6주차 과제

결과

`IterCatchHandlerPhis` 메소드 파라미터인 `catch_block` 포인터 객체(BasicBlock) 코드의 의미를 이해하였음.

분석

```
builder_.IterCatchHandlerPhis(
      catch_block_, [this, compact_frame, maglev_unit](
                  interpreter::Register owner, Variable var) {
       DCHECK_NE(owner, interpreter::Register::virtual_accumulator());

       const maglev::ValueNode* maglev_value =
          compact_frame->GetValueOf(owner, maglev_unit);
       DCHECK_NOT_NULL(maglev_value);

       while (maglev_value->Is<maglev::Identity>()) {
        maglev_value = maglev_value->input(0).node();
       }

       if (const maglev::VirtualObject* vobj =
             maglev_value->TryCast<maglev::VirtualObject>()) {
        maglev_value = vobj->allocation();
       }

       DCHECK(!maglev_value->Is<maglev::Identity>());
       DCHECK(!maglev_value->Is<maglev::VirtualObject>());
       V<Any> ts_value = builder_.Map(maglev_value);
       __ SetVariable(var, ts_value);
       builder_.RecordRepresentation(ts_value,
```

```
                                maglev_value->value_representation());
      });
```

위 코드는 `builder_` 클래스의 `IterCatchHandlerPhis()` 메소드며, **turboshaft**의 `turbolev-graph-builder.cc`에 구현되어 있다. 각 인자로 `catch_block_`과 람다 함수가 전달된다.

```
void IterCatchHandlerPhis(const maglev::BasicBlock* catch_block,
                          Function&& callback) {
  DCHECK_NOT_NULL(catch_block);
  DCHECK(catch_block->has_phi());
  for (auto phi : *catch_block->phis()) {
    DCHECK(phi->is_exception_phi());
    interpreter::Register owner = phi->owner();
    if (owner == interpreter::Register::virtual_accumulator()) {
      // The accumulator exception phi corresponds to the exception object
      // rather than whatever value the accumulator contained before the
      // throwing operation. We don't need to iterate here, since there is
      // special handling when processing Phis to use `catch_block_begin_`
      // for it instead of a Variable.
      continue;
    }

    auto it = regs_to_vars_.find(owner.index());
    Variable var;
    if (it == regs_to_vars_.end()) {
      // We use a LoopInvariantVariable: if loop phis were needed, then the
      // Maglev value would already be a loop Phi, and we wouldn't need
      // Turboshaft to automatically insert a loop phi.
      var = __ NewLoopInvariantVariable(RegisterRepresentation::Tagged());
      regs_to_vars_.insert({owner.index(), var});
    } else {
      var = it->second;
    }

    callback(owner, var);
```

```
    }
  }
```

위는 `IterCatchHanderPhis`의 원형이며, 이를 한 줄 한 줄 이해해보기로 하며 먼저 첫 번째 파라미터 `catch_block`을 분석한다. catch_block은 BasicBlock 객체를 전달 받는 상수 포인터다.

```
class BasicBlock {
public:
  explicit BasicBlock(MergePointInterpreterFrameState* state, Zone* zone)
      : type_(state ? kMerge : kOther),
      nodes_(zone),
      control_node_(nullptr),
      state_(state) {
  }

 ...
};
```

BasicBlock은 Public(공개)으로 선언된 많은 멤버 함수가 있으며, 초기화를 위한 생성자가 있다. 이는 객체 생성 시 기본적으로 설정되는 코드다. 만약 이 BasicBlock에 인자가 주어진다면, 각각 MergePointInterpreterFrameState 클래스 포인터 state와 Zone 클래스 포인터 zone이 세팅 된다.

type_

type_은 열거형 멤버 변수로서 `enum : uint8_t { kMerge, kEdgeSplit, kOther }` 로 구성되어 있으며, `type_(state ? kMerge : kOther)` 삼항 연산자로 state 포인터가 Null이면 kOther, Null이 아니면 kMerge 리터럴이 구성된다.

Nodes_

ZoneVector<Node*> nodes_;

control_node_

ControlNode* control_node_; 로 구성되어 있으며 ControlNode 클래스 객체의 포인터로 사용된다. 초기값은 nullptr로 세팅되어 있다.

state_

state 멤버 변수를 this→state_로 설정한다.

이 외에 코드는 멤버 함수에 대한 정의들인데, 이 객체에 들어오는 Input을 알 수가 없으므로 사실상 코드를 더 이해하는 건 무의미해 보인다. Input 데이터를 추적 할 시간이다.

```
class ExceptionHandlerInfo {
 public:
  using List = base::ThreadedList<ExceptionHandlerInfo>;
  enum Mode {
    kNoExceptionHandler = -1,
    kLazyDeopt = -2,
  };

  explicit ExceptionHandlerInfo(Mode mode = kNoExceptionHandler)
      : catch_block_(), depth_(static_cast<int>(mode)), pc_offset_(-1) {}

  ExceptionHandlerInfo(BasicBlockRef* catch_block_ref, int depth)
      : catch_block_(catch_block_ref), depth_(depth), pc_offset_(-1) {
    DCHECK_NE(depth, kNoExceptionHandler);
    DCHECK_NE(depth, kLazyDeopt);
  }

  ExceptionHandlerInfo(BasicBlock* catch_block_ref, int depth)
      : catch_block_(catch_block_ref), depth_(depth), pc_offset_(-1) {}

  bool HasExceptionHandler() const { return depth_ != kNoExceptionHandler; }
```

```
  bool ShouldLazyDeopt() const { return depth_ == kLazyDeopt; }

  Label& trampoline_entry() { return trampoline_entry_; }

  BasicBlockRef* catch_block_ref_address() { return &catch_block_; }

  BasicBlock* catch_block() const { return catch_block_.block_ptr(); }

  int depth() const {
    DCHECK_NE(depth_, kNoExceptionHandler);
    DCHECK_NE(depth_, kLazyDeopt);
    return depth_;
  }

  int pc_offset() const { return pc_offset_; }
  void set_pc_offset(int offset) {
    DCHECK_EQ(pc_offset_, -1);
    DCHECK_NE(offset, -1);
    pc_offset_ = offset;
  }

 private:
  BasicBlockRef catch_block_;
  Label trampoline_entry_;
  int depth_;
  int pc_offset_;

  ExceptionHandlerInfo* next_ = nullptr;
  ExceptionHandlerInfo** next() { return &next_; }

  friend List;
  friend base::ThreadedListTraits<ExceptionHandlerInfo>;
};
```

이것은 인자(인풋)로 넘어온 catch_block_ 구현 코드(maglev-ir.h)다. `BasicBlock* catch_block() const { return catch_block_.block_ptr(); }` 를 보면 알 수 있듯 `catch_block()` 메소드는

`BasicBlock*` 타입의 `block_ptr();`을 반환한다.

이 때 catch_block_은 `BasicBlockRef catch_block_;` 와 같이 private 멤버 변수로 선언되어 있다. 이 **클래스 멤버 변수(BasicBlockRef)**의 block_ptr() 메소드를 반환하여 BasicBlock*으로 리턴하게 된다.

```
explicit BasicBlockRef(BasicBlock* block) : block_ptr_(block) {

...

BasicBlock* block_ptr() const {
  DCHECK_EQ(state_, kBlockPointer);
  return block_ptr_;
}
```

결국은 `block_ptr_` 이라고 부르는 BasicBlock 타입의 멤버 변수이자 포인터를 반환한다.

```
void set_block_ptr(BasicBlock* block) {
  DCHECK_EQ(state_, kBlockPointer);
  block_ptr_ = block;
}
```

block_ptr_은 `set_block_ptr()` 메소드가 설정을 담당하는 것 같다. 그러나 x-ref가 여기까지이므로 이 메소드로 재분석을 시도해야한다.

```
class MaglevCodeGeneratingNodeProcessor {

...
```

```
private:
 // Jump threading: instead of jumping to an empty block A which just
 // unconditionally jumps to B, redirect the jump to B directly.
 template <typename NodeT>
 void PatchJumps(NodeT* node) {
  if constexpr (IsUnconditionalControlNode(Node::opcode_of<NodeT>)) {
   UnconditionalControlNode* control_node =
     node->template Cast<UnconditionalControlNode>();
   control_node->set_target(
     code_gen_state()->RealJumpTarget(control_node->target()));
  } else if constexpr (IsBranchControlNode(Node::opcode_of<NodeT>)) {
   BranchControlNode* control_node =
     node->template Cast<BranchControlNode>();
   control_node->set_if_true(
     code_gen_state()->RealJumpTarget(control_node->if_true()));
   control_node->set_if_false(
     code_gen_state()->RealJumpTarget(control_node->if_false()));
  } else if constexpr (Node::opcode_of<NodeT> == Opcode::kSwitch) {
   Switch* switch_node = node->template Cast<Switch>();
   BasicBlockRef* targets = switch_node->targets();
   for (int i = 0; i < switch_node->size(); ++i) {
    targets[i].set_block_ptr(                          // set_block_ptr 존재
      code_gen_state()->RealJumpTarget(targets[i].block_ptr()));
   }
   if (switch_node->has_fallthrough()) {
    switch_node->set_fallthrough(
      code_gen_state()->RealJumpTarget(switch_node->fallthrough()));
   }
  }
 }

}
```

`set_block_ptr()` 메소드의 Call Hierarchy를 따라가보면 `maglev-code-generator.cc`의 코드 중 `MaglevCodeGeneratingNodeProcessor` 클래스가 나온다. private로 정의된 `PatchJumps`의 메소드는 node라고 하는 정의되지 않은 타입인 NodeT를 가리키는 포인터 변수를 받는데, 이 node의 특성(Switch, Branch 등)에 따라 이후 캐스팅 된다.

즉 해당 노드가 `kSwitch`라면 `set_block_ptr`에서 JumpTarget으로 블록을 가져온다. 이렇게 가져 온 블록은 각 Opcode에 매칭되어 네이티브 코드로 변환된다. 따라서, 이 메소드는 본래 보고자 하려 했던 `IterCatchHanderPhis` 과는 직접적인 관련이 있어 보이진 않는다.

CFG의 개념을 이해하면서 알게 된 BasicBlock은 각 분기 별로 BB(BasicBlock)을 나누고 Edge를 잇는 형태란 것이다. (IDA에서 본 그래프와 비슷하다.) 이렇게 만들어진 CFG의 형태를 SSA 기반 IR로 만드는 역할이 **Maglev**이다.

한번 정리해보자면 BasicBlock 객체는 각 **노드(BB)** 정보를 담고 있고 이를 **block 포인터**로 관리한다. 이후에 Maglev에서 만든 IR은 직접 `MaglevCodeGeneratingNodeProcessor` 등으로 코드를 최적화 하거나 Turbolev에게 넘기고 이는 다시 `turbolev-graph-builder.cc`에 의해 다시 IR 그래프를 그리고 Turboshaft에게 전달한다.

```
DCHECK(catch_block->has_phi());
  for (auto phi : *catch_block->phis()) {
    DCHECK(phi->is_exception_phi());
```

블록엔 Phi가 존재할 수 있다. 특히, 선행 블록에 분기가 있었다면, 현재 블록엔 Phi가 있으므로, 선행 블록의 결과값을 Phi를 통해 가져온다.

```
class Phi : public ValueNodeT<Phi> {
 using Base = ValueNodeT<Phi>;

public:
 using List = base::ThreadedList<Phi>;

 // TODO(jgruber): More intuitive constructors, if possible.
 Phi(uint64_t bitfield, MergePointInterpreterFrameState* merge_state,
     interpreter::Register owner)
     : Base(bitfield),
       owner_(owner),
       merge_state_(merge_state),
```

```
      type_(NodeType::kUnknown),
      post_loop_type_(NodeType::kUnknown) {
  DCHECK_NOT_NULL(merge_state);
 }
```

Phi 클래스엔 생성자에 따라 멤버 변수들이 초기화된다.

```
class V8_EXPORT_PRIVATE Register final {
 public:
  constexpr explicit Register(int index = kInvalidIndex) : index_(index) {}

  constexpr int index() const { return index_; }
  constexpr bool is_parameter() const { return index() < 0; }
  constexpr bool is_valid() const { return index_ != kInvalidIndex; }

  static constexpr Register FromParameterIndex(int index);
  constexpr int ToParameterIndex() const;

  ...

  int index_;
}
```

`owner_`는 Register 클래스를 가지며 index의 값에 따른 레지스터 특성을 분류한다.