Olympiads in Informatics, 2025, Vol. 19, 145–158

DOI: 10.15388/ioi.2025.10

The Olympiad Trap and an Old Trampoline

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

Abstract. After some reminiscing, I describe the Olympiad trap and then delve into a technique to eliminate recursion by trampolining with continuations.

Keywords: programming, recursion, recursion elimination, continuations.

1. Introduction

Since I plan to retire in October 2025, I hope you will permit me to begin with a brief reminiscence, before telling you about the *Olympiad trap*. The bulk of this article, however, concerns a technical topic: how to eliminate recursion using an old technique known as a *trampoline* with *continuations*.

I graduated in 1985 (Applied Mathematics, Eindhoven University of Technology) and started there as a PhD candidate. In that same year, TU Eindhoven somehow was invited to participate in the preliminary regional round of the *ACM International Collegiate Programming Contest* (ICPC). We didn't know that PhD candidates were (at that time) allowed to participate. So, I became the team's coach (rather than a contestant). We went to London with an ad hoc team, and they qualified to participate in the ICPC World Finals in 1987 (Saint Louis, MO). Later that year, we organized a university-wide selection contest for the next regional round of the ICPC.

One thing led to another. In 1988, 1989, and 1990, I organized the preliminary round of the ICPC for Europe, Middle-East, and Africa (EMEA) in Eindhoven. The number of regions increased and they became smaller. In 1997, I organized the North-West European Regional Contest (NWERC), and in 1999, we had the honor of hosting the ICPC Word Finals in Eindhoven (the first time that it took place outside the USA). In 2004, I received the ICPC European Founders Award for my efforts.

It was because of my ICPC experience that Ries Kock of the Netherlands Informatics Olympiad (NIO) approached me in 1994. The Netherlands had been participating in the IOI since 1990, and Ries had taken up the challenge of organizing IOI 1995 in The Netherlands. He wanted me to head the Host Scientific Committee for IOI 1995. I went

to IOI 1994, in Sweden, as an observer, and got hooked. As part of the preparations for IOI 1995, I set up the IOI International Secretariat on the (then still very young) World-Wide Web. In 1999, the IOI International Scientific Committee was established, which I chaired until 2007. In that same year, I received the IOI Distinguished Service Award. Since then, I have followed the IOI on the side.

2. The Olympiad Trap

I already mentioned that I got hooked on the IOI during my first participation as an observer. At the IOI, it feels like you are part of an important mission: discovering, stimulating, and developing young talent (in informatics). You could consider the IOI a trap, because its attractive force keeps you involved. But that is not the trap I want to discuss here. I think that the IOI itself is trapped, viz. in its own format. That is what I mean by the Olympiad trap.

Science olympiads cannot cover the full breadth of their field, certainly not when the main event is a contest. The more prestigious and popular an olympiad becomes, the more the team leaders will want to select and prepare their contestants with a focus on what is relevant for the contest. This leads to training deeply for a narrow field. This in turn makes it harder to change the olympiad's format, because many people have invested in the current format. That is, the Olympiad is trapped in its format.

It is easy to lose sight of the breadth of the field and of social aspects when developing talent. This is particularly worrisome for a field like informatics that still evolves rapidly. Algorithmic problem solving plays a much smaller role nowadays, both in (higher) CS education and in research and industry than when the IOI was established. There is a host of other topics that attract attention. Of these, Data Science and AI are newcomers. Parents are already advising their children not to study informatics, because they fear that AI will affect the job market. That is why I think it should be mandatory to augment training for olympiads with other activities on the side, to help mitigate the Olympiad trap.

Such side activities should be interesting and challenging. In this article, I will explore such a side topic. Since this topic is still related to programming, it might even be useful for IOI contestants.

3. Limits on Recursion

Recursion is a great algorithmic technique, which can lead to more compact and clearer code for various problems. But recursion also has its dangers. One danger it shares with general `while` loops is that it can be hard to reason about such programs, in particular their termination (Verhoeff, 2018, 2023). Another danger is that recursion implicitly uses memory, viz. on the *call stack*, so that deeply nested recursion can run out of memory. In fact, some programming languages, such as Python and Java, impose a (configurable) limit on the recursion depth as built-in protection against infinite recursion. The default

limit for Python is 1000 levels and for Java 256; C++ does not impose a limit other than available memory.

In (Verhoeff, 2018), I discuss various aspects of recursion, in particular, *tail recursion* and how *linear* tail recursion can be mechanically turned into a loop to avoid burdening the stack (which I will recap below). In case of *branching* recursion, it may seem that only one recursive call can be a tail call, and thus the transformation into a loop fails. In (Verhoeff, 2021), we encountered functions that break the recursion by introducing an extra parameter, and then "tying the knot of recursion" on the outside, by making the snake eat its own tail (through a fixed-point construction, which still burdens the stack). It turns out that there is another technique to break recursion and make even branching recursion tail recursive.

I will illustrate this technique through two examples in Python. Source code and visualization of the stack usage during execution is available in (Verhoeff, 2025). The first example is based on function `tri(n)` that computes the n-th triangular number (similar to factorials, but using addition, so that the numbers don't grow so fast):

```
type nat = int  # with assumption >= 0

def tri(n: nat) -> int:
    if n == 0:
        return 0
    else:
        return n + tri(n - 1)
```

The second example is `total(t)` that sums the values in binary leaf tree t:

```
@dataclass
class Leaf:
    value: int

# binary tree type with int in leaves
type Tree = Leaf | tuple[Tree, Tree]

def total(t: Tree) -> int:
    if isinstance(t, Leaf):
        return t.value
    else:  # t is binary fork
        return total(t[0]) + total(t[1])
```

Function `tri` exhibits linear recursion and `total` exhibits branching recursion; neither is tail recursive, since more work is done after the recursive calls return.

The call `tri(1000)` will result in a `RecursionError`. In case of function `tri`, the standard technique of introducing an *accumulation* parameter yields a tail recursive definition:

```
def tri_acc(n: nat, acc: int = 0) -> int:
    if n == 0:
        return acc
    else:
        return tri_acc(n - 1, acc + n)
```

And this in turn is readily transformed into a loop, which avoids the dreaded `RecursionError`:

```
25 def tri_loop(n: nat, acc: int = 0) -> int:
26     while n != 0:
27         n, acc = n - 1, acc + n
28     return acc
```

Exercise: Show how one of the recursive calls in `total` can be transformed into a tail call by introducing an accumulation parameter. See Appendix A.1 for an answer.

4. The Trampoline

Even though the transformation from tail recursion to loop, shown above, is straightforward, it needs to be done for each tail recursive function separately. There is a simple technique that introduces only one loop, which can transform all tail recursive functions, after a small intervention. We don't want nested recursive calls, but we still want to keep the computation the same. This can be accomplished by returning the recursive call itself in *unevaluated form* and let the evaluation be continued from outside the recursive function, after it has returned. Sounds magical?

Python and many other languages (including Java and C++) offer syntax to define anonymous functions, that is, without giving them an explicit name. In Python, the syntax **`lambda`** `x, y: expr` defines a nameless function of two arguments with the result `expr`, where expression `expr` typically involves `x, y`. Similarly, we can use **`lambda:`** `expr` for a function without arguments that evaluates to `expr`.

Here is what `tri_acc` looks like after the intervention to make it return an unevaluated "recursive" call:

```
29 def tri_acc_lazy(n: nat, acc: int = 0) -> Thunk[int]:
30     if n == 0:
31         return acc
32     else:
33         return lambda: tri_acc_lazy(n - 1, acc + n)
```

It is no longer "truly" recursive, because it does not make the recursive call! We say that the call is *suspended*. In order to get the typing correct, we have defined type `Thunk[A]`:

```
34 # Thunk[A]: possibly nested suspended computation of type A
35 # A should not be callable
36 type Thunk[A] = A | Callable[[], Thunk[A]]
```

Note that in Python, `Callable[[], R]` denotes the type of functions *without arguments* returning a value of type `R`. So, a `Thunk[A]` is either a value of type `A` or a function without arguments returning a `Thunk[A]`. For value `thunk` of type `Thunk[A]`,

we can test whether it is actually suspended by `callable(thunk)`. And if it is suspended, it can be resumed by calling it as `thunk()`.

By deseign, `tri_acc_lazy` always immediately returns. How can we get the final result? That is where the *trampoline* gets to the rescue, since it repeatedly resumes a suspended computation until it gets a final (non-suspended) result:

```
37 def trampoline[A](thunk: Thunk[A]) -> A:
38     while callable(thunk):  # thunk is suspended
39         thunk = thunk()  # resume it
40     return thunk
```

Therefore, we have `tri(n) == trampoline(tri_acc_lazy(n))`. Note that `tri_acc_lazy` creates only one stack frame, and `trampoline` repeatedly resumes all suspended tail calls. The control flow bounces between the trampoline and the thunked (lazy) "tail recursive" function, where `lambda:` is placed in front of every tail recursive call to suspend it. Neat, isn't it?

Some notes:

- Upon superficial reading, the definition of `tri_acc_lazy` given above looks recursive, since the body of the function definition contains a call to the function itself.
- However, it does not execute that call itself; that is left to the client code. The "recursive knot" is tied on the outside, by `trampoline`.
- For this to work, the programming language must support function *closures* that capture the current values of variables. In case of `tri_acc_lazy`, the expression `lambda: tri_acc_lazy(n -1, acc + n)` involves two local variables, viz. `n, acc`, which evaporate after the function returns. Python binds their values in the returned lambda object.
- Thunking via `lambda:` resembles the *Command* design pattern from Object-Oriented programming (Gamma *et al.*, 1994).

5. Enforcing Tail Recursion Via Continuations

Before addressing `total`, let's generalize the technique with the accumulation parameter. In general, it may not be easy to find a simple accumulation parameter to make a definition tail recursive. And in case of branching recursion, it would be useless. But there is a way that is guaranteed to work: *Continuation Passing Style*, also known as CPS (Reynolds, 1993). With CPS, you introduce an extra parameter of a *function type*, known as a *continuation*. In Python, we abbreviate that function type to `Func[A, B]`:

```
41 type Func[A, B] = Callable[[A], B]  # functions from A to B
```

We name this continuation parameter `cont`. It represents work that still needs to be done to complete the computation. An example will make this clear. Let's specify `tri_cps(n, cont) == cont(tri(n))`. Then we have in mathematical notation

- $tri\,(n) = id\,(tri\,(n)) = tri\_cps\,(n, id)$, where id is the (polymorphic) identity function defined by $id\,(a) = a$;
- for $n = 0$, we have $tri\_cps\,(0, cont) = cont\,(tri\,(0)) = cont\,(0)$;
- and for $n > 0$, $tri\_cps\,(n, cont) = cont\,(tri\,(n)) = cont\,(n + tri\,(n-1))$.
  The latter expression can be viewed as a new function applied to $tri\,(n-1)$. Which function? The function f defined by $f\,(x) = cont\,(n + x)$. In Python, that function can be expressed as `lambda x: cont(n + x)`. So, we can rewrite further

$$\begin{aligned} cont\,(n + tri\,(n-1)) &= (\lambda\, x : cont\,(n + x))(tri\,(n-1)) \\ &= tri\_cps\,(n-1, \lambda\, x : cont\,(n + x)) \end{aligned}$$

This leads to the following tail recursive (!) definition of `tri_cps`:

```
42 def tri_cps(n: nat,
43              cont: Func[int, int] = id_
44              ) -> int:
45     if n == 0:
46         return cont(0)
47     else:
48         return tri_cps(n - 1, lambda x:
49                cont(n + x)
50                )
```

The identity function serves as default continuation. We have named it `id_`, because **`id`** is already a predefined different function in Python:

```
51 def id_[A](a: A) -> A:
52     return a
```

Function `tri_cps` first accumulates a (possibly big) continuation in n steps, which it then applies to 0. The evaluation of this continuation will also burden the stack. Thus, to make this suitable for trampolining and limiting the stack load, **`lambda:`** is also needed in front of both calls of `cont` (in addition to just the recursive call to `tri_cps_lazy`):

```
53 def tri_cps_lazy(n: nat,
54                  cont: Func[int, Thunk[int]] = id_
55                  ) -> Thunk[int]:
56     if n == 0:
57         return lambda: cont(0)
58     else:
59         return lambda: tri_cps_lazy(n - 1, lambda x:
60                lambda: cont(n + x)
61                )
```

Note that the two calls to `cont` were also tail calls. (This explains the somewhat odd layout of the code.)

6. Making Branching Recursion Tail Recursive

CPS is so powerful that it can even make functions with branching recursion tail recursive. Let's see how that works by revisiting `total` defined in §3. First, we specify `total_cps(t, cont) == cont(total(t))`. Then we have

- $total\,(t) = id\,(total(t)) = total\_cps\,(t, id)$;
- for $t = Leaf\,(v)$, we have $total\_cps\,(t, cont) = cont\,(total\,(t)) = cont\,(v)$;
- for $t = (t_0, t_1)$, we have

$$\begin{aligned}
&total_cps\,(t, cont)\\
&= cont\,(total(t))\\
&= cont\,(total\,(t_0) + total\,(t_1))\\
&= (\lambda\, tt_0 : cont\,(tt_0 + total\,(t_1)))(total\,(t_0))\\
&= total_csp\,(t_0, \lambda\, tt_0 : cont\,(tt_0 + total\,(t_1)))\\
&= total_csp\,(t_0, \lambda\, tt_0 : (\lambda\, tt_1 : cont\,(tt_0 + tt_1))(total\,(t_1)))\\
&= total_csp\,(t_0, \lambda\, tt_0 : total_csp\,(t_1, \lambda\, tt_1 : cont\,(tt_0 + tt_1))).
\end{aligned}$$

Thus, we have derived the following definition for `total_cps`:

```
62 def total_cps(t: Tree,
63                cont: Func[int, int] = id_
64                ) -> int:
65     if isinstance(t, Leaf):
66         return cont(t.value)
67     else:  # t is binary fork
68         return total_cps(t[0], lambda tt0:
69                total_cps(t[1], lambda tt1:
70                cont(tt0 + tt1)
71                ))
```

You may wonder whether this definition is really tail-recursive, because it contains two calls of `total_cps`, only one of which looks like a tail call. It is, since the call with `t[1]` is suspended (but not thunked) by **`lambda`** `tt0`. That call is incorporated into the continuation, and executed when that continuation reaches a leaf. Inside that **`lambda`** `tt0`, it is a tail call.

We can now easily prepare this for the trampoline by adding **`lambda:`** (4×):

```
72 def total_cps_lazy(t: Tree,
73                     cont: Func[int, Thunk[int]] = id_
74                     ) -> Thunk[int]:
75     if isinstance(t, Leaf):
76         return lambda: cont(t.value)
77     else:  # t is binary fork
78         return lambda: total_cps_lazy(t[0], lambda tt0:
79                lambda: total_cps_lazy(t[1], lambda tt1:
80                lambda: cont(tt0 + tt1)
81                ))
```

I hope that this example convinces you that through CPS the transformation of general, possibly branching, recursion to tail recursion can in fact be automated.

However, I need to temper your expectations somewhat. Exercise: Find a recursive function definition that cannot be made tail recursive via CPS. See Appendix A.2 for an answer.

7. Defunctionalization

What did we gain? Well, CPS with trampoline avoids using the stack, and deep recursion is no longer a problem when using a language that limits the recursion depth. Memory usage, however, has shifted from the stack to the continuation. This continuation grows and shrinks (in `total_cps` it can be discarded at leaves, once it has been applied, but that may give rise to a new continuation when hitting another recursive call). The computation is accumulated in the continuation (like meta-programming) and then applied.

That continuation contains all the information needed to complete the computation: both data (operands) and functionality (operations). Since the operations are fairly limited, it is inefficient to copy them multiple times into the continuation. Can't we take that duplicate functionality out and share it? Yes, that is possible through a technique known as *defunctionalization*.

Let's first do this for `tri_cps`. Consider its definition in §5. How are its continuations constructed? In particular, what data is involved and how is it structured? It starts off with `id_`, and "on top" of this, there appear multiple instances of **`lambda`** `x: cont(n + x)`, for varying `n: nat`. Hence, it seems plausible that we can encode a continuation as a **`list`**`[nat]`:

```
82 type tri_cps_data = list[nat]
```

We now define an auxiliary function `tri_cont` to reconstruct the continuation from the data and apply it:

```
83 def tri_cont(data: tri_cps_data, x: int) -> int:
84     if data:  # non-empty
85         n = data.pop()
86         return tri_cont(data, n + x)
87     else:  # empty, act as identity
88         return x
```

So, now the operation (viz. `n+x`) occurs once, viz. in `tri_cont`. The defunctionalized version of `tri_cps` is then given by

```
89 def tri_dcps(n: nat, data: tri_cps_data = None) -> int:
90     if data is None:
91         data = []  # to avoid mutable default argument
92     if n == 0:
```

```
93          return tri_cont(data, 0)
94      else:
95          data.append(n)
96          return tri_dcps(n - 1, data)
```

Of course, this can also be made lazy for the trampoline (see `tri_dcps_lazy` in (Verhoeff, 2025)).

For this particular function, we can go even further and reduce the data to a single natural number, because all continuations basically are compositions of `lambda x: n + x` for various values of n. The simplification hinges on associativity of function composition and this property:

$$(\lambda\, x : n + x) \circ (\lambda\, x : m + x) = (\lambda\, x : (n + m) + x))$$

Note that for $n = 0$, $\lambda\, x : n + x$ is the identity function. This simplification gives us back `tri_acc`, where the whole continuation is compressed into a single integer (`acc`).

It is also instructive to defunctionalize `total_cps` defined in §6. Its continuation grows in three ways:

- `id_`, without capturing any data, is the initial continuation;
- `lambda tt1: cont(tt0 + tt1)`, with `tt0: int` as captured data, extends `cont`;
- `lambda tt0: total_cps(t[1], lambda tt1: cont(tt0 + tt1))`, with `t[1]: Tree` as captured data, also extends `cont`.

Thus, we can encode a continuation as

```
97  type total_cps_data = list[int | Tree]
```

The `int` is the total of a left subtree that needs to be added to the total of a right subtree (which has not been determined yet), and the `Tree` is a right subtree which still must be totaled. Here is how to reconstruct the continuation from the data:

```
98  def total_cont(data: total_cps_data, x: int) -> int:
99      if data:  # non-empty
100         last = data.pop()
101         if isinstance(last, int):  # pending total of left tree
102             return total_cont(data, last + x)  # last is a tt0
103         else:  # pending right tree
104             data.append(x)
105             return total_dcps(last, data)  # last is a t[1]
106     else:  # empty, act as identity
107         return x
```

Notice the call of `total_dcps`. Thus we get the following defunctionalized version of `total_cps`:

```
108 def total_dcps(t: Tree,
109                 data: total_cps_data = None
110                 ) -> int:
111     if data is None:
112         data = []  # to avoid mutable default argument
113     if isinstance(t, Leaf):
114         return total_cont(data, t.value)
115     else:  # t is binary fork
116         data.append(t[1])  # postpone processing of t[1]
117         return total_dcps(t[0], data)
```

We now have two *mutually* tail-recursive definitions. Without knowing how these function definitions were derived, it would not be obvious why they terminate.

By the way, just as with `tri_dcps`, it is possible to merge some continuations for `total_dcps`, and to pass these on as a single integer (`acc`), and the remainder as a `list[Tree]`. We leave it as an exercise to the reader to fill in the missing details (see Appendix A.3 for some hints). The result is a classic tail recursive function, without suspended calls and without the need for an auxiliary function that explictly reconstructs the continuation.

```
118 def total_acc_dcps(t: Tree,
119                     acc: int = 0,
120                     data: list[Tree] = None
121                     ) -> int:
122     if data is None:
123         data = []  # to avoid mutable default argument
124     if isinstance(t, Leaf):
125         acc = acc + t.value
126         if data:  # non-empty
127             t1 = data.pop()
128             return total_acc_dcps(t1, acc, data)
129         else:  # data is empty
130             return acc
131     else:  # t is binary fork
132         data.append(t[1])  # postpone processing of t[1]
133         return total_acc_dcps(t[0], acc, data)
```

In hindsight, it is clear that in all these defunctionalized programs, parameter `data` serves as a custom stack, that stores exactly the information needed to support the (branching) recursion.

8. Conclusion

I hope to have created awareness of what I call the Olympiad trap, where the IOI is locked into its own contest format. One way of mitigating it, is to pay attention to interesting and challenging topics in informatics that fall outside the scope of the IOI. As an

example of such a topic, I explained how tail recursion can be transformed into a loop using thunking and a trampoline. And next, how Continuation Passing Style (CPS) can be used to transform arbitrary recursion into tail recursion. These ideas were discovered a long time ago and have become part of the CS folklore. For a history of continuations see (Reynolds, 1993), which traces it back to 1964, when Adriaan van Wijngaarden (designer of Algol 60 and Algol 68, and my father's promotor) first described it. The trampoline seems to have been introduced by Steele (1977). Gibbons (2022) is a modern exploration of CPS, accumulation, and defunctionalization.

The code for this article is available at (Verhoeff, 2025). It includes code that is instrumented to visualize how the stack is used, together with the output of that code. It also treats the example of flattening a binary leaf tree.

Acknowledgment

I would like to thank Ahto Truu and my colleague Berry Schoenmakers (one of the team members who made it to the ICPC World Finals in 1987) for helping me improve this article.

References

Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gibbons, J. (2022). Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *The Art, Science, and Engineering of Programming*, 6(2):7:1–7:28. https://doi.org/10.22152/programming-journal.org/2022/6/7

Reynolds, J.C. (1993). The Discoveries of Continuations. *LISP Symb. Comput.*, 6(3–4):233–248. https://doi.org/10.1007/BF01019459

Steele, Guy L. (1977). Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO. In: *ACM'77: Proceedings of the 1977 annual conference*. pp. 153–162. https://doi.org/10.1145/800179.810196

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. https://doi.org/10.1007/978-3-319-98355-4_35

Verhoeff, T. (2021). Look Ma, backtracking without recursion, *IOI Journal 2021*, 15, 119–132. https://doi.org/10.15388/ioi.2021.10

Verhoeff, T. (2023). "Understanding and designing recursive functions via syntactic rewriting", *IOI Journal*, 17, 99–119. https://doi.org/10.15388/ioi.2023.08

Verhoeff, T. (2025). Git repository with source code for "The Olympiad Trap and an Old Trampoline". (Accessed 30 May 2025) https://gitlab.tue.nl/t-verhoeff-software/code-for-cps-with-trampoline

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, model driven engineering, and functional programming. 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 IOI-related material and facilities for communication in the IOI community, and in establishing, developing, chairing, and contributing to the IOI International Scientific Committee from 1999 until 2007.

Appendix A. Answers to Exercises

A.1. Function `total` with Accumulation Parameter

One of the recursive calls in the definition of `total` can be made into a tail call by introducing an accumulation parameter:

```
def total_acc(t: Tree, acc: int = 0) -> int:
    if isinstance(t, Leaf):
        return acc + t.value
    else:
        return total_acc(t[1],  # tail call
               total_acc(t[0], acc))  # not a tail call
```

The other recursive call is not a tail call, since more work is done after it returns.

A.2. Recursive Function where CPS Fails

Continuation Passing Style (CPS) is a powerful technique that can be used to transform any recursive function definition into a tail-recursive definition. But this does not mean that the resulting function is useful. In particular, it may not terminate. One example where this happens is the *fixpoint combinator*.

For given function f, we call x a fixpoint of f when $f(x) = x$. The fixpoint combinator fix returns a fixpoint of a given function f from functions to functions. That is, we have $f(fix(f)) = fix(f)$ as function. It can be defined in Python as `fix`:

```
type Endo[A] = Func[A, A]  # endo-functions on A

def fix[A, B](f: Endo[Func[A, B]]) -> Func[A, B]:
    return f(lambda a: fix(f)(a))
```

Function `fix` is recursive but not tail recursive.

Let's see an application. Function `tri` is a fixpoint of function `pre_tri`:

```
def pre_tri(g: Func[nat, int]) -> Func[nat, int]:
    return lambda n: 0 if n == 0 else n + g(n - 1)
```

Note that `pre_tri` abstracts from the recursive call, by making the function called there a function parameter. Hence, it is itself not recursive. Obviously, we have

```
    pre_tri(tri)(n) == tri(n)
```

That is, `tri` is a fixpoint of `pre_tri`. Hence, we can define `tri` by `fix(pre_tri)`.

Can we make `fix` tail recursive by applying CPS? Let's try:

```
def fix_cps[A, B](g: Endo[Func[A, B]],
                  cont: Endo[Func[A, B]] = id_
                  ) -> Func[A, B]:
    return fix_cps(g, lambda h: cont(g(lambda a: h(a))))
```

This definition is indeed tail recursive. But when you apply it to `pre_tri`, it does not terminate, because the argument `g` remains unchanged and never reaches a base case. The continuation `cont` keeps on growing. So, `fix_cps` is useless and certainly not equivalent to `fix`.

A.3. Defunctionalization of **`total_cps`**

The defunctionalization of `total_cps` can be better understood by analyzing the structure of its continuations. These can always be written as a composition:

- id, the initial continuation, is an empty composition;
- $\lambda\, tt_1 : cont\,(tt_0 + tt_1) = cont \circ (\lambda\, tt_1 : tt_0 + tt_1) = cont \circ (tt_0 +)$, where $(n +)$ abbreviates the function $\lambda\, x : n + x$;
- $\begin{aligned}&\lambda\, tt_0 : total\_cps\,(t_1, cont \circ (tt_0 +))\\ &= \lambda\, tt_0 : (cont \circ (tt_0 +))(total\,(t_1))\\ &= \lambda\, tt_0 : cont(tt_0 + total\,(t_1))\\ &= cont\,(\lambda\, tt_0 : tt_0 + total\,(t_1))\\ &= cont \circ (+ total\,(t_1)).\end{aligned}$

Thus, every continuation is some composition of $(tt_0 +)$ and $(+ total(t_1))$ for varying values of tt_0 and t_1. Compositions of these two kinds of functions commute (due to associativity of addition; see below), and therefore all functions of the form $(tt_0 +)$ can be moved together and merged into one such function, as we have seen before, which can then be defunctionalized into a single integer (`acc`). The composition of the other functions is defunctionalized into a list of `Tree`.

Here is a proof that the composition of $(n+)$ and $(+m)$ commutes because addition is associative. For integer k, we calculate:

$$\begin{aligned}&((n+) \circ (+m))(k)\\ &= (n+)((+m)(k))\\ &= (n+)(k + m)\\ &= n + (k + m)\\ &= (n + k) + m\\ &= (+m)(n + k)\\ &= (+m)((n+)(k))\\ &= ((+m) \circ (n+))(k)\end{aligned}$$

Such calculations with side-effect-free functions lies at the foundation of modern functional programming.