Olympiads in Informatics, 2021, Vol. 15, 119–132

DOI: 10.15388/ioi.2021.10

Look Ma, Backtracking without Recursion

Dedicated to my colleague Ruurd Kuiper, on the occasion of his retirement

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

Abstract. I show how backtracking can be discovered naturally without using a recursive function (nor using a loop with an explicit stack). Rather, my approach involves a form of self application that can be elegantly expressed in an object-oriented program, and that is reminiscent of how recursion is done in lambda calculus. It also illustrates why reasoning about object-oriented programs can be hard.

Keywords: computer science, programming, object-oriented, functional, backtracking, recursion, fixed-point combinator, self application.

1. Introduction

This article can be viewed as an addition to Verhoeff (2018), which covers – what I believe to be – the basics of recursion. I will illustrate my approach to backtracking through the problem of solving simplified Binairo puzzles, explained below. Section 2 considers various reasoning strategies to solve such puzzles, culminating in a non-recursive backtracking strategy. Design details, Java code, and some refinements are discussed in Section 3, and the essence is extracted in Section 4. Section 5 concludes the article. I recommend that younger programmers first read §2 and §3.2 and then explore the source code provided in Verhoeff (2021), before reading the other sections.

2. Reasoning Strategies to Solve Puzzles

The puzzles in this article are simplified *Binairos*[1], consisting of a square grid with an even number of rows and columns, partly filled with zeroes and ones (see Fig. 1). The objective is to fill the grid completely with zeroes and ones such that (**Rule 1**) *nowhere three*

[1] A.k.a. binary or Takuzu puzzles; see `https://en.wikipedia.org/wiki/Takuzu`

0		**0**		**1**			
					1	**1**	
0	**0**			**1**			
0						**1**	**0**
			1				
				1			
			1				**0**

Fig. 1. Initial state of our 8 × 8 example puzzle.

or more equal symbols are horizontally or vertically adjacent, and (**Rule 2**) *in each row and in each column, the number of zeroes equals the number of ones*. The simplification is that we allow identical rows and columns. The given zeroes and ones cannot be changed when solving the puzzle, and you may assume that there is a unique solution. Try to solve the example in Fig. 1 if you have not done this kind of puzzle before.

Puzzles can often be solved just by reasoning, and there is no need for 'blind' backtracking. Given Rules 1 and 2 for our puzzles, there are two obvious strategies to fill in what one could call *forced* bits.

1. If in three horizontally or vertically adjacent cells two cells have the same bit b, then the other cell must have the opposite of b, i.e. $1 - b$.
2. If in a row or column half of its cells have bit b, then the remaining cells must have $1 - b$.

The correctness of these strategies follows from the fact that a solution exists. We call three horizontally or vertically adjacent cells a *triplet*, and a complete row or column a *line*. In pseudocode, these strategies can be expressed as follows.

1. *Triplet* strategy: Find a triplet t with one empty cell and a bit b, where b occurs twice in t, and fill the empty cell with bit $1 - b$.
2. *Line* strategy: Find a line ℓ with at least one empty cell and a bit b, where b occurs in half of ℓ's cells, and fill all empty cells of ℓ with bit $1 - b$.

Note that such strategies take a puzzle as argument, and return a, possibly updated, puzzle. We will leave that puzzle parameter implicit for now (think of it as a global variable; see the next section for details). These strategies are nondeterministic, and when they do not apply, they leave the puzzle unchanged. *Strategies must have the property that* (*i*) *they only change empty cells, and* (*ii*) *they preserve solvability*, i.e., if the puzzle was solvable *before* applying the strategy then it is still solvable *after* applying the strategy.

If you have applied a strategy and nothing changed, then it does not make sense to apply it again. But if it did bring some change, then it might be applicable again. Moreover, if the triplet strategy did not bring any changes, but subsequent application of the line strategy did, then it would be good to try the triplet strategy again (see Fig. 2).

This leads to the wish to define strategy *combinators* such as the following.

3. *Fixed-point* strategy: Repeatedly apply a given strategy until no further change occurs. It takes a strategy as parameter.

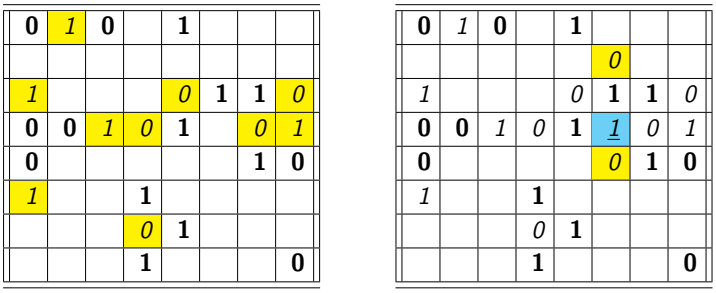

Fig. 2. State after applying the *FT* strategy to Fig. 1 (left), and after the *FPFTL* strategy (right), where the line strategy yielded the blue (underlined) 1.

4. *Pair* strategy: Apply two given strategies one after the other. It has two strategy parameters, and can be nested to combine more strategies.

We call these *meta-stategies*, because they take one or more strategies as parameter. If we (as humans) consider the triplet strategy to be simpler to apply than the line strategy, then we would probably use the following strategy *FPFTL*.

5. $FPFTL = \mathit{fixed\_point}(\mathit{pair}(FT, \mathit{line}))$ where
6. $FT = \mathit{fixed\_point}(\mathit{triplet})$.

Applying the *FT* and *FPFTL* strategies to the puzzle in Fig. 1 leads to Fig. 2.

For the example puzzle, we are then stuck. You may already have discovered the following strategy to help out.

7. *Contradiction* strategy: Find an empty cell c and a bit b, where putting bit b in cell c leads to an invalid state *after applying the FPFTL strategy*, and fill cell c with bit $1 - b$.

The contradiction strategy speculates and looks ahead. It is correct, because by assumption there exists a solution. You can see it in action in Fig. 3. In fact, $\mathit{fixed\_point}(\mathit{pair}(FPFTL, \mathit{contradiction}))$ solves our example puzzle.

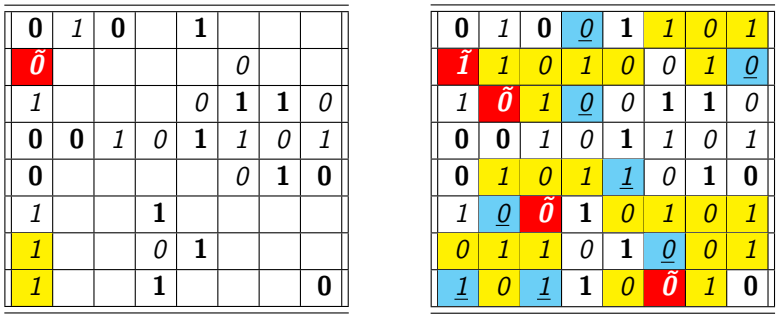

Fig. 3. When trying 0 in the red cell (with tilde), and applying *FPFTL*, the line strategy yields the 1s in the yellow cells, violating Rule 1 (left); therefore the red cell must contain 1; applying *fixed_point*(*pair*(*FPFTL, contradiction*)), we get the shown solution (right), where the colors encode the responsible strategy: yellow–triplet, blue(underline)–line, red(tilde)–contradiction.

Observe that the pair of triplet and line strategies together can be viewed as special case of the following direct contradiction strategy.

8. *Direct contradiction* strategy: Find an empty cell c and bit b, where putting bit b in cell c directly leads to an invalid state, and fill cell c with $1 - b$.

Note that this may be less efficient, but we now do have

9. $FPFTL = fixed\_point(direct\_contradiction)$.

To avoid code duplication (DRY = Don't Repeat Yourself), let's see if we can unify the code for these two contradiction strategies by generalization via a strategy parameter, making it a meta-strategy.

10. *General contradiction* strategy: Find an empty cell c and a bit b, where putting bit b in cell c leads to an invalid state *after applying a given strategy*, and then fill cell c with $1 - b$.

Its usefulness depends on how good the supplied strategy is at finding forced bits. But even when the supplied strategy does nothing, the general contradiction strategy is correct. Let's define the *empty* strategy as doing nothing (applying the identity function).

11. *Empty* strategy: Do nothing.

Then we see that both contradiction and direct contradiction are indeed special cases of general contradiction.

12. $contradiction = general\_contradiction(FPFTL)$.
13. $direct\_contradiction = general\_contradiction(empty)$.

In fact, now we no longer need $FPFTL$, because it can be expressed in terms of the general contradiction strategy via 9 and 13, and the contradiction strategy now becomes a double application of general contradiction:

14. $FPFTL = fixed\_point(general\_contradiction(empty))$.
15. $contradiction = general\_contradiction(fixed\_point(general\_contradiction(empty)))$.

2.1. *Self Application*

Then the idea occurred to me that instead of $FPFTL$ in the contradiction strategy, we should use the best strategy we can think of to find a sequence of forced bits leading to a contradiction. So, what about supplying itself as parameter? Like this:

16. $general\_contradiction(general\_contradiction(general\_contradiction(...)))$.

Of course, we cannot define it with those dots. But we can define a variant H of the general contradiction strategy that expects a hyper-strategy h as parameter, viz. a strategy that takes a hyper-strategy (not necessarily itself) as parameter.

17. $Hyper\text{-}contradiction(hyper\text{-}strategy\ h)$: Find an empty cell c and a bit b, where putting bit b in cell c leads to an invalid state after applying strategy $h(h)$, and fill cell c with $1 - b$. N.B. h could ignore its argument!

Therefore, the hyper-contradiction strategy is in fact also a hyper-strategy, and can be passed as parameter to itself. Now we can properly define the strategy in 16 as $H(H)$. Note that this is a regular strategy (not meta or hyper).

It works (thanks to the two key properties of strategies), but it is not guaranteed to solve puzzles by itself. It may have to be repeated, which we can do by applying the fixed-point strategy. But if that is a better strategy, then we want to use that as parameter in the hyper-contradiction strategy. This can be accomplished by hyper-strategy FH defined by

18. $FH(hyper\_strategy) = fixed\_point(hyper\_contradiction(hyper\_strategy))))$

and applying it to itself, which basically gives us backtracking!

19. $backtracking = FH(FH)$.

If there exists a *unique* solution (which is assumed), anything other than the correct bit in a cell must lead to a contradiction. And it terminates because strategies only fill empty cells. Note that if the puzzle has multiple solutions, then our *backtracking* strategy won't find any of them, because then there exists at least one cell where both 0 and 1 are valid, and there won't be a contradiction.

Observe that these function definitions are not recursive (though there is self application). Of course, there are some details to take care of to make all of this work in a real (strongly typed) programming language. For that, see the next section.

3. Design Details, Java Code, and Refinements

One could try to implement my approach to backtracking in a functional programming language, as sketched above. But this can lead to a typing problem because of the self application. I found it a nice challenge to code it in the object-oriented language Java (which we use in our education, and which used to be permitted at the International Olympiad in Informatics). To keep the code easily understandable for non-Java programmers, I stick to a minimal subset, avoiding interfaces and generic type parameters. Also see (Verhoeff, 2018, §6) for how to type and define functions that can be self-applied in Java.

First, I look at some design details in §3.1, still using a functional approach. Note that these are mathematical functions, and not programming-language functions, that is, without side effects operating on (immutable) values. Next, I present an overview of the Java source code, and finally, I discuss some refinements (that could serve as exercises).

3.1. *Design Details*

Here, I will elaborate on some of my design decisions to obtain an object-oriented (OO) program from the functional description in Section 2. The focus is on implementing strategies. Recall that a strategy is a function from the domain of puzzles, denoted by $\mathcal{P}$, to itself, i.e., they have type $\mathcal{S} = \mathcal{P} \to \mathcal{P}$. In the functional setting there is no mutable data, only values. So, in OO terms, a strategy returns a fresh object, copied from its

argument and possibly altered. In the OO world, this is frowned upon, because the updates to the puzzle are quite local. So, a mutable type of puzzles is preferred, to avoid copying lots of data that is unchanged. (In functional programming this is frowned upon, and one would use lazy updaters applied to the puzzle, but that is outside the scope of this article. See the Python code in Verhoeff (2021) for the idea.)

Before dealing with the puzzle parameter of a strategy, let's also look at meta-strategies. These have type $\mathcal{M} = \mathcal{S} \to \mathcal{S}$. In a functional setting, this is a *curried* function of two arguments, viz. first a strategy and then a puzzle yielding a puzzle: $\mathcal{M} = \mathcal{S} \to \mathcal{P} \to \mathcal{P}$, where the latter is to be read parenthesized as $\mathcal{S} \to (\mathcal{P} \to \mathcal{P})$. Thus, if $m \in \mathcal{M}$, $s \in \mathcal{S}$, and $p \in \mathcal{P}$, we have $m(s) \in \mathcal{S}$ and $m(s)(p) \in \mathcal{P}$. *Uncurried*, the latter would be written as $m(s, p)$. The call $m(s)$ is called a *partial application* of m, because it lacks the second argument, which is needed to start the evaluation.

Let's introduce shorter names for the various strategies introduced in §2.

- $T \in \mathcal{S}$ – triplet strategy (see §2).
- $L \in \mathcal{S}$ – line strategy (see §2).
- $F \in \mathcal{M}$ – fixed-point meta-strategy, satisfying

$F(s)(p) = p$ **if** $s(p) = p$ **else** $F(s)(s(p))$.

This is recursive, but in a Java program, this would be done via a `do-while` loop, not needing a stack. It terminates because every update that strategy s makes to p changes empty cells only.

- $P \in (\mathcal{S} \times \mathcal{S}) \to \mathcal{S}$ – pair meta-strategy, taking a pair of strategies as argument, such that $P(s_1, s_2)(p) = s_2(s_1(p))$; this corresponds to function composition: $P(s_1, s_2) = s_2 \circ s_1$.
- $E \in \mathcal{S}$ – empty strategy, with $E(p) = p$ for all $p \in \mathcal{P}$.
- $G \in \mathcal{M}$ – general contradiction meta-strategy (see §2).
- $H \in \mathcal{H} \to \mathcal{S}$ – hyper-contradiction hyper-strategy, where we have $\mathcal{H} = \mathcal{H} \to \mathcal{S}$ (this is an infinite type that most functional languages don't like, but that we can get to work in Java).

Unfortunately, in OO programming, currying and partial application don't come for free. Suppose we have a function $f \in A \times B \to C$ with curried version $f' \in A \to B \to C$. In that case, $f'(a) \in B \to C$ is a partial application of f, and $f'(a)(b) = f(a, b)$. How can this be done in Java, where there are no separate functions? Instead, we have a method m – static or not – in some class C:

```
1 class C {
2       int m(int x, int y) {
3             return x * x + y;
4       }
5 }
```

To use this function, create an instance of its class, and call the method:

```
6       C obj = new C();
7       System.out.println(obj.m(1, 3)); // 4
8       System.out.println(obj.m(2, 3)); // 7
```

To define a function partially applied only to its first argument, define a new class as carrier for such partially applied functions:

```
9  class PartialCm {
10      private C obj; // 'receiver'
11      private int x; // first argument
12
13      PartialCm(C obj, int x) {
14              this.obj = obj;
15              this.x = x;
                // don't call obj.m, but store x ('lazy')
16      }
17
18      int apply(int y) {
19              return obj.m(x, y); // call with both arguments
20      }
21 }
```

Finally, we create objects from this class to get partially applied versions of m:

```
22      // create partial applications
23      PartialCm m_1 = new PartialCm(obj, 1);
24      PartialCm m_2 = new PartialCm(obj, 2);
25      // apply them
26      System.out.println(m_1.apply(3)); // 4
27      System.out.println(m_2.apply(3)); // 7
```

You could consider this an OO *design pattern* for partial function application. Unfortunately, it is quite bureaucratic and verbose for such a simple idea. You can see a resemblance to the Command design pattern here. That pattern is typically used to capture *all* parameters, and just delay the call. The only reason to delay a call in an OO language is because there is a side effect that must be properly synchronized with other actions. In a functional language, there are no side effects in function calls, and because of lazy ('on-demand') evaluation, you just call the function right away, relying on the compiler and runtime system to decide whether its execution is really needed.

When defining a function in an OO language, client code can pass the arguments to a method in several ways:

- Via parameters of the method (at time of the call).
- Via instance variables in the method's object (in advance of the call), with these variants:

- via one or more, possibly parameterized, *constructors* (only once);
- via one or more, possibly parameterized, *setter methods*;
- via *direct access* to the (non-private) instance variables.

We need to decide how to handle the parameters of (meta-)strategies.

We opt for a puzzle class with mutable objects, and all the strategies will work on the same object. Because of mutability we now also need to worry about reverting changes made to the puzzle state in contradiction strategies. We choose to pass the puzzle parameter via a **`static`** instance variable, set once by the client code. That way we also avoid the need for defining constructors in subclasses that are not meta-strategies.

Concerning strategy parameter(s) of meta-strategies, we did the following. Since we cannot pass pure functions, we pass an object which the intended function as instance method. Also see Verhoeff (2012) on passing functions as arguments in Java.

- For the fixed-point and pair strategies, we pass them via their constructor and store them in instance variables. Polymorphism allows these strategies to abstract from the precise nature of their strategy parameters.
- For the contradiction strategies, we use instance variables set directly by the client code. In Section 4, you can see how this could have been avoided.

3.2. *Java Code*

The Java source code demonstrating that the approach to backtracking sketched in Section 2 really works is available in Verhoeff (2021). I have attempted to keep the code as simple as possible. To do so, I have sacrificed some good OO programming habits. For instance, I have omitted access modifiers (**`public`** and **`private`**) wherever possible. This makes instance variables non-encapsulated, and hence we don't need setters and getters.

First, an overview of the classes, focusing only on the essentials.

- `Puzzle` – a mutable puzzle with a square grid of `Cell`s; all cells are also available for easy traversal in a single `Group` (for the contradiction strategies), and via lists of all `Triplet`s and all `Line`s (for the corresponding strategies); boolean methods `isValid` (to check for rule violations) and `isSolved`.
- `Cell` – a mutable cell with its state, and some global constants.
- `Group` – a list of `Cell`s; generalizes `Triplet` and `Line`, i.e., it has common code for
  - constructing a group from a rectangular block in a puzzle's grid
  - frequency counting of cell states (for validity check and for strategies)
  - bulk filling of empty cells (for triplet and line strategies)

  Subclasses:
  - `Triplet` and `Line` – implement `isValid` (for `Puzzle.isValid` and the contradiction strategies)
- `Strategy` – abstract base class for strategies; carrier of the method `apply` that applies the strategy to the **`static`** `puzzle` injected by the client; `apply` must be

an instance method to allow meta-strategies, such as the pair strategy, to operate on arbitrary strategies; see below for parameters and returned value of `apply`.

Subclasses:

- `TripletStrategy` and `LineStrategy` – to fill in a forced bit based on the rules for validity of `Triplet` and `Line`
- `FixedPointStrategy` – to repeat given strategy until no change; the strategy to repeat is injected via the constructor
- `PairStrategy` – to apply two strategies after each other; these strategies are injected via the constructor
- Not present: `ContradictionStrategy` – this is done in `Tests` via `GeneralContradictionStrategy`
- `EmptyStrategy` – to do nothing
- `DirectContradictionStrategy` – special case of the general contradiction strategy using the empty strategy as helper; no longer needed
- `GeneralContradictionStrategy` – to look for a contradiction after applying a given strategy; the strategy to help find a contradiction is set directly by the client code; also used to define the hyper-contradiction hyper-strategy
- Not present: `HyperContradictionStrategy` – is already offered by `GeneralContradictionStrategy`, because the strategy parameter `strat` passed as instance variable, can be a hyper-strategy (which also has type `Strategy`); self application cannot work via the constructor; client code sets it directly; this needs to be done only once, since all applications of the hyper-contradiction strategy will have the same actual strategy parameter (viz. itself)

- `Command` – abstract base class for commands that modify the puzzle's state; supports reverting an executed command; uses the Command design pattern.

  Subclasses:

  - `SetStateCommand` – command to set and revert state of given cell
  - `CompoundCommand` – a list of commands executed in sequence; uses the Composite design pattern; in traditional recursive backtracking, the old cell state is stored in a local variable of the recursive invocation instead of a command

- `Logging` – a utility class with **`static`** methods to do logging.
- `Tests` – class with `main` method to run some tests.
- `LookMa` – class with `main` method to run and time self-applied strategies.

There is one more thing worth discussing here, and that is how the method `Strategy.apply()` evolved as we added strategies.

1. **`void`** `apply()` – suffices for triplet, line, pair, and empty strategies.
2. **`boolean`** `apply()` returning whether puzzle was changed – needed for fixed-point strategy (to avoid copying the old state and comparing it); other strategies ignore the result.

3. `int apply()` returning number of changes – useful for logging; fixed-point strategy compares result to 0; other strategies ignore the result.
4. `Command apply()` returning all applied puzzle changes, that can be reverted; needed for (general) contradiction strategy; the fixed-point strategy uses the command's size; other strategies ignore the result .
5. `Command apply(int level)` – useful to restrict self-nesting depth and to do indented logging; this is in the current code base.
6. instance variable `untilFirstChange` – (default value **`true`**) added later to control whether strategies stop at the first change (in an earlier version, strategies would complete one sweep, possibly accumulating multiple changes; that behavior can be obtained by setting `untilFirstChange` to **`false`**); the current behavior is useful when you want to give a single hint; repeated application must now be obtained through the fixed-point strategy.

3.3. *Refinements*

There are many ways in which this approach can be improved. I present some suggestions as exercises.

- The pair strategy can be generalized to the *compound* strategy, which operates on a list of strategies. Implement it using the Composite design pattern, where one can dynamically add strategies at run-time to an initially empty strategy. The empty compound strategy can now be used instead of the empty strategy.
- The triplet/line strategies, iterate over the triplets/lines and then over that group to fill empty cells as applicable. Instead, iterate over the grid's empty cells, and for each cell iterate over its triplets/lines, to fill the empty cells as applicable. For this, each cell needs to know in what triplets and lines it occurs. (This also turns out to be useful for other refinements.)
- Typically, one uses only the line strategy (instead of *FPFTL*) in the contradiction strategy. In fact, contradiction often only applies the line strategy to the two lines that intersect at the empty cell being considered. For that, it would help if each cell would know in what lines it occurs.
- The presented backtracking strategy performs badly. If it tries bit 0 and this is correct, then it will find a solution, which it ignores because there is no contradiction. Then it will try bit 1, which of course leads to a contradiction (because there is only one solution). And only then will it conclude that there must be a 0 in the tried cell. When the example puzzle in Fig. 1 is solved by the self-applied fixed-point hyper-contradiction hyper-strategy, it finds 33 554 431 solutions along the way.
- We can remedy this by reporting solutions early, e.g., by throwing an exception in `SetStateCommand`. Then we have a more traditional backtracking algorithm. It would also help to report contradictions early, again by throwing an(other) exception.
- Improve performance further, by maintaining an instance variable in `Group` with the frequency counts of the group's cell states, to avoid their repeated re-compu-

tation. For that, each cell needs to know in which groups it occurs, so that when it changes state, all (and only) the relevant counts can be updated. Instead of the counts, you could maintain a list of cells per state, to make it easy to traverse cells of a particular state in that group.

- To simplify experimentation with strategies, define a Domain-Specific Language (DSL) to construct strategies. For example, in the functional notation of §3.1, the strategy expression $F\,(P\,(F\,(P\,(F\,(T\,),L))\;C))$ is not so complicated, but in the Java code, it becomes a multi-statement code fragment. I would prefer to write an even shorter expression like $((T^*;L)^*;C)^*$ and have this interpreted or expanded automatically.

4. Object-Oriented Programming is Hard

Object-oriented programming (OOP) offers various powerful language features, but these need to be used with care. In this section, I will boil down my approach to the bare essence, thereby pinpointing one of the pains of OOP.

Edsger Dijkstra (1968) fulminated against the *goto* statement, because it made reasoning about (the correctness of) programs unnecessarily hard (look up: 'spaghetti code'). *Structured programming* did away with the goto statement, by restricting the flow of control to language constructs with unique entry and exit points (e.g., **`if-else`**, **`for`**, **`while`**). A next step in the evolution of programming languages incorporated *procedural abstraction*, where one can introduce named parameterized abbreviations for groups of statements, a.k.a. *functions*. And then came *data abstraction*, where one can introduce named parameterized (generic) type abbreviations for groups of variables and related operations, a.k.a. *classes*. Programming with such classes is often referred to as object-oriented programming. Dijkstra (1989) considered OOP "an exceptionally bad idea which could only have originated in California". Here is an example to show why. Consider the following class.

```
28 class Selfish {
29     void f(Selfish x) {
30             System.out.println("Working ...");
31             x.f(x);
               // Not iterative/recursive, but self-applicative
32             System.out.println("... and done!");
33     }
34 }
```

Let me talk you through the code, and in the meantime you can try to see the connection to backtracking via self application. The class `Selfish` has no instance variables and a single instance method `f`, that takes an object of type `Selfish` as parameter. Note that the Java compiler compiles this, even though it is a cyclic type definition. Moreover, note that the *compile-time* type of parameter `x` is `Selfish`. Thanks to the

support of *polymorphism*, however, the actual *run-time* type of the argument supplied to `f` can be any *subclass* of `Selfish`. Therefore, it is unclear (at compile-time) which `f`-functionality is invoked on line 31. This is worse than a goto statement, because with a goto statement, its control destination is known at compile-time. But with polymorphism, the control destination depends on the run-time circumstances (and could even depend on external input).

Here is a subclass of `Selfish`:

```
35 class Innocent extends Selfish {
36     @Override
37     void f(Selfish x) {
38         System.out.println("I'm innocent!");
39     }
40 }
```

Class `Innocent` overrides the behavior of `f`. Now consider the following objects and calls of `f` (see `DemoSelfish.java` in Verhoeff (2021)). Can you predict the execution result?

```
41     Selfish omega = new Selfish();
42     Selfish innocent = new Innocent();
43
44     innocent.f(innocent);
45     innocent.f(omega);
46     omega.f(innocent);
47     omega.f(omega);
```

No loop, no (static) recursion, but still a disaster happens. And that in such a small piece of code. Thanks to dynamic (re)configuration. Spaghetti code is bad, but goto statements are at least static. This example demonstrates something far worse: dynamic spaghetti. It makes for a great job interview question. Here, I rest my case.

5. Conclusion

The title of this article refers to the meme "Look ma, no hands"[2], said by kids proudly showing off to their mom that they can ride a bike without hands on the handlebars (it is also deployed in various jokes). That also captured my feeling when I discovered the approach to backtracking presented here.

I demonstrated that the power of self application can be discovered quite naturally in the context of what is traditionally called backtracking, e.g. when solving combinatorial puzzles. It gives rise to programs that do not employ *static* recursion,

[2] `https://wordhistories.net/2020/04/14/look-no-hands/`

where the body of function *f* contains function calls that can be traced statically (i.e., at compile time) to a call of *f* itself, but rather that employ a form of *dynamic* recursion. In the latter form, the traditional recursive calls are generalized (abstracted) to an additional function parameter, say *g*, of *f* (in OO, *g* will be a method of a parameter object). So, when one reasons about the program text of *f*, one does not know what the actual parameter *g* will be. Client code can later decide what function to provide for *g* in the call of *f*. You obtain recursion when calling *f* with itself as actual parameter for *g*. It is now also clear that one needs to reason about such programs in terms of *contracts*, that specify the pre-and post-conditions (assumptions and effects) of the functions *f* and *g*. Contractual reasoning is also the only way to get to grips with dynamic spaghetti.

To avoid misunderstandings, I mention two caveats.

- I am not claiming that the approach to backtracking in this article is to be preferred. It is purely meant as an interesting and possibly insightful approach. In fact, the version that I presented is inefficient, though it can be made efficient.
- The phrase 'without recursion' in the title is open to debate. But it is certainly not recursion in the traditional sense: a function having static calls to itself, or a container class having instance variables of its own type (think of a *BinaryTree* class containing instance variables *left* and *right* of type *BinaryTree*, a so-called recursive type). There is also no loop with an explicit stack. But in my Java code, you could argue, the class `GeneralContradictionStrategy` uses an instance variable referring to another strategy and this is like a recursively defined singly-linked list type. Dynamically it is configured as an 'infinite' list with itself as tail. However, we have shown in Section 4, that you could achieve the same without instance variables and just with parameters.

See (Verhoeff, 2018, §6) for more examples of 'recursion' without recursively defined functions, using self application instead and without using instance variables. See Verhoeff (2010) for a programming challenge (with interactive hints, using *Tom's JavaScript Machine*) that involves self referencing.

Acknowledgment

I would like to thank my colleagues Kees Huizing and Loek Cleophas for helping me improve this article.

References

Dijkstra, E.W. (March 1968). Go To Statement Considered Harmful. *Comm. ACM*, 11(3), 147–148.

Dijkstra, E.W. (1989). Quoted by Bob Crawford. *TUG lines*, *Journal of the Turbo User Group*, 32, Aug.–Sep.

Verhoeff, T. (2010). An enticing environment for programming. *Olympiads in Informatics*, 4, 134–141.

Verhoeff, T. (2012–2016). *From Callbacks to Design Patterns* (Version 1.7). Software Engineering & Technology, Eindhoven University of Technology, Netherlands. DOI: https://doi.org/10.13140/RG.2.2.28836.40320

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

Verhoeff, T. (2021). Git repository with Java source code for "Look Ma, Backtracking without Recursion". https://gitlab.tue.nl/t-verhoeff-software/code-for-backtracking-without-recursion (Accessed 26 May 2021).

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.