UDC 004.4'24

IMPROVING PERFORMANCE OF PYTHON CODE USING REWRITING RULES TECHNIQUE

Kostiantyn Zhereb[0000-0003-0881-2284]

Taras Shevchenko National University of Kyiv

Python is a popular programming language used in many areas, but its performance is significantly lower than many compiled languages. We propose an approach to increasing performance of Python code by transforming fragments of code to more efficient languages such as Cython and C++. We use high-level algebraic models and rewriting rules technique for semi-automated code transformation. Performance-critical fragments of code are transformed into a low-level syntax model using Python parser. Then this low-level model is further transformed into a high-level algebraic model that is language-independent and easier to work with. The transformation is automated using rewriting rules implemented in Termware system. We also improve the constructed high-level model by deducing additional information such as data types and constraints. From this enhanced high-level model of code we generate equivalent fragments of code using code generators for Cython and C++ languages. Cython code is seamlessly integrated with Python code, and for C++ code we generate a small utility file in Cython that also integrates this code with Python. This way, the bulk of program code can stay in Python and benefit from its facilities, but performance-critical fragments of code are transformed into more efficient equivalents, improving the performance of resulting program. Comparison of execution times between initial version of Python code, different versions of transformed code and using automatic tools such as Cython compiler and PyPy demonstrates the benefits of our approach – we have achieved performance gains of over 50x compared to the initial version written in Python, and over 2x compared to the best automatic tool we have tested.

Keywords: improving code performance, rewriting rules technique, Python, Cython.

Introduction

Python is now becoming increasingly popular in various areas of software development. In particular, this language is used in scientific computing, artificial intelligence and machine learning, for the collection and analysis of large amounts of data, in the development of web applications, to automate user actions and in other areas [1]. The advantages of Python are ease of learning, speed of writing code, the presence of many libraries and frameworks with support for extensive functionality. However, a significant disadvantage of this language is the performance of code execution [2]. Because Python is interpreted (scripting) language in many of its implementations, code execution time is much longer compared to the language that is compiled into machine code (such as C++ or Fortran) or bytecode or similar intermediate representation (such as Java or C#). Therefore, for tasks where performance is important, the Python language can be used to create prototypes, which are then rewritten using more efficient languages. Another approach is to use Python language extensions, which allow using a more efficient language (such as C / C++) to implement performance-critical snippets of code that are then called from other pieces of code written in Python. This approach allows striking a balance between runtime efficiency and programmer productivity.

Implementing Python extensions using C++ manually can be quite difficult for developers – as it requires detailed knowledge of both languages, the use of special interfaces and writing significant amounts of boilerplate code. There exist the tools to simplify the development of more efficient Python programs. In particular, the Cython language [3,4] allows to write efficient code with a syntax close to the Python language, and at the same time the execution efficiency is close to the C++ language. Also, using Cython makes it easy to connect components or extensions written in C++ to Python. However, the developer still has to learn a new language, and take into account the peculiarities of its implementation. There are also automatic tools to increase the efficiency of code execution in Python. In particular, a very popular tool is PyPy [5] – an alternative implementation of the Python language that uses JIT-compilation instead of interpretation. Using this tool allows to execute existing code more efficiently without requiring changes or the use of other languages [2,6]. However, the speed of code execution using PyPy is still inferior to the speed of code written using more efficient programming languages [7].

This paper proposes an approach for semi-automated conversion of Python code snippets into functionally equivalent Cython or C++ code snippets in order to increase code execution efficiency. This approach uses the technique of rewriting rules, in particular the Termware system, as well as the approach of building high-level algebraic code models..

1. Rewriting rules technique and Termware system

In this paper, the technique of rewriting rules implemented in the TermWare system is used to automate program code transformations [8, 9]. This allows to describe the conversion of programs in a declarative form, which simplifies the development and reuse of such transformations.

Rewriting rules are described using Termware syntax and executed using Termware tools. The basis of the language are terms, i.e. tree-like expressions of the form $f(x_1,...,x_n)$. Here x_i are either atomic terms (constants or variables written as $var) or other terms. Termware rules have the following general form:

source [condition] -> destination [action]

Four components of the rule are:

- *source* – input pattern;
- *destination* – output pattern;
- *condition* – condition (guard) that determines if the rule can be applied;
- *action* – imperative action performed when the rule is executed.

The action to perform and the conditions to check are optional components of the rule. They can call the imperative code contained in the fact database [8]. Due to this, the basic model of rewriting terms can be expanded with arbitrary additional features.

When using the technique of rewriting rules for program code transformation, it is necessary to convert the code from a text representation into a tree-like structure, which can be written using terms. To do this, first we use a parser (syntax analyzer) of the programming language. As a result, we get a parsing tree, which can be considered as a low-level syntactic code model. This model is not very convenient to use, in addition, the model depends on the programming language and implementation of the parser. Therefore, this paper uses the approach of transforming this model to a high-level algebraic code model [10, 11]. The high-level model is smaller in size and more convenient to process. It also does not depend on the programming language, so it can be used to switch to other programming languages. The transition from a low-level syntactic model to a high-level algebraic model is semi-automated using rewriting rules. After processing and transformations of the high-level model, the code is generated from it in the target language (which may or may not coincide with the original language), for which a code generator of the corresponding language is used [12, 13].

2. Building a low-level syntax code model

As an example of code in Python, consider the implementation of a simple algorithm for finding prime numbers [14]. The source code looks like this:

```
def primes_python(nb_primes):
    p = []
    n = 2
    while len(p) < nb_primes:
        for i in p:
            if n % i == 0:
                break
        else:
            p.append(n)
        n += 1
    return p
```

The code consists of two nested loops. The outer while loop is responsible for checking consecutive numbers n until the total number of prime numbers found exceeds the nb_primes parameter. The internal for loop checks whether the specified number n is divisible by previously found prime numbers (stored in the list p).

This code demonstrates the capabilities of Python, which differ from the basic capabilities of other languages, including C-like languages (e.g., C++, Java, C #, …). The basic for loop in Python is not a loop with a counter, but a loop of type "for-each", which goes through all the elements of a given list. Similar loops are available in other modern programming languages, but there they are additional to counter loops, whereas in Python this is the basic loop design. Another feature of the Python language is the else block in the for loop. This block is executed if all iterations of the corresponding cycle were executed completely, i.e. were not exited by the break statement. Many other programming languages do not have similar capabilities. Such features of programming languages complicate the automatic or semi-automated conversion of code from one language to another.

Based on the source code of the program, a low-level syntactic code model is built. This uses a Python parser. For this example, the low-level syntactic model is as follows:

FunctionDef(name='primes', args=arguments(args=[arg(arg='nb_primes', annotation=None)], vararg=None, kwonlyargs=[], kw_defaults=[], kwarg=None, defaults=[],),
body=[Assign(targets=[Name(id='p', ctx=Store())], value=List(elts=[], ctx=Load()),),
Assign(targets=[Name(id='n', ctx=Store())], value=Num(n=2),),

```
While( test=Compare( left=Call( func=Name(id='len', ctx=Load()), args=[Name(id='p', ctx=Load())], keywords=[], ), ops=[Lt()], comparators=[Name(id='nb_primes', ctx=Load())], ),
  body=[ For( target=Name(id='i', ctx=Store()), iter=Name(id='p', ctx=Load()), body=[
    If( test=Compare( left=BinOp( left=Name(id='n', ctx=Load()), op=Mod(), right=Name(id='i', ctx=Load()), ), ops=[Eq()], comparators=[Num(n=0)], ), body=[Break()], orelse=[],), ],
   orelse=[Expr(value=Call(func=Attribute( value=Name(id='p', ctx=Load()), attr='append', ctx=Load(), ), args=[Name(id='n', ctx=Load())], keywords=[],),),], ),
  AugAssign( target=Name(id='n', ctx=Store()), op=Add(), value=Num(n=1),),],orelse=[],),
 Return(value=Name(id='p', ctx=Load()),),], decorator_list=[],returns=None,0)
```

The constructed model is quite large in volume. It contains many details that can be useful in the general case – but for this particular algorithm are not very necessary. Therefore, it is difficult to work with such a model.

To build a model that is more suitable for further processing and transformation, it would be possible to implement a specialized parser that would generate only the data that will be needed in the future. However, this means that it will be necessary to develop a specialized parser, which is quite a challenge. It may also be necessary to build a new specialized parser many times, as different source code data is required to solve different problems.

Therefore, a more universal approach is used in this paper. This approach consists in the automated transformation of a low-level syntactic model into a high-level algebraic model suitable for solving this problem. To automate transformations, the technique of rewriting rules is used, in particular special rules – patterns [12]. In the general case, the pattern consists of two rules r_p and r_g. The r_p rule is applied to a low-level syntactic model and replaces certain of its constructions with higher-level analogues. The r_g rule works in the opposite direction – it reproduces the elements of a low-level model from the constructions of a high-level algebraic model. In most cases, the patterns have the form $t_L \leftrightarrow t_H$, i.e. a direct correspondence is established between the elements of the low-level model t_L and the high-level model t_H. In this case, the corresponding rules have the form $r_p = t_L \rightarrow t_H$ and $r_p = t_H \rightarrow t_L$.

As an example of the use of patterns, consider some of the constructions in the given code example. Let's start with the frequently used assignment operator:

1. Assign(targets=[Name(id=$name, ctx=Store())], value=$value) <-> Assign(Var($name), $value)

This pattern finds a special case of assignment – when there is a variable in the left part, and only one assignment is used (i.e. an expression like x = a as opposed to x = y = a).

Patterns for standard arithmetic operations and comparisons are described in a similar way:

2. AugAssign(target=Name(id=$name, ctx=Store()), op=Add(), value=Num(n=1)) <-> Increment(Var($name))
3. BinOp(left=$op1, op=Mod(), right=$op2) <-> Mod($op1, $op2)
4. Compare(left=$op1, ops=[Lt()], comparators=[$op2]) <-> Less($op1, $op2)
5. Compare(left=$op1, ops=[Eq()], comparators=[$op2]) <-> Equal($op1, $op2)

Pattern 2 describes the operation of increasing the variable by 1 (increment). Python does not have a special increment operator (unlike many languages that support n++ syntax). Instead, the usual assignment n + = 1 is used. However, the technique of rewriting rules makes it possible to identify such special cases. In the future, this makes it possible to switch to another programming language. It can also be useful for recognizing certain standard algorithms. Pattern 3 describes a binary operator of division remainder (modular division), which in many programming languages has the syntax a% b. Other arithmetic operations are described by similar patterns. Patterns 4 and 5 describe the comparison operations a <b and a == b. The low-level model supports comparisons with more than two operands, such as a <b <c or a == b == c. But such comparisons are not very common, so in a high-level algebraic model they are considered as a special case. This simplifies the structure of the model in more popular cases.

Next, we consider patterns for variables and constants of basic data types:

6. Name(id=$name, ctx=Load()) <-> Var($name)
7. Num(n=$value) [isInteger($value)] <-> Integer($value)

Pattern 6 works for cases where the variable is used to obtain a value, but its value does not change. Situations in which the value of a variable is modified are described by patterns like 1 and 2. Pattern 7 describes integer constants. The low-level Python code model does not distinguish between integers and real numeric constants. Therefore, to convert to a high-level algebraic model, the isInteger ($value) check is used, which calls the corresponding fact database method [8]. Similar patterns are used for other basic types.

Python also supports several standard container types. In particular, lists are used in this example. The use of patterns allows to describe the basic operations for working with these types of data:

8. List(elts=$items, ctx=Load()) <-> List($items)
9. Call(func=Var('len'), args= [$list], keywords=[]) <-> Length($list)

10. Expr(value=Call(func=Attribute(value= $list, attr='append', ctx=Load()), args=[$item], keywords=[])) <-> Append($list, $item)

Pattern 8 describes the creation of a list with a fixed set of initial values. This example creates an empty list, but the pattern also supports a more general case. Pattern 9 describes the operation of obtaining the length of the list. Pattern 10 describes the operation of adding one element to the end of the list. Patterns 9 and 10 demonstrate that some operations can be implemented differently in the target programming language – the length of the list in Python is obtained by calling the global function len ($ list), and adding an element to the end of the list – calling the method $ list.append ($ item). The high-level algebraic model allows to abstract from such details of implementation that are properties of different programming languages. This simplifies working with the high-level model, as well as allows to switch to other programming languages.

Consider patterns for standard control structure, namely conditions and cycles:

11. If(test=$condition, body=$then, orelse=[]) <-> If($condition, $then)
12. While(test=$condition, body=$body, orelse=[]) <-> While($condition, $body)
13. For(target=Name(id=$name, ctx= Store()), iter=$iterable, body=$body, orelse=$else) <-> IfNoBreak(ForEach(Var($name), $iterable, $body),$else)

Pattern 11 describes the conditional construction if without the else block. Pattern 12 describes a loop with a condition of type while. Pattern 13 describes a loop of type for-each, which traverses all elements of a given container and executes a loop body for each of these elements. This pattern implements support for the else block, a characteristic feature of Python loops. To support such a construction, a new term IfNoBreak($loop, $else) was introduced in the high-level algebraic model. An alternative to this approach would be to create extended constructs for loops that support additional parameters. However, the proposed approach with the implementation of a single structure allows to support different types of loops (while, for, for-each,…) using only one new structure, rather than extending each of the cyclic structures. Also, this approach simplifies the implementation of this design in other programming languages, which usually do not support the block of type else in loops.

Finally, consider the patterns for working with functions, namely to describe the functions in the program code:

14. arg(arg=$name, annotation=None) <-> Arg($name)
15. FunctionDef(name=$name, args= arguments(args=$args, vararg=None, kwonlyargs=[], kw_defaults=[], kwarg= None, defaults=[]), body=$body) <-> Function($name,$args, $body)

Pattern 14 describes the individual arguments of the function. Pattern 15 describes the definition of a function as a whole, in the simple case where there are no special types of arguments (support for a variable number of arguments, default values).

The described patterns are applied to a low-level syntactic model of code in the direction of generating a high-level model – that is, the corresponding rule r_p is selected from each pattern, and a set of these rules is applied to the term describing the low-level model. As a result, the following term was obtained:

```
Function("primes",
  [Arg("nb_primes")], [
  Assign(Var("p"),List([])), Assign(Var("n"),Integer(2)),
  While(Less(Length(Var("p")), Var("nb_primes")),[
    IfNoBreak( ForEach( Var("i"), Var("p"), [
      If(Equal( Mod(Var("n"), Var("i")), Integer(0)), [Break()])
    ]), [Append(Var("p"),Var("n"))]),
    Increment(Var("n"))
  ]) ,
  Return(Var("p"))
])
```

The obtained high-level algebraic model is much smaller in volume compared to the low-level syntactic model. The algebraic model is closer to the source code. In this case, it describes an algorithm that does not depend on the implementation language. Therefore, this algorithm can be converted into implementation in other programming languages.

It is worth noting the following feature of these patterns. Some of them use terms with the same names on both sides. For example, this applies to pattern 1 (term Assign), pattern 8 (term List), pattern 11 (term If), pattern 12 (term While). In the general case of using the technique of rewriting rules, the presence of terms with the same names on both sides of the rule can cause problems. After applying the rule, the same rule can work again, which will "loop" the rewriting process. To avoid this problem, the following modification is implemented: if there are terms with the same

name and arithmetic (number of arguments) in the pattern on both sides, the _pat modifier is first added to the converted term, which is then removed by a separate rule. Thus, pattern 1 will be implemented as the following rules:

16. Assign(targets=[Name(id=$name, ctx=Store())], value=$value) -> Assign_pat(Var($name), $value)
17. Assign_pat($op1,$op2) ->Assign($op1, $op2)

Rule 16 is used during the first stage of transforming a low-level syntactic model into a high-level algebraic model. Rule 17 is launched separately after all the rules of the first stage have been worked out. Thus, when rule 17 works, rule 16 is no longer active. This avoids the potential endless application of such rules.

3. Extension of the algebraic model with additional data

The constructed algebraic model completely describes the source code in the Python language from which it was generated. However, to work with the model, we need other data that was not explicitly presented in the source code of the program. In particular, this applies to data types.

Python traditionally uses dynamic typing, so source code does not describe variable types, function arguments, and so on. In recent versions of Python, it has become possible to specify data types for individual variables and function arguments [15]. However, data types are optional, and their absence does not cause compilation errors. Therefore, many Python projects do not specify data types – especially code written some time ago (support for type annotations appeared in Python version 3.5, which was released in September 2015).

Many other programming languages require a programmer to specify data types for all variables, function arguments, class fields, and so on. In particular, this applies to the C++ language. Cython can work with any Python code, so descriptions of data types in this language are optional. However, static typing is essential to increase the performance of Cython code. Therefore, the ability to add to the algebraic model information about the data types of its individual elements will generate more efficient code.

Information about the types of these variables can be obtained from examples of their use. To do this, we use the following rewriting rules:

1. Assign(Var($name), Integer($value)) -> _same_ [saveType($name, Integer)]
2. Assign(Var($name),List([])) -> _same_ [saveType($name, List_partial(Unknown))]

Rule 1 specifies the type of the variable to which the integer value is assigned, in which case the saveType method of the fact database is used to assign the Integer type to the variable. Rule 2 works similarly, but for the data type list. In this case, the variable is assigned an empty list, so it is impossible to determine the type of elements at this stage. Because of this, the type List_partial (Unknown) is written. The type of list items must be specified by the following rules. Similar rules exist for other data types.

Rules 1 and 2 use the special term _same_. It is used in the cases when the rule does not change the model directly, but only adds information to the database of facts. When processing a rule in the right part of which is the term _same_, this term is automatically replaced by a term similar to the left part of the rule, but with the addition of the modifier _same. This modifier is then removed in the next step of applying the rules. Thus, rule 1 becomes the following rules:

3. Assign(Var($name), Integer($value)) -> Assign_same(Var($name), Integer($value)) [saveType($name, Integer)]
4. Assign_same($op1, $op2) -> Assign($op1, $op2)

Rule 3 applies only once to each element of the model, because in its right part there is already a modified term. Rule 4 corrects the modification, but it works in the next step, when rule 3 is no longer active.

In order to specify the type of list items, we use the following rule:

5. Append(Var($list), $item) [getType($item,$type)] -> _same_ [saveType($list, List($type))]

Rule 5 finds elements of the model that describe adding values to the list. Next, the getType fact database method is called, which writes the value type to $type. The list type is then updated.

Once the list type is defined, it is possible to determine the type of another variable that is used to traverse the list. The following rule is used for this purpose:

6. ForEach(Var($item),Var($list)) [getType($list, List($type))] -> _same_ [saveType($item, $type)]

Rule 6 finds "for-each" loops for which the list type is known, and determines the type of the variable based on the list type. It should be noted that rule 6 will not work after rule 2 works, because the full type of list is not yet known. For this purpose in rule 2 the type List_partial is set, instead of simply List.

Among the variables present in the model, the argument type of the nb_primes function remains unknown. The type of this variable can be determined in two ways: based on function calls, or based on the use of this value in the function itself. The second approach is used in this work. The advantage of this approach is the possibility of independent analysis of individual functions, which greatly simplifies the transformation. A possible drawback is the lack of accuracy – because the function can transmit data of more general types than can be inferred based on the usage. For example, this case uses a comparison of the variable nb_primes with the size of the list, and from this we can conclude that the type of variable should be the same as the type of size (i.e. unsigned integer). However, there may be a situation where the variable has a different type, such as a real value – and the comparison will still be correct. One of the areas of further research is the implementation of both approaches and their comparison.

The following rule is used to implement the definition of the type of a variable based only on its use in the body of the function:

7. Less(Length($list), Var($len)) -> _same_ [saveType($len, UnsignedInteger)]

Rule 7 looks for a comparison of a certain variable with the length of the list, and assigns this variable the type UnsignedInteger. Similar rules can be applied to other comparison operators.

After applying all the rules in the fact database, information about the types of variables in this function is stored (see Table 1).

Table 1. Types of variables in the model

Variable name	Type in the Facts DB
n	Integer
p	List(Integer)
i	Integer
nb_primes	UnsignedInteger

In addition to defining variable types, rewriting rules can add other data to the model. In particular, in this example it is useful to know the expected size of the list p. By knowing the expected size of the list, we can reserve enough memory in advance to store all the items. This will avoid copying items as the list size increases. In the general case, determining the maximum list size may require building complex code models. But to get simple estimates, we can use simple rules that work in some cases. Because an incorrect estimate of the size of the list does not make the implementation incorrect, but only affects the performance of the code, such rules can be useful. In particular, in this case the following rule is used:

8. Less(Length($list), $val) -> _same_ [saveMaxLength($list, $val)]

This rule seeks the comparisons of the length of a particular list $list with the value $val and stores this value in the fact database. If such comparisons occur several times, the saveMaxLength method tries to determine the maximum value and uses it.

Thus, the technique of rewriting rules allows to expand the model of the program, adding to it the additional information that can be found in the model or derived from already known data. This allows to further build more efficient program conversions.

4. Convert code to other programming languages: Cython and C++

Based on the constructed high-level algebraic model of the program, it is possible to generate the code of the equivalent program in other languages. This paper discusses the Cython and C++ languages, because code snippets in these languages can be easily integrated into Python code. This allows to convert only small fragments of code that are most performance critical.

In simple cases, the structure of the high-level model corresponds to the code structure of the target language, so to convert it is only necessary to generate the appropriate structures of the target language for each element of the model. However, if the initial and target language are different enough, it is necessary to make some changes in the structure of the model. In particular, in case of transition from Python to Cython or C++ it is necessary to add the declaration of types of variables, and also more detailed initialization, in particular for lists. This can be done using the following rewriting rules:

1. Function($name, $args, $body) -> Function_todo($name,TypedArgs($args,$name), [DeclareVars($name) : $body])
2. TypedArgs([$arg1: $args],$func) -> [TypedArgs($arg1,$func): TypedArgs($args,$name)]
3. TypedArgs(Arg($name),$func) [getType($name,$func,$type)] -> Args($name,$type)
4. DeclareVars($name) [getTypedVars($name, $declarations)] -> $declarations
5. Assign($list,List([])) [getMaxLength($list,$len)] -> ReserveLength($list, $len)

Rule 1 adds TypedArgs service terms to populate information about function argument types, and DeclareVars to declare local variables. Rules 2-3 describe the extension of the term TypedArgs. Rule 2 describes how this term is consistently applied to the elements of the argument list. Rule 3 demonstrates adding to the argument information about the types extracted from the fact database. Rule 4 describes the creation of a declaration of variables based on data from a database of facts. Rule 5 describes the additional initialization of lists – instead of creating an empty list, which will then grow multiple times, the memory is immediately reserved for a certain number of elements, which is also taken from the database of facts.

After applying these rules, the model of this function is as follows:

```
Function("primes",
 [Arg("nb_primes", UnsignedInteger)], [
 Declare(Var("n"), Integer),  Declare(Var("i"), Integer),  Declare(Var("p"), List(Integer)),
 ReserveLength(Var("p"), Var("nb_primes")),
 Assign(Var("n"),Integer(2)),
 While(Less(Length(Var("p")), Var("nb_primes")),[
  IfNoBreak( ForEach( Var("i"), Var("p"), [
   If(Equal( Mod(Var("n"), Var("i")), Integer(0)), [Break()])
  ]), [Append(Var("p"),Var("n"))]),
  Increment(Var("n"))
 ]) ,
 Return(Var("p))
])
```

The high-level model has undergone minor changes due to the need to support new languages. However, if there is a need to generate code in Python, these additional features can be either ignored or used to generate annotations of types, comments, or other code constructs that are not used when compiling and executing code, but make the code more understandable to the developer. Thus, the high-level algebraic model still remains independent of the programming language, i.e. there is no need to support different types of models for different languages (even if the capabilities of these languages differ significantly, in particular languages with dynamic typing like Python, languages with optional static typing like Cython sample and languages with mandatory static typing such as C++).

Using the extended high-level model, the code of the corresponding function can be generated in Cython. This code looks as follows:

```
def primes(unsigned int nb_primes):
    cdef int n, i
    cdef vector[int] p
    p.reserve(nb_primes)
    n = 2
    while p.size() < nb_primes:
        for i in p:
            if n % i == 0:
                break
        else:
            p.push_back(n)
        n += 1
    return p
```

The general structure of the code is similar to the source code in Python. This makes the code more understandable for developers who know Python. However, in the process of code generation, some constructs received a different implementation (see Table 2).

Table 2. Elements of the model with different implementations in Python and Cython

Model element	Implementation in Python	Implementation in Cython
Length(Var("p"))	`len(p)`	`p.size()`
Append(Var("p"),Var("n"))	`p.append(n)`	`p.push_back(n)`

The same high-level model is used to generate C++ code, although the text representation of the corresponding code elements is different. The generated function code is as follows:

```
std::vector<int> primes(unsigned int nb_primes){
    int n, i;
```

```
    std::vector<int> p;
    p.reserve(nb_primes);
    n = 2;
    while (p.size() < nb_primes) {
        bool found = false;
        for(int i:p) {
            if (n % i == 0) {
                found = true;
                break;
            }
        }
        if (!found) {
            p.push_back(n);
        }
        n += 1;
    }
    return p;
}
```

The structure of the code is similar to the implementation on Cython. However, there is a difference caused by the lack of support for the `else` block in the `for` loop. The corresponding element of the IfNoBreak model is implemented in C++ using the additional logical variable `bool found`, which is false after exiting the loop, only if the break statement did not work in the body of the loop.

In addition to generating the function code in the appropriate languages, the code generator also supports additional settings. In particular, code file extensions are configured for each language, namely .pyx for Cython and .h for C++. Also, to use C++ code from a Python program, an additional .pyx file is generated that describes the corresponding functions.:

```
cdef extern from "primes.h":
  vector[int] primes(unsigned int nb_primes)
```

Thus, the generated code in Cython or C++ can be used directly with Python programs. This allows to convert only the most performance-critical pieces of code, leaving most of the program code unchanged.

5. Experimental verification of the approach

To verify the proposed approach, measurements of the execution time of different versions of one program for calculating prime numbers were performed. The initial version of the program in Python (this version is marked Python) was considered, as well as two converted programs – in Cython (marked Cython) and C++ (marked CPP). Also for comparison, two approaches to automatically increase the performance of Python code are considered. The first of these approaches is to compile code in Python using Cython tools (this version is labeled Compiled). The second approach uses the JIT compiler PyPy (this version is labeled PyPy).

All measurements were performed on a personal computer (Intel Core i7-8550U, 16 GB RAM, Windows 10). The execution time of the primes function was measured depending on the number of prime numbers to be found (argument of the nb_primes function), for values from 1000 to 16000 with a step of 1000. The measurement results are shown in Fig. 1-3.

Figure 1 shows the dependence of execution time (in seconds) on the number of primes for different versions of the program. As can be seen from the measurement results, the initial version of Python is the slowest. The Compiled version is a bit faster (1.5-2 times). All other versions are much faster. Therefore, they are shown in Fig. 2, which shows the same dependence of execution time on the number of primes, but with a smaller step in time. As can be seen from Fig. 2, all three versions of PyPy, Cython and CPP are much more efficient – their execution time for 16,000 numbers does not exceed 0.5s, compared to the execution time of the original version of Python (about 15 s). Of these three versions, the most efficient in most cases is CPP. The Cython version is a bit slower (although for a small number of primes the execution time is close and sometimes even shorter than the CPP version). The PyPy version is less efficient than both converted programs. Fig. 3 shows the speedup factor of different versions compared to the original version of Python. For large values of the nb_primes parameter, the speedup factor of the Compiled version is 1.5, for the PyPy version – 30, for the Cython version – 45 and for the most efficient version of the CPP – 55 times.

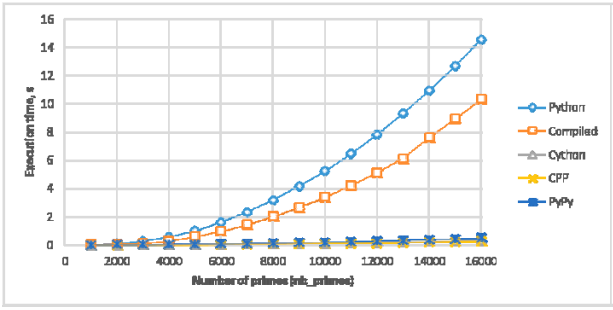

Fig. 1. Comparison of execution time of different versions of the program

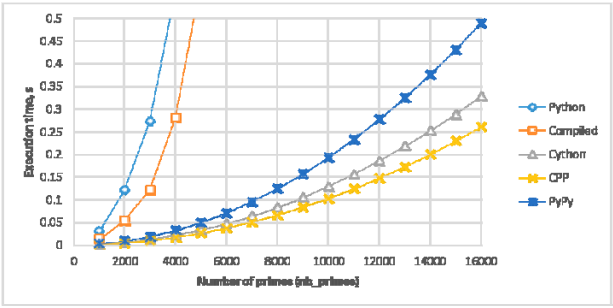

Fig. 2. Details of comparing the execution time of different versions of the program

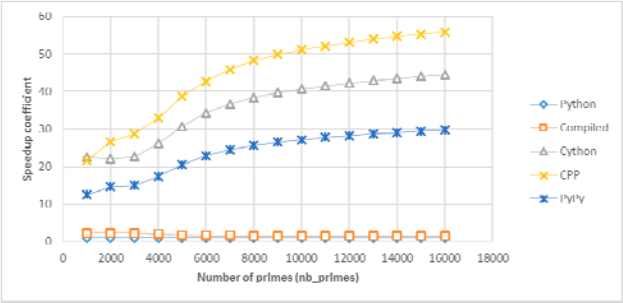

Fig. 3. Speedup factors for different versions of the program

Conclusions

The paper proposes an approach to improving the efficiency of Python code, based on the use of rewriting rules and high-level algebraic code models. Speed-critical code fragments are converted to more efficient Cython and C++ languages, which can significantly increase runtime performance. In this case, the converted fragments are called from existing code in Python, which simplifies their use. The experiments demonstrate the effectiveness of the proposed approach, compared with both the source code in Python and the use of automatic tools – compilation with Cython tools and the use of alternative implementations of PyPy. Further research in this area includes the development of transformations for programs in other subject areas, as well as testing the proposed approach on more complex software systems. It is also planned to use a similar approach to work with parallel programs on modern parallel platforms.

References

1. Gries, P., Campbell, J. and Montojo, J., 2017. Practical programming: an introduction to computer science using Python 3.6. Pragmatic Bookshelf.
2. Roghult, A., 2016. Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy.
3. Herron, P., 2016. Learning Cython Programming. Packt Publishing Ltd.
4. Cython: C-Extensions for Python [Online] Available from: https://cython.org/. [Accessed: 25th February 2020]
5. PyPy: a fast, compliant alternative implementation of Python [Online] Available from: https://www.pypy.org/. [Accessed: 25th February 2020]
6. Marowka, A., 2018. Python accelerators for high-performance computing. The Journal of Supercomputing, 74(4), pp.1449-1460.
7. Fua, P. and Lis, K., 2020. Comparing Python, Go, and C++ on the N-Queens Problem. arXiv preprint arXiv:2001.02491.
8. Doroshenko A., Shevchenko R. A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae. – 2006.– Vol. 72, N 1–3.– P. 95–108.

9. Termware. [Online] Available from: http://www.gradsoft.com.ua/products/termware_eng.html. [Accessed: 25th February 2020]
10. Andon P.I., Doroshenko A.Yu., Zhereb K.A., Shevchenko R.S., Yatsenko O.A., Methods of Algebraic Programming. Formal methods of parallel program development. - Kyiv, "Naukova dumka".-2017.- 440 p.
11. Doroshenko A.Yu., Zhereb K.A. Algebra-dynamic models for program parallelization // Problems in Programming. – 2010. – No. 1. – P. 39–55.
12. Zhereb K.A. Rule-based software toolset for automating application development on Microsoft .NET Platform // Control systems and machines. – 2009. – No. 4. – P. 51–59.
13. Doroshenko A.Yu., Khavryuchenko V.D., Tulika Ye.M., Zhereb K.A. Transformation of the legacy code on Fortran for scalability and cloud computing // Problems in Programming. – 2016 – No. 2-3. – P. 133–140.
14. Cython Tutorial. [Online] Available from: https://cython.readthedocs.io/en/latest/src/ tutorial/cython_tutorial.html. [Accessed: 25th February 2020]
15. G. van Rossum , J. Lehtosalo, Ł. Langa. PEP 484 -- Type Hints [Online] Available from: https://www.python.org/dev/peps/pep-0484/. [Accessed: 25th February 2020]

***About the authors*:**

Kostiantyn Zhereb,
Candidate of Physical and Mathematical Sciences (PhD in Computer Science), Assistant Professor of the Department of Intelligent Software Systems, Taras Shevchenko National University of Kyiv. Senior Research Fellow, Department of Theory of Computing, Institute of Software Systems, National Academy of Sciences of Ukraine. Number of scientific publications in Ukrainian publications – more than 40. Number of scientific publications in foreign publications – more than 10. h-index – 3. http://orcid.org/0000-0003-0881-2284 .

Author affiliation:
Taras Shevchenko National University of Kyiv,
Department of Intelligent Software Systems,
03022, Ukraine, Kyiv, Akademika Glushkova Ave., 4d.
Phone: +380 (44) 259-05-11, e-mail: zhereb@gmail.com, kzhereb@knu.ua