Bachelorthesis

Automatic Transformation of Iterative to Tail-Recursive Functions in Python

Alexander Mühlbauer

July 25th, 2018

Reviewer
Prof. Dr. Torsten Grust
Database Systems Research Group
University of Tuebingen

Supervisor
Christian Duta
Database Systems Research Group
University of Tuebingen
Mühlbauer, Alexander:
Automatic Transformation of Iterative to Tail-Recursive Functions in Python
Bachelorthesis in Computer Science
University of Tuebingen
Period: 04.04.2018 bis 25.07.2018
Abstract

Context The groundwork for this thesis is the paper *Automatic Transformation of Iterative Loops into Recursive Methods* [IS14] by Insa and Silva, published in 2014. In their work Insa and Silva provide transformation rules for all iterative loops into equivalent tail-recursive functions and give an implementation of their algorithm in Java. While *Python* and *Java* both support iterative loops and recursion, there are some fundamental differences that require to deviate and even extend the approach of Insa and Silva.

Objective This thesis not only aims to provide a methodology that can be used to transform all iterative loops into recursive functions, but also to offer a proof of concept for developing that transformation as a compiler for source code written in the Python programming language.

Results Taking the transformation rules provided by Insa and Silva as a starting point it was possible to modify and extend them to be applicable to the *Python* programming language. Furthermore this thesis offers an approach to transform iteration in the context of the comprehension syntax and even for the *Python* specific generator functions.

Conclusion Implementing an automatic loop to tail-recursive function transformation in another multi-paradigm programming language than Java is possible using the methodology offered by Insa and Silva. This proves their claim and serves as a proof of concept for their methodology to be adopted in other programming languages, supporting iteration and recursion, as well.
# Contents

1 Introduction 6  
1.1 Motivation .................................................. 6  
1.2 Objective ................................................... 6  
1.3 Implementation Language ................................. 7  
1.4 Structure ..................................................... 7  

2 Methodology 8  
2.1 While-Loop Transformation ............................... 8  
2.1.1 Building the tail-recursive function: simple case ............... 9  
2.1.2 Approach Control-Flow Changing Statements .................. 11  
2.1.3 Placement of the tail-recursive function ...................... 19  
2.1.4 Naming introduced variables .............................. 20  
2.2 For-Loop Transformation .................................. 21  
2.3 Comprehension to For Loop Transformation .................. 25  
2.4 Loops in Generator Functions ............................... 26  

3 Implementation 28  
3.1 Python AST .................................................. 28  
3.2 Transformation Pipeline ................................. 31  
3.3 Transformation Steps ....................................... 33  
3.3.1 Preprocessing steps ...................................... 33  
3.3.2 Loop transformation ..................................... 41  

4 Related Work 48  

5 Conclusion 49  

References 51
List of Figures

1. Transformation of a while loop (cf. [IS14]) .......................... 8
2. flow chart of Python’s while-else statement .......................... 9
3. extended transformation scheme, including return statements .... 11
4. flow-charts of loops containing break or continue .................... 13
5. flow-charts of loops containing break or continue .................... 13
6. extended transformation scheme, including continue ............... 15
7. transformation scheme for while loops with try/except statement inside body ......................................................... 17
8. Blackbox that transforms any for loop into a semantically equal while loop ................................................................. 21
9. opcode of the for-loop of fig. ; numbers on the left indicate the line number in the source code ........................................... 22
10. flow-chart of both loop implementations given in figure 11 ......... 23
11. for to while-loop transformation ........................................... 23
12. control flow of Python’s for loop with an else clause .............. 24
13. Transformation of list comprehension to for loop(s) [Fou18] ....... 25
14. Defining an infinite stream of integers, with a generator function .... 26
15. Infinite stream of integers generator, defined with recursion ....... 27
16. ast.Module node with its attributes .................................... 28
17. ast.While node with its attributes ...................................... 28
18. ast.If node with its attributes .......................................... 29
19. ast.Try node with its attributes ....................................... 29
20. ast.Assign node with its attributes ................................... 29
21. ast.ListComp node with its attributes ............................... 30
22. ast.FunctionDef node with its attributes ............................. 30
23. Transformation pipeline; input is a Python-AST; output is a modified version of the AST .................................................. 32
24. inspecting step 1 of the transformation pipeline ..................... 34
25. inspecting step 2 of the transformation pipeline ..................... 36
26. Statically typed vs. dynamically typed variables .................... 38
27. inspecting step 3 of the transformation pipeline ..................... 39
28. inspecting step □ of the transformation pipeline ...................... 40
29. inspecting step 4 of the transformation pipeline ..................... 41
30. Implementation of the Levenshtein distance algorithm ............. 43
31. Levenshtein distance algorithm after transformation step ② of the transformation pipeline ........................................... 44
32. Levenshtein distance algorithm after transformation step ③ of the transformation pipeline ........................................... 44
33. Levenshtein distance algorithm after transformation step ④ of the transformation pipeline ........................................... 47
1 Introduction

1.1 Motivation

Iterative loops and recursion are both different concepts capable of expressing the same intention; however, the concept of loops does not even exist in the functional programming paradigm. The way iteration (or repeating code) is dealt with in functional programming is recursion.

Despite that, there exist a lot of programming languages that support both concepts and leave the programmer with the choice of which to use. Regarding performance, iteration is often used over recursion, since a lot of compilers can deal with iteration more efficiently [IS14]. On the other hand, it is commonly accepted that many algorithms can be implemented much more cleanly using recursion [Fil94].

Currently, the chair for Database Systems Research (University of Tuebingen), does research on transforming (tail) recursive functions to declarative functions without recursive function calls. The goal is to be able to write certain algorithms in an intuitive way. A declarative implementation comes with observable performance benefits. Additionally, being able to transform all functions using loops to a tail-recursive formulation means, that this model can be extended by first converting all loops to tail-recursive functions. In consequence not only recursive functions may be automatically converted to SQL functions without recursive function calls, but also functions, using loops.

1.2 Objective

Although iteration is considered to be easily definable as (tail) recursion [Fil94], notably, very few implementations of automatic transformation from iterative loops to recursion can be found. Insa and Silva [IS14] came across this issue and published a publicly available Java library\(^1\) that is able to remove all iterative loops in Java-Code, replacing them with the corresponding (tail) recursive function. Along with their library, they propose a methodology to do this transformation, which they say is general enough to be adapted in other programming languages. The objective of this thesis is to establish a proof of concept by implementing their methodology in another programming language and to acquire insights to the process of implementing such a compiler.

\(^1\)http://www.dsic.upv.es/~jsilva/loops2recursion/
1 Introduction

1.3 Implementation Language

To realize automatic loop transformation, for a proof of concept, Python is chosen. It is a multi-paradigm programming language, that already provides the necessary tools to modify program text via the abstract syntax tree (AST). Furthermore it provides methods to programmatically transform an AST and to obtain source code from an AST. To present the underlying concept Python syntax will be used as well, because it is easy to read and understand. If not stated otherwise this work refers to the CPython implementation (v. 3.6)\(^2\).

1.4 Structure

This thesis is divided in five sections, including the introduction. In section 2 a general methodology to transform iterative loops to tail-recursive functions is provided. The methodology section also proposes an approach to be able to transform loops in the context of the Python specific concept of generator functions. Additionally a strategy to prepare the comprehension syntax in order to transform it to a tail recursive function is presented. Section 3 describes the actual implementation in Python. Therefore the Python abstract syntax tree (AST) is addressed briefly. The transformation process can be depicted as a pipeline, that is also introduced in this section to give an overview. In section 3.3 each step of the implementation is explained in detail. In section 4 we give an overview of related work and the final section 5 gives a conclusion and presents possible future work, related to this thesis.

\(^2\)https://docs.python.org/3.6/index.html
2 Methodology

In this section the approach for loop-transformation is discussed. Having the `for` and `while`-loop Python offers two iterative control-flow statements that are able to repeat code, subject to a condition. In the following sections transformation rules for both loop types will be provided, using the suggested transformation by Insá and Silva [IS14], serving as a starting point to adapt the methodology to Python programming language. Additionally an approach to transform the comprehension syntax, implicitly using `for` loops to create lists, sets or dictionaries is addressed in section 2.3 as well as a transformation rule for loops being used in the context of the Python specific concept of generator functions is provided in section 2.4.

2.1 While-Loop Transformation

Any `while` loop can be transformed into a corresponding tail-recursive function, with the actual loop code being replaced by a caller. Dependent on the statements used inside of the `while` loop’s `body` the transformation needs to be extended. Statements that can change the control-flow inside of the loop such as `break`, `continue`, `return`, `try/except` need special treatment if they occur inside the `body` of a loop. To explain the methodology the treatment for each of these statements is broken apart.

Starting with the simple case (fig. 1), where it is assumed that `body` does not contain any control-flow changing statements. The corresponding transformation scheme of the simple-case is depicted in figure 1.

![Figure 1: Transformation of a while loop (cf. [IS14]).](image)

The `while` statement in Python supports a special functionality, where an `else` clause can be placed directly after the loop-body to execute code if the condition (`<cond>`) evaluates to false. This code is not executed if the statements `break` or `continue` are used to alter the control-flow of the loop code inside `body` (see, fig. 2).

We can address this feature by carrying over the `else` clause - together with its body `else_body` - to the callers `if` statement.
2 Methodology

![Flow chart of Python's while-else statement]

2.1.1 Building the tail-recursive function: simple case

The tail-recursive function (\texttt{loop()}) expects certain parameters, being the variable names, that values are assigned to inside of body and all variables used in the loop condition \texttt{<cond>} as its arguments. The function body of \texttt{loop()} contains the original loop \texttt{body} and a recursive call. The body of a \texttt{while} loop is only executed if the condition (\texttt{<cond>}) evaluates to \texttt{True}, so recursive calls are always wrapped by an \texttt{if} statement first checking \texttt{<cond>}.

The values, assigned to the variables in \texttt{vars}, can change during the execution of \texttt{loop()}, therefore \texttt{vars} has to be returned by the \texttt{loop()} function when the recursion has finished.

The \texttt{caller} replaces the actual loop in the code. It is built by a call to the tail-recursive function (\texttt{loop()}) that is assigned to a variable (\texttt{result}), to be able to access all elements contained in \texttt{vars}.

In \texttt{reassign vars} the modified variable values, stored in the \texttt{result} list, are reassigned to their original variable names (e.g. \texttt{var1 = result[0], ...}). This assures that they can be used in the following code (\texttt{after}).
2 Methodology

Example   This example shows the transformation of a function fib(n) that implements an algorithm to return the $n$-th fibonacci number:

```python
def fib(n):
a, b = 0, 1
while n > 0:
a, b = b, a+b
n -= 1
return a
```

Applying the methodology to transform while loops yields a fib(n) function that implements the same algorithm using tail-recursion instead of an iterative while loop.

```python
def fib(n):
    def loop(n, a, b):
a, b = b, a+b
n -= 1
if n > 0:
    return loop(n, a, b)
return [[None], n, a, b]
a, b = 0, 1
if n > 0:
    result = loop(n, a, b)
n = result[1]
a = result[2]
b = result[3]
return a```
2 Methodology

2.1.2 Approach Control-Flow Changing Statements

In the following paragraphs it is addressed how to modify the methodology of the simple case in order to be able to deal with the control-flow changing statements return, break, continue, try/except.

Transforming Return-Statements Return statements may occur inside a loop, if it is enclosed by a function. Obviously a return statement can not be carried over into the body of the tail-recursive function that has to be built, because this would affect the control-flow of the tail recursive loop() function, instead it should invoke the original enclosing function to return. Therefore transforming such a loop needs the transformation scheme of the base case (1).

If a return statement is present inside a while loop’s body, this statement needs to be modified before it can be carried over to the functions body in order to propagate it to the surrounding function.

In consequence it is no longer enough to return vars, instead information about control-flow changing statements also need to be returned. To be able to tell if a return statement was executed in the loop() function, a dedicated tuple as the first list element, named control, is introduced. The control tuple stores information about a statement that changes the control-flow. Its first value is either a None value, indicating that the control-flow is not altered, or the actual statement as a string (i.e. ‘return’). If a return statement is executed control is also used to propagate the expression (<expr>), that must be returned by the enclosing function, as the second value of the tuple.

To process the new type of the return value of loop(), supporting control, we also need to adapt the caller accordingly. Therefore the first element of result is assigned to a variable called control and then checked for a control-flow changing statement. If the first element of control holds the string ‘return’ <expr> is returned.
2 Methodology

Example  This example shows the transformation of a function \( \text{lcm}(a,b) \) that implements an algorithm to return the least common multiple of two integers \( a \) and \( b \):

```python
def lcm(a,b):
    candidate = a if a >= b else b
    while True:
        if candidate % a == 0 and candidate % b == 0:
            return candidate
        candidate += 1
```

Applying the methodology to transform \( \text{while} \) loops, considering the \( \text{return} \) statement, yields a \( \text{lcm}(a,b) \) function that implements the same algorithm using tail-recursion instead of an iterative \( \text{while} \) loop.

```python
def lcm(a, b):
    def loop(candidate):
        if candidate % a == 0 and candidate % b == 0:
            return ['return', candidate], candidate
        candidate += 1
        if True:
            return loop(candidate)
    return [None], candidate

    candidate = a if a >= b else b
    if True:
        result = loop(candidate)
        control = result[0]
        if control[0] is not None:
            if control[0] == 'return':
                return control[1]
        candidate = result[1]
```

12
Transforming Break  Break can be used to alter the control-flow of a loop. If a break statement is reached during the execution of a loop, the program exits the enclosing loop and jumps directly to the next instruction after the loop.

Code that is located after a break statement, but still inside body, will be skipped. In the context of recursive functions, each iteration of a loop is regarded as one recursive call [IS14].

![Flow-charts of loops containing break or continue](image)

Figure 4: flow-charts of loops containing break or continue

Since the control-flow statement break cannot be used outside of an enclosing loop, we need to transform it. Having a look at figure 5, we can identify how to transform break to achieve the equal behavior.

Exiting the tail recursive function can be achieved by returning in the context of recursion. This implements the same behavior as a break statement inside a loop.

![Flow-charts of loops containing break or continue](image)

Figure 5: flow-charts of loops containing break or continue
2 Methodology

Example  This example shows the transformation of a function \texttt{first\_occurrence\_of\_letter\_in\_string(letter, string)} that implements an algorithm to return the position of the first occurrence of a letter in a string if it exists:

```python
def first_occurrence_of_letter_in_string(letter, string):
    count = 1
    length = len(string)
    while count <= length:
        if string[count-1] == letter:
            break
        count += 1
    if count > length:
        return 'No occurrence of {} in {}'.format(letter, string)
    return count
```

Applying the methodology to transform \texttt{while} loops, considering the \texttt{break} statement, yields a \texttt{first\_occurrence\_of\_letter\_in\_string(letter, string)} function that implements the same algorithm using tail-recursion instead of an iterative \texttt{while} loop.

```python
def first_occurrence_of_letter_in_string(letter, string):
    def loop(count):
        if string[count - 1] == letter:
            return [[None], count]
        count += 1
        if count <= length:
            return loop(count)
        return [[None], count]
    count = 1
    length = len(string)
    if count <= length:
        result = loop(count)
        count = result[1]
    if count > length:
        return 'No occurrence of {} in {}'.format(letter, string)
    return count
```
2 Methodology

**Transforming Continue** A `continue` statement causes the execution of a loop to continue with the next iteration. Code that is located after a `continue` statement, but still inside the loop body, will be skipped as well.

Continuing with the next iteration (of a loop) is equal to a recursive call. Combining these adaptations with the model of the previous paragraph (see, fig. 3) we obtain an expanded scheme (see, fig. 6).

![Extended transformation scheme, including continue](image)

**Example** This example shows the transformation of a function `double_all_even_numbers(xs)` that implements an algorithm to return the list `xs` with all even numbers multiplied by two:

```python
def double_all_even_numbers(xs):
    count = 0
    while count < len(xs):
        if xs[count] % 2 != 0:
            continue
        xs[count] *= 2
        count += 1
    return xs
```

Applying the methodology to transform `while` loops, considering the `continue` statement, yields a `double_all_even_numbers(xs)` function that implements the same algorithm using tail-recursion instead of an iterative `while` loop.
def double_all_even_numbers(xs):
    def loop(count):
        if xs[count] % 2 != 0:
            count += 1
            if count < len(xs):
                return loop(count)
        return [[None], count]
    xs[count] *= 2
    count += 1
    if count < len(xs):
        return loop(count)
    return [[None], count]

count = None
count = 0
if count < len(xs):
    result = loop(count)
    count = result[1]
return xs
2 Methodology

Transforming Try/Except  Typically there are two types of exceptions that interrupt the normal control flow during the execution of a program: Exceptions, thrown explicitly by the programmer, using the `raise`-statement and exceptions that are being thrown implicitly by an instruction (i.e. invalid mathematical operations).

If there is a `try/except` statement used inside the `body` of a `while`-loop, we need to apply some special modifications because an exception can alter the control-flow. Since we transfer the `body` to the newly created tail-recursive function, an exception is no longer caught inside the original function. To still be able to identify the very function, causing an exception we need to propagate any raised exception from our tail-recursive function to the original enclosing function, where we replaced the loop with the caller.

For this purpose we use `control` accordingly. Every `raise` instruction will be replaced by a `return` with the list consisting of `control` and `vars`. In the currently treated case `control` holds the original statement as a string (e.g. `raise`) and the exception to raise.

To be able to propagate exceptions that are caused implicitly by an instruction, we need to wrap the original `try/except` block within another one. This outer `try` block encloses the original `try/except` block and catches any exceptions; therefore we use the `BaseException` handler, which must be the base class of every exception. We also assign the exception to a new variable (`exc`), so that we can pass the exception in the `control`-list.

Applying these modifications to the model, we obtain a complete transformation scheme for `while` loops, taking into account all control-flow changing statements.

![Diagram](image)

Figure 7: transformation scheme for `while` loops with `try/except` statement inside body
Example  This example shows the transformation of a function `divide(xs, divisor)` that implements an algorithm to return the list `xs` with all numbers divided by divisor, rounded to three decimal places:

```python
def divide(xs, divisor):
    count = 0
    while count < len(xs):
        try:
            xs[count] = round(xs[count]/divisor,3)
            count += 1
        except ZeroDivisionError:
            return 'Dividing {}/{0} is not possible. Division by zero!'.format(xs[count], divisor)
    return xs
```

Applying the methodology to transform `while` loops, considering the `try/except` statement, yields a `divide(xs, divisor)` function that implements the same algorithm using tail-recursion instead of an iterative `while` loop.

```python
def divide(xs, divisor):
    def loop(count):
        try:
            try:
                xs[count] = round(xs[count] / divisor, 3)
                count += 1
            except ZeroDivisionError:
                return [['return', 'Dividing {}/{0} is not possible. Division by zero!'.format(xs[count], divisor)], count]
        except BaseException as exc:
            return [['raise', exc], count]
        if count < len(xs):
            return tail_rec(count)
        return [[None], count]
    count = None
    count = 0
    if count < len(xs):
        result = loop(count)
        control = result[0]
        if control[0] is not None:
            if control[0] == 'return':
                return control[1]
            if control[0] == 'raise':
                raise control[1]
        count = result[1]
    return xs
```
2 Methodology

2.1.3 Placement of the tail-recursive function

Inserting and naming the the tail-recursive function definition that has been created as a part of the while loop transformation can be a very difficult task depending on the implementation language and how its namespace and scope is defined. In the case of Python it is actually not so difficult to find the right place, to insert the function definition.

To explain the placement and naming strategy, the scope of Python is briefly explained in the following paragraph.

Python Scope The Python documentation\(^3\) gives the following execution model and name resolving rules. Variables in Python are resolved using the LEGB-rule, standing for Local $\rightarrow$ Enclosing $\rightarrow$ Global $\rightarrow$ Built-in. These terms are defined as follows:

1. Local: a function definition
2. Enclosing: enclosing function definitions
3. Global: top-level of the executing script itself (also called module)
4. Built-in: names that are reserved by Python itself.

The following program code, will be used in this paragraph to briefly address the name resolving strategy.

```python
i = 0

class Foo:
    i = 1

    def bar():
        i = 2
        def baz():
            return i

        return baz()

print(bar())  # >>> 2

print(i)     # >>> 1

print(i)     # >>> 0
```

In the example code the variable name $i$ is used at three different places. In the global namespace the value 0 is assigned; in line 4 in the namespace of the class Foo the value 1 is assigned; in line 7 in the local scope of the function bar the value 2 is assigned to the name $i$.

\(^3\)Python Execution Model: https://docs.python.org/3/reference/executionmodel.html
2 Methodology

The three print instructions at the end output three different values because the name i is resolved from within different scopes.

Placement Function Definitions (funcdef) belong to the compound statements\(^4\) (if, for, while, try, with, funcdef, classdef) of Python. This means that functions can also be nested inside other functions (inner functions), that are only available in the scope of the enclosing function and do not affect the rest of the program code.

By placing the function at the top of the scope of the transformed while loop the function can only be called and only affects the code inside this very scope, exactly like the while loop to be replaced.

2.1.4 Naming introduced variables

Due to the placement of the function definition, naming the function is also not raising a big issue, because the name only has to be unique to the local scope of the function. Variable names that are introduced to the code by the transformation algorithm, will always be unique to the current enclosing scope, by checking for naming collisions with already existing variable names in this scope. In case of a collision a universally unique identifier (UUID) is appended to the variable name that should be used to further guarantee uniqueness and avoid collisions.

\(^4\)https://docs.python.org/3.3/reference/compound_stmts.html
2 Methodology

2.2 For-Loop Transformation

The for-loop in Python is to be thought of as a foreach loop in other languages, such as Java. Python provides only one type of for-loop, iterating over so called iterator-objects. An iterator-object can be retrieved for any arbitrary data structure class (container), that implements the following two methods:

- **__iter__():** return an iterator object of the container
- **__next__():** get the next item from the container

A container, that implements these methods is also called iterable. Examples for iterables are the builtin data sequence types like list, tuple or range objects.

Insia and Silva propose a methodology that can transform Java’s foreach loop. However to transform Python’s for loop another approach is followed. The assumption is that any for loop can be expressed as a semantically equal while loop.

![Figure 8: Blackbox that transforms any for loop into a semantically equal while loop](image)

In figure 8 a desired model for a transformation of a for loop to a while loop is illustrated. The necessary algorithm to do this transformation is currently a black box that will be peer inside soon.

Taking a look at the operation code (opcode) of an arbitrary Python for loop using the disassembler module\(^5\) of Python, the CPython opcode of the for loop from fig. 8 can be analyzed. To be able to understand the opcode, the most relevant commands are introduced in the following and it is discussed how they affect the program stack:

- **SETUP_LOOP** pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of delta bytes.

- **GET_ITER** implements \(\text{TOS} = \text{iter(TOS)}\) (TOS: top of stack).

\(^5\)https://docs.python.org/3/library/dis.html
2 Methodology

- **FOR_ITER** needs **TOS** to be an iterator. Call its `__next__()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted **TOS** is popped, and the byte code counter is incremented by delta.

- **STORE_FAST** stores **TOS** into the local `co_varnames[var_num]` (set of locally bound variables).

Each line of the opcode is organized into columns. The first column is a reference to the actual source code line number; the second column shows the byte-address (2 bytes are used for each instruction); the third column holds the opcode-name; the fourth column contains the argument of the instruction; and the resolved arguments are in the fifth column (in parentheses).

For line 1 of the original source code, the generated opcode first initializes a block on the stack for the loop (**SETUP_LOOP**, address 0) and then pushes the iterable to the stack at address 2, which the **GET_ITER** instruction (address 4) transforms into an iterator object, pushing it to the stack.

The **FOR_ITER** instruction (address 6) starts the execution of the for loop by checking to see if there exists a next value in the iterator object; if so, it is bound to a variable (**STORE_FAST**, address 8) before finally executing the loops body (The opcode for the loop body is omitted, since it is not necessary to show the actual mechanism of a for loop).

After the last instruction inside the loop body **JUMP_ABSOLUTE** continues the loop at the starting address (6). If the iterator object is exhausted **FOR_ITER** directly jumps to address 12, where the loop block is popped off the stack.

```
1 0 SETUP_LOOP 12 (to 14)
  2 LOAD_CONST 1
  4 GET_ITER
|>>  6 FOR_ITER 4 (to 12)
  8 STORE_FAST 0

2 10 JUMP_ABSOLUTE 6
|>> 12 POP_BLOCK
```

Figure 9: opcode of the for-loop of fig. ; numbers on the left indicate the line number in the source code

There is actually a lot of work done implicitly behind the scenes (**GET_ITER**, **FOR_ITER**), that could also be expressed explicitly in program code.

Analyzing the opcode of a for loop, yields us a flow-chart 10, that shows each individual step of the execution of a for loop.
2 Methodology

Having obtained a flow-chart to model the execution of a `for` loop, it is possible to implement a `while` loop that is semantically equal (see, fig. 11). At first the iterable `<iter>` is converted to an iterator-object and bound to a variable (here: `iterator`). Inside an infinite `while` loop the call to the next function on the iterator-object (`iterator`) and the binding of result to `target_list` is wrapped by a `try` block. As long as the iterator-object is not exhausted the code inside [body] can be executed, otherwise the `StopIteration` exception is caught and the loop is exited directly (`break`).

![Flow-chart of both loop implementations given in figure 11](image)

Figure 10: flow-chart of both loop implementations given in figure 11

![For Loop to While Loop Transformation](image)

Figure 11: for to while-loop transformation
2 Methodology

Similar to the while loop in Python the for loop has an else clause as well. The else clause is entered only if the loops iterator is exhausted. In case the control-flow of the loop execution is altered by break or continue the else clause will not be executed (see, fig. 12).

Since the functionality of the else clause is analogous for both loops, this feature can be addressed by carrying over the else clause of the for loop to the while loop.

First transforming all for loops to equivalent while loops, the transformation rules for while loops, discussed in section 2.1 can be used to transform all loops, consequently.
2 Methodology

2.3 Comprehension to For Loop Transformation

In Python comprehension allows to create new lists, sets and dictionaries of iterables in a concise way. The comprehension syntax is syntactic sugar for the application of map(), filter() and lambdas. However the comprehension syntax contains implicitly executed for loops that should be transformed into an explicit implementation first, in order to remove iteration, replacing it with tail-recursion. In the left box of figure 13 the syntax of a list comprehension can be seen. It consists of an expression (<expr>) and at least one for clause (orange box) together with zero or more if clauses (predicates; blue box), all wrapped by square brackets (list constructor).

<table>
<thead>
<tr>
<th>List Comprehension</th>
<th>For Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>def comp():</td>
</tr>
<tr>
<td>mylist = [&lt;expr&gt; for &lt;elem&gt; in &lt;iterable&gt; if &lt;cond&gt;]*+]</td>
<td>tmp_list = list()</td>
</tr>
<tr>
<td>;</td>
<td>for &lt;elem&gt; in &lt;iterable&gt;:</td>
</tr>
<tr>
<td></td>
<td>if &lt;cond&gt;: *</td>
</tr>
<tr>
<td></td>
<td>tmp_list.append(&lt;expr&gt;)</td>
</tr>
<tr>
<td></td>
<td>return tmp_list</td>
</tr>
<tr>
<td></td>
<td>mylist = comp()</td>
</tr>
<tr>
<td></td>
<td>;</td>
</tr>
</tbody>
</table>

Figure 13: Transformation of list comprehension to for loop(s) [Fou18]

Multiple consecutive for clauses in the list comprehension syntax imply nested for loops. Likewise multiple consecutive if clauses are nested as well. Thus, the list comprehension syntax can be transferred into a new function (comp()), returning the same list. Therefore an empty list (tmp_list) is initialized first. As seen in figure 13 the nested loops are built as follows: After the first for clause, every other if or for clause in the list comprehension syntax creates a new nesting level in the explicit for loop. In the most inner loop the expression (<expr>) is appended to the temporary list (tmp_list). The list comprehension can then be replaced by a call to the newly created function (comp()) [Fou18].

The transformation of sets or dictionaries can be done analogous to the list comprehension, that was shown in an exemplary way in this section.
2 Methodology

2.4 Loops in Generator Functions

Generators are a unique concept to *Python*. Defining a generator is the same as a function definition, except instead of a return statement a generator function needs to have at least one `yield` statement. However, a generator function, can have additional `return` statements inside its definition, that will exit the execution of the generator function, as normally.

A function that is defined in that way returns a *generator* object (or *generator iterator*) when called. A generator iterator is very similar to an iterator object (see 2.2), providing the `__next__()` and the `__iter__()` methods. In fact every generator is an iterator.

As a result, generator functions are an easy way to create an iterator object. To define a generator function, in most cases, loops are used to produce a sequence of values. In figure 14 a generator function is defined that produces an endless stream of integers.

```python
def endless():
    i = 0
    while True:
        yield i
        i += 1
```

Figure 14: Defining an infinite stream of integers, with a generator function

This loops, having a `yield` statement in it, can not be transformed with the methodology, that was presented in section 2. Generator functions are a special concept to *Python* and require us to extend the transformation rules.

While the `yield` statement has to remain unchanged to provide the generator functionality, the recursive

While the `yield` statement has to remain unchanged, to provide the generator functionality, the recursive call to a generator function can not be done using a return statement. Instead *Python* provides some special syntax for this case. To yield a value from a recursive call (or delegate to a subgenerator) the `yield from` statement has to be used.

```
    i += 1
```

Strictly following the transformation scheme, given in figure 3, but using `yield from` to invoke the recursive call the generator function can be transformed, using recursion as well (fig. 15).

It is not necessary in the case of a generator function to return anything (e.g. modified variables), so we can omit the `return` statements.

---

2 Methodology

```python
def endless():
    def tail_rec(i):
        yield i
        i += 1
        if True:
            yield from tail_rec(i)
    i = 0
    if True:
        result = tail_rec(i)
        yield from result
Figure 15: Infinite stream of integers generator, defined with recursion
```
3 Implementation

This section offers insights into implementing the transformation of all loops, with the methodology given in section 2. At first fundamental knowledge about some Python specifics, like the ast module (3.1) as well as the variable scope and lifetime (2.1.3) is provided. To transform the code the abstract syntax tree (AST) of any valid input Python source code is modified. The output of that transformation pipeline is also an AST that does not contain iterative loops. The code-transformation consists of mainly four steps that build a pipeline (3.2).

In section 3.3 every step of this transformation-pipeline, given in figure 24, is explained in detail.

3.1 Python AST

Python offers a built-in module to programmatically inspect and modify the abstract syntax tree (AST) of any Python source code.

We use the ast module to generate the AST of an input program code, apply changes to the AST according to the transformation methodology (see sec. 2) and, as a last step, convert the modified AST back to program code.

In the following we will introduce the most relevant AST nodes, along with its attributes. Further information on the complete syntax grammar of the Python AST is provided in the documentation.

ast.Module

This node is the root of the AST. The ast.Module node has only one attribute (body), which holds a list of nodes that build the program.

\[
\text{ast.Module(stmt* body)}
\]

Figure 16: ast.Module node with its attributes

ast.For

The ast.For node consists of the attributes target, iter, body and orelse. The attribute target holds the variable name (or a tuple of variable names) that the next value of the iterable (attribute iter) will be bound to. The attributes body and orelse hold a list of statements.

\[
\text{ast.For(expr target, expr iter, stmt* body, stmt* orelse)}
\]

Figure 17: ast.For node with its attributes

\[\text{https://docs.python.org/3.6/library/ast.html}\]
3 Implementation

**ast.While**

The `ast.While` node consists of the attributes `test`, `body`, and `orelse`. The attribute `test` holds an expression that evaluates to a boolean. The attributes `body` and `orelse` hold a list of statements.

\[
\text{ast.While(expr test, stmt* body, stmt* orelse)}
\]

Figure 18: `ast.While` node with its attributes

**ast.If**

The `ast.If` node consists of the attributes `test`, `body`, and `orelse`. The attribute `test` holds an expression that evaluates to a boolean. The attributes `body` and `orelse` hold a list of statements.

\[
\text{ast.If(expr test, stmt* body, stmt* orelse)}
\]

Figure 19: `ast.If` node with its attributes

**ast.Try**

The `ast.Try` node consists of the attributes `body`, `handlers`, `orelse` and `finalbody`. The attribute `test` holds an expression that evaluates to a boolean. The attributes `body` and `orelse` hold a list of statements.

\[
\text{ast.Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)}
\]

Figure 20: `ast.Try` node with its attributes

**ast.Assign**

The `ast.Assign` node consists of the attributes `targets`, and `value`. The attribute `targets` holds a list of expressions. The attribute `value` contains a single expression.

\[
\text{ast.Assign(expr* targets, expr value)}
\]

Figure 21: `ast.Assign` node with its attributes
3 Implementation

ast.ListComp

The ast.ListComp node consists of the attributes targets, and value. The attribute targets holds a list of expressions. The attribute value contains a single expression.

```python
ast.ListComp(expr elt, comprehension* generators)
```

Figure 22: ast.ListComp node with its attributes

ast.FunctionDef

The ast.FunctionDef node consists of the attributes name, arguments, body, decorator_list, and returns. The attribute name is an identifier. The attribute arguments contains a list of arguments. The attribute body holds a list of statements. The attribute decorator_list holds a list of expressions and the optional attribute returns contains an expression.

```python
ast.FunctionDef(identifier name, arguments args,
                 stmt* body, expr* decorator_list, expr? returns)
```

Figure 23: ast.FunctionDef node with its attributes
3 Implementation

**AST Traversal**  The built-in `ast` module provides some base classes to traverse, process and modify the nodes of an AST. In this work the open source module `astor`\(^8\) is chosen, publicly available on `github` under the 3-clause-BSD license.

The `astor` module offers a `TreeWalk` superclass, that builds on top of the `ast` module, that allows to walk the AST in arbitrary fashion.

The `astor.TreeWalk` class can be subclassed to create a custom tree-walker. In a subclass methods can be defined to process certain types of nodes (see sec. 3.1), either before (`pre`) or after (`post`) visiting any child node.

In addition the `astor` module is able to generate Python source code from an AST.

### 3.2 Transformation Pipeline

To implement the iterative loop to tail recursive transformation methodology in Python the builtin `ast` module is used, that gives us the possibility to get the AST of any syntactically correct program code. Having the AST of the code it can be modified according to the transformation rules (see, section 2).

The code transformation consists of the following three main steps, that can be categorized into pre-processing steps and the actual transformation to tail recursive functions. These steps need to be applied in sequence and build a transformation pipeline (fig. 24).

- **Pre-processing**
  1. Transform comprehension to for loops
  2. Transform for-loops to while-loops
  3. Initialize variables used inside the loop

- **Loop Transformation**
  4. Transform while loops to tail recursive functions

\(^8\)https://github.com/berkerpeksag/astor
3 Implementation

Figure 24: Transformation pipeline; input is a Python-AST; output is a modified version of the AST
3 Implementation

3.3 Transformation Steps

This section gives details on how each step of the transformation pipeline (see, fig. 24) is implemented.

Notation

In the following the AST nodes and the attributes are used to be able to explain the implementation. An AST node is denoted in the way described in section 3.1. The dot-notation is used to name an attribute of a node. Meaning the test attribute of a `ast.While` node is referenced as `ast.While.test`.

3.3.1 Preprocessing steps

◊ Create scopes dictionary

To be able to introduce new variable names during the processing it has to be certain that this variable name is not already defined in the scope of the transformed loop. As stated in section 2.1.3 a scope in Python can only be a node of the type `ast.Module`, `ast.ClassDef` or `ast.FunctionDef`. Identifying the scope is done by traversing the AST starting at the root (`ast.Module`) and building a dictionary, where the keys are node instances (one of: `ast.Module`, `ast.ClassDef` or `ast.FunctionDef`) and the value is a set of all child nodes. Every node of these three types creates a new key.

This method yields a dictionary where a nodes’ scope can easily be looked up by searching the key of the set the node is contained in.

This step is done before the actual loop transformation happens in the steps ① and ②, because the scope dictionary is not affected by the individual loop transformation but changes after all for-loops have been replaced by while loops.

① Transform List-, Set- and Dict-Comprehension

In Python comprehension allows to create new lists, sets and dictionaries of iterables in a concise way. The comprehension syntax is syntactic sugar for the application of `map()`, `filter()` and `lambdas`. However the comprehension syntax uses implicit for loops (in `map()`) that must be transformed into an explicit implementation first, in order to remove iteration, replacing it with tail-recursion.
3 Implementation

The processing of any `ast.ListComp (ast)` instance consists of the following steps:

(a) Get the node instance that defines the scope of the currently processed `ast.For` node instance by checking the scopes dictionary, that is built in 3.

(b) Having the scope node, all instances of variable names (`ast.Name`) can be searched to ascertain that newly generated names are unique to the current scope. Generate a unique name for the function and the temporary list.

(c) create a new instance of an `ast.FunctionDef` node:
   - `ast.FunctionDef.name`: set to the previously generated function name using the prefix `comp` (comprehension).
   - `ast.FunctionDef.body`:
3 Implementation

- initialize an empty list and assign it to the previously generated helper variable name (using prefix tmp_list) used to temporarily store the list values
- transform the `ast.ListComp` node to a `ast.For` instance:
  * building the body from inside out: starting with the most inner node: expression (`ast.ListComp.elt`); wrapping it inside a call to append to the temporary helper list
  * for each of the reversed list of generators (for clauses) and each if clause contained in the reversed attribute `ifs` a corresponding `ast.For` or `ast.If` node is wrapped around the body
  * each resulting body is the body attribute of the next enclosing `ast.For` or `ast.If` node.
- Insert the function definition built in (c) at the top of the enclosing parent node. In case the parent node is an `ast.Assign` or `ast.Call` instance, the function definition is placed at the top of the scope of the parent node.

(d) replace the currently processed `ast.For` instance with the newly created `ast.While` instance
2 For-to-While Loop Transformation

In this step the transformation rules to replace each `for` loop with a semantically equal `while` loop (see, 2.2) are applied. Therefore the AST is traversed, processing each instance of an `ast.For` node. For nested `for` loops (i.e. other instances of `ast.For` nodes, that occur in the `body` or `orelse` attribute) the order to access these nodes, is not important. Each instance is transformed individually and is not dependent on other enclosing instances.

Figure 26: inspecting step 2 of the transformation pipeline

The processing of any `ast.For` instance consists of the following steps:

(a) Get the node instance that defines the scope of the currently processed `ast.For` node instance by checking the scopes dictionary, that is built in $\Diamond$. 

36
3 Implementation

(b) Having the scope node, all instances of variable names (`ast.Name`) can be searched and to ascertain that newly generated names are unique to the current scope.

(c) create a new instance of an `ast.While` node:

- generate a variable name (`iterator`) to store the iterator-object, the name has to be unique to the local scope of the currently processed `for` loop
- `ast.While.test`: set to the boolean literal `True` (`ast.NameConstant(value=True)`)  
- `ast.While.body`: insert a new `ast.Try` node:
  - `ast.Try.body`: insert a new variable definition (`ast.Assign`), where `target` is set to `ast.For.target` and the `value` attribute is set to a function call to `next()` with the generated variable (`iterator`) as its argument
  - `ast.Try.body`: append the `body` attribute of the `ast.For` node
- Insert a new variable definition (`ast.Assign`), where `target` is set to the generated variable name and value is a function call to `iter()` with the `ast.For.iter` attribute as its value. Place it in the enclosing node right before the current `for` loop instance

(d) replace the currently processed `ast.For` instance with the newly created `ast.While` instance

After these steps have been applied to every `ast.For` instance in the AST, only `ast.While` instances will remain.

3 Initialize Variables used inside the loop

As it comes to variables there is an obvious difference between Python and Java. On the one hand Java is a statically typed language where every variable name must be bound to an an explicit type and an object whose type must match the declared type. Additionally, once the variable name was bound to a type it must not change during the execution time of a program.

In Python, on the other hand, variables are dynamically typed and unless the value is `Null` it is only bound to an object whose type may change during execution time.
During the execution of a loop any variable that is used inside the body of the loop may change its value or even its type as shown above. To keep track of these changes in a recursive function, all variables, being stored inside the loop, must be provided to the recursive function as arguments, so they can be modified and passed on to the next recursive call.

Java, following the concept of statically typed variables, enforces that every variable that may change in a loop-construct has to be declared beforehand, outside of the loop. In Python though, variables can be declared directly inside the loop.

To replace the actual loop, with the caller (see, section 2.1) it has to be ensured that every variable is known to the compiler at execution time of the first call to the tail-recursive function. Therefore the variables that are not already known to the enclosing scope of the loop have to be initialized, at the top of the scope by assigning None to it.

In figure 29 you can see an overview of the necessary actions to build an implementation of this step, being further described in the following:

(a) Get the node instance that defines the scope of the currently processed ast.For node instance by checking the scopes dictionary, that is built in $\diamondsuit$.

(b) collect all variable names: walk the AST starting at the currently processed ast.While instance and processes every occurrence of an ast.Name node. It is checked if the ctx (context) attribute is set to ast.Store, meaning that the variable name is stored inside of the loop. For every ast.Name node, that fulfills this property, the ast.Name.id attribute is stored in a set.

(c) For every variable in this set, a new variable assignment (ast.Assign) is inserted, where the target attribute is set to the constant Null (ast.NameConstant(value=Null)). The insertion has to at the first position of the body of the enclosing scope element.
3 Implementation

Figure 28: inspecting step 3 of the transformation pipeline
3 Implementation

- **Building a dictionary of all while loops**

To be able to tell if a loop instance is nested inside another loop instance or if it is the outermost loop instance, a dictionary that contains all while loop instances has to be built. This dictionary is built by traversing the AST starting at root (`ast.Module`) looking for any node of type `ast.While`, which is set as key. A set is collected with every child node of the key that is also an instance of an `ast.While` node as its value.

This step is only done once before the loop to tail-recursive function conversion.

![Diagram](Image)

Figure 29: inspecting step □ of the transformation pipeline
3 Implementation

3.3.2 Loop transformation

4 While-Loop to Tail-Recursive Functions Transformation

At this stage of the transformation pipeline the AST is prepared to actually perform the conversion of the loops to tail-recursive functions. The AST only contains instances of `ast.While` nodes and the required variables are initialized before the loop code, so the first recursive call can pass them as arguments.

![Diagram of the transformation pipeline](image)

Figure 30: inspecting step 4 of the transformation pipeline
3 Implementation

The conversion of while loops is separated into the following steps. The strategy for multiple nested loops is to consecutively apply these steps to the most inner loop first:

(a) Find all return statements (ast.Return nodes) in the attributes of the currently processed ast.While instance and replace them using the transformation rule proposed in section 2.1.

(b) Get the node instance that defines the scope of the currently processed ast.For node instance by checking the scopes dictionary, that is built in Chapter 3. The scopes have not changed in step 2 and therefore have not been recreated before step 3.

(c) Get all variable names (ast.Name instances) that are being stored (ast.Name.ctx set to Store) inside of the loops body.

(d) With the help of the while loop dictionary, built in Chapter 2, it can be discovered if the currently processed ast.While instance is nested or not, which determines the caller; the caller either propagates its result to an enclosed function (nested loop) or returns the result directly (outermost loop).

(e) All control-flow changing statements (see, 2.1) that occur inside the ast.While loops body are collected.

(f) Building the tail-recursive function of the currently processed loop involves the following steps:
   - At first a new ast.FunctionDef node is created and a unique function name is generated.
   - The function arguments have to be all variables that are being stored inside of the loop (while_vars).
   - The body is built out of the original while-loop body (<while_body>) and a recursive call. The recursive call is wrapped inside an ast.If node that checks the loop condition (test=<while_cond>) first.
   - After that an ast.Return node is appended to the body that returns the control list (see, fig. 3) and the variables (<while_vars>.

(g) To build the caller new variable names (result and control) need to be introduced. If they are already unique to the enclosing scope of the loop these names are used directly, otherwise a unique identifier is appended to the variable name.

(h) Building the caller requires us to know the generated function name, the function arguments ([vars]), the generated result and control variable names, if it is a nested loop, the control statements and the scope, to know if the caller will be placed inside a function definition or in the ast.Module node.

(i) For all the variables gathered in (c) a variable assignment (ast.Assign) is inserted, using the result list.
3 Implementation

(j) In Python functions can be defined everywhere in the code. Now that the `ast.FunctionDef` node is built, the function definition directly can directly be inserted into the parent of the original `while` loop. Thus, the function is directly defined inside the same scope from where it will be called.

(k) The last step is to simply replace the loops node instance (`ast.While`) with the caller.

Example

The performed actions in each step of the transformation pipeline will be shown by reference to the following code example implementing the algorithm to calculate the Levenshtein distance [SM01]. An implementation\(^9\) of the Levenshtein distance algorithm is given in figure 31 using two nested `for` loops.

```python
def levenshtein(s1, s2):
    if len(s1) < len(s2):
        return levenshtein(s2, s1)
    # len(s1) >= len(s2)
    if len(s2) == 0:
        return len(s1)
    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1  # j+1 instead of j since previous_row and current_row are one character longer
            deletions = current_row[j] + 1         # than s2
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row
    return previous_row[-1]
```

Figure 31: Implementation of the Levenshtein distance algorithm

(1) The first step of the transformation pipeline 1 would transform comprehension into explicit `for` loop(s). It does not affect the program code in this example, since this implementation of the Levenshtein distance does not contain any list-, set- or dict-comprehension.

(2) The second step 2 removes all `for` loops, replacing them with equivalent `while` loops. The code after this step is shown in figure 32.

\(^9\)https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#Python
3 Implementation

```python
def levenshtein(s1, s2):
    if len(s1) < len(s2):
        return levenshtein(s2, s1)
    if len(s2) == 0:
        return len(s1)
    previous_row = range(len(s2) + 1)
    iterator = iter(enumerate(s1))
    while True:
        try:
            i, c1 = next(iterator)
            current_row = [i + 1]
            iterator = iter(enumerate(s2))
            while True:
                try:
                    j, c2 = next(iterator)
                    insertions = previous_row[j + 1] + 1
                    deletions = current_row[j] + 1
                    substitutions = previous_row[j] + (c1 != c2)
                    current_row.append(min(insertions, deletions, substitutions))
                except StopIteration:
                    break
            previous_row = current_row
        except StopIteration:
            break
    return previous_row[-1]
```

Figure 32: Levenshtein distance algorithm after transformation step 2 of the transformation pipeline

(3) Transformation step 3 prepares the while loops to be transformed into tail-recursive functions by initializing the used variables in the loop body and condition at the top of the scope of the loop.
def levenshtein(s1, s2):
    c2 = None
    i = None
    c1 = None
    previous_row = None
    current_row = None
    j = None
    deletions = None
    substitutions = None
    insertions = None
    if len(s1) < len(s2):
        return levenshtein(s2, s1)
    if len(s2) == 0:
        return len(s1)
    previous_row = range(len(s2) + 1)
    iterator = iter(enumerate(s1))
    while True:
        try:
            i, c1 = next(iterator)
            current_row = [i + 1]
            iterator = iter(enumerate(s2))
            while True:
                try:
                    j, c2 = next(iterator)
                    insertions = previous_row[j + 1] + 1
                    deletions = current_row[j] + 1
                    substitutions = previous_row[j] + (c1 != c2)
                    current_row.append(min(insertions, deletions,
                                           substitutions))
                except StopIteration:
                    break
            previous_row = current_row
        except StopIteration:
            break
    return previous_row[-1]

Figure 33: Levenshtein distance algorithm after transformation step ③ of the transformation pipeline

(4) Transformation step ④ removes the while loop, replacing it with the caller (see, sec. 2.1) and inserts a corresponding tail-recursive function at the top of the parent element containing the loop.
3 Implementation

def levenshtein(s1, s2):
    def loop1(i, c2, c1, deletions, insertions, j, substitutions, current_row, previous_row):
        try:
            try:
                def loop0(c2, deletions, insertions, j, substitutions):
                    try:
                        try:
                            j, c2 = next(iterator)
                            insertions = previous_row[j + 1] + 1
                            deletions = current_row[j] + 1
                            substitutions = previous_row[j] + (c1 != c2)
                            current_row.append(min(insertions, deletions, substitutions))
                        except StopIteration:
                            return
                        except BaseException as exc:
                            return
                    except StopIteration:
                        return [[None], c2, deletions, insertions, j, substitutions]
                    except BaseException as exc:
                        return [[‘raise’, exc], c2, deletions, insertions, j, substitutions]
                    if True:
                        return loop0(c2, deletions, insertions, j, substitutions)
                    return [[None], c2, deletions, insertions, j, substitutions]
        i, c1 = next(iterator0)
        current_row = [i + 1]
        iterator = iter(enumerate(s2))
        if True:
            result = loop0(c2, deletions, insertions, j, substitutions)
            control = result[0]
            if control[0] is not None:
                if control[0] == ‘raise’:
                    raise control[1]
                c2 = result[1]
                deletions = result[2]
                insertions = result[3]
                j = result[4]
                substitutions = result[5]
                previous_row = current_row
            except StopIteration:
                return [[None], i, c2, c1, deletions, insertions, j, substitutions, current_row, previous_row]
        except BaseException as exc:
            return [[‘raise’, exc], i, c2, c1, deletions, insertions, j, substitutions, current_row, previous_row]
        if True:
            return loop1(i, c2, c1, deletions, insertions, j, substitutions, current_row, previous_row)
        return [[None], i, c2, c1, deletions, insertions, j, substitutions, current_row, previous_row]
    # continues on next page
3 Implementation

```python
previous_row = None
current_row = None
substitutions = None
j = None
insertions = None
deletions = None
c1 = None
c2 = None
i = None
if len(s1) < len(s2):
    return levenshtein(s2, s1)
if len(s2) == 0:
    return len(s1)
previous_row = range(len(s2) + 1)
iterator0 = iter(enumerate(s1))
if True:
    result1 = (loop1(i, c2, c1, deletions, insertions, j, substitutions, current_row, previous_row))
    control1 = (result1[0])
    if control1[0] is not None:
        if control1[0] == 'return':
            return control1[1]
        if control1[0] == 'raise':
            raise control1[1]
i = result1[1]
c2 = result1[2]
c1 = result1[3]
deletions = result1[4]
insertions = result1[5]
j = result1[6]
substitutions = result1[7]
current_row = result1[8]
previous_row = result1[9]
return previous_row[-1]
```

Figure 34: Levenshtein distance algorithm after transformation step ④ of the transformation pipeline
4 Related Work

Considering the close relationship between iteration and tail recursion, there exists a lot of work studying the conversion from one to another. Due to the fact that, in general, compilers have optimized code for iteration the transformation from recursion to iteration has been of higher interest and is therefore better covered than vice versa.

The Transformation from iteration to tail recursion is also of big interest in the transformation of recursion to tail recursion. Even if recursion and tail recursion share the same intuition, they have fundamental differences and the transformation is not trivial at all. For this purpose a concept called incrementalization can be used to transform recursive functions to loops first [Liu00], which then can be transformed to tail-recursive functions.

Besides the method that is covered in this thesis, there exist some other approaches to transform iteration to recursion. Currently, as already stated in the introduction (sec. 1.1), transforming iterative loops to tail recursive functions is also ongoing work of the chair for Database Systems Research (University of Tuebingen). One interesting approach is the method of Yi et al., who have done research on the performance of recursion in multi-level memory hierarchies. Therefore they came up with a method to transform loops to recursion, using a transformation technique called *iteration space slicing* (see, [PR97]). Iteration space slicing applies *transitive dependence analysis* on the dependence graph to compute the instances of a particular statement that must precede or follow a given set of instances of another statement. Yi et al. stated that this is a powerful technique, which can contribute to compiler optimization [YAK00].

The method of Insa and Silva covered in this thesis, however, acts on the implementation level and follows transformation rules that involve building a recursive call around the original loop code, that is wrapped into a new function, the actual loop-code is replaced by a recursive function.
5 Conclusion

Automatic transformation of iterative to tail recursive functions is a topic from high interest in programming language theory. Having an automated translator from iterative loops to tail recursion can be beneficial for several purposes. It enables that algorithms can be written in their most intuitive way, may it be declarative or recursive. Another use case for this transformation is a technique for declarative debugging, where transforming loops into tail recursive functions can improve the interaction between the programmer and the debugger. This technique has been published by Insa and Silva in 2012 [IST12]. The groundwork for this thesis is another article of Insa and Silva [IS14], in which they provide transformation rules for automatic loop transformation using Java programming language.

This thesis had the goal to prove their claim, that the given transformation rules are general enough to be adapted to other programming languages and to provide insights to the implementation process of the given methodology.

The outcome of this thesis is a modified methodology, which adapts to the specifics of the Python programming language. To verify that the implementation applies the methodology correctly the implementation comes with multiple unit test cases. Also it is possible to transform the implementation by self-application and to successfully apply the unit tests to the transformed implementation.

Additionally it was possible to provide new transformation rules for the transformation of concepts, that are unique to Python, such as generator functions and list-, set- and dict-comprehension.

Alongside with the methodology comes an implementation of automatic loop transformation, with a description that may serve as a guide for the implementation in any other multi-paradigm programming language.

Optimizations The implementation of automatic loop transformation that has been developed in the course of this thesis is able to translate the whole Python programming language and output semantically equivalent program code for any input program code. However the generated output code is not optimal and leaves scope for improvement regarding the following points:

- The translation of for to while loops creates a lot of unnecessary code in the caller (see, sec. 2.2). The generated loop condition in this case is just the boolean value True, producing a lot of if True: clauses in the code. They may be completely removed from the code by replacing them with the statements in their body.

- In step 3 of the transformation pipeline, all variable names inside the while loops body and condition are initialized (assigning None at the top of the scope). This step can be improved by only initializing those variables, not being assigned any value before the while loop.
5 Conclusion

- Executing the generated Python source code, may lead to a stack overflow, since Python does not implement tail-call optimization. However this is not a drawback in perspective to this thesis, because the main focus was the methodology and to give insights to the implementation.

Future Work  Given the proposed transformation methodology (sec. 2) and the changes and extensions to adapt to Python programming language and the implementation of a working automatic loop transformation in Python opens some new topics that can be investigated in the future.

The fact that the current official CPython implementation does not implement tail call optimization (TCO)\textsuperscript{10} leads to a worse performance of the generated code not using loops. Hence it would be interesting to implement automatic loop transformation in another multi-paradigm language that supports TCO to see how performance is actually affected in a better suiting environment that Python offers for this use case.

\textsuperscript{10}Guido van Rossum (author of Python) gives a comprehensive argumentation in a blogpost: \url{http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html}
References


Appendix

Using of the Python Implementation

The implementation of Automatic Transformation of Iterative to Tail-Recursive Functions in Python can be found in the project root folder named `i2r_python`, alongside with that comes a detailed documentation in the `docs` folder. After setting up the Python environment according to the documentation, the tool is ready to be used.

Here are the basic commands to use the implementation in a tool that is able to convert a complete project, creating a copy of the project in another location by using command line:

Creating a copy of the project on the same directory level with the suffix _tr:

```bash
>>> itor_dir.py -d <project-root>
```

Creating a copy of the project on at target-root:

```bash
>>> itor_dir.py -d <project-root> -t <target-root>
```

Unit Testing the Implementation

The correctness of the implementation is verified by unit tests that can be invoked directly in the project root by using the `pytest` testing tools. Using the `pytest` command:

```bash
>>> pytest [-vv] [--capture=sys]
```
Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe.

Tübingen, den 25.07.2018