Chainer compilerのソースコード解析

2019/04/22 Chainer Compiler のソースコードをグダグダ語る会

加藤倫弘
AI本部AIシステム部 AI研究開発第二グループ
株式会社ディー・エヌ・エー

DeNA Delight and Impact the World

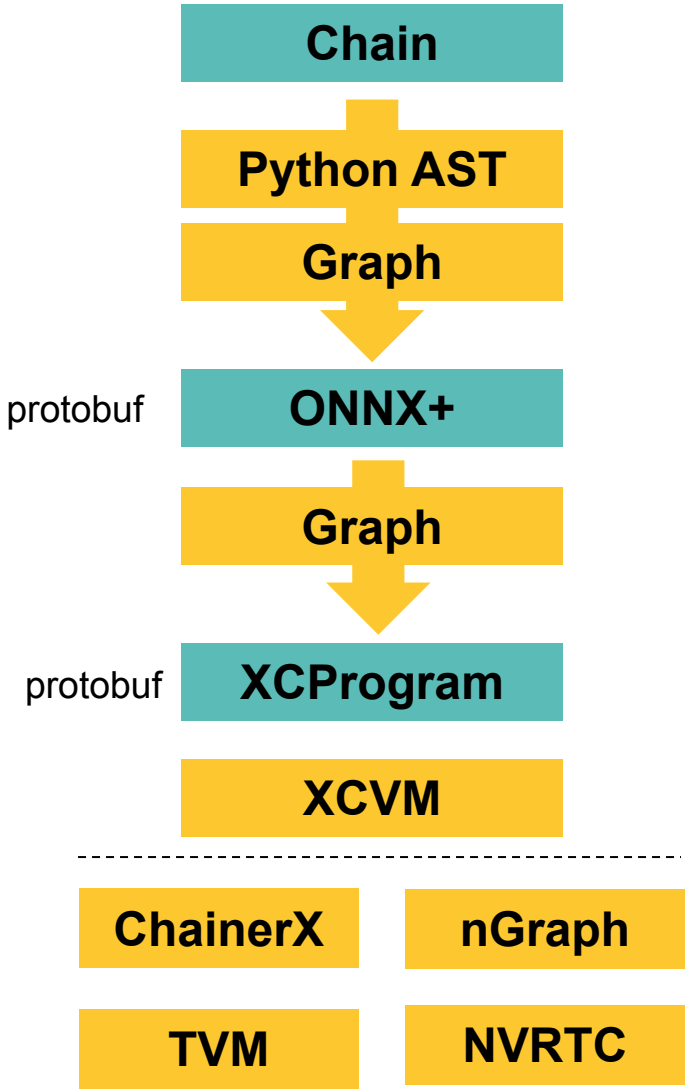
はじめに

- Chainerモデルのさらなる高速化、デプロイの簡便化、可搬性の向上に向けた実験的な取り組みについて | Preferred Research
- Chainerのモデルをコンパイルし最適化、 Pythonレスの推論、学習
- DeepLearningを含む行列演算のアルゴリズム全体を、低レイヤの言語で書き直すことなく高速にプロダクションに投入することを実現するために使っていきたい。
  https://dena.ai/work6/

これだけで、今までどおりChainerのモデルとして学習や推論ができるんだけど
内部では全く異なる仕組みで動く

```
import chainer_compiler

model = chainer_compiler.compile(model, inputs)
y = model(inputs)
```

コンパイルの流れ



ch2o / elichika

forwardメソッド内を ASTに変換
変数や関数を解決しながら再帰的に Graph（内部表現）を構築
GraphからONNX+へ変換

compiler

ONNXをGraphとして読み込み、
operator fusionやスケジューリングなどの Passを通し、
XCVMの命令列へ

runtime

XCVMの命令列を逐次実行するインタプリタ
基本ChainerXが動くが、命令によっては TVMなどが動く

Pythonパッケージchianer_compilerでは、
これらをChainやFunctionNodeでラップし、
既存のPythonコードへの組み込みを容易にしている

Elichika

Elichika

- Chainerの計算グラフをトレースするのではなく、 Pythonスクリプトとして直接パース
  - 普通に書いたモデルがコンパイルできるのは使いやすい
  - L.Linearやrange、np.arrayといった関数呼び出しを対応する仕組みが面白い

```
import chainer
import chainer.links as L

class A(chainer.Chain):

    def __init__(self):
        super(A, self).__init__()
        with self.init_scope():
            self.l0 = L.Linear(7)
            self.l1 = L.Linear(5)

    def g(self, y):
        return self.l1(y)

    def forward(sl, x):
        x1 = sl.l0(x)
        x2 = sl.g(x1)
        return x2
```

```
import chainer

class LazySelfInit(chainer.Chain):
    def __init__(self):
        super(LazySelfInit, self).__init__()
        self.y = None

    def forward(self, x):
        for i in range(x):
            if self.y is None:
                self.y = 42
            self.y += i
        return self.y
```

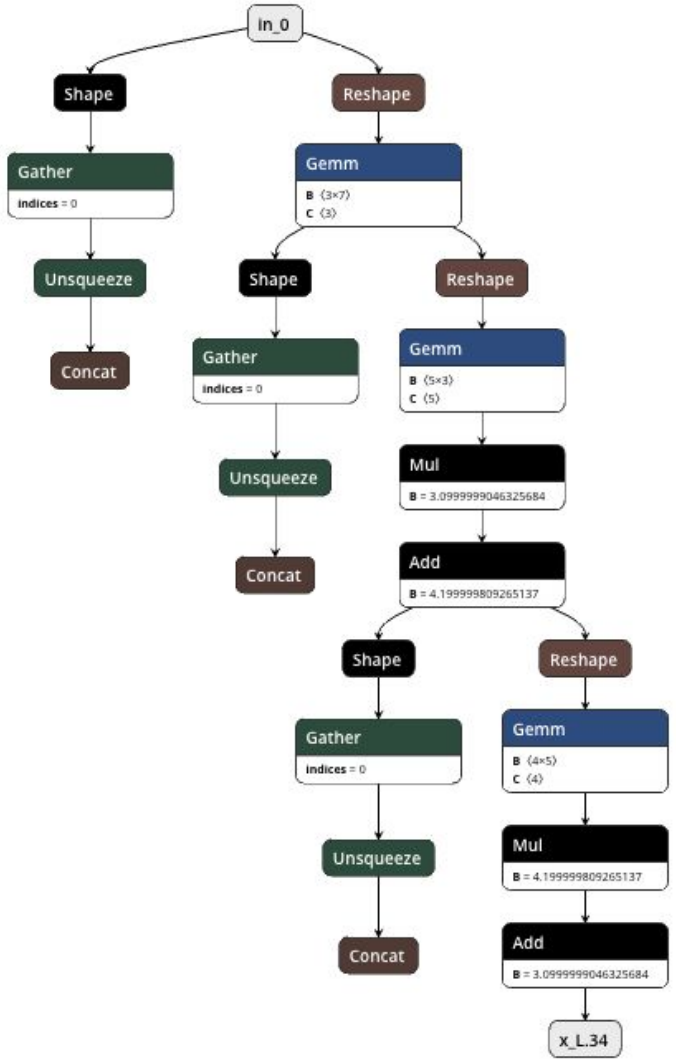
```
class Array(chainer.Chain):
    def forward(self):
        y1 = np.array([4.0, 2z.0, 3.0], dtype=np.float32)
        return y1
```

※elichikaのテストコードより

関数呼び出しを含むケース（ 1）

- Aからユーザー定義クラス Bを呼ぶケース

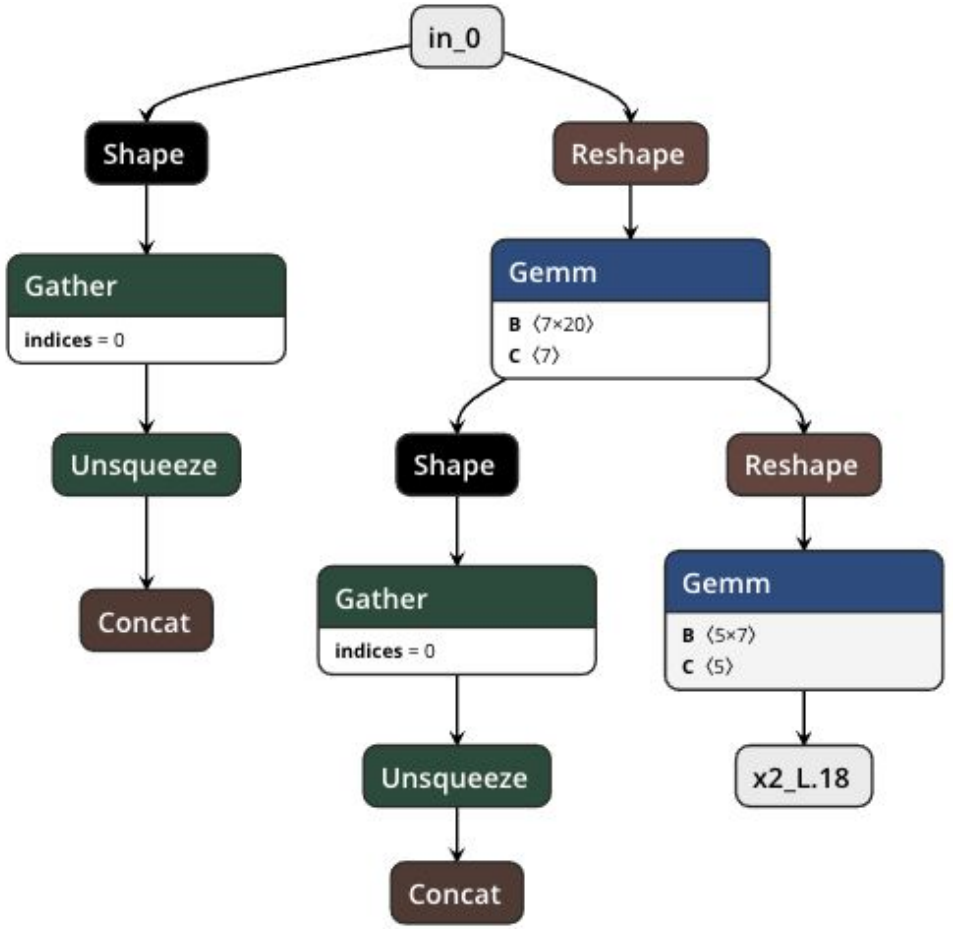
```
class B(chainer.Chain):
    def __init__(self, n_out, p):
        super(B, self).__init__()
        with self.init_scope():
            self.l = L.Linear(None, n_out)
        self.p = p

    def forward(self, x):
        x = self.l(x) * self.p
        return x

class A(chainer.Chain):

    def __init__(self):
        super(A, self).__init__()
        with self.init_scope():
            self.l0 = L.Linear(3)
            self.l1 = B(5, np.float32(3.1))
            self.l2 = B(4, np.float32(4.2))

    def forward(self, x):
        x = self.l0(x)
        x = self.l1(x) + self.l2.p
        x = self.l2(x) + self.l1.p
        return x
```



関数呼び出しを含むケース（ 2）

- メソッド呼び出しの解決

```
class A(chainer.Chain):

    def __init__(self):
        super(A, self).__init__()
        with self.init_scope():
            self.l0 = L.Linear(7)
            self.l1 = L.Linear(5)

    def g(self, y):
        return self.l1(y)

    def forward(sl, x):
        x1 = sl.l0(x)
        x2 = sl.g(x1)
        return x2
```



主要なAPI

- https://github.com/pfnet-research/chainer-compiler/blob/32ef948d11d90a45b86c04a548ea0988fd6f120a/elichika/elichika/chainer2onnx.py#L65-L74
- elichika/chainer2onnx.py::compile_model
- core.convert_model
  - Chainerのモデルのインスタンスから Graphクラスのインスタンスへ
  - forwardメソッドのソースコードを取得、 ASTに変換
    - ASTを1命令ずつ、Nodeに変換しGraphに追加していく
    - シンボルテーブル（ Module、Field）による変数や関数の解決
- onnx_converters.ONNXGenerator#generate_model
  - GraphからONNXGraphへの変換
  - GraphのNodeを繰り返しONNXの対応するOpに変換

```
def forward(self, x):
    h = self.conv1(x)
```

```
Assign(
  target=Name(id='h'),
  value=Call(args=[Name(id='x')],
             func=Attribute(
                  value=Name(id='self'),
                  attr='conv1')))
```

Graph

ONNX+

AST→Graph：関数呼び出し

- 関数呼び出しの gast.Call。シンボルテーブルから参照を解決 し、
  取得したFunctionBase#vcall呼び出しでNodeを作成する
- モジュールトップレベルのシンボルテーブル　Moduleとローカルのテーブル Field
  - テーブルは、inspect.getmembers(...) でスコープのメンバを取得してつくる
  - 呼び出したスコープから親スコープやトップレベルまで順に探す
- F.reluなどのビルトイン関数と、ユーザー定義の関数で処理を分けている
  - F.relu → ONNXのreluのようにONNXのOpと対応する関数
  - range → NodeGenerate→ ChainerSequenceRange のように特別な Opにしたい
  - ユーザー定義は、ResBlockなどブロック単位でクラスや関数に分けたいモデルで有効
- vcall時に前者は1つのNodeCallを作成。後者は 再帰的にAST→Nodeへ変換し展開する
  - ビルトイン関数の場合はシンボルテーブルを上書き し、本来の F.reluを参照しないようにしている

ONNX+ (extended ONNX)

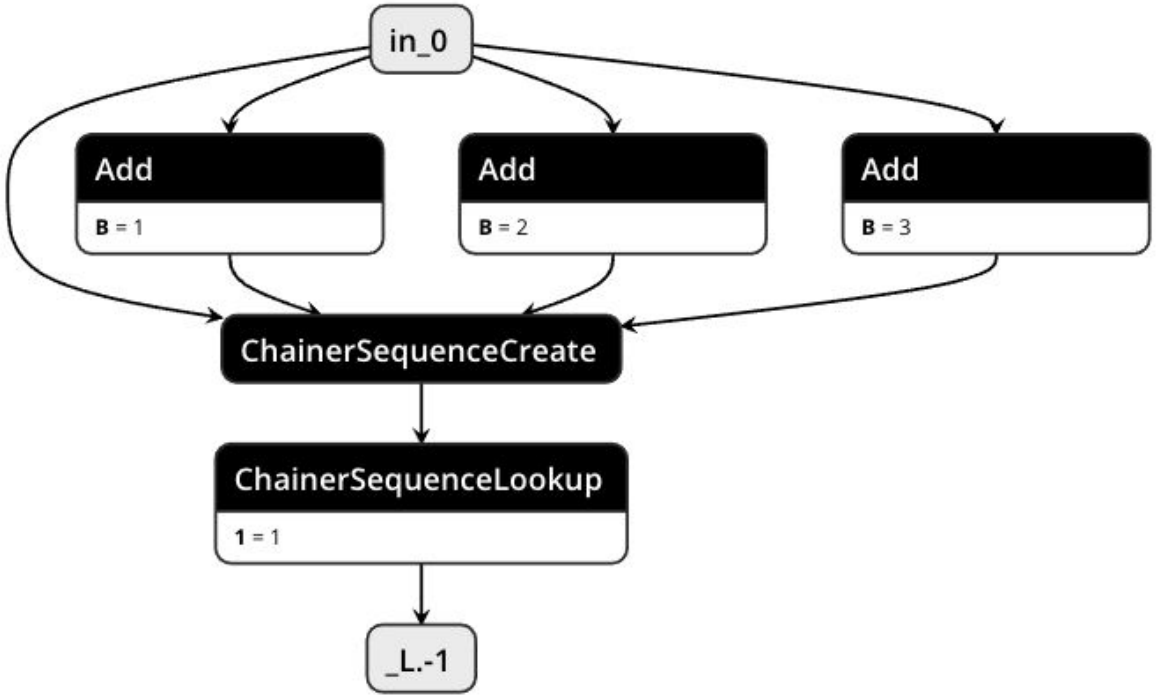
- **すべてをグラフ表現にする上で ONNXを拡張している**
- **Opsetの仕様は、 compiler/gen_node.pyを参照**
- **シーケンス操作や、ONNXが非対応なDeepLearning向けOpsが新規追加**

```
# compiler/gen_node.py
# op_type, num_inputs, num_outputs, **kwargs
NodeDef('Relu', 1, 1)
NodeDef('Gemm', 3, 1, alpha=1.0, beta=1.0, transA=False, transB=False)
NodeDef('ChainerROIMaxPool2D', 3, 1,
        output_shape=[int], spatial_scale=Required(float))
NodeDef('ChainerFusionGroup', None, None, subgraph=Graph, fusion_type=str)
NodeDef('ChainerGetItem', None, 1, slice_specs=[int])
NodeDef('ChainerSequenceCreate', None, 1)
NodeDef('ChainerSequenceLookup', 2, 1)
```

サブグラフをもつグラフは既存のONNXビューワで見れないものも

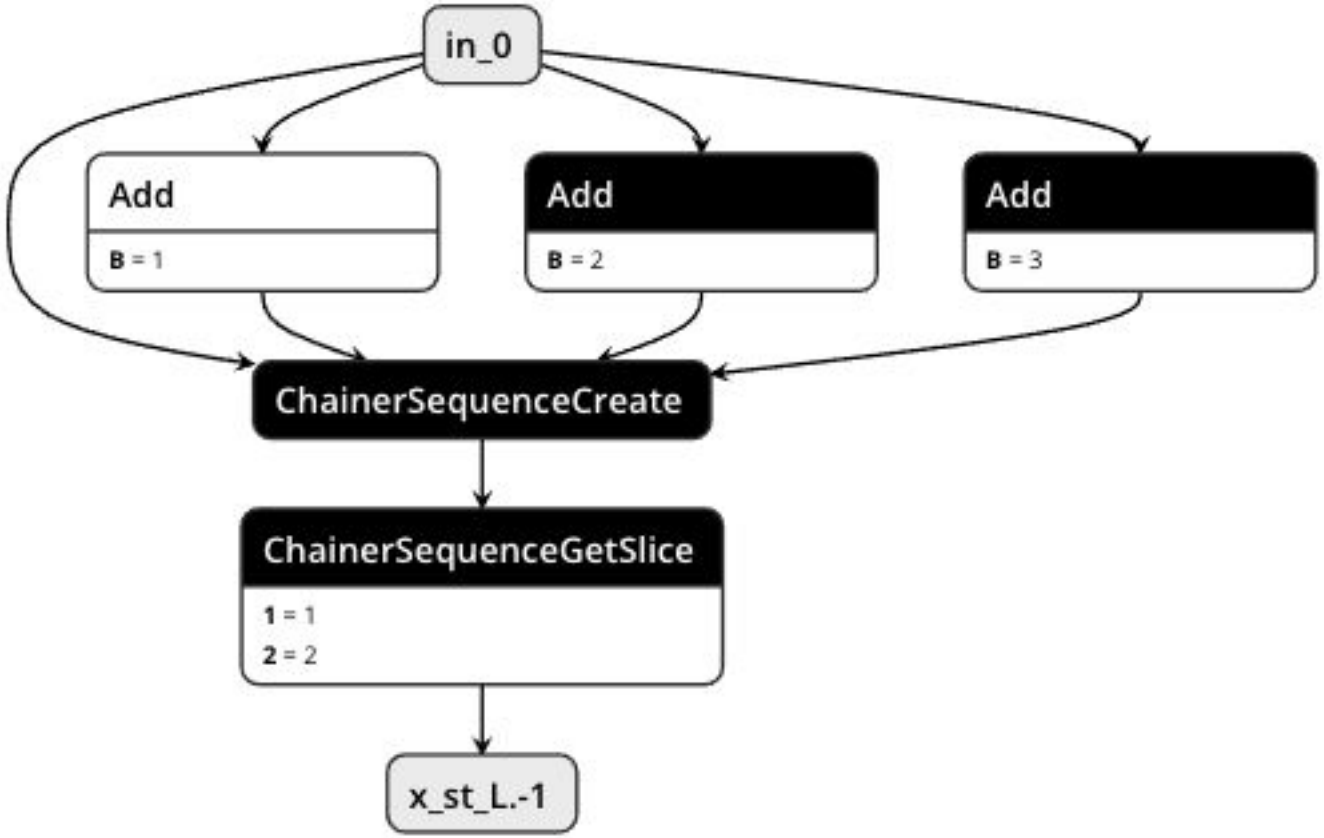
ChainerSequenceXxx Op (1)

```
class Index(chainer.Chain):
    def __init__(self):
        super(Index, self).__init__()

    def forward(self, v):
        x = [v,v+1,v+2,v+3]
        return x[1]
```



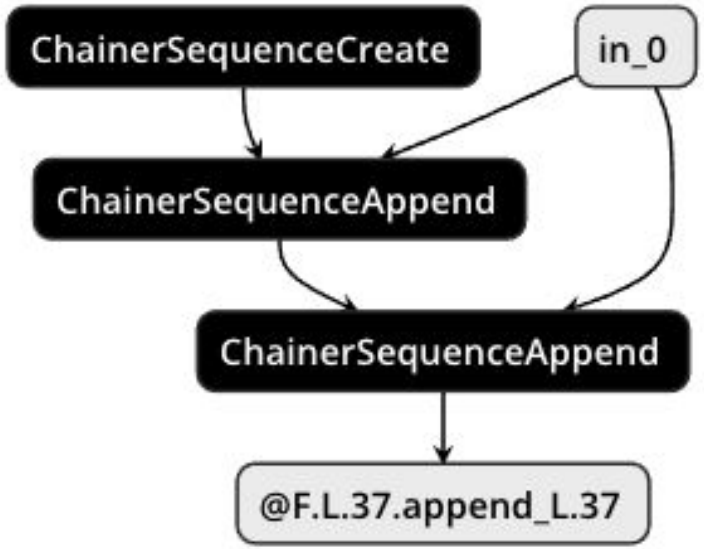
ChainerSequenceXxx Op (2)

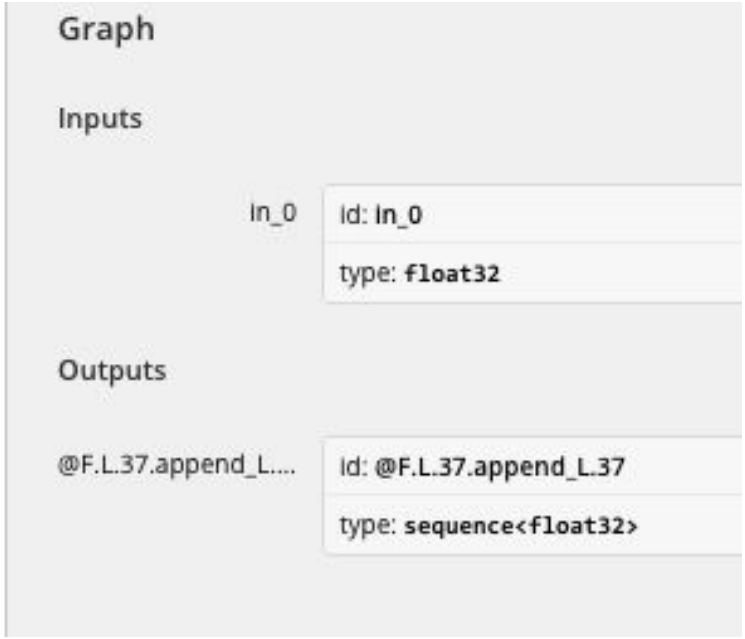
```
class Slice(chainer.Chain):
    def __init__(self):
        super(Slice, self).__init__()

    def forward(self, v):
        x = [v,v+1,v+2,v+3]
        return x[1:2]
```



ChainerSequenceXxx Op (3)

```
class Append(chainer.Chain):
    def __init__(self):
        super(Append, self).__init__()

    def forward(self, v):
        x = []
        x.append(v)
        x.append(v)
        return x
```





GraphからONNX+へ

- onnx_converters.ONNXGenerator#generate_model
- ONNXGeneratorが、GraphのNode種別に応じてONNXのNodeに変換していく
- NodeBinOp → Add, Sub, Mul, … のように決まってるが、
- NodeCallは、FunctionやLinkは予め登録したテーブルから引いて変換

```
for node in graph.nodes:
    if isinstance(node, nodes.NodeCopy):
        node_ = node  # type: nodes.Copy
        onnx_node = oh.make_node(
            'Identity',
            [value2onnx_parameter[node_.value].onnx_name],
            [value2onnx_parameter[node.outputs[0]].onnx_name])

        onnx_graph.nodes.append(onnx_node)

    if isinstance(node, nodes.NodeAugAssign):
        convert_node_aug_assign(onnx_graph, node)

    if isinstance(node, nodes.NodeBinOp):
        convert_node_bin_op(onnx_graph, node)

    if isinstance(node, nodes.NodeUnaryOp):
        convert_node_unary_op(onnx_graph, node)
```

NodeCallのONNX変換（F.reluの例）

- convert_node_callでCallNodeからONNX Graphへ
- おなじNodeCallでも以下に応じて分岐
  - chainer.functions, chainer.links, ndarray.{shape, size}, append
- 予めChainerのインスタンス から、Graph→ONNXへの変換関数の dictを用意
  - compile_modelの最初にテーブルを作成。ここに登録した Opをサポート
  - CallNodeはchainerのインスタンスも保持しているのでそれでテーブルを引く
- ONNXのGraphオブジェクトを wrapしたONNXGraphのヘルパーで、変換
  - F.elu(alpha=1.0)など引数に持つ関数も仕組みとしては対応しているように見える

```
def convert_relu(onnx_graph, node):
    onnx_graph.add_node('Relu',
                        [node.inputs[0]],
                        [node.outputs[0]],
                        name=str(node.lineprop))
```

if → NodeIf → IF

- IfはNodeのAttributeにGraph（サブグラフ）を持つので、再帰的に Graph→XGraphへ

```
#ONNXGenerator#generate_graph
if isinstance(node, nodes.NodeIf):
    node_ = node  # type: nodes.NodeIf

    true_graph = self.generate_graph(node_.true_graph.xxx ...)
    false_graph = self.generate_graph(node_.false_graph.xxx ...)

    onnx_node = oh.make_node(
        'If',
        [value2onnx_parameter[node_.cond].onnx_name] + ... ,
        [value2onnx_parameter[x].onnx_name for x in node.outputs],
        then_branch=true_graph,
        else_branch=false_graph)

    onnx_graph.nodes.append(onnx_node)
```

```
graph {
  node {
    op_type: "If"
    attribute {
      name: "then_branch"
      g {
        name: "True"
        node {...}
      }
      type: GRAPH
    }
    attribute {
      name: "else_branch"
      g {
        name: "False"
        node {...}
      }
      type: GRAPH
    }
  }
```

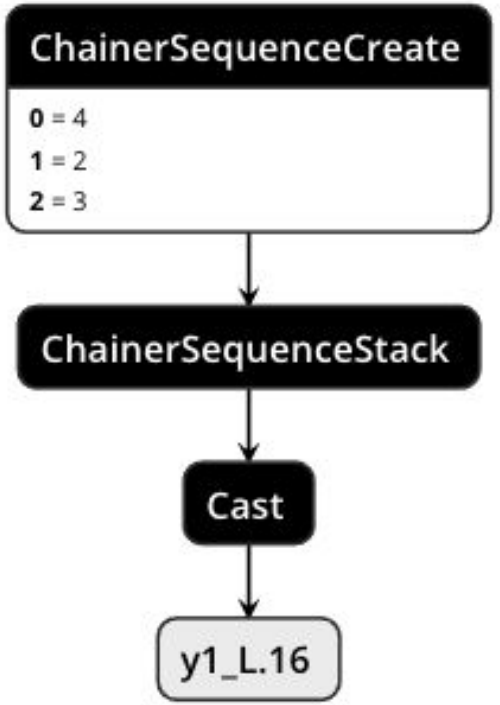
※ AttributeにGraphを持つことはONNXの仕様上OK
※ netronだとサブグラフの可視化できない

np.array → NodeGenerate → ChainerSequenceStack

- **np.arrayはONNX+に変換できる**
- **これもchainer.functions同様、シンボルテーブル Moduleを NDArrayFunction(functions.FunctionBase)で置き換えているため**

```
class ArrayCast(chainer.Chain):
    def forward(self):
        y1 = np.array([4.0, 2.0, 3.0], dtype=np.int32)
        return y1
```

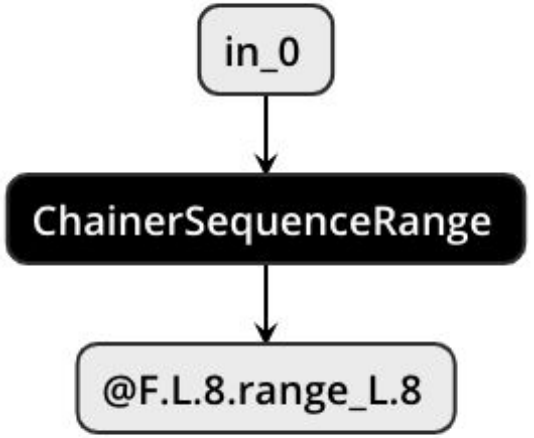
```
# Stacks elements in a sequence: ([T]) -> (T)
NodeDef('ChainerSequenceStack', 1, 1, axis=0)
```



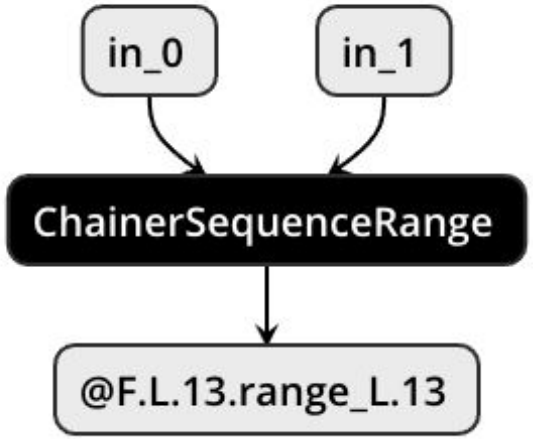
range → NodeGenerate → ChainerSequenceRange

- rangeもnp.arrayと同じNodeGenerateだが、別の ONNX Opへ
  - Node作成時に与える class typeでONNX変換時に区別

```
class Range(chainer.Chain):
    def forward(self, x):
        return range(x)
```




```
class RangeStop(chainer.Chain):
    def forward(self, x, y):
        return range(x, y)
```



3引数までオプションで取りますということかな

```
# Equivalent to Python's range.
NodeDef('ChainerSequenceRange', (1, 2, 3), 1)
```

Compiler

Compiler

- **ONNX+をGraphにパースし、XCProgram（XCVM Op）に変換する。ここは C++。**
- **Graphに対して、Passとして、正規化や Fuse、Opのスケジューリングなどを適用**

```
// ONNX Modelから内部表現へ変換
// Model, Graph, Node, Value, Tensor, それぞれONNXから変換
onnx::ModelProto xmodel(LoadLargeProto<onnx::ModelProto>(model_path));
Model model(xmodel);
RunDefaultPasses(&model);

// ModelをXCVMの命令列へ
std::ostringstream oss;  // protobufのシリアライズされたメッセージとして格納
xcvm::Emit(model, oss);
runtime::XCProgramProto program;
program.ParseFromString(oss.str());
```

XCVM Ops

- **ONNX+のOpset相当に加え、制御は JmpOpになり、In,Out,FreeのようなVMのバッファ管理用のOpや TVMOp, NGraphOpというOpも。144命令ほど**
- **compiler/xcvm/gen_xcvm_codegen.pyで生成。protobuf**
- **runtime/xcvm_defs.py**

```
// プログラム全体
message XCProgramProto {
    repeated XCInstructionProto instructions = 1;
    repeated string input_names = 2;
    repeated XCTypeProto input_types = 3;
}
```

```
// 個々の命令
message XCInstructionProto {
    required Op op = 1;
    repeated XCValueProto inputs = 2;
    repeated int32 outputs = 3;
    optional string debug_info = 4;
    optional int64 id = 5;
    repeated XCTypeProto output_types = 6;
    repeated string output_names = 7;
}
```

GraphからXCVM Opへ

- **CompileModel → Emit → EmitModel → EmitGraph → EmitNode**
- **Nodeのop_typeに応じて、AddXxxOp()をコール AddXxxOpは定義から生成**

```
void EmitNode(const Graph* graph,
const Node& node, XCProgramProto* prog) {
...
#define EMIT(op, ...)                                  \
    do {                                               \
        Add##op##Op(prog, __VA_ARGS__);                \
        FillOpInfo(node, node.ToString(), prog); \
    } while (0);
...
// Node → XCVMOp
} else if (node.op_type() == Node::kMatMul) {
    EMIT(MatMul, out(0), in(0), in(1));
} else if (node.op_type() == Node::kGemm) {
...
```

```
void AddReluOp(
    runtime::XCProgramProto* program,
    XCVMValue y, int x) {
    XCInstructionProto* inst = program->add_instructions();
    inst->set_op(XCInstructionProto::Relu);
    {
        XCValueProto* input_proto = inst->add_inputs();
        input_proto->set_type(XCValueProto::ARRAY);
        input_proto->set_array(x);
    }

    y.AddOutput(inst);
}
```

最適化: RunDefaultPasses

- InferAllDtypeAndShape
- CanonicalizeSubGraphs
- Simplify
- PropagateConstants
- EvaluateShapes
- AddGradientNodesForTraining
- GetComputationOrder
- AddGradientNodesForTrainingWithOrders

Simplify

- **Node毎にReplace関数を実行し、GraphBuilderで書き換えた新しい Nodeを作成**
- **元のNodeはDetach（あとでCollectGarbageNodeのパスで削除される）**

```
bool replaced = true;
while (replaced) {
    replaced = false;
    for (Node* node : graph->GetLiveNodes()) {
        auto found = simplifiers.find(node->op_type());
        if (found == simplifiers.end()) continue;
        // ここでReplaceHoge関数を呼ぶ
        if (found->second(graph, node)) {
            graph->DetachNode(node);
            replaced = true;
        }
    }
}
```

別の命令への置き換えや、Identityの削除など

- Min→Max
- Flatten→Reshape
- Grouped Conv→ Split+Conv*N+Concat

例えばXCVMとしてMinを実装しないで済む

最後のGrouped Convは、nGraph有効時のみその置き換えが走る。

Simplifyの例: ReplaceMin

- Min(A) → - Max(-A) に置き換える
- XCVMのOpがMaxしかサポートしていないため
- NodeとXCVM Opの対応はEmitNode参照（例 Node::kNeg →Neg ）

```
// Min(A) -> Neg(Max(Neg(A)))
bool ReplaceMin(Graph* graph, Node* node) {
    CHECK_EQ(1UL, node->outputs().size());
    GraphBuilder gb(graph, "SimplifyMin", node->output(0));
    std::vector<Value*> negs;
    for (Value* v : node->inputs()) negs.push_back(gb.Op(Node::kNeg, {v}));
    Value* r = gb.Op(Node::kMax, negs);
    gb.Op(Node::kNeg, {r}, node->output(0));
    return true;
}
```

FuseOperations

- **TVM, NGraph, NVRTCを利用したOperator Fusion**

```
std::set<Node*> fused_nodes = {base_node};

Node* node = base_node;
while (true) {
    // fuse可能なNodeをfused_nodesにためる
}
// 同一のfusion_groupにセット
++num_fusion_groups;
for (Node* node : fused_nodes) {
    node->set_chainer_fusion_group(num_fusion_groups);
}
// Node::kChainerFusionGroupを作成。fused_nodesをサブグラフとして持つ
// XCVMのOpを生成するEmit時に、tvmやNGraphでコンパイルされる
// 実行時はXCVM上だが、TVMならPackedFuncが呼び出されるTVMOpとして実行
CreateFusionGroup(graph, fused_nodes, "tvm", num_fusion_groups);
```

TVM AutoTuning

- Fuse時にTVMを使う場合、TVMのConvとしてコンパイルされる
- このときの ScheduleにAutoTuning結果のログを渡すことができる
- コンパイル時に Auto Tuning用のタスクを出力するオプションと
  それを使った Tuningスクリプト（ tune_task.py）もある
  - g_dump_autotvm_task_dir, g_autotvm_logのオプション
- Chainerのモデルを TVMのグラフとして
  プラットフォームに合わせて AutoTuningできるのは面白い

ScheduleComputation

- 計算の依存関係を考慮した上で、 Nodeを並び替える
- ここでソートした順番で XCVM Opが出力され、ランタイムの処理順序となる
- ScheduleNaively
  - トポロジカルソートした先頭の Nodeから順に計算する
- ScheduleGreedy
  - そのノードの計算前後でどれだけメモリ使用量が増えるかを計算し、小さい順に計算
  - A greedy scheduler which tries to reduce the current working memory in greedy mannar.

Runtime

XCVM

- **chainer_compiler_core.XCVMとしてラップされている**

```
void XCVM::Run(XCVMState* state) {
  state->SetProgram(&program_);

  while (true) {  // XCVM Opを順に実行するインタプリタ
    int pc = state->pc(); // プログラムカウンタで制御
    if (pc >= program_.size()) break;

    XCVMOp* op = program_[pc].get();

    // 各OpはChainerXやTVM、NGraphで動く
    // stateで入出力Arrayなどを管理。str-> XCVMVarのmap
    op->Run(state);

    state->set_pc(state->pc() + 1);
  }
```

制御構造

- IfやLoopはXCVMOpではより低レベルな Jmp命令。プログラムカウンタの制御

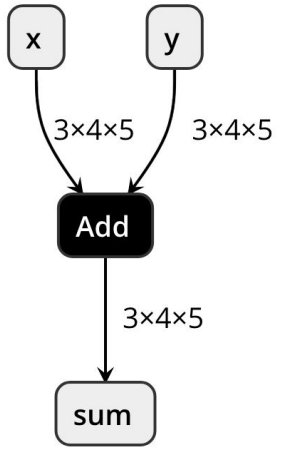
```
void JmpOp::RunImpl(XCVMState* st) {
    st->set_pc(pc - 1);
}

void JmpTrueOp::RunImpl(XCVMState* st, const chainerx::Array& cond) {
    if (static_cast<bool>(chainerx::AsScalar(cond))) {
        st->set_pc(pc - 1);
    }
}

void JmpFalseOp::RunImpl(XCVMState* st, const chainerx::Array& cond) {
    if (!static_cast<bool>(chainerx::AsScalar(cond))) {
        st->set_pc(pc - 1);
    }
}
```

XCVMOpの実装例: ReluOp

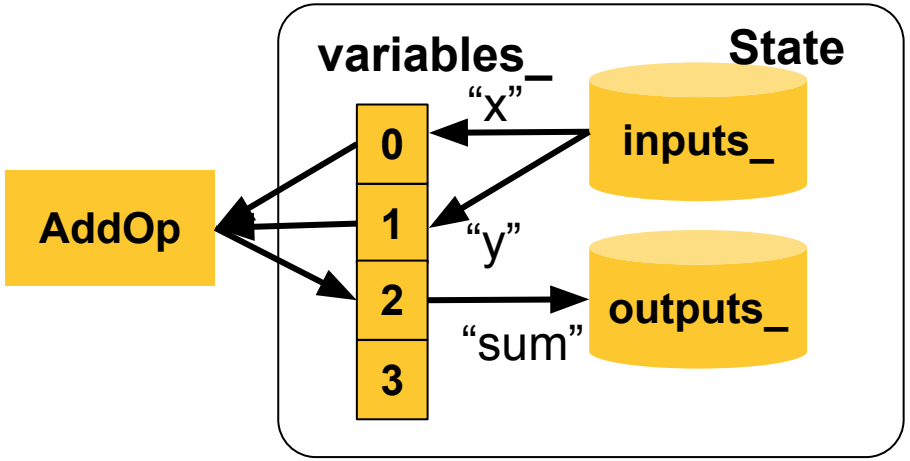
```
// XCVMOpのサブクラスとして実装する
// XCInstructionProtoで入出力
ReluOp::ReluOp(const XCInstructionProto& inst): XCVMOp(inst) {
    x = inst.inputs(0).array();  // x, yはprivateなメンバ(int)
    y = inst.outputs(0);         // 入出力の一時バッファのインデクス
}

// 計算の呼び出し
void ReluOp::Run(XCVMState* st) {
    // 一時バッファから入力のchainerx::Arrayを取得し計算後、結果を書き込み
    st->SetArray(y, RunImpl(st, st->GetArray(x)));
}

// ChainerXのAPIで計算
chainerx::Array ReluOp::RunImpl(
    XCVMState* st, const chainerx::Array& x) {
    return chainerx::Maximum(x, 0);
}
```

バッファ管理

- 入出力の制御を単純にするため一時バッファを使い、その制御用　Opを利用




```
XCProgramProto program;
xcvm::AddInOp(&program, xcvm::XCVMValue(0), "x");
xcvm::AddInOp(&program, xcvm::XCVMValue(1), "y");
xcvm::AddAddOp(&program, xcvm::XCVMValue(2), 0, 1);
xcvm::AddFreeOp(&program, 0);
xcvm::AddFreeOp(&program, 1);
xcvm::AddOutOp(&program, "sum", 2);
xcvm::AddFreeOp(&program, 2);
```



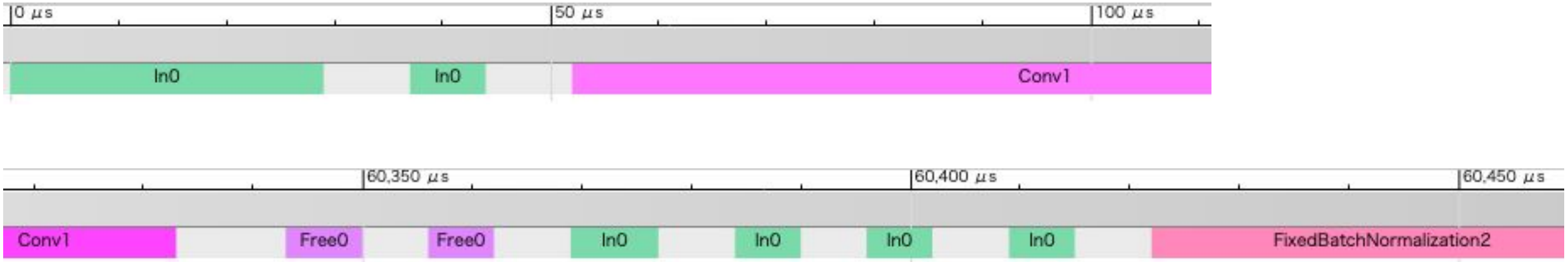
XCVMStateのprivateメンバ

```
int pc_; // プログラムカウンタ
std::vector<std::unique_ptr<XCVMVar>> variables_;
//変数名→XCVMVar（chainerx::Array）のmap
InOuts inputs_; // グラフの入力。
InOuts outputs_;  // グラフの出力
XCVMOptions options_;  // トレースレベルなど挙動の管理
const std::vector<std::unique_ptr<XCVMOp>>* program_;
```

AddOpは、入出力のArrayが変わるとしても、バッファの0,1,2に書くだけ

バッファ管理

- chromeのトレースを見てみると、たしかに InやFreeが入っている
- Opの入出力と一致していない場合もあるので、最適化されているのかも



TVMOp

- **XCVMOpのサブクラスとして実装しており、他の Opの間で呼び出せる**

```
void TVMOp::InitImpl() {
    impl_ = new TVMImpl();
    impl_->fn = LoadPackedFunc(dso_filename, func_name);
}

std::vector<chainerx::Array> TVMOp::RunImpl(
chainer_compiler::runtime::XCVMState* st, const std::vector<chainerx::Array>& orig_inputs) {

    tvm::runtime::TVMArgs tvm_args(tvm_values, tvm_type_codes, num_args);
    tvm::runtime::TVMRetValue tvm_ret;
    impl_->fn.CallPacked(tvm_args, &tvm_ret); // ここでPackedFuncを呼ぶ

    return outputs;
}
```

runtime/ops/tvm.cc

NGraphOp

```
void NGraphOp::InitImpl() {
    impl_ = new NGraphImpl();
    std::istringstream iss(onnx);
    impl_->func = ngraph::onnx_import::import_onnx_model(iss);
    ...
    const char* kBackend = "CPU";
    impl_->backend = std::move(ngraph::runtime::Backend::create(kBackend));

    impl_->handle = impl_->backend->compile(impl_->func);
    ...
}

std::vector<chainerx::Array> NGraphOp::RunImpl(
chainer_compiler::runtime::XCVMState* st, const std::vector<chainerx::Array>& orig_inputs) {
    ...
    impl_->handle->call_with_validate(impl_->result_tensors, arg_tensors);
    ...
    return impl_->outputs;
}
```

runtime/ops/ngraph.cc

EOF