Automatic break insertion

Author: Stef van Schie
Supervisor: Atze van der Ploeg

July 2022

Abstract

In this paper we present a novel loop optimization technique called "automatic break insertion", which can automatically determine points in a loop where the loop can safely be halted. We provide a compiler for a simple Java-like language which contains this optimization.

1 Introduction

Consider the following code snippet:

```
int[] array;
boolean found = false;

for (int index = 0; index < array.length; index++) {
    if (array[index] > 5) {
        found = true;
    }
}
```

Listing 1: A code snippet which searches for a number greater than 5.

This code snippet searches for the existence of a number in the array that is greater than five. However, this code is sub-optimal, since it may iterate more often than necessary. Once a number greater than five is found, any further iterations of the loop will not produce any change in the program state. A solution to this would be to add a break statement as the last line in the **if** block.

In this paper we aim to solve this issue by providing a static code analysis that can automatically analyse where such break statements are appropriate. To validate this analysis, we implement it by writing a compiler for a small, Java-like language and add this analysis as an optimization to this compiler. We will also link the techniques that are used for this analysis to analyses used in other code transformations. We demonstrate the effectiveness of this analysis on several tailor-made examples. We hope that this transformation gives significant gains in execution speed on realistic code, however we leave the investigation of this as future work.

We will first present a set of examples of code where we wish for automatic break insertion to take effect. Then we provide an overview of how the analysis works and how it can detect potential candidates for automatic break insertion. We then relate this analysis to other optimizations. Next we describe our compiler for the Java-like language and how the previously described analysis is implemented. Lastly, we present issues and questions which we leave open as future work.

2 Examples

A particular class of code for which automatic break insertion is of interest, is code which asks an existence or for-all question. Listing 1 is an example of an existence question. In particular, given an array of integers, it asks whether there exist a number greater than 5. This class of problems is of interest, because we can stop early. If the code acts as an existence quantifier, the search for a particular element can be halted if that element has been found. If the code acts as a for-all quantifier, the search can stop early as soon as one element contradicts the condition. These pieces of code are for this reason prime candidates for automatic break insertion. We will show some more examples where automatic break insertion can provide an improvement in the execution time of the code.

2.1 Primality testing

```
int n;
boolean isPrime = true;

for (int divisor = 2; divisor < n / 2; divisor++) {
    if (n % divisor == 0) {
        isPrime = false;
    }
}
```

Listing 2: A code snippet which checks if a number is prime.

Listing 2 shows code which checks if a number is prime. Of particular note is the existence question being asked: does there exist a number in $\{2, ..., n-1\}$ that is a divisor of n? This is another candidate for automatic break insertion, where the loop can be halted prematurely if we find a divisor of n. As soon as we have found one divisor, we know that n is not prime and do not need to look further for more divisors.

2.2 Palindromes

```
int n;
int factor = 1;

while (n >= factor * 10) {
    factor *= 10;
}

boolean isPalindrome = true;

for (int divisor = 1; divisor < factor; divisor *= 10) {
    if (n / factor % 10 != n / divisor % 10) {
        isPalindrome = false;
    }

    factor /= 10;
}
```

Listing 3: A code snippet which checks if a number is a palindrome.

Listing 3 shows a code snippet which checks if a number's decimal representation is a palindrome, such that the digits read the same from left to right as from right to left. A for-all question is asked, namely whether all the digits on the left side of the number equal the corresponding digit on the right side of the number. Automatic break insertion can optimize this code's execution time by halting the loop the moment one pair of digits is not equal. Once there is a mismatch between any pair of digits, the number can never be a palindrome.

2.3 Increasing digits

```
int n;
int highest = 9;

boolean isIncreasing = true;

while (n != 0) {
  if (n % 10 > highest) {
    isIncreasing = false;
  }

  highest = n % 10;
  n = n / 10;
}
```

Listing 4: A code snippet which checks if a number's digits are increasing.

Listing 4 checks whether all the digits of a number in its decimal representation are increasing. That is, a digit should never be preceded by a digit that is larger than itself. Another for-all question is asked: are all of the digits larger or equal to all of the digits before it? If a digit is found which is smaller than a preceding digit, then the loop can be halted. Automatic break insertion should be able to determine that this is the case and automatically apply this optimization.

3 Analysis

To implement automatic break insertion, we must first analyze the code we wish to apply this optimization to. The goal of our analysis is to find places where we can place a break statement, which does not alter the program's behaviour. The semantics of a program without this optimization should be the same as that of a program with this optimization applied. The analysis finds such places by looking at how different statements alter the state of the program. The break statement is then only inserted at a position where we can be certain that no future iterations of the loop would change the program's state. Because these iterations would no longer change the program's state we can halt the loop.

Our analysis consists of several steps which will detect the loops for which this optimization is possible and will also state where the break statement should be placed in the code. The following sections outline these steps. The order of the steps as presented here is a possible sequential sequence of steps to be applied, however some parts of the analysis may be executed in a different order or in parallel. Figure 1 below shows the possible ordering among the steps. In particular, from a step, any of the connected steps may be executed in any order, or even in parallel. A step may only be executed if all incoming steps already have been executed.

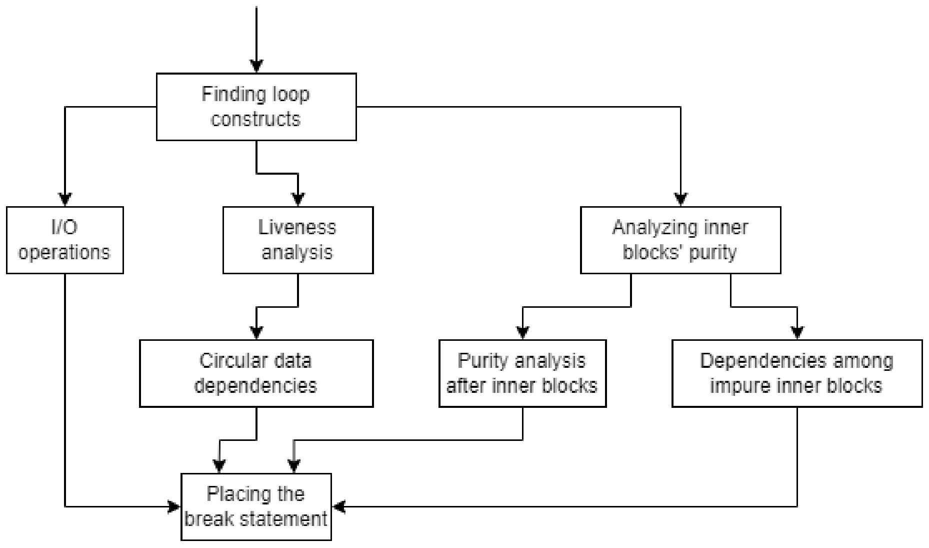
Finding loop constructs
I/O operations
Liveness analysis
Analyzing inner blocks' purity
Circular data dependencies
Purity analysis after inner blocks
Dependencies among impure inner blocks
Placing the break statement

Figure 1: A graph showing the possible orderings among the steps.

3.1 Finding loop constructs

To start analyzing which loops are eligible for automatic break insertion, we must first find all the loop constructs in the program. When talking about loops, we only consider natural loops. Natural loops are loops that have a single entry point and for which each iteration starts at this single entry point. A loop that is not natural, would for example be a loop where a **goto** construct may cause a jump to inside the loop. Loops which are not natural are not considered for automatic break insertion. All natural loops may be considered for automatic break insertion. The following steps will act upon a single natural loop. If the program contains multiple natural loops, the following steps will have to be applied for each loop independently. We will make the assumption that for each loop break insertion can be safely implemented, until we find evidence to the contrary in the following steps of the analysis.

3.2 I/O operations

When a break is inserted into a loop, some iterations might be skipped that would otherwise be executed. This is not an issue as long as the iterations that would be skipped do not change the behaviour of the program. In later sections, we will analyse in detail how modifications of variables within a loop may affect our ability to do automatic break insertion. In this section we will do a preliminary check, concerning I/O operations. I/O operations are operations that interact with the computer in some way beyond the program's internal state. This can be reading and writing files, sending and receiving data over the network, or interacting with a user via computer peripherals like a monitor, speakers, a camera, etc. These operations might not incur a state change for the program itself, but might cause changes in behaviour for the devices they interact with. For example, consider a loop that runs for a certain duration and plays a sound over the user's speakers. If this loop was subjected to automatic break insertion, then the sound would play for a possibly shorter duration. Similarly, if a loop sends data over the network, then halting this loop prematurely causes not all data to be sent. To avoid this happening, we will first check the loop to see if it contains such I/O operations. If this is the case, then the loop is no longer considered for automatic break insertion, because we cannot guarantee that its insertion does not cause the program to behave incorrectly. If the loop contains no I/O operations, then the loop continues to be considered for automatic break insertion.

3.3 Liveness analysis

To be able to implement automatic break insertion more aggressively, we are interested in knowing which variables are used after the loop has finished iterating. In other words, we are interested in which variables are live at the end of the loop. The live variables after the loop are the variables of which we must ensure that their state is the same with and without the inserted break. For the variables which are not live, we do not care about maintaining this property, since any possible differences in their values will not be noticed by the program. An example of a variable that may not be live is a loop induction variable. If the loop induction variable is not used after the loop it is used in, then it does not matter if the inserted break statement would cause this induction variable to have a different value. A slight exception is made for variables that are immediately assigned to after the loop. If a variable is used after the loop, but is assigned to before it is used, then we ignore that variable. Since the variable is immediately assigned to, its value after the loop will be overwritten and therefore not matter.

3.4 Circular data dependencies

Since a loop may be halted prematurely, it might be the case that certain iterations of a loop that would normally modify a variable, can no longer do this. In principle, we can only be certain that statements that are pure or idempotent will have the correct state, since the loop may only iterate once. Furthermore, a combination of idempotent statements may also combined not be idempotent, as can be seen in listing 5.

```
int a, b;

while (...) {
  a = b + 1;
  b = a + 1;
}
```

Listing 5: The two statements are idempotent individually, but not combined.

How do we identify when a combination of idempotent statements is not idempotent together? Statements may not be idempotent together when the variables that are changed by these statement are also dependencies of these statements. As an example, the statement n = n + 1 is not idempotent, since the variable it changes is the same variable it uses to determine the result of the variable. We are therefore interested in finding whether variables are circularly dependent on each other. If this is the case, then this might mean that the group of statements is no longer idempotent as a whole. Note that this might also detect a circular dependency which does not cause the group of statements to no longer be idempotent. See section 7 for a discussion on this. To address this potential issue, we will analyze the code inside the while loop for circular dependencies. To do this, we will create a directed graph of all variables, with edges pointing to the dependencies of a variable. For example, for a statement like a = b + c we would get a dependency graph as shown in figure 2. We analyze all statements in the loop, also those inside nested blocks, such as **if** blocks and nested loops. We will assume for now that all these statements are executed at once and are not influenced by the block they may be contained in. In later steps of the analysis, we will check for the correctness of this assumption and possibly rectify any incorrect assumptions made in this step.

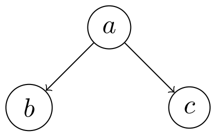

Figure 2: A graph showing the dependencies on the variables.

Once we have created a graph for all the statements inside the loop, we will analyze the dependencies of variables. We are only interested in the set of variables that we have found in section 3.3, since we must ensure the correctness of their values. For variables that are not used after the loop, their values may be different in the presence of automatic break insertion. To determine whether there is a circular dependency on a variable, we want to see if there is an infinite walk in the graph, starting from that variable. If there is an infinite walk, then this must mean that there is a loop reachable from the starting vertex. We perform this analysis for all variables from 3.3. If such a variable has no vertex in the graph, then this means that it is not used in an assignment statement in the loop. In this case, the variable has no circular dependencies. If we find any variable that has a circular dependency, then we cannot do automatic break insertion for this loop, since we cannot ensure the correctness of that variable's value. Only if all variables do not have a circular dependency, may break insertion be possible and do we continue our analysis.

Furthermore, we must also ensure that variables are updated in the correct order. It is possible for variables to not be circularly depend on each other, but still cause multiple iterations to reach a fixed value, as shown in listing 6. This occurs when the dependencies of a variable are assigned after the assignment of this variable. For example, as can be seen in listing 6 variable a depends on variable b which is assigned to after the assignment to a. Because of this, it now takes more than one iteration of the loop for this variable to get its final value. To account for this, a variable's dependencies may not be updated after the variable has been assigned to. This again, only in regards to variables that are live after the loop.

```
int a, b;

while (...) {
  a = b;
  b = 1;
}
```

Listing 6: The value of a will only be 1 after the second iteration, even though there are nor circular dependencies.

3.5 Analyzing inner blocks' purity

Up until now, we've assumed that all the code in the loop, was placed sequentially, with no possible change in program flow. Blocks, such as if statements and inner loops, might violate this assumption, forcing us to do more detailed analysis. In the following steps of the analysis we will analyze blocks inside of the loop to determine if these blocks might cause automatic break insertion to fail and to determine in which block the break statement should be placed. To understand in which blocks we are interested, we want to see which blocks are pure. In this case, with pure, we mean a block that does not change any of the variables we found in section 3.3 or any of the variables these variables are dependent upon as we found in section 3.4. If a block is pure, then its execution does not matter, since it does not change any variables that are live after the loop, or any dependencies of such variables.

We first go through the loop and collect all of the blocks that occur inside this loop. In particular we are interested in blocks that may cause a change in program flow, such as blocks that are only entered upon a condition being met, such as if statements and loops. Blocks that do not divert the program flow, i.e. they are always entered and always immediately continue with the code below them, do not necessitate this more detailed analysis, since they behave the same as if they were placed directly in the loop.

When we have found all of the blocks, we want to check which of these blocks are pure, as by the definition stated earlier. We do this by going through each block and checking for each of the blocks if they make a change to the set of variables as found in section 3.3 or the dependencies of those variables. If a block makes no changes to these variables and is therefore pure, we discard the block from further analysis. Since it is pure, we do not care whether the code inside of the block is executed at all, or even multiple times; it will not change any of the variables we are interested in. For blocks that are not pure, we keep them around and will analyze them further in the following steps.

Pure blocks are also not eligible for break insertion. This is because, even if a break statement inside the block would be valid, then we can also place the break statement just outside of the block, either directly before or after it, since the block will not make changes to variables we consider. Placing the break statement outside of the block before the pure block would be a better position for the break statement, since then the loop also skips the statements inside the pure block that do not alter the behaviour of the program, potentially saving additional execution time.

3.6 Purity analysis after inner blocks

After we've found all inner, impure blocks, we now must check whether these blocks are not followed by impure statements. Purity here again refers to the set of variables from section 3.3 and their dependencies not being changed. Impure statements after a block are problematic, because both possible positions for the break statements may not take into account the possibility of later changes.

If the break statement is placed after the block and the impure statements following the block, then the code inside the block may not have been executed, leaving the state of the program ambiguous. On the other hand, if the break statement is placed within the block, at the very end, then the impure statements succeeding the block may not have run yet, again possibly leaving the program in an undetermined state. To combat this, we must ensure that the statements following the block are pure. If this is the case, then the second option - placing the break statement inside the block - may be possible. Even though the statements following the block may still not have run when the break statement is hit, this is no longer an issue, since these statements are pure and will not cause any problematic changes to the program state.

To determine if the statements after the block are pure, we start after the block and read the statements up until the end of the loop. If any of these statements cause changes to the variables from section 3.3 or their dependencies, then this block is followed by impure statements. In that case, break insertion is not possible, because we cannot ensure the correct program state. If we find such a block, break insertion is not possible for the loop that contains this block. Only if all impure blocks are not followed by impure statements, may break insertion be possible. In that case, we continue the analysis.

3.7 Dependencies among impure inner blocks

It might be the case that there are multiple impure blocks that are not followed by impure statements as determined in the previous sections. In this case, we need to determine if break insertion is still possible and if so, in which block the break statement should go. Since all the blocks we consider are impure, we cannot afford to place a break statement in a position where the break statement can be encountered before all other blocks have been executed at least once. If that were to be the case, we might skip executing statements that are important for the correctness of this optimization. To solve this, we must analyze all of the impure blocks to create a dependency graph for them. If a single block is dependent upon all previous blocks, then this block can be used for placing the break statement safely. This is possible, since the execution of this block relies on the other blocks being executed at least once. Therefore, once this block is entered, all of the other blocks will have been executed prior. If this block is found, then this block is the only candidate for break insertion. If there does not exist a block which is dependent on all other blocks, then we cannot place the break statement in any block while maintaining the safety of the optimization. In this case, break insertion is not possible for this loop.

```
boolean x = false;
boolean y = false;

while (...) {
    //block 1
    if (!x) {
        x = true;
    }

    //block 2
    if (x) {
        y = true;
    }
}
```
Listing 7: A loop having two blocks where the second is dependent on the first.

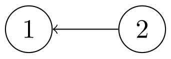

Figure 3: A graph showing the dependencies on the block from listing 7.

3.8 Placing the break statement

After going through the previously outlined steps for the analysis, we can finally determine where to place the break statement. There are two possible situations. Either, there is an impure block which is dependent on all other impure blocks as determined by section 3.7; or there are no impure blocks inside the loop. We will first go over the first situation.

If there is an impure block that is dependent on all other impure blocks, then the break statement should be inserted in that block. The position of the break statement within the block should be anywhere after the last non-pure statement inside the block. In this case, a non-pure statement is a statement that may cause changes to any of the variables as determined in section 3.3 and section 3.4. Since these statements will not affect any variables that are relevant after the loop, it is save to not execute them. Given the same reasoning, placing the break statement at a later position in the block is also possible, since executing pure statements does not matter for the program state. Placing the break statement earlier may improve execution speed, however, since it might cause pure statements to not be executed.

The second situation occurs when there are no impure blocks inside the loop. If this is the case, then the break statement should be positioned directly inside the loop itself, after the last non-pure statement in the block. Placing the break statement later in the block is also possible, since the execution of pure statements does not matter for the relevant program state after the loop.

An alternative approach for the second situation is to rewrite the loop and not place a break statement at all. Let us make the observation that, when a break statement is placed inside the loop, the loop either doesn't run or runs once. Looping zero times would occur if the condition for entering the loop does not hold. Looping once occurs when the loop condition holds and then the relevant break statement is executed. Since this break statement is directly inside the loop, it will always be executed when the loop is executed. It is therefore not possible that the loop iterates more than once. This makes the loop equivalent to an if statement, with the same condition. Replacing the loop by an if statement would make for a simpler control-flow graph, which may allow other optimizations to have a better understanding of the program and allow for more optimizations.[1].

When there are multiple loops in the program being analyzed, it might occur that two break statements would be placed in the same position as seen in listing 8. This is not an issue, but strategically choosing how these break statements interact might be beneficial for program execution time. If two or more break statements are to be placed at the same position, then these loops are inside of each other. In this case, halting the outer loop's execution would cause all of the loops to be halted. If one of the inner loops is halted, the outer loop will continue to the next iteration, even though this is not necessary. Therefore, when multiple break statements can be inserted, only the break statement corresponding to the outermost loop should be inserted to potentially improve execution time.

If the break is placed directly in the loop and not in an if statement, the loop can be rewritten to become an if statement and then no break should be inserted.

```
boolean x = false;

while (...) {
    while (...) {
        x = true;
    }
}
```

Listing 8: Nested loops, where both can have a break statement inside the inner loop.

After the break statement has been placed, the analysis for this loop has finished.

4 Infinite loops

While the analysis outlined in section 3 works correctly, there might still be an unexpected behavioral change after applying it to a program. The analysis does not check if a loop runs infinitely. If a loop that does not halt is eligible for automatic break insertion as outlined in section 3, a break statement will still be inserted, possibly halting the loop. This may cause an unexpected change, since the programmer might have intentionally wanted the loop to not end early, an example of which can be seen in listing 9.

[1]Care should be taken if a loop is changed to an if statement. If dangling else statements are present, then the if statement should not cause the dangling else to pair to this if statement, but should still pair to the original if statement [1].

```
int n = ...;

while (n > 1) {
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}

System.out.println("Collatz conjecture holds for input");
```

Listing 9: A program testing the Collatz conjecture for a specific number [2]. Automatic break insertion deems the loop unnecessary and will always print the success notice, even though the Collatz conjecture has not been proven and this program might therefore never terminate for some n [3].

The reason for this issue is that the analysis makes the assumption that the only state within a program, is contained within variables or state outside of an individual program. These are addressed in section 3.2 and 3.3. However, the program counter i.e., the position of the program's execution is not considered as part of the program state. As can be seen in listing 9 code may rely on the program counter to assert certain assumptions about the code. For listing 9 there is an implicit assertion being made that, at the end of the program, the value has reached the number 1 and therefore the Collatz conjecture holds for this input. Given the program, this must be the case, since for the loop to end !(n > 1) must hold. The analysis for automatic break insertion as described in section 3 does not consider that this implicit assumption may be made in the program and therefore does not take this into account.

Nonetheless, an implementation of this analysis might want to take into account infinite loops. There exists no general algorithm to assert whether a loop halts [4],[2] however there do exist algorithms for subsets of loops, which can determine whether a loop will halt or not [5, 6]. If one wishes to ensure that break statements are only inserted for loops which would otherwise also halt, one should analyze the loop using such an algorithm and ignore the loop as a candidate for automatic break insertion if the loop cannot be guaranteed to halt. The analysis as described in section 3 is otherwise unaltered. For the remainder of the paper, we will assume that such loops are not an issue.

5 Relation to other optimizations

5.1 Liveness analysis

Liveness analysis is a common procedure used in a variety of other optimizations and transformations.

[2]For languages which are Turing-complete.

An example of such an optimization is register allocation. Programming languages generally allow for a large or unbounded amount of variables to be declared. To enhance execution speed of this program, it would be optimal if all these variables resided in the processor's registers: these have faster read and write access than general random-access memory. However, the amount of registers in a processor is generally quite small. This means that not all variables may actually fit in the available registers. Register allocation deals with the choice of which variables should be put in registers at which point in time. Liveness analysis is important in this optimization. If a variable is no longer being used, keeping it in a register is wasteful and it would be better used by a variable that is actually still in use [7].

5.2 Dead code elimination for infinite loops

As shown in section 4, infinite loops might be terminated due to automatic break insertion, causing the loop to no longer be infinite. Listing 9 shows an example of this, where the code will always report that the Collatz conjecture holds, even though this might not be true. Another optimization, dead code elimination, might also remove the loop, due to the fact that none of the computations that are done inside the loop are ever used. When adjusting the code in listing 9 to C as shown in listing 10 and compiling with Clang 10.0.0, we can see the same issue occur. The generated Assembly code is void of any loops or variables and makes a call to the printf function directly. This, even though the loop above may never halt and therefore the message may normally never show.

```c
int n = ...;

while (n > 1) {
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}

printf("Collatz conjecture holds for input\n");
```

Listing 10: The adaptation of the Collatz program from listing 9 into C

6 Implementation

To show automatic break insertion in action, we created a compiler on which a form of automatic break insertion has been implemented. This compiler will compile the code and apply automatic break insertion, showing where it finds targets for this optimization and applies them during the compilation process.

6.1 Language

Our compiler compiles an adapted version of the language Orange Juice [8]. Orange Juice is a simplified Java-like language. In Orange Juice, there exist no classes or methods and the only type that is available is **int**. There are two types of blocks, a **while** loop and an if block. Unlike Java, the **else** block is mandatory [9]. To provide I/O, there are two constructs available, in() and out() taking input and output respectively. Input may only be provided in the form of an integer. In between the parentheses of out() a string must be provided to be printed as the output. Beyond that, standard integer operations and comparisons are available.

The language has been extended to provide support for integer arrays. Listing 11 shows the changes made to support this.

```
declaration ::= int_declaration
              | array_declaration

int_declaration ::= "int " variable ";"
array_declaration ::= "int[" expression "] " variable ";"

assignment ::= int_assignment
             | array_assignment

int_assignment ::= variable "=" expression ";"
array_assignment ::= variable "[" expression "]" "="
                     expression ";"

factor ::= variable
         | integer
         | "(" expression ")"
         | input_expression
         | array_load

array_load ::= variable "[" expression "]"
```
Listing 11: The changes made to the Orange Juice language to support arrays.

In essence, three new constructs have been added: an array declaration, an array assignment and an array load. The array declaration declares an array of integers of a given size. The array assignment assigns a value to an index in the array. The array load loads a value from the array at a certain index. The addition of these constructs allows a programmer to make use of arrays. It should be noted that this does not encompass the full complexity of integer arrays in Java, since a variable containing an array cannot be overwritten with a new array of a different size. This makes creating a dynamically sized array impossible in Orange Juice.

Listing 12 shows how listing 1 looks when converted to Orange Juice. While the code has differences compared to the original Java code, its structure is very similar to Java.

```
int[10] array;
int found;
int index;

while(index < 10) {
  if(array[index] > 5) {
    found = 1;
  } else ;

  index = index + 1;
}
```

Listing 12: Listing 1 converted to Orange Juice.

6.2 Compiler

We created a compiler to compile Orange Juice code into JVM bytecode which applies automatic break insertion [10]. The compiler is created in Java with the help of ANTLR for parsing Orange Juice and ASM for exporting JVM bytecode [11, 12]. The existing grammar for Orange Juice has been used and adapted for ANTLR, which was then used to generate the necessary classes for tokenization and parsing. When compiling, ANTLR will first read the Orange Juice file and tokenize it. After the file has been tokenized, the tokens will be parsed to generate an AST that resembles the structure of the original file. From here we apply a visitor over the entire AST and translate the written Orange Juice code into JVM bytecode [13]. To help with this, ASM is used to write the appropriate instructions and maintain the constant pool. All of the JVM bytecode is implemented in the main method of a class, which are the only methods and classes in the resulting JVM bytecode. This resembles the file layout of Orange Juice, since Orange Juice does not have classes or methods. The written JVM bytecode is compiled into a class file, which consist of all of the compiled code. To turn this into an executable JAR file, the class file is put in a ZIP file along with a manifest file. The manifest file points Java to the main class so it can be executed. The resulting file is an executable JAR which behaves the same as its corresponding Orange Juice file.

6.3 Implementation of automatic break insertion

In between the parsing of the Orange Juice file and compiling it, the analysis required for automatic break insertion takes place. This is done by using several visitors, going through the AST and collecting the information necessary to determine if automatic break insertion can be applied as described in section 3. First we visit and collect all **while** loops. These are the only loops that exist in Orange Juice and they're guaranteed to be natural loops. We then go through the while loops to find whether they contain I/O constructs. I/O in Orange

Juice can only be achieved via the use of in() or out() statements. If such a statement is found within the loop, we discard it from further analysis.

We then continue doing liveness analysis. We implement this by going over the AST after the end of the loop. We collect all referenced variables. An exception is made to variables that are assigned to before they are referenced. In this case, the value will be overwritten and its prior value does not matter. Arrays are handled differently. We cannot always be certain in which position of the array a value will be written to. The index may depend on user input, for example. Since we cannot know certainly if two assignments to the same array are in the same place or not, we assume that the entire array is referenced. This also means that the exception we provided does not hold for arrays. After all, if the array is being written to, we cannot be certain that all values are written to. In this case, we assume no indexes are being written to, i.e. the exception doesn't apply.

After doing liveness analysis, we create a dependency graph for all of the variables inside the loop. We do this by iterating through the loop and creating vertices for all referenced variables. When a variable is assigned to, we create an edge from the vertex representing this variable to the vertices for the variables that its value depends on. For arrays we treat the array as a single variable due to our inability of knowing the indexes that are being written to. Doing this for all assignment statements creates a dependency graph. We then use the variables from our earlier liveness analysis and check in the graph whether their vertices contain an infinite walk. We determine this by doing a depth-first search in the graph. If the same vertex is visited twice, there exists an infinite walk. If there exists an infinite walk for any of these vertices, the loop is not eligible for automatic break insertion.

Now we collect all of the blocks inside the loops. These are **if** and while blocks. We then first check if any of the blocks are pure. We do this by going through the loop and seeing if there is an assignment to any of the variables from liveness analysis or their dependencies. If this is not the case, we consider the block pure. Pure blocks are discarded from further analysis.

Continuing with the non-pure blocks, we check if there are any constructs after the impure blocks which are also impure. We do this by going to the end of the block and seeing if there is any assignment to a variable from liveness analysis or any of its dependencies. If this is the case, then the loop is not eligible for automatic break insertion.

Lastly for the analysis, we check if the impure blocks are dependent on each other. We implement a primitive version of this by checking if the blocks are nested. If the blocks are nested, then inside the innermost block we are sure that all blocks are executed. If this is the case, the loop is eligible for automatic break insertion. If this is not the case, the loop is not eligible for automatic break insertion.

We collect all the break statements that can be placed, along with where they can be placed and to which loop each of the break statements belongs. These are passed to the compiler. The compiler will place the break statement in the right block by adding a jump to jump to the end of the loop. If there are multiple break statements in the same position, then the break statement that jumps to the outermost loop is chosen.

7 Future work

In the future we would like to expand automatic break insertion as to extend the applicability of the optimization. In particular, there are several possible targets to which we can extend the analysis to allow for this optimization to be applied more aggressively.

One such target is non-natural loops. The analysis we described was constrained to natural loops only, immediately discarding non-natural loops as potential candidates for automatic break insertion. In the future, the applicability of this optimization on non-natural loops should be researched, altering the analysis where necessary to accommodate for these loops. We suspect that for jumps that go out of the loop, the analysis should ensure that, when placing a break statement such a jump statement could not be executed if the loop were to continue. This, because if this condition is not met, we cannot ascertain that the loop with automatic break insertion applied exits the loop at the same point as the loop without this optimization. For loops where a jump goes into the loop, the analysis should ensure that all of the non-pure statements in the loop that come before the jump site have been executed as well.

In our analysis, we have currently only considered the option of placing a break statement directly. In doing this, we have to make several compromises. For example, blocks inside the loop have to be dependent on each other. We can avoid this by introducing additional state that keeps track of which blocks have been entered. If all blocks have been executed once, then we could also break out of the loop. While this change would require minimal changes to the current analysis, the introduction of an additional variable and the necessity of checking the state of this variable before executing the break statement makes this an optimization that may not always provide an increase in execution speed. This work would have to be accompanied with research into the effects of this potential speed decrease and increased memory consumption and justify these trade-offs.

The current analysis does a circular dependency check to ensure that the state of the program does not change after multiple iterations. This circular dependency check works by creating a dependency graph and checking if there exists an infinite walk. While this functions correctly, it may mark circular dependencies that are not of interest in our analysis. For example, n = n % c where c is a constant has a circular dependency: n is dependent on itself. However, this is not of interest for our analysis. Repeated execution of this code will only change the state of n once at most, making this line of code idempotent. This is not an issue, because our analysis ensures that non-pure statements are executed at least once, but our current approach for circular dependency analysis is unaware of this. Future work should extend the circular dependency check to be more fine-grained, to improve upon this.

It is currently unclear as to when this optimization should take place. Our assumption is that this optimization should take place relatively early, due to the possibility of simplifying the control-flow graph. Recall that if automatic break insertion is possible directly in the loop and not inside one of its inner blocks, that the loop can be converted to an if statement. Due to this simplification in the control-flow graph, other optimizations may be able to perform better. Dead code elimination, however, can improve the functioning of automatic break insertion, if it can remove dead code inside loops. Therefore, we hypothesize that automatic break insertion is best performed after dead code elimination, but before other optimizations. Future research should analyse this in more detail, preferably giving a clear positioning of automatic break insertion among other optimizations.

Lastly, it is unclear how useful this optimization is in practice. We have shown a variety of examples that illustrate the possible advantages automatic break insertion might have to optimize code. However, many programming languages allow the programmer to place break statements manually. Therefore, it might be that in a lot of code, this optimization might have already been applied manually by the programmer, making our automatic optimization irrelevant. Work should be done to establish how often there is code that does not have this manual break insertion applied and why this was not the case. Better insight in this could also show where manual break insertion is difficult and show us how automatic break insertion can be extended most effectively.

References

[1] C. Clark, "What to do with a dangling else," *ACM SIGPLAN Notices*, vol. 34, no. 2, pp. 26–31, 1999.

[2] L. Collatz, "On the motivation and origin of the (3n+ 1)-problem," *reference 1, JC Lagarias, The Ultimate Challenge*, p. 244, 2010.

[3] J. C. Lagarias, *The ultimate challenge: The 3x+ 1 problem.* American Mathematical Soc., 2010.

[4] A. M. Turing *et al.*, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[5] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," *Acm Sigplan Notices*, vol. 41, no. 6, pp. 415–426, 2006.

[6] A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening, "Loop summarization and termination analysis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 81–95, Springer, 2011.

[7] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer languages*, vol. 6, no. 1, pp. 47–57, 1981.

[8] R. J. Botting, “The programming language oj,” Apr 2011.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*, ch. 14.9. Addison-Wesley Professional, 2000.

[10] S. van Schie, “Automatic break insertion.” https://github.com/sse245/AutomaticBreakInsertion, July 2022.

[11] T. J. Parr and R. W. Quong, “Antlr: A predicated-ll (k) parser generator,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[12] E. Bruneton, “Asm 3.0 a java bytecode engineering library,” *URL: http://download. forge. objectweb. org/asm/asmguide. pdf*, 2007.

[13] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides, *et al.*, *Design patterns: elements of reusable object-oriented software.* Pearson Deutschland GmbH, 1995.