Coroutine-based Combinatorial Generation

by

Sahand Saba

B.Sc., University of Victoria, 2010

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Coroutine-based Combinatorial Generation

by

Sahand Saba

B.Sc., University of Victoria, 2010

Supervisory Committee

Dr. Frank Ruskey, Supervisor

(Department of Computer Science)

Dr. Yvonne Coady, Departmental Member

(Department of Computer Science)

Supervisory Committee

Dr. Frank Ruskey, Supervisor

(Department of Computer Science)

Dr. Yvonne Coady, Departmental Member

(Department of Computer Science)

ABSTRACT

The two well-known approaches to designing combinatorial generation algorithms are the recursive approach and the iterative approach. In this thesis a third design approach using coroutines, introduced by Knuth and Ruskey, is explored further. An introduction to coroutines and their implementation in modern languages (in particular Python) is provided, and the coroutine-based approach is introduced using an example, and contrasted with the recursive and iterative approaches. The coroutine sum, coroutine product, and coroutine symmetric sum constructs are defined to create an algebra of coroutines, and used to give concise definitions of coroutine-based algorithms for generating ideals of chain and forest posets. Afterwards, new coroutine-based variations of several algorithms, including the Steinhaus-Johnson-Trotter algorithm for generating permutations in Gray order, the Varol-Rotem algorithm for generating linear extensions in Gray order, and the Pruesse-Ruskey algorithm for generating signed linear extensions of a poset in Gray order, are given.

Contents

List of Figures

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Frank Ruskey for his mentoring, support, and encouragement. Having a good supervisor is perhaps the most important part of a graduate-level degree, and I feel privileged and fortunate to have had Dr. Ruskey as my supervisor. I would also like to thank Dr. Yvonne Coady for her enthusiastic support and encouragement, and for dedicating her time to be on my supervising committee. I am also grateful to Francine Beaujot for providing numerous helpful suggestions that led to improvements of this thesis.

I was born not knowing and have had only
a little time to change that here and there.
Richard P. Feynman

DEDICATION

I would like dedicate this thesis to my parents, Mohammad and Parvin, for their never-ending encouragement and support.

Chapter 1

Introduction

1.1 Preface

Combinatorial generation algorithms are algorithms that exhaustively list a set of combinatorial objects (e.g. strings, permutations, and graphs) one at a time, often subject to some given constraints, and often in a desired order. Examples of generation orders are lexicographic order, and any order that minimizes the distance of consequent objects, also known as a *Gray* order [8]. The definition of distance varies depending on the combinatorial objects in question and the application, but in most cases it is a measure of how many atomic operations one needs to perform to turn one object to the next. For example, atomic operations can be single bit switches for binary strings, and (possibly adjacent) transpositions for permutations.

Two common approaches to solving a combinatorial generation task are the recursive approach and the iterative approach. In the recursive approach, the goal is to find a subproblem structure in the objects to be generated that allows for a recursive algorithm. For example, in the case of generating all the spanning trees of a given graph, the subproblems could be given by the subgraph with an edge removed, and

the graph that results from contracting an edge into a vertex [15]. This approach can be viewed as an instance of the divide-and-conquer problem-solving strategy.

In the iterative approach, the goal is to find a way to go from one given object to the next in the desired order by analyzing the object and modifying it directly. For example, a common approach to generate combinatorial objects such as strings and permutations in lexicographic order is to scan the object right-to-left to find the first index that can be incremented [8]. In some cases, algebraic or arithmetic properties of the combinatorial objects may be used to iteratively generate them. For example, binary strings can iteratively generated using simple counting in binary, and primitive polynomials over $\mathbb{F}_2$ can be used to iteratively generate De Bruijn sequences using linear feedback shift registers [4].

In this thesis a third approach using coroutines, introduced by Donald Knuth and Frank Ruskey [9], is further explored and used to introduce several new algorithms. Coroutines, which can be seen as generalizations of subroutines, can encompass both recursive and iterative algorithms. As such, they are very suitable mechanisms for combinatorial generation. In fact, one of the most popular coroutine use patterns in modern programming languages is the *generator* pattern, which will be discussed in detail in the next section. As the name "generator" suggests, generators provide a very convenient mechanism for implementing combinatorial generation algorithms, recursive or iterative. Moreover, since coroutines are generalizations of subroutines, we can exploit their generality to come up with combinatorial generation algorithms that are somewhere between recursive and iterative. It is also possible to link coroutines in intricate and flexible ways, leading to a set of operations on coroutines such as *coroutine products* and *coroutine symmetric sums*, which can be taken to form an *algebra of coroutines*. This thesis introduces the coroutine-based approach to combinatorial generation by providing an introduction to coroutines first, and then gently

introducing the algebra of coroutines, and the various ways that they can be used to tackle combinatorial generation problems. Ample examples, diagrams, and sample programs are used to demonstrate the concepts and algorithms throughout the thesis.

1.2 Outline

Section 2.1 begins by defining coroutines and providing an introduction to their implementation in Python. Several examples are provided to illustrate coroutine use-cases and behaviour. A brief introduction to the use of coroutines for multitasking is provided as well.

Afterwards, Section 2.2 delves into combinatorial generation by first reviewing the two well-known approaches used, namely the recursive and the iterative approaches, by using generation of multi-radix numbers in lexicographic order as the running example. Once the existing approaches are reviewed, the coroutine-based approach is introduced using the same example. This section finishes by abstracting some of the patterns used in the given coroutine-based algorithm, and defining the concepts of *local coroutine* and *coroutine product*, which will be used throughout the rest of the thesis.

Section 3.1 starts by generalizing the results from Section 2.2.4 to generate multi-radix numbers in Gray order instead. Generation of combinatorial objects in Gray order will require the introduction of *reflected* local coroutines. Afterwards, Section 3.2 will tackle the equivalent problem of the generation of ideals of a poset consisting of linear chains, using coroutines, and two further coroutine operations, the *coroutine sum* and the *coroutine symmetric sum*, are introduced for this task. Section 3.2 will use the same tools to generate forest poset ideals.

With the coroutine algebra and the related tools in place, the rest of Chapter 3

will give coroutine-based algorithms for generation of permutations in Gray order (based on the Steinhaus-Johnson-Trotter algorithm), generation of linear extensions of a poset (based on the Varol-Rotem algorithm), and generation of signed linear extensions of a poset in Gray order (based on the Pruesse-Ruskey algorithm), after providing introductions to the problems. These examples will demonstrate the versatility of the coroutine-based approach.

Chapter 4 concludes the thesis with a summary and a discussion of further possible work.

1.3 Contributions of This Thesis

Coroutines have been receiving increasing attention in recent years as as general control abstractions [12]. Many modern languages such as Python [29], JavaScript (in version 1.7 [25]), and C# [21] include at least partial support for coroutines, and many libraries for other languages have been created to add support for coroutines—for example, `libconcurrency` for C [24] and `Boost.Coroutine` for C++ [20]. The use of coroutines as "light-weight" or "pseudo" threads is partly responsible for this increase in interest in coroutines.

The three main contributions of this thesis are the following.

- This thesis contains, to the best of our knowledge, the first introductory treatment of coroutine-based combinatorial generation with working code in a modern programming language, which makes the treatment more accessible to those unfamiliar with coroutines.

- While implicitly used in [9], an algebra of coroutines is explicitly introduced and developed in more detail in this thesis, and used in multiple algorithms.

- Several new coroutine-based combinatorial generation algorithms that are based on existing algorithms are introduced.

The coroutine-based approach to combinatorial generation, apart from being an interesting example of the use of coroutines, is of interest for several reasons, a few of which are listed below. First, using coroutines for combinatorial generation can lead to code that is simpler and easier to understand. Secondly, the coroutines can also be reused and linked with other coroutines in very versatile ways. This is demonstrated in Chapter 3 with the use of the coroutine product and coroutine symmetric sum operations. This abstraction of the coroutine operations can allow for both more formalization of correctness proofs as well as automated optimization strategies. Thirdly, the coroutine-based implementations will also work seamlessly with event-loop environments that support coroutines, such as Python 3.4's `asyncio` module [27]. Finally, the coroutine-based approach is a natural approach if a Gray code is desired, since the coroutines can be seen as traversing the Hamiltonian path or cycle corresponding to the Gray code in the underlying graph of the combinatorial objects.

1.4 Conventions And Notation

All the sample source code in this thesis is intended for Python 3.3 or newer, unless otherwise indicated.

Sample source code both in figures and in text are typeset in monospaced font and syntax highlighted: keywords and built-in functions and constants such as `for` and `True` are in green, variables and user-defined functions such as `myfunction` are in black, exception and error types such as `StopIteration` are in red, string constants such as `"String"` are in dark red, and numeric constants such as `123` are in gray.

All the source code provided in this thesis is available online at `https://github.`

com/sahands/coroutine-generation.

Chapter 2

Preliminaries and Previous Work

2.1 Coroutines And Their Implementations

2.1.1 Definition of Coroutine

Subroutines are self-contained sequences of instructions that can be reused in different parts of a program, or by different programs [17]. Calling a subroutine involves pausing the execution of the current subroutine while keeping the current subroutine's state[1], generally in a stack data structure called the *call stack*, and transferring the control flow to the subroutine being called. The called subroutine then allocates space for its local variables (often done by pushing on to the call stack, and upon completion, popping from the call stack to deallocate them), runs its instructions in sequence, and upon competition returns the control flow back to the calling subroutine.

Coroutines, on the other hand, can be called multiple times while retaining their state in between calls, and can *yield* the control to another coroutine until they are given the control again [7, 11]. The word coroutine was first introduced by Melvin Conway [2], who defined it as "an autonomous program which communicates with

[1] *State* here refers to the value of the variables local to the subroutine, as well as where the execution is pausing in the subroutine, which is often stored as an instruction pointer.

adjacent modules as if they were input or output subroutines." That is, coroutines are generalizations of subroutines that allow for multiple entry points, that can yield multiple times, and that resume their execution when called again. On top of that, coroutines can transfer execution to any other coroutine and not just the coroutine that called them. Subroutines, being special cases of coroutines, have a single entry point, can only yield once, and can only transfer execution back to the caller coroutine. Figure 2.1 illustrates the difference between subroutines and coroutines using example flow diagrams.

The term *yielding* is used to describe a coroutine pausing and passing the control flow to another coroutine. Since coroutines can pass values along with the control flow to another coroutine, the phrase *yielding a value* is used to describe yielding and passing a value to the coroutine receiving the control.

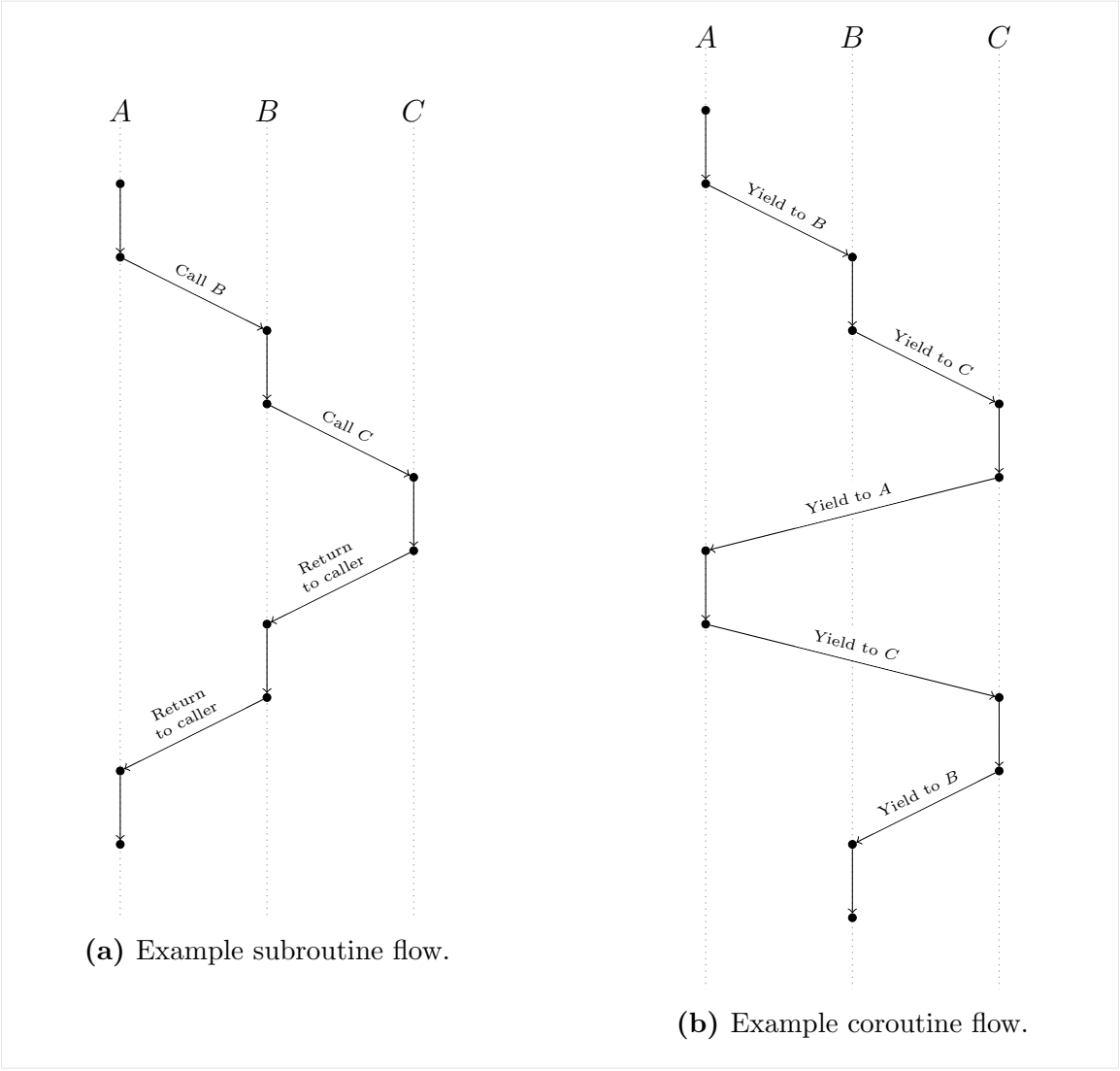
The next section goes over the implementation of coroutines in Python and provides multiple examples of simple coroutines in Python.

2.1.2 Coroutines in Python

In Python, *generators*, which are coroutines with a few restrictions, were introduced in PEP[2] 255 [29] and added to Python starting from version 2.2. Generators allow for multiple entry-points, and can therefore yield multiple times. Figure 2.2 contrasts calling a subroutine several times with calling the same instance of a generator several times, to highlight that calling the same generator does not create multiple instances of it.

Generators in Python are more restricted because they can only return the control to the caller and not any arbitrary coroutine. This restriction can be visualized by comparing Figure 2.2b, which shows an example flow diagram for a generator, with

[2]A Python Enhancement Proposal (PEP) is "a design document providing information to the Python community, or describing a new feature for Python or its processes or environment." [34]



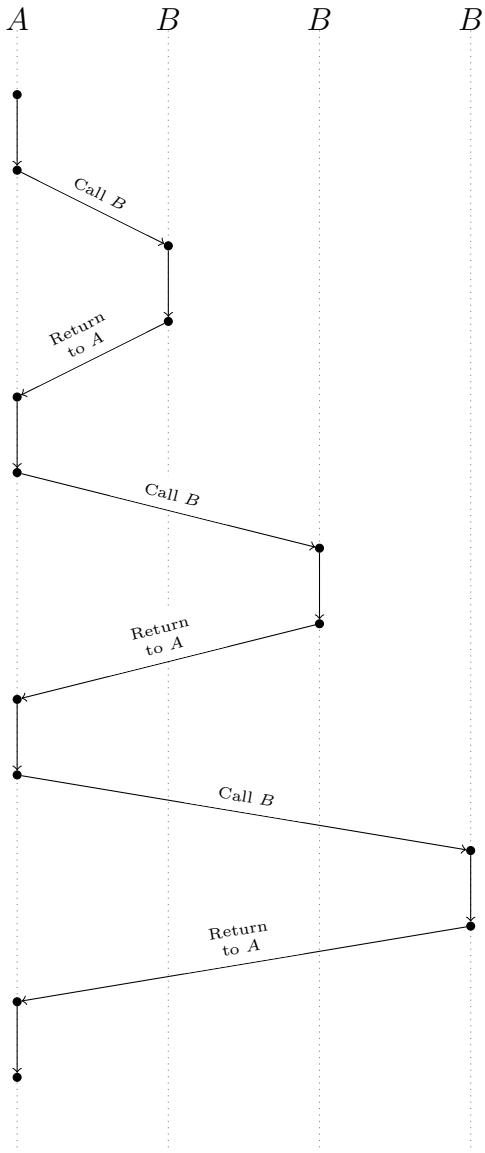
(a) Example subroutine flow.

(b) Example coroutine flow.

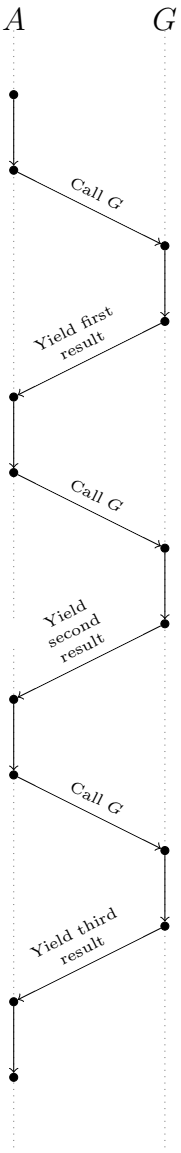
Figure 2.1: Example flows of coroutines and subroutines.

Figure 2.1b.

The syntax for defining generators in Python is very similar to that of Python functions, with the main difference being the use of the `yield` keyword instead of `return` to pause execution and yield to the caller. The syntax for using generators is rather different from Python functions, and is in fact closer to how classes are treated in Python: calling a generator function returns a newly created *generator object*, which is is an instance of the coroutine independent of other instances. To call



(a) Example flow of the same subroutine called three times.



(b) Example flow of the same generator instance called three times.

Figure 2.2: Example flows of the same subroutine called three times v.s. the same generator instance called three times.

the generator, the `next` built-in function is used, and the generator object is passed to `next` as the parameter. An example of a simple generator in Python is given in Figure 2.3 which is taken, with minor modification, from PEP 255 [29]. Here we

have a generator that yields the Fibonacci numbers ad infinitum. Each call to the generator slides the a and b variables ahead in the sequence, and then execution is paused and b is yielded. Usage of the `fib` generator is shown in Figure 2.4.

```
def fib():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b
```

Figure 2.3: Generating the Fibonacci sequence using a generator coroutine.

```
from fib import fib


f = fib()       # Create a new "instance" of the generator coroutine
print(next(f))  # Prints 1
print(next(f))  # Prints 1
print(next(f))  # Prints 2
print(next(f))  # Prints 3
print(next(f))  # Prints 5
print(next(f))  # etc...
```

Figure 2.4: Usage of the Fibonacci sequence generator.

```
def postorder(tree):
    if not tree:
        return

    for x in postorder(tree['left']):
        yield x

    for x in postorder(tree['right']):
        yield x

    yield tree['value']
```

Figure 2.5: Example of a generator used with a recursive algorithm.

Before continuing, let us look at a simple example of a recursive algorithm implemented using coroutines as well. In this example, the algorithm is a simple postorder

traversal of a binary tree. Notice how generators can be used recursively, with each recursive call creating a new instance of the generator coroutine. Usage of the recursive binary tree traversal is shown in Figure 2.6.

In Python 3, with PEP 380 [23], the above can be made even simpler by using the `yield from` statement, which delegates generation to a subgenerator[3]. This shorter syntax is shown in Figure 2.7.

```
from collections import defaultdict
from recursive_generator import postorder

tree = lambda: defaultdict(tree)
T = tree()
T['value'] = '*'
T['left']['value'] = '+'
T['left']['left']['value'] = '1'
T['left']['right']['value'] = '3'
T['right']['value'] = '-'
T['right']['left']['value'] = '4'
T['right']['right']['value'] = '2'

postfix = list(postorder(T))
print(postfix)  # ['1', '3', '+', '4', '2', '-', '*']
```

Figure 2.6: Usage of the `postorder` recursive generator.

```
def postorder(tree):
    if not tree:
        return
    yield from postorder(tree['left'])
    yield from postorder(tree['right'])
    yield tree['value']
```

Figure 2.7: Recursive generators using the `yield from` syntax of Python 3.

Python generators were further generalized to allow for more flexible coroutines in PEP 342 [28]. Prior to the enhancements in PEP 342, Python's generators could

[3]The `yield from` semantics are more complicated than the given example demonstrates. Using `yield from` to delegate to a subgenerator results in exceptions being delegated properly as well. It is also possible to use `yield from` as an expression with the resulting semantics somewhat different from using `yield` as an expression. These use cases are outside the scope of this thesis and the interested reader is referred to PEP 380 [23] for a full treatment.

not accept new parameters after the initial parameters were passed to the coroutine. With PEP 342's `send` method, a coroutine's execution can resume with further data passed to it as well. This is implemented by allowing the `yield` keyword to be used not just as a statement but also *as an expression*, the evaluation of which results in the coroutine pausing until a value is passed to it via `send`, which will be the value that the `yield` expression evaluates to. Figure 2.8 displays a simple example of a coroutine that yields `"Ready for x"` and then waits until a value for `x` is sent to it. Note that a `StopIteration` exception is raised as well. This exception is raised after a coroutine runs its last instruction.

```
def coroutine():
    x = yield "Ready for x"  # Yield "Ready for x", then wait to be passed x
    print(x)


def main():
    c = coroutine()
    value = next(c)
    print(value)  # Prints "Ready for x"
    c.send("Here is x")  # Prints "Here is x", and raises StopIteration

main()
```

Figure 2.8: Use of `send`, and `yield` as expressions, to pass values to executing coroutines.

Even though `StopIteration` is an exception, instances of it generally do not indicate errors. This exception is simply used to send a signal to the caller that a coroutine has finished running. Built-in Python loops (namely `for` and `while` loops) catch this exception to know when to stop looping when they loop over generators. In fact, this behaviour is standard for all Python *iterators*, as defined in PEP 234 [35], and Python generators are also iterators (i.e. invoking `iter` on a generator object simply gives back the generator object). Figure 2.9 shows the simple semantics of looping over a generator, and how `StopIteration` is used to break out of a loop (the two functions `loop` and `loop_alternative` are semantically equivalent).

```
def G():
    yield 1
    yield 2
    yield 3


def loop():
    for x in G():
        print(x)  # Prints 1, 2 and 3 on separate lines


def loop_alternative():
    g = G()
    while True:
        try:
            x = next(g)
        except StopIteration:
            break
        else:
            print(x)
```

Figure 2.9: Functions `loop` and `loop_alternative` are semantically equivalent.

It is worthwhile to note that even with PEP 342, Python's generators do not implement coroutines in full generality. To quote Python's official language reference [31]:

> All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where should the execution continue after it yields; the control is always transferred to the generator's caller.

Hence, unlike Knuth's definition of coroutines, Python's coroutines are not completely symmetric; an executing coroutine object is still coupled to the caller, which creates asymmetry.

It is also important to mention that in some Python literature the word coroutine means specifically coroutines that use `yield` in an expression and hence require the

use of `send`. See [19] for example (which is an excellent introduction to coroutines and their uses in I/O operations, parsing, and more). This use of the word is somewhat inaccurate, since coroutines are a general concept, and subroutines, generators with `next` or `send` or both, all fall under the concept of coroutines. In this thesis the word coroutine is used in its generality, as defined in the first paragraph of this section, in accordance with how Knuth defines the word in [7].

To summarize, on an abstract level, the set of coroutines contains the set of generators and subroutines, and more. See Figure 2.10 for an Euler diagram of of the sets of coroutines, Python coroutines, generators, and subroutines.

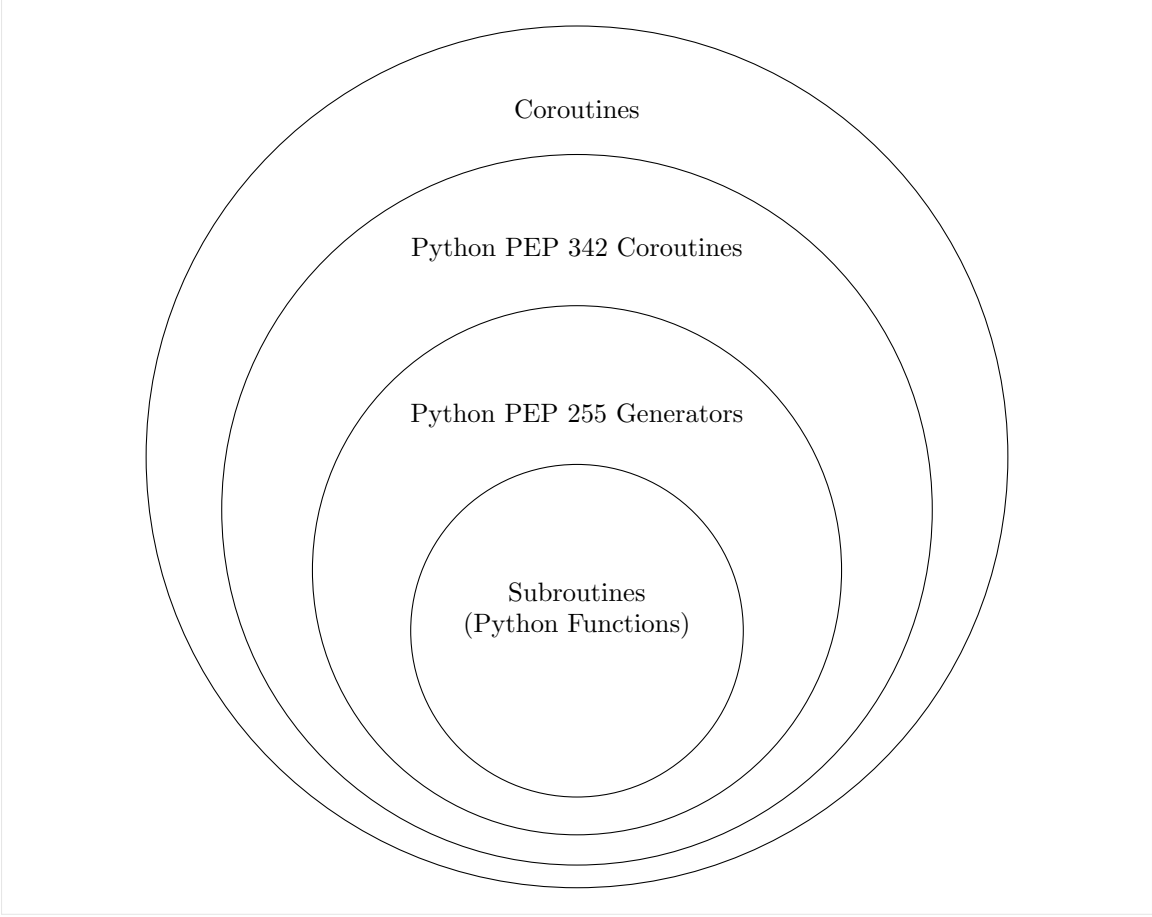


Figure 2.10: Euler diagram of the hierarchy of coroutines, Python coroutines, Python generators, and subroutines.

In most of the code in this thesis (with exceptions of examples in the next section on multitasking and coroutines), only the generator pattern is used, and consequently `yield` is used as a statement and not an expression. We will look at the use of `yield` as an expression and `send` in more detail in the next subsection where a brief introduction to multitasking using coroutines is given.

2.1.3 Coroutines And Multitasking

Coroutines are often used to implement *lightweight threads*. A full treatment of the use of coroutines for multitasking is outside the scope of this thesis, but for the sake of completeness, this section provides an introduction to the subject.

Consider a simplified file-server as an example. In this example, the server listens for incoming connections, and once an incoming connection is established, the connecting client sends a *request* containing a filename to the server. The server then processes the request and sends a *response* to the client. The processing will involve reading from disk, an operation that can have a relatively high latency—see Figure 2.11.

The simplest implementation of such a server would be a single-threaded loop that waits for the next request, processes it, and then repeats. However, such a server would have to wait until the current request is processed and the response is sent completely before the next request is served. This is not ideal since, as can be seen in Figure 2.11, much of the CPU time will be wasted waiting for file to be read from disk, leading to other clients that are connecting simultaneously to have to wait for their turn, which can lead to high wait-times and possible time-outs.

One solution to this conundrum is to use OS-level concurrency, that is threads or processes, to process multiple requests concurrently, with the immediate solution being to spawn a new process (or thread) for each request. This solution is considered

Event	Latency	Scaled Latency
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

Figure 2.11: Scaled average latency of various operations. Taken from [5].

to be "heavy-weight" in terms of resource use, since creating new processes or threads requires allocation of memory (each process will have its own memory space, and each thread will have at least a stack allocated for it) and initialization of the process or thread. In addition, the OS will then be in charge of switching between the tasks and the switches will involve relatively CPU-expensive context-switches.

One approach to mitigating the issues with the previous solution is to use process or thread *pools*. That is, a pool of request handler processes can be pre-allocated, and requests can be assigned to the first available request handler process in the pool. Alternatively, a pool of threads can be created and an available thread can be assigned to (or a new one created for) a new request. The Apache web-server, for example, can use a mix of these two strategies to handle requests [18], with a pool of processes each with their own pool of threads.

Using pools mitigates the performance problems of multi-processing or multi-threading servers to some extent, but the overhead of using multiple processes and threads is not completely eliminated, since each individual thread or process will still

be spending the vast majority of its time waiting for operations with relative high-latency. A better solution would be to use each individual process or thread more efficiently. Furthermore, other complications such as deadlocks, race conditions, etc; that arise with the use of multiple processes and threads, will still exist [26].

A different solution would be to work within a single-threaded environment, and use an *event-loop* (sometimes also referred to as a *trampoline*, especially when coroutines are used with the event-loop) instead of threads and processes. In the event-loop model, a single thread of execution continuously polls for *events* and executes corresponding event-handlers when there are new events available. This allows the server to continue to stay busy while operations with high latency such as disk I/O take place and the result becomes available.

The file-server described earlier can be setup using a hypothetical event-loop based environment as shown in Figure 2.12. Here, instead of sequentially executing the instructions with high latencies, the request handler asks the event-loop to execute the instructions and provides handlers to be run for when the instructions finish. Event handlers such as the one in Figure 2.12 are often called *callbacks*.

Event-loop based environments such as above can be very efficient at handling heavy requests loads using only a single process and a single thread. Since the environment is single-threaded, concurrency issues such as deadlocks and race conditions are also non-existent in these environments. However, event-loop based code using callbacks can arguably become very unreadable and difficult to debug. Even the code given in Figure 2.12, which is heavily simplified, portrays how unnatural programs can become when callbacks are used to control the flow of data in the program. Once error handling is added, this issue becomes even worse. To quote Guido van Rossum, the creator of Python, "it requires super human discipline to write readable code using callbacks [33]." Others have compared callbacks to modern-day equivalents

```
def handle_new_connection(event):
    connection = event.connection

    def close_connection():
        connection.close()

    def send_file(event):
        connection.send(event.file_data, on_completed=close_connection)

    def handle_new_request(event):
        read_file(event.data, on_completed=send_file)

    connection.read_line(on_completed=handle_new_request)


def run_main_loop():
    listen_for_connections(on_new_connection=handle_new_connection)
    while True:
        event = get_next_event()
        for handler in get_event_handlers(event):
            handler(event)
```

Figure 2.12: Example of a hypothetical event-loop based server using callbacks.

of GOTO statements [22], with "callback hell" the modern equivalent of "spaghetti code".

An alternative to using callbacks that addresses issues surrounding code readability and maintainability, as well as difficulties with error handling, is cooperative multitasking using coroutines. The idea is to still have an event-loop that calls event handlers but also resumes coroutines that are waiting on the result of another coroutine, or an event. This allows for handler coroutines to simply yield control back to the event-loop when they need the results of a high-latency operation, such as a disk I/O operation. When yielding, the coroutines also yield an instance of the coroutine whose results they need to continue. Figure 2.13 portrays the use of a event-loop using a flow diagram.

Using an event-loop setup that supports coroutines, the hypothetical server code is given in Figure 2.14. The Python Tornado web-server [32], and Python 3.4's `asyncio` module [27], both provide coroutine-based event-loop environments that support code

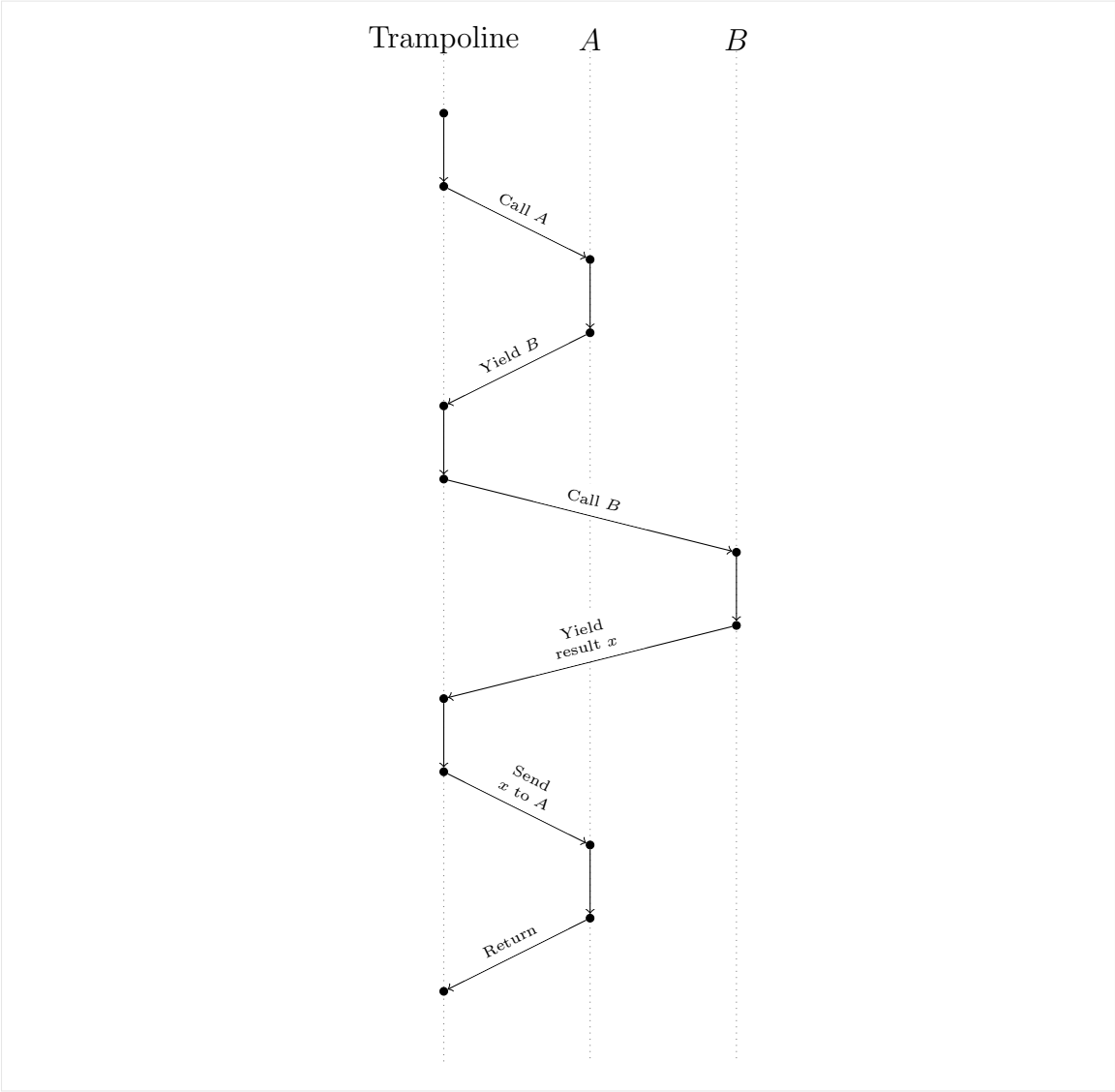


Figure 2.13: Use of a event-loop (trampoline) to dispatch control to coroutines with dependencies (A needs result x from B).

similar to that given in Figure 2.14.

2.2 Approaches to Combinatorial Generation

To introduce the use of coroutines for combinatorial generation, first a short introduction to the other two approaches (the recursive approach and the iterative approach)

```
def handle_connection(connection):
    filename = yield connection.read_line()
    file_content = yield read_file(filename)
    yield connection.send(file_content)
    connection.close()


def main():
    server = create_server(on_new_connection=handle_connection)
    server.listen()
```

Figure 2.14: Example of a hypothetical event-loop based server using coroutines.

is provided to set the context and allow for easy comparison of the approaches.

Generation of multi-radix numbers in lexicographic order will be used as the running example in this section. A *multi-radix base of length* n is defined as a set of positive integers $\{M_0, \ldots, M_{n-1}\}$ for $n > 0$. A *multi-radix number* a in base M is then defined as any string of integers $\{a_0, \ldots, a_{n-1}\}$ such that $0 \leq a_i < M_i$ for all applicable i. In this section, to introduce the coroutine-based algorithms, we look at three algorithms for generating all multi-radix numbers in lexicographic order given a base M.

2.2.1 Recursive Approach

To use recursion, we need to reduce the problem to a subproblem. Given that M has n items in it, we are producing multi-radix numbers with n digits. Let M' be a base of length $n-1$ with $M'_i = M_i$ for $0 \leq i < n-1$. That is, M' is the first $n-1$ elements of M. Then if we have a list of multi-radix numbers for base M' in lexicographic order, we can extend that list to a list of lexicographic numbers for M by appending 0 to $M_{n-1} - 1$ to each element of the list.

Letting A_k for $k > 0$ be the sequence of multi-radix numbers in lexicographic order

in base $\{M_0, M_1, \ldots, M_{k-1}\}$, the reduction to subproblems is given by the recursion

$$A_0 = \{\epsilon\}, \tag{2.1}$$

$$A_k = \{\mathbf{a}x : 0 \leq x < M_{k-1} \text{ and } \mathbf{a} \in A_{k-1}\}, \tag{2.2}$$

where ϵ is the empty string and $\mathbf{a}x$ is $\mathbf{a}$ concatenated at the end with x.

The above recursion leads to the recursive code given in Figure 2.15.

```
def multiradix_recursive(M, i):
    if i < 0:
        yield []
    else:
        for a in multiradix_recursive(M, i - 1):
            for x in range(M[i]):
                yield a + [x]


def gen_all(M):
    return multiradix_recursive(M, len(M) - 1)
```

Figure 2.15: Recursive generation of multi-radix numbers.

2.2.2 Iterative Approach

For an iterative solution, a common strategy for lexicographic generation is to scan the combinatorial object from right-to-left, looking for the first place an increment can be made, and then to make the increment. The elements to the right of the point that the increment is made need to then be reset to be the smallest possible in lexicographic order. In the case of multi-radix numbers, this means scanning right to left for an index i such that $a_i < M_i - 1$, and setting the digits for which $a_i = M_i - 1$ to 0 along the way. This iterative approach leads to the iterative algorithm given in Figure 2.16.

```
def gen_all(M):
    n = len(M)
    a = [0] * n
    while True:
        yield a
        k = n - 1
        while a[k] == M[k] - 1:
            a[k] = 0
            k -= 1
            if k < 0:
                return
        a[k] += 1
```

Figure 2.16: Iterative generation of multi-radix numbers by scanning right to left.

2.2.3 Coroutine-based Approach

The idea behind combinatorial generation with coroutines is to use multiple instances of the same sequence of instructions, each with their own independent state, and proper communication among them to generate the set of combinatorial objects. To explain the first example of a coroutine-based combinatorial generation algorithm, the allegory of a line of "friendly trolls" is borrowed from [9]. (The trolls were first used by Knuth in a lecture given at University of Oslo [6].)

Imagine a line of $n+1$ friendly trolls, numbered -1 to $n-1$ (the rationale behind the somewhat odd indexing is for indices to match Python's zero-based indexing—troll number -1 does not correspond to a bit in the binary string and hence the index for it does not need to be a valid array index). Trolls 0 to $n-1$ all behave the same way, and each is either asleep or awake, with all trolls starting asleep. They behave in the following way: when asleep and poked trolls 1 to n simply wake up and yell "moved". If awake when poked, the trolls simply poke their neighbour (defined as the troll behind them in line, i.e. for troll number k, the neighbour is troll number $k-1$) without yelling anything, and fall asleep immediately after. Troll number -1 is special, and always simply yells "done" when poked. Troll number $n-1$ is called

the *lead troll*. With this simple setup, poking the lead troll repeatedly, until either "moved" or "done" is heard results in the generation of binary strings in lexicographic order. Hearing "done", which indicates the last troll is poked, indicates the end of the combinatorial generation. The trolls here basically simulate our iterative algorithm that was given in Figure 2.16 if all $M_i = 2$. See Figure 2.17 for an illustration of the algorithm.

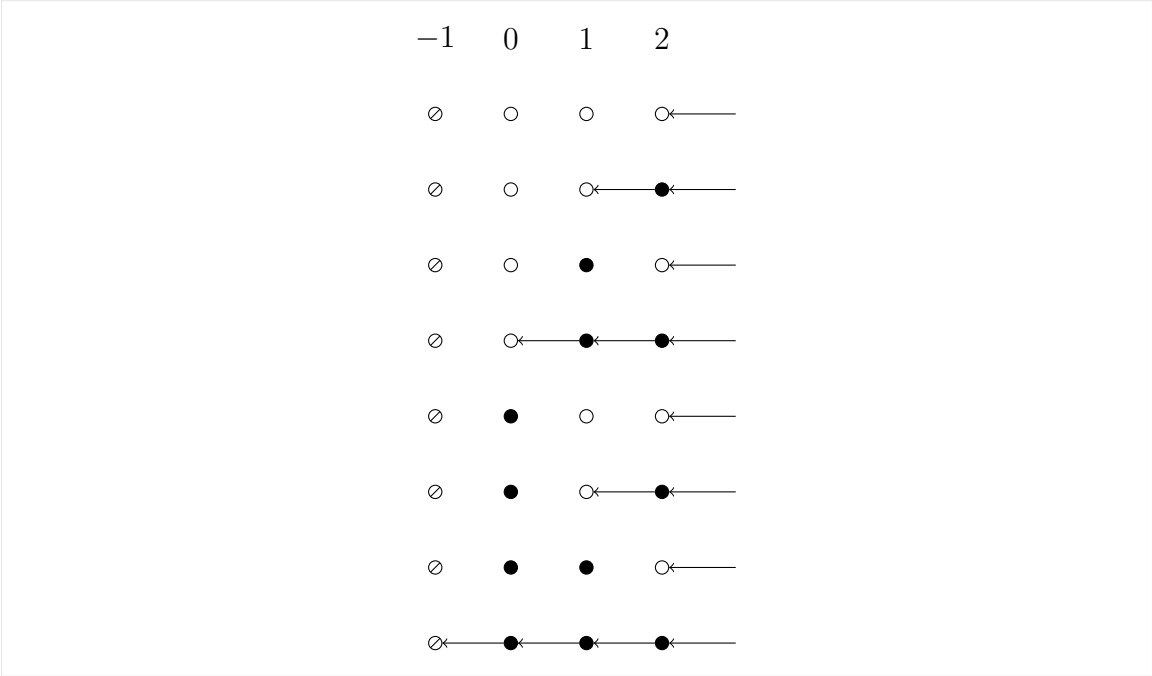


Figure 2.17: Generation of binary strings using "trolls"—arrows indicate sequences of pokes, empty circles indicate asleep troll, filled circles indicate awake troll.

In the above explanation, the trolls can naturally be implemented as coroutines, with their awake or asleep state naturally taking place based on which instruction will be run next. Finally, what the trolls yell can be passed on using the value they yield. We will use `True` for "moved" and `False` for "done". This leads to the coroutine given in Figure 2.18.

To use this coroutine, we first need to setup $n + 1$ instances of it, and link them to each other by passing the correct `neighbour` variable to each instance. We then

```
def troll(a, i=None, neighbour=None):
    while True:
        if neighbour is None:
            yield False  # If last troll in line, just yell "done"
        else:
            a[i] = 1  # Wake up
            yield True  # Yell "moved"
            a[i] = 0  # Go to sleep
            yield next(neighbour)  # Poke neighbour
```

Figure 2.18: Coroutine to generate binary strings in lexicographic order.

start by poking the lead coroutine until `False` is yielded, which indicates the end of the combinatorial generation task. The setup and use code is shown in Figure 2.19.

```
from binary_strings_coroutine import troll


def setup(n):
    a = [0] * n
    lead = troll(a, neighbour=None)  # Start with the last troll in line
    for i in range(n):
        lead = troll(a, i, neighbour=lead)
    return a, lead


def visit(a):
    print(''.join(str(x) for x in a))


def print_binary_strings_in_lex(n):
    a, lead = setup(n)
    while True:
        visit(a)
        if not next(lead):
            break


print_binary_strings_in_lex(3)
```

Figure 2.19: Usage of the `troll` coroutine to generate binary strings in lexicographic order.

2.2.4 Generalization To An Algebra of Coroutines

A few ideas can now be extracted from the example in the previous section and generalized. First, the "barrier" coroutine that always yields `False` (which was troll number -1), or variations of it, will be used in some of the later algorithms in this thesis to signal the end of the combinatorial generation task. In mathematical notation the symbol $\varnothing$ is used for this coroutine. This coroutine is given in Figure 2.20.

```
def barrier():
    while True:
        yield False
```

Figure 2.20: The `barrier` coroutine that repeatedly yields `False`.

Another notable pattern is how the coroutines poke each other, from which we can extract a pattern of linking coroutines that we will refer to as *coroutine product* (abbreviated to *coproduct*[4]). To define coproduct, first *coroutine multiplication* (abbreviated to *comultiplication*) is defined. The comultiplication $X \times Y$ of X and Y is defined as the coroutine that, given two coroutines X and Y, will repeatedly yield `True` while Y yields `True`, and then yield what X yields. It is important to note that the comultiplication operator $\times$ is associative, for reasons similar to why function composition is associative. The Python code for $X \times Y$ is given in Figure 2.21.

```
def comultiply(X, Y):
    while True:
        while next(Y):
            yield True
        yield next(X)
```

Figure 2.21: The `comultiply` coroutine to multiply two coroutines X and Y to get $X \times Y$.

The coproduct of a sequence of coroutines X_1 to X_k, written in mathematical no-

[4]The "co" in coproduct (and later, cosum and cosymsum) stands for coroutine. These coroutines are not meant to be directly related to the similarly named concepts in category theory.

tation as $\prod_{i=1}^{k} X_i$, is defined as the left-associative[5] comultiplication of the coroutines. That is, we have

$$\prod_{i=1}^{k} X_i = (\ldots(((X_1 \times X_2) \times X_3) \times \ldots) \times X_{k-1}) \times X_k. \tag{2.3}$$

The code for coproduct is given in Figure 2.22.

```
from combgen.common import comultiply


def coproduct(*Xs):
    iterator = iter(Xs)
    lead = next(iterator)
    for X in iterator:
        lead = comultiply(lead, X)
    return lead
```

Figure 2.22: The `coproduct` of a sequence of coroutines.

With these coroutines abstracted, the example from the previous section can quite easily be generalized to generate multi-radix numbers in lexicographic order. From here on, instead of "troll", the coroutines that form the individual unit of coroutine-based algorithms will be referred to as *local coroutines*, or simply *locos*, because they act locally on the combinatorial object, following a well-defined subpath of the Hamiltonian path or cycle. In the current example the troll coroutine with the `barrier` and `coproduct` parts extracted, becomes a simple loco: it simply switches a_i and yields `False` and `True` accordingly. See Figure 2.23 for this simplified loco. Letting X_i be an instance of this loco for the given i, the full sequence can be generated using simply

$$\prod_{i=0}^{n-1} X_i. \tag{2.4}$$

[5]Note that since the $\times$ operator is associative, the definition does not need to specify "left-associative" explicitly. However, since the code implements `coproduct` left-associatively, this is reflected in the definition to avoid any confusion.

```
def binary_strings_lex_local(a, i):
    while True:
            a[i] = 1
            yield True
            a[i] = 0
            yield False
```

Figure 2.23: Loco for binary strings with `coproduct` and `barrier` extracted.

To generalize the previous loco to generate multi-radix numbers instead of binary strings only, the only change necessary is to make the digit increment modulo M_i. This loco is given in Figure 2.24.

```
def multiradix_lex_local(M, a, i):
    while True:
        a[i] = (a[i] + 1) % M[i]
        yield a[i] != 0
```

Figure 2.24: Loco for generation of multi-radix numbers in lexicographic order.

Using the abstracted `coproduct` and `barrier` coroutines, the setup code is simplified to that given in Figure 2.25, which simply sets up the coroutines and calculates the coproduct corresponding to $\prod_{i=0}^{n-1} X_i$.

```
from combgen.common import coproduct
from .local import multiradix_lex_local


def setup(M):
    n = len(M)
    a = [0] * n
    coroutines = [multiradix_lex_local(M, a, i) for i in range(n)]
    lead = coproduct(*coroutines)
    return a, lead
```

Figure 2.25: Setup for generation of multi-radix numbers in lexicographic order using coroutines.

2.2.5 Summary

In the previous section, only a single operation on coroutines, namely coroutine product, was defined. This operation is in many ways similar to recurisve calls; hence its use in the problem given in that section. However, as the word *algebra* in the name of the previous section suggests, more operations on coroutines are possible. The next chapter introduces two other operations, namely *coroutine sum* and *coroutine symmetric sum*, which together with coroutine products can be used to solve a variety of combinatorial generations problems. With these operations, the coroutine-based approach is reduced to the following. First, creating a loco that, independent of the rest of the coroutines, makes the local changes needed to generate the combinatorial objects. Afterwards, the instances of the loco need to be set up and linked to each other using coroutine product, coroutine sum, or coroutine symmetric sum, resulting in a *lead* coroutine. Finally, the initial combinatorial object needs to be set up, and the lead coroutine can be called repeatedly until `False` is yielded, indicating the end of the combinatorial generation task. In the next chapter, several combinatorial generation problems, in increasing order of difficulty, will be approached using coroutines and the aforementioned operations.

Chapter 3

New Work

3.1 Generating Multi-radix Numbers In Gray Order

3.1.1 Problem Definition

In this section, the coroutine-based algorithm from the previous section is generalized to generate multi-radix numbers in Gray order instead of lexicographic order. A *Gray order* for a set of combinatorial objects is a listing of the objects such that the *distance* between consequent objects a and b is constant. For this definition to be meaningful, a precise definition of *distance* needs to be provided. For this section, we define the distance between two multi-radix numbers $a = \{a_0, \ldots, a_{n-1}\}$ and $b = \{b_0, \ldots, b_{n-1}\}$ in base M of length n to be

$$\text{dist}(a, b) = \sum_{i=0}^{n-1} |a_i - b_i|. \tag{3.1}$$

With this definition, a Gray order for multi-radix numbers is one in which the next number is generated by incrementing or decrementing a single digit by exactly

one. An example of a Gray code for base $M_0 = 3$, $M_1 = 2$ and $M_2 = 3$ is given in in Figure 3.1, where an edge exists between two vertices a and b if and only if $\text{dist}(a, b) = 1$, and the arrows indicate the Gray code, with the initial multi-radix number distinguished by being enclosed by a rectangle.

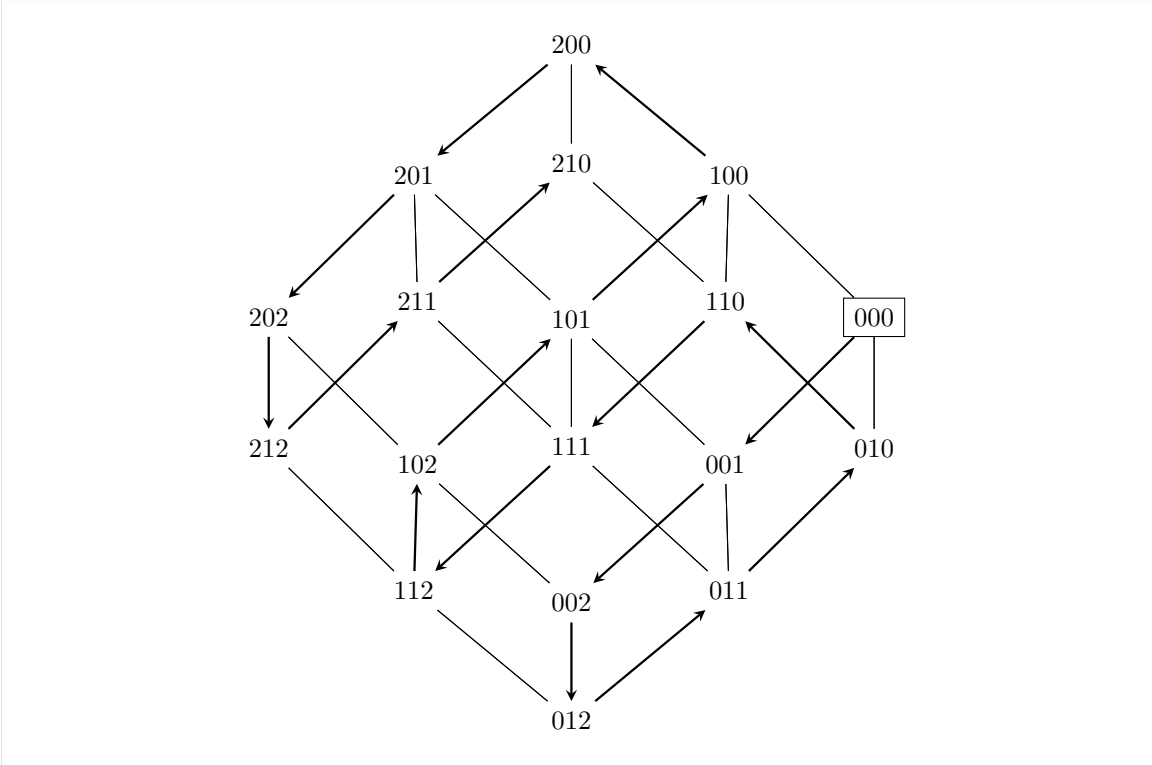


Figure 3.1: Graph corresponding to multi-radix numbers with base $M_0 = 3$, $M_1 = 2$ and $M_2 = 3$ with Hamiltonian path indicated using arrows.

3.1.2 Coroutine-based Algorithm

Assume that a loco X performs invertible operations $\alpha_1, \ldots, \alpha_k$ resulting in distinct combinatorial objects, after each of which it yields `True` and finally yields `False`. Since for Gray codes the locos will need to perform a single atomic operation each time, define the *reflected loco* corresponding to X as the coroutine that performs $\alpha_1, \ldots, \alpha_k$ first, yields `True` after each, then yields `False`, and then performs

$\alpha_k^{-1}, \ldots, \alpha_1^{-1}$, yields `True` after each, yields `False`, and then repeats the operations from the beginning. Therefore the sequence of operations performed by the reflected loco is:

$$\alpha_1, \ldots, \alpha_k, \alpha_k^{-1}, \ldots, \alpha_1^{-1}, \alpha_1, \ldots, \alpha_k, \alpha_k^{-1}, \ldots, \alpha_1^{-1}, \ldots \tag{3.2}$$

Note that the above collapses to the identity operation (i.e. leaving the combinatorial object unchanged) if and only if the reflected loco is called $2mk$ times, for some integer $m \geq 0$. The significance of this is that if all reflected locos X_i are called $2m_i k_i$ times, the resulting code will be a cyclical Gray code.

Section 2.2.3 provided a coroutine-based algorithm for generating multi-radix numbers in lexicographic order. The reflected loco corresponding to the loco used in that example is a loco that increments the digit at index i until the digit gets to $M_i - 1$, yields `True` after each change, and then yields `False`. Afterwards, the loco decrements the digit until it gets to 0, yields `True` after each change, and yields `False` at the end. The whole process is then repeated ad infinitum. This loco is is given in Figure 3.2.

```
def multiradix_gray_local(M, a, i):
    while True:
        while a[i] < M[i] - 1:
            a[i] += 1
            yield True
        yield False
        while a[i] > 0:
            a[i] -= 1
            yield True
        yield False
```

Figure 3.2: Reflected loco to generate multi-radix numbers in Gray order.

With this loco, the problem of generating multi-radix numbers in Gray order is achieved using the coroutine

$$\prod_{i=0}^{n-1} X_i, \tag{3.3}$$

where X_i is an instance of `multiradix_gray_local` for the given i.

3.2 Generating Ideals Of Chain Posets

3.2.1 Problem Definition

An equivalent problem to the generation of multi-radix numbers in Gray order is that of generating ideals of a poset consisting of a set of chains in Gray order. Approaching this problem with the different representation provides the opportunity to introduce two new ways of linking coroutines, namely *coroutine sum* and *symmetric coroutine sum*.

Given positive integers n and k, and a set of integer *end-points* given by

$$E = \{e_0, e_1, e_2, \ldots, e_k\} \tag{3.4}$$

with

$$-1 = e_0 < e_1 < e_2 < \cdots < e_k = n - 1, \tag{3.5}$$

let $\prec_E$ be the poset on the set $\{0, 1, \ldots, n-1\}$ given by

$$e_i + 1 \prec e_i + 2 \prec \cdots \prec e_{i+1} - 1 \prec e_{i+1} \tag{3.6}$$

for $0 \leq i < k$. The Hasse diagram of an example is shown in Figure 3.3 for $E = \{-1, 1, 2, 5\}$.

For any poset $\prec$ on the set $S = \{0, 1, \ldots, n-1\}$, an *ideal* is defined as a subset $I \subseteq S$ such that for any $x, y \in S$ if $y \in I$ and $x \prec y$ then $x \in I$. In terms of the Hasse diagram, this means a subset of the vertices such that if a vertex is in the subset, so are all its descendants. Ideals will be represented as binary strings of length n. That

is, ideal $I \subseteq S$ will be represented as binary string $a = \{a_0, a_1, \ldots, a_{n-1}\}$ with $a_x = 1$ if and only if $x \in I$. With this representation, the ideal condition becomes that if $x \prec y$ then $a_x \geq a_y$.

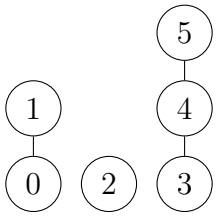


Figure 3.3: Hasse diagram of example chain poset $\prec_E$ with $E = \{-1, 1, 2, 5\}$.

Figure 3.4 shows all ideals of the poset given in Figure 3.3, of which there are a total of $3 \times 2 \times 4 = 24$, in Gray order.

3.2.2 Coroutine-based Algorithm

Let α_i be the operation of setting a_i to 1 and α_i^{-1} be setting it to 0. Now, as an example, consider having a single chain poset $\prec_E$ for $E = \{-1, 2\}$ with $n = 3$. Then the sequence of operations needed to get a reflected loco is:

$$\alpha_0, \alpha_1, \alpha_2, \alpha_2^{-1}, \alpha_1^{-1}, \alpha_0^{-1}. \tag{3.7}$$

This order inspires the definition of two new coroutine operations. The first is *coroutine sum*, or simply *cosum*, in mathematical notation $+$, which, given a list of coroutines X_1 to X_k, calls them in order until they each yield `False`, and then yields `False` and repeats from the beginning again. Note that this operator is also associative. The code for this coroutine is given in Figure 3.5. To allow for conciser

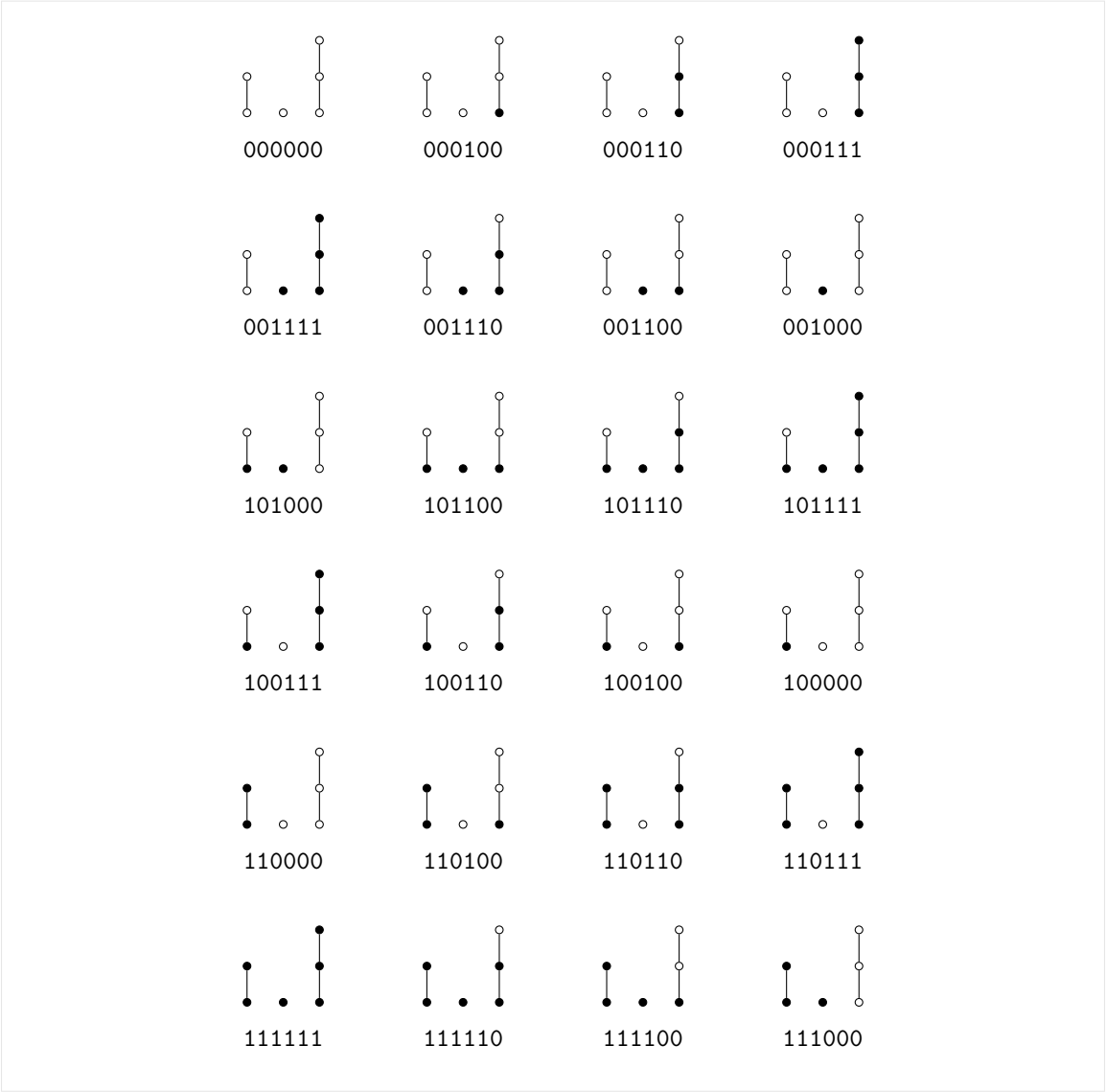


Figure 3.4: Gray code sequence of ideals of chain poset given in Figure 3.3. Filled circles represent 1 bits, empty circles 0. Order is left-to-right, then top-to-bottom.

formulations, we also define

$$\sum_{i=1}^{k} X_i = X_1 + X_2 + \cdots + X_k, \tag{3.8}$$

as is common practice with associative + operators.

Given two coroutines X and Y define the *coroutine join*, or *cojoin*, $X \vee Y$ to be

```
def cosum(*Xs):
    while True:
        for X in Xs:
            while next(X):
                yield True
        yield False
```

Figure 3.5: The coroutine sum (cosum) operator.

the coroutine that yields from X first until it yields `False`, and then yields from Y until it yields `False` and then repeats ad infinitum. The cojoin operator, with code given in Figure 3.6, will be used to define the next important operator.

```
def cojoin(*Xs):
    while True:
        for X in Xs:
            while next(X):
                yield True
            yield False
```

Figure 3.6: The coroutine join (cojoin) operator.

Given a sequence of locos X_1 to X_k, we can then define the *coroutine symmetric sum*, or simply *cosymsum*, in mathematical notation $\oplus$, as the following:

$$\bigoplus_{i=1}^{k} X_i = (X_1 + X_2 + \cdots + X_k) \vee (X_k + X_{k-1} + \cdots + X_1) \tag{3.9}$$

$$= (\sum_{i=1}^{k} X_i) \vee (\sum_{i=1}^{k} X_{k-i+1}). \tag{3.10}$$

The code for the cosymsum operator is given in Figure 3.7.

The three main operations defined on coroutines are demonstrated graphically in Figure 3.8, which shows the sequence of coroutine calls for $X + Y + Z$, $X \oplus Y \oplus Z$, and $X \times Y$.

Now, to get a reflected loco for the purposes of this section, a binary equivalent of

```
from combgen.common import cosum, cojoin


def cosymsum(*Xs):
    XY = cosum(*Xs)
    YX = cosum(*reversed(Xs))
    return cojoin(XY, YX)
```

Figure 3.7: The coroutine symmetric sum (cosymsum) operator.

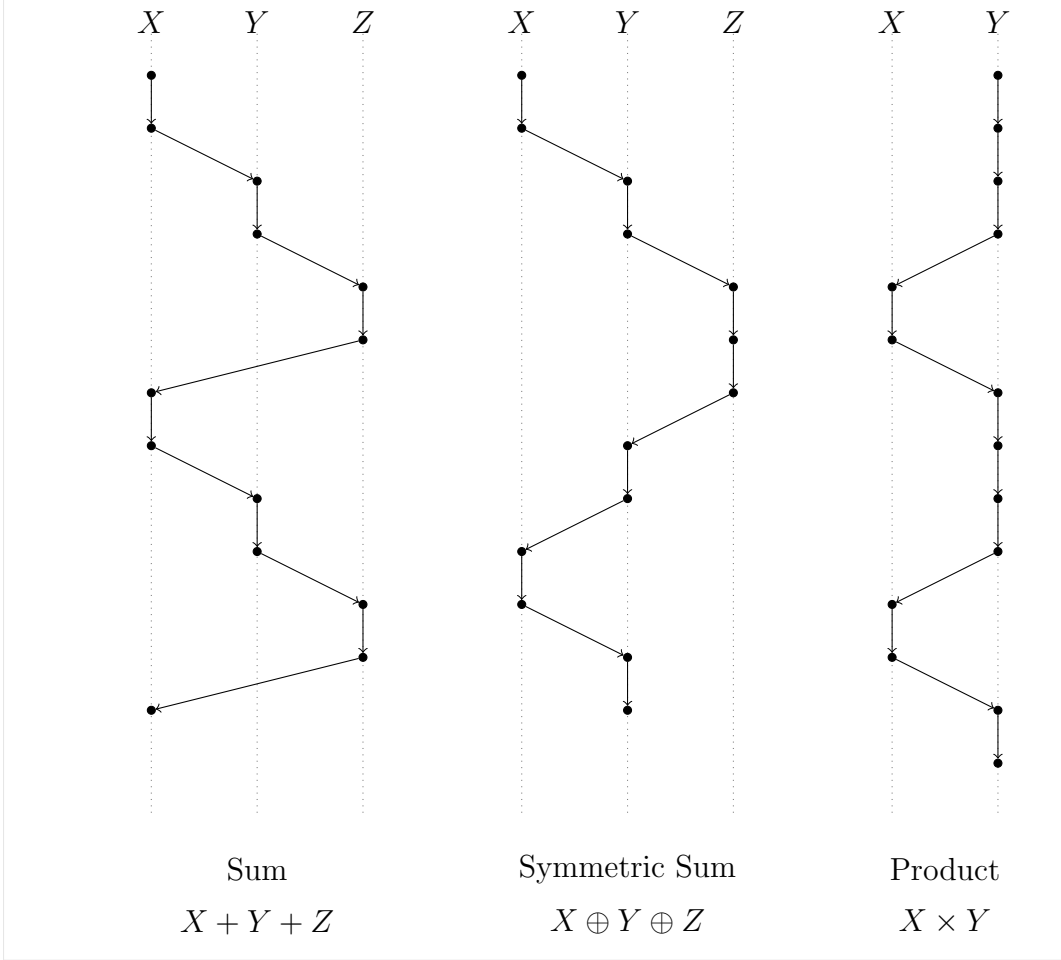


Figure 3.8: The coroutine sum, symmetric sum, and product operations.

the loco given in Figure 3.2 is needed. While we can use the same loco with $M_i = 2$ for all i, for simplicity a specialized reflected loco is provided in Figure 3.9.

Let X_i be an instance of this reflected loco for the given i, which switches bit i

```
def chain_poset_ideals_local(a, i):
    while True:
        a[i] = 1 - a[i]
        yield True
        yield False
```

Figure 3.9: Loco to generate ideals of a poset consisting of chains in Gray order.

and yields `True` then `False` after each switch. Then for each chain $i \prec i+1 \prec \cdots \prec j-1 \prec j$ the coroutine given by

$$X_i \oplus X_{i+1} \oplus \cdots \oplus X_{j-1} \oplus X_j \tag{3.11}$$

performs exactly the operations

$$\alpha_i, \alpha_{i+1}, \ldots, \alpha_{j-1}, \alpha_j, \alpha_j^{-1}, \alpha_{j-1}^{-1}, \ldots, \alpha_{i+1}^{-1}, \alpha_i^{-1} \tag{3.12}$$

in order, as needed. We then have the following coroutine to generate ideals of the poset $\prec_E$ for a given set of end-points $E = \{e_0, e_1, e_2, \ldots, e_k\}$, in Gray order:

$$\prod_{i=0}^{k-1} \bigoplus_{j=e_i+1}^{e_{i+1}} X_j. \tag{3.13}$$

For example, for $E = \{-1, 1, 2, 5\}$, the ideals are generated by the coroutine

$$(X_1 \oplus X_0) \times X_2 \times (X_5 \oplus X_4 \oplus X_3). \tag{3.14}$$

This leads to the setup code given in Figure 3.10.

It is interesting to note that the coroutines, linked in this way, can continue to run after finishing the combinatorial generation task. In the cases of reflected locos, calling the lead coroutine after the first generation results in the combinatorial objects

```
from combgen.common import cosymsum, coproduct
from .local import chain_poset_ideals_local as X


def setup(n, E):
    a = [0] * n
    Y = []
    for j in range(len(E) - 1):
        Z = [X(a, i) for i in range(E[j] + 1, E[j + 1] + 1)]
        Y.append(cosymsum(*Z))
    lead = coproduct(*Y)
    return a, lead
```

Figure 3.10: Setup of locos to generate ideals of a poset consisting of chains in Gray order.

being generated in reverse order the second time around, and back to the initial order the third time, and so on.

3.2.3 Generalization To Ideals Of Forest Posets

The results from the previous section can be generalized to generate ideals of forest posets, leading to an algorithm quite similar to the Koda-Ruskey algorithm [10]. To be more precise, we define *forest posets* as posets consisting of a set of disjoint *tree posets*, and define a tree poset as a poset in which the Hasse diagram is a rooted tree, with each non-root element covering exactly one other element. (An element y is said to be *covering* another element x if $x \prec y$ and no other element z with $x \prec z \prec y$ exists.) Figure 3.11 shows an example of a tree poset.

Ideals are defined in the same as last section. Figure 3.12 shows all ideals of the poset given in Figure 3.11 in Gray order, as produced by the algorithm given in this section.

Assume that x is the root of a tree poset, and that $y_1, y_2, \ldots, y_k$ are its immediate successors. That is, $x \prec y_i$ for all $1 \leq i \leq k$ and there exists no z such that $x \prec z \prec y_i$. For example, in Figure 3.11, $x = 0$ and $y_1 = 1$, $y_2 = 3$ and $y_3 = 7$. Then

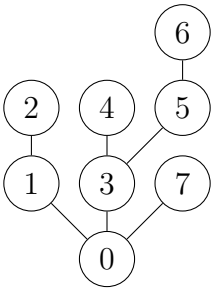


Figure 3.11: Hasse diagram of example tree poset.

each combination of ideals of subtrees rooted at each y_i is an ideal of the whole tree provided that x is included in the ideal. Letting $C(z)$ be the coroutine that generates all ideals of the tree rooted at z in Gray order, the following recursion holds:

$$C(x) = X_x \oplus \prod_{i=1}^{k} C(y_i), \tag{3.15}$$

with base case happening when $k = 0$, namely the leaves of the tree, for which $C(z) = X_z$, where X is the exact same loco as in last section.

For example, in Figure 3.11, we have $2, 4, 6$ and 7 as the leaves, giving

$$C(2) = X_2, \tag{3.16}$$

$$C(4) = X_4, \tag{3.17}$$

$$C(6) = X_6, \tag{3.18}$$

$$C(7) = X_7. \tag{3.19}$$

With those base cases, we have

$$C(1) = X_1 \oplus C(2) = X_1 \oplus X_2, \tag{3.20}$$

$$C(5) = X_5 \oplus C(6) = X_5 \oplus X_6, \tag{3.21}$$

$$C(3) = X_3 \oplus (C(4) \times C(5)) = X_3 \oplus (X_4 \times (X_5 \oplus X_6)), \tag{3.22}$$

$$C(0) = X_0 \oplus (C(1) \times C(3) \times C(7)) \tag{3.23}$$

$$= X_0 \oplus (\overbrace{(X_1 \oplus X_2)}^{C(1)} \times \underbrace{(X_3 \oplus \underbrace{(X_4 \times \overbrace{(X_5 \oplus X_6)}^{C(5)})}_{C(4)}) \times X_7}_{C(3)}). \tag{3.24}$$

This lead coroutine generates the ideals of the tree poset in the Gray order given in Figure 3.12. Extending this result to forests instead of just trees can be achieved by using the coproduct of the coroutines that generate each independent tree in the forest. The result can also be further generalized to produce ideals of completely-acyclic posets [9, 1].

3.3 Generating Permutations In Gray Order

3.3.1 Problem Definition

Define $\mathbb{S}_n$ for $n \geq 1$ as the set of bijections from the set $\{1, 2, \ldots, n\}$ onto itself. A function π in $\mathbb{S}_n$ is called a *permutation of length n*. In this thesis, when $n \leq 9$, permutations will be written in *one-line notation*, that is, as simple sequences of digits. For example, the permutation π with $\pi_1 = 2$, $\pi_2 = 3$ and $\pi_3 = 1$ will be written as simply 231.

A *transposition* is a permutation $\sigma^{a,b}$ such that $\sigma^{a,b}_x = x$ for all x except for exactly two distinct a and b such that $\sigma^{a,b}_a = b$ and $\sigma^{a,b}_b = a$. That is, a transposition

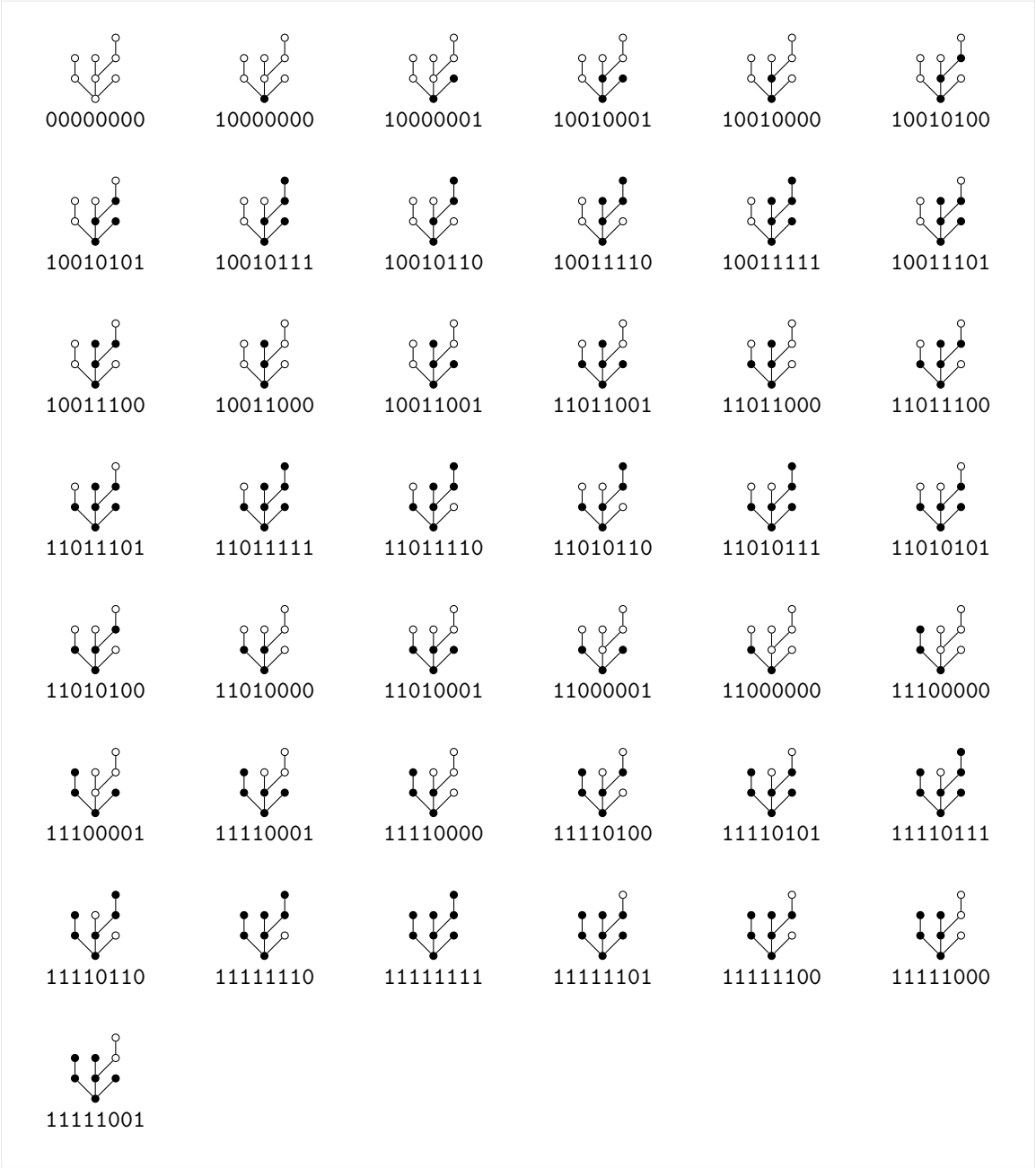


Figure 3.12: Ideals of the poset given in Figure 3.11. Filled circles represent 1 bits, empty circles 0. Order is left-to-right, then top-to-bottom.

is a permutation that switches the positions of exactly two elements. For example, $\sigma^{4,5} = 12354 \in \mathbb{S}_5$ is a transposition of 4 and 5. Every permutation can be written as a composition of transpositions, and while there may be many distinct ways of

composing the permutation as transpositions, for any given permutation the parity of the number of transpositions that it can be transposed into is always the same (see for example [13]). Based on this, the set $\mathbb{S}_n$ is partitioned into two sets, called the set of *even* and the set of *odd* permutations, based on the parity of the number of transpositions for any decomposition of the permutation into transpositions. For example, 4312 can be written as $\sigma^{1,3} \circ \sigma^{1,2} \circ \sigma^{1,4}$ and is hence an odd permutation.

An *inversion* of a permutation π is defined as a pair of indices (a, b) such that $a < b$ and $\pi_a > \pi_b$. In the previous example, the number of inversions of 4312 is 5, given by pairs $(1, 2)$, $(1, 3)$, $(1, 4)$, $(2, 3)$, and $(2, 4)$. Another basic result in abstract algebra is that the parity of the number of inversions is the same as the parity of the number of transpositions in any decomposition of the permutation into transpositions. The parity of the permutation can therefore be calculated using the parity of the number of inversions.

The Gray order for $\mathbb{S}_n$ is defined as any listing of the elements of $\mathbb{S}_n$ in which two consequent permutations differ by exactly one transposition. In any Gray code for $\mathbb{S}_n$ the parity of the permutations switches each time.

3.3.2 Coroutine-Based Algorithm

The coroutine-based algorithm in this section is based on Steinhaus-Johnson-Trotter (SJT) [8]. Defined recursively, the SJT Gray order for $\mathbb{S}_n$ is given by first recursively generating the SJT Gray order for $\mathbb{S}_{n-1}$, and then inserting n into each permutation starting from the very right and moving to the very left if the permutation is even, and from the very left and to the right if the permutation is odd. The base case is when $n = 1$, in which case $\mathbb{S}_n$ has only a single item 1. For example, for $n = 2$, the base case permutation 1 is an even permutation, so 2 starts at the very right, giving 12 and then moves to the left giving 21. Continuing in this manner, for $n = 3$, we

start with 12 which is an even permutation, so 3 starts at the very right and moves to the left, giving 123, 132 and 312, in that order. Then 21 is odd, so 3 starts at the very left and moves right, giving 321, 231 and 213 in that order.

An iterative implementation of SJT is possible by keeping track of the direction that each element in the permutation is moving in [3]. All elements begin with negative direction, indicating movement to the left. At each point an *active element* x is moved in its direction. Before the active element is moved, the next element in the direction of its move, say y, is checked and if x is less than y (i.e. $x < y$), x is not moved in that direction and instead the direction for x is switched, and the active element is changed to $x - 1$. Otherwise, after every move, the active element is changed to n. The code for this iterative algorithm is shown in Figure 3.13

```
from combgen.helpers.permutations import transpose


def gen_all(n):
    pi = [n + 1] + list(range(1, n + 1)) + [n + 1]
    inv = pi[:]
    d = [-1] * (n + 2)
    x = n  # x is the active element
    yield pi[1:-1]
    while x > 0:
        y = pi[inv[x] + d[x]]  # y is the element next to x in direction d[x]
        if x < y:
            d[x] = -d[x]  # Switch direction
            x -= 1  # Change active element to x - 1
        else:
            transpose(pi, inv, x, y)
            yield pi[1:-1]  # New permutation is generated
            x = n  # Change active element to n
```

Figure 3.13: Iterative algorithm to generate all permutations in Steinhaus-Johnson-Trotter Gray order.

The coroutine-based algorithm is similar to the iterative algorithm in that each element maintains a direction of movement. However, unlike the iterative algorithm, the active element is not maintained by the loco and ends up being maintained im-

plicitly based on how the locos are linked together using a coproduct.

For $1 \leq x \leq n$ let X_x be a reflected loco that moves x to left first until the element to left of x is greater than x, yields `False`, and then moves x to the right until the element to the right of x is greater than x, yields `False`, and then repeats the same process again and again. For this process to terminate, a greater element in both directions needs to exist. This is achieved by prepending and appending $n+1$ to π. That is, we always assume that $\pi_0 = \pi_{n+1} = n+1$ throughout. These two indexes will never change and are simply used as "barriers".

For example, the permutation 123 is represented as $\pi = 41234$, and with that π, X_3 performs the following operations:

$$\sigma^{3,2}, \sigma^{3,1}, \sigma^{3,1}, \sigma^{3,2}, \sigma^{3,2}, \sigma^{3,1}, \ldots. \tag{3.25}$$

Note that transpositions are involutions, meaning $(\sigma^{a,b})^{-1} = \sigma^{a,b}$, so the above sequence of operations conforms to our definition for reflected locos.

The code for this loco is shown in Figure 3.14. The `transpose` function is provided in the supplementary code given in Appendix A.1.

```
from combgen.helpers.permutations import transpose


def sjt_local(pi, inv, x):
    d = -1
    while True:
        y = pi[inv[x] + d]  # y is the element next to x in direction d
        if x < y:
            d = -d  # Switch direction
            yield False
        else:
            transpose(pi, inv, x, y)
            yield True
```

Figure 3.14: Loco to generate permutations in Steinhaus-Johnson-Trotter Gray order.

With this reflected loco, the correct active element will be maintained by setting

the lead coroutine to be the coproduct

$$\prod_{x=1}^{n} X_x, \tag{3.26}$$

and the SJT Gray code for permutations can be generated using this lead coroutine.

3.4 Generating Linear Extensions

3.4.1 Problem Definition

In the previous section an algorithm for generating all permutations in Gray order was given. Generation of all permutations can be seen as a special case of generating all *linear extensions* of a poset. Assume P is a poset given by partial-order $\prec$ on the set $S = \{1, 2, \ldots, n\}$. We also make the assumption, by means of a relabelling if necessary, that for all $x, y \in S$ if $x \prec y$ then $x < y$. This extra assumption means that, without loss of generality, the identity permutation $\iota = 123\ldots n$ is always a valid linear extension, regardless of the poset in question, and therefore simplifies the initialization step. The posets given in Section 3.1.2 are examples of posets satisfying this assumption. A *linear extension* of P is defined as a permutation π of S such that for all $x, y \in S$ if $x \prec y$ then $\pi^{-1}(x) < \pi^{-1}(y)$. That is, if $x \prec y$ then x occurs to the left of y in the one-line notation for π.

The running example of poset used in this section will be the *fence* or *zig-zag* poset, defined for $n \geq 1$ as the poset on the set $S = \{1, 2, \ldots, n\}$ given by

$$1 \prec \lceil \frac{n}{2} \rceil + 1 \succ 2 \prec \cdots \succ \lceil \frac{n}{2} \rceil - 1 \prec n \succ \lceil \frac{n}{2} \rceil. \tag{3.27}$$

For example, for $n = 5$ the poset is given by

$$1 \prec 4 \succ 2 \prec 5 \succ 3. \tag{3.28}$$

The Hasse diagram for this example is shown in Figure 3.15.

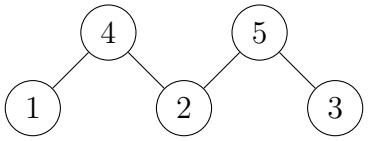


Figure 3.15: Zig-zag poset for $n = 5$ given by $1 \prec 4 \succ 2 \prec 5 \succ 3$.

Linear extensions of the zig-zag poset are sometimes referred to as *alternating permutations* [30]. Figure 3.16 shows the linear extensions of the zig-zag poset for $n = 5$, each linear extension represented as a linear Hasse diagram.

In code, posets will be represented using functions that take two variables a and b and return `True` if $a \prec b$ and `False` otherwise. The code for the zig-zag poset is given in Appendix A.2.

3.4.2 Coroutine-based Algorithm

The coroutine-based algorithm in this section is based on the Varol-Rotem algorithm [16], which is an algorithm for generating all linear extensions of a poset in a similar way as SJT. In Varol-Rotem, elements are transposed with elements to their left until a "boundary" is hit, similar to SJT, except that in Varol-Rotem the boundary is determined by the poset. The algorithm is as follows: let x be the *active* element, initialized to n. At each step, let y be the element to the left of x. Then there are two possibilities: either $y \prec x$ or $y \parallel x$, since if $x \prec y$ then the invariant of the algorithm, namely that π will always be a linear extension of the given poset,

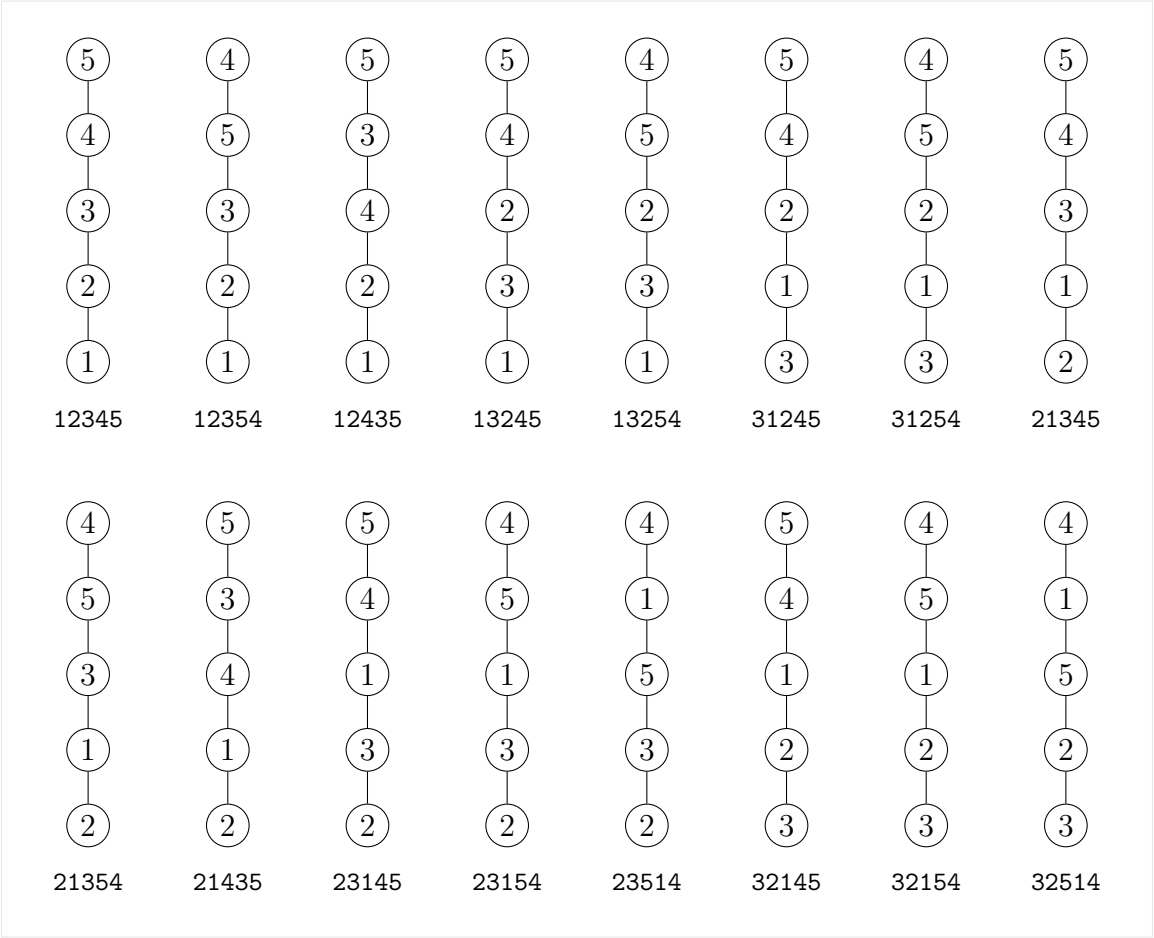


Figure 3.16: Linear extensions of the zig-zag poset for $n = 5$, as generated by the Varol-Rotem algorithm. Order is left-to-right then top-to-bottom.

is violated. The algorithm then behaves based on these two possibilities. If $y \prec x$ then x can not move to the left and the algorithm does a cyclic left-shift to place x back where it had started at the beginning (i.e. index x, since the starting linear extension is the identity permutation), and set the active element x to be $x - 1$. Otherwise, the algorithm transposes x and y (meaning it moves x to the left) and set the active element x to the initial element n. An iterative implementation is given in Figure 3.17. The `move` helper function attempts to move x in the given direction while maintaining π as a linear extension of the given poset. If such a move would violate the partial-order, the function returns `False` without modifying the linear ex-

tension. Otherwise, if the move is successful, it returns `True`. The code for the helper functions `move`, `left_cyclic_shift` and `add_min_max` is provided in Appendix A.1.

```
from combgen.helpers.permutations import move, LEFT, left_cyclic_shift
from combgen.helpers.posets import add_min_max


def gen_all(n, poset):
    poset = add_min_max(poset, 0, n + 1)
    pi = list(range(n + 2))
    inv = pi[:]
    yield pi[1:-1]
    x = n
    while x > 1:
        if move(pi, inv, x, LEFT, poset):
            yield pi[1:-1]
            x = n
        else:
            left_cyclic_shift(pi, inv, inv[x], x)
            x -= 1
```

Figure 3.17: Iterative Varol-Rotem algorithm for generating all linear extensions of a poset.

For example, consider the zig-zag poset with $n = 5$ given in Figure 3.15. Starting with the identity permutation 12345, the first step of the Varol-Rotem is to try to move 5 to the left, which is possible since $4 \not\prec 5$, so the next linear extension is 12354. Next, the algorithm tries to move 5 to the left again, but this time $3 \prec 5$ so such a move is not possible and hence a cyclic left-shift is done to put 5 back where it started, giving 12345 and the active element is changed to 4 which is then moved to the left, giving 12435 since $3 \not\prec 4$, and the active element is reset back to be 5, but in the next two steps, since $3 \prec 5$ and $2 \prec 4$, the linear extension gets set back to 12345 and the active element gets set to 3. Next, 3 is moved to the left since $3 \not\prec 2$, giving 13245, and the active element is reset to 5, which is then moved to the left giving 13254, etc.

This iterative algorithm is based on the recursive idea that linear extensions of $\{1, 2, \ldots, n-1, n\}$ can be generated by first generating all linear extensions of

$\{1, 2, \ldots, n-1\}$ and then in each of them, placing n at the very right and then moving it as far left as possible without violating the partial-order.

It is important to note that because of the cyclic left-shift in the algorithm, the generated linear extensions will not be in Gray order. Because of this, our loco in this section will not be a reflected loco. Instead, the loco for the coroutine-based algorithm will follow the iterative algorithm closely, moving x to the left while possible, and yielding `True` after each such successful move, and doing the cyclic left-shift and yielding `False` when such a move is not possible. The code for this loco is given in Figure 3.18.

```
from combgen.helpers.permutations import move, LEFT, left_cyclic_shift


def varol_rotem_local(poset, pi, inv, x):
    while True:
        while move(pi, inv, x, LEFT, poset):
            yield True
        left_cyclic_shift(pi, inv, inv[x], x)
        yield False
```

Figure 3.18: Loco for generating all linear extensions of a poset in Varol-Rotem order.

With this loco, all linear extensions of a given poset can be generated by setting the lead coroutine to be the coproduct

$$\prod_{x=1}^{n} X_x, \tag{3.29}$$

similar to the coroutine-based SJT algorithm given in the previous section.

The next section will provide an algorithm for generating *signed* linear extensions of posets in Gray order, which can also be used to generate linear extensions in near-Gray order, with adjacent linear extensions differing by one or two adjacent transpositions.

3.5 Generating Signed Linear Extensions In Gray Order

3.5.1 Problem Definition

In the previous section, the Varol-Rotem algorithm for generating linear extensions of a poset, and a coroutine-based variation of it, were discussed. And as noted in that section, the resulting sequence of linear extensions is not in Gray order due to the cyclic left-shift once a boundary is hit. It is not always possible to generate linear extensions in Gray order (with adjacent linear extensions differing by one adjacent transposition), since the transposition graph of the linear extensions will always be bipartite, and the number of vertices in the two parts of the bipartite graph do not always differ by a maximum of one. The graph is bipartite because transpositions always change the parity of the permutation.

Figure 3.19 shows an example of a poset for which no Gray order of linear extensions exist: the associated adjacent transposition graph for the linear extensions of the poset does not have a Hamiltonian path, since there are four odd linear extensions and only two even linear extensions. The existence of a Gray code would imply that the difference between the two numbers should be -1, 0 or 1.

Pruesse and Ruskey show in [14] that if each linear extension is given a sign, which is either positive or negative, then the resulting set of signed linear extensions can be generated by either transposing two adjacent elements in each step, or switching the sign. This is equivalent to taking the Cartesian product of the transposition graph with a single edge. An example of a Gray code of signed linear extensions for the poset given in Figure 3.19 is shown in Figure 3.20.

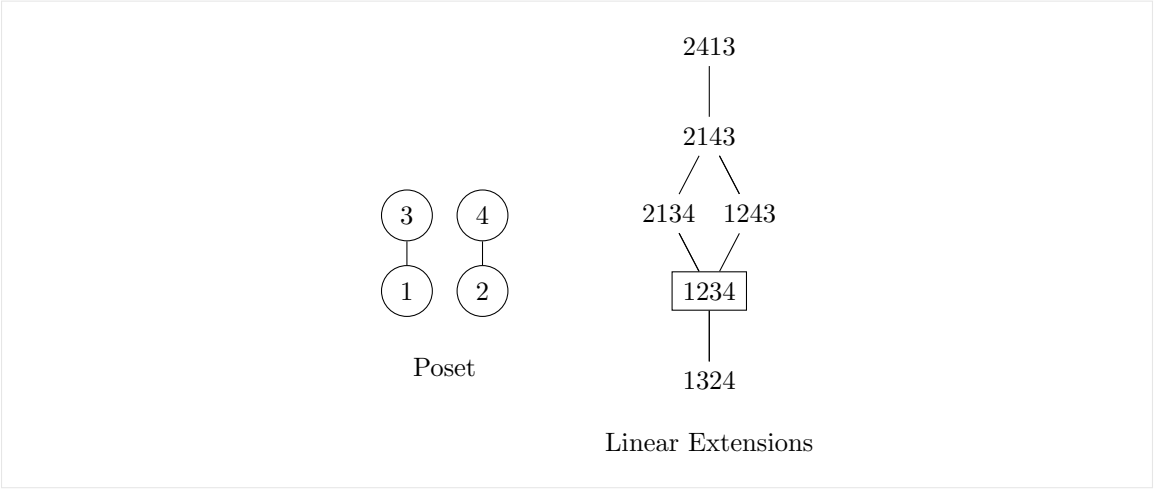


Figure 3.19: Poset with $1 \prec 3$ and $2 \prec 4$ and its linear extensions graph. Adjacent linear extensions differ by one transposition.

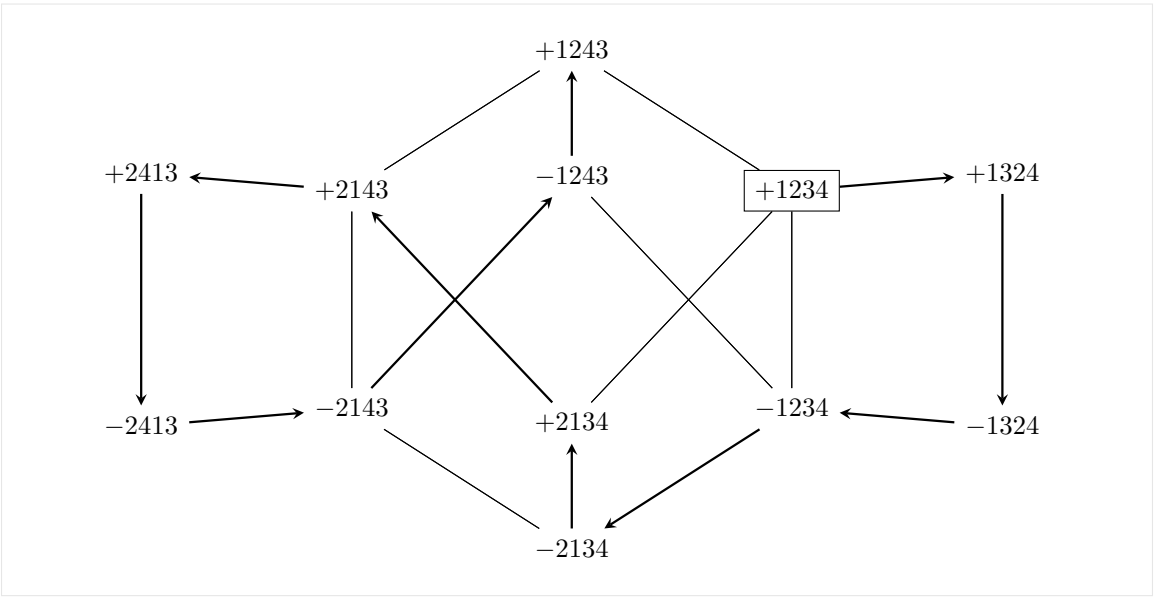


Figure 3.20: Graph corresponding to signed linear extensions of poset with $1 \prec 3$ and $2 \prec 4$, and the Hamiltonian path traversed by the Pruesse-Ruskey algorithm.

3.5.2 Coroutine-based Algorithm

The coroutine-based algorithm in this section is based on the Pruesse-Ruskey algorithm [14].

A basic description of the Pruesse-Ruskey algorithm is provided here, and the

reader is referred to [14] for more details and correctness proofs. For a poset P defined by partial-order $\prec$ on the set S, define $P \setminus A$ for some $A \subseteq S$ to be the poset on the set $S \setminus A$ given by partial-order $\prec'$ with $a \prec' b$ if and only if $a, b \in S \setminus A$ and $a \prec b$. In simpler terms, $P \setminus A$ is the poset given by removing elements of A from P.

Let $M(P)$ be the set of minimal elements of poset P. More precisely, let

$$M(P) = \{x \in S : \text{there exists no } y \in S \text{ such that } y \prec x\}. \tag{3.30}$$

For example, for the zig-zag poset with $n = 5$ given in Figure 3.15, $M(P) = \{1, 2, 3\}$, and with the poset given in Figure 3.19, $M(P) = \{1, 2\}$. The Pruesse-Ruskey algorithm is recursive, with the recursion having two cases. First, if $M(P)$ has exactly one element, say $M(P) = \{a\}$, then the signed linear extensions of P can be generated by prepending a to all the signed linear extensions of the poset $P \setminus \{a\}$ while maintaining the same sign, since if a is the unique minimal element then it must be the case that $a \prec x$ for all $x \neq a$.

If $M(P)$ has more than one element, then pick distinct a and b in $M(P)$ and let $P' = P \setminus \{a, b\}$. Then for any signed linear extension of P', if the sign is positive ab is prepended to the signed linear extension. Afterwards, the algorithm moves ab and b to the left and right, with a always to the left of b, following a path as given in Figures 3.21 and 3.22, depending on whether b can move an even or odd number of times to the right before hitting a constraining element.

If the sign of the signed linear extension is negative prepend ba to the signed linear extension and switch the sign to positive, and then similar to the previous case, follow the paths given in Figures 3.21 and 3.22, except this time with a and b swapped (that is, b will always be to the left of a.)

Calculating the a and b pairs can be done as a pre-calculation step, by processing the poset using an algorithm that closely resembles a breadth-first search topological

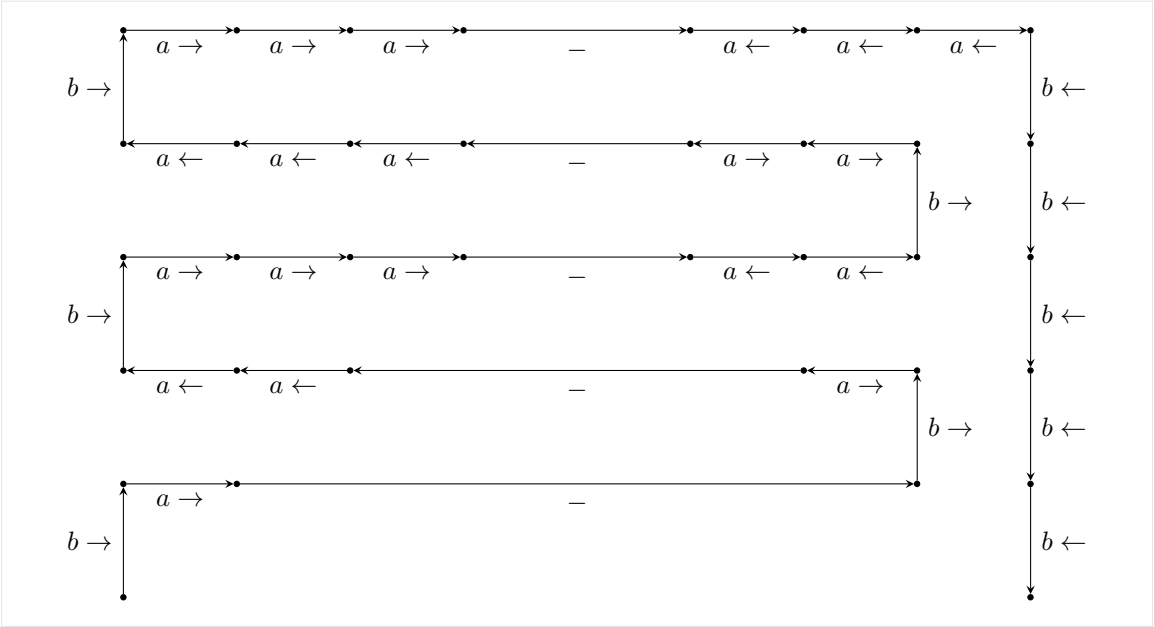


Figure 3.21: Sequence of a, b moves for odd number of possible b moves. Left/right arrow next to a or b on edge labels indicates direction of the move, $-$ indicates a sign switch.

sort. This algorithm (which is described fully in [14]) can also be used to setup the initial linear extension. This is an important step since the Pruesse-Ruskey generation algorithm makes the assumption that the a and b pairs are initially adjacent to each, with unique minimal elements interjected in-between them.

Assuming that the a and b pairs are already calculated, the first part of the coroutine-based Pruesse-Ruskey is creating a loco to traverse the paths given in Figures 3.21 and 3.22. Four variables are used by the `pruesse_ruskey_local` loco, as shown in Figure 3.23. These variables are `mra`, `mrb`, `mla` and `typical`. The two variables `mra` and `mrb` keep track of how many times a and b, respectively, have moved to the right. They are used to determine the correct number of moves when the loco starts moving a or b, respectively, to the left. The `typical` variable is used to keep track of whether a has moved to the right or not. In the atypical case, in which `typical` is set to `False`, the path becomes simply b moving to the right, switching sign, and then moving to the left. In the typical case, with `typical` set to `True`, the

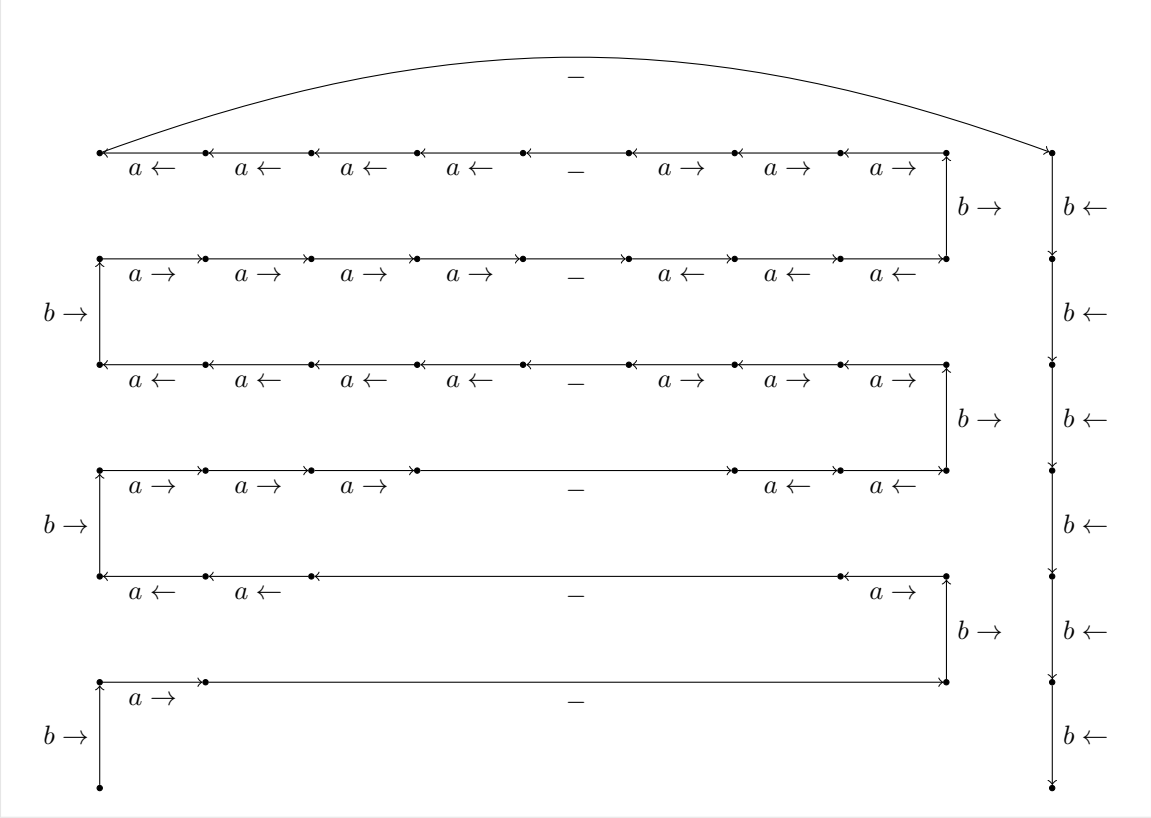


Figure 3.22: Sequence of a, b moves for even number of possible b moves. Left/right arrow next to a or b on edge labels indicates direction of the move, $-$ indicates a sign switch.

path will resemble one of the paths given in Figures 3.21 and 3.22. Finally, the `mla` variable holds how many times a needs to move to the left in each row of the path, and is calculated depending on whether `mrb` is even or odd. The reader is encouraged to carefully study the code and the diagrams to understand how the various cases in the diagrams are handled by the code.

One difficulty here is that the product coroutine will need to know when the loco switches sign during the path, since when that happens the product coroutine needs to transpose a and b. For this reason, the loco for this algorithm will yield three possible values. `True` and `False` will be used as before, to indicate a move along the path, and reaching the end of the path, respectively. In addition, a constant `SWITCH_SIGN`, defined to be -1, is used to indicate a change in sign. Since in Python

```
from combgen.helpers.permutations import move, LEFT, RIGHT

SWITCH_SIGN = -1  # Signals change of sign in a-b path


def pruesse_ruskey_local(poset, pi, inv, a, b):
    def extended_poset(x, y):  # Extend poset so that a < b
        return (x, y) == (a, b) or poset(x, y)

    while True:
        mrb = 0
        typical = False
        while move(pi, inv, b, RIGHT, extended_poset):
            mrb += 1
            yield True
            mra = 0
            while move(pi, inv, a, RIGHT, extended_poset):
                typical = True
                mra += 1
                yield True
            if typical:
                yield SWITCH_SIGN
                mla = mra + (-1 if mrb % 2 else 1)  # a left moves
                for __ in range(mla):
                    move(pi, inv, a, LEFT)
                    yield True
        if typical and mrb % 2 == 1:
            move(pi, inv, a, LEFT)
            yield True
        else:
            yield SWITCH_SIGN
        for __ in range(mrb):
            move(pi, inv, b, LEFT)
            yield True
        yield False
```

Figure 3.23: Loco to generate signed linear extensions of a poset in Pruesse-Ruskey Gray order. The code follows the paths given in Figures 3.21 and 3.22

`bool(-1)` is `True`, this means that `SWITCH_SIGN` will still count as a move along the path, as it should, provided the yielded result of the coroutines is treated as a boolean (as is done when the result is given to an `if` conditional, for example). The code for this loco is provided in Figure 3.23.

As mentioned, in this algorithm the product operation needs to transpose a and b when the loco yields `SWITCH_SIGN`. This requires having a specialized product

operator that takes a coroutine X and the a and b pair, creates $Y_{a,b}$ and $Y_{b,a}$ (where $Y_{a,b}$ is an instance of the loco given in Figure 3.23 for the given a and b), and initializes the active right-hand-side coroutine Y to $Y_{a,b}$. The operator then proceeds the same way as the previously defined coproduct operator $X \times Y$, with the addition that whenever Y yields `SWITCH_SIGN`, the active right-hand-side coroutine is switched (switching between $Y_{a,b}$ and $Y_{b,a}$) and a and b are transposed in the linear extension. The code for this specialized coroutine product is given in Figure 3.24.

```
from combgen.helpers.permutations import transpose
from .local import pruesse_ruskey_local, SWITCH_SIGN


def pruesse_ruskey_product(poset, pi, inv, X, a, b):
    Y_ab = pruesse_ruskey_local(poset, pi, inv, a, b)
    Y_ba = pruesse_ruskey_local(poset, pi, inv, b, a)
    while True:
        for result in Y_ab:
            if not result:
                break
            yield result

        for result in X:
            if not result:
                break
            if result is SWITCH_SIGN:
                transpose(pi, inv, a, b)
                Y_ab, Y_ba = Y_ba, Y_ab
            yield True
            for result in Y_ab:
                if not result:
                    break
                yield result
        yield False
```

Figure 3.24: Specialized coroutine product for the coroutine-based Pruesse-Ruskey algorithm.

One final detail that is different in this algorithm is that the base-case consists of a simple change of sign. Because of this, we use a variation of the `barrier` abstraction that was mentioned in Section 2.2.4. The `pruesse_ruskey_barrier` coroutine continuously yields `SWITCH_SIGN` and then `False` and is shown in Figure 3.25. The locos

```
from .local import SWITCH_SIGN


def pruesse_ruskey_barrier():
    while True:
        yield SWITCH_SIGN
        yield False
```

Figure 3.25: Specialized barrier coroutine for the coroutine-based Pruesse-Ruskey algorithm.

and one instance of the barrier coroutine then need to be linked together using the specialized coproduct, as shown in Figure 3.26. At the very top level, if `SWITCH_SIGN` is yielded the sign of the linear extension is switched, as shown in Figure 3.27. This algorithm can be used to generate all linear extensions in semi-Gray order (adjacent linear extensions differing by one or two adjacent transpositions) by ignoring the sign and reporting every other generated linear extension [14].

```
from .product import pruesse_ruskey_product
from .barrier import pruesse_ruskey_barrier


def setup(n, poset, a_b_pairs):
    pi = list(range(n + 1))
    inv = pi[:]
    pi[0] = 1
    lead = pruesse_ruskey_barrier()
    for a, b in a_b_pairs[::-1]:
        lead = pruesse_ruskey_product(poset, pi, inv, lead, a, b)
    return lead, pi
```

Figure 3.26: Setup code for the coroutine-based Pruesse-Ruskey algorithm.

```
from .local import SWITCH_SIGN
from .setup import setup


def gen_all(n, poset, a_b_pairs):
    lead, pi = setup(n, poset, a_b_pairs)
    yield pi
    for result in lead:
        if not result:
            return
        if result is SWITCH_SIGN:
            pi[0] = -pi[0]
        yield pi
```

Figure 3.27: Using the lead coroutine in the coroutine-based Pruesse-Ruskey algorithm.

Chapter 4

Conclusions

In this thesis we presented the first comprehensive study of the use of coroutines as an alternative approach to solving combinatorial generation problems. Several abstract coroutine operations were introduced and used to provide coroutine-based algorithms for a range of combinatorial generation problems. The abstract operations can be reused in different algorithms, which means that solving new problems involves only the design of a problem-specific local coroutine. The local coroutines then need to be linked together using the abstracted operations (e.g. coproduct and cosymsum). These abstractions together form an algebra of coroutines that allows for very concise and mathematical formulation of the algorithms. Of the coroutine-based algorithms given, generation of multi-radix numbers in lexicographic and Gray orders, generation of permutations in Steinhaus-Trotter-Johnson order, generation of linear extensions of posets in Varol-Rotem order, and generation of signed linear extensions of posets in Pruesse-Ruskey order are new.

There are several opportunities for further research in this area. First, the computational complexity of the coroutine-based algorithms can be formally studied. The amortized run-time complexity of the algorithms would be of particular interest,

since that is the most often looked at measure for combinatorial generation algorithms. With the algebraic abstractions, the possibility of proving general results about the amortized run-time complexity of algorithms using the coroutine operations, as a function of the amortized complexity of the locos, can be an interesting area of further research.

Secondly, the mentioned algebraic formulation of connected coroutines can be further formalized and studied. This might allow for algebraic correctness proofs of the algorithms, as well as potential for ways to optimize the run-time by algebraic manipulation of coroutine expressions.

Thirdly, more problems and algorithms can be approached using coroutines and the associated algebra. Particular problems that might allow for efficient coroutine-based algorithms are generation of various forms of trees or graphs, in Gray order.

Appendix A

Supplementary Code

A.1 Permutations

```
def transpose(pi, inv, x, y):
    i, j = inv[x], inv[y]
    inv[x], inv[y] = j, i
    pi[i], pi[j] = pi[j], pi[i]
```

Figure A.1: Transposing x and y in a permutation

```
def left_cyclic_shift(pi, inv, i, j):
    x = pi[i]
    for k in range(i, j):
        t = pi[k + 1]
        inv[t] -= 1
        pi[k] = t
    inv[x] = j
    pi[j] = x
```

Figure A.2: Left cyclic shift of a permutation starting from index i and ending at index j.

A.2 Posets

```
from .transpose import transpose

LEFT = -1
RIGHT = 1


def move(pi, inv, x, d, poset=None):
    if inv[x] + d >= len(pi) or inv[x] + d < 1:
        return False
    y = pi[inv[x] + d]
    if poset and ((d == LEFT and poset(y, x)) or (d == RIGHT and poset(x, y))):
        return False
    transpose(pi, inv, x, pi[inv[x] + d])
    return True
```

Figure A.3: Moving i in a permutation in direction given by d while maintaining π as a linear extension of the given poset. Used in SJT, Varol-Rotem, and Pruesse-Ruskey algorithms.

```
def zigzag(n):
    k = n // 2 + n % 2  # k = ceil(n/2)

    def poset(a, b):
        if a == 1:
            return b == k + 1
        if n % 2 and a == k:
            return b == n
        if a > k:
            return False
        return b in (a + k - 1, a + k)

    return poset
```

Figure A.4: The zig-zag poset (AKA fence poset) defined programmatically.

```
def add_min_max(poset, minimum, maximum):
    def poset_with_min_max(a, b):
        return a == minimum or b == maximum or poset(a, b)

    return poset_with_min_max
```

Figure A.5: Adding unique minimum and maximum elements to a given poset.

Bibliography

References

[1] Richard S Bird. "Spider spinning for dummies". In: *Advanced Functional Programming*. Springer, 2009, pp. 39–65.

[2] Melvin E. Conway. "Design of a Separable Transition-diagram Compiler". In: *Commun. ACM* 6.7 (June 1963), pp. 396–408. ISSN: 0001-0782.

[3] S. Even. *Algorithmic combinatorics*. Macmillan, 1973.

[4] Solomon W Golomb et al. *Shift register sequences*. Vol. 78. Aegean Park Press Laguna Hills, CA, 1982.

[5] Brenden Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall, 2013, p. 20. ISBN: 9780133390094.

[6] Donald E. Knuth. *Selected Topics in Computer Science, Part II*. Lecture Note Series. See page 3 of the notes entitled "Generation of combinatorial patterns: Gray codes." Blindern, Norway: University of Oslo, Institute of Mathematics, Aug. 1973, p. 3.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, 2nd Edition*. Addison-Wesley Professional, 1973.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms*. Addison-Wesley Professional, 2014.

[9] Donald E. Knuth and Frank Ruskey. "Deconstructing Coroutines". In: *Selected Papers on Computer Languages*. CSLI Publications, 2002, pp. 545–574.

[10] Yasunori Koda and Frank Ruskey. "A Gray code for the ideals of a forest poset". In: *Journal of Algorithms* 15.2 (1993), pp. 324–340.

[11] Christopher D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Lecture Notes in Computer Science. Springer, 1980.

[12] Ana Lucia De Moura and Roberto Ierusalimschy. "Revisiting Coroutines". In: *ACM Trans. Program. Lang. Syst.* 31.2 (Feb. 2009), 6:1–6:31. ISSN: 0164-0925.

[13] C.C. Pinter. *A Book of Abstract Algebra: Second Edition*. Dover Publications, 2012.

[14] Gara Pruesse and Frank Ruskey. "Generating the Linear Extensions of Certain Posets by Transpositions". In: *SIAM Journal on Discrete Mathematics* 4.3 (1991), pp. 413–422.

[15] Malcolm James Smith. "Generating Spanning Trees". MA thesis. University of Victoria, 1985.

[16] Yaakov. L. Varol and Doron Rotem. "An algorithm to generate all topological sorting arrangements". In: *The Computer Journal* 24.1 (1981), pp. 83–84.

[17] D. J. Wheeler. "The Use of Sub-routines in Programmes". In: *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*. ACM '52. ACM, 1952, pp. 235–236.

Online Sources

[18] *Apache MPM worker*. URL: http://httpd.apache.org/docs/2.2/mod/worker.html.

[19] David Beazley. *A Curious Course on Coroutines and Concurrency*. Mar. 2009. URL: http://www.dabeaz.com/coroutines/.

[20] *Boost Coroutines*. URL: http://www.boost.org/doc/libs/master/libs/coroutine/doc/html/coroutine/coroutine.html.

[21] *C# Language Specification Version 5.0*. URL: http://www.microsoft.com/en-us/download/confirmation.aspx?id=7029.

[22] Evan Czaplicki. *Escape from Callback Hell*. URL: http://elm-lang.org/learn/Escape-from-Callback-Hell.elm.

[23] Gregory Ewing. *PEP 380 – Syntax for Delegating to a Subgenerator*. Feb. 2009. URL: http://legacy.python.org/dev/peps/pep-0380/.

[24] *Light-weight concurrency for C featuring coroutines*. URL: https://code.google.com/p/libconcurrency/.

[25] *New in JavaScript 1.7*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/1.7.

[26] John Ousterhout. *Why Threads Are A Bad Idea (for most purposes)*. Sept. 1995. URL: http://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf.

[27] Guido van Rossum. *PEP 3156 – Asynchronous IO Support Rebooted: the "asyncio" Module*. Dec. 2012. URL: http://legacy.python.org/dev/peps/pep-3156/.

[28] Guido van Rossum and Phillip J. Eby. *PEP 342 – Coroutines via Enhanced Generators*. May 2005. URL: http://legacy.python.org/dev/peps/pep-0342/.

[29] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. *PEP 255 – Simple Generators*. May 2001. URL: http://legacy.python.org/dev/peps/pep-0255/.

[30] *The On-Line Encyclopedia of Integer Sequences (OEIS) - Sequence A000111*. URL: http://oeis.org/A000111.

[31] *The Python Language Reference - Expressions*. URL: https://docs.python.org/2/reference/expressions.html.

[32] *The Tornado Web Sever*. URL: http://www.tornadoweb.org/en/stable/.

[33] *Tulip: Async I/O for Python 3*. URL: https://www.youtube.com/watch?v=1coLC-MUCJc.

[34] Barry Warsaw et al. *PEP 1 – PEP Purpose and Guidelines*. June 2000. URL: http://legacy.python.org/dev/peps/pep-0001/.

[35] Ka-Ping Yee and Guido van Rossum. *PEP 234 – Iterators*. Apr. 2001. URL: http://legacy.python.org/dev/peps/pep-0234/.