Programmatic Planned Obsolescence

GEORGE ZAKHOUR

Abstract — Should lifetimes have a role beyond memory-safety? Should they be erased or should they be included in the compilation unit? Should lifetimes have dynamic semantics? In this paper we explore a model of programming where these questions are answered in the positive. First, we revisit the history of object-oriented programming and its broader cultural sphere to identify a dynamic semantic for lifetimes. Second, we argue that planned obsolescence is not only one such model, but the only model. Third, we develop this semantics in the framework of category theory and implement it for the Java Virtual Machine. And finally, We demonstrate empirically through four case studies that programming under planned obsolescence is possible and identify a surprising result: the paradigm which is the most industry friendly performs the worst while the one which is the least friendly performs the best.

CCS Concepts: • **Software and its engineering** → **Correctness**; **Abstraction, modeling and modularity**; **Software usability**; *Source code generation*; **Object oriented frameworks**; • **Social and professional topics** → Cultural characteristics.

Additional Key Words and Phrases: CPO, Model Theory, Object-Oriented Programming, Java, Virtual Machines

1 INTRODUCTION

Lifetimes last lifetimes: they are timeless and ubiquitous. While programming, programmers commonly reserve a region of memory to record objects within so that later these objects can be recalled on demand. However, as memory was sparse and valuable in the dawn of programming, unused objects needed to be freed and their memory reclaimed, thus one may speak of the *object's lifetime* as the duration an object occupies in some memory region. To determine the lifetime of objects, programmers employ *lifetime analysis* [58].

As programmers are fallible, their analyses are not always correct. Consequently, throughout the history of software production, a slew of memory-related bugs began surfacing and gnawing at critical infrastructure. As of writing, these bugs are deemed by the community and the world at large to be the most critical ones [11, 46, 48, 67]. For that reason, the United States White House in February 2024 following the United States National Security Agency [15] recommended the use of Rust [56]—a so-called memory-safe programming language [52].

Rust, drawing inspiration from Cyclone [66], offers syntactic capabilities to express lifetimes and the necessary automated checks. This offloads the task of tracking (and analysing) lifetimes from humans unto compilers. As compilation has been made entirely obsolete by machines, the confidence in the correctness of lifetime analyses is at its all time high [4, 75].

Litetimes are akin to types: both capture an aspect of computation and both are described by a plethora of calculi [34]. Yet, lifetimes are not given the same attention that types get. Throughout decades a myriad of papers have been written about the nature of types, yet, for example, it is still not clear whether types should be checked statically or dynamically (or even gradually) [21, 31], if they should be erased or included in the compilation unit [1, 33], or if they should have dynamic semantics or not [37, 49]. On the other hand, no such inquiries have been done into the nature of lifetimes. The thesis of this paper is to fill this semantic gap by giving dynamic semantics to lifetimes.

Semantic gaps in programming languages are addressed through the following two perspectives. The first, the *pragmatic* approach, motives the proposed semantics based on user studies, surveys, or limitations in expressivity. This approach requires a priori knowledge of potential target semantics which many intuit to be correct. The second, the *Curry-Howard-Lambek correspondence*, motivates a proposed semantics by relating it to a pre-existing similar concept in either the theories of logic, computation, or categories. Sadly, for lifetimes none of these instruments are applicable. Instead, we

propose applying *discourse analysis*, an established framework of analyses whose objects are statements and expressions [6, 71].

The research into lifetimes, for example, constitute a *discourse*: a framework of languages in which lifetime is given *semantics* [23]. In fact, we have already applied one discourse analysis to lifetime semantics: that we have identified the gap is thanks to the *negative space* analysis [24, 62].

The other analysis we wish to use is the *rule of formation* analysis [25] that allows us to identify a minimal set of tools that are sufficient to define and implement semantics. The first tool that is common to all studies of a programming language concept are formal models that define the concept based on its behavior, i.e. its *interface*. This model with its critical interface creates a *proto-technological power hierarchy* as it is lacking the *enforcer* of the dictated static and dynamic semantic rules. Thus to complete the hierarchy the second part must be such an enforcer. The existing literature is abound with proposals such as *compilers*, *type-checkers*, *static analysers*, *runtime systems*, or even *continuous integration systems*, to complete the hierarchy.

The final analysis we use is *historical contextualization* which we apply to objects since lifetimes are attached to them. Alan Kay and Adele Goldberg, designers of Smalltalk 72 [43], brought objects and *object-oriented programming* into the forefront of programming culture and fashion. But they did not invent either. Famously, objects were birthed in Oslo by Dahl and Nygaard in the early 1960s while designing Simula I and Simula 67 [51]. Dahl and Nygaard introduced the terminology *object* hastily as a neutral alternative to *process* which referred to concept of "self-initializing data/procedure object" [51]. Thus, objects come in two parts: data and procedures. Yet not every such pairing is an object. According to Ralph Johnson's *Scandinavian View*—which originated from Dahl and Nygaard—objects are the pairings that are meant to be models of physical objects [41], "simulating the behaviour of either a real or an imaginary part of the world" [45].

A New Semantics: Product Obsolescence. To be coherent, the natural dynamic semantics of lifetimes must be coherent with physical lifetimes. But coherence is not the only reason. James Noble, in his 2008 ECOOP banquet talk [50], presented the hypothesis that objects, classes, and object-oriented programming are a sequitur of *technological determinism* [60]. To paraphrase: objects are constantly recreated under the guise of Scheme closures [64], Prolog infinite loops [29, 50, 63], Erlang processes [36], and Haskell type-classes, monads, and lenses [55]. One may conclude that objects and the simulation of the physical world through programming seem to be a fundamental part of the act; that their creation is *deterministically* unavoidable [2].

Here two kinds of physical lifetimes present themselves: natural degradation and deliberate obsolescence [9]. However that is a false dichotomy as natural degradation is the trivial obsolescence: the planned obsolescence with the minimal plan.

To that end, the natural dynamic semantics we assign to object lifetimes in software is planned obsolescence. That is, informally, objects degrade in quality the more they are used.

Yet we must be weary of blind simulation: we must not apply all aspects of planned obsolescence to objects. For example, a light-bulb after one thousand hours will seize to light as its filament will physically snap in two [44]. A `LightBulb` object, on the other hand, ought not to violate its abstract—pre-planned obsolescence—invariants after calling a `turnOn()` method. It is imperative to stress that abstract planned obsolescence must not *break* objects but rather *degrade* their quality. Thus the measure of quality must be a function of only the non-functional requirements of the object. This narrows the possible metrics to two: space and time; the amount of memory and time a method call takes. Reaching the limits of a machine's storage renders it unresponsive and thus violates our degradation invariant. Thus, we argue that *time* is the only quality metric that can be degraded while remaining productive.

In summary, the dynamic semantics of lifetimes is the runtime degradation à-la planned obsolescence.

Contributions. In summary, there is a gap in the semantics of object lifetimes, precisely in their dynamic semantics. In this paper we address this gap by proposing a dynamic semantics of lifetimes. This modest proposal follows a natural extension of the Scandanavian View of objects-as-models and states that lifetimes ought to model the lifetimes of products—which objects are mathematically—through planned obsolescence. Our contributions are thus:

- **BCCC**^{po}, the first formal model of languages endowed with planned obsolescence (Section 4).
- jGeorge, an implementation of **BCCC**^{po} targeting Java Bytecode class files for the Java Virtual Machine (Section 5) named after J. George Frederick (Section 2).
- An evaluation of jGeorge answering four research questions through four case studies: two real-world Java programs, a real-world Scala program, and jEd, a text editor implemented in multiple paradigms. We demonstrate empirically the surprising result that the programming style that is most industry-friendly performs the worst while the style that is least industry-friendly performs the best (Section 6).

2 BACKGROUND

A Brief History of Single Use and the Birth of Obsolescence. The history of planned obsolescence starts in the late nineteenth century during the industrial revolution in the United States of America. *The* economic problem that every industry was attempting to solve was overproduction: people wanted industrial jobs, thus many were producing, thus many was produced, but the demand was low, hence produce was often unsold [47].

The earliest industries who addressed overproduction successfully were those who convinced their customers to buy their products multiple times. The quintessential example is *paper clothing*. These fabrics could be simply thrown out once "this apology for personal cleanliness" [3] became soiled and new, clean ones would be purchased. This business capitalized on laundry services being both expensive and exclusive to men with access to "spousal services" which proved successful as they were selling one-hundred and fifty million paper collars and cuffs annually leading to paper clothing becoming ubiquitous. Naturally, these manufacturers expanded their production into paper hats and paper coats [8]. And so, the first successful answer to overproduction became *single use*.

King Camp Gillette, inventor of the single use shaving blade, marketed his namesake product not only as a convenience but as a more hygienic alternative to the shaving apparatus of the time [19]. This new line of argumentation proved to so effective that Gillette became a house-hold name and a multi-millionaire. Inspired by Gillette, a whole line of hygiene-first single use products derived from his blade: *Kleenex* tissues, *Band-Aid* bandages, *Kotex* sanitary pads, and vulcanized rubber condoms [61].

The Kinds of Obsolescence. In other industries such as the automotive one, single use could not be rationally considered as a product design strategy as the repurchase of a product could not be expected to be done so frequently without filtering all but the wealthy consumers of which few existed. Thus, a more relaxed version of single use began circulating: obsolescence. With time, industries realized that obsolescence strategies fell under one of two umbrellas: technological and psychological [7].

Technological obsolescence is the engineer's go-to obsolescence model which nowadays is referred to as *updates*. The premise of this is the following observation: a product can be made obsolete, on purpose, by simply developing a better product. And so a positive feedback loop is set in motion: the marketplace forces will favor the new and improved products solely based on their technological merit, and those very same forces will favor and encourage the developers that consider improving a product.

Psychological, or stylistic, obsolescence is the salesman's go-to obsolescence model. In this model, a product becomes obsolete because it merely seizes to be in fashion. The earliest practice of psychological obsolescence can be traced back to the rivalry between Ford and General Motors, more precisely, to 1923 when Arthur P. Sloane joined General Motors. When he attempted to make Ford's Model T

obsolete through technological obsolescence—precisely, by improving on the cooling mechanism of the engine—and failed due a fault in the mechanism, he switched strategies promptly and instead sold a visually redesigned Chevrolet that mimicked the style of luxury cars. That strategy proved successful and America's most-selling automotive company became General Motors [61].

J. George Frederick and Progressive Obsolescence. In the late twenties obsolescence was not systematic, and while it was intentional its practice was mostly reactionary. The first person to argue for and make a framework out of deliberate and systematic obsolescence was J. George Frederick [61]. Frederick was an authority in sales, business, and advertisement with a fascination for writing and cooking [68]. He wrote books ranging from cookery such as *Cooking as Men Like it* to self-help such as *What Is Your Emotional Age? And 65 Other Mental Tests* to advertisement such as *Selling by Telephone* to sales such as the classic *Modern Sales Management*. However, his most influential essay is his Advertising and Selling's piece titled *Is Progressive Obsolescence a Path Toward a Sustainable Economy?* [20, 27] in which he coined and defined *progressive obsolescence* as follows:

> I refer to a principle which, for want of a simpler term, I name *progressive obsolescence*. [... Namely] buying goods *not to wear out, but to trade in or discard after a short time, when new and more attractive goods or models come out.*

His principal argument for businesses to engage in progressive obsolescence was surprisingly addressed to the common American consumer. He argued that a patriotic American had the civil duty of enabling technological progress by engaging in *Consumerism* as we know it now, since:

> Every time the American consumer decides on liking something new, it means that factory wheels spin, smoke-stacks belch smoke, and high wages and full employment occur. Every time an American consumer contents himself with antique furniture, [...] and old goods [...], he is tightening the brake band around the American wheel of progress and is retarding our standard of living.

In Frederick's mind, if the American business did not prevent the consumer from stagnating in tradition and did not offer them the opportunity to revel in progress then this patriotic duty of the American consumer could not be prevented. Thus, his call to the American businesses was to be patriotic and realize that:

> how is this acceleration of the idea of progressive obsolescence to be accomplished, there need be offered no "brilliant" new panacea. Advertising is the proved and tried tool.

In other words, that it is their duty to advertise the latest product so as to create *want* in consumers for the new and *aversion* for the old. It is in his honour that we call our system jGeorge.

3 PROGRAMMING WITH PLANNED OBSOLESCENCE

In this section we explore the effects of planned obsolescence on programming. Precisely, we reason informally about the performance of the Fibonacci program with the dynamic semantics of lifetimes motivated in Section 1 that jGeorge uses.

Timely and Methodic Obsolescence. jGeorge targets the Java Virtual Machine (JVM) by modifying some given class file. Briefly, jGeorge performs the following changes to a class file: 1) it creates a new integer field `_uses` that increments on every method call, 2) it creates a new method `_slowDown()` that increments `_uses` and starts a busy loop that terminates after `_uses` nanoseconds, and 3) it modifies every method—with the exception of the `_slowDown` method—so that it always starts by calling the `_slowDown` method.

We refer to this implementation as the *timely and methodic obsolescence* model as it only degrades the running time of a JVM program without affecting its correctness through modifying its methods.

One important consequence of this model is that an object is marked as used not only when it's internal state is modified, but also when it's read through *getters* or *view* methods. In reality, many physical objects have this property. In the extreme case quantum systems are affected by reads: their state is said to collapse after observations. Less extreme cases are organs and bones whenever they are scanned by high-frequency electromagnetic radiation. But more common systems, for example boxes, cabinets, cupboards, drawers, and installed analog photography films all have their hinges degrade when looking within them.

The Two Fibonaccis. The Fibonacci program is one favored among both functional and imperative programmers. The functional programmer is attracted to its simple recursive definition while the imperative programmer is attracted to the massive speed-up that loops and mutation offer it.

In Figure 1a we recreate the functional implementation in Java and in Figure 1b the imperative one.

```java
class FuncFibonacci {
  public int fib(int n) {
    return n <= 1 ? 1 : fib(n-1) + fib(n-2);
  }
}
```

(a) The functional Fibonacci in Java.

```java
class ImpFibonacci {
  public int fib(int n) {
    int a, b;
    for (a=b=1; n>0; n--, b=a+(a=b));
    return a;
  }
}
```

(b) The imperative Fibonacci in Java.

Fig. 1. A functional and imperative implementation of the Fibonacci function in Java.

It is simple to conclude that the program in Figure 1b computes the n-th Fibonacci number in $O(n)$ steps. The runtime of the program in Figure 1a can be found by solving the recurrence $T(n) = T(n-1) + T(n-2)$ with the boundary conditions that $T(0) = T(1) = O(1)$ which can be expressed as the following:

$$\begin{pmatrix} T(n+1) \\ T(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} T(n) \\ T(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} O(1) \\ O(1) \end{pmatrix}$$

The matrix has two eigenvalues: $\lambda_1 = \frac{1}{2}(1 - \sqrt{5})$ and $\lambda_2 = \frac{1}{2}(1 + \sqrt{5})$. With this information we can diagonalize this matrix to get:

$$\begin{pmatrix} T(n+1) \\ T(n) \end{pmatrix} = \frac{1}{5} \begin{pmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}^n \begin{pmatrix} -\sqrt{5} & 5\lambda_2 \\ \sqrt{5} & 5\lambda_1 \end{pmatrix} \begin{pmatrix} O(1) \\ O(1) \end{pmatrix}$$

Or simply,

$$T(n) = O(\lambda_1^n) + O(\lambda_2^n) \approx O(1.618^n) + O(-0.618^n) \approx O(1.618^n)$$

Fibonacci with Planned Obsolescence. To understand the effect that planned obsolescence has on the two Fibonacci programs in Figure 1 we have to study how the runtime degrades when the `fib` method is used multiple times. In the functional definition this is already the case: each recursive call is reusing the `FuncFibonacci` object. In the imperative definition this is not the case. Thus to make the comparison fair, let's consider the situation when we are interested in computing m Fibonacci numbers of roughly the same size n by calling `fib` m times.

Recall that timely and methodic planned obsolescence incurs a cost of one *extra* nanosecond on every function call; so the first call costs one nanosecond, the second costs two nanoseconds, the third costs three, etc... Thus, the cost of calling `ImpFibonacci`'s `fib` m times is $O(m \cdot n)$.

Computing the cost of `FuncFibonacci`'s `fib` is much more involved. We approach it through two steps: first, we find the cost of computing a single Fibonacci number under planned obsolescence, then we find the cost of computing multiple ones. Firstly, observe that after computing the n-th Fibonacci number, the `_uses` counter will be equal to the number of recursive calls, that is $O(\lambda_2^n)$. This is in fact an invariant. Therefore, the recursive computation spent $\sum_{i \leq O(\lambda_2^n)} i = O(\lambda_2^{2n})$ nanoseconds in planned obsolescence. Secondly, notice that this expression is correct only if the first recursive call waited one nanosecond. If instead it had to wait t nanoseconds, then every call would have to also wait t nanoseconds. Thus, we must add a $(t-1)\lambda_2^n$ factor. Now, to compute the full amount of waiting time we observe that at the beginning $t = 1$. And that after computing the m-th number (a Fibonacci number of size n) the cost is that of computing the number with t being the cost of computing the previous number. Finally, to compute the full runtime we solve the following recurrence: $T(0) = 1$ and $T(m) = O(\lambda_2^{2n}) + T(m-1)\lambda_2^n$ which gives the answer $O(\lambda_2^{n \cdot m})$.

First Results. Both the imperative and the functional computation of the Fibonacci numbers get slower under planned obsolescence. The imperative implementation is linearly slower while the functional one is exponentially slower than its previous implementation.

The Objectively Recursive Fibonacci. Naturally, the next question to ask is if the results generalize: first, that all imperative programs become slower, and second, that all functional programs become slower.

In this section, we answer the last question with the negative following a single simple yet crucial observation.

Crucial Observation. Planned Obsolescence punishes consumers who hold onto objects and rewards those who abandon objects shortly after using them.

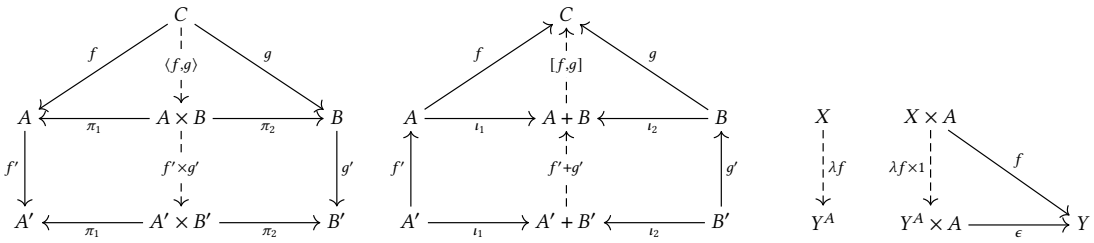
The observation lead us to conclude that Planned Obsolescence is punishing our implementations as they are reusing the same object to compute the m Fibonacci numbers. In the case of the functional implementation this punishment is doubled: the same object is used in the runner *and* in the recursive call.

```
1  class ObjFuncFibonacci {
2    public int fib(int n) {
3      ObjFuncFibonacci a = new ObjFuncFibonacci(),
4                       b = new ObjFuncFibonacci();
5      return n <= 1 ? 1 : a.fib(n-1) + b.fib(n-2);
6    }
7  }
8
9  class Runner {
10   public static void main(String[] args) {
11     for (int m=0; m<Integer.parseInt(args[1]); m++) {
12       System.out.println((new ObjFuncFibonacci()).fib(100));
13     }
14   }
15 }
```

Fig. 2. The Objectively Functional Fibonacci in Java.

What would be the consequences if we apply the *single use* practice to our programmatic objects to appease Planned Obsolescence? In Figure 2 we implement the recursive program from Figure 1a in a style we dub *Objectively Functional*. In this style, objects are only ever used once with state mutations being represented by returning a copy with the necessary modifications.

Observe that in the `fib` method of `ObjFuncFibonacci` is never called more than once on any object, even the recursive ones! In other words, not a single method call must wait more than one nanosecond during obsolescence. Thus, performing a simple complexity analysis, we can deduce that the running time of the objectively functional implementation is $O(m\lambda_2^n)$. This leads us to the next result:



(a) The Product. (b) The Coproduct. (c) The Exponential.

Fig. 3. A bi-cartesian closed category is a category with all products, coproducts, and exponentials.

Final Result. Using the objectively functional paradigm, the objectively recursive implementation *outperforms*, exponentially, the functional implementation under planned obsolescence.

4 BCCC^{po}: PLANNED OBSOLESCENCE, FORMALLY

In this section, we give a formal model, **BCCC**^{po}, for systems endowed with timely and methodic planned obsolescence as described in Section 3. Our presentation uses the standard categorical semantics of programs as morphisms in a bi-cartesian closed category **BCCC** [57]. Moreover, as is standard in categorical semantics, we represent abstract classes as a collection of co-algebras, classes as particular co-algebras, and objects as a product of a co-algebra and a state [39].

4.1 Background, Notation, and Basic Definitions

A bi-cartesian closed category is any category with all products, coproducts, and exponentials. We set some notation and recall the definitions that we will use in Figure 3. The product is usually defined as the upper triangle in Figure 3a. We use the lower rectangle in the diagram to introduce the notation $\langle f', g' \rangle$. By duality, the coproduct is also defined by the upper triangle in Figure 3b. We use the lower rectangle in the diagram to introduce the notation $f' + g'$. The exponential is defined in Figure 3c. We use the notation λg for the transpose of the g morphism and ϵ for the application of an exponential to an argument.

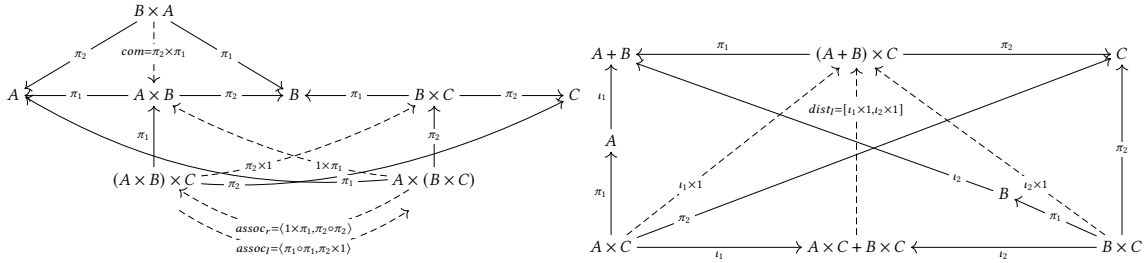
The product and coproduct satisfy the usual commutativity, associativity, and distributivity laws. In Figure 4 we define constructively[1] some of the terms—namely those we will use throughout this section—that witness these laws. In Figure 4a we define in one go *com*—the commutativity of the product—which is its own isomorphism and $assoc_l$ and $assoc_r$—the associativity of the product—which are isomorphic. In Figure 4b we define $dist_l$ that satisfies the distributivity of the product over the co-product. Its isomorphism is in fact not trivial to construct and in general it may not exist. However in a cartesian closed category where exponentials exist $dist_r$ also exists. Its construction is cumbersome, but it can be found in its entirety in Benini [5].

We recall that a functor F maps the objects and the morphisms of a category to another such that $F(id) = id$ and $F(f \circ g) = F(f) \circ F(g)$. If F maps the objects and morphisms of a category to other objects and morphisms in the same category then F is said to be an *endofunctor*.

4.2 BCCC^{po}: A Natural Transformation

In Definition 4.1 we define formally **BCCC**^{po} to be the composition of two natural transformations in some bi-cartesian closed category. The two natural transformations which define **BCCC**^{po} redefine

[1] As we wish to implement the formalism presented in this section for it to be executable we must adopt a constructive attitude and provide all the morphisms in terms of the fundamental ones.



(a) The proof that the product is commutative and associative.

(b) The proof that the product distributes over coproducts (to the left, from one side).

Fig. 4. Products and Coproducts satisfy the usual commutativity, associativity, and distributivity laws.

objects so that all their methods become objectively recursive and extend them with a lifetime tag that enables our timely and methodic planned obsolescence model.

Definition 4.1 ($\mathbf{BCCC}^{po}$*).* In the context of some category $\mathcal{C}$, $\mathbf{BCCC}^{po}$ is the natural transformation $po \circ \lambda$ such that the following conditions hold:

- $\mathcal{C}$ is a bi-cartesian closed category,
- λ is the *objectively recursive* natural transformation as defined in Definition 4.3, and,
- po is the *timely planned obsolescence* natural transformation as defined in Definition 4.5.

In the rest of this section we show how to derive the definition of the λ and the po natural transformations. In the process, thanks to our constructive point of view, we not only show that λ and po exist and uniquely so, we also show their definition.

Before we proceed, we recall that classes can be encoded as a co-algebra: $X \to T(X)$ for some polynomial functor T [39] as defined in Definition 4.2. Intuitively, a class contains some methods, which given some argument of type A, operate on an unobservable state X and return either a B without modifying the state, or return a C while modifying the state. In general, as a class contains multiple methods, it is more commonly defined as the co-algebra $X \to \prod_{i \leq n} (B_i + C_i \times X)^{A_i}$. In this paper however, we choose the simpler encoding $X \to \left(\left(\sum_{i \leq n} B_i\right) + \left(\sum_{i \leq n} C_i\right) \times X\right)^{\sum_{i \leq n} A_i}$ or simply $X \to (B + C \times X)^A$. The encoding we use is bigger than the standard one and hence our results hold for structures beyond classes. Nonetheless, our results are easily adapted to the standard encoding.

Definition 4.2 (Polynomial Functor). Given constant objects A, B, and C, a polynomial functor $T(X)$ takes the shape of $(B + C \times X)^A$.

Characterizing the Natural Transformations. The objectively recursive natural transformation λ is the one that intuitively modify the methods that return only a value to return the same value *and* the original unmodified state. Therefore, λ is the natural transformation that maps every polynomial functor $T(X) = (B + C \times X)^A$ into another polynomial functor $T^\lambda(X) = ((B + C) \times X)^A$ while leaving every other functor untouched. Our requirement that every method of a class $c : X \to T(X)$ mapped to $c^\lambda = \lambda(c) : X \to T^\lambda(X)$ return the same value and the original state if left untouched is equivalent to demanding that Figure 5 commutes.

The timely planned obsolescence natural transformation po maps an objectively functional class: a class in the codomain of λ, into a class whose state contains a lifetime tag while leaving everything untouched. Formally, po maps a co-algebra $c^\lambda : X \to ((B + C) \times X)^A$ into the co-algebra $c^\dagger : X \times \mathcal{L} \to$

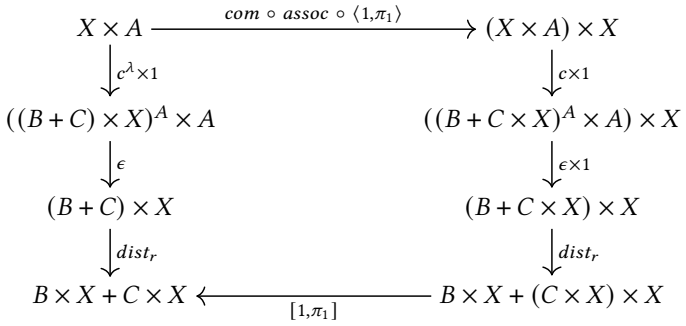


Fig. 5. The characterization of the λ natural transformation

$((B + C) \times (X \times \mathcal{L}))^A$. Again, our requirement that the method be left untouched—with the exception of the lifetime tag which ought to increment—can be expressed by the commuting diagram in Figure 6.

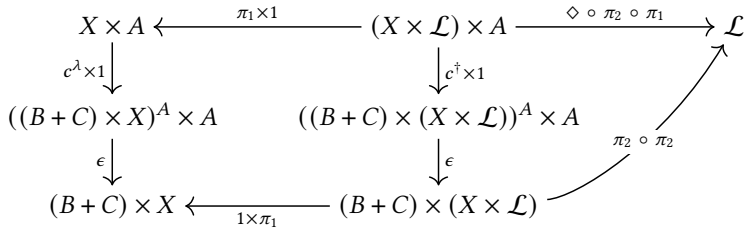


Fig. 6. The characterization of the *po* natural transformation

Lifetimes are added to our category through the object $\mathcal{L}$. We additionally assume two new morphisms: $0_{\mathcal{L}} : 1 \to \mathcal{L}$ and $\diamond : \mathcal{L} \to \mathcal{L}$ where 1 is the terminal object. Intuitively, $0_{\mathcal{L}}$ is the lifetime tag indicating no-use and the $\diamond$ morphism[2] increases the usage tag.

Defining the Natural Transformations. The main observation we use to derive the definition of λ is to use the isomorphism $dist_l$ of $dist_r$ instead, which gives us the commutative diagram in Figure 7.

From this diagram we can immediately read the definition of c^λ to be as in Definition 4.3.

Definition 4.3. The natural transformation λ maps every functor $T(X) = (B + C \times X)^A$ into $T^\lambda(X) = ((B + C) \times X)^A$ and every $c : X \to T(X)$ into $c^\lambda : X \to T^\lambda(X)$ such that:

$$c^\lambda = \lambda\ [\iota_1 \times \pi_1, \iota_2 \times \pi_1] \circ dist \circ (\epsilon \circ c) \times 1 \circ com \circ assoc \circ 1 \times \pi_1$$

Theorem 4.4. *c^λ as defined in Definition 4.3 is the only morphism that makes Figure 5 commute.*

Proof. Follows from $dist_l$ and $dist_r$ being isomorphic and the definition of the exponential (Figure 3c) □

To define the *po* natural transformation the same observation can be used on Figure 6 alongside a rearrangement to collapse the right-most triangle. This yields the diagram in Figure 8.

And again, from this diagram we can read the definition of $c^\dagger$ to be as in Definition 4.5

[2]We borrow this morphism from Linear Temporal Logic.

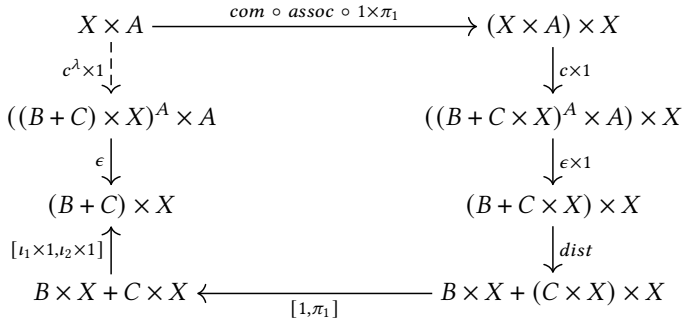


Fig. 7. The defining diagram of λ.

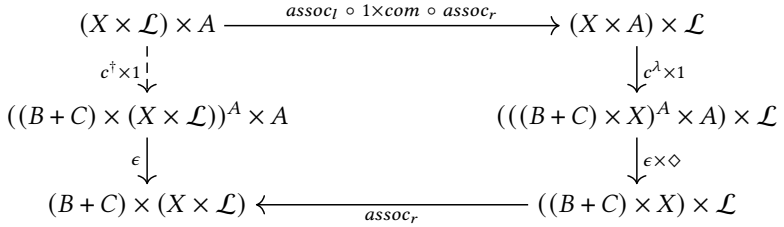


Fig. 8. The defining diagram of *po*

Definition 4.5. The natural transformation *po* maps every functor $T^\lambda(X) = ((B + C) \times X)^A$ into $T^\dagger(X) = ((B + C) \times (X \times \mathcal{L}))^A$ and every $c^\lambda : X \to T^\lambda(X)$ into $c^\dagger : X \times \mathcal{L} \to T^\dagger(X)$ such that:

$$c^\dagger = \lambda\, assoc_r \circ (\epsilon \circ c^\lambda) \times \diamond \circ assoc_l \circ 1 \times com \circ assoc_r$$

Theorem 4.6. *$c^\dagger$ as defined in Definition 4.5 is the only morphism that makes Figure 6 commute.*

Proof. Follows from the definition of the exponential (Figure 3c). □

5 JGEORGE: AN IMPLEMENTATION FOR THE JAVA VIRTUAL MACHINE

As described in Section 3, jGeorge targets class files executable on the Java Virtual Machine (JVM) and modifies them to add an integer field, `_uses`, a method, `_slowDown`, and a method call for every method.

In particular, the `_slowDown` method increments the `_uses` counter and starts a busy loop that terminates after `_uses` nanoseconds.

jGeorge targets the JVM as we believe it to be the natural target of a system like ours as it is the only (virtual) machine that we are aware of that treats objects as first-class values. jGeorge also targets class file executables as opposed to Java code for two reasons: first, it lends planned obsolescence to other JVM languages such as Scala, Clojure, or Jython for free. The second reason is more social: we wish to remain faithful to the design principles of the Java ecosystem: the Java compiler ought to be as simple as possible while the JVM ought to do all the heavy-lifting [30]. As we did not want to modify some given JVM's implementation we followed the instrumentation path to also allow for programs endowed with planned obsolescence to run on any JVM. The added bonus that this achieves is that programmers and library developers can run jGeorge on their source code once and distribute the binaries without burdening downstream consumers on injecting planned obsolescence.

The choice of units, the nanoseconds, was chosen empirically: We originally chose the microsecond, alas, many programs proved to take a considerable amount of time spanning hours and days on one occasion! To measure nanoseconds we use standard `long nanoTime()` method of `System` [70] which is available since Java 1.5. As a consequence, jGeorge has been developed with Java 1.5+ in mind. Moreover, `nanoTime()`'s Java documentation point out that under some circumstances unexpected results may be observed which would spawn a bug in jGeorge. Particularly:

> Differences in successive calls that span greater than approximately 292 years (263 nanoseconds) will not correctly compute elapsed time due to numerical overflow. [53]

jGeorge makes no effort into correctly handling successive calls to `nanoTime()` spanning more than 292 years even though, under planned obsolescence, this may be likely. We leave this as future work for the community to contribute.

jGeorge is implemented in a single Rust program with dependencies only on `std::string`, `std::str`, `std::fs`, and `std::env` over 860 lines. The program includes a parser for the binary format as documented in Chapter 4 of the JVM specification [54], code to perform the three injections mentioned earlier, and the necessary logic to adapt the type verification frames of every method.

The jGeorge binary accepts two command-line arguments: the class executable which must be modified and, optionally, the severity of the planned obsolescence: a multiplier for `_uses` with a default value of one. All the reported experiments in this paper use the default value.

5.1 Source Code and Data Availability

jGeorge and the source code necessary to run the evaluation in Section 6 are available under an open-source license and published on https://gitlab.com/gzakhour/jgeorge. In particular Appendix A includes the Rust source code of jGeorge formatted to fit on a single page. We took great care in simplifying the onboarding of fellow academics and artifact reviewers.

jGeorge can be compiled by executing `rustc jGeorge.rs` to produce the executable—on Linux. There is additionally a `run.sh` Bourne Again SHell—bash—script which reruns the case studies in Section 6.

6 EVALUATION

In this section we evaluate jGeorge empirically. Our evaluation is guided towards an answer to the following research questions:

RQ1 How applicable is jGeorge?
RQ2 How effective is jGeorge?
RQ3 How are different programming paradigms affected by planned obsolescence?
RQ4 How is the user-experience affected by planned obsolescence?

We answer the research questions in the context of four projects executable on the Java Virtual Machine. Three of these codebases are existing real-world projects, and a fourth codebase which we have written in two styles. We elaborate on each codebase in the following.

jEd. jEd is a subset of the ed text editor which we have rewritten in Java in an imperative style in a single `ImperativeEd.java` file. The ed text editor was originally developed in 1969 by Ken Thompson who developed it for the purpose of developing the UNIX operating system [16].
We have used jEd to modify its source code so that it becomes written in the objectively recursive style, `FunctionalEd.java`, as described in Section 3 and Definition 4.3. We recorded every interaction made with jEd to rewrite it and produced a trace of 1,387 instructions which can be automatically replayed to

	Without Planned Obsolescence	With Planned Obsolescence
`ImperativeEd`	241.16 ms	8328.92 ms
`FunctionalEd`	265.00 ms	347.80 ms

Table 1. Time to apply the rewrites from `ImperativeEd` to `FunctionalEd` using `jEd`

reproduce the objectively recursive implementation. In this case study we benchmark the two variants against eachother and against their timely and methodic planned obsolescence modifications.

The results are in Table 1. They show that `FunctionalEd.java` can be produced in roughly 250 milliseconds without planned obsolescence using both the imperative and functional implementation. Unsurprisingly, when planned obsolescence is enabled, the imperative implementation's balloons up: it takes eight seconds to reproduce `FunctionalEd.java`. Surprisingly though, the objectively functional implementation takes only 350 milliseconds thanks to its single use policy, massively outperforming the imperative implementation.

Propel. Propel is an automated inductive theorem prover developed by Zakhour et. al [76, 77] that is written in Scala that can verify algebraic properties of purely-functional Scala code. For example it can prove that addition is commutative and associative, or that the "pair-wise" operation parametrized over some function is commutative, associative, and idempotent (or any combination) whenever its parametrized function is commutative, associative, and idempotent (or any combination respectively). Using Scala native, Propel is normally distributed as a native binary. However as it, and its dependencies, are written exclusively in Scala we only compile it as is standard to the JVM.

Propel comes with 128 benchmark programs out of the box including a selection from the TIP (Tons of Inductive Proofs) benchmark [12], a few CRDTs (Conflict-free Replicated Data Types) [59], and some type-class laws [35, 73]. In this evaluation we compare Propel against itself with planned obsolescence enabled on its provided benchmarks.

We plot the results in Figure 9 where every point is a theorem from the Propel benchmark. The x-axis is the time required to be prove the theorem on the Java Virtual Machine and the y-axis is the time required to prove it on the JVM with planned obsolescence enabled. The diagonal line is the $y = x$ line. It is clear that most data points are relatively close to that diagonal but still above it while a few others are way higher, showing that jGeorge does indeed implement planned obsolescence. Surprisingly, upon closer inspection, a small minority is in fact below the line, showing that planned obsolescence could speed up the program in some cases. We examined these data points and we conjecture that these are the theorems that are proven without back-tracking: where the recursive calls never unwind and accumulate usage penalties.

Jayway JsonPath. Jayway JsonPath [42] is an actively developed open-source Java library implementing an XPath-like query language targeting JSON documents. As of writing, JsonPath has nine thousand stars on Github, ten releases, ninety-one contributors, and shy of two thousand forks. It is being used by almost one hundred thousand other Github repositories.

Jayway JsonPath comes with 748 unit tests executed via JUnit. We measure the effects of timely and methodic planned obsolescence on Jayway JsonPath by comparing JUnit's test execution statistics on it and on the version with planned obsolescence enabled.

We plot the results in Figure 10 in a scatter plot similar to Propel's report. The effects of planned obsolescence on the test runner are also similar to Propel's: almost all points are above the diagonal with the majority relatively close to the diagonal. Moreover, a very small minority of the tests are faster when executed with planned obsolescence. The difference with respect to Propel though is that many more test cases are way above the diagonal.

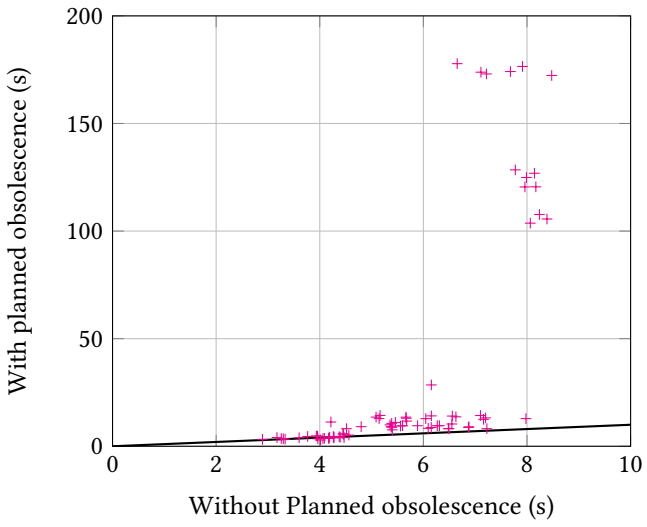


Fig. 9. Time to prove a theorem with Propel. Every point is a theorem.

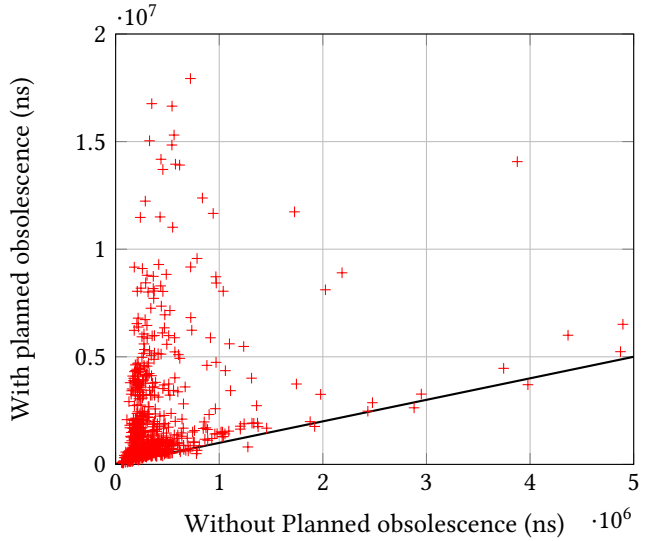


Fig. 10. Time to execute the JsonPath tests. Every point is a test.

FizzBuzzEnterpriseEdition. FizzBuzzEnterpriseEdition [13] is an open-source Java application that aims at implementing all the best practices from enterprise software around a simple and small logic: that of *FizzBuzz*, the well known interview question [69]. FizzBuzzEnterpriseEdition has twenty-three thousand Github stars, seven hundred forks, and thirty contributors. We use FizzBuzzEnterpriseEdition for multiple reasons: first, we use it to study the effects of timely and methodic planned obsolescence on enterprise software, second, FizzBuzzEnterpriseEdition uses Spring Boot, a popular Java framework and a hallmark of mature and enterprise software which we also wish to evaluate.

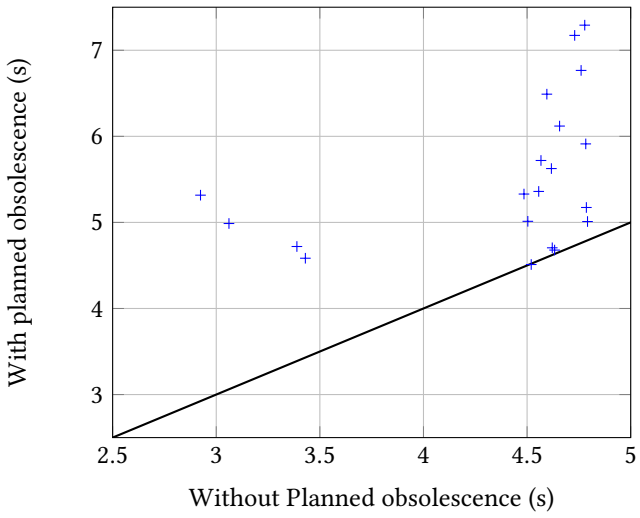


Fig. 11. Time to compute the FizzBuzz sequence. Every point is an input number.

It is worth relating that not one case terminated within twenty hours in our evaluation of FizzBuzzEnterpriseEdition when we applied planned obsolescence to the class files of Spring Boot. Thus, in this case study, we limit the reporting to FizzBuzzEnterpriseEdition proper. We have not investigated why Spring Boot performs so poorly under planned obsolescence.

The results are presented in Figure 11 similarly to Propel's and JsonPath's. Unlike the other reports, there are no data points which are faster under planned obsolescence.

The results of the experiments on the four use cases described earlier help us answer the original research questions. We elaborate on these answers in the remainder of this section.

RQ1: How applicable is jGeorge? jGeorge is widely applicable to the Java Virtual Machine ecosystem. That we were able to run modified Spring Boot applications and Scala applications relying on the Scala runtime answer the question with the positive.

RQ2: How effective is jGeorge? jGeorge effectively degrades the runtime of multi-use objects. With the exception of a very small set of theorems in Propel and tests in JsonPath, jGeorge successfully degraded the runtime with varying severity.

RQ3: How are different programming paradigms affected by planned obsolescence? The jEd case study demonstrate that the objectively functional paradigm as described in Section 3 is hardly affected by planned obsolescence thanks to its single use property. On the other hand, the imperative paradigm suffers greatly.

Moreover, as Propel is written in Scala, it can be said to be more functional than JsonPath that is written in an enterprise-friendly object-oriented paradigm. In Figure 9 we see many theorems close to the diagonal while a few are higher, on the other hand in Figure 10 we see that the amount of tests that are much higher above the diagonal is significant. This additionally supports our claim.

RQ4: How is the user-experience affected by planned obsolescence? In general, the user experience is degraded which contributes positively to planned obsolescence. Nonetheless, the user need not experience this degradation. If a program is implemented in an objectively functional paradigm, i.e.

when the programmer engages with planned obsolescence and adopts single use, creating and destroying objects frequently, the user is not made aware of planned obsolescence through experience.

7 RELATED WORKS

While we are the first to propose a dynamic semantics of lifetimes as planned obsolescence, the latter is not new to software systems. In other words, neither degradation of the runtime is new, nor its deliberateness. In this section, we summarize the existing litterature on these lines of work.

Planned Obsolescence Guides for Technology. The first "guide" to obsolescence in technology is due to Arthur Sloane. His decision to redesign General Motor's cars *yearly* has influenced almost every technology subsequently. From his decision in the twenties we can trace a line into yearly fashions such as the release cycle of the Apple iPhone.

Nonetheless, Sloane's decisions are hardly guides, but rather policies. The original guide to planned obsolescence is, as mentioned in Section 2 due to J. George Frederick [27], the namesake of jGeorge. From his foundational essay a slew of papers was written about the positivity of planned obsolescence on technological obsolescence: the latter is drastically slowed down or even halted without the former [22]. Waldman [74] argues that monopolies investing in research and development of technologies is an existential threat—to the monopoly—if planned obsolescence is not baked in the technologies it is developing and researching. That last point is corroborated by Grout and Park [32] and extended, not only to monopolies, but to any company in a competitive market. Strausz [65] argues that planned obsolescence is generally good, for both the producer and the consumer. Since it encourage frequent repurchase and frequent redevelopment, planned obsolescence creates a tight feedback loop in which customers can communicate back their opinions on whether the redeveloped product has improved or regressed in quality. Thus, just as a shorter software life cycle aids in the development of a high-quality software—as popularized by the Agile manifesto for software [26]—a shorter production–consumption cycle aids in the development of a high-quality hardware.

Obsolescence in Software. Software goes obsolete everyday for a myriad of reasons. The most common being that a software loses its user base, either because the software's host—the hardware—being no longer relevant or because a better software has been developed. This mode of obsolescence is in line with *technological obsolescence* as we describe it in Section 2.

Nonetheless occurrences of deliberate obsolescence of software, if not documented, are suspected. For example, in 2018, the Italian Government opened an investigation into Apple and Samsung about their deliberate use of degradation in the *software* as a means of demonstrating a non-existing degradation in the hardware [28]. In 2014, Epson, HP, and Canon have been accused of using software that would refuse to print if the cartridge inks were not replaced [28]. Recently, on March 5th 2025, Cory Doctorow penned a piece about Brother starting to engage in the very same practice that its competitors are accused of [18]. These examples however use software degradation as a means to an end of hardware degradation and not as an end as we have done.

Software *rot* is a well-observed phenomenon in the developer culture [38]. But software rot is accidental. An example of deliberate software obsolescence comes from 2016: after a heated argument between a single developer and a software company which dragged in a package manager, who sided with the software company, Azer Koçulu, in protest, unpublished his 11-line long left-pad project from the package manager which lead to major software, such as Airbnb and Facebook, breaking [10].

Jang et al. [40] explore other vectors that could make software obsolete. For example, the deliberate choice of the developer depending on a cloud provider or a third party dependency, accelerate the eventuality of obsolescence of the software.

Technofeudalism and Enshittification. While Varoufakis' Technofeudalism [72] and Doctorow's Enshittification [17] can be considered as technological guides for planned obsolescence, we choose to discuss them separately.

Varoufakis's main observation is that a few companies such as Amazon and Microsoft own the digital landscape, and can thus enforce a large *cloud rent* fee, upwards of 40% to digital platforms, which must trickle the fees down unto their users. Technofeudalism is thus the metaphor that these few companies are equivalent to medieval Europe's feudal lords and the platforms are equivalent to the lord's vassals.

Here, Doctorow's Enshittification principle kicks in. In particular, it applies to "platforms" such as Facebook, TikTok, Instagram, etc... which can be reduced down to four components: the software, the software's owner—often a company, the software's users, and crucially, the company's stakeholders who have an invisible hand into the decisions done by the company and hence into the software thanks to Conway's Law [14]. In order to grow the software, its owners appease the software's users. Then, in order to grow the software more, the owners select a small subset of the users: the business users, to appease. However, at this point, large value is attached to the software. To appease the stakeholders, the owners must abuse their users, business users included. At this stage, Doctorow's enshittification principle kicks in, and the platform, i.e. the software, "begins to die".

REFERENCES

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 237–268. https://doi.org/10.1145/103135.103138

[2] Jonathan Aldrich. 2013. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 101–116.

[3] Princeton Graphic Arts. 2021. The Supreme Court and Paper Collars. https://graphicarts.princeton.edu/2021/03/21/the-supreme-court-and-paper-collars/ Accessed: 2025-03-15.

[4] Jeffrey M. Barth. 1977. Shifting garbage collection overhead to compile time. *Commun. ACM* 20, 7 (July 1977), 513–518. https://doi.org/10.1145/359636.359713

[5] Marco Benini. 2014. Cartesian closed categories are distributive. arXiv:1406.0961 [math.CT] https://arxiv.org/abs/1406.0961

[6] Jan Blommaert and Chris Bulcaen. 2000. Critical discourse analysis. *Annual review of Anthropology* 29, 1 (2000), 447–466.

[7] Gaspar Brändle. 2015. Obsolescence: Planned, Progressive, Stylistic. *The Wiley Blackwell Encyclopedia of Consumption and Consumer Studies* (2015), 1–2.

[8] Jane Celia Busch. 1983. *The throwaway ethic in America.* University of Pennsylvania. 84–84 pages.

[9] Talib E Butt, M Camilleri, Parneet Paul, and KG Jones. 2015. Obsolescence types and the built environment–definitions and implications. *International Journal of Environment and Sustainable Development* 14, 1 (2015), 20–39.

[10] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. 2022. On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2695–2708. https://doi.org/10.1109/TSE.2021.3068901

[11] Chromium Security. 2020. Memory Safety. https://www.chromium.org/Home/chromium-security/memory-safety/ Accessed: 2025-03-15.

[12] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2015. TIP: tons of inductive problems. In *International Conference on Intelligent Computer Mathematics*. Springer, 333–337.

[13] Enterprise Quality Coding. 2012. FizzBuzz Enterprise Edition. https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition Accessed: 2025-03-15.

[14] Melvin E Conway. 1968. How do committees invent. *Datamation* 14, 4 (1968), 28–31.

[15] Cybersecurity and Infrastructure Security Agency. 2022. *Software Memory Safety.* Technical Report. U.S. Department of Homeland Security. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF Accessed: 2025-03-15.

[16] L. Peter Deutsch and Butler W. Lampson. 1967. An online editor. *Commun. ACM* 10, 12 (Dec. 1967), 793–799. https://doi.org/10.1145/363848.363863

[17] Cory Doctorow. 2024. 'Enshittification' is coming for absolutely everything. *Financial Times* 8 (2024).

[18] Cory Doctorow. 2025. Show me the incentives, I will show you the outcome. https://pluralistic.net/2025/03/05/printers-devil/#show-me-the-incentives-i-will-show-you-the-outcome Accessed: 2023-10-01.

[19] Tim Dowling. 2001. *Inventor of the Disposable Culture: King Camp Gillette 1855-1932.* Faber & Faber Limited.

[20] Econospeak. 2023. Is Progressive Obsolescence a Path Toward a Sustainable Economy? https://econospeak.blogspot.com/p/is-progressive-obsolescence-path-toward.html Accessed: 2025-03-15.

[21] Matthias Felleisen. 2019. The Laffer Curve of Types. https://felleisen.org/matthias/Thoughts/The_Laffer_Curve_of_Types.html Accessed: 2025-03-15.

[22] Arthur Fishman, Neil Gandal, and Oz Shy. 1993. Planned Obsolescence as an Engine of Technological Progress. *The Journal of Industrial Economics* 41, 4 (Dec. 1993), 361. https://doi.org/10.2307/2950597

[23] Michel Foucault. 1971. Orders of discourse. *Social science information* 10, 2 (1971), 7–30.

[24] Michel Foucault. 2013. *Archaeology of knowledge*. routledge.

[25] Michel Foucault and Anthony M Nazzaro. 1972. History, discourse and discontinuity. *Salmagundi* 20 (1972), 225–248.

[26] Martin Fowler, Jim Highsmith, et al. 2001. The agile manifesto. *Software development* 9, 8 (2001), 28–35.

[27] Justus George Frederick. 1928. Is progressive obsolescence the path toward increased consumption? *Advertising and Selling* 5 (1928), 19–20.

[28] Baris Batuhan Gecit. 2020. Planned obsolescence: a keyword analysis. *Pressacademia* 7, 4 (Dec. 2020), 227–233. https://doi.org/10.17261/pressacademia.2020.1335

[29] Maria Gini. n.d.. Object-Oriented Programming in Prolog. https://www-users.cse.umn.edu/~gini/prolog/oop.html Accessed: 2025-03-15.

[30] James Gosling. 2017. Simula: a personal journey. https://www.youtube.com/watch?v=ccRtIdlTqlU Uploaded by: UiO Realfagsbiblioteket, Accessed: 2025-03-15.

[31] Michael Greenberg. 2019. The dynamic practice and static theory of gradual typing. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 6–1.

[32] Paul A. Grout and In-Uck Park. 2005. Competitive Planned Obsolescence. *The RAND Journal of Economics* 36, 3 (2005), 596–612. http://www.jstor.org/stable/4135231

[33] Robert Harper and Greg Morrisett. 1995. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 130–141. https://doi.org/10.1145/199448.199475

[34] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1029873.1029883

[35] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[36] InfoQ. 2010. Interview: John Johnson and Armstrong on Object-Oriented Programming. https://www.infoq.com/interviews/johnson-armstrong-oop/ Accessed: 2025-03-15.

[37] John A Interrante and Mark A Linton. 1990. *Runtime access to type information in C++*. Computer Systems Laboratory, Stanford University.

[38] Clemente Izurieta and James M Bieman. 2013. A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal* 21 (2013), 289–323.

[39] Bart Jacobs. 1995. Objects and classes, co-algebraically. In *Object orientation with parallelism and persistence*. Springer, 83–103.

[40] Esther Jang, Matthew Johnson, Edward Burnell, and Kurtis Heimerl. 2017. Unplanned Obsolescence: Hardware and Software After Collapse. In *Proceedings of the 2017 Workshop on Computing Within Limits* (Santa Barbara, California, USA) *(LIMITS '17)*. Association for Computing Machinery, New York, NY, USA, 93–101. https://doi.org/10.1145/3080556.3080566

[41] Ralph Johnson. 2007. Erlang, the Next Java. http://web.archive.org/web/20071014185458/http://cincomsmalltalk.com/userblogs/ralph/blogView?entry=3364027251 Accessed: 2025-03-15.

[42] @json-path (Github). 2024. JsonPath. https://github.com/json-path/JsonPath Accessed: 2025-03-15.

[43] Alan C. Kay. 1996. *The early history of Smalltalk*. Association for Computing Machinery, New York, NY, USA, 511–598. https://doi.org/10.1145/234286.1057828

[44] Markus Krajewski. 2014. The great lightbulb conspiracy. *IEEE Spectrum* (2014). https://doi.org/10.1109/MSPEC.2014.6905492

[45] Ole Lehrmann Madsen, Birger Mø-Pedersen, and Kristen Nygaard. 1993. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., USA.

[46] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (San Jose, California) *(ASID '06)*. Association for Computing Machinery, New York, NY, USA, 25–33. https://doi.org/10.1145/1181309.1181314

[47] James Livingston. 1987. The social analysis of economic history and theory: Conjectures on late nineteenth-century American development. *The American Historical Review* 92, 1 (1987), 69–95.

[48] Matt Miller. 2019. Trends, Challenges, and Shifts in Software Vulnerability Mitigation. In *BlueHat*. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf Accessed:

2025-03-15.
[49] Jaime Niño. 2007. The cost of erasure in Java generics type system. *Journal of Computing Sciences in Colleges* 22, 5 (2007).
[50] James Noble. 2009. The myths of object-orientation. In *European Conference on Object-Oriented Programming*. Springer.
[51] Kristen Nygaard and Ole-Johan Dahl. 1978. *The development of the SIMULA languages.* Association for Computing Machinery, New York, NY, USA, 439–480. https://doi.org/10.1145/800025.1198392
[52] Office of the National Cyber Director. 2024. *Back to the Building Blocks: A Path Toward Secure and Measurable Software.* Technical Report. The White House. http://web.archive.org/web/20240405180001/https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf Accessed: 2025-03-15 (The Ides of March; the day the world was freed of Julius Caeser, the tyrant, the destroyer of the republic, and the mass murderer.).
[53] Oracle. 2014. System (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime-- Accessed: 2025-03-15.
[54] Oracle. 2023. Chapter 4: The class File Format – Java Virtual Machine Specification, Java SE 23. https://docs.oracle.com/javase/specs/jvms/se23/html/jvms-4.html Accessed: 2025-03-15.
[55] Matt Parsons. 2017. OOPH: Data Inheritance. https://www.parsonsmatt.org/2017/02/17/ooph_data_inheritance.html Accessed: 2025-03-15.
[56] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 3 (April 2021), 73 pages. https://doi.org/10.1145/3443420
[57] Benjamin C Pierce. 1991. *Basic category theory for computer scientists.* MIT press.
[58] C. Ruggieri and T. P. Murtagh. 1988. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 285–293. https://doi.org/10.1145/73560.73585
[59] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.
[60] William H Shaw. 1979. "The Handmill Gives You the Feudal Lord": Marx's Technological Determinism. *History and Theory* 18, 2 (1979), 155–176.
[61] Giles Slade. 2007. *Made to break: Technology and obsolescence in America.* Harvard University Press.
[62] Allie Slemon. 2025. Absences and Silences in Critical Discourse Analysis: Methodological Reflections. *International Journal of Qualitative Methods* 24 (Feb. 2025). https://doi.org/10.1177/16094069251321250
[63] Ed Stabler. 1986. Object-oriented programming in Prolog. *AI Expert* 1, 2 (Nov. 1986), 46–57.
[64] Guy Steele. 2003. RE: What's so cool about Scheme? https://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03269.html Accessed: 2025-03-15.
[65] Roland Strausz. 2009. Planned Obsolescence as an Incentive Device for Unobservable Quality. *The Economic Journal* 119, 540 (2009), 1405–1421. http://www.jstor.org/stable/40271396
[66] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe manual memory management in Cyclone. *Science of Computer Programming* 62, 2 (2006), 122–144. https://doi.org/10.1016/j.scico.2006.02.003 Special Issue: Five perspectives on modern memory management - Systems, hardware and theory.
[67] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19 (2014), 1665–1705.
[68] The New York Times. 1964. J. G. Frederick, 82, a Writer, Is Dead; Author of Books on Business. *The New York Times* (1964). https://www.nytimes.com/1964/03/24/archives/j-g-frederick-82-a-writer-is-dead-author-of-books-on-business.html Accessed: 2023-10-01.
[69] Ben Tyers. 2017. *Fizz Buzz.* Apress, Berkeley, CA, 117–118. https://doi.org/10.1007/978-1-4842-2644-5_59
[70] @typeable@mastodon.social. 2023. Did you know #Java's System has a 'nanoTime()' method? That's a note in its javadoc: (Post on Mastodon). https://mastodon.social/@typeable/114133247051175038 Accessed: 2025-03-15.
[71] Teun A Van Dijk. 2015. Critical discourse analysis. *The handbook of discourse analysis* (2015), 466–485.
[72] Yanis Varoufakis. 2024. *Technofeudalism.* Melville House Publishing, Brooklyn, NY.
[73] Philip Wadler. 1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–14.
[74] Michael Waldman. 1996. Planned Obsolescence and the R&;D Decision. *The RAND Journal of Economics* 27, 3 (1996), 583. https://doi.org/10.2307/2555845
[75] JM Yohe. 1967. Machine Language Programming How and Why. In *Proceedings of the 1967 Army Numerical Conference, ARO-D Report 67-3*. US Army Research Office, 3.
[76] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Checking CRDT Convergence. *Proc. ACM Program. Lang.* 7, PLDI, Article 162 (June 2023), 24 pages. https://doi.org/10.1145/3591276
[77] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2024. Automated Verification of Fundamental Algebraic Laws. *Proc. ACM Program. Lang.* 8, PLDI, Article 178 (June 2024), 24 pages. https://doi.org/10.1145/3656408

A JGEORGE RUST SOURCE CODE[3]

```rust
macro_rules! get_n_bytes { ($bytes:expr, $i:expr, $n:expr, $t:ty) => {{ if $i+$n - 1 < $bytes. len() { let res: $t = (0 .. $n). fold(0, |r, j| r. overflowing_shl(8). 0+ ($bytes[$i+ j] as $t)); $i += $n; Ok(res) } else { Err(format!("unexpected EOF: expected at least {} byte(s)", $n)) } }}; } macro_rules! parse_many { ($num:expr, $parser:expr) => { (0 .. $num). map(|_| $parser). collect:: <Result<Vec<_>, _>>() }} macro_rules! extend_with { ($buf:expr, $val:expr, $n:expr) => {{ let bytes: Vec<u8> = Vec:: from($val. to_be_bytes()); assert!(bytes. len() >= $n); $buf. extend(bytes[bytes. len()-$n .. ]). into_iter()); }}; } struct Parser { index: usize, bytes: Vec<u8> } impl Parser { fn from(bytes: Vec<u8>) -> Parser { Parser { index: 0, bytes } } fn u1(& mut self) -> Result<u8, String> { get_n_bytes!(self. bytes, self. index, 1 , u8) } fn u2(& mut self) -> Result<u16, String> { get_n_bytes!(self. bytes, self. index, 2, u16) } fn u4(& mut self) -> Result<u32, String> { get_n_bytes!(self. bytes, self. index, 4, u32) } } struct Class { minor_version: u16, major_version: u16, constant_pool: Vec<ConstantPoolEntry>, access_flags: u16, this_class: u16, super_class: u16, interfaces: Vec<u16>, fields: Vec<Field>, methods: Vec<Method>, attributes: Vec<Attribute>, } impl Class { fn from_parser(parser:& mut Parser) -> Result<Self, String> { let magic = parser. u4()?; if magic != 0xCAFEBABE { return Err(format!("magic is wrong: expected 0xcafebabe, found 0x{magic:x}")); } let minor_version = parser. u2()? ; let major_version = parser. u2()?; let constant_pool_size = parser. u2()?; let mut constant_pool = Vec:: with_capacity(constant_pool_size as usize); constant_pool. push(ConstantPoolEntry:: Invalid); let mut index = 1; while index < constant_pool_size { constant_pool. push(ConstantPoolEntry:: from_parser(parser)?); if let Some(ConstantPoolEntry:: Long { .. }) | Some(ConstantPoolEntry:: Double { .. }) = constant_pool. last() { constant_pool . push(ConstantPoolEntry:: Invalid); index += 2; } else { index += 1; } } let class = Class { minor_version, major_version, access_flags: parser. u2()?, this_class: parser. u2()?, super_class: parser. u2()?, interfaces: parser. many!(parser. u2()?, parser. u2())?, fields: parse_many!(parser. u2()?, Field:: from_parser(parser,& constant_pool))?, methods: parse_many!(parser. u2()?, Method:: from_parser(parser,& constant_pool))?, attributes: parse_many!(parser. u2()?, Attribute:: from_parser(parser,& constant_pool))?, constant_pool, }; if parser. index != parser. bytes. len() { Err(format!("[Class] Exptected EOF, {} bytes left", parser. bytes. len() - parser. index)) } else { Ok(class) } } fn dump(& self) -> Vec<u8> { let mut buf = Vec:: from([ 0xCA, 0xFE, 0xBA, 0xBE ]); extend_with!(buf, self. minor_version , 2); extend_with!(buf, self. major_version , 2); extend_with!(buf, self. constant_pool. len(), 2); for cp_item in& self. constant_pool { cp_item. dump(& mut buf); } extend_with!(buf, self. access_flags, 2); extend_with!(buf, self. this_class, 2); extend_with!(buf, self. super_class, 2); extend_with!(buf, self . interfaces. len(), 2); for interface in& self. interfaces { extend_with!(buf, interface, 2); } extend_with!(buf, self. fields. len(), 2); for field in& self . fields { field. dump(& mut buf); } extend_with!(buf, self. methods. len(), 2) ; for method in& self. methods { method. dump(& mut buf); } extend_with!(buf, self. attributes. len(), 2); for attr in& self. attributes { attr. dump(& mut buf); } buf } fn cp_entry(& mut self, entry: ConstantPoolEntry) -> u16 { self. constant_pool. iter(). position(|item| *item == entry). unwrap_or_else(|| { let index = self. constant_pool. len(); self. constant_pool. push(entry); index }) as u16 } fn cp_utf8(& mut self, name:& str) -> u16 { self. cp_entry(ConstantPoolEntry:: ValidUtf8 { value: name. to_string() }) } fn enable_planned_obsolescence(& mut self, severity: u8) { use ConstantPoolEntry :: *; if self. access_flags& (0x0200 | 0x1000 | 0x2000 | 0x4000 | 0x8000) > 0 { return } let [uses_field_hi, uses_field_lo] = { let name_index = self. cp_utf8("_uses"); let descriptor_index = self. cp_utf8("I"); let name_and_type_index = self. cp_entry(NameAndType { name_index, descriptor_index }); self. cp_entry(Fieldref { class_index: self. this_class, name_and_type_index }). to_be_bytes() }; { let name_index = self. cp_utf8("_uses"); let descriptor_index = self. cp_utf8("I"); self. fields. push(Field { access_flags: 1, name_index, descriptor_index, attributes: vec![] }) } let [slowdown_method_hi, slowdown_method_lo] = { let name_index = self. cp_utf8("_slowDown"); let descriptor_index = self. cp_utf8("()V"); let name_and_type_index = self. cp_entry(NameAndType { name_index, descriptor_index }); self. cp_entry(Methodref { class_index: self. this_class, name_and_type_index }). to_be_bytes() }; let [nano_time_method_hi, nano_time_method_lo] = { let class_index = { let name_index = self. cp_utf8("java/lang/System"); self. cp_entry(Class { name_index }) }; let name_index = self. cp_utf8("nanoTime"); let descriptor_index = self. cp_utf8("()J"); let name_and_type_index = self. cp_entry(NameAndType { name_index, descriptor_index }); self. cp_entry(Methodref { class_index, name_and_type_index }). to_be_bytes() }; let code_name_index = self. cp_utf8("Code"); let stack_map_table = self. cp_utf8("StackMapTable"); for method in self. methods. iter_mut() { method. enable_planned_obsolescence(& self. constant_pool, slowdown_method_hi, slowdown_method_lo); } let code = vec![ 42, 89, 180, uses_field_hi, uses_field_lo, 4, 96, 181, uses_field_hi, uses_field_lo, 184, nano_time_method_hi, nano_time_method_lo, 64, 42, 180, uses_field_hi, uses_field_lo, 16, severity, 104, 133, 66, 184, nano_time_method_hi, nano_time_method_lo, 31, 101, 33, 148, 156, 0, 6, 167, 255 , 246, 177, ]; let code_len = code. len() as u32; let stack_map_bytes = vec![0, 2, 253, 0, 23, 4, 4, 12]; let stack_map_bytes_len = stack_map_bytes. len() as u32; let method = Method { access_flags: 1, name_index: self. cp_utf8("_slowDown"), descriptor_index: self. cp_utf8("()V"), attributes: vec![ Attribute:: Code { attribute_name_index: code_name_index, max_stack: 4, max_locals: 5, code, exception_table: vec![], attributes: vec![ Attribute:: Other { attribute_name_index: stack_map_table, attribute_length: stack_map_bytes_len, bytes: stack_map_bytes } }, attribute_length: 2+ 2+ 4+ code_len+ 2+ 2+ (2+ 4+ stack_map_bytes_len), }] }; self. methods. push(method) ; } } #[derive(Debug, PartialEq)] enum ConstantPoolEntry { InvalidUtf8 { bytes: Vec<u8> }, ValidUtf8 { value: String }, Integer { value: i32 }, Float { value: f32 }, Long { value: i64 }, Double { value: f64 }, Class { name_index: u16 }, String { string_index: u16 }, Fieldref { class_index: u16, name_and_type_index: u16 }, Methodref { class_index: u16, name_and_type_index: u16 }, InterfaceMethodref { class_index: u16, name_and_type_index: u16 }, NameAndType { name_index: u16, descriptor_index: u16 }, MethodHandle { reference_kind: u8, reference_index: u16 }, MethodType { descriptor_index: u16 }, Dynamic { bootstrap_method_attr_index: u16, name_and_type_index: u16 }, InvokeDynamic { bootstrap_method_attr_index: u16, name_and_type_index: u16 }, Module { name_index: u16 }, Package { name_index: u16 }, Invalid, } impl ConstantPoolEntry { fn from_parser(parser:& mut Parser) -> Result<Self, String> { use ConstantPoolEntry:: *; match parser. u1()? { 1 => { let length = parser. u2()?; let bytes = parse_many!(length, parser. u1())?; Ok(if let Ok(value) = std:: string:: String:: from_utf8(bytes. clone()) { ValidUtf8 { value } } else { InvalidUtf8 { bytes } }) }, 3 => Ok(Integer { value: parser. u4()? as i32 }) , 4 => Ok(Float { value: f32:: from_bits(parser. u4()?) }), 5 => Ok(Long { value: (parser. u4()? as i64). overflowing_shl(32). 0+ (parser. u4()? as i64) }), 6 => Ok(Double { value: f64:: from_bits((parser. u4()? as u64). overflowing_shl(32). 0+ (parser. u4()? as u64)) }), 7 => Ok(Class { name_index: parser. u2()? }), 8 => Ok(String { string_index: parser. u2()? }), 9 => { let class_index = parser. u2()?; let name_and_type_index = parser. u2()?; Ok(Fieldref { class_index, name_and_type_index }) }, 10 => { let class_index = parser. u2()?; let name_and_type_index = parser. u2()?; Ok(Methodref { class_index, name_and_type_index }) }, 11 => { let class_index = parser. u2()? ; let name_and_type_index = parser. u2()?; Ok(InterfaceMethodref { class_index, name_and_type_index }) }, 12 => { let name_index = parser. u2()?; let descriptor_index = parser. u2()?; Ok(NameAndType { name_index, descriptor_index }) }, 15 => { let reference_kind = parser. u1()?; let reference_index = parser . u2()?; Ok(MethodHandle { reference_kind, reference_index }) }, 16 => { let descriptor_index = parser. u2()?; Ok(MethodType { descriptor_index }) } 17 => { let bootstrap_method_attr_index = parser. u2()?; let name_and_type_index = parser. u2()?; Ok(Dynamic { bootstrap_method_attr_index, name_and_type_index }) } 18 => { let bootstrap_method_attr_index = parser. u2()?; let name_and_type_index = parser. u2()?; Ok(InvokeDynamic { bootstrap_method_attr_index, name_and_type_index }) }, 19 => Ok(Module { name_index: parser. u2()? }), 20 => Ok(Package { name_index: parser. u2()? }), b => Err(format!("Invalid constant pool tag: {b}")), } } fn dump(& self, buf:& mut Vec<u8>) { use ConstantPoolEntry:: *; match self { InvalidUtf8 { bytes } => { extend_with!(buf, 0u8, 1); extend_with!(buf, bytes. len(), 2); buf. extend(bytes); }, ValidUtf8 { value } => { extend_with!(buf, 1u8, 1); extend_with!(buf, value. len(), 2); buf. extend(value. as_bytes()); }, Integer { value } => { extend_with!(buf, 3u8, 1); extend_with!(buf, value, 4); }, Float { value } => { extend_with!(buf, 4u8, 1); extend_with!(buf, value, 4); }, Long { value } => { extend_with!(buf, 5u8, 1); extend_with!(buf, value, 8); }, Double { value } => { extend_with!(buf, 6u8, 1); extend_with!(buf, value, 8); } Class { name_index } => { extend_with!(buf, 7u8, 1); extend_with!(buf, name_index, 2); }, String { string_index } => { extend_with!(buf, 8u8, 1); extend_with!(buf, string_index, 2); }, Fieldref { class_index, name_and_type_index } => { extend_with!(buf, 9u8, 1); extend_with!(buf, class_index, 2); extend_with!(buf, name_and_type_index, 2); }, Methodref { class_index, name_and_type_index } => { extend_with!(buf, 10u8, 1); extend_with!(buf, class_index, 2); extend_with!(buf, name_and_type_index, 2); } InterfaceMethodref { class_index, name_and_type_index } => { extend_with!(buf , 11u8, 1); extend_with!(buf, class_index, 2); extend_with!(buf, name_and_type_index, 2); }, NameAndType { name_index, descriptor_index } => { extend_with!(buf, 12u8, 1); extend_with!(buf, name_index, 2); extend_with!(buf , descriptor_index, 2); }, MethodHandle { reference_kind, reference_index } => { extend_with!(buf, 15u8, 1); extend_with!(buf, reference_kind, 1); extend_with!(buf, reference_index, 2); }, MethodType { descriptor_index } => { extend_with!(buf, 16u8, 1); extend_with!(buf, descriptor_index, 2); }, Dynamic { bootstrap_method_attr_index, name_and_type_index } => { extend_with!(buf, 17u8, 1); extend_with!(buf, bootstrap_method_attr_index, 2); extend_with!(buf, name_and_type_index, 2); }, InvokeDynamic { bootstrap_method_attr_index, name_and_type_index } => { extend_with!(buf, 18u8, 1); extend_with!(buf, bootstrap_method_attr_index, 2); extend_with!(buf, name_and_type_index, 2); }, Module { name_index } => { extend_with!(buf, 19u8, 1); extend_with!(buf, name_index, 2); }, Package { name_index } => { extend_with!(buf, 20u8, 1); extend_with!(buf, name_index, 2); }, Invalid => {} } } } #[derive(Debug)] struct Field { access_flags: u16, name_index: u16, descriptor_index: u16, attributes: Vec<Attribute>, } impl Field { fn from_parser(parser:& mut Parser, constant_pool:& Vec<ConstantPoolEntry>) -> Result<Self, String> { Ok(Field { access_flags: parser. u2()?, name_index: parser. u2()?, descriptor_index: parser. u2()?, attributes: parse_many!(parser. u2()?, Attribute:: from_parser(parser, constant_pool))? }) } fn dump(& self, buf:& mut Vec<u8>) { extend_with!(buf, self. access_flags, 2); extend_with!(buf, self. name_index, 2); extend_with!(buf, self. descriptor_index, 2); extend_with!(buf, self. attributes. len(), 2); for attr in& self. attributes { attr. dump(buf); } } } #[derive(Debug)] struct Method { access_flags: u16, name_index: u16, descriptor_index: u16, attributes: Vec<Attribute>, } impl Method { fn from_parser(parser:& mut Parser, constant_pool:& Vec<ConstantPoolEntry>) -> Result<Self, String> { Ok(Method { access_flags: parser. u2()?, name_index: parser. u2()?, descriptor_index: parser. u2()?, attributes: parse_many!(parser. u2()?, Attribute:: from_parser(parser, constant_pool))? }) } fn dump(& self, buf:& mut Vec<u8>) { extend_with!(buf, self. access_flags, 2); extend_with!(buf, self. name_index, 2); extend_with!(buf, self. descriptor_index, 2); extend_with!(buf, self. attributes. len(), 2); for attr in& self. attributes { attr. dump(buf); } } } #[derive(Debug)] struct Attribute { attribute_name_index: u16, attribute_length: u32, max_stack: u16, max_locals: u16, code: Vec<u8>, exception_table: Vec<ExceptionTableEntry>, attributes: Vec<Attribute>, }, StackMapTable { attribute_name_index: u16, attribute_length: u32, entries: Vec<StackMapFrame>, }, LocalVariableTable { attribute_name_index: u16, attribute_length: u32, local_variable_table: Vec<LocalVariableTableEntry>, }, Other { attribute_name_index: u16, attribute_length: u32, bytes: Vec<u8> }, } impl Attribute { fn from_parser(parser:& mut Parser, constant_pool:& Vec<ConstantPoolEntry>) -> Result<Self, String> { use Attribute:: *; let attribute_name_index = parser. u2()?; let attribute_length = parser. u4()?; let attribute_name =& constant_pool[attribute_name_index as usize]; if let ConstantPoolEntry:: ValidUtf8 { value } = attribute_name { Ok(match value. as_str() { "Code" => Code { attribute_name_index, attribute_length, max_stack: parser. u2()?, max_locals: parser. u2()?, code: parse_many!(parser. u4()?, parser. u1())?, exception_table: parse_many!(parser. u2()?, ExceptionTableEntry:: from_parser(parser))?, attributes: parse_many!(parser. u2()?, Attribute:: from_parser(parser, constant_pool))?, }, "StackMapTable" => StackMapTable { attribute_name_index, attribute_length, entries: parse_many!(parser. u2()?, StackMapFrame:: from_parser(parser))?, }, "LocalVariableTable" | "LocalVariableTypeTable" => LocalVariableTable { attribute_name_index, attribute_length, local_variable_table: parse_many!(parser. u2()?, LocalVariableTableEntry:: from_parser(parser))?, }, _ => Other { attribute_name_index, attribute_length, bytes: parse_many!(attribute_length, parser. u1())? }, })} else { return Err(format!("Attribute's name must be a utf8 string in the constant pool")); } } } fn dump(& self, buf:& mut Vec<u8>) { match self { Attribute:: Other { attribute_name_index, attribute_length, bytes } => { extend_with!(buf, attribute_name_index, 2); extend_with!(buf, attribute_length, 4); buf. extend(bytes. clone()); }, Attribute:: StackMapTable { attribute_name_index, attribute_length, entries } => { extend_with!(buf, attribute_name_index, 2); extend_with!(buf, attribute_length, 4); extend_with!(buf, entries. len(), 2); for entry in entries { entry. dump(buf); } }, Attribute:: LocalVariableTable { attribute_name_index, attribute_length, local_variable_table } => { extend_with!(buf, attribute_name_index, 2); extend_with!(buf, attribute_length, 4); extend_with!(buf, local_variable_table . len(), 2); for entry in local_variable_table { entry. dump(buf); } }, Attribute:: Code { attribute_name_index, attribute_length, max_stack, max_locals, code, exception_table, attributes } => { extend_with!(buf, attribute_name_index, 2); extend_with!(buf, attribute_length, 4); extend_with!(buf, max_stack, 2); extend_with!(buf, max_locals, 2); extend_with!(buf, code. len(), 4); buf. extend(code. clone()); extend_with!(buf , exception_table. len(), 2); for entry in exception_table { entry. dump(buf); } extend_with!(buf, attributes. len(), 2); for attr in attributes { attr. dump(buf); } }, } } } #[derive(Debug)] struct ExceptionTableEntry { start_pc: u16, end_pc: u16, handler_pc: u16, catch_type: u16, } impl ExceptionTableEntry { fn from_parser(parser:& mut Parser) -> Result<Self, String> { Ok(ExceptionTableEntry { start_pc: parser. u2()?, end_pc: parser. u2()?, handler_pc: parser. u2()?, catch_type: parser. u2()?, }) } fn dump(& self, buf: &mut Vec<u8>) { extend_with!(buf, self. start_pc, 2); extend_with!(buf, self. end_pc, 2); extend_with!(buf, self. handler_pc, 2); extend_with!(buf, self. catch_type, 2); } } #[derive(Clone, Debug)] enum StackMapFrame { SameFrame { offset_delta: u8 }, SameLocals1StackItemFrame { offset_delta: u8, stack: VerificationType }, SameLocals1StackItemFrameExtended { offset_delta: u16, stack: VerificationType }, ChopFrame { frame_type: u8, offset_delta: u16 }, SameFrameExtended { offset_delta: u16 }, AppendFrame { frame_type: u8, offset_delta: u16, locals: Vec<VerificationType> }, FullFrame { offset_delta: u16, locals: Vec<VerificationType>, stack: Vec<VerificationType> } } impl StackMapFrame { fn from_parser(parser:& mut Parser) -> Result<Self, String> { use StackMapFrame:: *; Ok(match parser. u1()? { c if c <= 63 => SameFrame { offset_delta: c }, c if 64 <= c&& c <= 127 => SameLocals1StackItemFrame { offset_delta: c - 64, stack: VerificationType:: from_parser(parser)? }, 247 => SameLocals1StackItemFrameExtended { offset_delta: parser. u2()?, stack: VerificationType:: from_parser(parser)? }, c if 248 <= c&& c <= 250 => ChopFrame { frame_type: c, offset_delta: parser. u2()? }, 251 => SameFrameExtended { offset_delta: parser. u2()? }, c if 252 <= c&& c <= 254 => AppendFrame { frame_type: c, offset_delta: parser. u2()?, locals: parse_many!(c-251, VerificationType:: from_parser(parser))? }, 255 => FullFrame { offset_delta: parser. u2()?, locals: parse_many!(parser. u2()?, VerificationType:: from_parser(parser))?, stack: parse_many!(parser. u2()?, VerificationType:: from_parser(parser))?, }, c => Err(format!("Stack frame of type {c} is not yet defined in the spec"))?, }) } fn dump(& self, buf:& mut Vec<u8>) { use StackMapFrame:: *; match self { SameFrame { offset_delta } => { extend_with!(buf, offset_delta, 1); }, SameLocals1StackItemFrame { offset_delta, stack } => { extend_with!(buf, offset_delta+ 64, 1); stack. dump(buf); }, SameLocals1StackItemFrameExtended { offset_delta, stack } => { extend_with!(buf, 247u8, 1); extend_with!(buf, offset_delta, 2); stack. dump(buf); }, ChopFrame { frame_type, offset_delta } => { extend_with!(buf, frame_type, 1); extend_with!(buf, offset_delta, 2); }, SameFrameExtended { offset_delta } => { extend_with!(buf, 251u8, 1); extend_with!(buf, offset_delta, 2); }, AppendFrame { frame_type, offset_delta, locals } => { extend_with!(buf, frame_type, 1); extend_with!(buf, offset_delta, 2); for local in locals { local. dump(buf); } } FullFrame { offset_delta, locals, stack } => { extend_with!(buf, 255u8, 1); extend_with!(buf, offset_delta, 2); extend_with!(buf, locals. len(), 2); for local in locals { local. dump(buf); } extend_with!(buf, stack. len(), 2); for entry in stack { entry. dump(buf); } } } } fn covering_extra(& self, num_bytes: u16) -> (StackMapFrame, u16) { use StackMapFrame:: *; match self. clone() { SameFrame { offset_delta } => if num_bytes+ (offset_delta as u16) <= 63 { (SameFrame { offset_delta: (num_bytes as u8)+ offset_delta }, 0) } else { (SameFrameExtended { offset_delta: num_bytes+ (offset_delta as u16) }, 2) }, SameLocals1StackItemFrame { offset_delta, stack } => if num_bytes+ (offset_delta as u16) <= 63 { (SameLocals1StackItemFrame { offset_delta: (num_bytes as u8)+ offset_delta, stack }, 0) } else { (SameLocals1StackItemFrameExtended { offset_delta: num_bytes+ (offset_delta as u16), stack }, 2) }, SameLocals1StackItemFrameExtended { offset_delta, stack } => { (SameLocals1StackItemFrameExtended { offset_delta: num_bytes+ offset_delta as u16), stack }, 0) }, ChopFrame { frame_type, offset_delta } => (ChopFrame { frame_type, offset_delta: offset_delta+ num_bytes }, 0), SameFrameExtended { offset_delta } => (SameFrameExtended { offset_delta: offset_delta+ num_bytes }, 0), AppendFrame { frame_type, offset_delta, locals } => (AppendFrame { frame_type, offset_delta: num_bytes+ offset_delta, locals }, 0), FullFrame { offset_delta, locals, stack } => (FullFrame { offset_delta: num_bytes+ offset_delta, locals, stack }, 0) } } } #[derive(Clone, Debug)] enum VerificationType { Basic { tag: u8 }, Object { tag: u8, cpool_index: u16 }, Uninitialized { tag: u8, offset: u16 }, } impl VerificationType { fn from_parser(parser:& mut Parser) -> Result<Self, String> { let tag = parser. u1()?; if tag < 7 { Ok(VerificationType:: Basic { tag }) } else if tag == 7 { Ok(VerificationType:: Object { tag, cpool_index: parser. u2()? }) } else if tag == 8 { Ok(VerificationType:: Uninitialized { tag, offset: parser. u2()? }) } else { Err(format!("Unknown Verification tag {tag}")) } } fn dump(& self, buf: &mut Vec<u8>) { use VerificationType:: *; match self { Basic { tag } => { buf. push(*tag) } VerificationType:: Object { tag, cpool_index } => { buf. push(*tag); extend_with!(buf, cpool_index, 2); } VerificationType:: Uninitialized { tag, offset } => { buf. push(*tag); extend_with!(buf, offset, 2); } } } fn cover_extra(& mut self, num_bytes: u16) { if let VerificationType :: Uninitialized { offset, .. } = self { *offset += num_bytes; } } } #[derive(Clone, Debug)] struct LocalVariableTableEntry { start_pc: u16, length: u16, name_index: u16, descriptor_index: u16, index: u16, } impl LocalVariableTableEntry { fn from_parser(parser:& mut Parser) -> Result<Self, String> { Ok(LocalVariableTableEntry { start_pc: parser. u2()?, length: parser. u2()?, name_index: parser. u2()?, descriptor_index: parser. u2()?, index: parser. u2()?, }) } fn dump(& self, buf:& mut Vec<u8>) { extend_with!(buf, self. start_pc, 2); extend_with!(buf, self. length, 2); extend_with!(buf, self. name_index, 2); extend_with!(buf, self. descriptor_index, 2); extend_with!(buf, self. index, 2); } } fn run(args: Vec<String>) -> Result<(), (i32, String)> { if args. len() < 2 { return Err((1, format!("Usage: {} File. class [severity = 1]", args[0]))) } let class_file = args[1]. clone(); use std:: str:: FromStr; let severity = u8:: from_str(args. get(2). unwrap_or(& format!("1"))). map_err(|err| (5, format!("Could not parse severity: {err}")))?; let name3 = std:: fs:: read(& class_file). map_err(|err| (2, format!("Could not read {class_file}: {err}")))?; let mut class = Class:: from_parser(& mut Parser:: from(bytes)). map_err(|err| (3, format!("Could not parse {class_file}: {err}")))?; class. enable_planned_obsolescence(severity); std:: fs:: write(& class_file,& class. dump()). map_err(|err| (4, format!("Could not write to {class_file}: {err}")))?; Ok(()) } fn main() { if let Err((code, message)) = run(std:: env:: args(). collect:: <Vec< _>>()) { eprintln!("{message}"); std:: process:: exit(code) } }
```

[3] Also available in an electronic format at https://gitlab.com/gzakhour/jgeorge