A framework for dataflow analyses for the While language in Java

Written by:
Adrian Brink - adbr@itu.dk
Andrea Bitzili - abit@itu.dk
Date: 06/02/2017

Table of Contents

Abstract

In this paper we describe our framework, which enables the user to create any dataflow analysis for the While language. In order to enable build such analyses the user only has to implement to interfaces, one to provide the transfer functions and the other to provide the applicable lattice. Furthermore, a user can choose between the three different algorithms for finding the fixed-point.
During the process, we learnt that choosing Java as a host language for a dataflow analysis framework is not an ideal choice as working with lattices over a control-flow graph is best done in an immutable fashion. Due to this, we would recommend to use Scala or Rust next time. Furthermore, finding the right level of abstraction is one of the main challenges. However, this can easily be overcome by implementing one easy and concrete analysis, such as sign-analysis while keeping the overall architecture in mind. This provides ample room to experiment with different levels of abstraction while also being close enough to the problem to not get lost.
Lastly, we learnt that if you have to write bloated classes it means that you have not found the correct level of abstraction and are heading down the wrong path. Our entire framework is implemented in less than 500 lines of code and the code becomes easier as you start lifting the abstraction.

"It is a great learning experience to implement your own analysis framework and will be widely applicable to real life languages."

Introduction

In this paper we will explain how we implemented a generic analysis framework for dataflow analyses for the While language (source language) in Java (host language).
Our framework allows users to easily implement any dataflow analysis for the While language by implementing two core interfaces, which will be explained in-depth later. Moreover, it comes pre-built with Sign- and Liveness-Analysis. It works for any arbitrary program of the While language as long as it can be parsed using the included parser.

Most importantly, currently to see the output of the program one has to run it in debug mode in IntelliJ with a break-point on the last print statement.

In the remainder of the paper, we will discuss the design and implementation of the framework, followed by two concrete examples usages. From there we will explain how to implement any dataflow analysis and will finish by highlighting some future work and improvements for this framework.

Design and implementation of the framework

We build this framework with the intention of using it for any dataflow analysis for the While language. It was implemented in tandem with the Sign-analysis that uses this framework. This approach allowed us to remain focused on the goal while still keeping the overall architecture of the framework in mind.
The first noteworthy package is the CFG package. It contains all components necessary to build a CFG and its accompanying CFGNodes as well as the CFGState which is used to track the execution of an analysis. Constructing the CFG itself is not majorly complicated. Care has to be taken to lift the abstraction early. For example, one can easily implement each CFGNode as a sub-type of some abstract class in order to model the different language constructs and treat them differently. However, it is much cleaner to abstract away from that and treat all nodes uniformly is containing just one statement or expression as well as their predecessors and successors in the form of lists. To unroll the list of statements from the parser into a complete CFG one has to write a recursive function that allows for arbitrary nesting of language structures, such as a while-loop inside a while-loop inside an if-statement. The biggest hurdle here is the implementation detail that such a recursive descent always has to return the last node in the unrolled statement, since otherwise the CFG will be malformed. Since this framework does not rely on special confluence nodes the least-upper bound operation during the execution happens on in- and out-lattices. These are stored within a CFGState object. The reason for this is that everything related to the execution of the analysis should not be stored on the CFGNode itself as to not unnecessarily pollute that class.

The second interesting package is the Engine package. It contains a simple interface that all engines have to conform to. This allows the easy swapping of concrete implementations. Furthermore, it contains implementations of the three different algorithms to solve fixed-point problems. Generally they all use a very similar approach. The three basic operations of all engines are to run the transfer functions, to propagate the lattice updates and to compare the lattices for equality. The fixed-point checking happens through the setting of a boolean flag whenever a lattice update causes the lattice of its successor/predecessor nodes to change, since it means that the following transfer functions might yield a different result. The naive engine runs all transfer functions for all nodes, followed by all lattice updates for all nodes and then checks the boolean flag for whether a fixed-point is reached or not. The chaotic engine runs the lattice update for one node, then the transfer function for that node, for all nodes in the CFG. Lastly it checks the fixed-point flag. This different approach means that changes to the lattices are propagated faster and hence the chaotic engines performs better. Lastly the worklist engine starts by enqueuing all CFGNodes and then pops one node of the queue and processes the lattices updates as well as the transfer functions for that node. If the transfer function caused a change, it also enqueues all the nodes that are dependent on it, such as the next nodes in a forward analysis. Most textbooks and lecture notes claim that one only has to enqueue the first node in the program, however this doesn't work for the While language. This is, because an Output node could be the first node and while running Sign-Analysis over the first node, the transfer function would not update the lattice and hence none of the following nodes would get enqueued.

The next Lattice and the Framework package in this framework are not majorly interesting as they only contain the interfaces, which the user has to implement. The ILattice interface represents a lattice and the user has to decide on how to represent that lattice. A computationally very efficient solution is a concrete lattice that is backed by a bitset. The user is only

required to implement the join() method, which represents the least-upper bound operation of two lattices. The other required method is isEquals() which is used for checking for a fixed-point.
The IAnalysisFramework interface contains three functions. The user has to implement whether the analysis is forward or backward by implementing isBackward() and has to specify the initial element for the appropriate lattice in getInitialLattice(). Most importantly the user has to define transferFunction() which needs to return a function from the user-defined lattice to the user-defined lattice for every language construct. Special care has to be taken to ensure that these transfer functions are monotonic to ensure the termination of the analysis.

An interesting statistic for this framework is that it is implemented in 752 lines of Java of which around 200 lines could be optimised away and 100 lines are actual implementation of the Sign- and Liveness-Analysis. However in these 452 lines of code lies the power to implement any dataflow analysis for the While language. This figure will be similar for major programming languages.

Concrete example usages of the framework

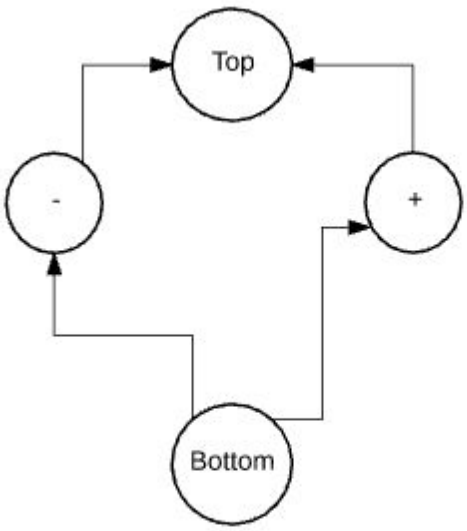
In the following two sections we demonstrate *Sign* and *Liveness analysis,* for programs written in the *while* language, using the aforementioned framework. Initially, we carry out *Sign* analyses by providing the corresponding implementations of *ILattice* and *IAnalysisFramework* interfaces. Furthermore, we show that the framework produces the expected results. Likewise, the same process is followed for the *Liveness analysis.*

Sign analysis

Sign analysis aims to determine the sign (+, -) for all expressions in a given program. Sign analysis is an example of forward-must analysis.

Lattice Implementation

The lattice we used in this case is:



The class representing the sign Lattice is:

```
public class SignLattice implements ILattice<SignLattice> {
        public BitSet element;

        public SignLattice join(SignLattice that) {
          BitSet oldBitSet = this.element;
          BitSet newBitSet = (BitSet) oldBitSet.clone();
          newBitSet.or(that.element);
          SignLattice signLattice = new SignLattice();
          signLattice.element = newBitSet;
          return signLattice;
}

        @Override
        public boolean isEquals(SignLattice that) {
          return element.equals(that.element);
        }
}
```

The lattice elements are represented by *java.util.BitSet* instances. In more detail, the *bottom, -, +, top* are encoded to *00, 01, 10, 11* BitSet instances.

The *join()* method defines how we combine information in case of confluence program point. The crux of the method is the bitwise **OR** operation. It is this operation that computes the result of a confluence. In more detail, the result of combining two lattice elements, is the value returned by the bitwise **OR** of these elements. For instance, let flows *f1, f2* having 01 (-) and 10 (+) lattice elements respectively and pointing to the same program point *p3*. The resulting lattice element inserting *f3* in *p3* is: 01 **OR** 10 which results to 11 (top).

Apart from the *Sign* lattice implementation, an Environment lattice is also needed for tracking the lattice values of each variable in the program. Its class representation is:

```
public class EnvironmentLattice implements ILattice<EnvironmentLattice> {
        public Map<String, SignLattice> environment;

        public EnvironmentLattice(Set<String> programVariables) {
          Map<String, SignLattice> map = new HashMap();
```

```
  for (String parameter : programVariables) {
          BitSet bitSet = new BitSet();
          SignLattice lattice = new SignLattice();
          lattice.element = bitSet;
          map.put(parameter, lattice);
        }
  this.environment = map;
}
private EnvironmentLattice() {}

@Override
public EnvironmentLattice join(EnvironmentLattice that) {
  EnvironmentLattice newLattice = new EnvironmentLattice();
  Map<String, SignLattice> newEnvironment = new HashMap();
  newLattice.environment = newEnvironment;

  for (String key : this.environment.keySet()) {
          SignLattice thatValue = that.environment.get(key);
          SignLattice newA = environment.get(key).join(thatValue);
          newEnvironment.put(key,newA);
  }
  return newLattice;
}

@Override
public boolean isEquals(EnvironmentLattice that) {
  for (String k : environment.keySet())
        if (!environment.get(k).isEquals(that.environment.get(k)))
          return false;
   return true;
 }
 }
```

The result of combining two Environment lattice elements is a new EnvironmentLattice instance with every variable pointing to the value computed by joining the two previous values of that variable. For instance, let *env1, env2* two Environment lattice elements having a variable *x* pointing to 01 in *env1* and 00 in *env2*. The result of combining these two lattice elements is a new element that now variable *x* points to 01 (01 **OR** 00 = 01). This means that the join() of the environment lattice is just an elementwise application of the sign lattice's join method.

Transfer Functions Implementation

A transfer function defines how every language construct affects a given lattice element. Transfer functions are represented by the value returned by the method transferFunction defined in *IAnalysisFramework* interface. The transfer function for the sign analysis is represented by the java class:

```
class SignAnalysis implements IAnalysisFramework<EnvironmentLattice>.
```

We do not provide the whole class body due to its size. However, the method, declaration and definition, that returns the transfer function for a given language construct (cfgNode) is:

```
public Function<EnvironmentLattice, EnvironmentLattice> transferFunction(CFGNode
cfgNode) {
   AST statementOrExpression = cfgNode.getStatementOrExpression();
   if (statementOrExpression instanceof Assignment) {
     String variable = ((Assignment) statementOrExpression).x;
     Expression expression = ((Assignment) statementOrExpression).e;

     if ((expression instanceof Expression) && (!(expression instanceof
BoolExpression))) {
       return (EnvironmentLattice one) -> {
         BitSet bitSet = SignAnalysis.eval(expression, one);
         EnvironmentLattice two = one.join(one);
         SignLattice lattice = new SignLattice();
         lattice.element = bitSet;
         two.environment.put(variable, lattice);
         return two;
        };
      }
    }
    return (EnvironmentLattice one) -> {
      EnvironmentLattice two = one.join(one);
        return two;
    };
  }
```

The value returned by the transferFunction(CFGNode cfgNode) is a lambda that takes an environment lattice element, and returns a new element with changes done according to the given CFGNode. As the method definition shows, the assignment statement is the language

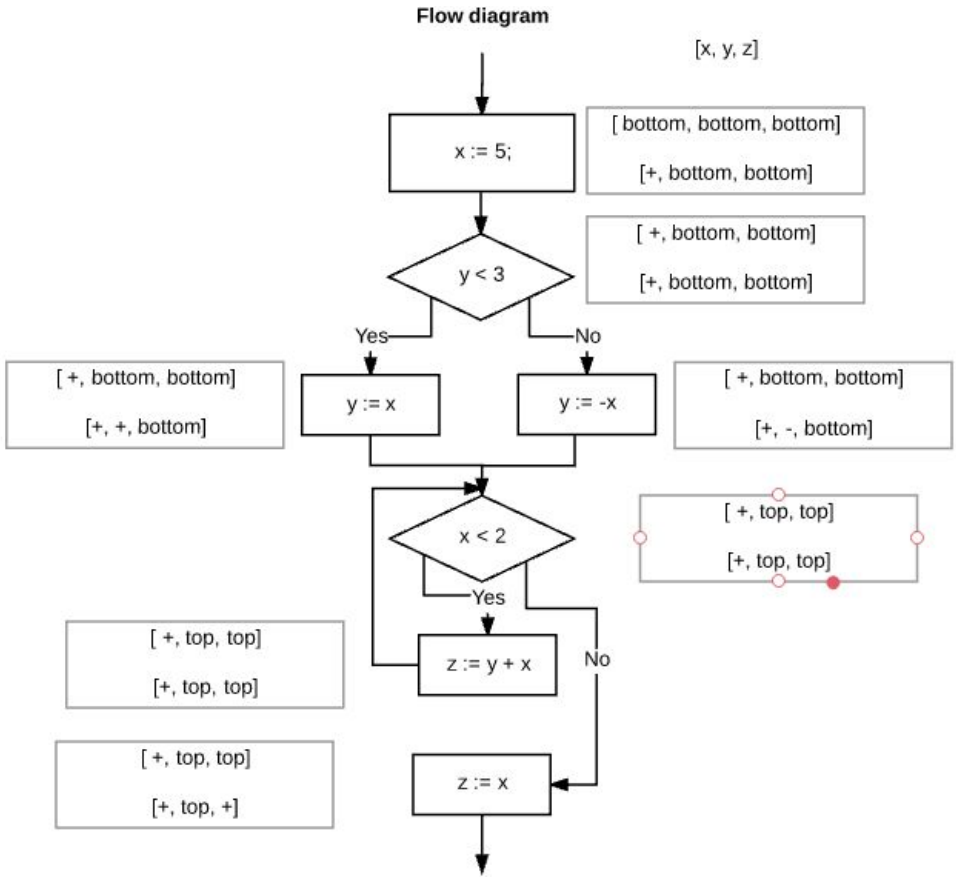
construct that can change the value of an incoming lattice element. For instance, let *x := 1 + 3* an assignment in the program and an incoming environment lattice element *el* having *x* pointing to bottom (00). The resulting lattice element is a new Environment lattice with *x* pointing to + (00 join (01) = 01). In case of any other construct we just return a lattice element equal to the incoming one.

Example Run

In this section we carry out a sign analysis for the program:

```
x := 5
if (y < 3) { y := x } else { y := -x }
while (x < 2) { z := y + x }
z := x
```

The control flow that represents the program and the fixed point computed by the analysis is:



It can observed that in $x < 2$ test, the value of x is positive making the body in *while* dead code that can be easily deleted without changing the semantics of the program.

Liveness analysis

Liveness analysis aims to approximate the set of variables that may be read during the remaining execution of the program. Liveness analysis is an example of backwards - may analysis.

Lattice Implementation

The lattice used for liveness analysis is parameterized with the given program which is a power lattice, where all the elements are the variables in the program.
The java class representing the Liveness analysis lattice is:

```
public class LivenessLattice implements ILattice<LivenessLattice> {
  public Set<String> elements =  Collections.emptySet();

  public LivenessLattice(Set<String> elem) {
    elements = elem;
  }
  @Override
  public LivenessLattice join(LivenessLattice that) {
    Set<String> res = new HashSet<>();
    res.addAll(elements);
    res.addAll(that.elements);
    return new LivenessLattice(res);
  }

   @Override
   public boolean isEquals(LivenessLattice that) {
     return elements.equals(that.elements);
   }
 }
```

A lattice element in liveness analysis is represented by a java.util.Set<String> instance holding references to the name of the variables that may be read in the remaining program.

Thus, the result of combining two lattice elements is simply a new element with the concatenation of the two former lattice elements, as shown in the body of the join() method.

Transfer Functions Implementation

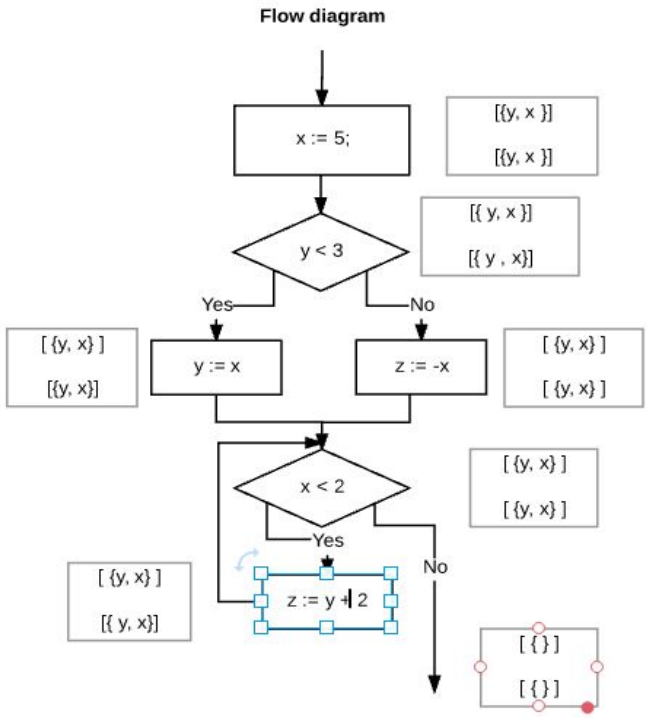
In this section we define how each language construct affects a given element in Liveness lattice.
The transfer function for each given program point is represented by the lambda returned by the *transferFunction(CFGNode)* method. We do not show the concrete implementation of the class and method, that returns the lambda, due to its size. However, the intuition is to check if there are expression values that rely on variables. If it is so, we add the variable to the existing lattice element, otherwise we just pass the existing lattice element. In more detail, let *l1* a *liveness lattice* element having empty set of variables, also let x := y + 1 the statement in a program point which the *l1* points to. The resulting lattice element *l2* leaving the program point is a new lattice element with a set containing reference to variable *y*, since the expression *y + 1* is relied on variable *y*.

Example Run

In this section we conduct liveness analysis for the program:

```
x := 5
if (y < 3) { y := x } else { z := -x }
while (x < 2) { z := y + x }
```

The control flow representing the program and the fixed point computed by the analysis is:



It can be observed that the value of variable *z* is never read. Thus, we can delete all the assignments to the variable *z* without changing the semantics of the program.

Explanation of how to implement any other dataflow analysis using the framework

As a user of this library it is very straightforward to implement another analysis.

For example, when tasked with implementing an Available Expression Analysis for the While language, the user only has to provide the source code and implement two interfaces.

The user first needs to implement the ILattice interface, where he can choose the underlying data structure to his liking. The only important aspect is that the join method is correctly implemented and hence can be

used by the analysis. The second vital part is the implementation of the IAnalysisFramework. Its main job is to provide a transfer function for every possible type of CFGNode. Here special care should be taken in that the implemented function needs to be monotonic. If a user provides a non-monotonic implementation for one of the CFGNodes this framework will fail to halt and will instead loop forever.
From here, the user only has to give the custom analysis and lattice to the library and run it.

Future work

In the future we hope to develop this framework further. One current disadvantage is that the engines currently have to know the direction of the analysis. This leads to a lot of duplicate code in the Engine package, since it currently contains almost 12 methods (run transfer and run lattice update methods) that are almost identical. In the next version of this framework, we will lift the abstraction level of the engine so that all three different algorithms share the same methods for running and updating. Furthermore we will ensure that the engines do not have to be concerned by the direction of the analysis by instead reversing the pointers between the CFGNodes appropriately before running the analysis.
Moreover, we hope to enable better integration with other tools, by defining clearer boundaries between the parser and the framework and clear interfaces to govern that boundary. Furthermore, a clean API to consume the results of the framework would be nice and hence that will get developed too.

References

Brabrand, C. (2015). *While_v1 language parser*.

Christophersen, M. (2016). *While_v2 language parser*.

Aho, A., Sethi, R. and Ullman, J. (1986). *Compilers, principles, techniques, and tools*. 1st ed. Reading, Mass.: Addison-Wesley Pub. Co.

Muchnick, S. (1997). *Advanced compiler design and implementation*. 1st ed. San Francisco, Calif.: Morgan Kaufmann Publishers.