CSCD70 Compiler Optimization

Tutorial #4 Dataflow Analysis (ii)

Bojian Zheng
bojian@cs.toronto.edu

Department of Computer Science, University of Toronto

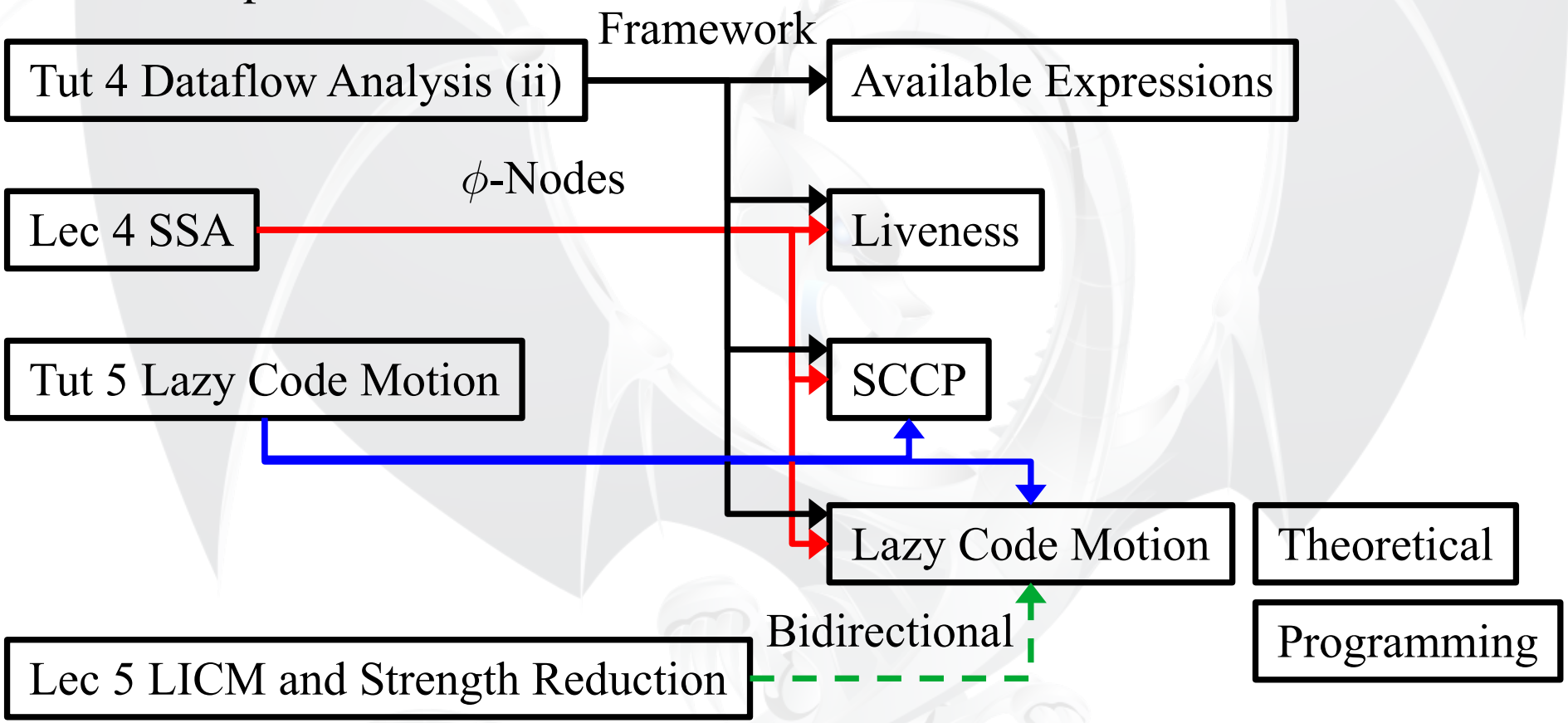
Acknowledgement: Thanks to Professor Gennady Pekhimenko, Professor Nandita Vijaykumar, and students from previous offerings of CSCD70.

Announcement

- Assignment 1 has been graded.
- Assignment 2 will be released this Friday, due on **Mar. 3rd** (Friday).
- Midterm is tentatively scheduled to be on **Mar. 6th**.
  - Please plan accordingly.

Announcement

Our roadmap:



Assignment 1 Feedback

- Posted on the GitHub classroom PR.
  - Please do NOT merge it.
- Please include your name in the report.
- Common Problems:
  - Programming
    - `CHECK` directives have to be completed.
  - Theoretical Questions
    - CFG should be illustrated in a hierarchical way.
    - Available expressions (and other dataflow analyses that are on expressions) should be in the form of binary operators, not statements.

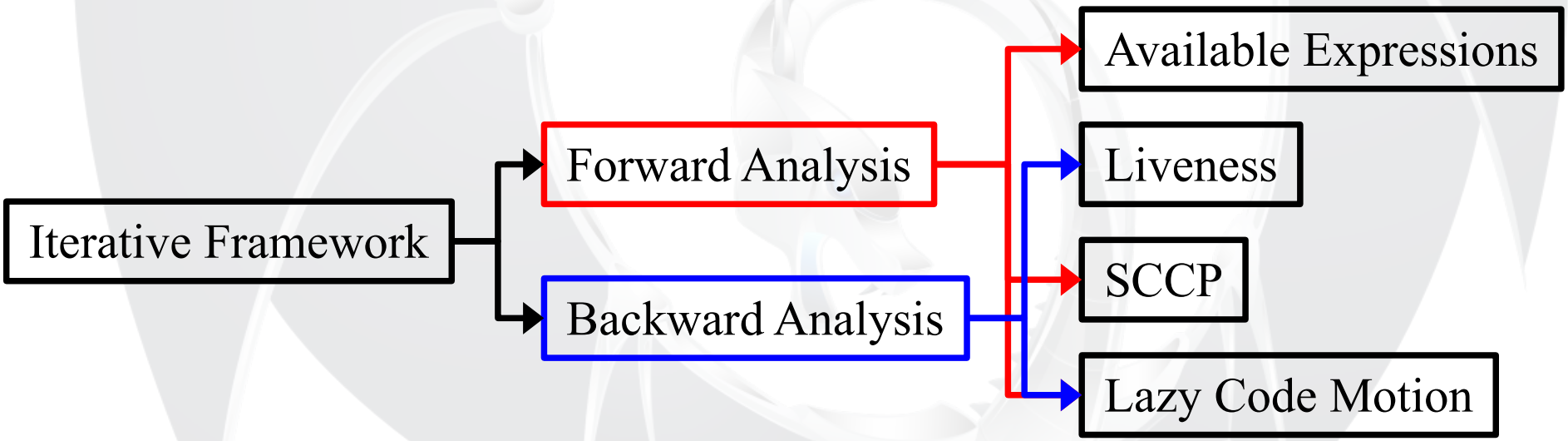
TL'DR

```
                // Available Expressions:
                //   {}
x = a + b;
                //   {a + b}
y = a + c;
                //   {a + b, a + c}
```

Iterative Framework $\rightarrow$ Available Expressions

Iterative Framework

Iterative Framework



Iterative Framework

```
template <typename TDomainElem, typename TValue, typename TMeetOp,
          typename TBBRange, typename TInstRange> ❶
class Framework {
  set<TDomainElem>                    Domain;
  map<BasicBlock, vector<TValue>>     BBDomainValMap; ❷
  map<Instruction, vector<TValue>>    InstDomainValMap;
  void run(const Function&); ❸
};
```

❶ Domain, Value, and Meet Operator

E.g., For available expressions, Domain is binary expressions, Value is boolean (`true` for available, `false` otherwise), and Meet Operator is AND.

❷ Mapping from basic block/instruction to domain values

❸ Top-level `run` function

`run(const Function&)`

Algorithm 1: runOnFunction

Data: Domain D, Instruction-Domain Value Mapping M

initialize *Domain* ❶;
initialize *Instruction-Domain Value Mapping* ❷;
do
  traverse through the CFG update M $\forall \text{inst} \in F$;
while *NOT converge*;

❶ Fill the domain with binary expressions.

❷ How to initialize? ☞ `MeetOp::top() -> vector(D.size(), true)`

CFG Traversal

Algorithm 2: traverseCFG

Data: Domain D, Instruction-Domain Value Mapping M
Arguments : Function F
Return : Whether M has been modified

for $bb \in \text{BBRange}(F)$ ❶ **do**
  ❷
  if *bb has no meet operands* **then**
    initialVal $\leftarrow$ Boundary Condition;
  else
    initialVal $\leftarrow$ MeetOp(MeetOperands(bb));
  ❸
  inputVal $\leftarrow$ initialVal;
  for *each instruction* $i \in \text{InstRange}(bb)$ **do**
    TransferFunc(i, inputVal, $M[i]$);
    inputVal $\leftarrow M[i]$;

❶ Traversal order? ☞ Reverse post-order for forward analysis

❷ Apply the boundary condition if there is no predecessor, or the meet operator otherwise.

❸ Apply the transfer function ∀instruction.

Available Expressions

Forward Analysis

```
template <typename TDomainElem, typename TValue
          typename TMeetOp>
class ForwardAnalysis :
    public Framework< ❶
        TDomainElem, TValue, TMeetOp,
        iterator_range<Function::iterator>,  // BBRange ❷
        iterator_range<BasicBlock::iterator> // InstRange
    > {
  BBRange getBBRange(const Function &F) const {
    return make_range(F.begin(), F.end());
  }
  InstRange getInstRange(const BasicBlock &BB) const {
    return make_range(BB.begin(), BB.end());
  }
};
```

❶ Inherit from the `Framework` class.

❷ The starter code uses the default iterator. You will have to change the type names and the method definitions.

Domain

```
class Expression {
  const unsigned Opcode;
  const Value *const LHS = nullptr;
  const Value *const RHS = nullptr;

  class Initializer : InstVisitor<Initializer> ❶ {
    void visitBinaryOperator(BinaryOperator &);
  };
};
```

❶ `InstVisitor` allows us to only handle instructions of interest.

Meet Operator

```
template <typename TValue>
class Intersect {
  static vector<TValue>
      apply(const vector<TValue>&,
            const vector<TValue>&); ❶
  static vector<TValue> top(const size_t); ❷
};
```

❶ Apply the meet operator (i.e., $\cap$), invoked by the framework on the basic block boundary.

❷ Return the top element, invoked by the framework when initializing the instruction-domain value mapping.

Meet Operation

Algorithm 3: MeetOp

Data: Domain D, Instruction-Domain Value Mapping M
Arguments : BasicBlock bb
Return : Merged BitVector

return $\bigwedge_{i \in \text{MeetOperands}} (\overbrace{M[\text{back}(\text{pred}(bb))]}^{\text{❶ MeetOperands}})$;

❶ The meet operands are the outputs (i.e., the domain value that corresponds to the last instruction) of the predecessor basic blocks.

Available Expressions

```
class AvailExpr :
    public ForwardAnalysis<Expression, Available/*bool*/,
                           Intersect> {
  bool transferFunc(const Instruction&,
                    const vector<Available>&,
                    const vector<Available>&);
};
```

Transfer Function

Algorithm 4: transferFunc

Data: Domain D, Instruction-Domain Value Mapping M
Arguments : Instruction i, BitVector bv_i, bv_o
Return : Whether bv_o has been modified

❶ $bv'_o \leftarrow bv_i$;
❷ ———————————————————————————
for *each element* $e \in D$ **do**
  if $\text{lhs}(e) = i$ **or** $\text{rhs}(e) = i$ **then**
    $bv'_o[\text{position}(e)] = \text{false}$;
❸ ———————————————————————————
$\text{iter} \leftarrow \text{find}(D, \text{Expression}(i))$;
if $\text{iter} \neq \text{end}(D)$ **then**
  $bv'_o[\text{position}(e)] = \text{true}$;
❶ ———————————————————————————
$\text{hasChanges} \leftarrow bv'_o = bv_o?$;
$bv_o \leftarrow bv'_o$;
return hasChanges;

❶ Check whether the bitvector has been modified or not.
❷ Kill expressions whose LHS/RHS are the current assigned value (?).
❸ Set the expression as generated.

Recap

- Iterative Framework
- Available Expressions
  - Forward Analysis
  - Expression: `InstVisitor`
  - Intersect
  - Transfer Function

☞ **Homework Assignment**: Available Expressions