Olympiads in Informatics, 2023, Vol. 17, 99–119
© 2023 *IOI, Vilnius University*
DOI: 10.15388/ioi.2023.08

# Understanding and Designing Recursive Functions via Syntactic Rewriting

Tom VERHOEFF
*Mathematics and Computer Science, Eindhoven University of Technology*
*Groene Loper 5, 5612 AE, Eindhoven, Netherlands*
*e-mail: t.verhoeff@tue.nl*

**Abstract.** Recursion is considered a challenging programming technique by many students. There are two common approaches intended to help students understand recursion. One of them is based on the operational semantics of function execution involving a stack, where students trace the execution of a recursively defined function for some concrete arguments. The other approach is based on the axiomatic semantics involving inductive reasoning with the contract of the recursively defined function. The former approach is not so helpful when designing recursive functions, whereas the latter can be helpful (being a special case of divide and conquer) but contracts can be hard to discover.

In this article, I will show a third approach. It is based neither on an operational nor an axiomatic semantics. Rather, it involves a rewriting semantics using program transformation by substitution, thereby inlining function calls. We show that this approach not only may help understanding, but can also be used to design recursive functions.

**Keywords:** computer science, education, programming, recursion.

## 1. Introduction

I have written about recursion before (Verhoeff, 2018, 2021) and until recently I would not have thought to have something significant to add. But while preparing a programming Q&A session for first-year mathematics students, I was struck by a (for me) new idea, which is the topic of this article.

Verhoeff (2018) presents the two common approaches to understand recursion. As example, consider the following Python code for the recursively defined function `print_bit_strings` (we left it undocumented for now on purpose):

```python
def print_bit_strings(n: int, s: str = "") -> None:
    if n == 0:
        print(s)
    else:
```

```
5          for b in "01":
6              print_bit_strings(n - 1, s + b)
```

The call `print_bit_strings(2)` prints

```
00
01
10
11
```

One could wonder why there is a need for this extra parameter `s`, and why the recursive call on line 6 has argument `s + b` rather than `b + s`.

To understand this, students could use the *operational* approach, where they trace the execution of a specific call of the recursive function in question through multiple levels of subsequent recursive calls. Beginning programmers perceive this execution as magical and confusing, because there is a single definition of the given recursive function, but multiple calls are simultaneously executing that same piece of code independent of each other. That is, each invocation can be at a different location in that code, and the parameters and other local variables can have different values. The execution of a recursive function traverses an imaginary *dynamic call tree* (Verhoeff, 2018, §3.1), where each node corresponds to the execution of a call, which then gives rise to zero or more subsequent recursive calls. The active invocations form a *root path* in this tree, and the 'instruction pointer' and values of local variables are stored on a *stack*, that grows and shrinks as the root path.

For example, the call `print_bit_strings(2, "s")` gives rise to the call tree in Fig. 1. This approach helps in understanding how a stack machine can correctly execute a recursively defined function in an imperative programming language. But in my experience it is neither helpful for reasoning about (the correctness) of recursive function definitions nor for designing them.

Alternatively, there is the *axiomatic* approach (Verhoeff, 2018, §4), which requires a specification of the recursive function in terms of a *contract* consisting of a *precondition* and a *postcondition*, such that

- *if* the precondition is satisfied *before* the function call,
- *then* the postcondition is satisfied *after* the function call.

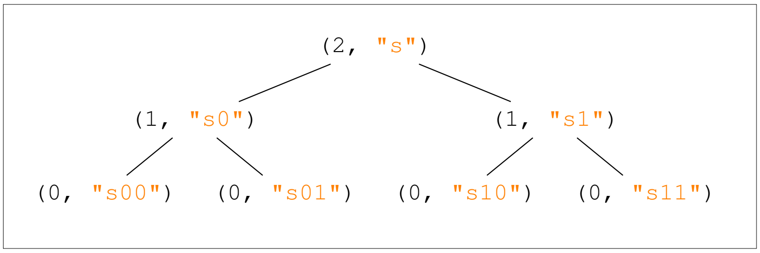

Fig. 1. Call tree for `print_bit_strings(2, "s")`, only showing parameters.

The docstring for `print_bit_strings(n, s)` could read:

```
 7      """Print s + t for all strings t over "01" of length n,
 8      in lexicographic order.
 9
10      Assumption: n >= 0
11      """
```

That is, its precondition is `n >= 0` and its postcondition is: ‘for all strings `t` over `"01"` of length `n`, strings `s + t` have been printed’. To reason about the call with parameters `(n, s)`, we assume as *induction hypothesis* that calls with parameters `(n_, s_)` where `n_ < n` work as specified by the contract. The design of function `print_bit_strings` can be argued as follows. The goal is to prove that lines 2–6 satisfy the contract, under the assumption that the recursive call on line 6 satisfies its contract; that is, it satisfies ‘if `n > 0`, then for all strings `u` of length `n - 1` over `"01"`, it prints strings `s + b + u`’.

- If `n == 0`, then there is only one string `t` of length `n`, viz. the empty string. Observe that `s` extended with the empty string equals `s` (line 3). Thus, the contract is fulfilled by printing just `s`.
- If `n > 0` then extensions of length `n` over `"01"` start with a single bit, say `b`, in `"01"`, followed by `n - 1` more bits over `"01"`. Thus, by the induction hypothesis, the loop on lines 5–6 prints all required strings.

In this reasoning style, the induction hypothesis is sometimes referred to as the *recursive leap of faith* (Roberts, 1986; Rubio, 2018). I find this terminology unfortunate, because faith has nothing to do with it. There is already good terminology, viz. *induction hypothesis*. Trusting the compiler, runtime system, and hardware to execute recursive definitions faithfully, could be called a leap of faith. Students need to understand this implementation only once, e.g., by tracing an execution through the call tree and observing the role of the stack. But this is not needed (nor helpful) to understand specific recursive definitions, and certainly not for designing them.

## 2. Syntactic Rewriting

There is a third kind of semantics for programming languages, viz. based on *rewriting*. It is typically used to describe the semantics of more advanced language constructs in terms of simpler language constructs. For example, `i += 1` can be rewritten into `i = i + 1`. That way, the meaning of `+=` is defined, without the need to speak of how `+=` works operationally, nor how to reason about it axiomatically. This rewriting is a purely syntactic operation and is also called a *semantics-preserving program transformation*. One sometimes calls the notation `+=` *syntactic sugar*, because it does not make the language more expressive. It is a mere abbreviation that can be eliminated by rewriting, also known as *syntactic desugaring*.

## 2.1. *Rewriting Function Calls, by Inlining*

In the absence of recursion, the function mechanism is syntactic sugar for abbreviations that can be eliminated by a program transformation. Consider a void[1] function definition without **`return`** statements of the form

```
def func(param_1, param_2, ...):
    body # containing param1, param2, ..., but no return
```

The call `func(expr_1, expr_2, ...)` can be eliminated as follows.

1. Replace the call `func(expr_1, expr_2, ...)` by

   ```
   param_1, param_2, ... = expr_1, expr_2, ...
   body
   ```

   where `param_1, param_2, ...` are local variables.
2. Systematically rename any local variables in `body` whose name clashes with a name occurring in the context of the call.

When the expressions `expr_i` are free of *side effects*, also known as *referential transparency* (which will be the case in our examples), Step 1 can be replaced by a *double substitution*:

1.a. Replace the call `func(expr_1, expr_2, ...)` by `body`,
1.b. in which all occurrences of parameters `param_i` are simultaneously replaced by the corresponding argument expressions `(expr_i)`. The parentheses are needed to guarantee the proper evaluation order.

The result of this transformation is a program that is semantically equivalent to the original program. As a *compiler optimization* technique and as a *code refactoring* technique, it is also known as *inlining*. In Lambda Calculus, the double substitution corresponds to *β-reduction*.

For example, consider the Python function definition

```
def f(x, y):
    z = x * y
    print(z)
```

Then we can rewrite as follows

```
    f(z - 1, z + 1)
    print(z)

# inline call: x, y = z - 1, z + 1

```

[1] In some programming languages void functions are known as procedures. In Python, their body does not contain **`return`** `expr`. For non-void functions and **`return`**, see Appendix A.

```
    z1 = (z - 1) * (z + 1) # z1: fresh local variable
    print(z1)
    print(z)
```

Note that in this case the parentheses are really needed to preserve the evaluation order of operators. When they are not needed, we silently omit them.

Non-recursive function definitions can be completely eliminated by inlining all their calls. This may result in faster execution, at the expensive of a larger code foot print. Recursive function definitions cannot be completely eliminated this way. Well, they can, provided we allow *infinite* program texts. Conceptually, there is nothing wrong with an infinite program text. Termination of the recursion corresponds to guaranteeing that only a finite (though unbounded) part of that infinite program gets executed.

## 2.2. *Rewriting* **`if`***-statements, by Deleting Dead Branches*

It turns out that for the examples in §3 we also need rewriting rules for **`if`**-statements with constant conditions. These are the two relevant rewrite rules:

```
    if True:
        statement_suite_1
    else: # unreachable
        statement_suite_2

# delete dead else-branch

    statement_suite_1
```

and

```
    if False: # unreachable
        statement_suite_1
    else:
        statement_suite_2

# delete dead if-branch

    statement_suite_2
```

## 2.3. *Rewriting Expressions, by Constant Folding*

The final rewrite step that we use in the examples below is that of evaluating an expression involving only constants. As a compiler optimization technique this is known as *constant folding*. We label such rewrites by `simplify`.

## 3. Example for Understanding Via Rewriting

Let’s rewrite the call `print_bit_strings(2)`:

```
    print_bit_strings(2)

# inline call: n, s = 2, ""

    if 2 == 0:
        print("")
    else:
        for b in "01":
            print_bit_strings(2 - 1, "" + b)

# delete dead if-branch: 2 != 0 ; simplify: 2 - 1 == 1 ; "" + b == b

    for b in "01":
        print_bit_strings(1, b)

# inline call: n, s = 1, b; rename local variables

    for b1 in "01":
        # print_bit__strings(1, b1)
        if 1 == 0:
            print(b1)
        else:
            for b2 in "01":
                print_bit_strings(1 - 1, b1 + b2)

# delete dead if-branch: 1 != 0 ; simplify: 1 - 1 == 0

    for b1 in "01":
        for b2 in "01":
            print_bit_strings(0, b1 + b2)

# inline call: n, s = 10, b1 + b2; rename local variable

    for b1 in "01":
        for b2 in "01":
            # print_bit_strings(0, b1 + b2)
            if 0 == 0:
                print(b1 + b2)
            else:
                for b3 in "01":
                        print_bit_strings(0 - 1, b1 + b2 + b3)

# delete dead else-branch: 0 == 0

    for b1 in "01":
        for b2 in "01":
            print(b1 + b2)
```

So, by *equational reasoning*, the recursive function applied to argument 2 is equivalent to 2 nested **for**-loops. It is now believable that for argument **n**, the call is equivalent to **n** nested **for**-loops, since each recursive call adds a level of nesting:

```
    # print_bit_strings(n)
    for b1 in "01":
        for b2 in "01":
            ...
                for bn in "01":
                    print(b1 + b2 + ... + bn)
```

The role of (accumulation) parameter `s` can apparently be viewed as collecting the state of all enclosing **for**-loops. We see that recursion enables one to write programs with a variable number of nested **for**-loops. Without recursion this is often not possible in an imperative programming language. However, see §4 for a way of doing it in Python without recursion.

To my delight, the IntelliJ IDE can do the refactoring steps of inlining calls of recursive functions and deleting dead branches in **if**-statements, but only for Java. It can't do them for Python (neither can VS Code). It can inline nonrecursive Python function calls, but not for direct recursive functions. Surprisingly, if the recursive function is duplicated and turned into a pair of mutually recursive functions, the IDE will refactor their function calls properly.

Finally, it is important to note the difference between tracing the execution and rewriting the program as ways of understanding recursion. Execution tracing suffers from the exponential blow up in branching recursion, whereas syntactic program rewriting does not. The latter reasons about the (unexecuted) program as a whole.

## 4. Designing Recursive Function Definitions

The rewriting approach described in the previous section to help understand recursive function definitions can also be used to help design such functions. The steps are as follows.

1. Write down a non-recursive program that solves a particular instance of the problem.

2. Decide which part will be done in a single layer of the recursion, which part will be handled by recursion, and which part is handled by preceding layers. Focus on the perspective of that single layer.
3. Introduce appropriate parameters to feed in data that comes from the preceding layers, and modify and pass them on to the lower layers. In particular, there will also be a parameter for the problem size.
4. Decide on the base case(s).
5. Define the recursive function; in particular, introduce an **if**-statement to distinguish the base case(s) and the 'general' case that adds a single layer.

## 4.1. *First Design Example*

As an example, consider the problem of printing all bit strings of length `n`.

1. A straightforward non-recursive program for `n = 3` consisting of 3 nested **for**-loops:

```
for b1 in "01":
    for b2 in "01":
        for bn in "01":
            print(b1 + b2 + b3)
```

2. A single layer of the recursion does one **for**-loop, say with control variable `b2`, the loops nested inside will be handled by the recursive call, and the outer loops were done in the preceding layers:

```
# print_bit_strings(3)
##################
for b1 in "01":  # preceding layers
##################
    for b2 in "01":  # < this will be done in one layer
        #########################
        for b3 in "01":           # handled by
            print(b1 + b2 + b3)   # recursion
        #########################
```

   Note that the part handled by recursion concerns a *generalization*, because those **for**-loops do not just contain `print(b3)`. Rather, they contain `print(b1 + b2 + b3)`. So, the generalization is that the recursive call must print `s = b1 + b2` extended with `b3`. That first part must come in via an extra parameter, say `s`. This also means that the call itself will receive that parameter, but then its value will be `b1`, received from the preceding layers. For the top-level call, we can take `s = ""`, because it is the unit of string concatenation.

3. Thus, we obtain the following structure:

```
##################
for b1 in "01":  # preceding layers
##################
    # print_bit_strings(2, b1) # call being designed
    for b2 in "01":
        print_bit_strings(1, b1 + b2) # recursive call
        # which should inline as
        # for b3 in "01":
        #     print(b1 + b2 + b3)
```

4. In case `n == 0`, there are no **`for`**-loops, and only a `print` statement, which can be fed with the parameter `s`.
5. This leads to the following definition, which we have seen before:

```
def print_bit_strings(n: int, s: str = "") -> None:
    """Print s + t for all strings t over "01" of length n,
    in lexicographic order.

    Assumption: n >= 0
    """

    if n == 0:
        # t == ""
        print(s)
    else:
        # n > 0, write t == b + u for b in "01"
        for b in "01":
            print_bit_strings(n - 1, s + b)
```

   Note the default value `s = ""`, corresponding to an empty context of preceding recursive layers.

In this approach, one can use the IDE refactoring technique known as *extract function*. It will introduce appropriate parameters to feed in values that are must be supplied.

## 4.2. *Second Design Example*

Consider a binary tree of depth 2 (Fig. 2, left). How can it be used to create a binary tree of depth 3? One way is to copy the binary tree of depth 2 and combine the two instances with a fork on top (Fig. 2, middle). This is the view we followed in the preceding example. But one can also grow a binary fork on each of the leaves of the binary tree of depth 2 to get a binary tree of depth 3 (Fig. 2, right).

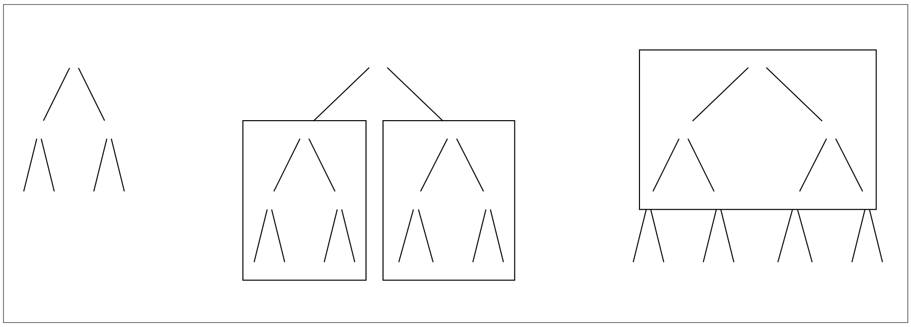

Fig. 2. Binary tree of depth 2 (left); two ways (middle, right) of constructing a binary tree of depth 3 from trees of depth 2.

Let's see how that alternative choice can be worked out in case of function `print_bit_strings`.

1. The non-recursive program is the same as above in §4.1.
2. A single layer of the recursion does one **for**-loop, but now the recursion will do the *outer* **for**-loops and the preceding layers did the *inner* loops:

```
# print_bit_strings(3)
##################
for b1 in "01":   # handled by recursion
##################
    for b2 in "01":   # < this will be done in one layer
        ##########################
        for b3 in "01":             # preceding
            print(b1 + b2 + b3)   # layers
        ##########################
```

   This may look strange, but bear with me. Apparently, in the recursion, bit strings of length one shorter are produced, and these still need to be extended and printed. So, again, we see a *generalization*: rather then just print all bit strings, we need to apply a function to them, and this function is going to be an extra parameter, say `f`. The call we are designing receives this parameter, which represents the work to be done in the preceding layers (inner loops) For the top-level call, we can take `f = print`. The recursive call will receive a function (as parameter) that adds one **for**-loop around the given `f`.
3. Thus, we obtain the following structure:

```
# print_bit_strings(2, f)
def g(s: str) -> None:
    for b2 in "01":
        f(s + b2)

    print_bit_strings(1, g)
    # which should eventually inline as
    # for b1 in "01":
    #     g(s + b1)

    # where def f(s: str) -> None:
    ########################
        for b3 in "01":    # preceding
            print(s + b3)  # layers
    ########################
```

4. In case `n == 0`, there are no `for`-loops and `f` can just be applied to `""`.
5. This leads to the following recursive definition:

```
def print_bit_strings(n: int,
                      f: Callable[[str], None] = print
                     ) -> None:
    """Apply f to each string over "01" of length n,
    in lexicograhpic order.

    Assumption: n >= 0
    """
    if n == 0: # only the empty string has length 0
        f("")
else: # n > 0
        def g(s: str) -> None:
            for b in "01":
                f(s + b)

        print_bit_strings(n - 1, g)
```

Let's now see if we can understand this recursive definition by rewriting the call `print_bit_strings(2)`:

```
    print_bit_strings(2)

# inline call: n, f = 2, print

    if 2 == 0:
        print("")
    else:
        def g(s: str) -> None:
            for b in "01":
                print(s + b)

        print_bit_strings(2 - 1, g)

# delete dead if-branch: 2 != 0 ; simplify: 2 - 1 == 1

    def g(s: str) -> None:
        for b in "01":
            print(s + b)

    print_bit_strings(1, g)

# inline call: n, f = 1, g; rename new local function g -> g2

    def g(s: str) -> None:
        for b in "01":
            print(s + b)

    if 1 == 0:
        g("")
    else:
        def g2(s: str) -> None: # renamed local function
            for b in "01":
                g(s + b)

        print_bit_strings(1 - 1, g2)

# delete dead if-branch: 1 != 0 ; simplify: 1 - 1 == 0

    def g(s: str) -> None:
        for b in "01":
            print(s + b)

    def g2(s: str) -> None:
        for b in "01":
            g(s + b)

    print_bit_strings(0, g2)

# inline call g(s + b); rename control variables

    def g2(s: str) -> None:
        for b1 in "01": # renamed control variable
            for b2 in "01": # renamed control variable
                print((s + b1) + b2)

    print_bit_strings(0, g2)

# inline call: n, f = 0, g2; rename new local function g -> g3

    def g2(s: str) -> None:
        for b1 in "01":
            for b2 in "01":
                print((s + b1) + b2)

    if 0 == 0:
        g2("")
    else:
        def g3(s: str) -> None:
            for b in "01":
                g2(s + b)

        print_bit_strings(0 - 1, g3)

# delete dead else-branch: 0 == 0

    def g2(s: str) -> None:
        for b1 in "01":
            for b2 in "01":
                print((s + b1) + b2)

    g2("")

# inline call g2("")

    for b1 in "01":
        for b2 in "01":
            print("" + b1 + b2)

# simplify: "" + b1 = b1

    for b1 in "01":
        for b2 in "01":
            print(b1 + b2)
```

Now, the extra parameter (`f`) accumulates the work to be done, and in the base case it is applied to the empty string. Such a parameter is called a *continuation*. Observe that this definition of `print_bit_strings` is *tail recursive* and thus can easily be transformed into a **`while`**-loop:

```python
def print_bit_strings(n: int) -> None:
    """Print all strings over "01" of length n,
i    n lexicographic order.

    Assumption: n >= 0
    """
    f = print

    while n > 0:
        def g(s: str, f=f) -> None:
            for b in "01":
                f(s + b)

        n, f = n - 1, g

    f("")
```

Note that here `g` needs an extra parameter `f` with default value `f`, to ensure that the definition of `g` is a *closure* that properly captures the function object currently bound to the name `f` at the moment of definition. Without that `f` parameter, the definition of `g` would contain an ‘open’ (un-dereferenced) name `f`, which will be looked up during execution of `g` to find the value bound to `f` at the moment of execution, rather than at the moment of definition.

By redefining `f` directly, `print_bit_strings` can be simplified to

```python
    f = print

    for _ in range(n):
        def f(s: str, f=f) -> None:
            for b in "01":
                f(s + b)

    f("")
```

Note that `f` here is not defined recursively, since the `f` in its body is the parameter, which is bound to the earlier value of `f`.

Apparently, in Python one can write a non-recursive program that behaves like a variable number of nested loops. Actually, the program constructs a function that behaves like those nested loops. Thus, this is a form of *metaprogramming*.

## 5. Conclusion

I have described and illustrated how a rewriting semantics can help understand and design recursive function definitions. This approach complements the traditional approaches based on operational semantics (execution tracing) and axiomatic semantics (contracts). I am not claiming that the approach via rewriting is the best, but I do find it better than execution tracing, because (i) it is purely syntactic, (ii) does not need a stack to distinguish the states of concurrently active recursive calls, (iii) nor does it suffer from any exponential blow up. Furthermore, the rewriting approach may help in discovering and formulating generalized contracts, which are needed for the approach via axiomatic semantics.

There is a clear relationship to *functional programming*, whose semantics can be based on Lambda Calculus, which has a rewriting semantics via $\beta$-reduction (inlining). Note that a rewriting semantics does not need a stack. In this article, I have shown that this approach also can work for imperative programs. It thus allows *equational reasoning* on the level of whole programs. However, some care is needed, in particular when `return` statements are used and when expressions can have side effects, causing a lack of *referential transparency*.

The approach via rewriting would benefit from IDE support for the relevant *refactoring techniques*, because manual rewriting is tedious and error-prone. Unfortunately, such support is currently rather limited (JetBrains IntelliJ can do it for Java).

A word of warning is in place concerning the examples. The rewriting approach can help to come up with recursive definitions. But these definitions may still have performance issues. There are other techniques to improve the performance of recursively defined functions, e.g., see Verhoeff (2018). Appendix B offers better ways of printing bit strings in Python. To show the power of a purely functional programming language, I have included some Haskell programs for generating bit strings in Appendix C.

I hope also that I have shown some nifty uses of Python. Maybe the recent Python compiler named Codon (Shajii, 2023) can make Python interesting for use in programming contests such as the IOI.

## Acknowledgment

I would like to thank Berry Schoenmakers and Sten Wessel (TU Eindhoven, Netherlands) and Radu Negulescu (Ontario, Canada) for helping me improve this article.

## References

Roberts, E. (1986). *Thinking Recursively* (1st Ed.). Wiley.
Rubio-Sánchez, M. (2018). *Introduction to Recursive Programming.* Taylor & Francis.

DOI: 10.1201/9781315120850

Shajii, A. *et al.* (2023). Codon: A Compiler for High-Performance Pythonic Applications and DSLs. In: *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. ACM, pp. 191–202. DOI: 10.1145/3578360.3580275

Verhoeff, T. (2018). A Master Class on Recursion. In: *Adventures Between Lower Bounds and Higher Altitudes*. Lecture Notes in Computer Science Vol. 11011. Springer, pp. 610–633. DOI: 10.1007/978-3-319-98355-4_35

Verhoeff, T. (2021). Look Ma, Backtracking without Recursion. (IOI Conference 2021). *Olympiads in Informatics*, 15, 119–132. DOI: 10.15388/ioi.2021.10

Verhoeff, T. (2023). Git repository with source code for "Understanding and Designing Recursive Functions via Syntactic Rewriting". (Accessed 29 April 2023) `https://gitlab.tue.nl/t-verhoeff-software/code-for-understanding-recursion`

**T. Verhoeff** is Assistant Professor in Computer Science at Eindhoven University of Technology, where he works in the group Software Engineering & Technology. His research interests are support tools for verified software development and model driven engineering. He received the IOI Distinguished Service Award at IOI 2007 in Zagreb, Croatia, in particular for his role in setting up and maintaining a web archive of IOIrelated material and facilities for communication in the IOI community, and in establishing, developing, chairing, and contributing to the IOI Scientific Committee from 1999 until 2007.

## Appendix A. Python Example with Non-void Function

Eliminating the call of a non-void[2] function is a bit more involved than for void functions. To keep things simple, we will assume that the return statements in the function body occur at *tail positions*, that is, if the `return` statement would have been a function call, it would be the last thing done in the body (a so-called tail call). Now consider a non-void function definition of the form

```
def func(param_1, param_2, ...):
    body # containing 'return expr' in tail positions only
```

A call of this function occurs as a (sub)expression, which is part of some statement, such as for example

```
    print(func(expr_1, expr_2, ...) + 1)
```

In general, such a call takes the form

```
    stmt(func(expr_1, expr_2, ...))
```

where `stmt` is a void function. Assuming, for simplicity, that the argument expressions have no side effects, it can be eliminated as follows.

1. Replace `stmt(func(expr_1, expr_2, ...)` by `body`.
2. Replace every occurrence of `return expr` in `body` by `stmt(expr)`. N.B. The expression may need to be parenthesized as `(expr)`.
3. Simultaneously replace all occurrences of parameters `param_i` by their corresponding argument expressions `(expr_i)`.
4. Systematically rename any local variables in `body` whose name clashes with a name occurring in the context of the call.

If a return statement would not occur in a tail position, then a 'jump' to the end of the body would also be needed. Note, however, that Python does not support `goto` statements (except on April's Fool Day 2004).

Here is an example involving the famous recursive factorial function:

```
def fac(n: int) -> int:
    """For n >= 0, return n factorial.
    """
    if n == 0:
        return 1
    else:
        return n * fac(n - 1)
```

[2] Non-void functions are sometimes known as fruitful functions. In Python, they contain `return expr`.

Note that the two return statements occur in tail positions. Let‘s rewrite the statement `print(fac(2))`:

```
    print(fac(2))

# inline call: n = 2

    if 2 == 0:
        print(1)
    else:
        print(2 * fac(2 - 1))

# delete dead if-branch: 2 != 0 ; simplify: 2 - 1 == 1

    print(2 * fac(1))

# inline call: n = 1

    if 1 == 0:
        print(2 * 1)
    else:
        print(2 * 1 * fac(1 - 1))

# delete dead if-branch: 1 != 0 ; simplify: 1 - 1 == 0

    print(2 * 1 * fac(0))

# inline call: n = 0

    if 0 == 0:
        print(2 * 1 * 1)
    else:
        print(0 * 2 * * 1 * fac(0 - 1))

# delete dead else-branch: 0 == 0 ; simplify: 2 * 1 == 2

    print(2 * 1 * 1)
```

In general, `print(fac(n))` rewrites to

```
    print(n * (n - 1) * ... * 2 * 1 * 1)
```

where there are `n + 1` factors in the expression.

If the body of a non-void function consists of single return statement, then one can do equational reasoning directly on the level of expressions rather than statements. For instance, consider the following definition of function `fac`:

```
def fac(n: int) -> int:
    return 1 if n == 0 else n * fac(n - 1)
```

Now, rewriting call `fac(2)` is less complicated:

```
    fac(2)

# inline call: n = 2

    1 if 2 == 0 else 2 * fac(2 - 1)

# delete dead if-branch: 2 != 0 ; simplify: 2 - 1 == 1

    2 * fac(1)

# inline call: n = 1

    2 * (1 if 1 == 0 else 2 * 1 * fac(1 - 1))

# delete dead if-branch: 1 != 0 ; simplify: 1 - 1 == 0

    2 * 1 * fac(0)

# inline call: n = 0

    2 * 1 * (1 if 0 == 0 else 0 * 2 * 1 * fac(0 - 1))

# delete dead else-branch: 0 == 0 ; simplify: 2 * 1 == 2

    2 * 1 * 1
```

## Appendix B. Better Python Solutions for Bit Strings

For completeness sake, let me note that the problem of printing all bit strings of a given length can be solved in a better way. Decompose the problem into:

1. Constructing all bit strings of a given length.
2. Printing them all.

In Python, one might be tempted to construct all those bit strings, by putting them in a list. But then they are all stored in memory before printing them (or doing whatever one wants, for instance, count them). For that reason, generator expressions were introduced in Python. They allow on-demand construction. Here is a Python solution based on Fig. 2 (middle):

```python
from typing import Iterator

def generate_bit_strings(n: int) -> Iterator[str]:
    """Yield all strings over "01" of length n,
    in lexicographic order.

    Assumption: n >= 0
    """
    if n == 0:
        yield ""
    else:  # n > 0
        yield from (b + u
                    for b in "01"
                    for u in generate_bit_strings(n - 1)
                   )
```

It can be invoked to print the bit strings like this:

```python
print(*generate_bit_strings(3), sep='\n')
```

And here is a solution based on Fig. 2 (right):

```python
def generate_bit_strings(n: int) -> Iterator[str]:
    if n == 0:
        yield ""
    else:  # n > 0
        yield from (u + b
                    for u in generate_bit_strings(n - 1)
                    for b in "01"
                   )
```

## Appendix C. Haskell Solutions for Bit Strings

It is illustrative to see the same definitions in a pure non-strict functional programming language like Haskell. Using the recursive decomposition in Fig. 2 (middle), combining two recursively grown trees of size one smaller:

```haskell
bitStrings :: Int -> [String]
-- bitStrings n = list of strings over "01" of length n (n >= 0),
-- in lexicographic order
bitStrings 0 = [""]
bitStrings n = [b : u | b <- "01", u <- bitStrings (n - 1)]
```

and when growing one tree of size one smaller and splitting all its leaves (Fig. 2, right):

```haskell
bitStrings 0 = [""]
bitStrings n = [u ++ [b] | u <- bitStrings (n - 1), b <- "01"]
```

The latter is not efficient, because appending at the end of a list is not efficient in Haskell. But this can be improved by introducing an accumulation parameter:

```haskell
gbitStrings :: [String] -> Int -> [String]
-- gbitStrings s n = [t ++ u | t <- bitStrings n, u <- s]
-- hence, gbitStrings [""] = bitStrings
gbitStrings s 0 = s
gbitStrings s n = gbitStrings [b : t | b <- "01", t <- s] (n - 1)
```

Observe that this definition is more efficient, because it now prepends to a list. Moreover, it is tail recursive, and thus the recursion can be compiled into a loop. Also note that in Haskell, lists are lazy, that is, they are only constructed in so far as needed (like the generator expressions in Python used in Appendix B).