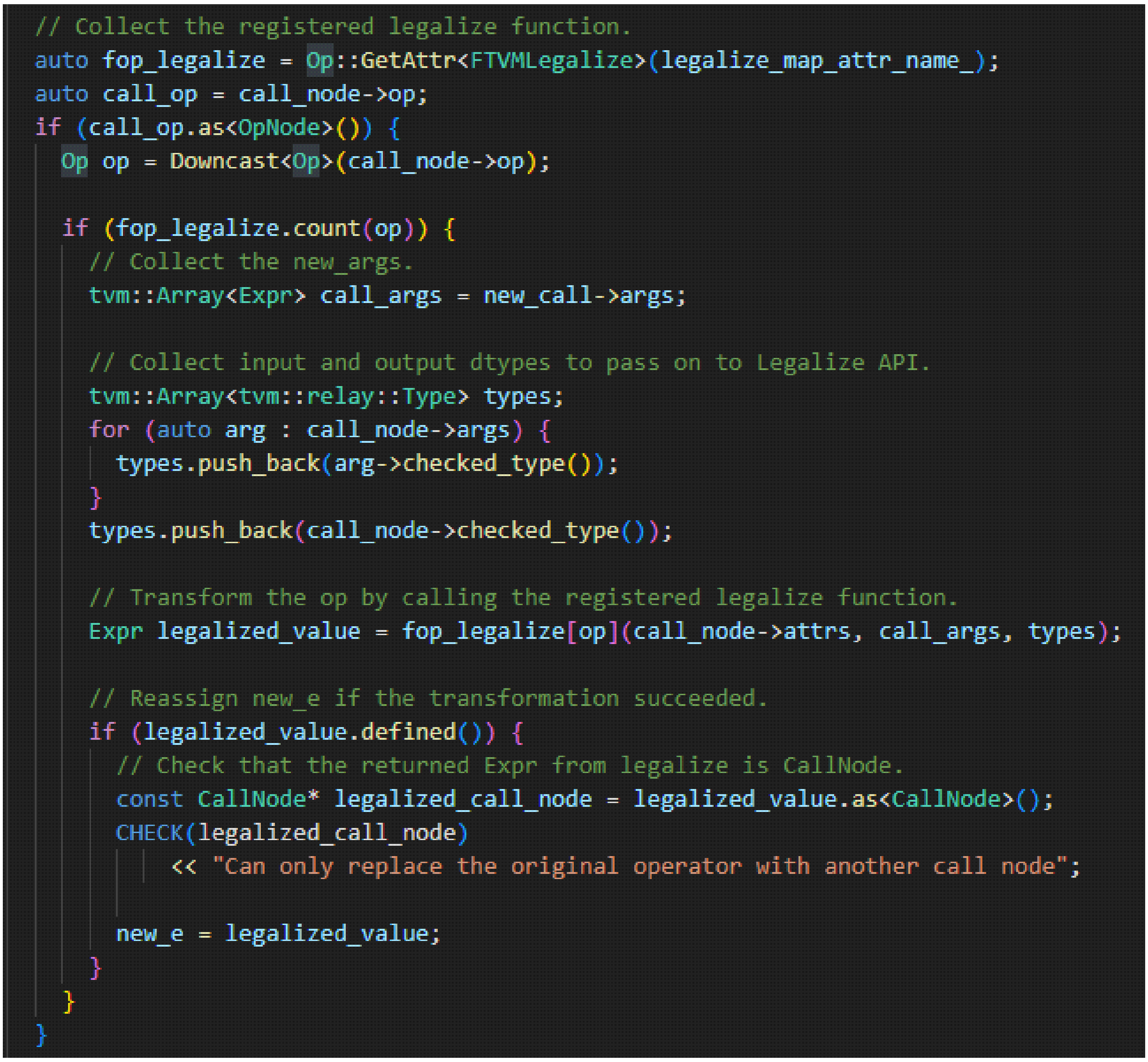
# Relay Pass

2022年10月9日 11:15

Legalize()

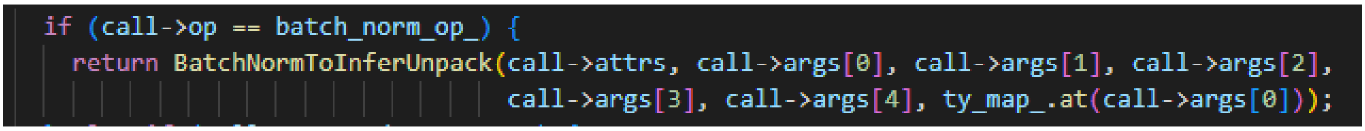
等价转换，将某些算子转换为等价的relay op



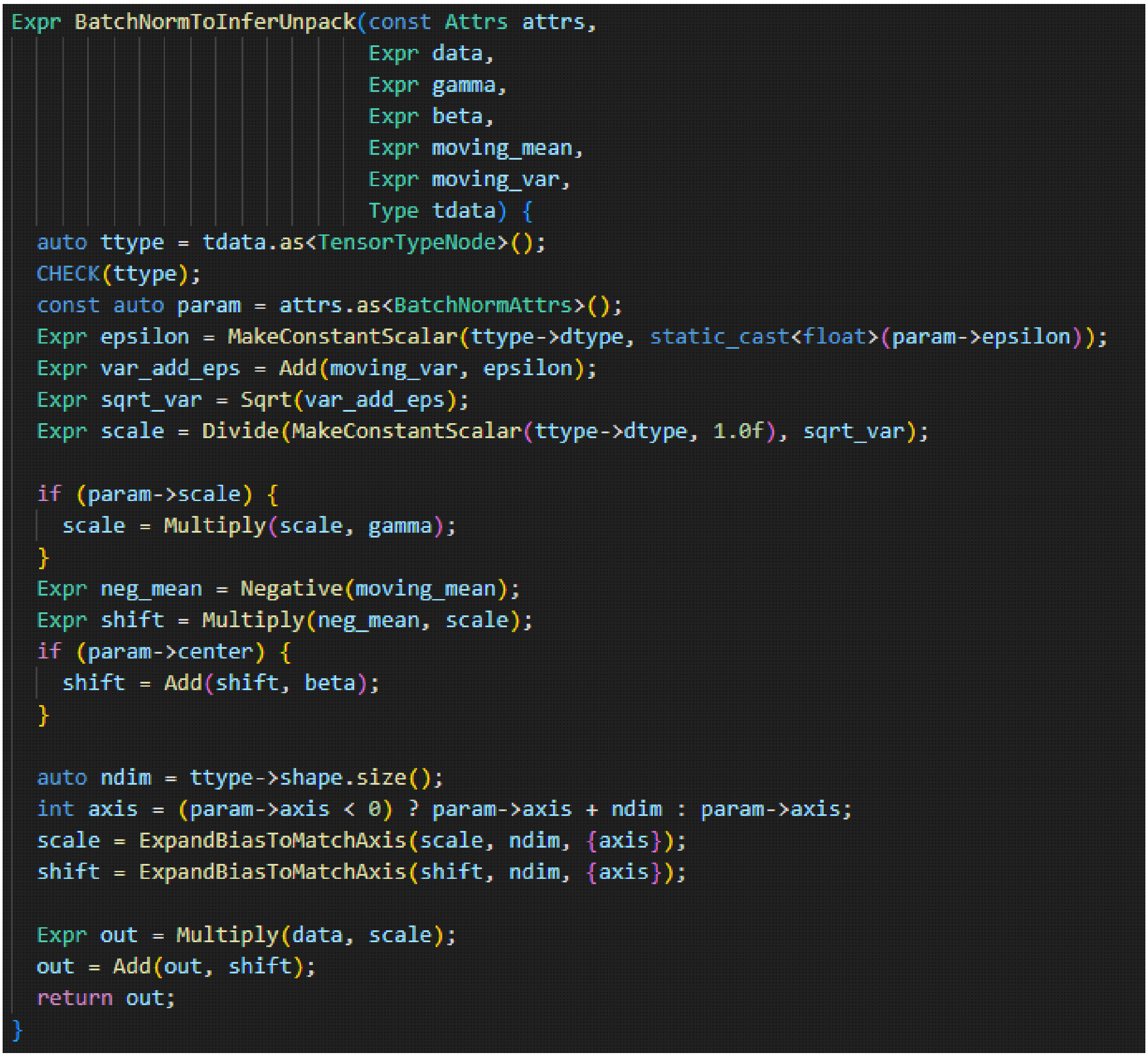
```
// Collect the registered legalize function.
auto fop_legalize = Op::GetAttr<FTVMLegalize>(legalize_map_attr_name_);
auto call_op = call_node->op;
if (call_op.as<OpNode>()) {
  Op op = Downcast<Op>(call_node->op);

  if (fop_legalize.count(op)) {
    // Collect the new_args.
    tvm::Array<Expr> call_args = new_call->args;

    // Collect input and output dtypes to pass on to Legalize API.
    tvm::Array<tvm::relay::Type> types;
    for (auto arg : call_node->args) {
      types.push_back(arg->checked_type());
    }
    types.push_back(call_node->checked_type());

    // Transform the op by calling the registered legalize function.
    Expr legalized_value = fop_legalize[op](call_node->attrs, call_args, types);

    // Reassign new_e if the transformation succeeded.
    if (legalized_value.defined()) {
      // Check that the returned Expr from legalize is CallNode.
      const CallNode* legalized_call_node = legalized_value.as<CallNode>();
      CHECK(legalized_call_node)
          << "Can only replace the original operator with another call node";

      new_e = legalized_value;
    }
  }
}
```

SimplifyInference()

将BN Dropout Instance_norm Layer_norm算子拆分，便于后续优化（和其他算子融合）

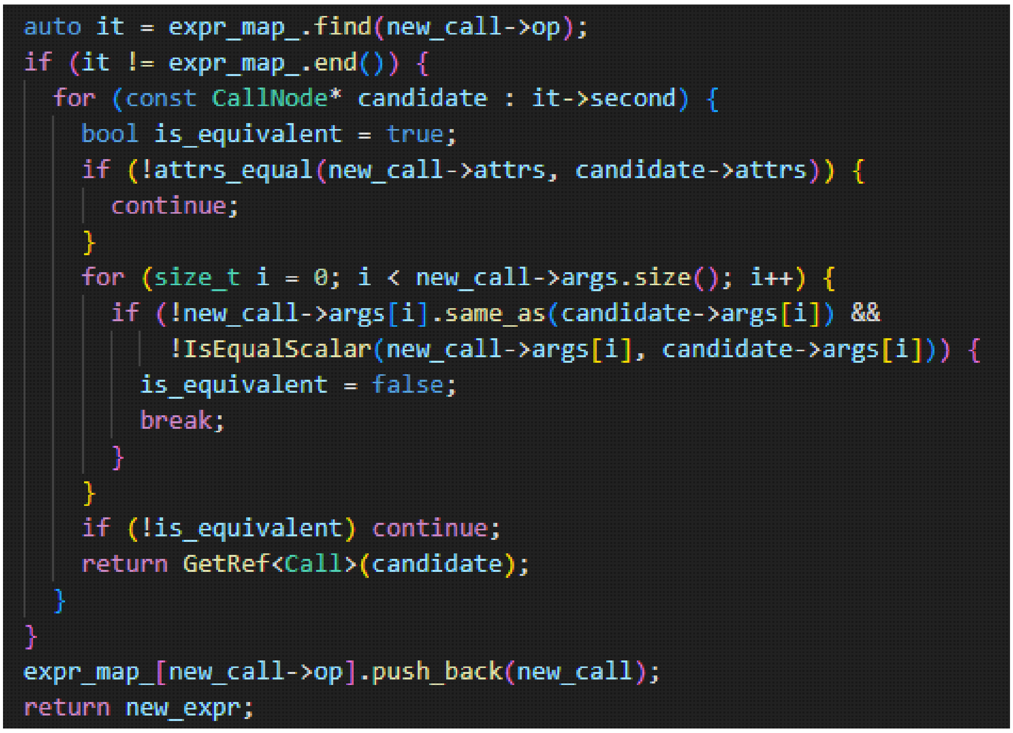


```
if (call->op == batch_norm_op_) {
  return BatchNormToInferUnpack(call->attrs, call->args[0], call->args[1], call->args[2],
                                call->args[3], call->args[4], ty_map_.at(call->args[0]));
```



```
Expr BatchNormToInferUnpack(const Attrs attrs,
                            Expr data,
                            Expr gamma,
                            Expr beta,
                            Expr moving_mean,
                            Expr moving_var,
                            Type tdata) {
  auto ttype = tdata.as<TensorTypeNode>();
  CHECK(ttype);
  const auto param = attrs.as<BatchNormAttrs>();
  Expr epsilon = MakeConstantScalar(ttype->dtype, static_cast<float>(param->epsilon));
  Expr var_add_eps = Add(moving_var, epsilon);
  Expr sqrt_var = Sqrt(var_add_eps);
  Expr scale = Divide(MakeConstantScalar(ttype->dtype, 1.0f), sqrt_var);

  if (param->scale) {
    scale = Multiply(scale, gamma);
  }
  Expr neg_mean = Negative(moving_mean);
  Expr shift = Multiply(neg_mean, scale);
  if (param->center) {
    shift = Add(shift, beta);
  }

  auto ndim = ttype->shape.size();
  int axis = (param->axis < 0) ? param->axis + ndim : param->axis;
  scale = ExpandBiasToMatchAxis(scale, ndim, {axis});
  shift = ExpandBiasToMatchAxis(shift, ndim, {axis});

  Expr out = Multiply(data, scale);
  out = Add(out, shift);
  return out;
}
```

EliminateCommonSubexpr()
消除公共子表达式，即op、args和args顺序均相同的表达式，如
a=b+c
d=b+c

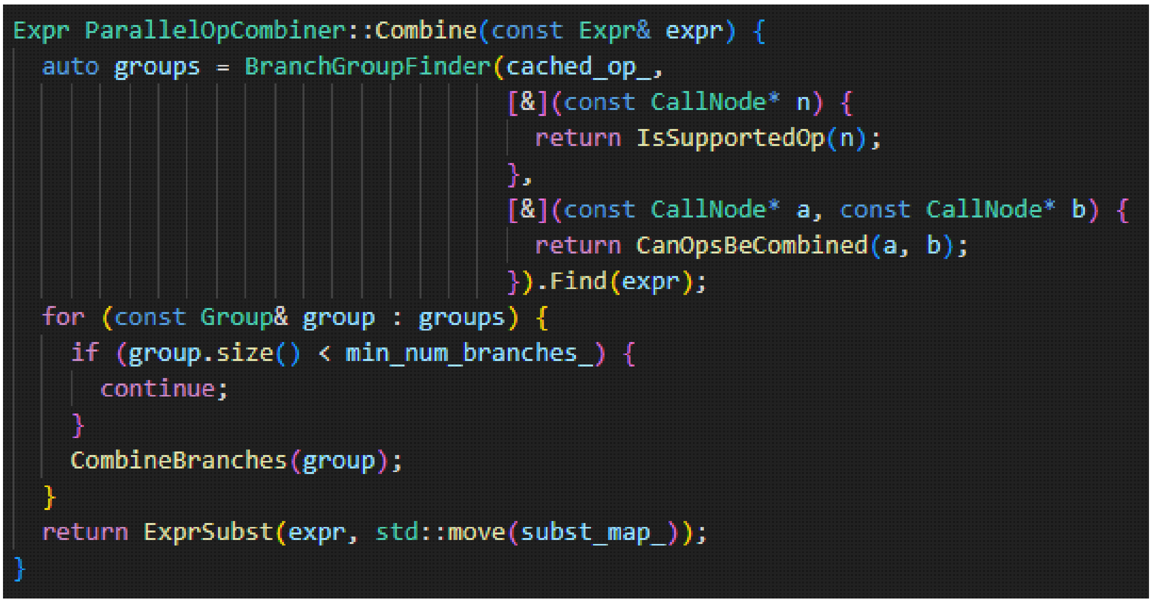


```
auto it = expr_map_.find(new_call->op);
if (it != expr_map_.end()) {
  for (const CallNode* candidate : it->second) {
    bool is_equivalent = true;
    if (!attrs_equal(new_call->attrs, candidate->attrs)) {
      continue;
    }
    for (size_t i = 0; i < new_call->args.size(); i++) {
      if (!new_call->args[i].same_as(candidate->args[i]) &&
          !IsEqualScalar(new_call->args[i], candidate->args[i])) {
        is_equivalent = false;
        break;
      }
    }
    if (!is_equivalent) continue;
    return GetRef<Call>(candidate);
  }
}
expr_map_[new_call->op].push_back(new_call);
return new_expr;
```

CombineParallelConv2D(min_num_branches)

合并并行的Conv2D运算，如batch间

min_num_branches：最小的合并数目



```
Expr ParallelOpCombiner::Combine(const Expr& expr) {
  auto groups = BranchGroupFinder(cached_op_,
                                  [&](const CallNode* n) {
                                    return IsSupportedOp(n);
                                  },
                                  [&](const CallNode* a, const CallNode* b) {
                                    return CanOpsBeCombined(a, b);
                                  }).Find(expr);
  for (const Group& group : groups) {
    if (group.size() < min_num_branches_) {
      continue;
    }
    CombineBranches(group);
  }
  return ExprSubst(expr, std::move(subst_map_));
}
```

CombineParallelDense(min_num_branches)

合并并行的Dense运算

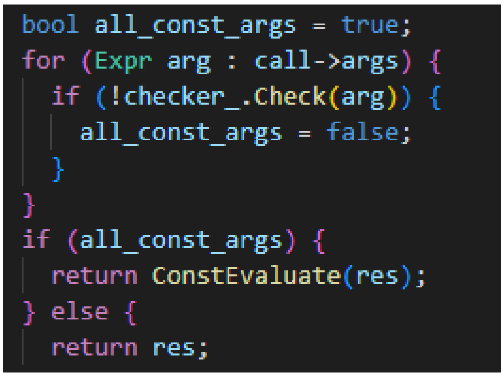
min_num_branches：最小的合并数目

与CombineParallelConv2D类似

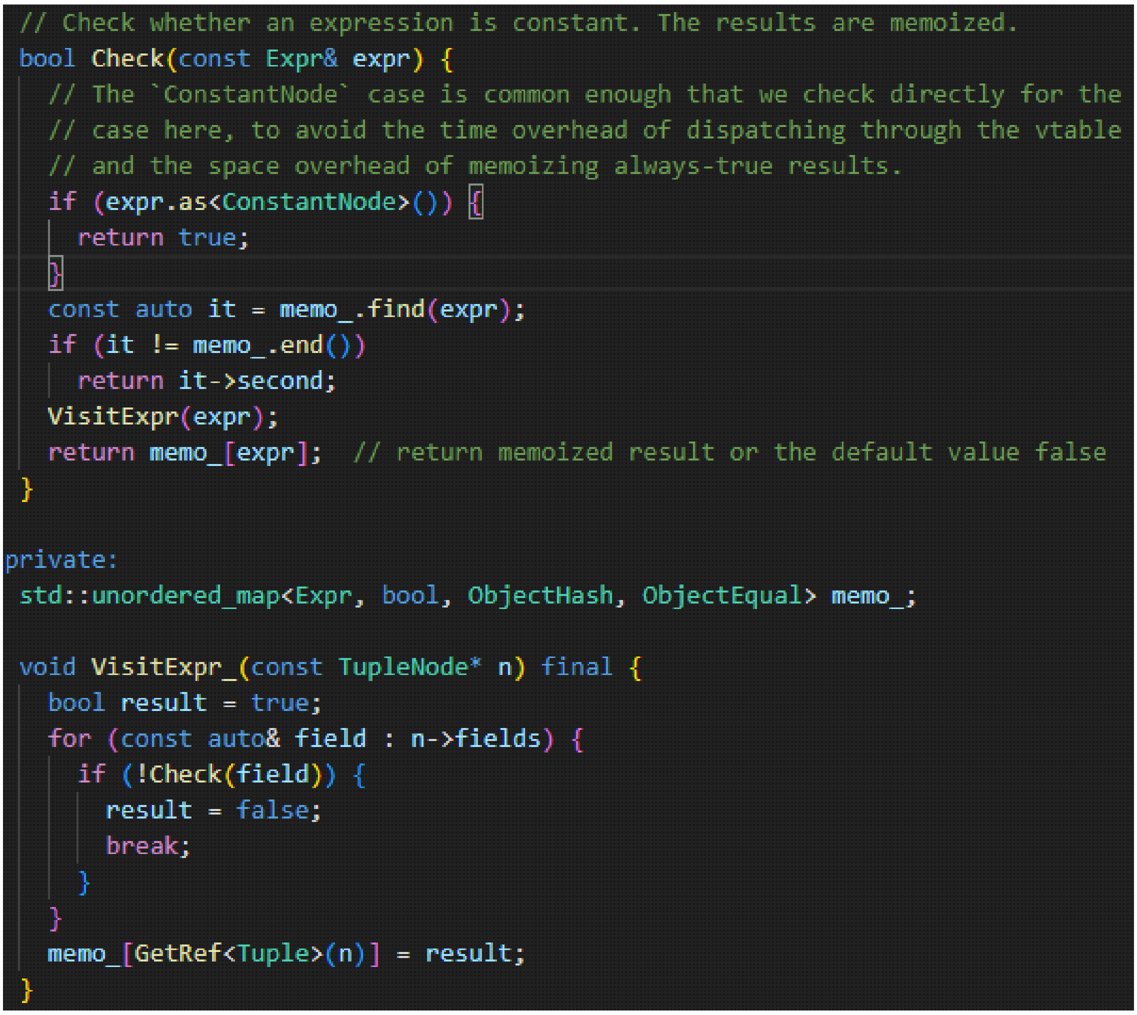
FoldConstant()

折叠常量，提前计算所有所有仅依赖于常量的节点

ConstantFolder调用Constant Checker判断节点是否为constant



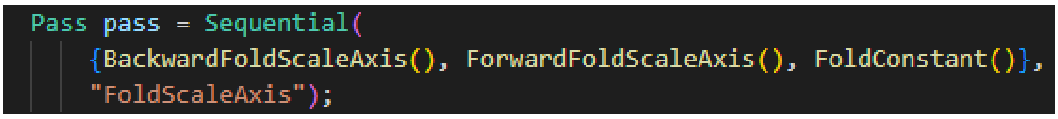
```
bool all_const_args = true;
for (Expr arg : call->args) {
  if (!checker_.Check(arg)) {
    all_const_args = false;
  }
}
if (all_const_args) {
  return ConstEvaluate(res);
} else {
  return res;
```



```
 // Check whether an expression is constant. The results are memoized.
 bool Check(const Expr& expr) {
   // The `ConstantNode` case is common enough that we check directly for the
   // case here, to avoid the time overhead of dispatching through the vtable
   // and the space overhead of memoizing always-true results.
   if (expr.as<ConstantNode>()) {
     return true;
   }
   const auto it = memo_.find(expr);
   if (it != memo_.end())
     return it->second;
   VisitExpr(expr);
   return memo_[expr];  // return memoized result or the default value false
 }

private:
 std::unordered_map<Expr, bool, ObjectHash, ObjectEqual> memo_;

 void VisitExpr_(const TupleNode* n) final {
   bool result = true;
   for (const auto& field : n->fields) {
     if (!Check(field)) {
       result = false;
       break;
     }
   }
   memo_[GetRef<Tuple>(n)] = result;
 }
```

FoldScaleAxis()

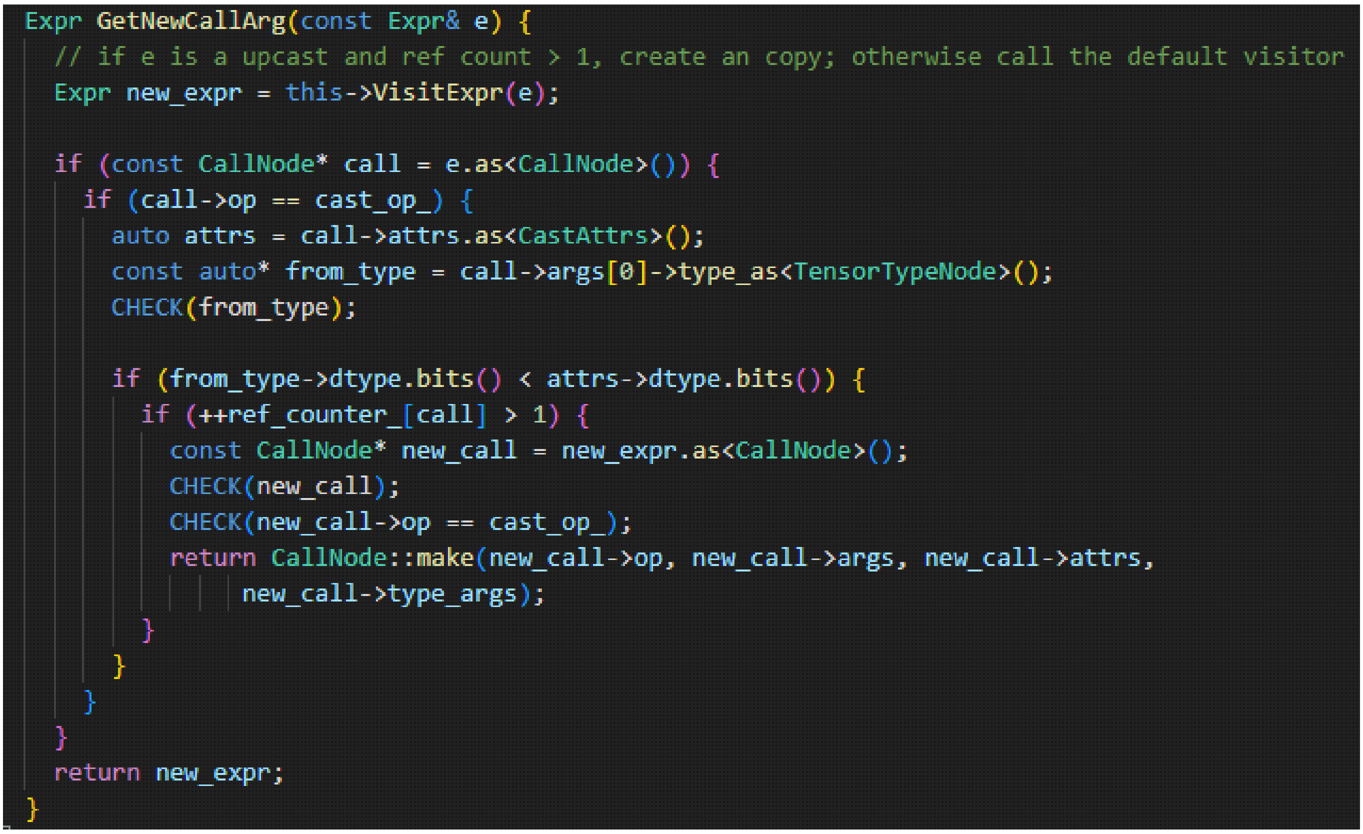
将Scale操作合并到Conv或Dense的参数中，包含三部分



```
Pass pass = Sequential(
    {BackwardFoldScaleAxis(), ForwardFoldScaleAxis(), FoldConstant()},
    "FoldScaleAxis");
```

CanonicalizeCast()

规范化cast，便于做op融合

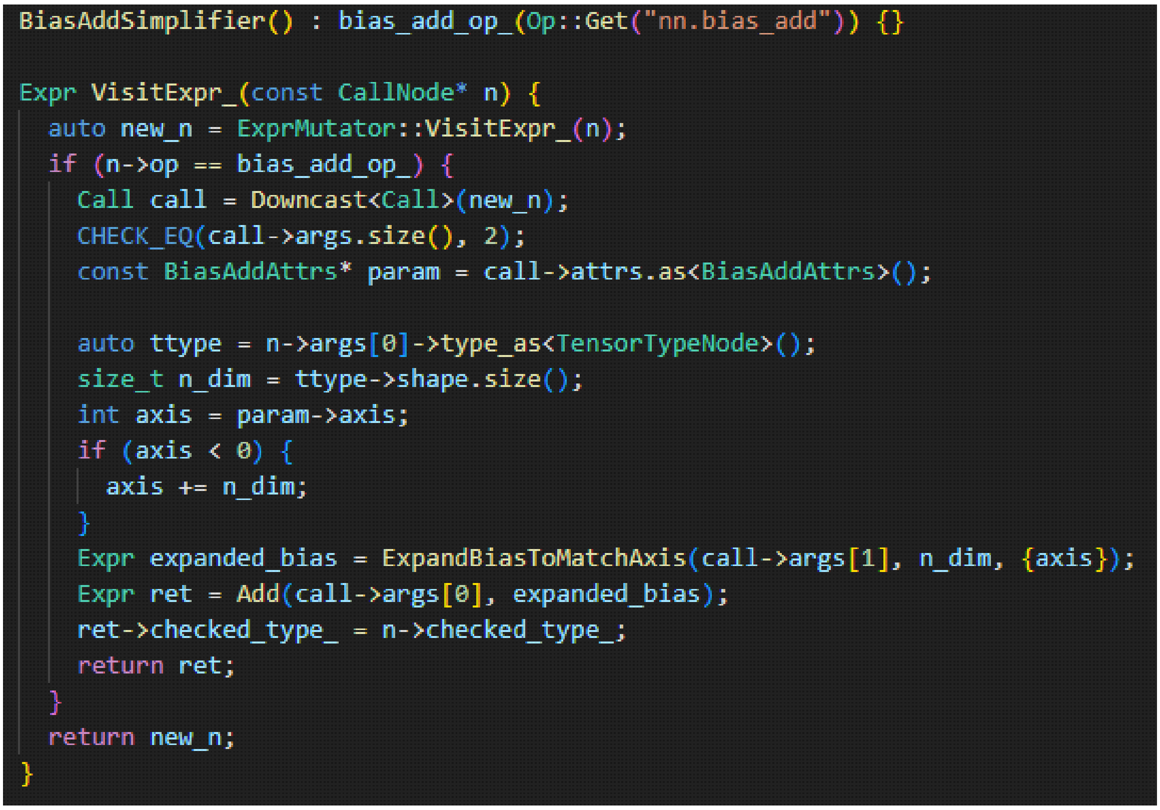


```
Expr GetNewCallArg(const Expr& e) {
  // if e is a upcast and ref count > 1, create an copy; otherwise call the default visitor
  Expr new_expr = this->VisitExpr(e);

  if (const CallNode* call = e.as<CallNode>()) {
    if (call->op == cast_op_) {
      auto attrs = call->attrs.as<CastAttrs>();
      const auto* from_type = call->args[0]->type_as<TensorTypeNode>();
      CHECK(from_type);

      if (from_type->dtype.bits() < attrs->dtype.bits()) {
        if (++ref_counter_[call] > 1) {
          const CallNode* new_call = new_expr.as<CallNode>();
          CHECK(new_call);
          CHECK(new_call->op == cast_op_);
          return CallNode::make(new_call->op, new_call->args, new_call->attrs,
                                new_call->type_args);
        }
      }
    }
  }
  return new_expr;
}
```

CanonicalizeOp()

本质是BiasAddSimplifier()

将bias_add展开成升维和Add

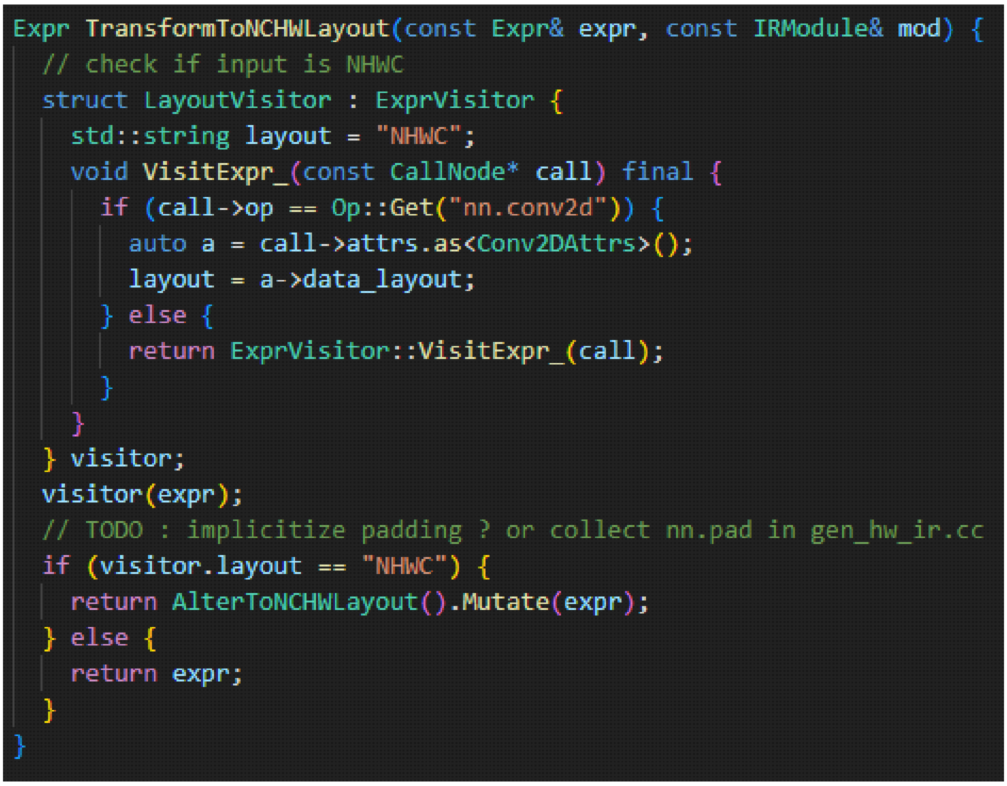


```
BiasAddSimplifier() : bias_add_op_(Op::Get("nn.bias_add")) {}

Expr VisitExpr_(const CallNode* n) {
  auto new_n = ExprMutator::VisitExpr_(n);
  if (n->op == bias_add_op_) {
    Call call = Downcast<Call>(new_n);
    CHECK_EQ(call->args.size(), 2);
    const BiasAddAttrs* param = call->attrs.as<BiasAddAttrs>();

    auto ttype = n->args[0]->type_as<TensorTypeNode>();
    size_t n_dim = ttype->shape.size();
    int axis = param->axis;
    if (axis < 0) {
      axis += n_dim;
    }
    Expr expanded_bias = ExpandBiasToMatchAxis(call->args[1], n_dim, {axis});
    Expr ret = Add(call->args[0], expanded_bias);
    ret->checked_type_ = n->checked_type_;
    return ret;
  }
  return new_n;
}
```

ToNCHWLayout()

自定义Pass

将数据格式转换到NCHW



```
Expr TransformToNCHWLayout(const Expr& expr, const IRModule& mod) {
  // check if input is NHWC
  struct LayoutVisitor : ExprVisitor {
    std::string layout = "NHWC";
    void VisitExpr_(const CallNode* call) final {
      if (call->op == Op::Get("nn.conv2d")) {
        auto a = call->attrs.as<Conv2DAttrs>();
        layout = a->data_layout;
      } else {
        return ExprVisitor::VisitExpr_(call);
      }
    }
  } visitor;
  visitor(expr);
  // TODO : implicitize padding ? or collect nn.pad in gen_hw_ir.cc
  if (visitor.layout == "NHWC") {
    return AlterToNCHWLayout().Mutate(expr);
  } else {
    return expr;
  }
}
```

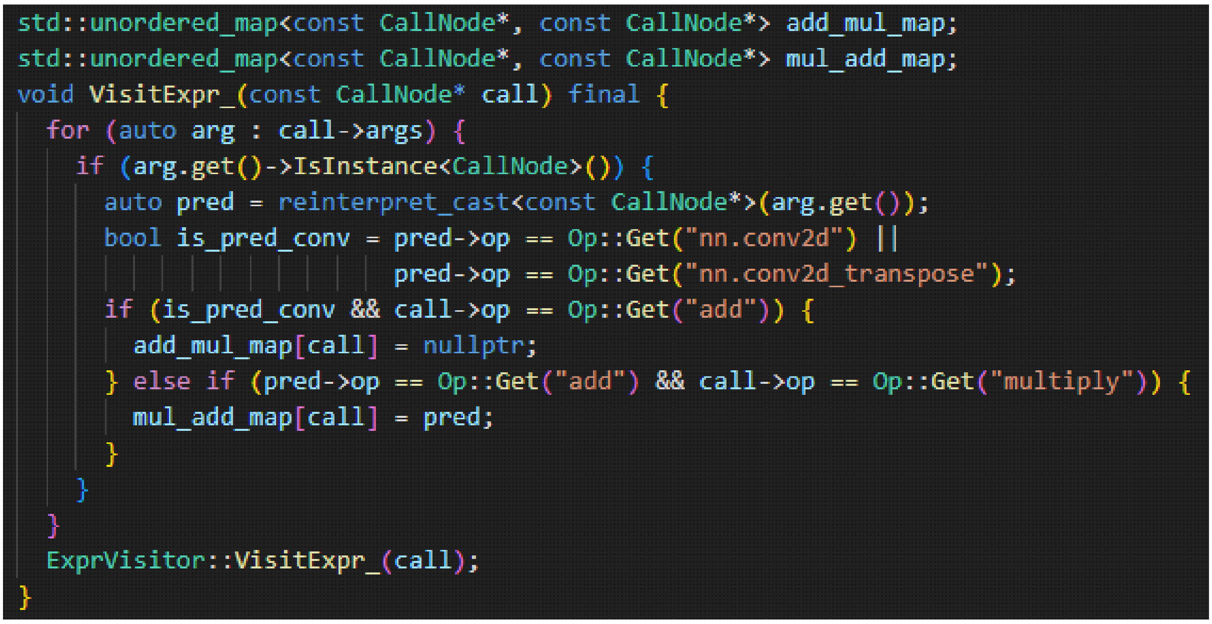
Peephole()

自定义Pass

一些针对OPU的customized优化

PatternTransformer1:

第一步建立索引，add_mul_map保存conv+add调用，格式（add, nullptr），mul_add_map保存add+mul调用（mul, add）

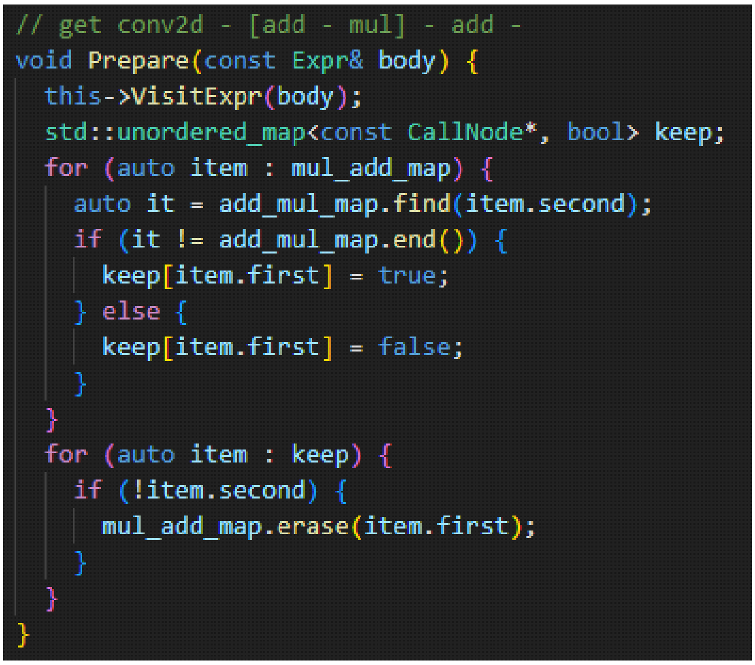


```
std::unordered_map<const CallNode*, const CallNode*> add_mul_map;
std::unordered_map<const CallNode*, const CallNode*> mul_add_map;
void VisitExpr_(const CallNode* call) final {
  for (auto arg : call->args) {
    if (arg.get()->IsInstance<CallNode>()) {
      auto pred = reinterpret_cast<const CallNode*>(arg.get());
      bool is_pred_conv = pred->op == Op::Get("nn.conv2d") ||
                          pred->op == Op::Get("nn.conv2d_transpose");
      if (is_pred_conv && call->op == Op::Get("add")) {
        add_mul_map[call] = nullptr;
      } else if (pred->op == Op::Get("add") && call->op == Op::Get("multiply")) {
        mul_add_map[call] = pred;
      }
    }
  }
  ExprVisitor::VisitExpr_(call);
}
```

第二步

寻找满足conv+add+mul的组合，并使用keep记录

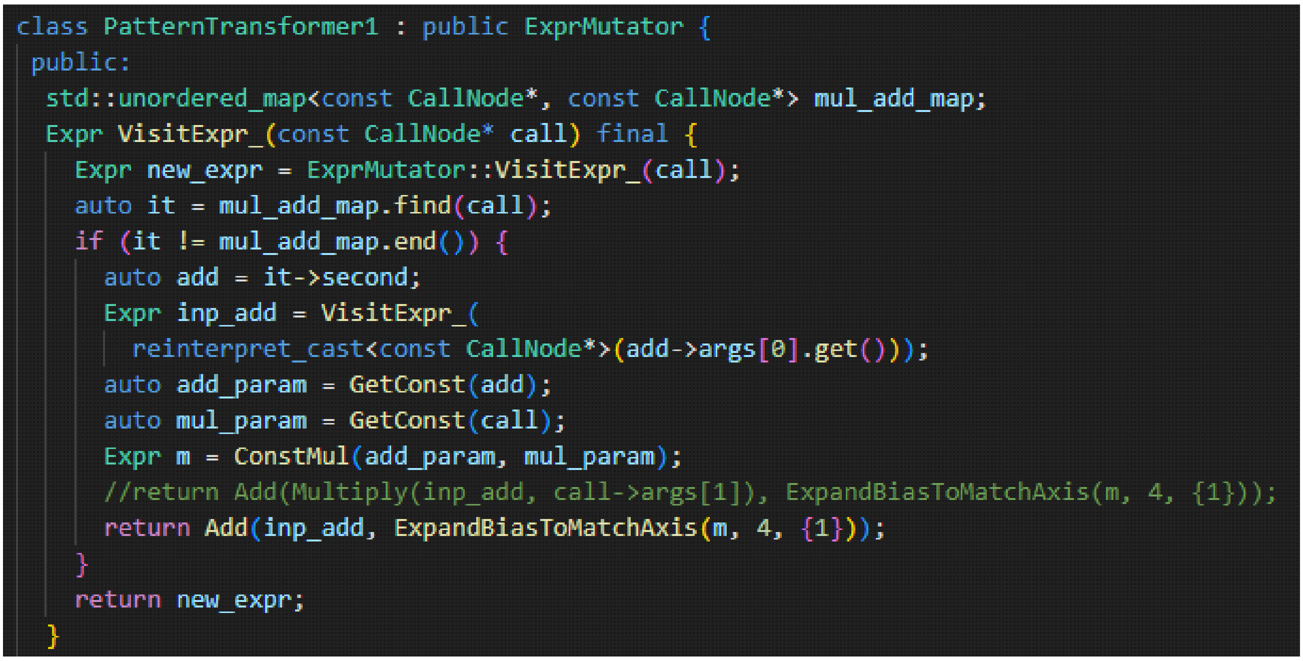
随后依据keep从mul_add_map中删去不满足conv_add_mul的项



```
// get conv2d - [add - mul] - add -
void Prepare(const Expr& body) {
  this->VisitExpr(body);
  std::unordered_map<const CallNode*, bool> keep;
  for (auto item : mul_add_map) {
    auto it = add_mul_map.find(item.second);
    if (it != add_mul_map.end()) {
      keep[item.first] = true;
    } else {
      keep[item.first] = false;
    }
  }
  for (auto item : keep) {
    if (!item.second) {
      mul_add_map.erase(item.first);
    }
  }
}
```

第三步

理解不能，有空再看

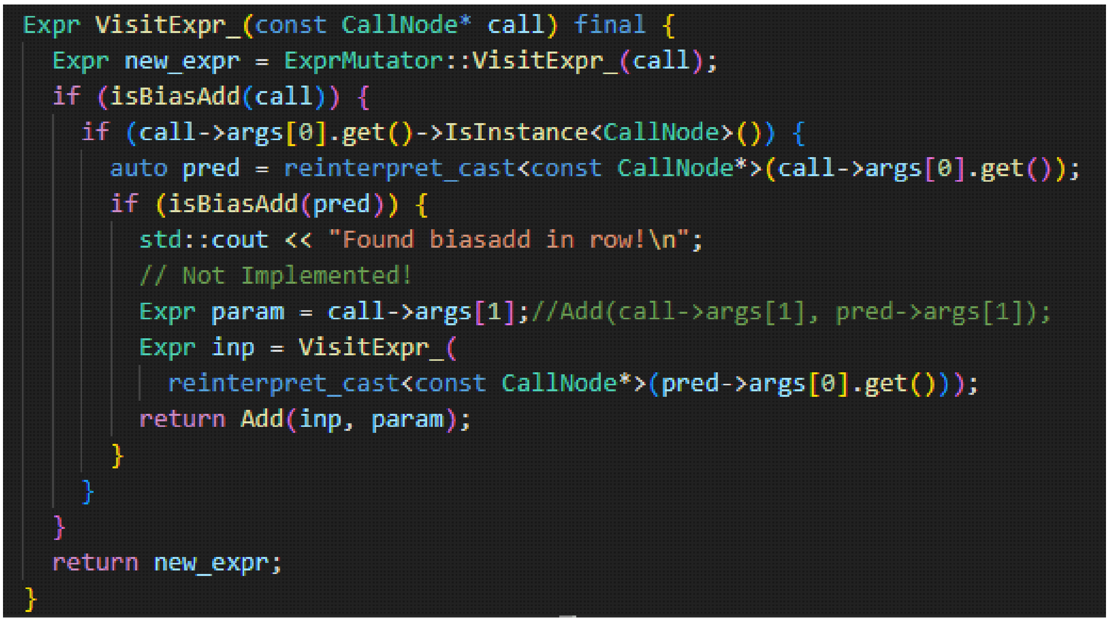


```
class PatternTransformer1 : public ExprMutator {
 public:
  std::unordered_map<const CallNode*, const CallNode*> mul_add_map;
  Expr VisitExpr_(const CallNode* call) final {
    Expr new_expr = ExprMutator::VisitExpr_(call);
    auto it = mul_add_map.find(call);
    if (it != mul_add_map.end()) {
      auto add = it->second;
      Expr inp_add = VisitExpr_(
        reinterpret_cast<const CallNode*>(add->args[0].get()));
      auto add_param = GetConst(add);
      auto mul_param = GetConst(call);
      Expr m = ConstMul(add_param, mul_param);
      //return Add(Multiply(inp_add, call->args[1]), ExpandBiasToMatchAxis(m, 4, {1}));
      return Add(inp_add, ExpandBiasToMatchAxis(m, 4, {1}));
    }
    return new_expr;
  }
```

PatternTransformer2：

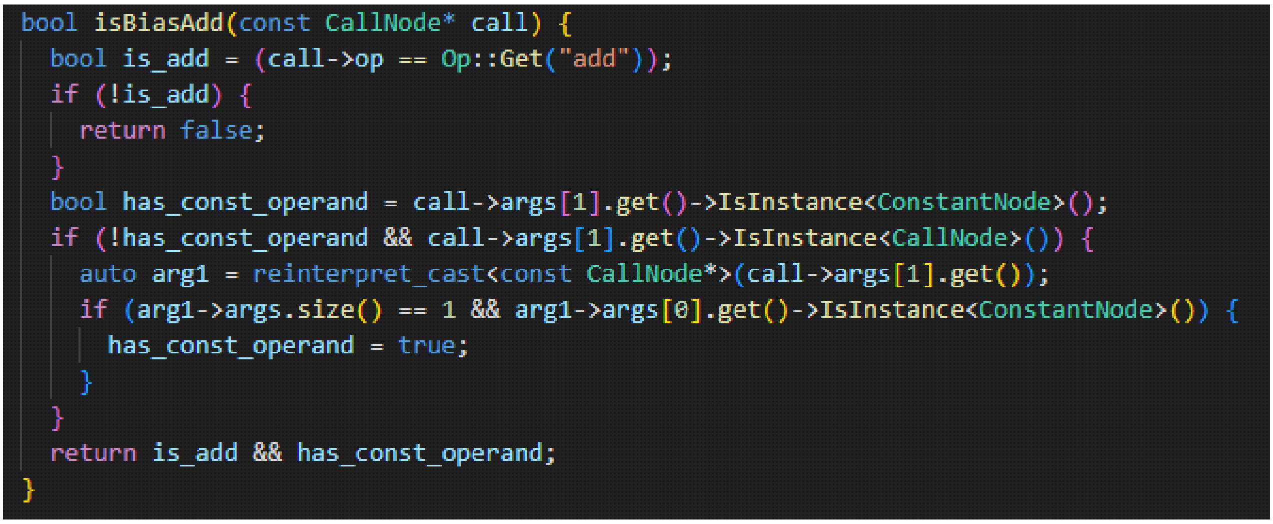
将连续两个常量加法的第二个操作数合并，即(?+c)+c ---> ?+c'

但是好像还没写完



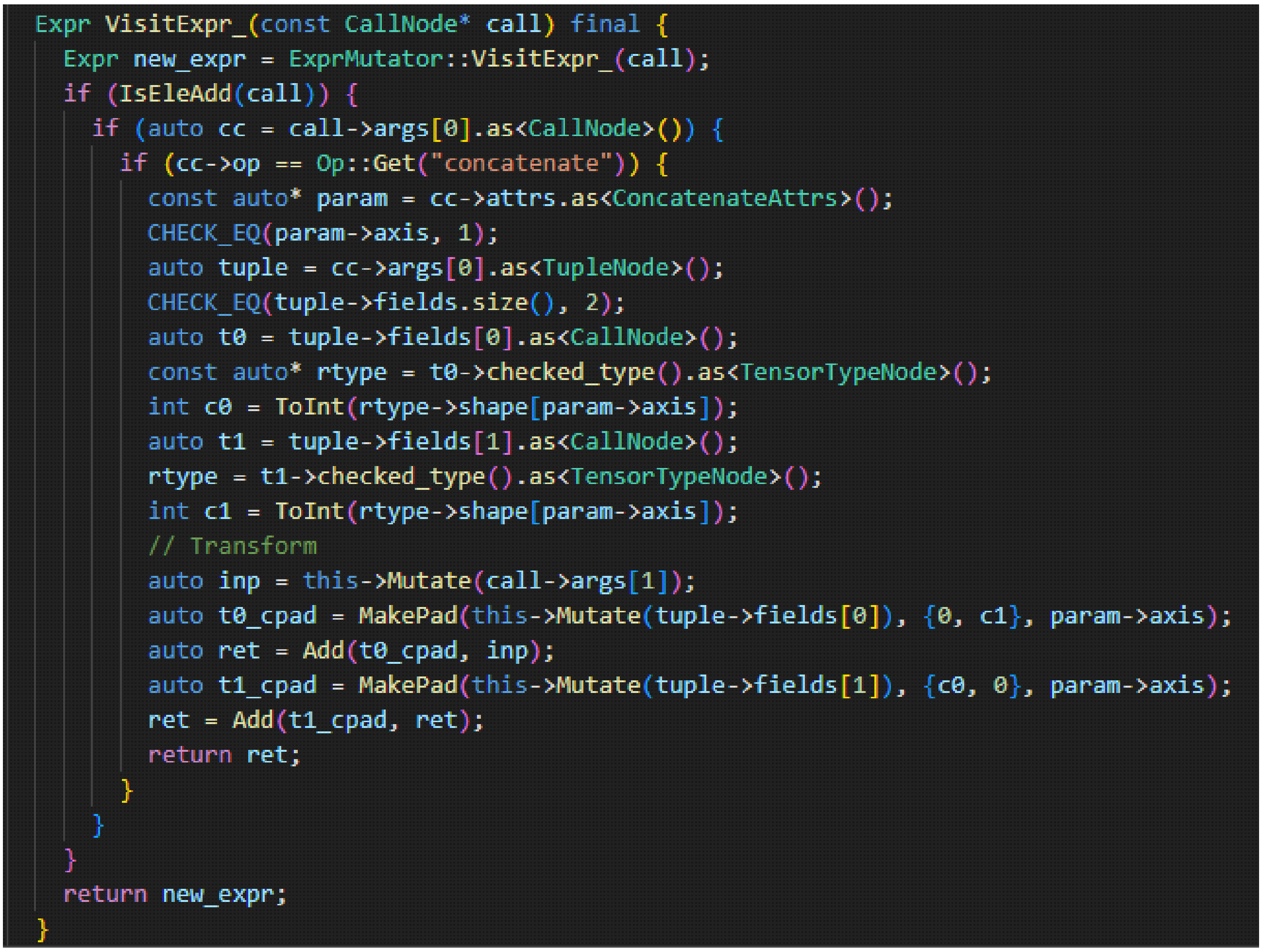
```
Expr VisitExpr_(const CallNode* call) final {
  Expr new_expr = ExprMutator::VisitExpr_(call);
  if (isBiasAdd(call)) {
    if (call->args[0].get()->IsInstance<CallNode>()) {
      auto pred = reinterpret_cast<const CallNode*>(call->args[0].get());
      if (isBiasAdd(pred)) {
        std::cout << "Found biasadd in row!\n";
        // Not Implemented!
        Expr param = call->args[1];//Add(call->args[1], pred->args[1]);
        Expr inp = VisitExpr_(
          reinterpret_cast<const CallNode*>(pred->args[0].get()));
        return Add(inp, param);
      }
    }
  }
  return new_expr;
}
```

其中调用了isBIasAdd() 判断Add的第二个操作数是不是常数



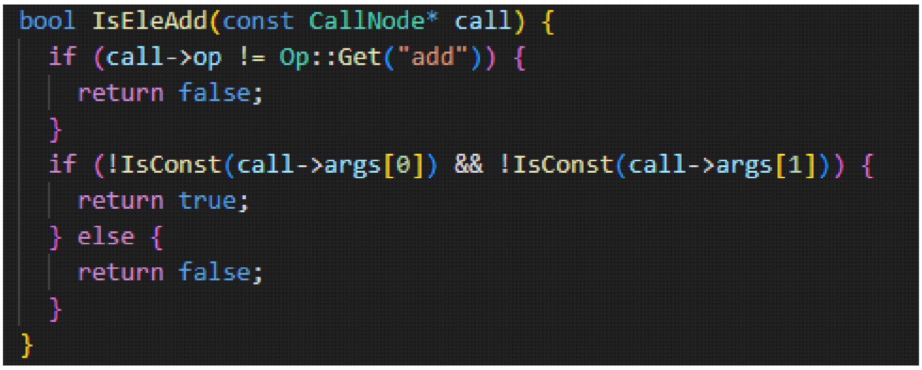
```
bool isBiasAdd(const CallNode* call) {
  bool is_add = (call->op == Op::Get("add"));
  if (!is_add) {
    return false;
  }
  bool has_const_operand = call->args[1].get()->IsInstance<ConstantNode>();
  if (!has_const_operand && call->args[1].get()->IsInstance<CallNode>()) {
    auto arg1 = reinterpret_cast<const CallNode*>(call->args[1].get());
    if (arg1->args.size() == 1 && arg1->args[0].get()->IsInstance<ConstantNode>()) {
      has_const_operand = true;
    }
  }
  return is_add && has_const_operand;
}
```

PatternTransformer3：



```
Expr VisitExpr_(const CallNode* call) final {
  Expr new_expr = ExprMutator::VisitExpr_(call);
  if (IsEleAdd(call)) {
    if (auto cc = call->args[0].as<CallNode>()) {
      if (cc->op == Op::Get("concatenate")) {
        const auto* param = cc->attrs.as<ConcatenateAttrs>();
        CHECK_EQ(param->axis, 1);
        auto tuple = cc->args[0].as<TupleNode>();
        CHECK_EQ(tuple->fields.size(), 2);
        auto t0 = tuple->fields[0].as<CallNode>();
        const auto* rtype = t0->checked_type().as<TensorTypeNode>();
        int c0 = ToInt(rtype->shape[param->axis]);
        auto t1 = tuple->fields[1].as<CallNode>();
        rtype = t1->checked_type().as<TensorTypeNode>();
        int c1 = ToInt(rtype->shape[param->axis]);
        // Transform
        auto inp = this->Mutate(call->args[1]);
        auto t0_cpad = MakePad(this->Mutate(tuple->fields[0]), {0, c1}, param->axis);
        auto ret = Add(t0_cpad, inp);
        auto t1_cpad = MakePad(this->Mutate(tuple->fields[1]), {c0, 0}, param->axis);
        ret = Add(t1_cpad, ret);
        return ret;
      }
    }
  }
  return new_expr;
}
```

其中调用了IsEleAdd()判断某一运算是否是Add，且两个操作数均不为常量



```
bool IsEleAdd(const CallNode* call) {
  if (call->op != Op::Get("add")) {
    return false;
  }
  if (!IsConst(call->args[0]) && !IsConst(call->args[1])) {
    return true;
  } else {
    return false;
  }
}
```

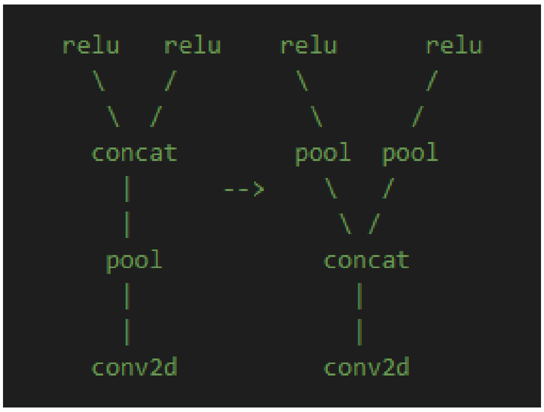
PatternTransformer5：

将input channel数量不超过2048的fc层转换为1*1卷积层

完成上述优化后，再执行一次BackwardFoldScaleAxis(), ForwardFoldScaleAxis(), FoldConstant()

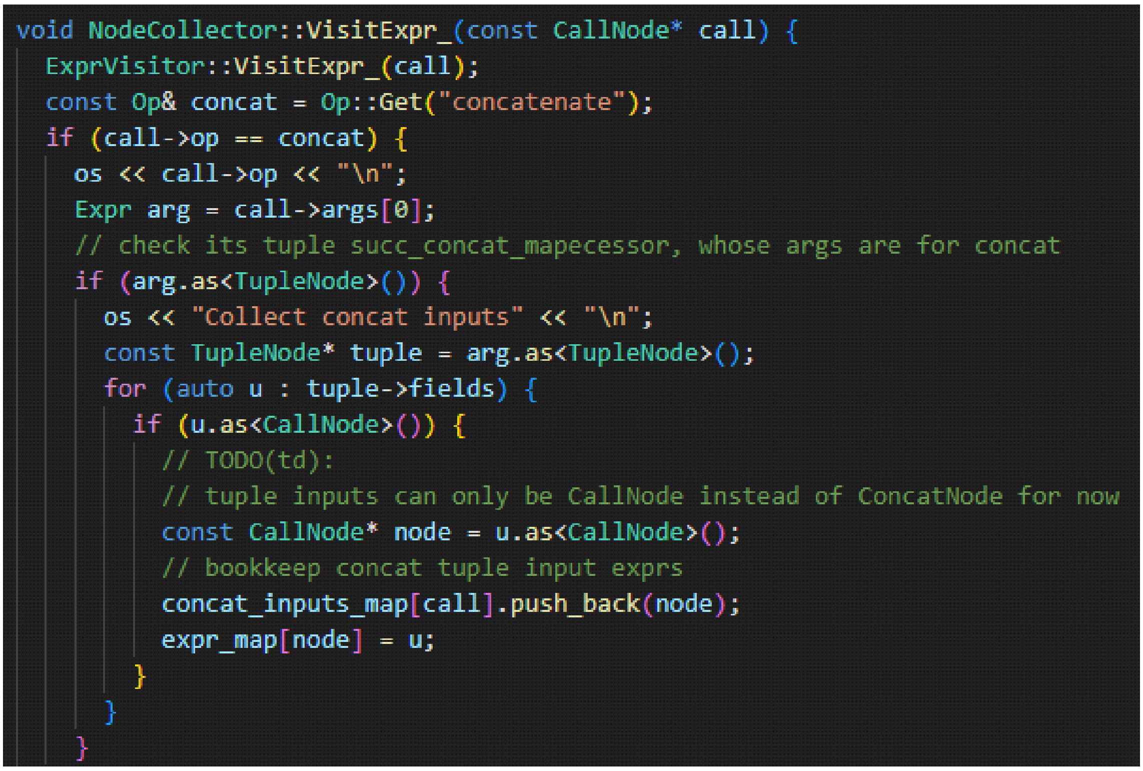
CanonicalizeConcatPos()

如果在concatenate层和fc/conv层之间有其他运算，则将其移动到concatenate层之前

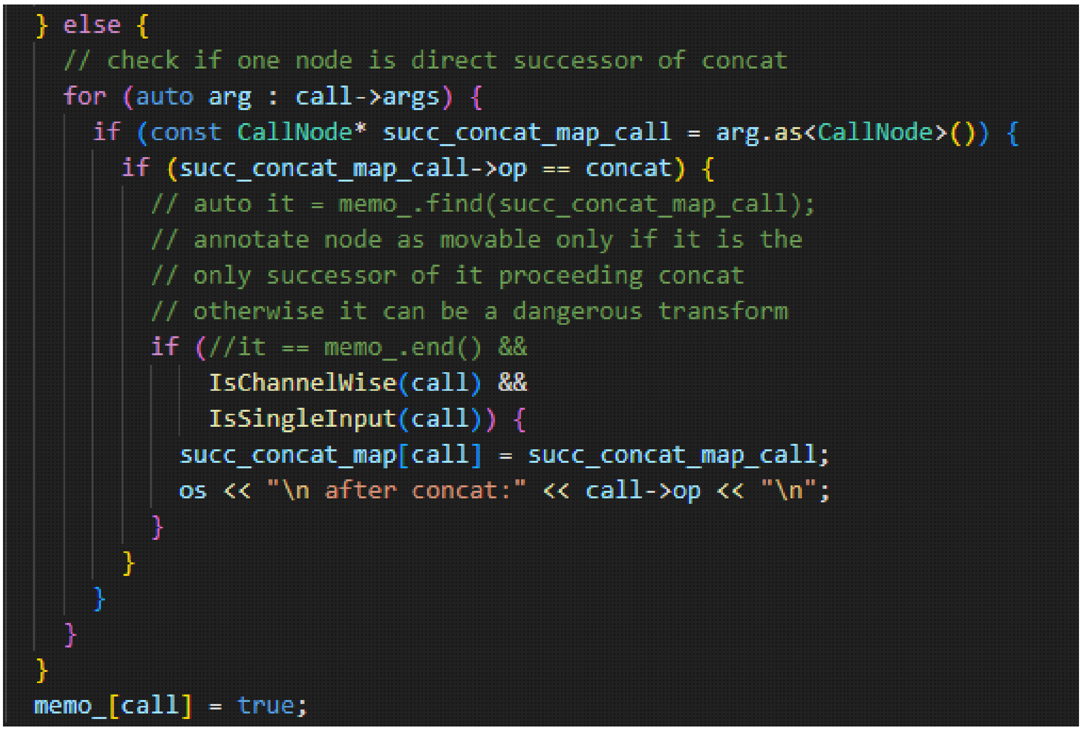


```
   relu   relu    relu      relu
     \     /        \        /
      \   /          \      /
     concat          pool  pool
       |      -->      \   /
       |                \ /
      pool             concat
       |                 |
       |                 |
     conv2d            conv2d
```

第一步，建立索引。concat_inputs_map保存每个concatenate操作及其输入，即（concat, <输入>）；succ_concat_map保存每个concatenate操作及其下一层，且必须满足ChannelWise和SingleInput条件，方可确保上述转换不会引入问题。

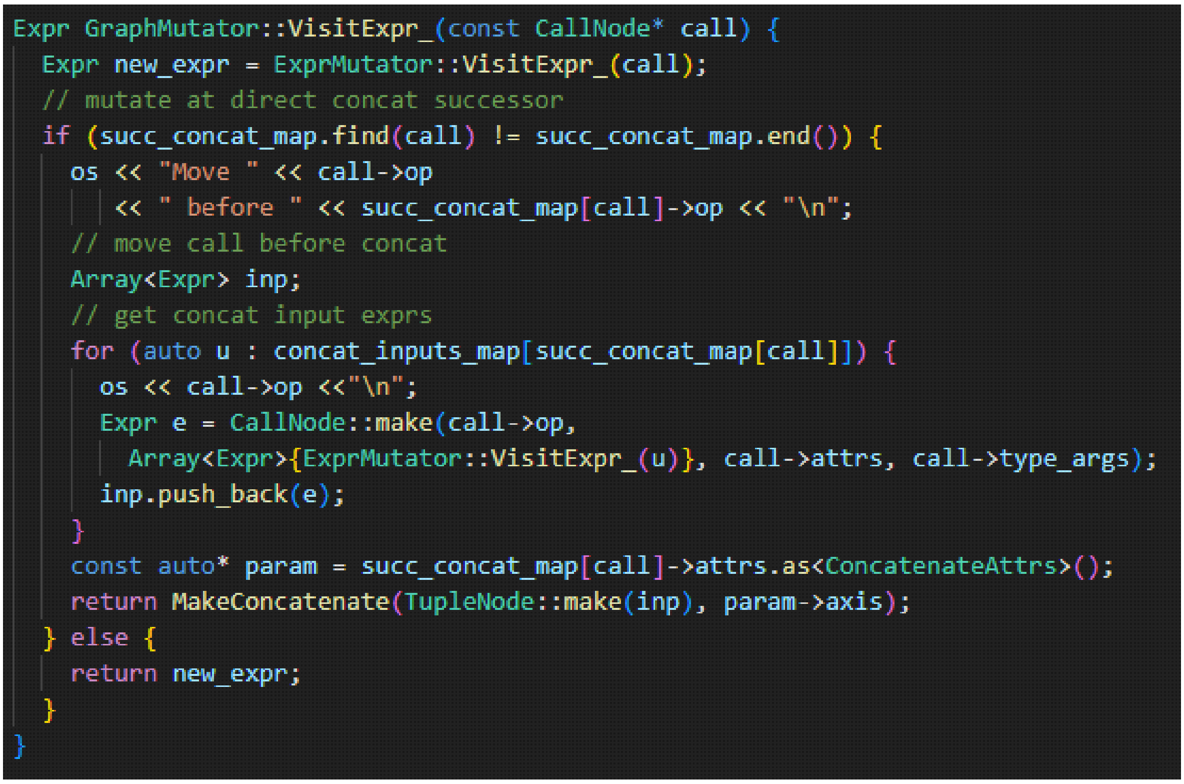


```
void NodeCollector::VisitExpr_(const CallNode* call) {
  ExprVisitor::VisitExpr_(call);
  const Op& concat = Op::Get("concatenate");
  if (call->op == concat) {
    os << call->op << "\n";
    Expr arg = call->args[0];
    // check its tuple succ_concat_mapecessor, whose args are for concat
    if (arg.as<TupleNode>()) {
      os << "Collect concat inputs" << "\n";
      const TupleNode* tuple = arg.as<TupleNode>();
      for (auto u : tuple->fields) {
        if (u.as<CallNode>()) {
          // TODO(td):
          // tuple inputs can only be CallNode instead of ConcatNode for now
          const CallNode* node = u.as<CallNode>();
          // bookkeep concat tuple input exprs
          concat_inputs_map[call].push_back(node);
          expr_map[node] = u;
        }
      }
    }
```



```
} else {
  // check if one node is direct successor of concat
  for (auto arg : call->args) {
    if (const CallNode* succ_concat_map_call = arg.as<CallNode>()) {
      if (succ_concat_map_call->op == concat) {
        // auto it = memo_.find(succ_concat_map_call);
        // annotate node as movable only if it is the
        // only successor of it proceeding concat
        // otherwise it can be a dangerous transform
        if (//it == memo_.end() &&
            IsChannelWise(call) &&
            IsSingleInput(call)) {
          succ_concat_map[call] = succ_concat_map_call;
          os << "\n after concat:" << call->op << "\n";
        }
      }
    }
  }
}
memo_[call] = true;
```

第二步，变换。满足条件的concatenate及其后续层已经保存于succ_concat_map中，将concatenate的后续层移动到concatenate之前，即与concatenate的输入融合，此处会用到concat_inputs_map中保存的concatenate输入信息。

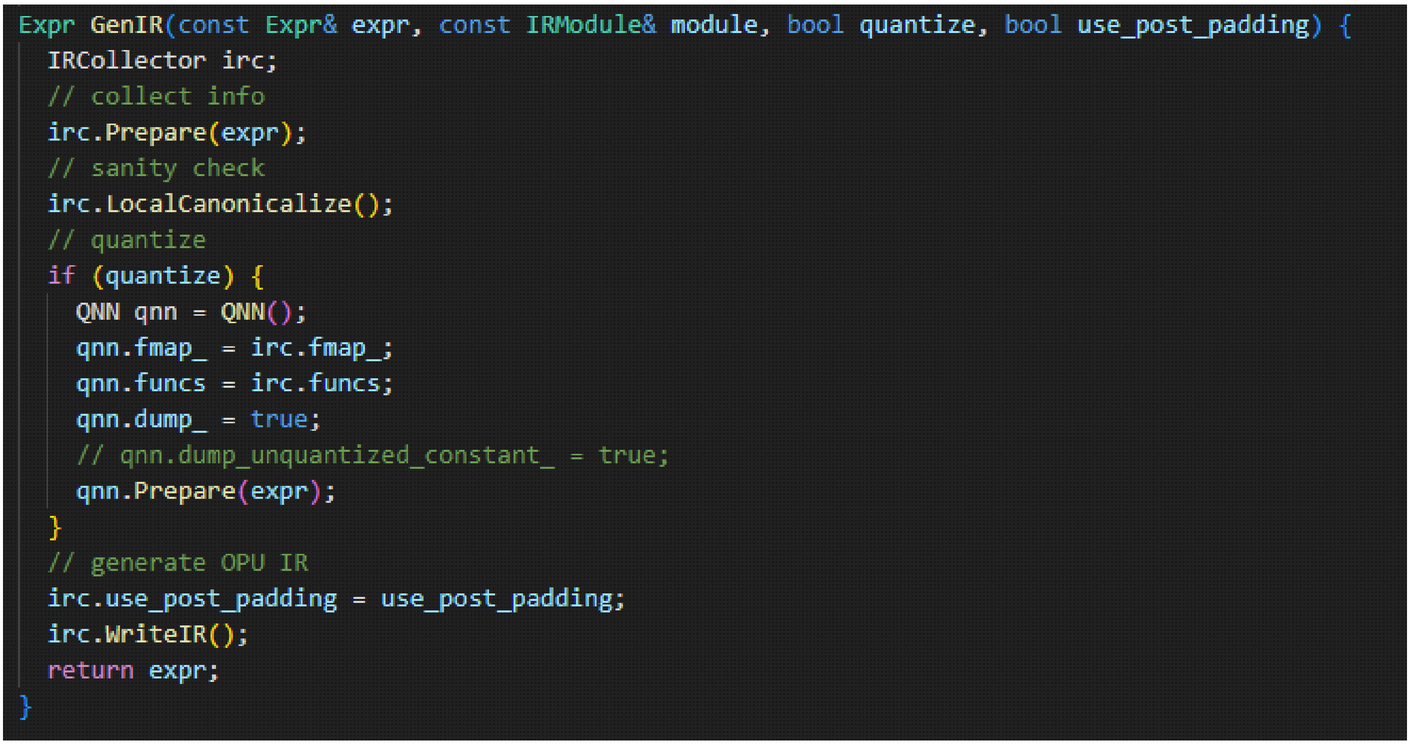


```
Expr GraphMutator::VisitExpr_(const CallNode* call) {
  Expr new_expr = ExprMutator::VisitExpr_(call);
  // mutate at direct concat successor
  if (succ_concat_map.find(call) != succ_concat_map.end()) {
    os << "Move " << call->op
       << " before " << succ_concat_map[call]->op << "\n";
    // move call before concat
    Array<Expr> inp;
    // get concat input exprs
    for (auto u : concat_inputs_map[succ_concat_map[call]]) {
      os << call->op <<"\n";
      Expr e = CallNode::make(call->op,
        Array<Expr>{ExprMutator::VisitExpr_(u)}, call->attrs, call->type_args);
      inp.push_back(e);
    }
    const auto* param = succ_concat_map[call]->attrs.as<ConcatenateAttrs>();
    return MakeConcatenate(TupleNode::make(inp), param->axis);
  } else {
    return new_expr;
  }
}
```

PrintIR()
打印IR

GenIR()
自定义Pass

量化，输出IR



```
Expr GenIR(const Expr& expr, const IRModule& module, bool quantize, bool use_post_padding) {
  IRCollector irc;
  // collect info
  irc.Prepare(expr);
  // sanity check
  irc.LocalCanonicalize();
  // quantize
  if (quantize) {
    QNN qnn = QNN();
    qnn.fmap_ = irc.fmap_;
    qnn.funcs = irc.funcs;
    qnn.dump_ = true;
    // qnn.dump_unquantized_constant_ = true;
    qnn.Prepare(expr);
  }
  // generate OPU IR
  irc.use_post_padding = use_post_padding;
  irc.WriteIR();
  return expr;
}
```