A Dataflow Framework for Java

https://checkerframework.org/

October 1, 2026

1 Introduction

This document describes a *Dataflow Framework* for the Java programming language. This is an industrial-strength framework. The Dataflow Framework is used in the Checker Framework, Google's Error Prone, Uber's NullAway, Meta's Nullsafe, and in other contexts.

The primary purpose of the Dataflow Framework is to estimate values: for each line of source code, it determines properties for each variable, that are true for every value the variable might contain.

The Dataflow Framework's result (Section 2.2.10) is an abstract value for each expression (an estimate of the expression's run-time value) and a store at each program point. A store maps variables and other expressions to abstract values.

As a pre-pass, the Dataflow Framework transforms an input AST into a control flow graph (Section 3) consisting of basic blocks made up of nodes representing single operations. An analysis operates over the control flow graph. The effect of a single node on the dataflow store is represented by a transfer function, which takes an input store and a node and produces an output store. Once the analysis reaches a fixed point, the result can be accessed by client code.

The *easiest way* to create a dataflow analysis for Java is to use the Checker Framework. The Checker Framework performs type checking along with flow-sensitive type inference, which is equivalent to a dataflow analysis or an abstract interpretation. In the Checker Framework, the abstract values to be computed are annotated types. You would define a type hierarchy corresponding to the lattice for your dataflow analysis. An individual checker can customize its analysis by extending the abstract value class and by overriding the behavior of the transfer function for particular node types. This approach requires you to write very little code. The downside is that it will run relatively slowly because of the Checker Framework's rich functionality. If you do not need that functionality, you can write a faster analysis directly on the Dataflow Framework. Some users first create an analysis using the Checker Framework, and then if it is not fast enough, they rewrite it for the Dataflow Framework.

The Dataflow Framework was designed with several goals in mind. First, to encourage use beyond the Checker Framework, it is written as a separate package that can be built and used with no dependence on the Checker Framework. Second, the framework supports analysis but not transformation, so it provides information that can be used by a type checker or an IDE, but it does not support optimization. Third, the framework aims to minimize the burden on developers who build on top of it. In particular, the hierarchy of analysis classes is designed to reduce the effort required to implement a new flow-sensitive type checker in the Checker Framework. The Dataflow User's Guide gives an introduction to customizing dataflow to add checker-specific enhancements.

1.1 Potential version conflicts if you export the Dataflow Framework

If your tool uses the Dataflow Framework, then **please shade the Dataflow Framework in your tool**. You can do this using the Maven Shade Plugin, https://maven.apache.org/plugins/maven-shade-plugin/, or the Gradle Shadow Plugin, https://imperceptiblethoughts.com/shadow/. If you cannot shade the Checker Framework and Dataflow Framework classes, then please ask the Checker Framework developers to create a shaded artifact just for you, every time they release the Checker Framework. See `dataflow/build.gradle` for instructions.

The reason to shade the Dataflow Framework is to permit users to run multiple tools. Suppose that two tools both utilize the Dataflow Framework. If the two tools use different versions of the Dataflow Framework, then it may be impossible for users to run both tools due to a version conflict. The Checker Framework uses an unshaded version the Dataflow Framework, since the Dataflow Framework is part of the Checker Framework. If you do not shade the Dataflow Framework in your tool, then users will not be able to use both your tool and the Checker Framework (or any other tool that does not shade the Checker Framework and Dataflow Framework classes).

2 Organization

2.1 Projects

The source code of the combined Checker Framework and Dataflow Framework is divided into multiple projects: `javacutil`, `dataflow`, `framework`, and `checker`, which can be built into distinct jar files. `checker.jar` is a fat jar that contains all of these, plus the Stub Parser.

`javacutil` provides convenient interfaces to routines in Oracle's javac library. There are utility classes for interacting with annotations, elements, trees and types, as well as `InternalUtils`, which gives direct access to internal features of javac that are not part of a supported interface. There are interfaces or abstract classes for reporting errors, for processing types in an AST, and for providing the annotations present on an Element. The `org.checkerframework.javacutil.trees` package provides a class to parse expressions into javac Trees (`TreeParser`), a class to build new Trees from scratch (`TreeBuilder`), and a class to represent newly introduced variables that are not part of an input program (`DetachedVarSymbol`).

`dataflow` contains the classes to represent and construct control flow graphs and the base classes required for flow analysis. These classes are described in detail in Section 2.2.1.

`framework` contains the framework aspects of the Checker Framework, including the derived classes for flow analysis of annotated types which are described later in this document.

`checker` contains the type system-specific checkers.

The `dataflow` project depends only on `javacutil`.

2.2 Classes

This section gives an overview of the major Java classes and interfaces in the implementation of the Dataflow Framework and the flow-sensitive type checking feature of the Checker Framework. It includes both the base classes in the `dataflow` project and the derived classes in the `framework` project. The class and interface declarations are given with full package names to indicate which project they belong to.

2.2.1 Nodes

Dataflow doesn't actually work on trees; it works on Nodes. A Node class represents an individual operation of a program, including arithmetic operations, logical operations, method calls, variable references, array accesses, etc. Nodes simplify writing a dataflow analysis by separating the dataflow analysis from the original source code. Table 1 on page 13 lists the Node types.

```
package org.checkerframework.dataflow.cfg.node;

abstract class Node
class *Node extends Node
```

2.2.2 Blocks

The Block classes represent basic blocks.

```
package org.checkerframework.dataflow.cfg.block;

interface Block
abstract class BlockImpl implements Block
interface SingleSuccessorBlock extends Block
abstract class SingleSuccessorBlockImpl extends BlockImpl implements SingleSuccessorBlock
```

A RegularBlock contains no exception-raising operations and has a single control-flow successor.

```
package org.checkerframework.dataflow.cfg.block;
interface RegularBlock extends SingleSuccessorBlock
class RegularBlockImpl extends SingleSuccessorBlockImpl implements RegularBlock
```

An ExceptionBlock contains a single operation that may raise an exception, with one or more exceptional successors and a single normal control-flow successor.

```
package org.checkerframework.dataflow.cfg.block;
interface ExceptionBlock extends SingleSuccessorBlock
class ExceptionBlockImpl extends SingleSuccessorBlockImpl implements ExceptionBlock
```

A SpecialBlock represents method entry or exit, including exceptional exit which is represented separately from normal exit.

```
package org.checkerframework.dataflow.cfg.block;
interface SpecialBlock extends SingleSuccessorBlock
class SpecialBlockImpl extends SingleSuccessorBlockImpl implements SpecialBlock
```

A ConditionalBlock contains no operations at all. It represents a control-flow split to either a "then" or an "else" successor based on the immediately preceding boolean-valued Node.

```
package org.checkerframework.dataflow.cfg.block;
interface ConditionalBlock extends Block
class ConditionalBlockImpl extends BlockImpl implements ConditionalBlock
```

2.2.3 ControlFlowGraph

A ControlFlowGraph represents the body of a method or an initializer expression as a graph of Blocks with distinguished entry, exit, and exceptional exit SpecialBlocks. ControlFlowGraphs are produced by the CFGBuilder classes and are treated as immutable once they are built.

```
package org.checkerframework.dataflow.cfg;
class ControlFlowGraph
```

2.2.3.1 CFGBuilder The CFGBuilder classes visit an AST and produce a corresponding ControlFlowGraph as described in Section 3.3.

```
package org.checkerframework.dataflow.cfg;
class CFGBuilder
```

The Checker Framework derives from CFGBuilder in order to desugar enhanced for loops that make explicit use of type annotations provided by the checker in use.

```
package org.checkerframework.framework.flow;
class CFCFGBuilder extends CFGBuilder
```

2.2.3.2 CFGVisualizeLauncher The CFGVisualizeLauncher generates a DOT or String representation of the control flow graph for a given method in a given class.

```
package org.checkerframework.dataflow.cfg;
class CFGVisualizeLauncher
```

2.2.4 JavaExpressions

The Dataflow Framework records the abstract values of certain expressions, called JavaExpressions: local variables, field accesses, array accesses, references to `this`, and pure method calls. JavaExpressions are keys in the store of abstract values.

```
package org.checkerframework.dataflow.analysis;
class JavaExpressions
```

Java expressions that appear in method pre- and postconditions are parsed into JavaExpressions using helper routines in `org.checkerframework.dataflow.expression.JavaExpressionParseUtil`.

2.2.5 AbstractValue

AbstractValue is the internal representation of dataflow information produced by an analysis. An AbstractValue is an estimate about the run-time values that an expression may evaluate to. The client of the Dataflow Framework defines the abstract value, so the information may vary widely among different users of the Dataflow Framework, but they share a common feature that one can compute the least upper bound of two AbstractValues.

```
package org.checkerframework.dataflow.analysis;
interface AbstractValue<V extends AbstractValue<V>>
```

For the Checker Framework, abstract values are essentially AnnotatedTypeMirrors.

```
package org.checkerframework.framework.flow;
abstract class CFAbstractValue<V extends CFAbstractValue<V>> implements AbstractValue<V>
class CFValue extends CFAbstractValue<CFValue>
```

For the Nullness Checker, abstract values additionally track the meaning of PolyNull, which may be either Nullable or NonNull. The meaning of PolyNull can change when a PolyNull value is compared to the null literal, which is specific to the Nullness Checker. Other checkers often also support a Poly* qualifier, but only the Nullness Checker tracks the meaning of its poly qualifier using the dataflow analysis.

```
package org.checkerframework.checker.nullness;
class NullnessValue extends CFAbstractValue<NullnessValue>
```

2.2.6 UnusedAbstractValue

UnusedAbstractValue is an AbstractValue that is not used during dataflow analysis. This class should only be used as a type argument in transfer functions, for analyses that do not need and do not have their own AbstractValue classes.

For example, LiveVariable analysis (Section 6), needs to compute the least upper bound of successors' stores, and it is meaningless to further calculate the least upper bound of two individual live variables at a smaller granularity. Since there is no computation between two AbstractValues, LiveVariable analysis uses the UnusedAbstractValue rather than implementing a specific AbstractValue for this analysis.

```
package org.checkerframework.dataflow.analysis;
public final class UnusedAbstractValue implements AbstractValue<UnusedAbstractValue>
```

2.2.7 Store

A Store is a set of dataflow facts computed by an analysis, so it is a mapping from JavaExpressions to AbstractValues. As with AbstractValues, one can take the least upper bound of two Stores.

```
package org.checkerframework.dataflow.analysis;
interface Store<S extends Store<S>>
```

The Checker Framework store restricts the type of abstract values it may contain.

```
package org.checkerframework.framework.flow;
abstract class CFAbstractStore<V extends CFAbstractValue<V>,
        S extends CFAbstractStore<V, S>>
    implements Store<S>
class CFStore extends CFAbstractStore<CFValue, CFStore>
```

An InitializationStore tracks which fields of the "self" reference have been initialized.

```
package org.checkerframework.checker.initialization;
class InitializationStore<V extends CFAbstractValue<V>,
        S extends InitializationStore<V, S>>
    extends CFAbstractStore<V, S>
```

A NullnessStore additionally tracks the meaning of PolyNull.

```
package org.checkerframework.checker.nullness;
class NullnessStore extends InitializationStore<NullnessValue, NullnessStore>
```

2.2.8 Transfer functions

A transfer function (Section 2.2.8.3) is explicitly represented as a node visitor that takes a TransferInput (Section 2.2.8.1) and produces a TransferResult (Section 2.2.8.2).

2.2.8.1 TransferInput The TransferInput represents the set of dataflow facts known to be true immediately before the node to be analyzed. A TransferInput may contain a single store, or a pair of "then" and "else" stores when following a boolean-valued expression.

```
package org.checkerframework.dataflow.analysis;
class TransferInput<V extends AbstractValue<V>,
        S extends Store<S>>
```

2.2.8.2 TransferResult A TransferResult is the output of a transfer function. In other words, it is the set of dataflow facts known to be true immediately after a node. A Boolean-valued expression produces a ConditionalTransferResult that contains both a "then" and an "else" store, while most other Nodes produce a RegularTransferResult with a single store.

```
package class org.checkerframework.dataflow.analysis;
abstract TransferResult<V extends AbstractValue<V>,
        S extends Store<S>>
class ConditionalTransferResult<V extends AbstractValue<V>,
        S extends Store<S>>
    extends TransferResult<A, S>
class RegularTransferResult<V extends AbstractValue<V>,
        S extends Store<S>>
    extends TransferResult<A, S>
```

2.2.8.3 TransferFunction A TransferFunction is a NodeVisitor that takes an input and produces an output.

```
package org.checkerframework.dataflow.analysis;
interface TransferFunction<V extends AbstractValue<V>,
        S extends Store<S>>
    extends NodeVisitor<TransferResult<V, S>, TransferInput<V, S>>
interface ForwardTransferFunction<V extends AbstractValue<V>,
        S extends Store<S>>
    extends TransferFunction<V, S>
interface BackwardTransferFunction<V extends AbstractValue<V>,
        S extends Store<S>>
    extends TransferFunction<V, S>
```

The Checker Framework defines a derived class of TransferFunction to serve as the default for most checkers. The class constrains the type of abstract values and it overrides many node visitor methods to refine the abstract values in their TransferResults.

```
package org.checkerframework.framework.flow;
abstract class CFAbstractTransfer<V extends CFAbstractValue<V>,
        S extends CFAbstractStore<V, S>,
        T extends CFAbstractTransfer<V, S, T>>
    extends AbstractNodeVisitor<TransferResult<V, S>, TransferInput<V, S>>
    implements ForwardTransferFunction<V, S>

class CFTransfer extends CFAbstractTransfer<CFValue, CFStore, CFTransfer>
```

The Initialization Checker's transfer function tracks which fields of the "self" reference have been initialized.

```
package org.checkerframework.checker.initialization;
class InitializationTransfer<V extends CFAbstractValue<V>,
        T extends InitializationTransfer<V, T, S>,
        S extends InitializationStore<V, S>>
    extends CFAbstractTransfer<V, S, T>
```

The Regex Checker's transfer function overrides visitMethodInvocation to special-case the `isRegex` and `asRegex` methods.

```
package org.checkerframework.checker.regex;
class RegexTransfer extends CFAbstractTransfer<CFValue, CFStore, RegexTransfer>
```

2.2.9 Analysis

An Analysis performs iterative dataflow analysis over a control flow graph using a given transfer function. Both forward and backward analyses are supported.

```
package org.checkerframework.dataflow.analysis;
interface Analysis<V extends AbstractValue<V>,
        S extends Store<S>,
        T extends TransferFunction<V, S>>
abstract class AbstractAnalysis<V extends AbstractValue<V>,
        S extends Store<S>,
        T extends TransferFunction<V, S>>
    implements Analysis<V, S, T>
interface ForwardAnalysis<V extends AbstractValue<V>,
        S extends Store<S>,
        T extends ForwardTransferFunction<V, S>>
    extends Analysis<V, S, T>
interface BackwardAnalysis<V extends AbstractValue<V>,
        S extends Store<S>,
        T extends BackwardTransferFunction<V, S>>
    extends Analysis<V, S, T>
```

The Checker Framework defines a derived class of Analysis for use as the default analysis of most checkers. This class adds information about the type hierarchy being analyzed and acts as a factory for abstract values, stores, and the transfer function.

```
package org.checkerframework.framework.flow;
```

```
abstract class CFAbstractAnalysis<V extends CFAbstractValue<V>,
        S extends CFAbstractStore<V, S>,
        T extends CFAbstractTransfer<V, S, T>>
    extends ForwardAnalysisImpl<V, S, T>

class CFAnalysis extends CFAbstractAnalysis<CFValue, CFStore, CFTransfer>
```

The Nullness Checkers' analysis overrides the factory methods for abstract values, stores, and the transfer function.

```
package org.checkerframework.checker.nullness;
class NullnessAnalysis extends CFAbstractAnalysis<NullnessValue,
        NullnessStore, NullnessTransfer>
```

The RegexChecker's analysis overrides the factory methods for abstract values, stores, and the transfer function.

```
package org.checkerframework.checker.regex;
class RegexAnalysis extends CFAbstractAnalysis<CFValue, CFStore, RegexTransfer>
```

2.2.10 AnalysisResult

An AnalysisResult preserves the dataflow information computed by an Analysis for later use by clients. The information consists of an AbstractValue for each node in the CFG and a Store that is valid at the start of each Block. The AnalysisResult class can return AbstractValues for either Nodes or Trees and it can re-run the transfer function to compute Stores that are valid immediately before or after any Tree.

```
package org.checkerframework.dataflow.analysis;
class AnalysisResult<V extends AbstractValue<V>,
        S extends Store<S>>
```

2.2.11 AnnotatedTypeFactory

AnnotatedTypeFactorys are not part of the Dataflow Framework, per se, but they are parameterized by the Dataflow Framework classes that they use.

```
package org.checkerframework.framework.type;
class AnnotatedTypeFactory implements AnnotationProvider
```

In the Checker Framework, dataflow analysis is performed on demand, one class at a time, the first time that a ClassTree is passed to getAnnotatedType. This is implemented in the abstract class GenericAnnotatedTypeFactory with concrete implementation in BaseAnnotatedTypeFactory.

```
package org.checkerframework.framework.type;
abstract class GenericAnnotatedTypeFactory<Checker extends BaseTypeChecker<?>,
        Value extends CFAbstractValue<Value>,
        Store extends CFAbstractStore<Value, Store>,
        TransferFunction extends CFAbstractTransfer<Value, Store, TransferFunction>,
        FlowAnalysis extends CFAbstractAnalysis<Value, Store, TransferFunction>>
```

```
1 class Test {
2   void test(boolean b) {
3     int x = 2;
4     if (b) {
5       x = 1;
6     }
7   }
8 }
```

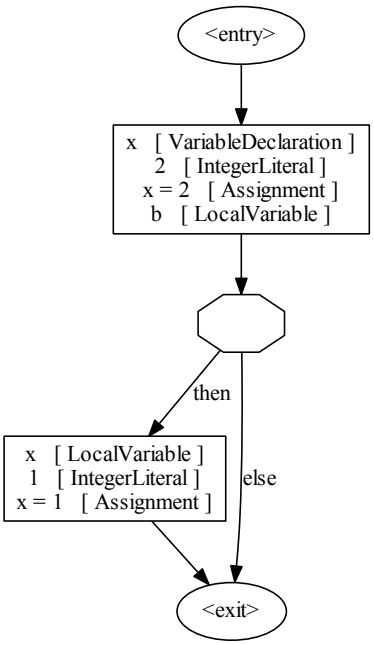


Figure 1: A simple Java code snippet to introduce the CFG. In CFG visualizations, special basic blocks are shown as ovals; conditional basic blocks are polygons with eight sides; and regular and exception basic blocks are rectangles.

```
    extends AnnotatedTypeFactory

package org.checkerframework.common.basetype;
class BaseAnnotatedTypeFactory
    extends GenericAnnotatedTypeFactory<CFValue, CFStore, CFTransfer, CFAnalysis>
```

3 The Control-Flow Graph

A control-flow graph (CFG) represents a single method or field initialization. (The Dataflow Framework performs an intra-procedural analysis. This analysis is modular and every method is considered in isolation.) This section also describes the translation from the abstract syntax tree (AST) to the CFG. We start with a simple example, then give a more formal definition of the CFG and its properties, and finally describe the translation from the AST to the CFG.

As is standard, a control-flow graph is a set of basic blocks that are linked by control-flow edges. Possibly less standard, every basic block consists of a sequence of so-called nodes, each of which represents a minimal Java operation or expression.

Consider the method `test` of Figure 1. The if conditional got translated to a *conditional basic block* (octagon) with two successors. There are also two special basic blocks (ovals) to denote the entry and exit point of the method.

3.1 Formal Definition of the Control-Flow Graph

The control-flow graph models all paths that can possibly be taken by an execution of the method.

Definition 3.1 (Control-Flow Graph)**.** *A* control-flow graph *consists of a set of* basic blocks *and a set of directed edges between these basic blocks, some of which are labeled.*

Definition 3.2 (Basic Block)**.** *A* basic block *is a sequence of* nodes*, where the only control flow between the nodes inside the basic block is sequential. Furthermore, there is no control flow occurring between those nodes and nodes of other basic blocks, except between the last node of one block b_1 and the first node of another block b_2, if b_2 is a successor of b_1. A basic block may have multiple successors.*

Definition 3.3 (Types of Basic Blocks)**.** *There are four* types *of basic blocks in a control-flow graph:*

1. ***Regular basic block.*** *A* regular basic block *contains any non-empty sequence of nodes and has exactly one successor. None of the nodes in the block can throw an exception at run time.*
2. ***Special basic blocks.*** *A* special basic block *contains the empty sequence of nodes (i.e., is empty) and denotes either the entry or one of the exit blocks of a method. There are three types of special basic blocks:*
   - *Entry block. This basic block is the (only) entry point of the method and thus is the only basic block without predecessors.*
   - *Exit block. This basic block denotes the (normal) exit of a method, and it has no successors.*
   - *Exceptional exit block, which indicates exceptional termination of the method. As an exit block, this block has no successors.*

   Every method has exactly one entry block, zero or one exit blocks, and zero or one exceptional exit blocks. There is always either an exit block, an exceptional exit block, or both.
3. ***Exception basic block.*** *An* exception basic block *contains exactly one node that* might *throw an exception at run time (e.g., a method call). There are zero or one non-exceptional successors (only a basic block containing a* `throw` *statement does not have a non-exceptional successor). There are one or more exceptional successors (see 3.4). In all cases there is at least one successor (regular or exceptional).*
4. ***Conditional basic block.*** *A* conditional basic block *does not contain any nodes and is used as a* split point *after the execution of a node of boolean type. It has exactly two successors (both non-exceptional): the* then *successor that is reached when the previous node evaluates to true and the* else *successor that is reached when the previous node evaluates to false. There is always exactly a single predecessor block for every conditional basic block, which is either a regular basic block or an exception basic block. In both cases, the last node in the predecessor will be of boolean type and the boolean value controls which successor of the conditional block is executed.*

The Java implementation of the four block types above is described in Section 2.2.2.

Definition 3.4 (Control-Flow Graph Edges)**.** *The basic blocks of a control-flow graph are connected by directed* edges*. If b_1 and b_2 are connected by a directed edge (b_1, b_2), we call b_1 a predecessor of b_2, and we call b_2 a successor of b_1. In a control-flow graph, there are three types of edges:*

1. ***Exceptional edges****. An* exceptional edge *connects an exception basic block with its exceptional successors, and it is labeled by the most general exception that might cause execution to take this edge during run time. Note that the outgoing exceptional edges of a basic block do not need to have mutually exclusive labels; the semantics is that the control flow follows the most specific edge. For instance, if one edge is labeled with type* `A` *and another is labeled with type* `B` *where* `B` *is a subtype of* `A`*, then the execution only takes the first edge if the exception is of a subtype of* `A`*, but not a subtype of* `B`*.*

> *There is not necessarily a most specific exception type in the program text; in that case, does the translation add a most specific case that will never be executed at run time?*
>
> *In general, what is the relation of the ordering in source code to the one here?*

There is at most one successor for every exception type.

2. ***Conditional edges**. A* conditional edge *is a non-exceptional edge that connects a conditional basic block with one of its successors, and is labeled with either "true" or "false".*

3. ***Regular, non-conditional edge.** Any other edge is a* regular edge*, and does not carry a label. Only regular basic blocks, the entry basic block, and exception basic blocks have outgoing regular edges.*

Definition 3.5 (Nodes)**.** *A* node *is a minimal Java operation or expression. It is minimal in the sense that it cannot be decomposed further into subparts between which control flow occurs. Examples for such nodes include integer literals, an addition node (which performs the mathematical addition of two nodes) or a method call. Control flow such as* `if` *and* `break` *are not represented as nodes. The full list of nodes is given in Table 1 and several of them are described in more detail in Section 3.2.*

It is important to note that, even though nodes can contain references to other nodes, it is only the "top-level" node which is considered at that point in the basic block. In the example of the addition node, this means that only the addition operation is to be executed, and its operands would occur earlier in the control-flow graph (as they are evaluated first, before performing the addition).

In the visualization, a string representation of the node is used, followed by the node type in square brackets. Note that the string representation often also includes more than just the "top-level" node. For instance, the addition expression `a + b[0];` will appear as "a + b[0] [NumericalAddition]" rather than "a" plus some temporary variable. This is done for clarity, so that it is easy to see what expressions are summed up and because we don't create internal names for expression results.

Table 1 lists all node types in the framework. All classes are in package `org.checkerframework.dataflow.cfg.node`.

Table 1: All node types in the Dataflow Framework.

Node type	Notes	Example
`Node`	The base class of all nodes	
`ValueLiteralNode`	The base class of literal value nodes	
`BooleanLiteralNode`		`true`
`CharacterLiteralNode`		`'c'`
`DoubleLiteralNode`		`3.14159`
`FloatLiteralNode`		`1.414f`
`IntegerLiteralNode`		`42`
`LongLiteralNode`		`1024L`
`NullLiteralNode`		`null`
`ShortLiteralNode`		`512`
`StringLiteralNode`		`"memo"`
	Accessor expressions	
`ArrayAccessNode`		`args[i]`
`FieldAccessNode`		`f`, `obj.f`
`MethodAccessNode`		`obj.hashCode`
`LocalVariableNode`	Use of a local variable, either as l-value or r-value	

Continued on next page

Node type	Notes	Example
`ThisNode`	Base class of references to `this`	
`ExplicitThisNode`	Explicit use of `this` in an expression	
`ImplicitThisNode`	Implicit use of `this` in an expression	
`SuperNode`	Explicit use of `super` in expression	`super(x, y)`
`MethodInvocationNode`	Note that access and invocation are distinct	`hashCode()`
	Expression supertypes	
`BinaryOperationNode`	Base class of binary expression nodes	
`UnaryOperationNode`	Base class of unary expression nodes	
	Arithmetic and logical operations	
`BitwiseAndNode`		a & `b`
`BitwiseComplementNode`		`~b`
`BitwiseOrNode`		`a | b`
`BitwiseXorNode`		`a ^ b`
`ConditionalAndNode`	Short-circuiting	a && `b`
`ConditionalNotNode`		`!a`
`ConditionalOrNode`	Short-circuiting	`a || b`
`FloatingDivisionNode`		`1.0 / 2.0`
`FloatingRemainderNode`		`13.0 % 4.0`
`LeftShiftNode`		`x << 3`
`IntegerDivisionNode`		`3 / 2`
`IntegerRemainderNode`		`13 % 4`
`NumericalAdditionNode`		`x + y`
`NumericalMinusNode`		`-x`
`NumericalMultiplicationNode`		`x * y`
`NumericalPlusNode`		`+x`
`NumericalSubtractionNode`		`x - y`
`SignedRightShiftNode`		`x >> 3`
`StringConcatenateNode`		`s + ".txt"`
`SwitchExpressionNode`		
`TernaryExpressionNode`		`c ? t : f`
`UnsignedRightShiftNode`		`x >>> 5`
	Relational operations	
`EqualToNode`		`x == y`
`NotEqualNode`		`x != y`
`GreaterThanNode`		`x > y`
`GreaterThanOrEqualNode`		`x >= y`
`LessThanNode`		`x < y`
`LessThanOrEqualNode`		`x <= y`
`SwitchExpression`	A switch expression (not statement!)	
`CaseNode`	Case of a switch. Acts as an equality test.	
`AssignmentNode`		`x = 1`
`ArrayCreationNode`		`new double[]`
`ObjectCreationNode`		`new Object()`
`FunctionalInterfaceNode`	A member reference or lambda	
`TypeCastNode`		`(float)42`

Continued on next page

Node type	Notes	Example
`InstanceOfNode`		`x instanceof Float`
	Conversion nodes	
`NarrowingConversionNode`	Implicit conversion	
`StringConversionNode`	Might be implicit	`obj.toString()`
`WideningConversionNode`	Implicit conversion	
	Non-value nodes Types appearing in expressions, such as `MyType.class`	
`ArrayTypeNode`		
`ParameterizedTypeNode`		
`PrimitiveTypeNode`		
`ClassNameNode`	Identifier referring to Java class or interface	`java.util.HashMap`
`PackageNameNode`	Identifier referring to Java package	`java.util`
`ThrowNode`	Throw an exception	
`ReturnNode`	Return from a method	
`AssertionErrorNode`		`assert x != null : "Hey"`
`ExpressionStatementNode`	An expression that is used as a statement	`m();`
`SynchronizedNode`	Start or end of a synchronized code block	
`MarkerNode`	No-op node used to annotate a CFG with information of the underlying Java source code. Mostly useful for debugging and visualization. An example is indicating the start/end of switch statements.	
`NullChkNode`	Null checks inserted by javac	
`VariableDeclarationNode`	Declaration of a local variable	
`ClassDeclarationNode`	Declaration of a class	
`LambdaResultExpressionNode`	Body of a single-expression lambda	

Table 1: All node types in the Dataflow Framework.

In theory, nearly any statement can throw an `Error` such as `OutOfMemoryError` or `NoSuchFieldError`. The Dataflow Framework does not represent all that possible flow. It only creates the exceptional edges shown in Table 2.

Other AST constructs are desugared into the above nodes (see Section 3.3.1).

3.2 Noteworthy Translations and Node Types

In this section we mention any non-straightforward translations from the AST to the CFG, or special properties about individual nodes.

3.2.1 Program Structure

Java programs are structured using high-level programming constructs such as loops, if-then-else constructs, try-catch-finally blocks or switch statements. During the translation from the AST to the CFG some of

Node type	Exception type
`ArrayAccessNode`	`NullPointerException`, `ArrayIndexOutOfBoundsException`
`FieldAccessNode`	`NullPointerException`
`MethodAccessNode`	`NullPointerException`
`MethodInvocationNode`	`Throwable`, types in throws clause of the signature
`IntegerDivisionNode`	`ArithmeticException`
`IntegerRemainderNode`	`ArithmeticException`
`ObjectCreationNode`	`Throwable`, types in throws clause of the signature
`ArrayCreationNode`	`NegativeArraySizeException`, `OutOfMemoryError`
`TypeCastNode`	`ClassCastException`
`ThrowNode`	Type of `e` when `throw e`
`AssertionErrorNode`	`AssertionError`
`ClassNameNode`	`ClassCircularityError`, `ClassFormatError`, `NoClassDefFoundError`, `OutOfMemoryError`

Table 2: All node types that could throw Exception, and the types to be thrown. All exception types are in package `java.lang`.

this program structure is lost and all non-sequential control flow is represented by two low-level constructs: conditional basic blocks and control-flow edges between basic blocks. For instance, a `while` loop is translated into its condition followed by a conditional basic block that models the two possible outcomes of the condition: either the control flow follows the "true" branch and continues with the loop's body, or control goes to the "false" successor and executes the first statement after the loop.

3.2.2 Assignment

As described in JLS §15.26.1, the execution of an assignment is in general not strictly left-to-right. Rather, the right-hand side might be evaluated even if the left-hand side of the assignment causes an exception. This semantics is faithfully represented in the CFG produced by the translation. An example of a field assignment exhibiting this behavior is shown in Figure 2.

3.2.3 Postfix/Prefix Increment/Decrement

Postfix and prefix increment and decrement have a side effect to update the variable or field. To represent this side effect, the Dataflow Framework creates an artificial assignment node like `n = n + 1` for `++n` or `n++`. This artificial assignment node is stored in `unaryAssignNodeLookup` of `ControlFlowGraph`. The assignment node is also stored in `treeLookup` for prefix increment or decrement so that the result of it is after the assignment. However, the node before the assignment is stored in `treeLookup` for postfix increment or decrement because the result of it should be before the assignment. For further information about node-tree mapping, see Section 3.3.2.

3.2.4 Conditional stores

The Dataflow Framework extracts information from control-flow splits that occur in `if`, `for`, `while`, and `switch` statements and in conditional operator expressions. In order to have the information available at the split, we eagerly produce two stores contained in a `ConditionalTransferResult` after certain boolean-valued expressions. The stores are called the *then* and *else* stores. So, for example, after the expression `x == null`, two different stores will be created. The Nullness Checker would produce a then store that maps `x` to @Nullable and an else store that maps `x` to @NonNull.

```
1 class Test {
2   int f;
3
4   void test(Test x) {
5     x.f = 1;
6   }
7 }
```

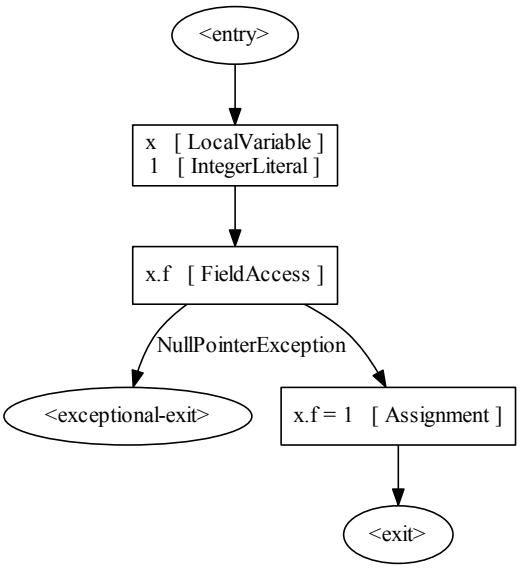


Figure 2: Control flow for a field assignment is not strictly left-to-right (cf. JLS §15.26.1), which is properly handled by the translation.

The Dataflow Framework allows a maximum of two stores and when there are two distinct stores, they always refer to the most recent boolean-valued expression. Stores are propagated through most nodes and they are reversed for conditional not expressions. The transfer functions for many nodes merge conditional stores back together because they cannot maintain the distinction between them. Merging just means taking the least upper bound of the two stores and it happens automatically by calling `TransferInput.getRegularStore`.

Assignments usually merge conditional stores. For the translation of conditional operator expressions, a special non-merging assignment node is used (see Section 3.2.6 below).

3.2.5 Branches

The control flow graph represents all non-exceptional control-flow splits, or branches, as `ConditionalBlock`s that contain no nodes. If there is one store flowing into a conditional block, then it is duplicated to both successors. If there are two stores flowing into a conditional block, the then store is propagated to the block's then successor and the else store is propagated to the block's else successor.

Consider the control flow graph generated for the simple if statement in Figure 3. The conditional expression `b1` immediately precedes the `ConditionalBlock`, represented by the octagonal node. The `ConditionalBlock` is followed by both a then and an else successor block, after which control flow merges back together at the exit block. The edge labels `EACH_TO_EACH`, `THEN_TO_BOTH`, and `ELSE_TO_BOTH` are flow rules described in Section 4.4. As described above, the then store propagates to (both stores of) the block's then successor according to rule `THEN_TO_BOTH` and the else store propagates to (both stores of) the block's else successor according to rule `ELSE_TO_BOTH`. More precise rules are used to preserve dataflow information for short-circuiting expressions, as described in Section 3.2.6.

3.2.6 Conditional Expressions

The conditional and (`&&`, cf. JLS §15.23) and the conditional or (`||`, cf. JLS §15.24) expressions are subject to short-circuiting: if evaluating the left-hand side already determines the result, then the right-hand side is not evaluated. This semantics is faithfully represented in the constructed CFG and more precise flow rules

```
1  class Test {
2    void testIf(boolean b1) {
3      int x = 0;
4      if (b1) {
5        x = 1;
6      } else {
7        x = 2;
8      }
9    }
10 }
```

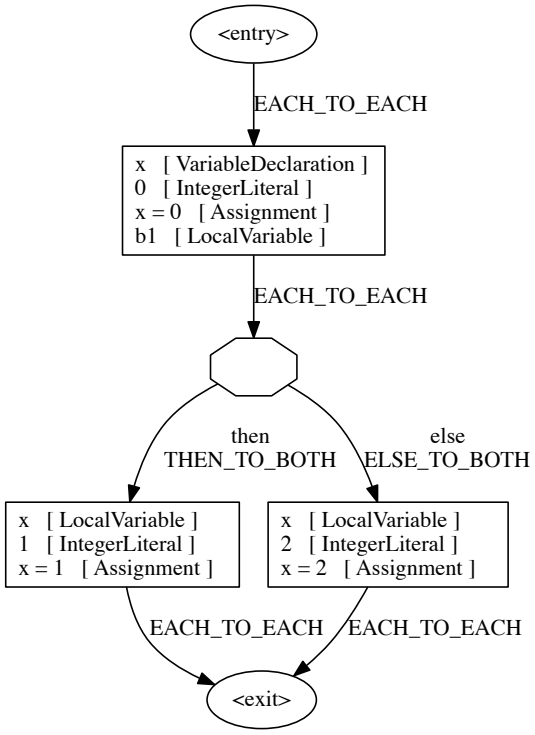


Figure 3: Example of an if statement translated into a `ConditionalBlock`.

(Section 4.4) are used to preserve additional dataflow information.

An example program using conditional or is shown in Figure 4. Note that the CFG correctly represents short-circuiting. The expression `b2 || b3` is only executed if `b1` is false and `b3` is only evaluated if `b1` and `b2` are false.

Observe in Figure 4 that the flow rule between the first conditional block and its then successor is `THEN_TO_THEN`, rather than the default flow rule for such edges `THEN_TO_BOTH`, which is present on the edge from the last conditional block to its then successor. `THEN_TO_THEN` is a more precise rule which propagates the then store from the predecessor of the conditional block to the then store of the then successor and leaves the else store of the successor untouched. This is a valid rule for propagating information along the short-circuit edge of a conditional or expression because `b1 || (b2 || b3)` being false implies that `b1` is false, so dataflow information that obtains when `b1` is true has no effect on the dataflow information obtains when `b1 || (b2 || b3)` is false. To put it another way, if control reaches the block containing `b1 || (b2 || b3)` and that expression is false, then control must have flowed along the else branches of both conditional blocks and only the facts that obtain along those edges need to be kept in the else store of the block containing `b1 || (b2 || b3)`.

Conditional operator expressions (using the ternary conditional operator `? :`, cf. JLS §15.25) use an additional synthetic local variable in the CFG. The second (then) and third (else) expressions are assigned to this local variable. In contrast to other assignments, this assignment does not merge the stores, to allow the propagation of conditional stores from these sub-expressions. The `TernaryExpressionNode` at the merge of the two branches can then either be treated like a local variable read or return a conditional store.

```
1 class Test {
2   void test(boolean b1,
        boolean b2, boolean b3) {
3     int x = 0;
4     if (b1 || (b2 || b3)) {
5       x = 1;
6     }
7   }
8 }
```

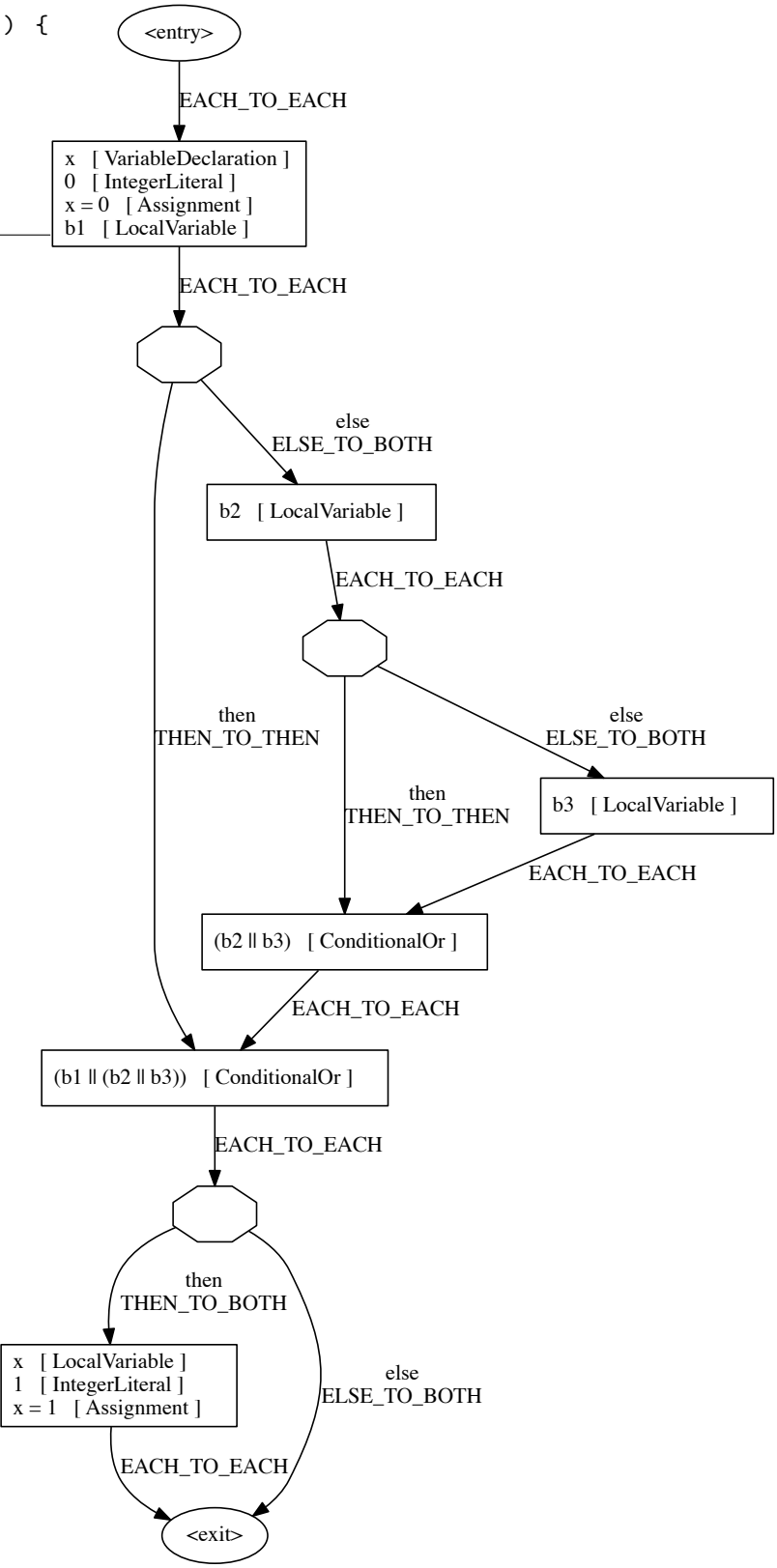


Figure 4: Example of a conditional or expression (||) with short-circuiting and more precise flow rules.

```
class
    Test
    {
  int
      test(boolean
      b)
      {
    int
        x
        =
        b
        ?
        this.hashCode()
        :
        5;
    return
        x;
  }
}
```

3.2.7 Implicit `this` access

The Java compiler AST uses the same type (`IdentifierTree`) for local variables and implicit field accesses (where `this.` is left out). To relieve the user of the Dataflow Framework from manually determining the two cases, we consistently use `FieldAccessNode` for field accesses, where the receiver might be an `ImplicitThisNode`.

3.2.8 Assert statements

Assert statements are treated specially by the CFG builder because it is unknown at CFG construction time whether or not assertions will be enabled when the program is run. When assertions are enabled, the dataflow information gained by analyzing the assert statement can improve precision and allow the programmer to avoid redundant annotations. However, when assertions are disabled, it would be unsound to assume that they had any effect on dataflow information.

The user of the Dataflow Framework may specify that assertions are enabled or disabled. When assertions are assumed to be disabled, no CFG Nodes are built for the assert statement. When assertions are assumed to be enabled, CFG Nodes are built to represent the condition of the assert statement and, in the else successor of a ConditionalBlock, CFG Nodes are built to represent the detail expression of the assert, if any.

If assertions are not assumed to be enabled or disabled, then the CFG is conservative and represents the fact that the assert statement may execute or may not. This takes the form of a ConditionalBlock that branches on a fake variable. For example, see Figure 6. The fake variable named `assertionsEnabled#num0` controls the first ConditionalBlock. The then successor of the ConditionalBlock is the same subgraph of CFG Nodes that would be created if assertions were assumed to be enabled, while the else successor of the ConditionalBlock is the same, empty, subgraph of CFG Nodes that would be created if assertions were assumed to be disabled.

3.2.9 Varargs method invocation

In Java, varargs in method or constructor invocation is compiled as new array creation (cf. JLS §15.12.4.2). For example, `m(1, 2, 3)` will be compiled as `m(new int[]{1, 2, 3})` when the signature of `m` is `m(int... args)`. Dataflow Framework creates an `ArrayCreationNode` with initializer for varargs in the same way as the Java compiler does. Note that it doesn't create an `ArrayCreationNode` when the varargs is an array with the same depth as the type of the formal parameter, or if `null` is given as the actual varargs argument.

3.2.10 Default case and fall through for switch statement

A switch statement is handled as a chain of `CaseNode` and nodes in the case. `CaseNode` makes a branch by comparing the equality of the expression of the switch statement and the expression of the case. Note that the expression of a switch statement must be executed only once at the beginning of the switch statement. To refer to its value, a fake variable is created and it is assigned to a fake variable. A `THEN_TO_BOTH` edge goes to nodes in the case and a `ELSE_TO_BOTH` edge goes to the next `CaseNode`. When the next case is the default case, it goes to nodes in the default case. If a break statement is in nodes, it creates an edge to next node of the switch statement. If there is any possibility of fall-through, an edge to the first node in the next case is created after nodes in the case. For example, see Figure 7. The fake variable named `switch#num0` is created and each case node creates the branches.

3.2.11 Switch expressions

A switch expression is translated similarly to a switch statement. An additional fake variable `switchExpr` is used to store the value from the cases. The `SwitchExpressionNode` itself can then be handled like a read of that variable.

```
1 class Test {
2   void testAssert(Object a) {
3     assert a != null :
          "Argument is null";
4   }
5 }
```

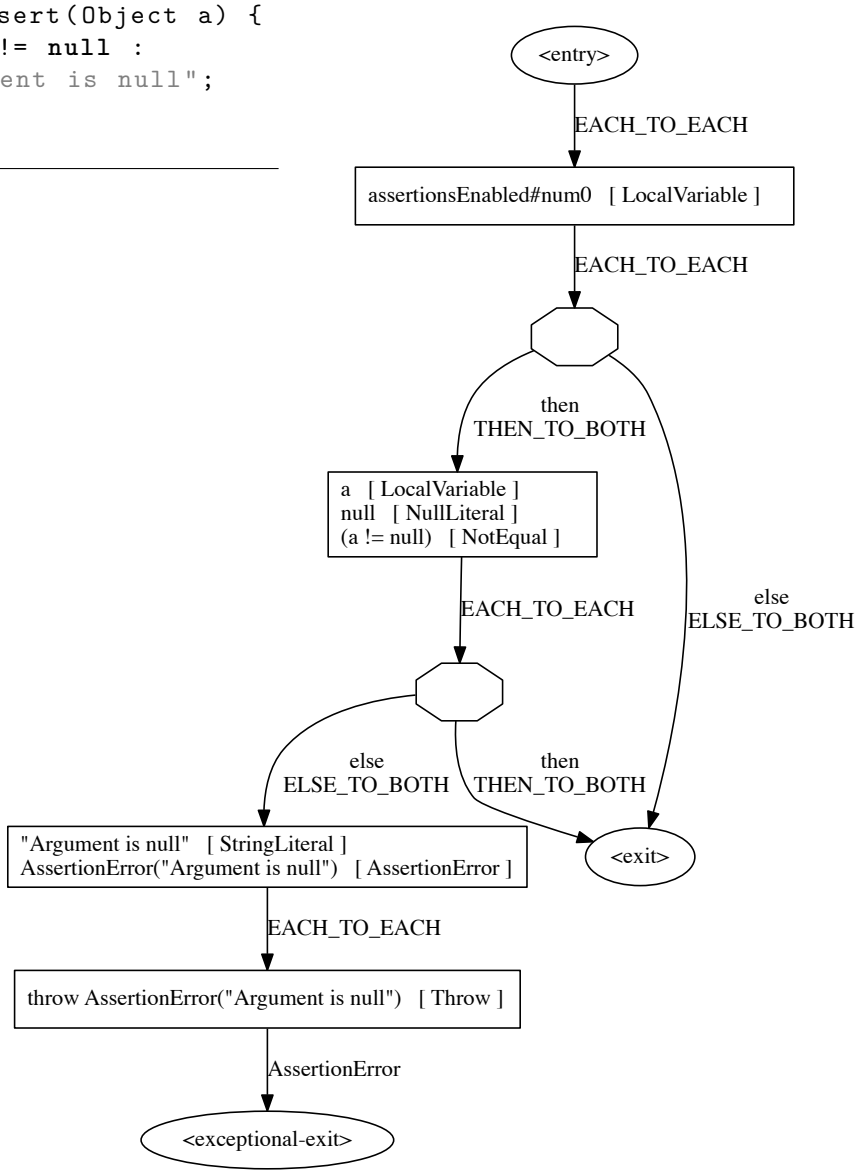


Figure 6: Example of an assert statement translated with assertions neither assumed to be enabled nor assumed to be disabled.

3.2.12 Handling `finally` blocks

Control flow statements, like `return`, `break`, and `continue`, within `try` blocks will cause execution of the `finally` block before continuing at the target of the jump. The Dataflow Framework models this behavior by adding a jump to a duplicate of the finally block before the jump to the original target of the control flow statement.

```
1  class Test {
2    void test(int x) {
3      switch (x) {
4        case 1:
5          int a = x;
6          break;
7        case 2:
8          int b = x;
9        default:
10         int c = x;
11         break;
12     }
13   }
14 }
```

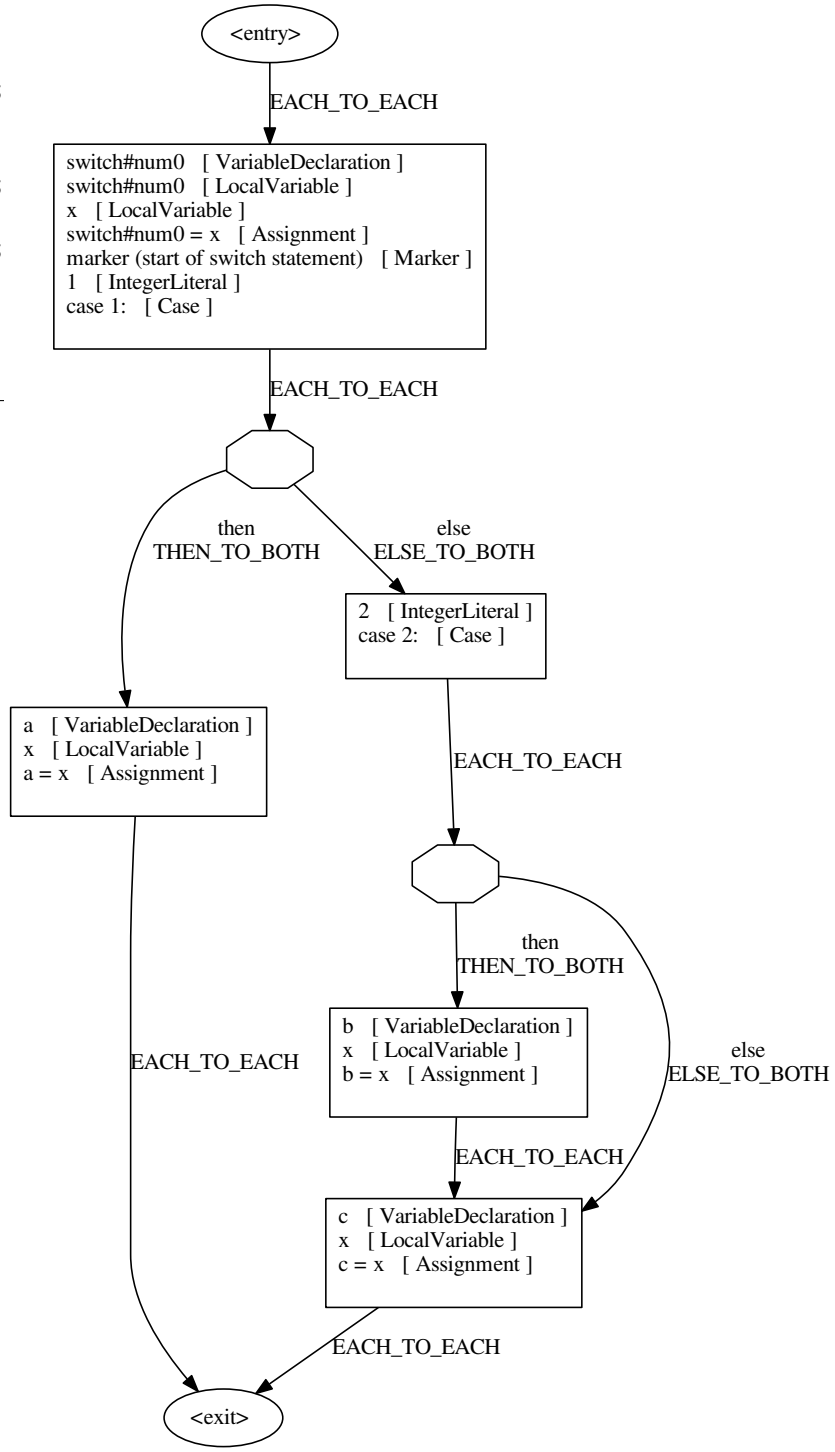


Figure 7: Example of a switch statement with case, default and fall through.

3.3 AST to CFG Translation

This section gives a high-level overview of the translation process from the abstract syntax tree to the control-flow graph as described in Section 3.1.

First, we define several entities, which will be used in the translation.

Definition 3.6 (Extended Node)**.** *In the translation process the data type* extended node *is used. An extended node can be one of four possibilities:*

- ***Simple extended node.*** *An extended node can just be a wrapper for a node that does not throw an exception, as defined in Definition 3.5.*
- ***Exception extended node.*** *Similar to a simple node, an exception extended node contains a node, but this node might throw an exception at run time.*
- ***Unconditional jump.*** *An unconditional jump indicates that control flow proceeds non-sequentially to a location indicated by a target label.*
- ***Conditional jump.*** *A conditional jump can follow an extended node that contains a node of boolean type. It contains two target labels, one if the node evaluates to true and one for false.*

Comparison of nodes and extended nodes. Nodes themselves never contain control flow information; they only represent computation.

An extended node is a wrapper around a node that represents control flow information. It contains: a node, a label, a predecessor, and a successor.

An extended node is temporarily used to keep track of some control flow information. Later, the basic block data structures are created, and they represent the control flow. (And at that point the extended nodes are discarded.)

Definition 3.7 (Label)**.** *A* label *is a marker that is used to refer to extended nodes. It is used only temporarily during CFG construction.*

The process of translating an AST to a CFG proceeds in three distinct phases.

1. **Phase one.** In the first phase, a single linear sequence of extended nodes is created. The control flow is implicitly assumed to be sequential through the sequence of extended nodes, until a (conditional or unconditional) jump is encountered in an if, for, while, or switch statement, in which case the jump target decides where execution proceeds.

   The labels used as targets of jumps are associated with positions in this sequence and are managed by maintaining a binding function from labels to sequence positions. The advantage of having this indirection is that one can create a label and associate with the next free position in the sequence, without knowing which exact extended node will be placed there. Furthermore, labels can be created and used before they are actually bound to their correct position in the sequence (e.g., when that position is not yet known). At the end, the binding function can be used to resolve labels to extended nodes.

   Furthermore, phase one also computes a mapping from AST tree elements to nodes, as well as a set of leaders. A *leader* is an extended node for which one of the following conditions applies:

   - It is the first extended node in the sequence.
   - It is the target of a jump (i.e. there is a label bound to the location of the node in the sequence).
   - It is the first node following a jump.

2. **Phase two.** Phase two translates the linear representation to a control-flow graph by performing the following transformations:

   - Simple extended nodes are translated to regular basic blocks, where multiple nodes can be grouped in one regular basic block.
   - Exception extended nodes are translated to exception basic blocks with the correct edges.
   - Unconditional jumps are replaced with edges between the correct basic blocks.
   - Conditional jumps are replaced by a conditional basic block.

   To greatly simplify the implementation, phase two is allowed to produce a degenerated control-flow graph. In particular, the following deficiencies are possible:

   - Regular basic blocks might be empty.
   - Some conditional basic blocks might be unnecessary, in that they have the same target for both the "then" as well as the "else" branch.
   - Two consecutive, non-empty, regular basic blocks can exist, even if the second block has only exactly one predecessor and the two blocks could thus be merged.

3. **Phase three.** In the third and last phase, the control-flow graph is transformed such that the deficiencies remaining from phase two are removed. It is ensured that removing one kind of deficiency does not create another degenerate case.

3.3.1 Desugaring

Desugaring means replacing complicated source language constructs by simpler ones, or removing syntactic sugar from an input program. Originally, we intended for the control flow graph representation to be as close as possible to the Java abstract syntax tree to simplify the mapping from tree to CFG node and back and to reuse existing checker code written in terms of trees. However, we ran into several cases that were better handled by desugaring.

- We decided to represent implicit conversion operations like boxing, unboxing, widening, and narrowing as explicit CFG nodes because they change the type of a value. For example, implicit unboxing of an `Integer` will be translated into a call to `Integer.intValue`. The pre-conversion type can be associated with the original node and the post-conversion type can be associated with the explicit conversion node. It also makes it possible for the transfer function to operate on the conversion nodes.
- Enhanced for loops are defined in terms of a complicated translation into simpler operations, including field accesses, branches, and method calls that could affect dataflow information. It would be prohibitively difficult for a checker writer to write a transfer function that correctly accounted for all of those operations, so we desugar enhanced for loops.
- Once we decided to make conversion nodes explicit it made sense to desugar compound assignments. A compound assignment like

  ```
  Integer i; i += 3;
  ```

  performs both an unboxing and a boxing operation on `i`. Desugaring all compound assignments greatly reduced the total number of node classes.
- String concatenation assignments are also desugared. A string concatenation assignment like

  ```
  String s; s += "str";
  ```

is represented as

```
String s; s = s + "str";
```

in CFGs. This avoids duplicating the same logic that is necessary in assignment and concatenation nodes, which was error prone.

In order to desugar code and still maintain the invariant that every CFG node maps to a tree, we needed to create new AST tree nodes that were not present in the input program. Javac allows us to do this through non-supported APIs and we wrote some utility classes in `javacutil` to make the process easier. The new trees are created during CFG building and they persist as long as some CFG node refers to them. However, the trees are not inserted into the AST, so they are not type-checked or seen by other tree visitors. Their main purpose is to carry Java types and to satisfy AnnotatedTypeFactory methods.

A further complication is that these newly-introduced AST trees are not part of the TreePath when visiting the AST. We work around this problem by giving the AnnotatedTypeFactory a mapping, called the `pathHack`, from newly-introduced trees to their containing MethodTree and ClassTree.

Possibly even worse, we needed to create fake symbols for variables created when desugaring enhanced for loops. Javac does not expose the ability to create a symbol, so we created a new subclass of Symbol.VarSymbol called `javacutil.tree.DetachedVarSymbol` for this purpose. AnnotatedTypeFactory explicitly checks for DetachedVarSymbols in its DeclarationFromElement method.

3.3.2 Conversions and node-tree mapping

As mentioned in Section 3.3.1, we represent implicit Java type conversions such as boxing, unboxing, widening, and narrowing by explicit CFG nodes. This means that some AST tree nodes correspond to multiple CFG nodes: a pre-conversion node and a post-conversion node. We will describe how the conversions work and how the node-tree mappings are implemented.

Boxing and unboxing are represented in terms of calls to Java standard library methods. Boxing corresponds to a call to `BoxedClass.valueOf` while unboxing corresponds to a call to `BoxedClass.*Value`. This allows annotations on the library methods, as well as transfer functions for method invocations, to apply to the conversions with no special work on the part of a checker developer.

Widening and narrowing conversions are represented as special node types, although it would be more consistent to change them into type casts.

We maintain the invariant that a CFG node maps to zero or one AST tree and almost all of them map to a single tree. But we can't maintain a unique inverse mapping because some trees have both pre- and post-conversion nodes. Instead, we remember two mappings, one from tree to pre-conversion node and, for those trees that were converted, one from tree to post-conversion node. Both the CFGBuilder and the ControlFlowGraph store two separate mappings. The Analysis class explicitly stores the tree to pre-conversion node mapping as `treeLookup` and it indirectly uses the tree to post-conversion mapping in `Analysis.getValue(Tree)`. This has effectively hidden the distinction between pre and post-conversion nodes from the Checker Framework, but in the long run it may be necessary to expose it.

4 Dataflow Analysis

This section describes how the dataflow analysis over the control-flow graph is performed and how to implement a particular analysis.

Roughly, a dataflow analysis in the framework works as follows. Given the abstract syntax tree of a method, the framework computes the corresponding control-flow graph as described in Section 3. Then, a simple

forward or backward iterative algorithm is used to compute a fix-point, by iteratively applying a set of transfer functions to the nodes in the CFG. These transfer functions are specific to the particular analysis and are used to approximate the run-time behavior of different statements and expressions.

4.1 Managing Intermediate Results of the Analysis

Conceptually, the dataflow analysis computes an abstract value for every node and flow expression[1]. The transfer function (Section 4.3) produces these abstract values, taking as input the abstract values computed earlier for sub-expressions. For instance, in a constant propagation analysis, the transfer function for addition (`+`) would look at the abstract values for the left and right operand, and determine that the `AdditionNode` is a constant if and only if both operands are constant.

An analysis result contains two parts:

1. The *node-value mapping* (`Analysis.nodeValues`) maps `Node`s to their abstract values. Only nodes that can take on an abstract value are used as keys. For example, in the Checker Framework, the mapping is from expression nodes to annotated types.

   The framework consciously does not store the abstract value directly in the node, to remove any coupling between the control-flow graph and a particular analysis. This allows the control-flow graph to be constructed only once, and then reused for different dataflow analyses.

2. A set of *stores*. Each store maps a flow expression to an abstract value. Each store is associated with a specific program point.

   The stores tracked by an analysis implement the `Store` interface, which defines the following operations:

   - Least upper bound: Compute the least upper bound of two stores (e.g., at a merge-point in the control-flow graph).
   - Equivalence: Compare two stores if they are (semantically) different, which is used to determine if a fix-point is reached in the dataflow analysis. Note that reference-equality is most likely not sufficient.
   - Copy mechanism: Clone a store to get an exact copy.

   The store is analysis-dependent, but the framework provides a default store implementation which can be reused. The default implementation is

   ```
   org.checkerframework.framework.flow.CFStore
   ```

   What information is tracked in the store depends on the analysis to be performed. Some examples of stores include

   ```
   org.checkerframework.checker.initialization.InitializationStore
   org.checkerframework.checker.nullness.NullnessStore
   ```

   Every store is associated with a particular point in the control-flow graph, and all stores are managed by the framework. It saves an explicit store for the start of each basic block. When dataflow information is requested for a later point in a block, the analysis applies the transfer function to recompute it from the initial store.

After an analysis has iterated to a fix-point, the computed dataflow information is maintained in an AnalysisResult, which can map either nodes or trees to abstract values.

[1] Certain dataflow analyses might choose not to produce an abstract value for every node. For instance, a constant propagation analysis would only be concerned with nodes of a numerical type, and could ignore other nodes.

4.2 Answering Questions

After the flow analysis for a particular method has been computed, there are two kinds of information that have been computed. Firstly, the node-value mapping stores an abstract value for every node, and secondly, the information maintained in various stores is available.

Two kinds of queries are possible to the dataflow analysis after the analysis is complete:

1. For a given AST tree node, what is its abstract value? Both pre- and postconversion values can be retrieved. A discussion of conversions can be found in Section 3.3.2.
2. For a given AST tree node, what is the state right after this AST tree node? Examples of questions include:
   - Which locks are currently held?
   - Are all fields of a given object initialized?

The store may first need to be (re-)computed, as the framework does not store all intermediate stores but rather only those for key positions as described in Section 4.1.

To support both kinds of queries, the framework builds a map from AST tree nodes (of type `com.sun.source.tree.Tree`) to CFG nodes. To answer questions of the first type it is then possible to go from the AST tree node to the CFG node and look up its abstract value in the node-value mapping (this is provided by the framework). By default, the abstract value returned for a tree by `Analysis.getValue(Tree)` includes any implicit conversions because it uses the mapping from tree node to post-conversion CFG node. To request the pre-conversion value, one currently uses the `ControlFlowGraph.treelookup` map directly.

To support questions of the second kind, every node has a reference to the basic block it is part of. Thus, for a given AST tree node, the framework can determine the CFG node and thereby the CFG basic block, and compute the necessary store to answer the question.

4.3 Transfer Function

A transfer function is an object that has a transfer method for every `Node` type, and also a transfer method for procedure entry.

- A transfer method for a `Node` type takes a store and the node, and produces an updated store. This is achieved by implementing the `NodeVisitor<S, S>` interface for the store type `S`.

  These transfer methods also get access to the abstract value of any sub-node of the node `n` under consideration. This is not limited to immediate children, but the abstract value for any node contained in `n` can be queried.

- A transfer method for procedure entry returns the initial store, given the list of parameters (as `LocalVariableNode`s that represent the formal parameters) and the `MethodTree` (useful if the initial store depends on the procedure signature, for instance).

4.4 Flow Rules

As mentioned in Section 3.2.4, dataflow analysis conceptually maintains two stores for each program point, a then store containing information valid when the previous boolean-valued expression was true and an else store containing information valid when the expression was false. In many cases, there is only a single store because there is no boolean-valued expression to split on or there was an expression, but it yielded no useful

dataflow information. However, any CFG edge may potentially have a predecessor with two stores and a successor with two stores.

We could simply propagate information from both predecessor stores to both successor stores, but that would throw away useful information, so we define five flow rules that allow more precise propagation.

- `EACH_TO_EACH` is the default rule for an edge with a predecessor that is not a `ConditionalBlock`. It propagates information from the then store of the predecessor to the then store of the successor and from the else store of the predecessor to the else store of the successor.
- `THEN_TO_BOTH` is the default rule for an edge from a `ConditionalBlock` to its then successor. It propagates information from the then store of its predecessor to both the then and else stores of the then successor, thereby splitting the conditional store to take advantage of the fact that the condition is known to be true.
- `ELSE_TO_BOTH` is the default rule for an edge from a `ConditionalBlock` to its else successor. It propagates information from the else store of its predecessor to both the then and else stores of the else successor, thereby splitting the conditional store to take advantage of the fact that the condition is known to be false.
- `THEN_TO_THEN` is a special rule for a short-circuit edge from a `ConditionalBlock` to its then successor. It only propagates information from the then store of its predecessor to the then store of its successor. This flow rule is used for conditional or expressions because the else store of `a || b` is not influenced by the then store of `a`.
- `ELSE_TO_ELSE` is a special rule for a short-circuit edge from a `ConditionalBlock` to its else successor. It only propagates information from the else store of its predecessor to the else store of its successor. This flow rule is used for conditional and expressions because the then store of `a && b` is not influenced by the else store of `a`.

Note that the more precise flow rules `THEN_TO_THEN` and `ELSE_TO_ELSE` improve the precision of the store they do not write to. In other words, `THEN_TO_THEN` yields a more precise else store of its successor by not propagating information to the else store which might conflict with information already there, and conversely for `ELSE_TO_ELSE`. See Section 3.2.6 for more details and an example.

Currently, we only use flow rules for short-circuiting edges of conditional ands and ors. The CFG builder sets the flow rule of each short-circuiting edge as it builds the CFG for the conditional and/or expression.

The dataflow analysis logic requires that both the then and the else store of each block contain some information before the block is analyzed, so it is a requirement that at least one predecessor block writes the then store and at least one writes the else store.

4.5 Concurrency

By default, the Dataflow Framework analyzes the code as if it is executed sequentially. This is unsound if the code is run concurrently. Use the `-AconcurrentSemantics` command-line option to enable concurrent semantics.

In the concurrent mode, the dataflow analysis cannot infer any local information for fields. This is because after a local update, another thread might change the field's value before the next use.

An exception to this are monotonic type properties, such as the `@MonotonicNonNull` annotation of the nullness type system. The meta-annotation `@MonotonicQualifier` declares that a qualifier behaves monotonically, however it is not yet used to preserve dataflow information about fields under concurrent semantics.

5 Example: Constant Propagation

As an example, consider a constant propagation analysis for local variables and integer values. To run constant propagation and create a PDF of the CFG, run `java org.checkerframework.dataflow.cfg.playground.ConstantPropagationPdf MyFile.java`.

Abstract values. A class `Constant` is used as an abstract value, which can either be *top* (more than one integer value seen), *bottom* (no value seen yet), or *constant* (exactly one value seen; in which case the value is also stored).

The store. The store maps `Node`s to `Constant`, where only `LocalVariableNode`s and `IntegerLiteralNode`s are used as keys. Only those two nodes actually are of interest (there is no addition/multiplication/etc. yet, and other constructs like fields are not yet supported by the analysis).

Two different instances of `LocalVariableNode` can be uses of the same local variable, and thus the `equals` method has been implemented accordingly. Therefore, every local variable occurs at most once in the store, even if multiple (equal) `LocalVariableNode`s for it exist.

The transfer function. The transfer function is very simple. The initial store contains *top* for all parameters, as any value could have been passed in. When an integer literal is encountered, the store is extended to indicate what abstract value this literal stands for. Furthermore, for an assignment, if the left-hand side is a local variable, the transfer function updates its abstract value in the store with the abstract value of the right-hand side (which can be looked up in the store).

To illustrate how we can have different information in the then and else block of a conditional, I also implemented another transfer function that considers the `EqualToNode`, and if it is of the form `a == e` for a local variable `a` and constant `e`, passes the correct information to one of the branches. This is also shown in Figure 8.

Example. A small example is shown in Figure 8.

6 Example: Live Variable

A live variable analysis for local variables and fields was implemented to show the backward analysis works properly. To run live variable analysis and create a PDF of the CFG, run `java org.checkerframework.dataflow.cfg.playground.LiveV MyFile.java`.

Abstract values. A class `LiveVarValue` is a live variable (which is represented by a node) wrapper turning node into abstract value. A node can be `LocalVariableNode` or `FieldAccessNode`.

The store. The live variable store `LiveVarStore` has a field `liveVarValueSet` which contains a set of `LiveVarValue`s. Only `LocalVariableNode` or `FieldAccessNode` will be considered as a live variable and added to `liveVarValueSet`. The store defines methods `putLiveVar(LiveVarValue)` and `killLiveVar(LiveVarValue)` to add and kill live variables.

The transfer function. The transfer function `LiveVarTransfer` initializes empty stores at normal and exceptional exit blocks (because this is a backward transfer function). The transfer function visits assignments to update the live variable values in the stores.

Example. An example is shown in Figure 9.

7 Example: Very Busy Expression Analysis

An expression is "very busy" if no matter what path is taken, the expression is used before any variables occurring in it are redefined. This analysis is a backward analysis that intersects abstract states from

```
1  class Test {
2    void test(boolean b, int a) {
3      int x = 1;
4      int y = 0;
5      if (b) {
6        x = 2;
7      } else {
8        x = 2;
9        y = a;
10     }
11     x = 3;
12     if (a == 2) {
13       x = 4;
14     }
15   }
16 }
```

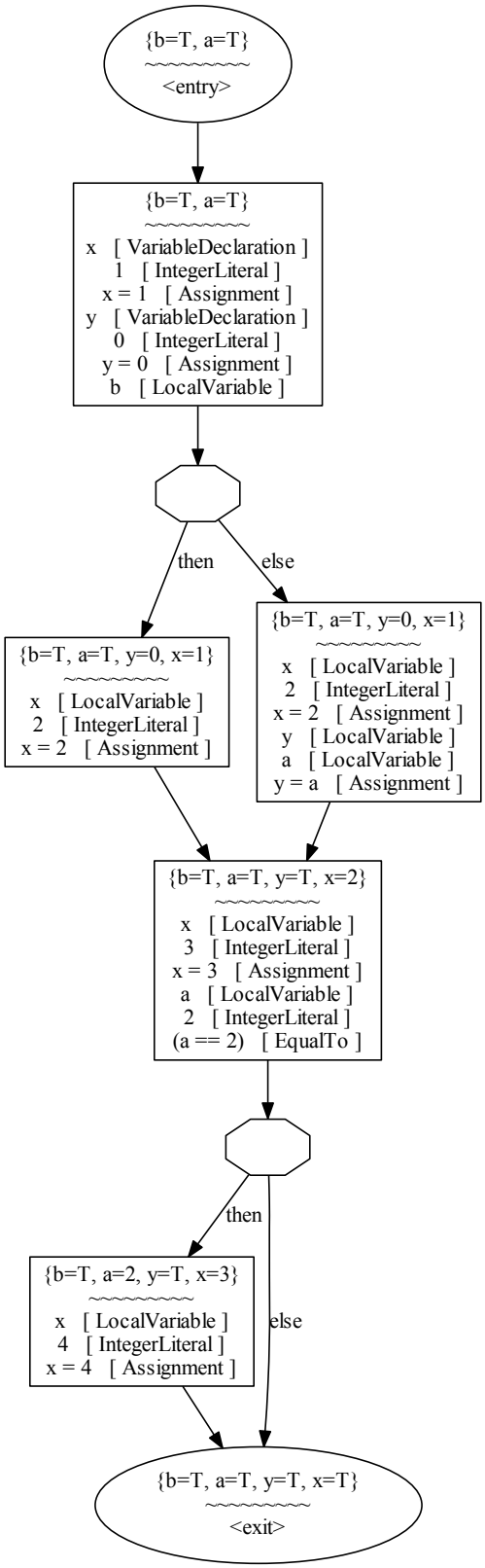


Figure 8: Simple sequential program to illustrate constant propagation. Intermediate analysis results are shown.

```
1  class Test {
2    public void test() {
3      int a = 1, b = 2,
           c = 3;
4      if (a > 0) {
5        int d = a + c;
6      } else {
7        int e = a + b;
8      }
9    }
10 }
```

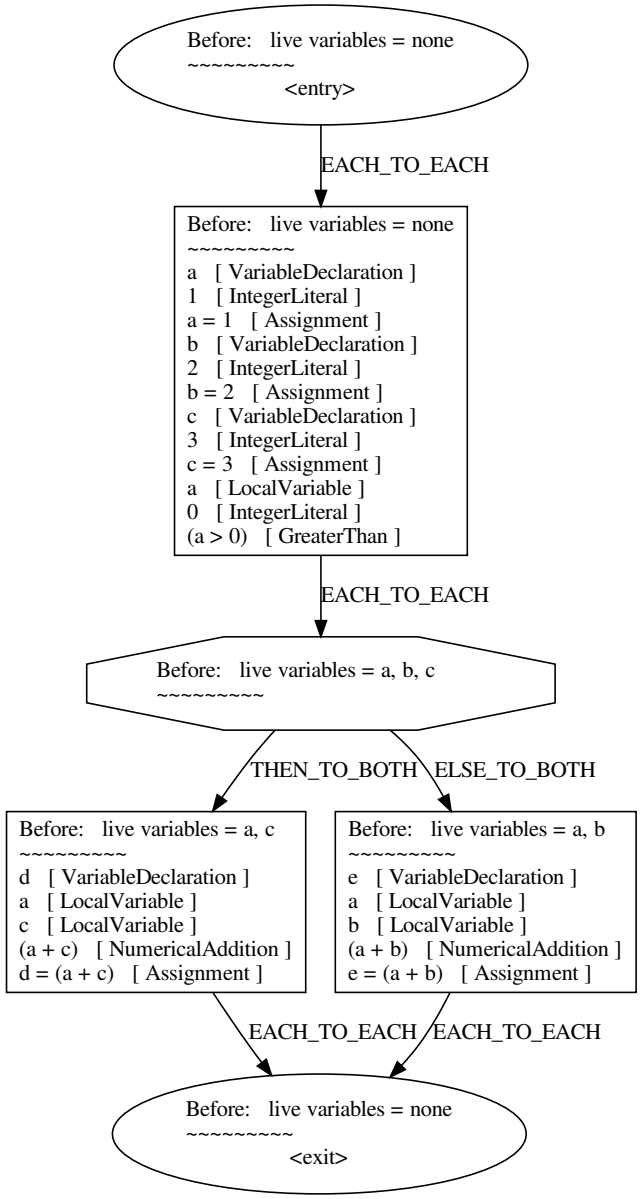


Figure 9: Simple sequential program to illustrate live variable. Intermediate analysis results are shown.

successors.

To run very busy analysis and create a PDF of the CFG, run

```
java org.checkerframework.dataflow.cfg.playground.BusyExpressionPdf MyFile.java
```

Abstract values. The `BusyExprValue` is a very busy expression, represented by a node. The node can be any `BinaryOperationNode`, such as `NumericalAdditionNode` or `LeftShiftNode`.

The store. The busy expression store `BusyExprStore` has a field `busyExprValueSet` which contains a set of

BusyExprValues. If a node is a BinaryOperationNode, addUseInExpression(Node) will recursively analyze the subexpressions of the node to determine if they are nested BinaryOperationNodes. It then uses putBusyExpr(BusyExprValue) to put the expression in the set. killBusyExpr(Node) is used when a variable is re-assigned.

The transfer function. The transfer function BusyExprTransfer initializes empty stores at normal and exceptional exit blocks (because this is a backward transfer function). The transfer function visits assignment, method invocation, and return statement nodes to update the busy expression values in the stores.

Example. An example is shown in Figure 10.

8 Example: Reaching Definition Analysis

The reaching definitions for a given program point are those assignments that may have updated the current values of variables. This reaching definition analysis is a standard example of a forward analysis.

Abstract values. A class ReachingDefinitionNode is used as an abstract value, which can only wrap AssignmentNode. The reaching definition analysis processes such values in the store.

The store. The reaching definition store ReachingDefinitionStore has a field reachingDefSet which contains a set of ReachingDefinitionNodes. The store defines methods putDef(ReachingDefinitionValue) and killDef(Node) to add and kill reaching definitions.

The transfer function. The transfer function visits AssignmentNode to update the information of reaching definitions in the stores. A reaching definition will be killed in the store when its left-hand side is same as that of the new generated value, and the new generated value will be added into the store.

Example. An example is shown in Figure 11.

9 Default Analysis

9.1 Overview

The default flow-sensitive analysis org.checkerframework.framework.flow.CFAnalysis works for any checker defined in the Checker Framework. This generality is both a strength and a weakness because the default analysis can always run but the facts it can deduce are limited. The default analysis is extensible so checkers can add logic specific to their own qualifiers.

The default flow-sensitive analysis takes advantage of several forms of control-flow to improve the precision of type qualifiers. It tracks assignments to flow expressions, propagating type qualifiers from the right-hand side of the assignment. It considers equality and inequality tests to propagate the most precise qualifiers from the left or right-hand side to the true (resp. false) successor. It also applies type qualifiers from method postconditions after calls.

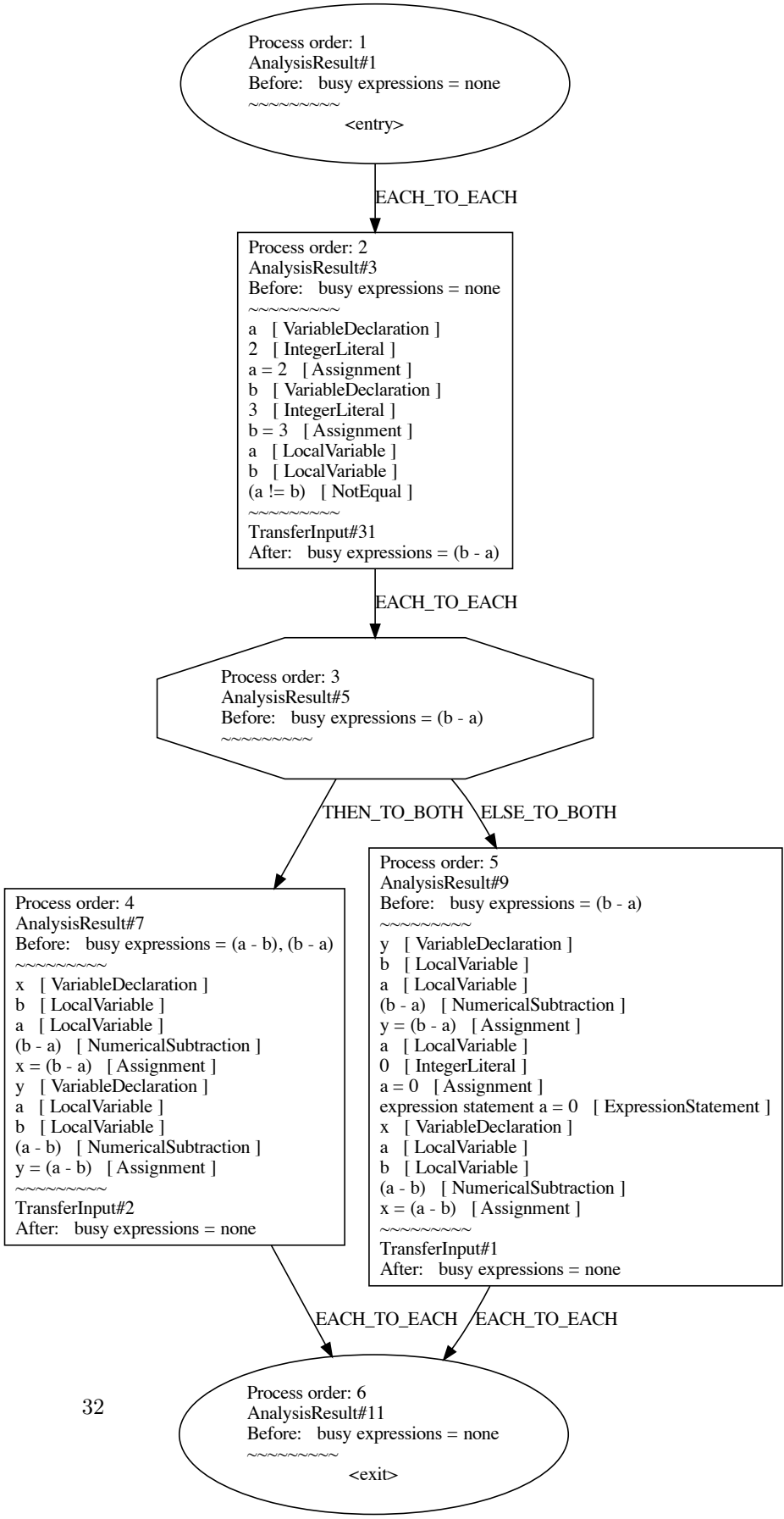
9.2 Interaction of the Checker Framework and the Dataflow Analysis

This section describes how the dataflow analysis is integrated into the Checker Framework to enable flow-sensitive type checking.

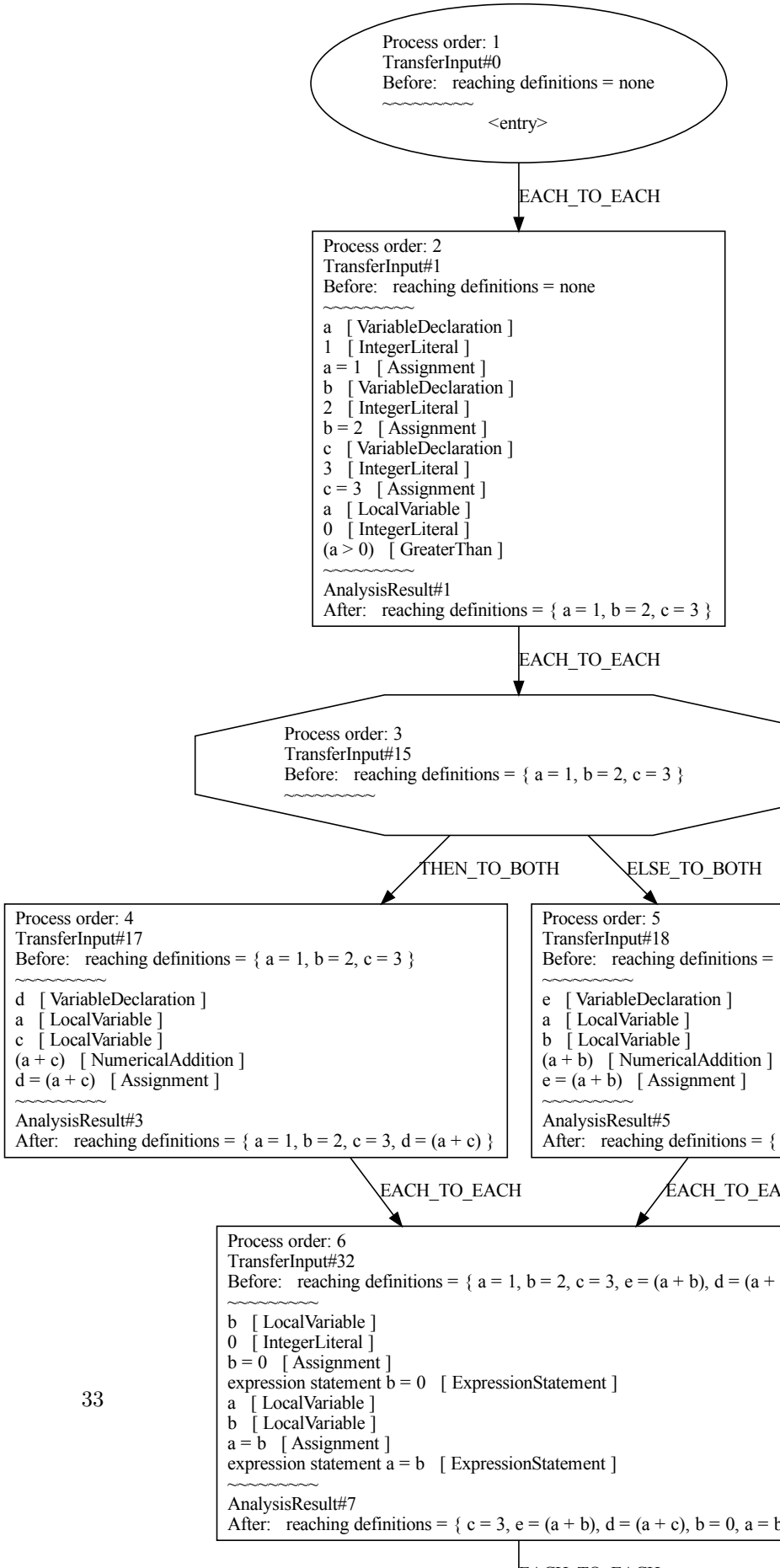
A main purpose of a type factory is to create an AnnotatedTypeMirror based on an input tree node. Using the results of the dataflow analysis, the type factory can return a more refined type than otherwise possible.

Type factories that extend from GeneralAnnotatedTypeFactory and set the constructor parameter useFlow to true will automatically run dataflow analysis and use the result of the analysis when creating an AnnotatedTypeMirror.

```
1  class Test {
2    public void test() {
3      int a = 2, b = 3;
4      if (a != b) {
5        int x = b - a;
6        int y = a - b;
7      } else {
8        int y = b - a;
9        a = 0;
10       int x = a - b;
11     }
12   }
13 }
```



```
1  class Test {
2    public void test() {
3      int a = 1, b = 2,
           c = 3;
4      if (a > 0) {
5        int d = a + c;
6      } else {
7        int e = a + b;
8      }
9      b = 0;
10     a = b;
11   }
12 }
```



The first time that a `GenericAnnotatedTypeFactory` instance visits a `ClassTree`, the type factory runs the dataflow analysis on all the field initializers of the class first, then the bodies of methods in the class, and then finally the dataflow analysis is run recursively on the members of nested classes. The result of dataflow analysis is stored in the `GenericAnnotatedTypeFactory` instance.

When creating an `AnnotatedTypeMirror` for a tree node, the type factory queries the result of the dataflow analysis to determine if a more refined type for the tree node was inferred by the analysis. This is the first type of query described in Section 4.2.

Dataflow itself uses the type factory to get the initial `AnnotatedTypeMirror` for a tree node in the following way.

For a given node `n`

- If it has no corresponding AST tree node, use "top" as its abstract value.
- If it has a corresponding AST tree node, ask the `AnnotatedTypeFactory` about the type of the tree node. The factory will then use its checker-dependent logic to compute this type. A typical implementation will look at the type of sub-trees and compute the overall type based on the information about these sub-trees.

  Note that the factory recursively uses information provided by the flow analysis to determine the types of sub-trees. There is a check in `Analysis.getValue` that the node whose type is being requested is a sub-node of the node to which the transfer function is currently being applied. For other nodes, the analysis will return `null` (i.e., no information) and the factory will return the flow-insensitive annotated type.

The `AnnotatedTypeFactory` and helper classes need to be prepared to work even when dataflow analysis is not finished yet. Code should either check whether dataflow analysis is still in progress (using `analysis.isRunning()`) or handle possible null values. The factory should return conservative, flow-insensitive types if the analysis is still in progress.

9.3 The Checker Framework Store and Dealing with Aliasing

> Word of caution: The exact rules of what information is retained may or may not be implemented exactly as described here. This is a good starting point in any case, but if very precise information is needed, then the source code is very readable and well documented.

The Dataflow Framework provides a default implementation of a store with the class `CFAbstractStore`, which is used (as `CFStore`) as the default store if a checker does not provide its own implementation. This implementation of a store tracks the following information:

- Abstract values of local variables.
- Abstract values of fields where the receiver is an access sequence compose of the following:
  - Field access.
  - Local variable.
  - Self reference (i.e., `this`).
  - Pure or non-pure method call.

The most challenging part is ensuring that the information about field accesses is kept up to date in the face of incomplete aliasing information. In particular, at method calls and assignments care needs to be taken about which information is still valid afterwards.

The store maintains a mapping from field accesses (as defined in Section 9.3.1) to abstract values, and the subsequent sections describe the operations that keep this mapping up-to-date.

A side-effect-free method has no visible side-effects, such as setting a field of an object that existed before the method was called. A deterministic method returns the same value (according to ==) every time it is called with the same arguments and in the same environment. A pure method is both side-effect-free and deterministic. The Dataflow Framework respects the `@SideEffectFree`, `@Deterministic`,and `@Pure` annotations of the Checker Framework

9.3.1 Internal Representation of field access

To keep track of the abstract values of fields, the Dataflow Framework uses its own representation of field accesses (that is different from the `Node` type introduced earlier). This data type is defined inductively as follows:

⟨*FieldAccess*⟩	::=	⟨*Receiver*⟩ ⟨*Field*⟩	Java field (identified by its `Element`)
⟨*Receiver*⟩	::=	⟨*SelfReference*⟩	`this`
	|	⟨*LocalVariable*⟩	local variable (identified by its `Element`)
	|	⟨*FieldAccess*⟩	
	|	⟨*MethodCall*⟩	Java method call of a method
	|	⟨*Unknown*⟩	any other Java expression
⟨*MethodCall*⟩	::=	⟨*Receiver*⟩ '.' ⟨*Method*⟩ '(' ⟨*Receiver*⟩,* ')'	

⟨*Unknown*⟩ is only used to determine which information needs to be removed (e.g., after an assignment), but no field access that contains ⟨*Unknown*⟩ is stored in the mapping to abstract values. For instance, ⟨*Unknown*⟩ could stand for a non-pure method call, an array access, or a ternary expression.

9.3.2 Updating Information in the Store

In the following, let o be any ⟨*Receiver*⟩, x a local variable, f a field, m a pure method, and e an expression. Furthermore, we assume to have access to a predicate might_alias(o_1, o_2) that returns true if and only if o_1 might alias o_2; see Section 9.3.2.5.

9.3.2.1 At Field Assignments For a field update of the form $o_1.f_1 = e$, the dataflow analysis first determines the abstract value e_{val} for e. Then, it updates the store S as follows.

1. For every field access $o_2.f_1$ that is a key in S, remove its information if o_2 lexically contains a ⟨*Receiver*⟩ that *might* alias $o_1.f_1$ as determined by the might_alias predicate. Note that the "lexically contains" notion for pure method calls includes both the receiver as well as the arguments.

   This includes the case where $o_1 = o_2$ (that is, they are syntactically the same) and the case where o_2 and $o_1.f_1$ might be aliases.

2. $S[o_1.f_1] = e_{\text{val}}$

9.3.2.2 At Local Variable Assignments For a local variable assignment of the form $x = e$, the dataflow analysis first determines the abstract value e_{val} for e. Then, it updates the store S as follows.

1. For every field access $o_2.f_2$ that is a key in S, remove its information if o_2 lexically contains a ⟨*Receiver*⟩ that *might* alias x as determined by the might_alias predicate.
2. $S[x] = e_{\text{val}}$

9.3.2.3 At Other Assignments For any other assignment $z = e$ where the assignment target z is not a field or local variable, the dataflow analysis first determines the abstract value e_{val} for e. Then, it updates the store S as follows.

1. For every field access $o_2.f_2$, remove its information if o_2 lexically contains a ⟨*Receiver*⟩ that *might* alias z as determined by the might_alias predicate.

9.3.2.4 At Non-Pure Method Calls A non-pure method call might modify the value of any field arbitrarily. Therefore, at a method call, any information about fields is lost.

9.3.2.5 Alias Information The Checker Framework does not include an aliasing analysis, which could provide precise aliasing information. For this reason, we implement the predicate might_alias as follows:

$$\text{might_alias}(o_1, o_2) := (\text{type}(o_1) <: \text{type}(o_2) \;\text{ or }\; \text{type}(o_2) <: \text{type}(o_1))$$

where type(o) determines the Java type of a reference o and $<:$ denotes standard Java subtyping.