Compiler

Static Analysis: Monotone Dataflow Framework

Dataflow Framework

- We have seen
  - Reaching Definition (RD)
  - Available Expressions (AE)
  - Very Busy Expressions (VBE)
  - Live Variables (LV)
- Based on their similarities, we can generalize to data flow frameworks

Observation

	RD	AE	VBE	LV
Direction	forward	forward	backward	backward
Solution	least	greatest	greatest	least
b_{init} (f) / b_{last} (b) Value	$\{ (x,\bullet) \mid x \in (Param_* \cup Field_*)\} \cup \{ (x,?) \mid x \in Local_* \}$	$\emptyset$	$\emptyset$	$\emptyset$
Combining Operator	$\cup$	$\cap$	$\cap$	$\cup$
Description	may	must	must	may
Paths	some	all	all	some

RD Flow Equations

$$\mathrm{RD}_{entry}(l) = \begin{cases} \{ (x,\bullet) \mid x \in (\mathrm{Param}_* \cup \mathrm{Field}_*)\} \cup \{ (x,?) \mid x \in \mathrm{Local}_* \}, \text{ if } l = b_{init} \\ \bigcup \{ \mathrm{RD}_{exit}(l') \mid l' \in preds\,(l) \}, \text{ otherwise} \end{cases}$$

$$\mathrm{RD}_{exit}(l) = (\mathrm{RD}_{entry}(l) \setminus kill_{\mathrm{RD}}(l)) \cup gen_{\mathrm{RD}}(l)$$

AE Flow Equations

$$AE_{entry}(l) = \begin{cases} \emptyset, \text{ if } l = b_{init} \\ \bigcap \{ AE_{exit}(l') \mid l' \in preds(l) \}, \text{ otherwise} \end{cases}$$

$$AE_{exit}(l) = (AE_{entry}(l) \setminus kill_{\mathrm{AE}}(l)) \cup gen_{\mathrm{AE}}(l)$$

VBE Flow Equations

$$\mathrm{VBE}_{exit}(l\,) = \begin{cases} \emptyset, \text{ if } l = b_{last} \\ \bigcap \{\, \mathrm{VBE}_{entry}(l'\,) \mid l' \in succs\,(l\,) \,\}, \text{ otherwise} \end{cases}$$

$$\mathrm{VBE}_{entry}(l\,) = (\mathrm{VBE}_{exit}(l\,) \setminus kill_{\mathrm{VBE}}(l\,)) \cup gen_{\mathrm{VBE}}(l\,)$$

LV Flow Equations

$$LV_{exit}(l) = \begin{cases} \emptyset, \text{ if } l = b_{last} \\ \bigcup \{ LV_{entry}(l') \mid l' \in succs(l) \}, \text{ otherwise} \end{cases}$$

$$LV_{entry}(l) = (LV_{exit}(l) \setminus kill_{\mathrm{LV}}(l)) \cup gen_{\mathrm{LV}}(l)$$

Equation Systems

- All of the equation systems have similar form

$$\text{Analysis}_{in}(l\ ) = \begin{cases} \iota, \text{ if } l = b_\iota \\ \\ □\ \{\ \text{Analysis}_{out}(l'\ )\ |\ l' \in \mathit{next}\ (l\ )\ \},\ \text{otherwise} \end{cases}$$

$$\text{Analysis}_{out}(l\ ) = f_l\,(\text{Analysis}_{in}(l\ ))$$

…where □, *next*, b_ι, ι, and f_l are parameters of the framework

Generalization of Classic Problems

- For our problems the parameters ranged over a small set of values
  - □ is $\cap$ or $\cup$
  - *next* is *preds* or *succs*
  - b_ι is b_{init} or b_{last}
  - ι gives the analysis information for b_ι, i.e., $\text{Analysis}_{in}(b_\iota) = \iota$
  - f_l is the transfer associated with block l, e.g., $\text{Analysis}_{in}(l) \setminus kill_{\text{Analysis}}(l) \cup gen_{\text{Analysis}}(l)$

Forward vs. Backward Analysis

Param	Forward Analysis	Backward Analysis
next	*preds*	*succs*
b_ι	b_{init}	b_{last}
ι	analysis information for b_{init}	analysis information for b_{last}
Analysis_{in}	statement entry information	statement exit information
Analysis_{out}	statement exit information	statement entry information

Greatest vs. Least Solution Problems

- Greatest Solution
  - □ is $\cap$
  - aka *must* analysis, *all paths/universal* analysis
- Least Solution
  - □ is $\cup$
  - aka *may* analysis, *some paths/existensial* analysis

Definition of Dataflow Framework

- Dataflow facts are termed *properties*
- The set of dataflow facts are termed a *property space* and denoted as *L*
- Typically *L* will be a complete lattice
- Property sets flowing along different paths are combined according to a *combining* operator □: $P(L) \rightarrow L$

Ascending Chain Condition

- An ascending chain in a lattice is a sequence of ordered values from the lattice, $(l_1, \ldots, l_n)$ s.t. $\forall\ 0 < i < n.\ l_i \sqsubseteq l_{i+1}$
- We require that all such chains for property spaces eventually stabilize, i.e., $\exists\ n.\ \forall\ i \geq n.\ l_n = l_i$
- Do the classic examples satisfy the condition?

Transfer Functions

- The set of transfer functions, $f_l : L \rightarrow L$ for $l \in \mathbf{Lab}_*$, is contained in the function space F
- In addition to the transfer functions F has the following properties
  - F contains the identity function
  - F is closed under function composition
- We usually require monotonicity which is
  - sensible as it says that an increase in data flow information at an input cannot produce a decrease in information at the associated output
  - necessary for the development terminating flow analysis algorithms

A Monotone Dataflow Framework

- consists of
  - a complete lattice *L* satisfying the ascending chain condition
  - a set *F* of monotone functions from *L* to *L* that contains the identity and is closed under composition
  - Note: this is the most common type of framework, but not the only one that is used in practice.

A Framework Instance

- Parameterizes the abstract framework with information about the program to be analyzed.
- An instance consists of
  - complete lattice *L* and function space *F*
  - a finite flow relation *next* (e.g., *succs* or *preds*)
  - an extremal label b_ι (i.e., b_{init} or b_{last})
  - an extremal value $\iota \in L$
  - a mapping, *f* , from the labels Lab$_*$ to functions in *F*

Classic Problem Instances

	Reaching Definitions	Available Expressions	Very Busy Expressions	Live Variables
L	$\mathcal{P}(\mathrm{Var}_* \times D)$	$\mathcal{P}(\mathrm{AExp}_*)$	$\mathcal{P}(\mathrm{AExp}_*)$	$\mathcal{P}(\mathrm{Var}_*)$
$\sqsubseteq$	$\subseteq$	$\supseteq$	$\supseteq$	$\subseteq$
□	$\bigcup$	$\bigcap$	$\bigcap$	$\bigcup$
ι	$\ldots \cup \{ (x,?) \mid x \in \mathrm{Local}_* \}$	$\emptyset$	$\emptyset$	$\emptyset$
b_ι	b_{init}	b_{init}	b_{last}	b_{last}
next	*preds*	*preds*	*succs*	*succs*
F	$\{ f : L \rightarrow L \mid \exists\, \ell_k, \ell_g : f(\ell) = (\ell \setminus \ell_k) \cup \ell_g \}$			
f_l	$f_l(\ell) = (\ell \setminus kill(l)) \cup gen(l)$ where $l \in \mathrm{Lab}_*$			

Reaching Definition as a Monotone Framework Instance

- Does the lattice *L* satisfy the ascending chain condition?
  - Using Var_* x *D* means we have a finite set of pairs of variables and labels (or ? and •)
  - Thus, the power-set lattice is finite
  - All chains are of finite length and they stabilize at $\top$
- Does the set of functions *F* have the necessary properties?

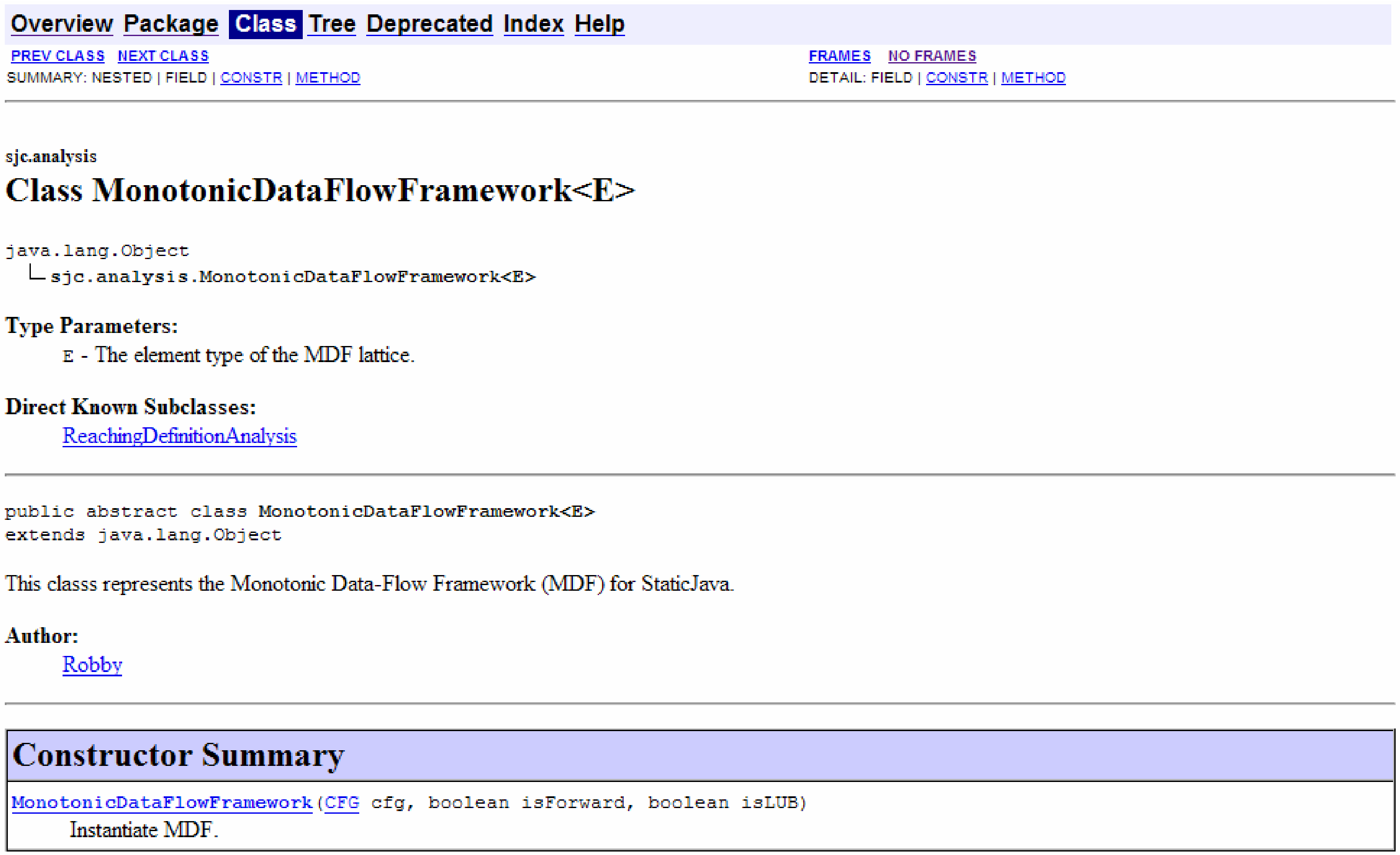
Is *F* a Monotone Function Space?

- Monotonicity
  - let $\ell \sqsubseteq \ell'$
  - it is clear that $(\ell \setminus \ell_k) \sqsubseteq (\ell' \setminus \ell_k)$ since ℓ_k is constant for a statement
  - similarly $((\ell \setminus \ell_k) \cup \ell_g) \sqsubseteq ((\ell' \setminus \ell_k) \cup \ell_g)$ since ℓ_g is constant for a statement
  - thus, $f(\ell) \sqsubseteq f(\ell')$ and f is monotone.
- Identity
  - let $\ell_k = \emptyset$ and $\ell_g = \emptyset$
  - then $f(\ell) = ((\ell \setminus \emptyset) \cup \emptyset) = \ell$
  - so the identity function is in *F*

Is F Closed Under Composition?

- let $f(\ell) = (\ell \setminus \ell_k) \cup \ell_g$ and $f'(\ell) = (\ell \setminus \ell_k') \cup \ell_g'$
- then $(f \circ f')(\ell) = ((\ell \setminus \ell_k') \cup \ell_g' \setminus \ell_k) \cup \ell_g$
- which is $(((\ell \setminus \ell_k') \cup \ell_g') \setminus \ell_k) \cup \ell_g$
- which is $(\ell \setminus (\ell_k' \cup \ell_k)) \cup ((\ell_g' \setminus \ell_k) \cup \ell_g)$
- let $\ell_k''=(\ell_k' \cup \ell_k)$ and $\ell_g''=(\ell_g' \setminus \ell_k) \cup \ell_g$
- then $(f \circ f')(\ell) = (\ell \setminus \ell_k'') \cup \ell_g''$ which is in F

Monotone Dataflow Framework



Overview Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

sjc.analysis

Class MonotonicDataFlowFramework<E>

```
java.lang.Object
  └sjc.analysis.MonotonicDataFlowFramework<E>
```

Type Parameters:

`E` - The element type of the MDF lattice.

Direct Known Subclasses:

ReachingDefinitionAnalysis

```
public abstract class MonotonicDataFlowFramework<E>
extends java.lang.Object
```

This classs represents the Monotonic Data-Flow Framework (MDF) for StaticJava.

Author:

Robby

Constructor Summary
`MonotonicDataFlowFramework(CFG cfg, boolean isForward, boolean isLUB)` Instantiate MDF.

MDF – Constructor

```java
public abstract class MonotonicDataFlowFramework<E> {
  protected @NonNull CFG cfg;
  protected @NonNullElements Map<Statement, Set<E>> outMap =
                                        new HashMap<Statement, Set<E>>();
  protected @NonNullElements Set<E> init;
  protected boolean isForward;
  protected boolean isLUB;

  public MonotonicDataFlowFramework(@NonNull CFG cfg,
                                            boolean isForward,
                                            boolean isLUB) {
    assert cfg != null;
    this.cfg = cfg;
    this.isForward = isForward;
    this.isLUB = isLUB;
  }
}
```

MDF – Abstract Methods

```
public abstract void computeFixPoint();

public abstract @NonNull String toString(@NonNull E e);

public abstract @NonNull String getAnalysisName();

protected abstract Set<E> gen(Set<E> set, Statement s);

protected abstract Set<E> kill(Set<E> set, Statement s);
```

MDF – Equation

$\text{Analysis}_{out}(l) = f_l(\text{Analysis}_{in}(l)),\ f_l(\ell) = (\ell \setminus kill(l)) \cup gen(l)$

```
protected boolean compute(Statement s) {
  Set<E> inSet = getInSet(s);
  inSet.removeAll(kill(inSet, s));
  inSet.addAll(gen(inSet, s));
  Set<E> outSet = getOutSet(s);
  if (outSet.size() != inSet.size() || !outSet.containsAll(inSet)
      || !inSet.containsAll(outSet)) {
    outSet.clear();
    outSet.addAll(inSet);
    inSet.clear();
    return true;
  }
  inSet.clear();
  return false;
}
```

MDF – Equation

$$\text{Analysis}_{in}(l) = \begin{cases} \iota, \text{ if } l = b_\iota \\ \square \, \{ \text{Analysis}_{out}(l') \mid l' \in next(l) \}, \text{ otherwise} \end{cases}$$

```java
public @NonNullElements Set<E> getInSet(@NonNull Statement s) {
  assert s != null; Set<E> inSet;
  if (isForward ? s == cfg.start : s == cfg.end) {
    inSet = new HashSet<E>(init);
  } else {
    inSet = new HashSet<E>(); boolean first = true;
    for (Statement predS : (isForward ? cfg.preds.get(s) :
                                        cfg.succs.get(s))) {
      if (first) {
        inSet.addAll(getOutSet(predS)); first = false;
      } else {
        if (isLUB) { inSet.addAll(getOutSet(predS)); }
        else { inSet.retainAll(getOutSet(predS)); }
      } } } return inSet; }
```

Chaotic Iteration

- The weakest specification of an iterative algorithm for calculating fix point solutions for flow analysis problems

```
$RD_1 := \emptyset$
…
$RD_{14} := \emptyset$
while $\exists\, j.\ RD_j \neq F_j(RD_1, \ldots, RD_{14})$ do
   $RD_j := F_j(RD_1, \ldots, RD_{14})$
```

- The algorithm continues to loop until all components have reached a local fixed point.
- So, if the algorithm terminates a fixed point of *F* is reached.

MDF — DFS-based Iteration

- Based on termination of chaotic iteration, it is guaranteed that any iterative implementation will terminate
- We are going to use a depth-first search (DFS) on CFG per iteration
- Thus, we do DFSs until we reach fix point

MDF — DFS-based Iteration

```
protected boolean iterate(Set<Statement> seen, Statement s) {
  if (seen.contains(s)) {
    return false;
  }
  boolean hasChanged = compute(s);
  seen.add(s);
  Set<Statement> succs = isForward ? cfg.succs.get(s)
                                   : cfg.preds.get(s);
  if (succs != null) {
    for (Statement succS : succs) {
      hasChanged = iterate(seen, succS) || hasChanged;
    }
  }
  return hasChanged;
}
```

MDF — DFS-based Iteration

```
protected void computeFixPoint(@NonNullElements Set<E> init) {
  this.init = init;
  Set<Statement> seen = new HashSet<Statement>();
  while (iterate(seen, isForward ? cfg.start
                                 : cfg.end)) {
    seen.clear();
  }
}
```

RD – Constructor

```
public class ReachingDefinitionAnalysis
    extends MonotonicDataFlowFramework<Pair<String, ASTNode>> {
  protected Map<ASTNode, Object> symbolMap;

  public ReachingDefinitionAnalysis(
        @NonNull SymbolTable symbolTable,
        @NonNull CFG cfg) {
    super(cfg, true, true);
    assert symbolTable != null;
    symbolMap = symbolTable.symbolMap;
  }
}
```

RD — Gen/Kill

```
@Override protected Set<Pair<String, ASTNode>> gen(
                             Set<Pair<String, ASTNode>> set, Statement s) {
  Set<Pair<String, ASTNode>> result = new HashSet<Pair<String, ASTNode>>();
  String lhsLocalName = getLHSVarName(s);
  if (lhsLocalName != null) {
    result.add(new Pair<String, ASTNode>(lhsLocalName, s));
  }
  return result;
}

@Override protected Set<Pair<String, ASTNode>> kill(
                             Set<Pair<String, ASTNode>> set, Statement s) {
  Set<Pair<String, ASTNode>> result = new HashSet<Pair<String, ASTNode>>();
  String lhsLocalName = getLHSVarName(s);
  if (lhsLocalName != null) {
    for (Pair<String, ASTNode> p : set) {
      if (p.first.equals(lhsLocalName)) {
        result.add(p);
      }
    } } return result; }
```

RD – Computing Fix Point

```
@Override public void computeFixPoint() {
  Set<Pair<String, ASTNode>> init = new HashSet<Pair<String, ASTNode>>();
  for (ASTNode n : symbolMap.keySet()) {
    if (n instanceof SimpleName) {
      Object o = symbolMap.get(n);
      if (o instanceof FieldDeclaration) {
        init.add(new Pair<String, ASTNode>(((SimpleName) n).getIdentifier(),
                                           (FieldDeclaration) o)); } } }
  for (Object o : cfg.md.parameters()) {
    SingleVariableDeclaration svd = (SingleVariableDeclaration) o;
    init.add(new Pair<String, ASTNode>(svd.getName().getIdentifier(), svd)); }
  for (Object o : cfg.md.getBody().statements()) {
    if (o instanceof VariableDeclarationStatement) {
      VariableDeclarationStatement vdf = (VariableDeclarationStatement) o;
      init.add(new Pair<String, ASTNode>(((VariableDeclarationFragment)
                                         vdf.fragments().get(0))
                                         .getName().getIdentifier(), vdf));
    } else { break; } }
  computeFixPoint(init);
}
```