Dataflow Framework for Checker Framework

Andreas Abel
abel@cs.washington.edu

Kivanc Muslu
kivanc@cs.washington.edu

Brandon Myers
bdmyers@cs.washington.edu

ABSTRACT

This paper presents a general dataflow framework for Java that can be used to replace the data flow implementation in the Checker Framework, a tool that adds pluggable type systems to Java. Currently, the Checker Framework uses a generalization and extension of the data flow analysis in Oracle's javac, which is hardcoded to perform a few analyses like definite assignment. Because of this, the analyses in the Checker Framework suffer from imprecision. Moreover, no detailed documentation for the code is available.

Our framework aims at overcoming these flaws. It is implemented to be modular, precise and we believe that it is easy to implement different data flow analyses using it. To show the latter, we have completely eliminated the existing data flow analyses in the Checker Framework and implemented the same analyses using our framework. The fact that all these analyses were written easily and with fewer lines of code shows the modularity of our framework. Finally, we have run the checkers with and without our framework and seen that checkers running with our framework issue fewer false positives and false negatives, which shows its precision.

We believe that our framework is a useful contribution because it can improve the expressiveness and precision of the Checker Framework. Moreover, as we tried to keep the coupling between the Checker Framework and our data flow framework to a minimum, it could also be used with other tools that need data flow analyses.

Categories and Subject Descriptors

D3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms

Languages

1. INTRODUCTION

Dataflow analysis (DFA) [4] is a technique to statically derive information about the dynamic behavior of a program. This information includes facts about the use and definition of variables and information about control and data dependencies. Typical problems that can be solved using DFA are: Which definitions can reach a particular program point? Which variables are live (i.e. read before their next write update) at a program point? Which values might a variable have at a program point? Which variables might be uninitialized?

A dataflow framework as we refer to it in this paper is a structure that, given three components of a DFA (section 3.5), creates a DFA, performs the DFA over the control-flow graph of the program, and provides inferred values for the nodes of the abstract syntax tree.

Oracle's implementation of javac relies on dataflow analysis for computing definite assignment, but it only has a hard-coded analysis that is not useful for other DFAs. Perhaps there was no perceived need for a general DFF, since DFAs are often used to enable compiler optimizations but the Java Virtual Machine's just-in-time compiler can produce better optimizations at runtime.

However, having a general DFF for Java would be useful for other purposes. For example, the Checker Framework[1] is a tool that enhances Java's type system by letting the developers add pluggable types to their code. It can be used to prove a given kind of runtime() error, such as a null pointer dereference, does not exist in a program, and it needs DFA to help make the analysis more precise. Currently, the Checker Framework uses a modified version of javac's `Flow` implementation. But as this implementation is highly specialized and poorly documented, the attempt to generalize it leads to a number of problems and restrictions; section 2 discusses some of the problems in detail.

Our framework replaces the current flow inference with a well documented DFF that overcomes some deficiencies of the current version. This makes the existing checkers more precise. Being able to use arbitrary lattices, it is also possible to build a wider range of checkers.

Our framework could also be used as a basis for a building a stand-alone DFF. The way in which our framework depends

[1] `http://types.cs.washington.edu/checker-framework/`

on Checker Framework is `TreeUtils`, which is a helper class that implements some useful operations on `Tree`s. In section 5 we explain how this dependence can be removed. As a result, we believe that our framework also has the potential to be used for other projects, such as other partial verification tools, bug-finding and property-proving tools, and tools that perform optimizations.

The remainder of the paper is organized as follows: section 2 explains our motivation for replacing the the existing DFA in the Checker Framework. Section 3 explains the implementation of the important parts of our framework. Section 4 discusses our evaluation methodology and results. Section 5 presents related work, and section 6 concludes.

2. MOTIVATION

Our motivation to replace the current flow sensitive analysis of the Checker Framework with a new DFF implementation has two main components: (1) flow sensitive analysis in the Checker Framework is imprecise and should be improved, (2) it is worth improving because the Checker Framework is a highly scalable and easy-to-use tool that can find documentation and execution bugs in well-tested and widely used open source software.

2.1 Problems with the Checker Framework

In this section we state the existing problems in the Checker Framework's internal implementation: Its flow sensitive analysis is imprecise, and it does not properly handle aliases.

2.1.1 Imprecise Merge of Variable Values

The Checker Framework's flow sensitive analysis cannot always infer the best type when inferred values for variables need to be merged during the traversal of the AST. The analysis was implemented as an extension and generalization of the definite assignment analysis in javac. Javac's definite assignment is highly optimized and so uses only the one required bit of information for each variable at each program point: a variable is either initialized or uninitialized. This is not true of pluggable types checking as the sub-type hierarchy (lattice) of the analysis can be wider and taller. As an example, a part of the Signature Checker's type hierarchy is given in Figure 1. Clearly seen from the example, an `String` can get 6 different types during the program analysis.

To permit a variable to have multiple annotations during the program analysis, `Flow` implementation in the Checker Framework uses a `GenKillBits` implementation, which is a map from `AnnotationMirror` (annotations) to `BitSet` (bits). This implementation is briefly explained by Papi et al. in [7, §3.7]. It basically creates a bit vector for each type in the hierarchy, where each bit indicates whether the variable has that type. It attempts to perform flow operations like `split` and `merge` over these bits. However, the implementation is incomplete: it does not define a *least* upper bound in `merge`. As the result, whenever two different types need to be joined at a program point, `merge` just returns the top element in the sub-type hierarchy.

Consider the following example that uses the type hierarchy presented in Figure 1:

```
1. @BinaryName String bn
       = "java.lang.Thread$ThreadLocal";
2. @SourceName String sn
       = "java.lang.Thread";
3. String temp;
4. if (Math.random() == 0) { temp = bn; }
5. else { temp = sn; }
// After line 5, temp should be @BinaryName.
6. @BinaryName String bn2 = temp;
```

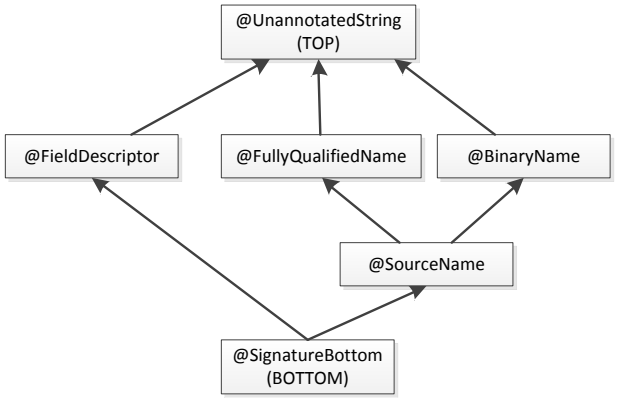

Figure 1: Type hierarchy for Signature Checker. Arrows represent super types. (e.g., join of @BinaryName and @SourceName is equal to @BinaryName since @BinaryName is a super type of @SourceName)

When we run this code with Signature Checker (which uses the flow inferences of the Checker Framework flow analysis), it raises an error at line 6. The type-checker complains that 'bn2' is defined as a `BinaryName`, however an `UnannotatedString` (temp) is given instead. Therefore, current `merge` implementation in `Flow` is sound (since it is still safe to return top type), however not precise (reports a false positive).

As demonstrated in the example, the Checker Framework's current flow analysis cannot always infer the best type when program points need to be merged during the traversal of the AST. Our framework strengthens the existing analysis by replacing the `Flow` implementation with a `CheckerAbstractValue` and `CheckerLattice` that represents the lattice of the existing sub-typing relation of the qualifier hierarchy that the checker uses. By explicitly mapping subtypes to abstract values and maintaining their relation in a lattice, our implementation makes it very easy to implement `join`. In addition, our implementation uses an `IterativeFixedPointSolver` which finds (if the lattices are not infinite height) the best possible abstract value at every program point. With these improvements, when the Checker Framework uses our DFF, the same example does not raise the error.

2.1.2 Aliasing: False Positives & Negatives

The Checker Framework does not have any implementation to handle aliasing. It uses `javax.lang.model.element.VariableElement` to represent the program variables internally. `VariableElement` works well in general; however, it

only defines one object for each field of a class: all instances of a class share the same `VariableElement`. This leads to an incorrect treatment of aliasing, which can cause false negatives.

Consider the following example:

```
class Alias {
    String field = null;
    void method() {
        Alias obj1 = new Alias();
        Alias obj2 = new Alias();
        /* After this line, all 'field'
        instances become @NonNull. */
        obj1.field = ``not-null";
        obj2.field.toString();
    }
}
```

When we tried to type-check this example with Nullness Checker (a checker specialized to find null dereferencing errors), it does not issue an error message. However, we know that `obj2.field.toString()` would throw a `NullPointerException`. Due to the fact that the flow inference keeps only one `VariableElement` to represent `Alias.field` for all instances of `Alias`, it is as if all the actual `Aliases` are aliases and all the `fields` are the same. This is a wrong assumption and ends up with a false negative.

We fix this issue by storing the global variables (i.e., fields) in a map from its global identifier to its `VariableElement`. Global identifiers are simple strings at this moment. In the above example, though `VariableElement`s are the same for both `field`s, their global identifiers are different: `obj1` and `obj2` respectively. So, they are represented as different elements which solves the aliasing problems. We also made sure that when there is a possibility of aliasing, the analysis would take this into account and effect all variables that have the same `VariableElement`. Thus, our aliasing analysis is complete, however it still have some limitations due to the representation of global identifiers. Simple strings cannot encode all the program semantics. A better analysis might represent the global identifiers as `VariableElements` too, which we leave as future work.

2.2 Value of the Checker Framework

The Checker Framework can be used for finding bugs or proving program properties in large scale projects. This process contains two phases: annotating the source code and running a checker to type-check the annotated code. In its current distribution, the Checker Framework includes more than ten checkers and on average, a checker takes 700 ($\sim$ 400 NBNC) lines of code. Moreover, Papi et al. showed that the Checker Framework can be used to annotate Daikon[2], Annotation File Utilities[3] and other open source projects. They ran different checkers that cover 600 K LoC and found 56 errors combined in these different open source projects during this process [7].

Though it might be easy to write a checker, or use the checkers included in the distribution of the Checker Framework, without a more precise DFA, it is not possible to leverage the Checker Framework's capabilities to the full extent. As mentioned proving program properties also needs the annotation of the client code, which is not difficult but does require extra effort by the programmer. Proving that Annotation File Utilities[4] does not dereference a `null` value, required writing more than 1000 annotations [7], though Nullness Checker uses a specialized flow inference to make better type inferences. As the number of annotations needed can greatly increase if there were no flow inference, it can also potentially decrease with a more precise DFA. Our framework aims to improve this aspect of the Checker Framework and decrease the total time needed for manually annotating these programs.

[2] `http://code.google.com/p/daikon/`

[3] `http://types.cs.washington.edu/annotation-file-utilities/`

3. TECHNICAL APPROACH

This section describes the design and implementation of the DFF, specifically: the interface it provides for performing analyses (3.1), definition of DFA (3.2), choice of program representation for the analysis (3.3), architecture (3.4), interface for describing DFAs (3.5), algorithmic details (3.6), and how the DFF fits into the Checker Framework (3.7).

3.1 Dataflow Framework Interface

Our framework presents a simple interface for users (e.g., tools like Checker Framework) to perform analysis on an AST and obtain the results. After specifying the DFA to perform, the user calls `DataflowFramework.scan` on the root of the program AST. This begins the analysis, doing intraprocedural analysis on every method declaration in the AST. To use the results of the analysis, there are test methods that return abstract values of variables of AST nodes and program points. `DataflowFramework.test` takes a tree from the program and returns the inferred abstract value of that tree. `DataflowFramework.testAfterProgramPoint` returns the abstract value of a variable after a given statement. The user can use the abstract values to perform checks or optimizations on the program. A constant propagation optimization could test an identifier to see if it is a constant, and if so, replace it with its constant value. A type-checker could ask for the inferred type of the right-hand-side of an assignment to see if the assignment is legal. This legality check could succeed even if the declared type of the right-hand-side (RHS) is not a subtype of the left-hand-side (LHS) as long as the refined type of the RHS is a subtype of the LHS.

3.2 Dataflow Analysis

A DFA involves abstract interpretation of the program. The state of the program at some point can be represented by an abstract store, containing mappings of variables to abstract values. Interpretation proceeds by applying the transfer function for a program statement or basic block to the current store [1, §2].

A DFA can be described with three components: transfer functions, a lattice, and a function α that maps concrete values into an abstract domain [4, §3]. There is a transfer function for each basic block which maps the input store of the block to the output store (i.e., the abstract values

[4] `http://types.cs.washington.edu/annotation-file-utilities/`

of variables before and after that basic block). The lattice is a partially ordered set with a unique least upper bound and unique greatest lower bound for every pair of elements. One way a lattice can be defined is with a join function that gives the least upper bound of two elements. In a DFA, the elements of the lattice are abstract values. The abstraction function α maps values in the concrete domain (e.g. primitive values) to values in an abstract domain, which is necessary for performing abstract interpretation on an AST. As examples: for constant propagation analysis $\alpha(5) = \{5\}$, and for nullness inference $\alpha(\text{“}foo\text{”}) = @\text{NonNull}$.

3.3 Analysis over the Control Flow Graph

A central design point is what type of internal representation to perform the DFA over. We choose to do analysis over an explicit control flow graph (CFG) data structure. The AST would have been a reasonable alternative because it already exists and thus there is no need to write a new structure, but it was not clear how to enable general DFA without requiring new DFAs to deal with control-flow. The AST method is used in the flow implementations in javac and the Checker Framework. However, the implementations are each written for a specific type of analysis, and the code is written with various assumptions; for example, nodes need not be visited more than twice (loops are processed in two passes: one for the entrance and another to propagate information down the back edges of the control graph).

DFA relies on knowledge of control-flow. We think that using a structure where control-flow is explicit makes the implementation of a fixed-point algorithm easier and clearer because it decouples performing the fixed-point computation from handling what particular control structures mean. Using an AST representation, the solver, in addition to its main task, would have to find the targets of jumps, like break statements or exceptions. In our approach, these structures are represented generally as control-flow edges in the CFG. Handling the control-flow details in a separate class makes reasoning and debugging easier; for example possible, to just look at the CFG to understand if control structures of the language are handled incorrectly. In addition, it is more flexible to separate how the CFG is built from how to perform DFA over it; if some analysis does not rely on certain control-flow edges, like those for exceptions, then a different CFG could be used with the same solver.

In our implementation of the CFG, each `BasicBlock` contains a list of statement trees from the AST. So, although our DFF internally performs the analyzes over the CFG, the interface that it provides for users to define DFAs (see 3.5) exposes the Java Compiler Tree API, which supports a visitor pattern over the AST. Thus there is no need to become familiar with a new program representation or API.

More specifically, the basic blocks contain sub-trees of the AST that do not include control structures, with the exception of expressions with control-flow: those using ternary operator and the binary && and || operators. Handling these in the DFA is easy and it avoids the need to mutate the AST or add nodes to the basic blocks that are not part of the original AST.

For expressions whose result affects the control flow, like the conditions of if-statements and loops, we use a subclass of `BasicBlock`. It has additional methods that return only the `BasicBlock`s that can be reached if the expression evaluates to true and false, respectively. This makes path-sensitive analyses possible which increases the precision.

3.4 Architecture

The DFF consists of the following classes: a top level `DataflowFramework`, a `CFGTransformer` to convert AST sub-trees into CFGs, a `Solver` for performing analysis, and a set of interfaces and abstract classes to define DFAs and customize the behavior of the analysis. The `DataflowFramework` provides the interface to the user of the analysis, handles adding global variables to the abstract store, and initiates the processing for each method of the AST. The `CFGTransformer` produces a CFG given a method AST. The `Solver` performs fixed-point computation over a CFG, and it is also capable of answering queries about flow inference values at program points. The abstract `DataflowAnalysis` provides basic functionality for implementing transfer functions of DFAs, particularly information propagation, traversal of expression sub-trees, and control flow within expressions for &&, ||, and conditional expressions.

The interactions between the parts of the DFF are illustrated in Figure 2. When the analysis starts by a call to `scan` (step 1), `DataflowFramework` visits all fields of the class to add them–and initial values, if given–to the abstract store. For each method tree, it uses `CFGTransformer` to create the CFG of the method (steps 2, 3). This CFG is passed to the `Solver` to initiate the analysis (step 4). The `Solver` drives the analysis, handling splitting and merging of control flow, while calling `DataflowAnalysis.interpretExpression` on various AST nodes to apply transfer functions to the store (steps 5, 6). The interface for implementation of DFAs is covered in section 3.5. While applying a transfer function, the `DataflowAnalysis` updates a mapping from AST expressions (such as identifiers) to inferred abstract values to keep track of the results of the analysis; this mapping is used to satisfy calls to the `DataflowFramework.test` method, mentioned in section 3.1. In addition, the `Solver` maintains a mapping of statement trees to the basic blocks in which they reside. Since the input store is kept for every basic block, the mapping allows the Solver to answer queries about variable values at program points.

3.5 DFA Descriptions

In our framework, a DFA is described by extending an abstract `DataflowAnalysis` class to implement abstract interpretation and lattice operations corresponding to the three components of a DFA (defined in 3.2). Below we describe how each component of a DFA is implemented. We use the intuitive running example of `ConstantPropagationDFA`, which describes a DFA for inferring constants in the program.

3.5.1 Transfer functions

Transfer functions are not explicitly defined for each basic block; rather, AST visit methods of `DataflowAnalysis` class and its subclasses define the abstract interpretation of statements. The program store is updated when assignments within the basic block are interpreted. To handle assignments, values are propagated from the RHS to the LHS,

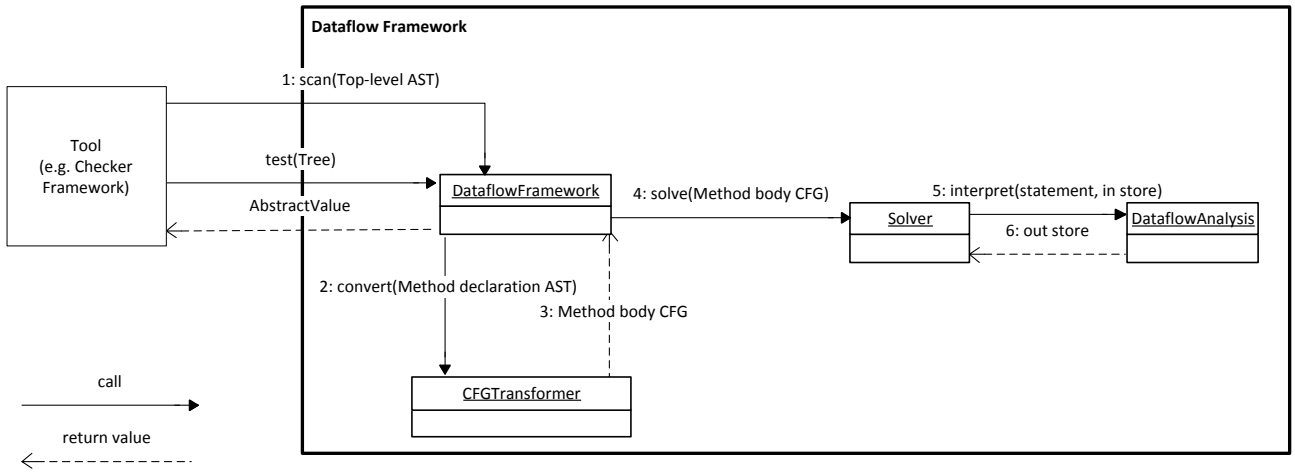

Figure 2: The high level interaction within the DFF. The tool using the DFF calls `scan` to begin flow analysis of a class. Steps 2,3,4 repeat for each method declaration of the AST. The Solver repeats steps 5,6 multiple times for every statement of each basic block in the CFG until a fixed-point is reached. After or during the `scan` process, a tool can `test` an AST node to get its inferred abstract value.

as specified by `DataflowAnalysis.propagate`. The `ConstantPropagationDFA` only requires the definition for interpreting operations, such as binary add, by overriding some interpretation methods. For example, `ConstantPropagationDFA.interpretBinary` is implemented to perform an operation on two abstract values that represent the constant or non-constant operands.

3.5.2 Lattice

The lattice is defined by together the representation of abstract values and a join method. Abstract values are represented in the analysis by defining a class that implements the `AbstractValue` interface. An `AbstractValue` implementation can be as simple as a single field whose value will correspond to the value in the lattice. For example, `ConstantPropagationAV` just contains the constant value itself, like 5 or "foo", and the `Top` of the lattice means `non-constant`. The join method, which defines the structure of the lattice, takes in two abstract values and returns their least upper bound. `ConstantPropagationLattice.join` returns the same abstract value if both inputs are equal (i.e., the same constant) or the `Top` otherwise.

3.5.3 Abstraction function

The function α that maps concrete values to abstract values is defined by implementing AST visit methods to interpret leaf tree values, such as those for literals, in the abstract domain. In `ConstantPropagationDFA`, the abstraction function is defined by implementing methods like `interpretLiteral`, which just returns an abstract value containing the literal value as a constant.

3.6 Solver Algorithm

In this section we present the iterative fixed point solver algorithm, as defined in [4, §5.3], that our framework uses to infer the best `AbstractValue` at program points (stores). Solver traverses the control flow graph representation of the method to be analyzed. It reaches a fixed point for the input store and the output store for each basic block in the control flow graph. To achieve this, the solver uses a worklist algorithm. We explain different parts of the solver algorithm in the following order: the preparation step that happens before the solving algorithm starts (3.6.1), the worklist algorithm for iterative fixed point solver (3.6.2), the modularity of the algorithm and some possible extensions (3.6.3).

3.6.1 Preparation

For the analysis of each method, `Solver.solve` is called with three arguments: control flow graph representation of the method, a DFA that will interpret the statement trees that comprise the `BasicBlock`s, and the global `Store`. Solver gets the entry point of the control flow graph, which is a basic block, and copies the global store to the input store of this basic block, which contains only the method parameters at this point. Note that during this copy of stores, we don't lose the information whether a variable is global or not since `Store`s keep track of these.

3.6.2 Worklist Algorithm

We present the worklist algorithm used by the solver in pseudocode in Figure 3. The algorithm starts by adding the entry point of the control flow graph to the worklist and continues until the worklist is empty. In each iteration, the top basic block is taken out of the worklist, its analysis is done (by `process`) and the input store of its successors are updated (by `updateSuccessor`).

`Process` analyzes the current basic block by calculating the output store of this basic block using its input store and contents. Since basic blocks don't contain control flow, the contents of the basic block is a combination of the following `Tree`s: `VariableTree`, `ExpressionTree`, `ExpressionStatementTree`. Each tree is analyzed by the current DFA, which

```
solve(cfg, dfa, gStore) {
    // do preparation step.
    worklist.add(cfg.getEntry());
    while (!worklist.isEmpty()) {
        block = worklist.remove();
        process(block);
        updateSuccessors(block);
    }
}
process(block) {
    store = block .getInputStore().copy();
    for(tree: block.getContents()) {
        dfa.interpretTree(tree, store);
    }
    block.setOutputStore(store);
}
updateSuccessors(block) {
    os = block.getOutputStore();
    ms = new Store();
    for (successor: block.getSucc()) {
        is = successor.getInputStore();
        changed = merge(os, is, ms, dfa);
        successor.setInputStore(ms);
        if (changed)
            worklist.add(successor);
    }
}
```

Figure 3: Worklist algorithm in pseudocode. `solve` is the main loop, performing the worklist processing. `process` applies the transfer function for each statement in a basic block to produce an output store. `updateSuccessors` updates the input stores of successor basic blocks and adds them to the worklist if the input stores have changed.

updates the copy of the input store. This copy (i.e., `store`) is updated by the DFA gradually, which creates the output store of the current basic block at the end of `process`.

`UpdateSuccessors` does the propagation of the stores (i.e., abstract value of the variables) along the paths of the control flow graph. For each successor of the current basic block, the output store of the current basic block and the input store of the basic block is merged. The merge store (i.e., `ms`) is set as the new input store of the same successor. For simplicity, the algorithm presented above assumes that the input store of the successor already exists. If it doesn't, (i.e., this is the first path to that successor) we simply set the output store of the current basic block as the input store of the successor.

`Merge` (omitted for simplicity) takes three stores and a DFA. For each variable that exist in both first two stores, the abstract values of this variable coming from two different stores are joined. Join operation is done using DFA and the same variable is added to the last store with the result of the join. If there is at least one change in the abstract values of the variables (i.e., the join value is different than the value coming from the second store), then `merge` returns true indicating that the fixed point is not reached for this basic block and it is added to the worklist.

3.6.3 Extension to Other Solvers

We provide an interface `ISolver`, an abstract implementation `Solver` and the actual implementation of the iterative fixed point solver `IterativeFixedPointSolver` in the solver package. Due to this design, it is very easy to extend our framework with other kind of solvers such as elimination solver [4, §5.4] and path algebra solver [4, §5.5]. A new concrete class needs to be defined for the new solver, which needs to implement only the `solve` method. We leave the implementation of other types of solvers as future work.

3.7 DFF in the Checker Framework

The DFF fits into the Checker Framework by providing functionality and abstraction that replaces the existing flow implementation, found in `Flow`. A default `DataflowAnalysis` for Checker Framework, `CheckerDFA`, implements functionality in `Flow` that was specific to DFAs for type checkers. More sophisticated flow sensitive inferences such as `NullnessDFA` are implemented by classes that extend `CheckerDFA`. `DataflowFramework` provides the same interface (section 3.1) as `Flow`, so the only change to the Checker Framework required was instantiating a `CheckerDFA` rather than a `Flow`.

The abstract values for the `CheckerDFA` are instances of `CheckerAV`, which just contains an `AnnotationMirror`, modeling of an annotation by `javax.lang.model.element`. The lattice is defined by the sub-type hierarchy built by the checker used in the analysis, and the `join` of two elements is their closest common ancestor in the hierarchy. The abstraction function α is essentially implemented using the Checker Framework's `AnnotatedTypeFactory`, which gives the annotated types for AST nodes.

Doing nullness checking requires more kinds of inference than for required for basic checkers, so a `NullnessDFA` that extends `CheckerDFA` is necessary. An example of more a more sophisticated inference is conditional inference on the branches, which is supported by our framework. It overrides the `DataflowAnalysis.interpretCondition` (called by the solver for `ConditionBasicBlocks`) in order to create a true branch store and false branch store. For example, for the condition $(x! = null)$, the entry store of the true branch will have $x = @NonNull$ and the entry store of the false branch will have $x = @Nullable$. We also borrow the method of inference on expressions, represented as strings, that Checker Framework's original NullnessFlow does, which allows expressions other than just identifiers, like pure functions with the same arguments, to have inferred values. When implementing the `NullnessDFA` we found that there is certain functionality it introduces, like inference on a wider class of expressions, that could be provided for more general class of DFAs. We currently have documentation in the code on where special inference functionality can be generalized, and we leave it to future work.

4. EVALUATION

Since our framework's integration with the Checker Framework is almost complete and one of our goals was to replace the flow inference of the Checker Framework with a DFA (`CheckerDFA`), we evaluate our framework by using the Checker Framework and existing checkers. In this section we present the results from running the test cases for the

Table 1: False positives and false negatives issued by the Checker Framework with old flow and with DFF.

Codebase	Checker	CF False Neg.	DFF CF False Neg.	CF False Pos.	DFF CF False Pos.	Ave. CF Compile Time	Ave. DFF CF Compile Time	DFF Overhead
AFU	Signature	0	0	30	30	27s	28s	1%
ASM	Signature	0	0	46	46	01m05s	01m12s	10.5%
CF Tests (basic)	Various	17	0	13	0	42s	44s	4%
CF Tests (nullness)	Nullness	0	13	0	55	22s	24s	9.9%

Checker Framework and from running Signature Checker on a completely annotated open source project. We show the usability of our framework by comparing the implementation of the old flow inference and the new `CheckerDFA`. Finally we evaluate the quality of our implementation by the documentation of our framework.

4.1 Performance on the test suite

We evaluate the DFF basic functionality by the number of tests it passes in the existing Checker Framework test suite, as well as the number of valid tests it passes that we added in the course of our work. We have written 25 Java code tests to test different concepts during the development. Since Nullness Checker uses more advanced type inference, and its integration with our framework is is still in progress, we present the tests written for Nullness Checker separately. We also exclude 36 tests from the original suite since they were written to test the Checker Framework itself and not a particular checker or the flow inference. The results are presented in Table 2.

Since the modified version of the Checker Framework passes all basic inference tests (which are implemented to test the different properties of at least 10 checkers), we conclude that our framework's integration with the Checker Framework is complete. Notice that the current version (which uses `Flow`) fails 11 of these tests. Five of these tests come from the existing test suite: `Flow` implementation wasn't able to handle annotations that have values (e.g., `@Fenum("A")`). We have fixed this problem in our implementation of `CheckerDFA` and the same tests pass in our version. The remaining six failing tests come from the new tests we have added; most fail due to the loss of information during the joins in control flow merges and not handling aliasing properly.

Currently our `NullnessDFA` when run on the nullness checker tests, produces a number of false negatives and false positives. False negatives show that our analysis is not yet safe, and they are due to bugs in the `NullnessDFA` when it does inference. An example is that our DFA cannot report a null dereference when a @Nullable value is dereferenced in a condition. This is related to the fact that after such a dereference, the variable is always considered @NonNull to reduce the number of reported errors later in the program. False positives are due to our DFA missing some inferences that refine variables to @NonNull the nullness tests expect to happen.

4.2 Checking real code

We evaluated the use of our framework in the Checker Framework in realistic situations to make sure that it can actually handle large projects and perform at least as well as Checker Framework on real code. To perform at least as well, it should find all errors and report no more false positives than the original. We ran the new framework on completed (i.e., annotated for a particular checker) case studies that run on different checkers. So far we have run Signature Checker on Annotation File Utilities (AFU) and we plan to run different checkers on different case studies in the future. AFU contains a modified version of the ASM[5] and has 70 K LoC (43 K NCNB). The completed case study contains 327 annotations. Annotations can be seen as documentation of the source code to give the Signature Checker more information about 'what to check' and extended types of some variables in various program points. We counted the number of false negatives and false positives reported. We also measured the running time to show that our framework does not impose a great deal of overhead over the `Flow` implementation. We report our results for AFU (with ASM) and complete test suite of the Checker Framework (to analyze the results in a different perspective) in Table 1.

Table 2: Success of our framework vs. Checker Framework (CF) for new tests we have written

	CF& DFF	CF & Flow
# of passed tests (basic)	163	152
# of failed tests (basic)	0	11
# of passed tests (nullness)	49	75
# of failed tests (nullness)	26	0

The results show that our DFA is better than the current `Flow` implementation (for inferences other than nullness) since it gives less false positives and false negatives over all. Our framework performs relatively bad on nullness tests since `NullnessDFA` is not completely implemented yet. The table also shows that, for the default flow analysis, our framework's integration is complete since modified version of the Checker Framework can match the expected output in AFU case study.

The Checker Framework is intended to be used every time code is compiled, and a high overhead would damage this goal. We wanted to make sure that running time of the Checker Framework using our framework is not significantly more than the original. For timing calculations, we have run the Checker Framework with our framework and with `Flow` implementation on AFU and the Checker Framework test suite 5 times and take the average of the compile (typecheck) times. Confidence intervals for this timing is given in Table 3. In addition, table 1 shows that the worst overhead is seen in ASM: a 10% increase in compile time over the original Checker Framework. The results suggest that our implementation, without optimizations, does not put much

[5] `http://asm.ow2.org/`

Table 3: Confidence intervals (CI, in seconds) for timing calculations (for confidence level 95%)

	CF& DFF CI	CF & Flow CI
# ASM	0.39	1.06
# ASM & AFU	1.85	0.75
# CF Tests (all)	0.88	0.42
# CF Tests (nullness)	0.77	0.36

overhead over the old implementation.

4.3 Usability

We designed our framework such that developers can easily and effectively implement DFAs. To accommodate this goal, all common behavior for DFF and DFAs is implemented in our framework and our framework provides a clean API over abstract methods that defines the behavior changes across the different DFAs. Defining a particular DFA would not even require the implementation of traversal methods of the AST, in fact the traversal methods in `DataflowAnalysis` are final. Instead of forcing the developer to write the whole traversal method, our framework provides the required information, such as the abstract values returned by the traversal of sub trees, in `abstract interpret*` methods so that many DFA could easily implement the transfer functions without the need to think about the whole traversal details. In addition to this, we also present the actual AST node in interpret method in case the DFA must traverse the original node itself.

To show this property, we analyze the number of lines of code (LoC) and non-comment non-blank lines of code (NCNB) of our core framework and the Checker Framework extensions. We define our core framework to be the general purpose implementation that can be used by any DFA such as `DataflowAnalysis` and `Solver`. The Checker Framework extensions are the classes that would replace the current flow sensitive analysis in the Checker Framework, written especially for the analysis it uses.

Our core framework contains 4230 LoC (2362 NCNB). The default inference implementation in the Checker Framework contains 764 LoC (443 NCNB). Our implementation completely eliminates two of the existing classes in the Checker Framework: `Flow` and `GenKillBits`, which contain 1134 LoC (659 NCNB). Notice that though the new implementation is more precise, it contains one third less code compared to the old implementation. Our nullness inference, which would replace `NullnessFlow`, contains 1122 LoC (703 NCNB), which is almost the same as the existing implementation. However, we strongly believe that most of the current behavior in `NullnessDFA` can be transferred to the `DataflowAnalysis` or be implemented more clearly and efficiently. Therefore, we believe that our framework will help developers to write DFAs for javac more efficiently and easily.

4.4 Quality of Implementation

A final important goal in building the DFF was the quality of the documentation, understandability, and ease of maintenance. We ensure that all declarations are properly documented in the Javadoc-specified manner and that all non-self-explanatory code has descriptive comments. Critical parts of the code should be understandable to other developers or at least be easily explainable, but we have not yet evaluated this. We leave this as future work. Quantitatively speaking, our core framework (i.e., the basic functionality such as `DataflowAnalysis` and `Solver`, etc.) has ∼1500 lines of comments. This is 35% of the implementation. Our Checker Framework extensions (i.e., `CheckerDFA` and `CheckerAV`, etc.) have ∼250 lines of comments, which is 34% of the implementation. As having a 1/3 documentation (Javadoc and normal comments about the implementation) overall, we believe that our framework will be easy to maintain, understand and extend in the future.

5. RELATED WORK

In this section we present some other literature on DFF and DFA for javac as well as for other Java compilers and we argue in what aspects our framework is different from them.

5.1 DFFs and DFAs for javac

Various research ([3], [5], [7]) has tried to implement DFFs for javac, the Java compiler. In most of these cases, instead of creating a completely general framework, researchers decide to integrate their DFF or DFA closely to their main product for reasons such as performance, fast implementation, and limited interest in the DFF itself. Because of this integration, most of these analyses or frameworks are not general enough to be reused by similar research. The integration also fuses the main program logic with the framework's logic which makes it very hard to maintain or enhance either parts. In this section, we present some of the other attempts to write a DFA or DFF for javac and argue that our approach is different.

The Checker Framework already does some type inference as introduced by Papi et al. in [7, §3.7]. The default inference handles all basic flow sensitive analysis needed by pluggable types, including flow of the inferred types of the variables during the traversal of the AST, propagation of the inferred type of the right hand side of an assignment to the left hand side, propagation of the inferred types in the different branches of a conditional to the end of the conditional, and a (not complete, see 2.1.1) join. This analysis is implemented as a generalization of javac's `Flow` implementation as discussed in Section 1.

Our DFF is different from the inference rules that already exist in the Checker Framework in 3 ways: It is more general, more abstract, and more precise.

The DFA in the Checker Framework only does analyses for pluggable types. Besides the ability to support pluggable type inference, our framework is able to handle other DFAs, such as constant propagation[6]. Moreover, the Checker Framework provides the inferred type information on certain AST nodes, whereas our framework also provides the inferred abstract value of the variables before and after each basic block in the converted program (see section 3.1). With these additions, our framework is more general.

[6]Our framework also includes a simple constant propagation analysis implementation that is tested on a sample code.

Current flow sensitive inference in the Checker Framework is embedded into the implementation very tightly. So, it is not possible, without considerable changes, to use this inference implementation somewhere other than the Checker Framework itself. On the other hand, our framework minimizes this coupling by abstracting out the current `Flow` implementation and providing a clean and usable API for `Flow` inference users. All matters specific to annotated type-checking are implemented in a user DFA extension class. Other than `TreeUtils`, which is an internal helper class that provides different operations on `Tree`s, our framework does not use any construct from the Checker Framework implementation. This coupling can also be removed by implementing the same functionality offered by `TreeUtils` in our framework. We leave this as future work.

`Flow` inference in the Checker Framework uses a bitwise representation of abstract values for variables, so it loses precision when results of sub-analyses from different control flow paths are joined in a program point. A correct implementation of `merge` and `split` over bits is theoretically possible, though it is very hard to implement due to the fact that the implementation does not have an actual representation of the abstract values and their relations. Our framework contains a complete implementation of abstract values, (finite) lattices, and the default DFA is fully integrated with this representation. Details about this problem and our approach are already given in section 2.1, so we skip further discussion here.

JavaCOP[7] is another pluggable type system that can do flow sensitive type analysis. Its DFF is introduced by Markstrum et al. in [3, §4]. The paper includes very little implementation detail about the DFF. The DFF provides an interface, `FlowFacts`, which is very similar to the type inference results stored as a map in the Checker Framework, but it provides an additional level of abstraction. Similarly it does a GEN/KILL analysis: the concrete class implementing `FlowFacts` overrides the generate and kill methods. Generate methods (i.e., `FlowFacts.genSet`) supply information that a new type inference is valid at that program point and kill methods (i.e., `FlowFacts.killSet`) supply information that a type inference, existing or not, is not valid anymore. At program join points, these GEN/KILL information from different branches are merged (by using `FlowFacts.meetWith`). The checkers that need flow sensitive analysis can implement `FlowFacts` to implement their analysis in JavaCOP.

The paper also argues that their DFF can handle every possible control situation in a Java program analysis since it is an extension of the flow analysis in javac, which is very well tested, maintained and optimized. We believe that this is both true and wrong. It is true in the sense that the flow sensitive analysis implemented in javac (definitive assignment, undefinitive assignment, liveness and exception analyses) are complete and highly tested. However, it is a misconception that these analyses can be generalized directly to pluggable type-checking flow sensitive analysis. The analysis included in javac are highly optimized for their cases, and we have already shown that a trivial direct generalization of these analyses does not work for pluggable type-checking. JavaCOP paper does not explain how and to what extend the analyses in javac are generalized to their framework.

Both the Checker Framework's current flow sensitive analysis and JavaCOP's DFF are generalizations of the already existing Flow in javac for pluggable type-checking. Also, they both offer a minimal interface. Due to this similarity, our framework differs from the JavaCOP flow framework in a similar way: it is not limited to pluggable type-checking analysis, rather this analysis is one of the many kinds of analyses handled and offered by our framework. It also provides a more explicit and usable interface and some functionality classes for different kind of analyses.

jDFA [5] is a "Java framework for data-flow based program analysis". The paper states that this DFF allows users to "easily implement specific analyses, test their correctness, and evaluate their performance". Besides the DFAs on general graphs, it supports the definition of intra-procedural analyses for JVM code on a high level of abstraction. Like our framework, jDFA is also implemented in Java and makes extensive use of the interface concept. This makes it, for example, possible to use different implementations of abstract domains. The major difference between our framework and jDFA is that our framework works on the Control Flow Graph (i.e. a representation of the source code) while jDFA works on JVM code (i.e. a representation of the program in a later stage of the compilation process) or on general graphs. There is however no direct way to perform a DFA on a source code representation of the program in jDFA.

5.2 DFFs and DFAs for other Java compilers

Other projects, often involving extensible Java compilers rather than javac, have enabled user implementation of intraprocedural DFAs.

[6] adds a control-flow and DFA extension to the JastAdd Extensible Java Compiler. It defines the extension declaratively using support for reference attributes provided by JastAdd. Reference attributes of AST nodes are references to other nodes, used to superimpose different kinds of graphs over the AST. The control flow graph is described by specifying a successor function for each statement type. A class of DFAs are built atop the control flow graphs by declaring equations that use union and intersection to build (1) value sets of variables (analogous to our transfer functions that transform the store) and (2) "in/out" sets of basic blocks in terms of the value sets (analogous to merges of stores done in our solver). JastAdd's circular attribute is used to automatically provide fix-point computations. The declarative implementation of this control-flow and DFA extension is relatively concise, an advantage enabled by using the JastAdd platform. The declarative descriptions of new analyses themselves can also be concise and can correspond closely to analysis formulations in the literature. However, there are possible tradeoffs in flexibility not addressed in the paper: they only demonstrate binary analyses, and so it is not clear how simple it would be to support lattices of a parameterized number of elements, which would be required to support a type checker framework, or lattices of infinite width (like that for constant propagation), since their method relies on explicit sets for abstract values. Our framework supports

[7] `http://javacop.sourceforge.net/`

both of these qualities in a straight-forward way by allowing any lattice to be defined using abstract values and a join method.

The Java Compiler Kit (JKit) [8] is an extensible Java compiler that includes support for defining DFAs. Their `dfa` package includes an interface `FlowSet` for defining a store–and implicitly, the lattice–and simply includes a method to clone the store for control splits and a method to join stores. It provides abstract classes for forward and backward analyses, which are analogous to the functionality in our abstract `DataflowAnalysis` and solver together. These classes provide abstract `transfer` methods for the user to define the abstract interpretation of their analysis. This framework has the advantage of presenting a very distilled and clean interface. However, it seems there is no indication of condition expressions, meaning that users building their own DFAs must sacrifice inference on conditions or else take on the burden of dealing with control flow statements themselves. Mocha, an extension of Java built on JKit, performs type inference with DFA in order to relax the requirement of static type declarations. In [2, §3.2.2] it seems that the Mocha implementation must handle control statements itself to be able to do its conditional inference. Unlike the `dfa` package in JKit, our DFF provides users an interface for doing inference on conditions without the burden of handling control flow.

Soot [9], a mature Java Optimization Framework for analyzing and optimizing bytecode, has a DFF very similar to ours. The intraprocedural analysis is done over the CFG methods, and control-flow is handled by the framework. Like our framework, there are interfaces for implementing stores with join and clone operations, a method interface for implementing transfer functions of statement "Units", and it supports analyses that infer different stores in branches. Since the DFF is part of Soot, users can use the same optimization framework for both the analysis and the resulting optimizations or checks. The Soot DFF differs from ours in that it works with Soot rather than javac, it is slightly more feature rich, and transfer functions are applied over bytecode units rather than source-level AST nodes. The advantage of our DFF operating over the source level is that users might be more familiar with language constructs than bytecode. However, the advantage of the DFF being part of a bytecode framework is that the analyses can be applied to any legal Java class file created by any compiler[8]. Our DFF provides more basic functionality, such as value propagation, for DFAs; however, conceivably, implementations of basic classes of DFAs could be distributed with Soot.

6. CONCLUSION

This paper described a DFF for Checker Framework. Our framework increases the precision of the extended type-checking of the checkers and improves the extensibility of the Checker Framework. It also decouples the core implementation of the DFAs and AST traversal from the Checker Framework. In addition, we believe that our framework is ready to be used in other projects with minor modifications. Thus, we provide a well documented and highly tested (by using the checker in real programs) DFF that can be used in general. In summary, we believe that this paper makes the following contributions:

- it improves the precision and extensibility of an open source product that is used widely.
- it provides a DFF implementation for javac that can be used by any project that needs to define a DFA over Java programs using the compiler tree API

[8]Technically, our framework is not restricted to javac, but it uses the Java Compiler Tree API for ASTs, which is not an enforced Standard for all Java compilers. It currently has some utilities that depend on the JDK implementation of the Java Compiler Tree API.

7. REFERENCES

[1] N. D. Jones and F. Nielson. Handbook of logic in computer science (vol. 4). chapter Abstract interpretation: a semantics-based tool for program analysis, pages 527–636. Oxford University Press, Oxford, UK, 1995.

[2] C. J. Male. Mocha: Type inference for java. Master's thesis, Victoria University of Wellington, 2009.

[3] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. Javacop: Declarative pluggable types for java. *ACM Trans. Program. Lang. Syst.*, 32:4:1–4:37, February 2010.

[4] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Inf.*, 28:121–163, December 1990.

[5] M. Mohnen. An open framework for data-flow analysis in java: extended abstract. In *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, PPPJ '02/IRE '02, pages 157–161, Maynooth, County Kildare, Ireland, Ireland, 2002. National University of Ireland.

[6] E. Nilsson-Nyman, G. Hedin, E. Magnusson, and T. Ekman. Declarative intraprocedural flow analysis of java source code. *Electron. Notes Theor. Comput. Sci.*, 238:155–171, October 2009.

[7] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[8] D. J. Pearce. The java compiler kit (jkit). http://homepages.ecs.vuw.ac.nz/ djp/jkit, September 2010.

[9] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.