Olympiads in Informatics, 2024, Vol. 18, 113–128

DOI: 10.15388/ioi.2024.09

Staying DRY with OO and FP

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

Abstract. We discuss the coding principle Don't Repeat Yourself (DRY) and compare various tactics to achieve DRY code, both in the context of object-oriented (OO) programming and functional programming (FP). Neither OO nor FP play a significant role in the International Olympiad in Informatics (IOI), but these paradigms are highly relevant in industry. This article aims to make clear that they offer useful computational insights that should appeal to talented students and that could somehow make their appearance in the IOI. We conclude with an AHA insight and some WET advice for talent development and programming contests.

Keywords:

1. A DRY Introduction

Like any product, software goes through a life cycle. In the case of programming contests, the life of a program can be pretty short: a couple of hours. And during that life, depending on the format of the contest, one person or at most only a few people have seen the code, while it was conceived. Nowadays, evaluation in contests is done without anyone actually seeing that code. If, however, you earn a living by writing source code, then the life cycle extends quite a bit further. In fact, initial development then only accounts for a small fraction of the life cycle. The rest concerns operation and maintenance. Especially maintenance brings other qualities to the game, other than functional correctness, speed, and memory usage.

When programming under time pressure (as in a contest), it is tempting to copy-paste-edit code, also known as code *cloning*. For one thing, in a contest, code size often hardly matters, and finding a more clever solution can cost valuable time. When earning a living with software, cloning code might be tempting if you are paid by number of lines of code delivered. In general, however, copy-pasting hurts in the longer run. Imagine that you have ten clones of the same piece of code in your software. If you find a defect, or a way to improve performance, then you will want to update all copies (first you need

to find them all). If the copies were edited after pasting, then updating can become a challenge. You don't want that under time pressure either.

Hunt and Thomas (2019) introduced the software engineering principle *Don't Repeat Yourself* (DRY) in 1999 to create awareness of the problems of code cloning. Note that code clones are a purely syntactic phenomenon, that is, they concern a static property of code. This is also why it is not too hard to build clone detection into IDEs. DRY should not be confused with the Single Responsibility Principle (SRP), which concerns a semantic property. Solving the same problem twice is also a different concern, as can be seen from these two exercises:

1. Give two functions, solving the same problem, but with minimal code duplication.
2. Give two functions, solving different problems, but with considerable code duplication.

Here is a solution to the first exercise, in Python (we use version 3.12; all source code is available (Verhoeff, 2024); Java and C++ would look similar):

```
1  def fac_iter(n: int) -> int:
2      """Compute factorial of n, iteratively.
3      """
4      result = 1
5
6      for i in range(1, n + 1):  # see footnote 1
7          result = result * i
8
9      return result
10
11 def fac_rec(n: int) -> int:
12     """Compute factorial of n, recursively.
13     """
14     if n == 0:
15         return 1
16     else:
17         return n * fac_rec(n - 1)
```

Apart from the header and the docstring, the code for `fac_iter` and `fac_rec` couldn't be more different, even though they compute the same thing. For the second exercise, compare `fac_iter` and `sum_array` defined by

```
18 def sum_array(array: list[int], n: int) -> int:
19     """Sum first n items in array, iteratively.
20     """
21     result = 0
22
23     for i in range(0, n):
24         result = result + array[i]
25
26     return result
```

[1] `i in range(a, b)` implies `a <= i < b`; that is, upper bound `b` is excluded.

Now we have two functions that compute two very different things, but there is significant overlap in code: they are near code clones.

2. DRY via Object-Oriented Design Patterns

What can one do to avoid code clones, such as we have seen above? Typically this is done by defining a function that has the copied code as body, and replacing each copy by a call to that function. In case of near clones, one can introduce parameters, so that the body becomes more general, and the calls can be specialized by providing appropriate arguments to the calls.

For instance, the two statements on lines 4 and 21 are near clones, and values 0 and 1 can be generalized to integer parameter `initial`. The statement in the body then would become `result = initial`. The same holds for lines 6 and 23, which could be generalized to `for i range(start, start + n)`.

But for the statements on lines 7 and 24 this won't work so easily, because these involve two different ways of *updating* `result`. In the world of object-oriented (OO) programming, the language mechanisms involving classes, inheritance, overriding, and polymorphism can be used effectively to address this situtation. Because these OO language mechanisms are easy to abuse, various standard ways have evolved to use them safely, known as *Design Patterns* (Gamma *et al.*, 1994).

2.1. *Template Method Design Pattern*

For our example, the *Template Method* pattern can be used, because it captures a common algorithmic structure with some variation points. The common algorithm is programmed in a so-called *template method* in an *abstract base class*. That template method calls so-called *hook methods* at the variation points. These hook methods are *abstract*, that is, they are not implemented in the base class. Each clone is now replaced by a call to the template method in a *sub*class, *inheriting from the base class*, while *overriding* the abstract hook methods with the code that is specific for each clone.

Let's apply this to our example. Here is the abstract base class that captures the common part of the iteration algorithm:

```
27 from abc import ABC, abstractmethod
28 from typing import override  # requires Python 3.12
29
30 class Iteration(ABC):
31     """Abstract Base Class for iteration
32     using the Template Method design pattern.
33     """
34     @abstractmethod
35     def initialize(self) -> int:
36         """Abstract hook method to initialize result."""
```

```
37
38      @abstractmethod
39      def update(self, i: int, result: int) -> int:
40          """Abstract hook method to update result."""
41
42      def iterate(self, start: int, n: int) -> int:
43          """The template method with the common algorithm.
44          """
45          result = self.initialize()
46
47          for i in range(start, start + n):
48              result = self.update(i, result)
49
50          return result
```

Note that we used a hook method to initialize `result`, but avoided a hook method to provide the start value for `i`, and instead passed it to the template method directly as argument. That way, both options are illustrated.

The base class can be specialized as follows for computing factorials.

```
51  class Factorial(Iteration):
52      """Specialize the iteration algorithm for factorials.
53      """
54      @override
55      def initialize(self) -> int:
56          """Override abstract initialization hook method.
57          """
58          return 1
59
60      @override
61      def update(self, i: int, result: int) -> int:
62          """Override abstract update hook method.
63          """
64          result = result * i
65          return result
```

And the subclass `Factorial` is then used as:

```
66  def fac_iter_TM(n: int) -> int:
67      return Factorial().iterate(1, n)
```

For summing an array, the following specialization can be used, where the array is provided via the constructor:

```
68  class ArraySummer(Iteration):
69      """Specialize the iteration algorithm for summing an array.
70      """
71      def __init__(self, array: list[int]) -> None:
72          self.array = array
73
74      @override
75      def initialize(self) -> int:
```

```
76          return 0
77
78      @override
79      def update(self, i: int, result: int) -> int:
80          return result + self.array[i]
81
82  def sum_array_TM(array: list[int], n: int) -> int:
83      return ArraySummer(array).iterate(0, n)
```

There are several reasons to be less happy with this way of avoiding copied code. For one thing, the code is now considerably longer. In fact, you could argue that classes `Factorial` (lines 51–65) and `ArraySummer` (lines 68–80) are near code clones and in practice probably would have been programmed by copy-paste-edit. The code is also harder to understand, because the control flow is rather contorted due to inheritance and overriding. Consider the call `fac_iter_TM(5)`. The following happens during its execution:

1. `Factorial()` instantiates an object of type `Factorial`.
2. Its method `iterate(1, 5)` is called.
3. The definition of this method isn't found in `Factorial`.
4. The search for an implementation proceeds up the inheritance chain (this is done at runtime), arriving at the implementation in the base class `Iteration`. This is known as *dynamic dispatch*.
5. The body of that method executes, behaving like

```
84      result = self.initialize()
85
86      for i in range(1, 1 + 5):  # start == 1
87          result = self.update(i, result)
88
89      return result
```

6. Inside the body, hook methods `self.initialize()` and `self.update()` are called. But these aren't implemented in the base class. However, keep in mind that `iterate()` was called in the scope of `Factorial`. Thus, their implementations are first looked for there, so that the body of `iterate` in fact behaves like:

```
90      result = 1
91
92      for i in range(1, 1 + 5):
93          result *= i
94
95      return result
```

There are even more objections, such as the runtime overhead, in terms of function calls, in particular of the hook methods. Note that `update()` occurs inside the loop. Finally, this solution is rather rigid. Whenever, you want to specialize `Iteration`, you "hard code" the initialization and update functionality into a new subclass. You cannot easily reuse hook methods separately from one specialization in another. There is, what we call, *tight coupling*: concrete hook methods are bound in the concrete subclass *at design time*.

2.2. *Strategy Design Pattern*

To allow specialization at runtime and independent variation, we can use the *Strategy* design pattern. The code at each variation point (hook) is considered a strategy obeying a fixed interface but admitting multiple implementations. This lets one inject concrete hook "strategies" via the constructor. First, we define interfaces (abstract base classes) to specify the signatures of the initialization and update hook strategies:

```
96  class Initializer(ABC):
97      """Interface (Abstract Base Class) for initialization.
98      """
99      @abstractmethod
100     def initialize(self) -> int:
101         """Return initial value."""
102
103 class Updater(ABC):
104     """Interface (Abstract Base Class) for updating.
105     """
106     @abstractmethod
107     def update(self, i: int, result: int) -> int:
108         """Return updated value."""
```

Next, we define the class with the template method (that class is no longer abstract and is also no longer subclassed):

```
109 class IterationS:
110     """Class for iteration with a template method,
111     using the Strategy design pattern for hook strategies.
112     """
113     def __init__(self, initializer: Initializer,
114                  updater: Updater) -> None:
115         """Store the hook strategies.
116         """
117         self.initializer = initializer
118         self.updater = updater
119
120     def iterate(self, start: int, n: int) -> int:
121         """The template method with the common algorithm.
122         """
123         result = self.initializer.initialize()
124
125         for i in range(start, start + n):
126             result = self.updater.update(i, result)
127
128         return result
```

These three classes are used as follows to implement factorial:

```
129 class Initialize1(Initializer):
130     """Initialize with 1.
131     """
132     @override
133     def initialize(self) -> int:
134         return 1
135
136 class FacUpdater(Updater):
137     """Updater for factorial.
138     """
139     @override
140     def update(self, i: int, result: int) -> int:
141         return result * i
142
143 def fac_iter_S(n: int) -> int:
144     return IterationS(Initialize1(), FacUpdater()
145                       ).iterate(1, n)
```

The array summer is implemented in a similar way:

```
146 class Initialize0(Initializer):
147     """Initialize with 0.
148     """
149     @override
150     def initialize(self) -> int:
151         return 0
152
153 class ArraySumUpdater(Updater):
154     """Updater for array_sum.
155     """
156     def __init__(self, array: list[int]) -> None:
157         self.array = array
158
159     @override
160     def update(self, i: int, result: int) -> int:
161         return result + self.array[i]
162
163 def sum_array_S(array: list[int], n: int) -> int:
164     return IterationS(Initialize0(),
165                       ArraySumUpdater(array)
166                       ).iterate(0, n)
```

We achieved increased flexibility (*loose coupling*), because the various initializers and updaters are more general and can now easily be mixed and matched at runtime. But we did pay a price for this: still more code and also more runtime overhead (but a simpler control flow). Concerning more runtime overhead, note the extra indirection in the calls of the hook strategies (lines 123 and 126): `self.initializer.initialize()` and `self.updater.update()`.

This may leave you wondering why object-oriented programming and OO design patterns were invented in the first place. And it also clarifies why these are not used in (most) programming contests.

3. DRY via Functional Programming

Rather than using classes, methods, and objects, we now stick to *pure functions*, that is, functions *without side effects*. For some, that may feel like tying your hands behind your back. But it is interesting to see where it gets you.

The class `IterationS` above is not really needed if we can inject the initialization and update functionality directly into its method (function) `iterate`. For updating, that requires a parameter of a function type, mapping two integers to an integer, typed in Python as `Callable[[int, int], int]`. Here is the definition of `iterate` as a pure function (not inside any class):

```
167 from typing import Callable  # to type functions
168
169 def iterate(initialize: Callable[[], int], start: int,
170             update: Callable[[int, int], int]
171            ) -> Callable[[int], int]:
172     """Return a function that iterates, using
173     given start value and initialize and update functions.
174     """
175     def f(n: int) -> int:
176         result = initialize()
177
178         for i in range(start, start + n):
179             result = update(i, result)
180
181         return result
182
183     return f
```

To stay close to the object-oriented solutions, we have also made `initialize` a (parameterless) function parameter. Furthermore, note that `iterate` returns a function, in this case mapping an integer `n` to an integer. So, you could think of it as a *function factory*: from `initialize`, `start`, and `update`, it constructs a function (of `n`). This is how to use it to get the factorial function:

```
185 from operator import mul  # multiplication
186
187 # type: Callable[[int], int]
188 fac_iter_FP = iterate(lambda: 1, 1, mul)
```

We can call our newly created function simply as `fac_iter_FP(5)`. But what about letting this factory function create a function to sum an array? We don't want to give

`iterate` an extra parameter, because that would be in the way when defining functions like factorial. Think about it. There is no object to inject the array into via a constructor. Therefore, the generated function needs to take the array as extra parameter. See if you can find a way out.

As Nikita Sobolev (2020) wrote: "Functional programmers are smart people. Really. They can do literally everything with just pure functions." And so it is. The trick is to *generalize* the result type **`int`** (of the functions that our factory creates) to an arbitrary type `X`:

```
189 def iterateG[X](initialize: Callable[[], X], start: int,
190                  update: Callable[[int, X], X]
191                  ) -> Callable[[int], X]:
192     """Return a generic function int -> X that iterates
193     using given start value and initialize and update functions.
194     (Generic function definitions require Python 3.12.)
195     """
196     def f(n: int) -> X:
197         result: X = initialize()
198 
199         for i in range(start, start + n):
200             result = update(i, result)
201 
202         return result
203 
204     return f
```

The types of the function parameters `initialize` and `update` had to be generalized accordingly. Function `iterateG` is known as a *generic* factory function, parameterized by type `X`. For factorial, `X` is simply **`int`** and we can still define:

```
205 fac_iter_FPG = iterateG(lambda: 1, 1, mul)
```

For array summing, we use the following *function* type as result type `X`:

```
206 type IntFromArray = Callable[[list[int]], int]
```

Values of this type are functions that map an array of integers to an integer. So, the factory now can create a function that takes integer `n` as parameter and produces a *function that takes an array as parameter*, which in turn computes the array sum. The factory creates what is known as a *curried* function: you need to feed it two parameters *in sequence*, rather than together.

Thus we have sneaked in a way of injecting the array into the iteration that repeatedly updates the variable `result`, which also has that function type. So, the loop now computes with functions of type `IntFromArray`. The initial value of `result` is

```
207 ifa_0: IntFromArray = lambda array: 0
```

because the initial value does not depend on the array (though for other specializations it could). Using lambda expressions, the `result` variable can now be updated by

```
# type: Callable[[int, IntFromArray], IntFromArray]
update_ifa = (lambda i, result_ifa:
                lambda array: result_ifa(array) + array[i])
```

This may be a bit tricky to read at first, but by considering the types, the definition should make sense (hang on for an example). Finally, we define

```
# type: Callable[[int], IntFromArray]
sum_array_FPG = iterateG(lambda: ifa_0, 0, update_ifa)
```

The call `sum_array_FPG(n)(array)` sums the first `n` values of `array`. Let's reason through the three updates in the call `sum_array_FPG(3)`. The initial value of `result` is `ifa_0`:

```
1:  update_ifa(0, ifa_0)
 ==   { definition of ifa_0 }
    update_ifa(0, lambda array: 0)
 ==   { definition of update_ifa }
    (lambda array:
     (lambda array:
                0
     )(array) + array[0]
    )

2:  update_ifa(1, (lambda array:
                    (lambda array:
                                0
                    )(array) + array[0]
                   ))
 ==   { definition of update_ifa }
    (lambda array:
     (lambda array:
      (lambda array:
                 0
      )(array) + array[0]
     )(array) + array[1]
    )

3:  update_ifa(2, (lambda array:
                    (lambda array:
                     (lambda array:
                                 0
                     )(array) + array[0]
                    )(array) + array[1]
                   ))
 ==   { definition of update_ifa }
    (lambda array:
     (lambda array:
      (lambda array:
       (lambda array:
                    0
```

```
250          )(array) + array[0]
251         )(array) + array[1])
252       )(array) + array[2]
253      )
```

This can be considered *meta-programming*, where the factory function actually creates a program to solve the array summing problem for a specific value of `n`. When the Python interpreter evaluates a plain lambda expression like **`lambda`** `array: ...array...`, it won't evaluate the body `...array...`. It just produces a code object that is only put to work when the lambda expression is *called* on a specific argument. A plain lambda expression is a function treated as data. In languages that have better support for functional programming, the expressions above could be evaluated (simplified) further (at runtime). For instance, we have:

```
254      (lambda array:
255       (lambda array:
256                  0
257       )(array) + array[0]
258      )
259   ==   { beta-reduction on inner lambda expression }
260      (lambda array:
261       0 + array[0]
262      )
```

Repeated beta-reductions would simplify the expression on lines 245–253 to

```
263      (lambda array:
264       0 + array[0] + array[1] + array[2]
265      )
```

which is clearly a program that sums the first three items of an array that is given as argument. In fact, the beta-reductions would be done during each update, so that the value of `result` would never be as complicated as above. That value would always have the shape

```
266      (lambda array:
267       0 + array[0] + ... + array[i]
268      )
```

since

```
269      update_ifa(i + 1, (lambda array:
270                         0 + array[0] + ... + array[i]
271                        ))
272   ==   { definition of update_ifa }
273      (lambda array:
274       (lambda array:
275        0 + array[0] + ... + array[i]
276       )(array) + array[i + 1]
```

```
277       )
278   ==    { beta-reduction }
279       (lambda array:
280        0 + array[0] + ... + array[i] + array[i + 1]
281       )
```

In a truly functional programming language, we could have written the factory such that it grows the program from the *inside*, rather than the *outside*. That way, if the array argument is already available before the factory starts, then the evaluation of the updates could already use that array to simplify the accumulating result expression even further.[2]

In conclusion, the DRY functional code is pretty and relatively short. The definition of the generic factory function `iterateG`, isn't much longer than the duplicated code (see lines 196–202), it is more general, and it includes documentation (type hints and a docstring). There is still execution overhead when doing this in a language like Python (or Java or C++). But by writing in a language better suitable for functional programming, that execution overhead can be reduced considerably by a good compiler.

The evaluation of a functional program can look contorted in its own way. So, you might think that it isn't much better than the contorted control flow in the execution of an object-oriented program. But here we need to remind you that the evaluation order of a program involving only pure functions (so, no mutable data either) does not matter. You can leave it to the compiler and runtime system to find an efficient order. In particular, function arguments need not be (fully) evaluated before putting the function body to work. Python won't do that without explicit help and its default eager execution order is not so good for functional programs. Moreover, pure function evaluation is easy to parallelize.

Due to their higher generality, reading and writing of functional programs does place higher demands on the programmer's abstraction skills. But for the talented participants of olympiads that should not be an obstacle. See Appendix A for more Python examples.

4. A WET Conclusion with an AHA Insight

The examples that we gave are clearly out of proportion. Functions `fac_iter` and `sum_array` have only five nearly duplicated lines of code. You see such loops everywhere in code. It would seem to make little sense to eliminate them all. For the object-oriented style that is certainly the case, because the overhead of abstraction is considerable. It only pays off when code clones are larger.

In the functional style, the situation is quite different. There, one can have small building blocks that are very general. Since purely functional programming languages

[2] See `https://t-verhoeff-software.pages.tue.nl/code-for-staying-dry-with-oo-and-fp` for an interactive evaluation explorer.

have no loops, one needs to use recursion (which is not an easy thing (Verhoeff, 2023)). Many forms of recursion have been encapsulated in *recursion schemes* such as *folds* and *unfolds*. These act like our factory function `iterateG`, and these are preferred over writing your own recursive function definitions. You hardly see explicit recursion in good functional programs.

Once a beginning programmer has seen the benefits of abstraction, and how it can help avoid code duplication, there lurks the temptation to apply it at every opportunity to ensure that code is DRY from the start. Creating abstractions is indeed intellectually satisfying, but one needs to be careful.

Programmers with more experience will know that abstractions are hard to get right and that they may not fit a future situation that you had envisioned. Adjusting an abstraction and then also all its instantiations is about as dangerous as updating duplicated code. Therefore, another common advice is to *Avoid Hasty Abstractions* (AHA). If you generalize code too early, you may end up with abstractions that later turn out to be less useful and that need adjusting. In fact, function `iterateG` could be considered a hasty abstraction. In functional programming, a more appropriate abstraction turns out to be `foldN` (see Appendix A).

Don't be afraid to write some duplicated code. That way you can better see in what direction and how far it makes sense to generalize your code. This is captured in yet another common advice: *Write Everything Twice* (WET), or even better *Write Everything Thrice*. By explicitly writing out similar code multiple times, it becomes easier to recognize the relevant patterns and how to abstract from the differences.

That brings me to programming contests. Are these techniques to achieve DRY code of any use there? Code written in a contest is throw-away code. But generalization is a great problem solving technique. So, getting better at that can help. Also, applying abstraction in your code can be helpful to stay in control. It would then be useful if you can use a programming language with good support for abstraction. Unfortunately, on the theoretical side we have much more advanced notions for abstraction than are available in common practical programming languages. The mind can see things that are hard to express in code. I find this a shortcoming of the current format of the IOI.

I hope that team leaders are going to study the examples in this article together with their contestants. Higher-order functions, functions with functions as parameters and returning functions, are very powerful devices. It takes some practice to get used to the functional style, but I am convinced that clever contestants will enjoy it.

Acknowledgment

I would like to thank Berry Schoenmakers (TU Eindhoven, Netherlands) and Mārtinš Opmanis (Latvia) for helping me improve this article.

References

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

Hunt, A., Thomas, D. (20th Anniversary Ed. 20219; 1st ed. 1999). *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley.

Sobolev, N. (2020). "Typed Functional Dependency Injection in Python", blog post, 17 Feb. 2020. `https://dev.to/wemake-services/typed-functional-dependency-injection-in-python-4e7b`

Verhoeff, T. (2023). Understanding and designing recursive functions via syntactic rewriting. *IOI Journal*, 17, 99–119.

Verhoeff, T. (2024). Git repository with source code for "Staying DRY with OO and FP". ‚`https://gitlab.tue.nl/t-verhoeff-software/code-for-staying-dry-with-oo-and-fp` (Accessed 27 April 2024)

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 IOI-related 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
Two more Python examples, using `foldN`

Generic factory function `iteratG` was a hasty abstraction. It does not need both parameters `initialize` and `start`. They can be combined into one parameter, say `initial`. A better generalization is generic factory function `foldN`:

```
282 def foldN[X](initial: X,
283               update: Callable[[X], X]
284              ) -> Callable[[int], X]:
285     """Return a generic function int -> X
286     that applies update repeatedly to initial.
287     """
288     def f(n: int) -> X:
289         result: X = initial
290 
291         for i in range(0, n):
292             result = update(result)
293 
294         return result
295 
296     return f
```

It cleanly captures the essence of the natural numbers. Each natural number has a unique construction using `zero = 0` and `succ = lambda n: n + 1`:

```
297     n == succ(succ(...succ(zero)...))  # n copies of succ
```

Similarly,

```
298     foldN(i, u)(n) == u(u(...u(i)...))  # n copies of u
```

Thus, folding replaces constructor `zero` by `i` and `succ` by `u`.

Let's use `foldN` to compute the famous Fibonacci numbers:

```
299 def fib(n: int) -> int:
300     return foldN((0, 1),
301                  lambda t: (t[1], t[0] + t[1]))(n)[0]
```

Here, the the generic type parameter `X` is instantiated by `tuple[int, int]`.

And what do you think of this application of `foldN`:

```
302 type Converter = Callable[[list[int]],
303                            tuple[int, list[str], int, int]]
304 
305 init_c: Converter = lambda bits: (1, [], 1, 0)
306 
307 def update_c(result_c: Converter) -> Converter:
308     def f(bits: list[int]) -> tuple[int, list[str], int, int]:
309         index, strings, power, value = result_c(bits)
310         bit = bits[-index]  # indexed from the end
```

```
311             strings.append(f"{bit} * 2^{index - 1}")
312             return index + 1, strings, power * 2, value + bit*power
313
314         return f
315
316     def convert(bits: list[int]) -> str:
317         _, strings, _, value = foldN(init_c,
318                                      update_c)(len(bits))(bits)
319         return ' + '.join(strings) + f" = {value}"
```

Generic type parameter `X` is now instantiated by `Converter`, which is a function type. That is why the function produced by `foldN` (see lines 320–321) takes two parameters in succession: an integer and then an array of bits. Here is an example usage of `convert`:

```
320     print(convert([1, 1, 0, 1]))
```

It produces as output:

```
1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 = 13
```