s);
    _expr->code(s, env);
    emit_addiu(SP, SP, 4, s);

    s << "\t# Jumpto finish" << endl;
    emit_branch(finish, s);
    ++caseidx;
}
```

block_class::code()比较简单，对 block 内的语句依次进行代码生成即可。

```
void block_class::code(ostream& s, Environment env) {
    for (int i = body->first(); body->more(i); i = body->next(i)) {
        body->nth(i)->code(s, env);
    }
}
```

let 语句先为初始化表达式生成代码。如果初始化表达式为空，且声明类型为基本类型

str、int 或 bool，则调用对应的 load 函数。之后先 push 将 let 变量加入栈中，再生成函数体的代码，然后再进行 pop。

```
void let_class::code(ostream& s, Environment env) {
    s << "\t# Let expr" << endl;
    s << "\t# First eval init" << endl;
    init->code(s, env);

    if (init->IsEmpty()) {
        // We still need to deal with basic types.
        if (type_decl == Str) {
            emit_load_string(ACC, stringtable.lookup_string(""), s);
        } else if (type_decl == Int) {
            emit_load_int(ACC, inttable.lookup_string("0"), s);
        } else if (type_decl == Bool) {
            emit_load_bool(ACC, BoolConst(0), s);
        }
    }

    s << "\t# push" << endl;
    emit_push(ACC, s);
    s << endl;

    env.EnterScope();
    env.AddVar(identifier);

    body->code(s, env);

    s << "\t# pop" << endl;
    emit_addiu(SP, SP, 4, s);
    s << endl;
}
```

加减乘除运算的 code()较为一致。首先计算表达式 e1 和 e2，分别 push 入栈中。然后将 e1 和 e2 分别 pop 到 t1 和 t2，执行相应地计算并将值存储到 t2。

```
void divide_class::code(ostream& s, Environment env) {
    s << "\t# Int operation : Div" << endl;
    s << "\t# First eval e1 and push." << endl;
    e1->code(s, env);
    emit_push(ACC, s);
    env.AddObstacle();
    s << endl;

    s << "\t# Then eval e2 and make a copy for result." << endl;
    e2->code(s, env);
    emit_jal("Object.copy", s);
    s << endl;

    s << "\t# Let's pop e1 to t1, move e2 to t2" << endl;
    emit_addiu(SP, SP, 4, s);
    emit_load(T1, 0, SP, s);
    emit_move(T2, ACC, s);
    s << endl;

    s << "\t# Extract the int inside the object." << endl;
    emit_load(T1, 3, T1, s);
    emit_load(T2, 3, T2, s);
    s << endl;

    s << "\t# Modify the int inside t2." << endl;
    emit_div(T3, T1, T2, s);
    emit_store(T3, 3, ACC, s);
    s << endl;

}
```

lt、eq、leq 三个比较和四则运算也类似，分别计算 e1 和 e2，然后读入 t1 和 t2，进行比较，将结果存入 ACC 中，并定义一个 label。

```
void lt_class::code(ostream& s, Environment env) {
    s << "\t# Int operation : Less than" << endl;
    s << "\t# First eval e1 and push." << endl;
    e1->code(s, env);
    emit_push(ACC, s);
    env.AddObstacle();
    s << endl;

    s << "\t# Then eval e2." << endl;
    e2->code(s, env);
    s << endl;

    s << "\t# Let's pop e1 to t1, move e2 to t2" << endl;
    emit_addiu(SP, SP, 4, s);
    emit_load(T1, 0, SP, s);
    emit_move(T2, ACC, s);
    s << endl;

    s << "\t# Extract the int inside the object." << endl;
    emit_load(T1, 3, T1, s);
    emit_load(T2, 3, T2, s);
    s << endl;

    s << "\t# Pretend that t1 < t2" << endl;
    emit_load_bool(ACC, BoolConst(1), s);
    s << "\t# If t1 < t2 jumpto finish" << endl;
    emit_blt(T1, T2, labelnum, s);

    emit_load_bool(ACC, BoolConst(0), s);
    emit_label_def(labelnum, s);

    ++labelnum;
}
```

取负的 node 和判定是否为 0 的 cond 则是单目运算，可以理解为单目的四则运算和比较操作。不再详细介绍。

new 指令的代码生成涉及堆内存的分配，需要调用 object.copy()。每个类型，都必须有一个原型，存在于 Data 这一区域。我们需要符合原型的规范。在为 new 生成代码时，首先调用 Object.copy()分配空间，其大小正好是一个该类型原型的大小，然后调用该类型的 init 方法进行初始化。需要注意的是，new 的参数可能是 SELT_TYPE。需要进行一定的处理。

```
    std::string dest = type_name->get_string();
    dest += PROTOBJ_SUFFIX;
    emit_load_address(ACC, dest.c_str(), s);
    emit_jal("Object.copy", s);
    dest = type_name->get_string();
    dest += CLASSINIT_SUFFIX;
    emit_jal(dest.c_str(), s);
}
```

is_void 语句的代码生成核心是 BEQ。首先假设 t1 就是 void，将 1 存入 ACC。如果 t1 是 void，就 jump 到 finish。ACC 保持是 1。如果 t1 不是 void，就会执行到将 0 存入 ACC。

```
void isvoid_class::code(ostream& s, Environment env) {
    e1->code(s, env);

    s << "\t# t1 = acc" << endl;
    emit_move(T1, ACC, s);

    s << "\t# First pretend t1 = void: acc = bool(1)" << endl;
    emit_load_bool(ACC, BoolConst(1), s);

    s << "\t# if t1 = void: jumpto finish" << endl;
    emit_beq(T1, ZERO, labelnum, s);
    s << endl;

    s << "\t# acc != void" << endl;
    emit_load_bool(ACC, BoolConst(0), s);

    s << "# finish:" << endl;
    emit_label_def(labelnum, s);

    ++labelnum;
}
```

空语句的 code 也是 trivial 的。

```
void no_expr_class::code(ostream& s, Environment env) {
    emit_move(ACC, ZERO, s);
}
```

四、整体代码生成

代码生成部分通过 CgenClassTable::code()来实现。首先，我们解读一下已经提供给我们的 3 个函数：code_global_data()、code_select_gc()以及 code_constants()。

```
void CgenClassTable::code()
{
  if (cgen_debug) cout << "coding global data" << endl;
  code_global_data();

  if (cgen_debug) cout << "choosing gc" << endl;
  code_select_gc();

  if (cgen_debug) cout << "coding constants" << endl;
  code_constants();

//                 Add your code to emit
//                   - prototype objects
//                   - class_nameTab
//                   - dispatch tables
//
```

code_global_data()做的事情是比较 trivial 的，其效果可见标准生成器给出的结果。

这一步用到的宏定义在 emit.h 中，对比即可了解。

```
1      .data
2      .align  2
3      .globl  class_nameTab
4      .globl  Main_protObj
5      .globl  Int_protObj
6      .globl  String_protObj
7      .globl  bool_const0
8      .globl  bool_const1
9      .globl  _int_tag
10     .globl  _bool_tag
11     .globl  _string_tag
12 _int_tag:
13     .word   2
14 _bool_tag:
15     .word   3
16 _string_tag:
17     .word   4
```

cgen.h | cool-tree.h | cgen.cc | emit.h | examp

```
22  #define CLASSNAMETAB          "class_nameTab"
23  #define CLASSOBJTAB           "class_objTab"
24  #define INTTAG                "_int_tag"
25  #define BOOLTAG               "_bool_tag"
26  #define STRINGTAG             "_string_tag"
27  #define HEAP_START            "heap_start"
28
29  // Naming conventions
30  #define DISPTAB_SUFFIX        "_dispTab"
31  #define METHOD_SEP            "."
32  #define CLASSINIT_SUFFIX      "_init"
33  #define PROTOBJ_SUFFIX        "_protObj"
34  #define OBJECTPROTOBJ         "Object"PROTOBJ_SUFFIX
35  #define INTCONST_PREFIX       "int_const"
36  #define STRCONST_PREFIX       "str_const"
37  #define BOOLCONST_PREFIX      "bool_const"
```

code_select_gc 产生的代码如下：

```
18      .globl  _MemMgr_INITIALIZER
19  _MemMgr_INITIALIZER:
20      .word   _NoGC_Init
21      .globl  _MemMgr_COLLECTOR
22  _MemMgr_COLLECTOR:
23      .word   _NoGC_Collect
24      .globl  _MemMgr_TEST
25  _MemMgr_TEST:
26      .word   0
```

code_constants()会对 stringtable 和 inttable 中的每个常量生成如下代码。

```
str_const6:
    .word   4
    .word   6
    .word   String_dispTab
    .word   int_const2
    .ascii  "String"
    .byte   0
    .align  2
    .word   -1
```

```
int_const4:
    .word   2
    .word   4
    .word   Int_dispTab
    .word   2
    .word   -1
```

接下来，为了完成代码生成这个过程，介绍我们添加的几个函数。

```
if (cgen_debug) cout << "coding name table" << endl;
code_class_nameTab();

if (cgen_debug) cout << "coding object table" << endl;
code_class_objTab();

if (cgen_debug) cout << "coding dispatch tables" << endl;
code_dispatchTabs();

if (cgen_debug) cout << "coding prototype objects" << endl;
code_protObjs();
```

```
if (cgen_debug) cout << "coding object initializers" << endl;
code_class_inits();

if (cgen_debug) cout << "coding class methods" << endl;
code_class_methods();
```

首先使用 code_class_nameTab 函数，为类的名称表生成代码。通过 GetClassNodes() 函数，可以得到所有类别的节点。然后遍历所有的 CgenNode，为每一个类别名查找其在 stringtable 中的 entry，然后生成相关的代码。代码中还包含其子类的注释信息。这一部分是 cool 要求固定在汇编码中，以便运行时系统查找需要。

```
void CgenClassTable::code_class_nameTab() {
    str << CLASSNAMETAB << LABEL;

    std::vector<CgenNode*> class_nodes = GetClassNodes();
    for (CgenNode* class_node : class_nodes) {
        Symbol class_name = class_node->name;
        StringEntry* str_entry = stringtable.lookup_string(class_name->get_string());

        str << WORD;
        str_entry->code_ref(str);
        str << endl;
        std::vector<CgenNode*> _children = class_node->GetChildren();
        for (CgenNode* _child : _children) {
            str << "# child: " << _child->name << endl;
        }
        str << std::endl;
    }
}
```

```
class_nameTab:
    .word   str_const5
# child: IO
# child: Int
# child: Bool
# child: String
# child: Main
    .word   str_const6
    .word   str_const7
    .word   str_const8
    .word   str_const9
    .word   str_const10
```

然后调用 code_class_objTab，为类的对象表生成代码。具体细节和 code_class_table 类似。

```cpp
void CgenClassTable::code_class_objTab() {
    str << CLASSOBJTAB << LABEL;

    std::vector<CgenNode*> class_nodes = GetClassNodes();
    for (CgenNode* class_node : class_nodes) {
        Symbol class_name = class_node->name;
        StringEntry* str_entry = stringtable.lookup_string(clas

        str << WORD;
        emit_protobj_ref(str_entry, str);
        str << endl;
        str << WORD;
        emit_init_ref(str_entry, str);
        str << endl;
    }
}
```

```
class_objTab:
    .word   Object_protObj
    .word   Object_init
    .word   IO_protObj
    .word   IO_init
    .word   Int_protObj
    .word   Int_init
    .word   Bool_protObj
    .word   Bool_init
    .word   String_protObj
    .word   String_init
    .word   Main_protObj
    .word   Main_init
```

code_dispatchTabs 函数为 dispatch 表生成代码。该表类似于 C++中的虚表，指明每个类下可以调用哪些方法，对于多态的方法具体调用哪一个，涉及动态的类型检查。该表在运行时不会被使用到，但在编译的时候很重要。其中，get_dispatch_class_tab 和 get_dispatch_idx_table 都是访问 get_full_methods 函数中得到的副产物 dispatch_class_tab 和 dispatch_idx_tab，分别是方法到类名的 map 和方法到类内向量中所在的下标的 map，也就是说可以知道该方法是在哪个（父）类定义的，是第几个方法，这样就可以为他生成代码。

```cpp
void CgenClassTable::code_dispatchTabs() {
    std::vector<CgenNode*> class_nodes = get_class_nodes();

    for (CgenNode* _class_node : class_nodes) {
        emit_disptable_ref(_class_node->name, str);
        str << LABEL;
        std::vector<method_class*> full_methods = _class_node->get_full_methods();
        std::map<Symbol, Symbol> dispatch_class_tab = _class_node->get_dispatch_class_tab
        std::map<Symbol, int> dispatch_idx_tab = _class_node->get_dispatch_idx_table();
        for (method_class* _method : full_methods) {
            Symbol _method_name = _method->name;
            Symbol _class_name = dispatch_class_tab[_method_name];
            int _idx = dispatch_idx_tab[_method_name];
            str << "\t# method # " << _idx << endl;
            str << WORD;
            emit_method_ref(_class_name, _method_name, str);
            str << endl;
        }
    }
}
```

生成的 dispatch table 形如：

```
Object_dispTab:
    .word   Object.abort
    .word   Object.type_name
    .word   Object.copy
```

```
Bool_dispTab:
    .word   Object.abort
    .word   Object.type_name
    .word   Object.copy
```

code_protObjs 为每个类的原型对象生成代码，即对每个类结点调用 code_protObj 函数。每个类都有一个原型。

```
void CgenClassTable::code_protObjs() {
    std::vector<CgenNode*> class_nodes = get_class_nodes();
    for (CgenNode* class_node : class_nodes) {
        class_node->code_protObj(str);
    }
}
```

参考标准的生成器给出的格式，实现原型对象生成的代码。调用的方法基本是提供好的 emit 函数，只要对参考标准输出，对汇编有一定了解，就可以写出来。

```
    .word   -1
Object_protObj:
    .word   0
    .word   3
    .word   Object_dispTab
```

```
    .word   -1
Main_protObj:
    .word   5
    .word   3
    .word   Main_dispTab
```

```
void CgenNode::code_protObj(ostream& s) {
    std::vector<attr_class*> attribs = get_full_attributes();

    s << WORD << "-1" << endl;
    s << get_name() << PROTOBJ_SUFFIX << LABEL;
    s << WORD << class_tag << "\t# class tag" << endl;
    s << WORD << (DEFAULT_OBJFIELDS + attribs.size()) << "\t# size" << endl;
    s << WORD << get_name() << DISPTAB_SUFFIX << endl;
```

```
for (int i = 0; i < attribs.size(); ++i) {
    if (attribs[i]->name == val) { // _val
        if (get_name() == Str) {
            s << WORD;
            inttable.lookup_string("0")->code_ref(s);
            s << "\t# int(0)";
            s << endl;
        } else {
            s << WORD << "0\t# val(0)" << endl;
        }
    } else if (attribs[i]->name == str_field) { // _str_field
        s << WORD << "0\t# str(0)" << endl;
    } else { // normal attribute.
        Symbol type = attribs[i]->type_decl;
        if (type == Int) {
            s << WORD;
            inttable.lookup_string("0")->code_ref(s);
            s << "\t# int(0)";
            s << endl;
        } else if (type == Bool) {
            s << WORD;
            falsebool.code_ref(s);
            s << "\t# bool(0)";
            s << endl;
        } else if (type == Str) {
            s << WORD;
            stringtable.lookup_string("")->code_ref(s);
            s << "\t# str()";
            s << endl;
        } else {
            s << WORD;
            s << "0\t# void" << endl;
        }
    }
}
```

接下来调用 code_global_text()函数，该函数也是已经提供给我们的。这部分开始定义堆空间的起始位置，以及各基本类型的 init 方法和 Main 的 main 方法声明。

```
void CgenClassTable::code_global_text()
{
  str << GLOBAL << HEAP_START << endl
      << HEAP_START << LABEL
      << WORD << 0 << endl
      << "\t.text" << endl
      << GLOBAL;
  emit_init_ref(idtable.add_string("Main"), str);
  str << endl << GLOBAL;
  emit_init_ref(idtable.add_string("Int"),str);
  str << endl << GLOBAL;
  emit_init_ref(idtable.add_string("String"),str);
  str << endl << GLOBAL;
  emit_init_ref(idtable.add_string("Bool"),str);
  str << endl << GLOBAL;
  emit_method_ref(idtable.add_string("Main"), idtable.add_string("main"), str);
  str << endl;
}
```

```
heap_start:
    .word   0
    .text
    .globl  Main_init
    .globl  Int_init
    .globl  String_init
    .globl  Bool_init
    .globl  Main.main
```

code_class_inits()函数为类别的初始化生成代码。

```
void CgenClassTable::code_class_inits() {
    std::vector<CgenNode*> class_nodes = get_class_nodes();
    for (CgenNode* class_node : class_nodes) {
        class_node->code_init(str);
    }
}
```

首先，被调用者保存寄存器，创建栈帧，然后，调用父类的初始化，对属性赋予初值，再进行寄存器和栈帧的恢复，返回分配的地址。

```
Bool_init:
    # push fp, s0, ra
    addiu   $sp $sp -12
    sw  $fp 12($sp)
    sw  $s0 8($sp)
    sw  $ra 4($sp)

    # fp now points to the return addr in stack
    addiu   $fp $sp 4

    # SELF = a0
    move    $s0 $a0

    # init parent
    jal Object_init

    # init attrib _val
    # ret = SELF
    move    $a0 $s0

    # pop fp, s0, ra
    lw  $fp 12($sp)
    lw  $s0 8($sp)
    lw  $ra 4($sp)
    addiu   $sp $sp 12

    # return
    jr  $ra
```

按照这个流程写代码即可。只要掌握了 inits 函数要干什么、emit 函数如何调用，就能写出来。

```
void CgenNode::code_init(ostream& s) {
    s << get_name();
    s << CLASSINIT_SUFFIX;
    s << LABEL;
    s << "\t# push fp, s0, ra" << endl;
    emit_addiu(SP, SP, -12, s);
    emit_store(FP, 3, SP, s);
    emit_store(SELF, 2, SP, s);
    emit_store(RA, 1, SP, s);
    s << endl;

    s << "\t# fp now points to the return addr in stack" << endl;
    emit_addiu(FP, SP, 4, s);
    s << endl;

    s << "\t# SELF = a0" << endl;
    emit_move(SELF, ACC, s);
    s << endl;
```

最后调用 code_class_methods()函数为非基本类的方法生成代码。

```
void CgenClassTable::code_class_methods() {
    std::vector<CgenNode*> class_nodes = get_class_nodes();
    for (CgenNode* class_node : class_nodes) {
        if (!class_node->basic()) {
            class_node->code_methods(str);
        }
    }
}
```

code_methods 将依次调用各 method 的 code 方法，即第三部分中已经介绍的代码生成。

```
void CgenNode::code_methods(ostream& s) {
    std::vector<method_class*> methods = get_methods();
    for (method_class* method : methods) {
        method->code(s, this);
    }
}
```

五、结果

make dotest

```
cool-tree.h   cgen.cc   example.s
496     jr  $ra
497
498 Main.main:
499     # push fp, s0, ra
500     addiu   $sp $sp -12
501     sw  $fp 12($sp)
502     sw  $s0 8($sp)
503     sw  $ra 4($sp)
504
505     # fp now points to the return addr in st
506     addiu   $fp $sp 4
507
508     # SELF = a0
509     move    $s0 $a0
510
511     # evaluating expression and put it to AC
512     la  $a0 int_const0
513
514     # pop fp, s0, ra
515     lw  $fp 12($sp)
516     lw  $s0 8($sp)
517     lw  $ra 4($sp)
518     addiu   $sp $sp 12
519
520     # Pop arguments
521     addiu   $sp $sp 0
522
523     # return
524     jr  $ra
525
```

/usr/class/cs143/bin/spim example.s

```
root@gdmGPU3:/usr/class/cs143/assignments/PA5# /usr/class/cs143/bin/spim example.s
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ../lib/trap.handler
COOL program successfully executed
```

六、感想

PA4 好难，我自闭了。这个项目需要对 cool、MIPS 都有很深的理解，需要结合标准生成器的输出来进行分析，而且需要自己完成的地方很多，不像 PA1、PA2 比较 strateforward，在设计上有很多考量。直接上手太过困难，还是需要对相关知识在事前有充分了解，并参考

一下别人的做法。