Software Libraries and Their Reuse: Entropy, Kolmogorov Complexity, and Zipf's Law *

[Extended Abstract]

Todd L. Veldhuizen
Open Systems Laboratory
Indiana University Bloomington
Bloomington, IN, USA
tveldhui@acm.org

ABSTRACT

That software libraries are useful is something of a paradox. Theory tells us almost no programs can be made shorter, and hence reuse of code from libraries — which shortens programs — should not be happening on any appreciable scale; yet it is. The paradox arises from two somewhat contradictory notions of compressibility: that of Kolmogorov complexity, and that of information theory. Their interplay leads to some surprising results and a nice characterization of the role libraries play in reducing program size. We identify a parameter H_0 of a problem domain that measures the diversity of its programs. This parameter yields upper bounds on code reuse and the scale of components with which we may work. This suggests a way to distinguish the (perhaps rare) problem domains in which the CBSE dream of "programming-by-composing-components" may be realized. Along the way we demonstrate a Zipf-style law for the frequencies with which library components are reused, propose that library components are the prime numbers of software, and prove bounds on the amount of code that may be reused from libraries. As a legacy to future generations of library authors, we prove an incompleteness theorem for domain-specific libraries, guaranteeing them continual employment.

General Terms

Software Libraries, Reuse, Entropy, Information Theory, Kolmogorov Complexity, Zipf's Law, Component-Based Software Engineering

1. INTRODUCTION

Software reuse offers the hope that software construction can be made easier by systematic reuse of well-engineered components. In practice reuse has been found to improve productivity and reduce defects [16, 9, 7, 15, 2]. But what of the limits of reuse — will large-scale reuse make software construction easy? Thinking here is varied, but for the sake of argument let me artificially divide the opinions into two competing hypotheses. First the more enthusiastic end of the spectrum, which I associate with the Component-Based Software Engineering (CBSE) movement.

> **Strong reuse hypothesis:** Large-scale reuse will allow mass-production of software, with applications being assembled by composing pre-existing components. The activity of programming will consist primarily of choosing appropriate components from libraries, adapting and connecting them.

Strong reuse is thought to be possible in problem domains in which there is a great concentration of effort and similarity of purpose, i.e., many people writing similar software whose needs show only minor variation. However, the question of whether strong reuse can succeed for software construction considered globally, across disciplines and organizations, remains uncertain. A more cautious view of reuse is the following.

> **Weak reuse hypothesis:** Large-scale reuse will offer important reductions in the effort of implementing software, but these savings will be a fraction of the code required for large projects. Nontrivial projects will always require the creation of substantial quantities of new code that cannot be found in existing component libraries.

Representative of weak reuse thinking is the following prescription for code reuse in well-engineered software [16]: up to 85% of code ought be reused from libraries, with a remaining 15% custom code, written specifically for the application and having little reuse potential.

A compression view of reuse. The view developed in this paper is that the extent to which reuse can happen is an intrinsic property of a problem domain, and that improving the ability of programmers to find, adapt, and deploy components will have only marginal impact on reuse rates if the

*In submission to LCSD 2005.

domain is inimical to reuse. We propose to associate with problem domains a parameter $H_0 \in [0,1]$ closely related to information-theoretic entropy that measures how *diverse* software is within the domain. When $H_0 = 1$, software is extremely diverse and we should expect very little potential for reuse; in fact, we show that the proportion of an application we can draw from libraries approaches zero for large projects. For problem domains with $0 < H_0 < 1$, software is somewhat homogeneous, and with decreasing H_0 comes increasing potential for reuse. The theory we develop suggests that an expected proportion of *at most* $(1 - H_0)$ of an application's code may be reused from domain-specific libraries, with a remaining proportion H_0 being custom code written specifically for the application. As $H_0 \to 0$ we near the strong reuse utopia of "programming by wiring together large components." The possibilities of reuse are *strictly limited* by the parameter H_0, which is an intrinsic property of the problem domain.

We develop this theory through examining our ability to *compress* or *compactify* software by the use of libraries. We shall speak throughout this paper of *compressed programs*, by which we mean programs written using libraries, and *uncompressed programs* that are stand-alone and do not refer to libraries. The principle tools we employ are information theory and Kolmogorov complexity.

Library components and prime numbers. Integers factor into a product of primes; software can be factored into an assembly of components. Library components are the prime numbers of software. This would be a terribly naive thing to say were it not for the many wonderful parallels that turn up:

- There are infinitely many primes; in Section 5.3 we prove there are infinitely many components for a problem domain that reduce expected program size (thus guaranteeing employment for library writers.)
- The n^{th} prime is a factor of $\sim \frac{1}{n \ln n}$ of the integers. We predict the n^{th} most frequently used library component has a reuse rate of about $\frac{1}{n \log n \log^+ n}$ (Section 4.2).
- The Erdös-Kac theorem states that the number of factors of an integer tends to a normal distribution; we measure experimental data that suggests a similar theorem might be provable for software components (Figure 5).
- The Prime Number Theorem states that the n^{th} prime is $\sim \log(n \ln n)$ bits long. We show that a likely configuration for libraries is that the n^{th} most frequently used component is of size $\sim H_0^{-1}(1 - H_0) \log^+ n$. (Section 5.4).

1.1 Reuse and Zipf's law

Useful software abstractions are provided in three basic ways: in hardware, in programming languages, or in libraries. The level at which abstractions are realized is strongly correlated with how frequently they are needed. Integer and floating-point arithmetic are provided in hardware, since almost every computer user runs software requiring these. Some common abstractions that don't merit machine instructions are provided by programming languages: for example, complex numbers and string manipulation are often provided by languages. Less frequently used abstractions are relegated to software libraries, and these constitute the bulk of reusable abstractions. We can think of abstractions as forming an iceberg, with the tip being those abstractions provided in hardware and languages, and the vast majority lurking in libraries (Figure 1). In our view, machine instructions are a special case of "software components."

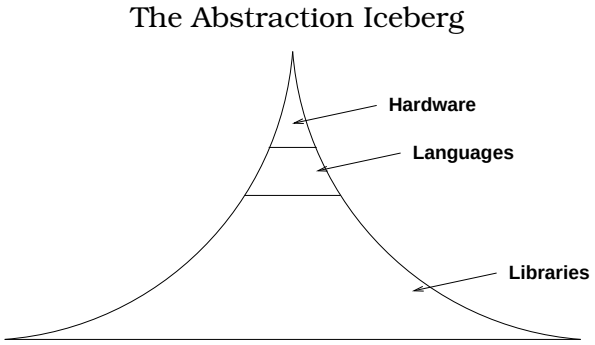


Figure 1: An illustration of where software abstractions are provided. The few abstractions needed very frequently are provided in hardware or by programming languages; the vast majority of abstractions are provided by software libraries.

It is known that hardware instruction frequencies follow an iconic distribution described by George K. Zipf for word use in natural languages [10, 19, 12]. Zipf noted that if you rank words in a natural language according to use frequency, the frequency of the n^{th} word is about n^{-1}. Such distributions are called Zipf's law in his honour [17], and they crop up in many fields. Evidence suggests programming language constructs also follow a Zipf-like law [3, 11].

It is natural then to wonder if this result might extend down the iceberg to library components. Our results support this conclusion. Figure 2 shows the reuse counts of subroutines in shared objects on three Unix platforms, clearly showing Zipf-like n^{-1} curves. These results are described in detail in Section 6.

2. MODELLING LIBRARY REUSE

We propose an abstract model capturing some essential aspects of software reuse within a problem domain. The basic scenario is this: we have a library, possibly many libraries that we collectively consider as one, that contains a great number of software components. These components may be subroutines, architectural skeletons, design patterns, generics, component generators, or whatever form of abstraction we may yet invent; their precise nature is unimportant for the argument. In using a component from the library we achieve some reduction in the size of the program, and perhaps consequently, in the effort required to implement it. Program size serves as a rough lower bound to effort, but it would be a grave error to confuse the two.

2.1 Distribution of programs in a domain

We presume that the projects undertaken by programmers can be modelled by a probability distribution on programs particular to the problem domain. The probability distribu-

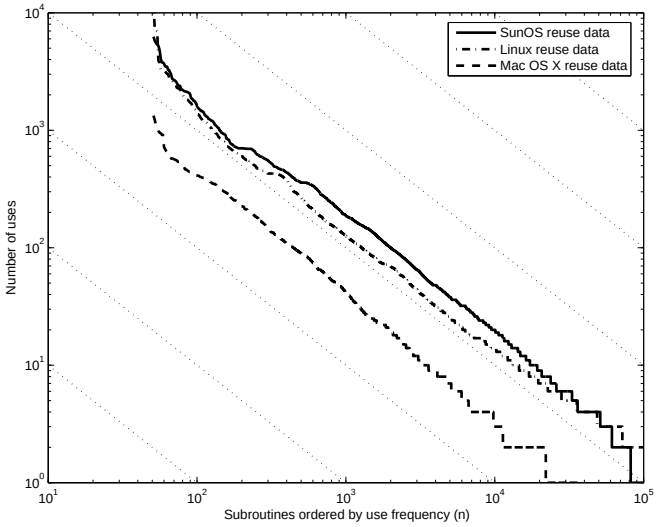


Figure 2: Data collected from shared objects on several unix platforms, showing the number of references to library subroutines. The observed number of references shows good agreement with a Zipf-law frequency laws of the form $c \cdot n^{-1}$ (dotted diagonal lines). A detailed explanation of this data is given in Section 6.

tion is defined on "uncompressed" programs that do not use any library components.

We consider compiled programs modelled by binary strings on the alphabet $\{0, 1\}$. We shall write $\|w\|$ for the length of a string w.[1] To avoid probability distributions on infinite sets of programs in which the probability of encountering individual programs vanishes, we shall work with a family of conditional distributions $\{p_{s_0}\}_{s_0 \in \mathbb{N}}$ whose domains consist of programs $\leq s_0$ bits in size, that is,

$$p_{s_0} : \left(\bigcup_{i \leq s_0} \{0,1\}^i \right) \to \mathbb{R}$$

and satisfying $\sum p_{s_0} = 1$ and $p_{s_0}(w) \geq 0$ for all w and s_0. Then $p_{s_0}(w)$ gives the probability that someone working in the problem domain will set out to realize the particular (uncompressed) program w, given $\|w\| \leq s_0$. For this family of distributions to be compatible with one another we shall require that $p_{s_0}(w) = p_{s_0+k}(w \mid \|w\| \leq s_0)$ for $k \in \mathbb{N}$.

In the sequel we will use the usual notation for expectation, with the implied assumption of $s_0 \to \infty$:

$$E\left[f(w)\right] \equiv \lim_{s_0 \to \infty} \sum_{w : \|w\| \leq s_0} f(w) p_{s_0}(w)$$

For example, a mean program size $E[\|w\|]$ may exist for a problem domain, but we do not require this.

We do not presume that such distributions can be effectively defined for a specific problem domain.

[1]The use of $|w|$ for string length, while traditional in some quarters, leads to confusing notations such as $P(w \mid |w| < m)$ for probability conditioned on string length. We use $\|\cdot\|$ to avoid this.

2.2 The diversity parameter H_0

We introduce a parameter H_0 for problem domains measuring how far their probability distribution departs from uniform. This is largely the same as entropy rate from information theory, slightly adapted for finite program sizes. When $H_0 = 1$ the distribution is uniform, modelling extreme diversity of software, with little opportunity for reuse. For $H_0 < 1$ there is some potential for reuse. In fact as we shall see shortly, we may expect that up to a proportion $1 - H_0$ of programs may be reused from libraries.

A key, perhaps defining, feature of a problem domain is that there is similarity of purpose in the programs people write. We would not expect the distribution of programs written in a problem domain to be uniform, but rather concentrated on programs that solve certain classes of problems that are likely to crop up in that domain. We measure the departure from uniformity by examining the entropy of the distributions $\{p_{s_0}\}$. Define the entropy of the distribution p_{s_0} in the standard way (see, e.g., [13, §1.11]):

$$H_{s_0} = \sum_{w : \|w\| \leq s_0} -p_{s_0}(w) \log_2 p_{s_0}(w)$$

This is the expected number of bits required to represent a program of size $\leq s_0$ in this domain.

We define H_0 to be the least value such that

$$H_{s_0} \leq H_0 s_0$$

almost surely as $s_0 \to \infty$.[2] This bounds the 'entropy rate' for uncompressed programs in the problem domain.

We cannot hope to calculate H_0 from first principles when the problem domain is defined by the interests of a group of people, but there is hope we might estimate it empirically. We introduce H_0 as a theoretical tool to model problem domains in which people have great similarity of purpose ($H_0 \to 0$) or diffuse interests ($H_0 \to 1$). The main impact of H_0 is the following.

Claim 2.1. *In a problem domain with diversity parameter H_0, the expected proportion of code that may be reused from a library is at most $1 - H_0$.*

This is a consequence of the Noiseless Coding Theorem of information theory (e.g., [1, §2.5]), which states that coding random data with entropy H requires (on average) at least H bits. In our situation, the data we wish to represent are the (uncompressed) programs that programmers set out to realize, and the 'codes' are the (compressed) programs written with use of a library. Suppose an uncompressed program

[2]We do not assume the limit $\lim_{s_0 \to \infty} \frac{H_{s_0}}{s_0}$ exists; there might be unbounded oscillations in H_{s_0}. That a least H_0 exists follows from all such H_0 being bounded below by 0, since $H_{s_0} \geq 0$; and there is at least one such bound, namely $H_{s_0} \leq (1)s_0$; therefore a least upper bound exists. This definition of H_0 differs slightly from the usual definition of entropy rate from information theory, in which the entropy rate of infinite streams or random processes are considered. But the two can be considered equivalent for most purposes as $s_0 \to \infty$.

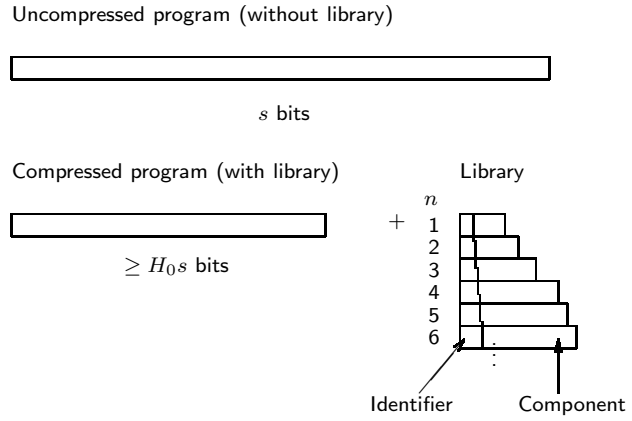


Figure 3: The basic scenario: programmers in a problem domain set out to realize a program that can be represented in s bits when compiled without the use of a library. By using library components, they are able to reduce the size of the compiled program, down to an expected size of $\geq H_0 s$ bits.

has size $s \leq s_0$. We defined H_0 so that $H_{s_0} \leq sH_0$ almost surely, so we can compress programs to an expected size of at best sH_0 by the Noiseless Coding Theorem. Therefore the expected amount of code saved by use of the library is at most $(1-H_0)s$, and it is reasonable to equate this with the amount of code reused from the library. An immediate implication is that blanket reuse prescriptions such as "effective organizations reuse 70% of their code from libraries" are unrealistic; reuse goals need to be tailored to domain's value of H_0.

Figure 3 illustrates the basic scenario we consider in this paper: we are given an uncompressed program of length s written without use of a library, drawn from the distribution for the problem domain. A programmer implements the program making use of the library, effectively "compressing" it. The expected size of the compressed program is at least $H_0 s$ bits, by the previous arguments. The library consists of a set of components, each with an identifier or *code* by which they are referred to. We always take programs to be *compiled*, so as not to care about the high compressibility of source representations.

2.3 Libraries maximize entropy

> A truly great computer programmer is lazy, impatient and full of hubris. Laziness drives one to work very hard to avoid future work for a future self. — Larry Wall

Programmers, so we read, are lazy— they write libraries to capture commonly occurring abstractions so they do not have to write them over and over again. The social processes that drive programmers to develop libraries have an interesting mathematical effect. We can view programmers contributing to domain-specific libraries as collectively defining a system for *compressing programs* in that domain. If there is a common pattern, eventually someone will identify it and put it in a library. Since the absence of "factorable patterns" in code is indicated by high entropy, we propose the following principle.

Principle 1 (Entropy maximization). Programmers develop domain-specific libraries that minimize the amount of frequently rewritten code for the problem domain. This tends to maximize the entropy of compiled programs that use libraries.

As evidence for this principle, we show in Section 6 that the rate at which library components are reused is empirically observed to approach a maximum entropy configuration.[3]

In practice, of course, programmers have to strike a balance between the succinctness of their programs and their readability; see, e.g., [6] for an elegant dissection of such tradeoffs. However, we maintain that the drive toward terseness of programs is a defining pressure on library development: programmers edge as close to maximum entropy as they can while maintaining source-code understandability.[4]

2.4 The Platonic library

In the early days of computing libraries held a hundred subroutines at most; these days it is common for computers to have a hundred thousand subroutines available for reuse (cf. Section 6). Let us suppose that as time goes on we shall continue to add components to our libraries, as we discover more abstractions and algorithms. Our current libraries might be viewed as a truncated version of some infinite (but countable) library toward which we are slowly converging. It is convenient to pretend that this limit already exists as some infinite "Platonic library" for the problem domain, and that we are discovering ever-larger fragments of it, recalling Erdös' book of divine mathematical proofs. Were we granted access to the entire library, we might write software in a very efficient way. We use the Platonic library as a device — a convenient fiction — to reason about how useful finite libraries might be.

Infinite objects need to be treated with care. We shall not assume that some "optimal infinite library" exists that is the best possible such library. Nor shall we assume there is some finite description or computable enumeration of its contents. We merely assume that fragments of the Platonic library give us snapshots of what shall be in our software libraries over time.

2.5 Existence of reuse rates

Numerous metrics have been proposed for measuring reuse. We focus on the *reuse rate* of a component, which we write $\lambda(n)$ and define as the expected rate at which references are made to the n^{th} library component in a compressed program. The units of $\lambda(n)$ are expected references per bit of compiled code. We assume mean reuse rates exist in a problem domain, in the following sense.

Assumption 1. *Let $\mathrm{Refs}_n(w)$ count the number of references to the n^{th} component in a compressed program w of size $\leq s_0$. We assume that*

$$E[\mathrm{Refs}_n(w)] \;\sim\; \lambda(n)\|w\| + o(\|w\|) \qquad (1)$$

[3]Note that Principle 1 is *not* intended to appeal to the maximum entropy principle as advocated by Jaynes, which deals with uncertainty in inference.

[4]We re-emphasize that we are speaking of the entropy rate of *compiled* programs; source representations remain highly compressible to support readability.

where $o(\|w\|)$ denotes some error term growing asymptotically slower than $\|w\|$.

We unfortunately do not have a good sense of how to go from the problem domain's distributions p_{s_0} on *uncompressed* programs to rates of components in *compressed* programs; with humans involved it is tricky to pin down. Hence the need to *assume* that the mean rates $\lambda(n)$ exist. Thankfully this is not a demanding assumption; many sensible random process models would imply Assumption 1, for example modelling component uses as a renewal process (see, e.g., [18, §3]).[5]

2.6 Ordering of library components

For convenience we shall suppose the library components are arranged in decreasing order of expected reuse rate in the problem domain: that is,

$$\lambda(n) \geq \lambda(n+1)$$

There are two reasons for this. The first is tidiness, so that when we plot $\lambda(n)$ vs n we see a monotone function and not noise. The asymptotic bounds we derive on $\lambda(n)$ do not rely on this ordering. The second reason is that to compress programs well, we need to assign shorter identifiers to more frequently used components. This is easiest to reason about if the Platonic library is sorted by use frequency.[6]

3. BACKGROUND

3.1 Kolmogorov Complexity

Kolmogorov complexity, also known as Algorithmic Information Theory, was founded in the 1960s by R. Solomonoff, G. Chaitin, and A.N. Kolmogorov. We shall only make use of some basic facts; for a more thorough introduction the survey article [14] or the book [13] are recommended. The central idea is simple: measure the 'complexity' of an object by the length of the least program that generates it. This generalizes to the study of *description systems*, that is, systems by which we define or describe objects, of which programming languages and logics are prominent examples. The source code of a program, for example, describes a program behaviour; a set of axioms describes a class of mathematical structures. In the general case we have some objects we wish to describe, and a description system ϕ that maps from a description w (for us, a program) to objects. The usual situation is to describe an object by exhibiting a program that generates it; in this case we may also provide some inputs to the program, which we shall call *parameters*. The Kolmogorov complexity of an object x in the description system ϕ, relative to a parameter y is defined by:

$$C_\phi(x \mid y) = \min_w \{\|w\| : \phi_w(y) = x\} \qquad (2)$$

In Eqn. (2), w ranges over all possible programs. In the case where the description system ϕ is a programming language, we may read Eqn. (2) as finding the shortest program that, given input parameter y, outputs x. The parameter y does not contribute to the measured description length $C_\phi(x \mid y)$. Without a parameter we have the simpler case $C_\phi(x) = C_\phi(x \mid \epsilon)$ where ϵ is the empty string.

For example, we might choose the programming language Java as our description system; then for some string x, its Kolmogorov complexity $C_{\mathsf{Java}}(x)$ is the length of the shortest program that outputs x. To determine whether use of a library L offers a reduction in program size, we can consider the combination of Java and the library L as a description system itself which we might call $\mathsf{Java} + \mathsf{L}$, and compare $C_{\mathsf{Java}+\mathsf{L}}(x)$ to $C_{\mathsf{Java}}(x)$.

Fact 3.1 (Invariance [13, §2.1]). *There exists a universal machine U such that if ϕ is some effective description system (e.g., a programming language) then there is a constant c such that $C_U(x) < C_\phi(x) + c$ for any x.*

That is, the machine U is optimal up to a constant factor. For this reason the subscript U can be dropped and one can write $C(x)$ for the Kolmogorov complexity of x, knowing it is only defined up to some constant factor.[7]

This is a suitable time to break the news that we shall be juggling two somewhat contradictory notions of compressibility. The information theory notion deals with compressing objects by assigning short descriptions to objects that appear frequently. The Kolmogorov notion of compressibility describes our ability to find a short description of a single object in isolation, without appealing to any notion of frequency. Some strings have very short descriptions: a string of a trillion zeros may be produced by a short program. Others require descriptions as long as the strings themselves, for instance a million digit binary string obtained from a physical random bit generation device.[8] A recurrent theme in Kolmogorov complexity is that there are never enough descriptions to go around so as to give short descriptions to most objects. In the case where both the objects and their descriptions are binary strings, we have the following well-known result that the probability we can save more than a constant number of bits in compressing randomly selected strings is zero.

Fact 3.2 (Incompressibility [13, §2.2]). *Suppose g is a positive integer function with $g \in \omega(1)$, that is, $\lim_{n\to\infty} g(n) = \infty$. Let x be a string chosen uniformly at random. Then almost surely:*

$$C_\phi(x) \geq \|x\| - g(\|x\|) \qquad (3)$$

Theorem 3.2 implies, for example, that one cannot devise a coding system that compresses strings by even $\log\log n$ or $\alpha^{-1}(n,n)$ (inverse Ackermann) bits with nonzero probability. The proof of Theorem 3.2 uses counting arguments only, with

[5] For readers familiar with coding theory we forestall confusion by mentioning that the rates $\lambda(n)$ are not the same as the usual notion of probabilities over countable alphabets. The rates $\lambda(n)$ are drawn from compressed programs and so already incorporate code lengths.

[6] Jeremiah Willcock made the useful suggestion that we may regard the Platonic library as containing already every possible component, and the only question is the order in which they are placed.

[7] There is an easy way to see why this is true: if ϕ is a programming language, then we can write a ϕ-interpreter for the universal machine U. We can then take any program for ϕ, prepend the interpreter, and it becomes a U-program. The constant mentioned reflects the size of such an interpreter.

[8] Unless you are lucky.

no appeal to effective computability of the description system.[9] Therefore the inequality (3) applies to *any* description system ϕ, even description systems that are not computable. For example, Fact 3.2 even applies if we permit ourselves to use an infinite, not computably enumerable library as we described in Section 2.4.[10]

In the remainder of this paper we shall assume compiled programs are almost surely Kolmogorov incompressible. To connect this assumption with reality, we make the following claim.

Claim 3.1. *Compiled C programs on most extant architectures are almost surely Kolmogorov incompressible.*

To bolster Claim 3.1, we show that the number of distinct behaviours described by compiled programs of s bits grows as $\sim 2^s$ on current machines, which implies compiled programs are almost surely (Kolmogorov) incompressible. The C language has the useful ability to incorporate chunks of binary data in a program. For example, the binary string $z = 0110100111011010$ may be encoded by the C declaration

$$\mathsf{unsigned\ char\ z}[2] = \{\mathsf{0x69}, \mathsf{0xda}\};$$

Moreover, such arrays are laid out as contiguous binary data in the compiled program, so that a binary string of length m bytes requires exactly m bytes in the compiled program. Now we can combine such arrays with a short program of constant size that that reads the binary string from memory and outputs it to the console. Every binary string of m bytes may be encoded by such a compiled program of size at most $c + m$ bytes, where c is a constant representing the overhead of a read-print loop. Every such program yields a unique behaviour, so the number of distinct behaviours of compiled programs of s bits is $\sim 2^s$. We can then adapt the argument used to prove Fact 3.2, replacing strings by compiled programs, which shows compiled C programs are almost surely incompressible. [11]

Note that uncompiled programs are *highly* compressible. For example, C language source code may not contain certain bytes (e.g., control characters) such as the null character $0x00$. This means they can be compressed by a factor of (at least) $\frac{1}{256} \sim 0.39\%$. Restricting our attention to *compiled* programs is crucial.

4. A BOUND ON REUSE RATES

In this section we derive a bound on the reuse rate $\lambda(n)$ at which the n^{th} library component may be of use.

[9]There are $2^{n-g(n)+1} - 1$ descriptions of length at most $n - g(n)$, and $2^{n+1} - 1$ strings of length at most n. Therefore the fraction of strings compressible by $g(n)$ bits is at most $\frac{2^{n-g(n)+1}-1}{2^{n+1}-1}$, which behaves in the limit as $2^{-g(n)}$. If $g \in w(1)$ this value vanishes as $n \to \infty$, so $C_\phi(x) \geq \|x\| - g(\|x\|)$ almost surely.

[10]This would be terrible news for software reuse were it the final word. However, there is more to this story yet to come.

[11]Note that "almost surely incompressible" does not imply anything about the compressibility of *typical* compiled programs one finds on a real computer. Rather, it means that if one chooses a valid compiled program uniformly at random, with probability 1 it cannot be replaced by a shorter program with the same behaviour. In subsequent sections we investigate problem domains where there *is* a nonuniform distribution on programs, i.e., $H_0 < 1$, where the situation is rosier.

4.1 Coding of references

We need some rudimentary accounting of what we gain and lose by use of the library: we save some by using a library component, at the cost of having to refer to it. Let us first consider the cost of referring to components.

We presume that codes (i.e., identifiers) are assigned to library components so that every component has a unique code. Let $c(n)$ be the binary code for the n^{th} library component, and $\|c(n)\|$ its length. Optimal strategies such as Shannon-Fano or Huffman codes assign shortest codes to the most frequently needed components. Since our library is sorted in order of use frequency (Section 2.6), we may presume that $\|c(n)\| \leq \|c(n+1)\|$, i.e., code lengths are nondecreasing as we go down the list of components.

Now in what follows we want to make asymptotic arguments, and fixing an identifier size (e.g., 64 bits) would lead to wildly wrong conclusions.[12] Instead we require that the identifier size grows with the number of components, albeit slowly. It is easy to prove that $\|c(n)\| \geq \log_2 n$. Having identifiers of length only $\log_2 n$ leads to difficulties, because they are not *uniquely decodable*. That is, if I am presented with a string of such identifiers I have no way to tell where one identifier stops and the next starts. (This does not arise in current architectures because of fixed word size, but as we said, care is needed in asymptotic arguments). A more accurate requirement is the following, which draws on Kraft's inequality that uniquely decodable codes must satisfy $\sum_{n=1}^{\infty} 2^{-\|c(n)\|} \leq 1$.

Proposition 4.1. *For identifiers to be uniquely decodable,*

$$\|c(n)\| \geq \log^+ n$$

where $\log^+ n = \log n + \log\log n + \log\log\log n + \cdots$ *and the sum is taken only over the positive real terms.*

We omit the proof; see e.g., [13, §1.11.2] (in particular problem 1.11.13).

4.2 Derivation of reuse rate bound

We now derive an asymptotic upper bound on the rates $\lambda(n)$ at which library components may be reused. We do this under the assumption that each time a library component is used in a program, the same identifier is used to refer to it, i.e., there is no *recoding of identifiers*.[13] Our argument follows standard lines [17] but adapted to coding of library references under the model laid out in Section 2.

[12]For instance, the time required to search a linked list of n elements is $O(n)$. But if we fix memory addresses to be representable in 64 bits, then the time is $O(1)$ since there are at most 2^{64} steps the algorithm must go through.

[13]There are two reasons for this assumption. (1) On the architectures from which we collect empirical data, there is no recoding of identifiers in programs. (2) The reason one might want to recode identifiers is to save space by introducing shorter aliases for components for use within the program, after the initial reference. However, this only saves space if a component is more likely to be used again given it is used once. While this is intuitively true of real programs, it is false under a maximum entropy assumption (Section 2.3): in an encoding that maximizes entropy, the sequence of identifiers in a program behaves statistically as if independent and identically distributed.

Theorem 4.1. *Without recoding of identifiers, the asymptotic reuse rates $\lambda(n)$ must satisfy $\lambda(n) \prec (n \log n \log^+ n)^{-1}$.*

Proof. We count the size of the references to library components within compressed programs (i.e., those written with use of a library). Consider programs of length at most s. As $s \to \infty$, the expected number of occurrences of the n^{th} component tends to $\lambda(n)s + o(s)$ under Assumption 1. Referring to the n^{th} component requires at least $\log^+ n$ bits (Proposition 4.1). We need only consider components whose identifier length is less than s, since identifiers longer than the program would not fit. Therefore we consider only up to component number 2^s since $\log^+ 2^s \geq s$.

The expected total size of all the references to components is then at least:

$$\sum_{n=1}^{2^s} \underbrace{(\lambda(n)s + o(s))}_{\text{\# refs}} \underbrace{\log^+ n}_{\text{ref size}}$$

The references to components are contained within the program, and therefore their total size must be less than s, the size of the program. Therefore we have an inequality:[14]

$$\sum_{n=1}^{2^s} (\lambda(n)s + o(s)) \log^+ n \leq s \tag{4}$$

Dividing through by s and taking the limit as $s \to \infty$,

$$\lim_{s\to\infty} \sum_{n=1}^{2^s} \frac{1}{s}(\lambda(n)s + o(s)) \log^+ n \leq 1 \tag{5}$$

Since $\lim_{s\to\infty} \frac{1}{s} o(s) = 0$ by definition,[15]

$$\sum_{n=1}^{\infty} \lambda(n) \log^+ n \leq 1 \tag{6}$$

We now consider conditions under which this sum converges. (Section A.1 summarizes the asymptotic notations used here.) We argue using Proposition A.1, using a diverging series to bound the terms of Eqn. (6). The simple argument is to note that the harmonic series diverges, and therefore the terms of Eqn. (6) must grow slower than this, so $\lambda(n) \log^+ n \prec \frac{1}{n}$, or $\lambda(n) \prec \frac{1}{n \log^+ n}$. However, this bound is quite loose. A more slowly diverging series is $\sum_n \frac{1}{n \log n}$. Using this,

$$\lambda(n) \log^+ n \prec \frac{1}{n \log n}$$

or,

$$\boxed{\lambda(n) \prec \frac{1}{n \log n \log^+ n}} \tag{7}$$

This completes the proof. □

The bound of Theorem 4.1 is not tight. No tightest bound is possible since there is no slowest diverging sequence with which to bound a convergent sequence, a classical result due to Niels Abel. However, the bound is tight to within a factor n^ϵ for any $\epsilon > 0$.

This provides an upper bound on $\lambda(n)$, but it could well be the case that $\lambda(n) \sim \frac{1}{n^3}$, for example. Why do the curves we see in practice hug the bound of Theorem 4.1?

Entropy maximization. The answer to why we see $\lambda(n) \approx \frac{1}{n}$ in practice appears to be due to the tendency of libraries to evolve so that programmers can write as little code as possible, which in turn implies evolution toward maximum entropy in compiled code (Principle 1). It turns out that the entropy of the component references is maximized when $\lambda(n) \approx \frac{1}{n}$ (see, e.g., [8]).

A maximum-entropy explanation for Zipf's law in natural languages has been advocated by Harremoës and Topsøe [8]. They suggest that vocabulary learning be modelled by convergence to a Zipf-like distribution and gradually increasing communication bit rate, a possibly interesting model both for library development within a problem domain, and for library learning progressions.

5. REUSE POTENTIAL

In the following sections we consider the possibilities of code reuse in two cases: (1) when $H_0 = 1$ and we have a uniform distribution on programs; (2) when $0 < H_0 < 1$ and we have some degree of compressibility in the problem domain. We do not look at the case $H_0 = 0$ due to space limitations.

5.1 The uniform case: $H_0 = 1$

The uniform case of $H_0 = 1$, in which every program is equally likely to be implemented, reduces our scenario to the classical situation of Kolmogorov complexity. It has some surprising properties that suggest $H_0 = 1$ to be an unlikely scenario for real problem domains.

Our first result concerns the number of library components we might expect to use in a program. Let $N^{(s)}$ be a random variable indicating for a program of uncompressed size s the number of components whose use reduces program size. Surprisingly, as program size increases the expected number of components that reduce program size is bounded above by a constant.

Theorem 5.1. *If $H_0 = 1$ there exists a constant n_{crit} independent of program size s such that $N(s) \leq n_{\text{crit}}$ almost surely.*

Proof. Suppose each component used saved at least 1 bit. If $\lim_{s\to\infty} E[N(s)]$ were unbounded, use of the library could compress random programs by an unbounded amount, contradicting incompressibility (Fact 3.2). □

This has a simple corollary concerning the potential for code reuse.

[14] Inequality (4) becomes an equation if we consider programs to consist solely of a sequence of component references, with no control flow or other distractions. This is possible by building components and programs from combinators, which can be made self-delimiting [13, §3.2]. This provides a theoretically elegant framework, if not entirely intuitive. The Invariance Theorem of Kolmogorov complexity ensures that translating programs and libraries into combinatory form is not costly in program size (Fact 3.1).

[15] Recall that $f \in o(g)$ means $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$.

Corollary 5.1. *When $H_0 = 1$ the expected proportion of a program that can be reused from libraries tends to zero as program size increases.*

These results smack of paradox: a primary purpose of software libraries is to save code, i.e., *compress programs*, but Kolmogorov complexity tells us almost every program is incompressible. Is reuse a pipe-dream? No. In the next sections we show that if $H_0 < 1$ then we can compress programs, even ones that are (Kolmogorov) incompressible, by use of a library. Resolving this paradox requires distinguishing between the Kolmogorov notion of a *single program being incompressible*, versus the information theory notion of compressibility that involves a probability distribution over an *ensemble* of programs.

5.2 Libraries compress the incompressible

Much more interesting than the uniform case is the situation when $0 < H_0 < 1$. This models problem domains that have some potential for code reuse, and libraries are of central importance. In such problem domains libraries let us compress[16] the incompressible.[17] Recall from Section 2.2 that we can expect to compress programs in such domains from uncompressed size s to at best $H_0 s$ by use of a library. A standard result from information theory can be adapted to show this bound is achievable, at least in a theoretical sense.

Claim 5.1. *There exists a library with which uncompressed programs of size s can be compressed to expected size $\sim H_0 s$.*

The proof of this is not particularly illustrative and we banish it to a footnote.[18] The gist is to stuff every possible program into the library, but order them so that the most likely programs for the problem domain come soonest in the library order and thus are assigned the shortest codes. This is a wildly impractical construction but demonstrates the claim. In practice we decompose software into reusable chunks that we put in libraries.

[16] In the information theory sense.

[17] In the Kolmogorov complexity sense.

[18]

Proof. We first describe an encoding that compresses programs to achieve an expected size $H_0 s$, and then explain how to construct the library. Recall the Shannon-Fano code [13, §1.11] allows a finite distribution with entropy H to be coded so that the expected code length is $\leq H + 1$. We adapt this as follows. For each $s_0 \in \mathbb{N}$, we produce a Shannon-Fano codebook for all programs of length $\leq s_0$ achieving average code size $\leq H_{s_0} + 1$ for the distribution p_{s_0} (Section 2.2). By definition $H_{s_0} \leq H_0 s$ almost surely, so this achieves a compression ratio of H_0 almost surely for each s_0 as $s_0 \to \infty$. To combine all the codebooks into one, we preface a compressed program with an encoding of its uncompressed length, which we use to select the appropriate codebook. This can be done by adding to each code $c + 2 \log s$ bits for some constant c, which is negligible with respect to $H_0 s$ when $H_0 > 0$. Therefore this encoding achieves expected program size $\sim H_0 s$. We use the codebook as the library: each component identifier is a Shannon-Fano code, each component is a program. Note that the reuse rates vanish for this construction, i.e., $\lambda(n) \to 0$ as $s_0 \to \infty$, and so the bound of Theorem 4.1 is trivially satisfied. □

Unlike the situation of $H_0 = 1$ where the number of components useful for a program was at most a constant, when $0 < H_0 < 1$ we have a much more pleasing situation where the number of useful components increases steadily as we increase program size. In the next few sections we explore some further interesting properties.

5.3 The incompleteness of libraries

Under reasonable assumptions we prove that no finite library can be complete: there are always more components we can add to the library that will allow us increase reuse and make programs shorter. To make this work we need to settle a subtle interplay between the Kolmogorov complexity notion of compressibility (there is a shorter program doing the same thing) and the information theoretic notion of compressibility (low entropy over an *ensemble* of programs). Now because we defined probability distributions on programs (rather than behaviours), we run into the possibility that the probability distribution might weight heavily programs that are *Kolmogorov* compressible, i.e., the distribution might prefer programs w with $\|w\| >> C(w)$. For example, a problem domain might have programs that are usually compressible to half their size not because the probability distribution focuses on a particular class of problems, but because we artificially defined p_{s_0} to select only those programs that are twice as large as they might be (for example, we might pad every likely program with lots of `nop` instructions.) To avoid this difficulty we require the distributions be *honest* in the following sense.

Definition 1 (Honesty). We say the distributions p_{s_0} for a problem domain are *honest* if the programs are Kolmogorov incompressible. Specifically,

$$E\left[\frac{C(w)}{\|w\|}\right] \to 1 \quad \text{as } s_0 \to \infty \tag{8}$$

where the expectation is taken over the distributions p_{s_0}. This requires that the probability distribution does not artificially prefer verbose programs with $\|w\| >> C(w)$.

If the distribution for a problem domain is honest and has $H_0 < 1$, the programs are expected to be *information-theoretically compressible* by use of a library, but not *Kolmogorov compressible*. In other words, our ability to compress programs is due to a "focus" on a class of problems of interest to the domain, not just an artificial selection of overly-verbose programs.

Euclid proved there are infinitely many primes; with the honesty assumption we can prove there are infinitely many *reusable* software components that make programs shorter.

First we need two smaller pieces of the puzzle.

Lemma 5.1. *If $H_0 > 0$ then for any finite k, $Pr(\|w\| \leq k) \to 0$ as $s_0 \to \infty$.*

Proof. We know from definition of H_0 that $H_{s_0} = H_0 s_0$ infinitely often as $s_0 \to \infty$ (Section 2.2). Consider how probability must be distributed among programs of different lengths to account for this much entropy. We try to account for as much entropy as we can by short programs, setting a uniform distribution $p(w) = \frac{1}{2^{H_0 s_0}}$ on the first $2^{H_0 s_0}$ programs— this

is the fewest number of programs that would produce this much entropy. To programs of length $\leq k$ we can account for

$$\sum_{i=0}^{k} 2^i \cdot \left(-\frac{1}{2^{H_0 s_0}} \log \frac{1}{2^{H_0 s_0}} \right) \sim 2^{k+1-H_0 s_0} H_0 s_0$$

bits of entropy. But as $s_0 \to \infty$, $2^{k+1-H_0 s_0} H_0 s_0 \to 0$ so we can account for none of the entropy by programs of length $\leq k$. Therefore $Pr(\|w\| \leq k) \to 0$ as $s_0 \to \infty$. □

Lemma 5.2. *If $H_0 > 0$ then $E\left[\frac{1}{\|w\|}\right] \to 0$ as $s_0 \to \infty$.*

Proof. Suppose $E\left[\frac{1}{\|w\|}\right] = c$ for some $c > 0$. Then there would be a finite probability that $\|w\| \leq c^{-1}$ as $s_0 \to \infty$, contradicting Lemma 5.1. □

Now we are ready for the main theorem, which proves no finite library can be "complete" in the sense of achieving a compression ratio of H_0 when $0 < H_0 < 1$.

Theorem 5.2 (Library Incompleteness). *If a problem domain has $0 < H_0 < 1$ and honest distributions (Defn. 1), no finite library can achieve a compression ratio of H_0.*

Proof. (By contradiction). Suppose a finite library of components achieves a compression factor H_0. Call the programming language ϕ and the library L. We can write an interpreter for ϕ that incorporates the library L; since the library is finite this is a finite program. We call the resulting machine model $\phi + L$. Consider Kolmogorov complexity for this machine, writing $C_{\phi+L}(w)$ for the size of the smallest ϕ-program using L that has the same behaviour as w. Saying the machine $\phi + L$ achieves the compression factor H_0 implies

$$E\left[\frac{C_{\phi+L}(w)}{\|w\|}\right] = H_0 \tag{9}$$

From the invariance theorem of Kolmogorov complexity (Fact 3.1) we have that there exists a constant c such that

$$C(w) \leq C_{\phi+L}(w) + c \tag{10}$$

for every program w. Dividing through by $\|w\|$ and taking expectation,

$$E\left[\frac{C(w)}{\|w\|}\right] \leq \underbrace{E\left[\frac{C_{\phi+L}(w)}{\|w\|}\right]}_{=H_0} + E\left[\frac{c}{\|w\|}\right] \tag{11}$$

From honesty $E\left[\frac{C(w)}{\|w\|}\right] \to 1$, and from Lemma 5.2 we have $E\left[\frac{c}{\|w\|}\right] \to 0$. Therefore (11) is:

$$1 \leq H_0 + 0$$

but this contradicts $H_0 < 1$. □

Claim 5.1 showed that an infinite library can achieve expected size $\sim H_0 s$; Theorem 5.2 shows that no finite library can. Therefore only infinite libraries can compress programs of size s to expected size $H_0 s$. With finite libraries we can get arbitrarily close to achieving $H_0 s$ by including more and more components. Doug Gregor suggested calling Theorem 5.2 the *Full Employment Theorem for Library Writers*, after Andrew Appel's boon to compiler writers. Theorem 5.2 has a straightforward implication: no finite library can be complete; there are always more useful components to add. This suggests the importance of designing libraries for extensibility.

A minor change to the above proof yields a similar but slightly stronger result.

Corollary 5.2. *If a problem domain has $0 < H_0 < 1$ and honest distributions, no computably enumerable library can achieve a compression ratio of H_0.*

Proof. Repeat the proof of Theorem 5.2, replacing "finite library" with "c.e. library." In particular the choice of a c.e. library guarantees that the interpreter for $\phi + L$ is a finite program: whenever a library subroutine is required, it is generated from the program enumerating the library. □

I casually equate "not computably enumerable" with "requires human creativity." Corollary 5.2 indicates that the process of discovering new and useful library components is not a process that can can be fully automated.

5.4 Size of library components.

We now consider how big library components might be. If we want to support "programming by wiring together components," this suggests that components ought to be quite large compared to the wiring. The following theorem sheds light on the conditions when this is possible.

Let $S(n)$ denote the expected amount of code (in bits) saved by each use of the n^{th} component. We consider the case when $\lambda(n) \sim \frac{1}{n\|c(n)\|f(n)}$, where $\|c(n)\|$ is the codeword (identifier) length, and $f(n)$ is a function $f \in o(n^\epsilon)$ for $\epsilon > 0$ that ensures convergence (cf. Section 4.2). This coincides with a Zipf-style law as observed in practice (Figure 2).

Theorem 5.3. *If a library achieves a compression factor of $H_0 > 0$ in an honest problem domain, then $S(n) \sim \frac{1-H_0}{H_0} \cdot \|c(n)\| \cdot o(f(n))$.*

Proof. Summing over all components, the total code saved is:

$$\sum_{n=1}^{\infty} \underbrace{(\lambda(n) H_0 s + o(s))}_{\text{expected \# uses}} \cdot \underbrace{S(n)}_{\text{savings per use}} = \underbrace{(1 - H_0)s}_{\text{total savings}} \tag{12}$$

Dividing through by $H_0 s$ and taking the limit as $s \to \infty$, and substituting $\lambda(n) \sim \frac{1}{n\|c(n)\|f(n)}$,

$$\sum_{n=1}^{\infty} \frac{1}{n\|c(n)\|f(n)} S(n) = \frac{1 - H_0}{H_0}$$

Now if $S(n) \sim n^a$ for $a > 0$ then the sum would diverge. Therefore $S(n)$ is not polynomial in n; in fact for the sum to converge we must have $S(n) \prec f(n)$ which means $S(n)$ behaves asymptotically as

$$S(n) \sim \|c(n)\| \cdot \frac{1 - H_0}{H_0} \cdot o(f(n))$$

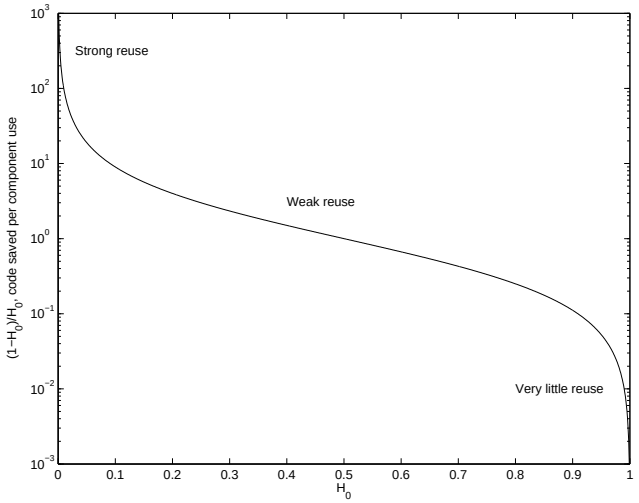


Figure 4: Plot of $\frac{1-H_0}{H_0}$ versus H_0, indicating how much code is saved, proportionately, per component use. When $H_0 \to 1$ there is almost no reuse; $H_0 \to 0$ coincides with the "strong reuse" ideal of wiring together large components. In between is weak reuse, with moderate amounts of code drawn from libraries.

where $\|c(n)\|$ is the code length for the n^{th} component. □

Strong reuse? The interpretation of Theorem 5.3 is fairly intuitive. Roughly it says the savings we can expect per component are linear in the size of the component identifier. Which is to say, we should expect savings for the n^{th} component to grow roughly logarithmicly in n. This is consistent with findings in the reuse literature that *small* components are much more likely to be reused. The important factor here is the multiplier $\frac{1-H_0}{H_0}$. As $H_0 \to 0$, this multiplier becomes arbitrarily large. This suggests that "strong reuse" (Section 1) corresponds to the region $H_0 \to 0$. For example, if programs in a problem domain are thought to be solvable by wiring together components that are (say) 10^5 times bigger than the wiring itself, this suggests $\frac{1-H_0}{H_0} \approx 10^5$ or about $H_0 \approx 0.00001$. The key result is that whether one is able to achieve strong reuse depends critically on the parameter H_0 — which measures how much diversity there is in the problem domain. We therefore have a mathematical model that reproduces our intuitions about when strong and weak reuse may occur.

6. EXPERIMENTAL DATA COLLECTION

Preliminary empirical data was collected from three large Unix installations. The problem domain is not particularly well-defined, but is rather "the mishmash of things one wants to do on a typical research Unix machine." On each machine we located every shared object and used the unix commands `nm` or `objdump` to obtain a listing of the relocatable symbols (i.e., references to subroutines in shared libraries).

Operating System	# Objects	# Subroutines
SunOS	23774	110306
Linux (SuSE)	12136	710691
Mac OS X	2334	37677

We counted the number of references to each subroutine, sorted these by frequency, and this data is plotted in Figure 2. The observed counts match nicely the asymptotic prediction made in Section 4.2 (the family of curves cn^{-1} is shown as dotted lines). To account for machine instructions, which are not included in the tally but constitute by far the most frequently occurring software components, we started numbering the subroutines at $n = 50$. Without this adjustment the rates have a characteristic "flat top" and then rapidly converge to n^{-1} lines; this is an artifact of the log-log scale.

The pronounced "steps" in the data for large n occur because there are many rarely-used subroutines with only a few references; this is typical of such plots (see, e.g., [17]).

Another prediction that may follow simply from our model is that the number of components used in a program should approach a normal distribution with a mean roughly linear in the program size, and variance about the square of the program size. This is reminiscent of the Erdös-Kac theorem [4] that the number of prime factors of integers converges to a normal distribution. Figure 5 shows some preliminary results that support this result, drawn from the SuSE Linux data. The number of component references have been normalized by an estimated variance of $\sigma^2 = cs^2$ where s is the program size. Subfigure (c) shows a suggestively shaped distribution for the inset box of (a), a region where there is good "saturation" of the problem domain with programs.

7. CONCLUSION

We have developed a theoretical model of reuse libraries that provides good agreement, we feel, with our intuitions, the literature, and the preliminary experimental data we have collected on reuse on Unix machines. Much of what we have done has served to emphasize the importance of this one quantity, H_0, the entropy rate we associate with a problem domain. It determines if we can have strong reuse ($H_0 \to 0$), or whether we can have weak reuse ($0 < H_0 < 1$), and how much code we might be able to reuse from libraries: at most $1 - H_0$.

We have shown that libraries allow us to "compress the incompressible," reducing the size of programs that are Kolmogorov-incompressible, by taking advantage of the commonality exhibited by programs within a problem domain. We have also shown that libraries are essentially incomplete, and there will always be room for more useful components in any problem domain.

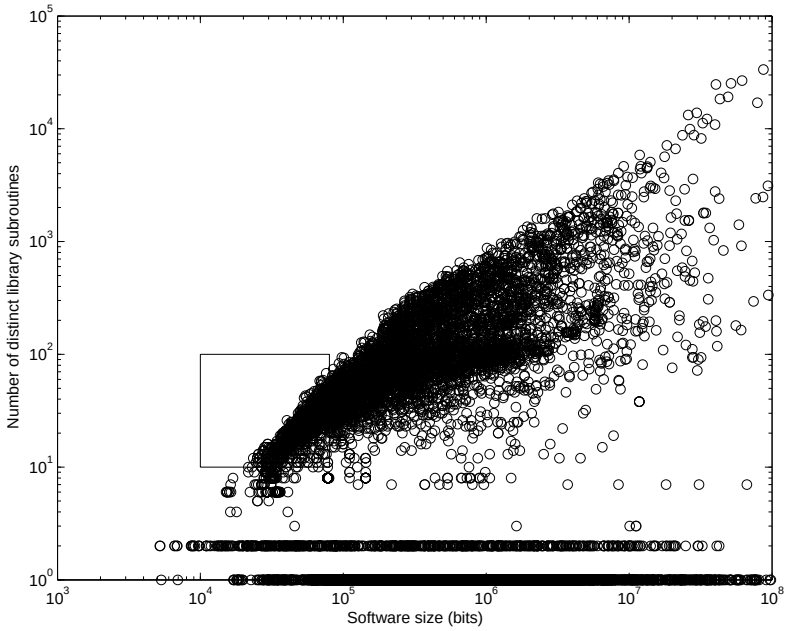
8. ACKNOWLEDGMENTS

This paper benefited immeasurably from discussions with my colleagues at Indiana University Bloomington. In particular I thank Andrew Lumsdaine, Chris Mueller, Jeremy Siek, Jeremiah Willcock, Douglas Gregor, Matthew Liggett, and Brian Barrett for their valuable suggestions. I thank Harald Hammerström for letting me disappear with his copy of Li and Vitányi [13] for most of a year.

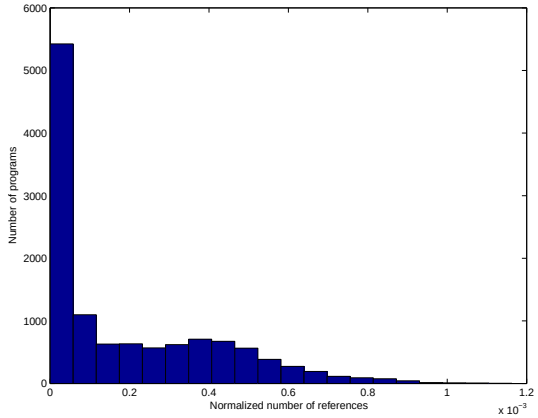
9. REFERENCES

[1] Robert Ash. *Information Theory*. John Wiley & Sons, New York, 3 edition, 1967.

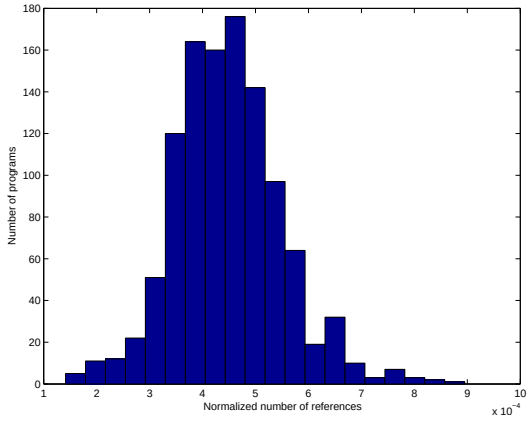
[2] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo.



(a) Scatter-plot of the number of subroutine references



(b) Distribution histogram



(c) Distribution for inset box

Figure 5: Data illustrating the library analogue of the Erdös-Kac theorem. (a) A scatter-plot showing the number of distinct library subroutines used vs. software size for the Linux RPM data. (b) Histogram for the number of references, normalized (see text). (c) Histogram only for the inset box of (a), illustrating an Erdös-Kac-style normal distribution for the number of components used in software. Such plots might provide a useful tool for assessing the extent of reuse vs. ideal predictions from a model.

How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, 1996.

[3] Daniel M. Berry. A new methodology for generating test cases for a programming language compiler. *SIGPLAN Not.*, 18(2):46–56, 1983.

[4] P. Erdös and M. Kac. The Gaussian law of errors in the theory of additive number theoretic functions. *Amer. J. Math.*, 62:738–742, 1940.

[5] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[6] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the HCI'89 Conference on People and Computers V*, Cognitive Ergonomics, pages 443–460, 1989.

[7] M. L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, 32(4):548–566, 1993.

[8] P. Harremöes and F. Topsøe. Maximum entropy fundamentals. *Entropy*, 3(3):191–226.

[9] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[10] D. Kuck. *The Structure of Computers and Computations, Volume 1*. John Wiley and Sons, New York, NY, 1978.

[11] A. Laemmel and M. Shooman. Statistical (natural) language theory and computer program complexity. Technical Report POLY/EE/E0-76-020, August 15 1977.

[12] Mario Latendresse and Marc Feeley. Generation of fast interpreters for Huffman compressed bytecode. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 32–40, New York, NY, USA, 2003. ACM Press.

[13] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag, New York, 2nd edition, 1997.

[14] M. Li and P. M. B. Vitányi. Kolmogorov complexity and its applications. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier, New York, NY, USA, 1990.

[15] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Jeffrey S. Poulin. *Measuring Software Reuse: Princples, Practices, and Economic Models*. Addison-Wesley, 1997.

[17] David M. W. Powers. Applications and explanations of Zipf's law. In Jill Burstein and Claudia Leacock, editors, *Proceedings of the Joint Conference on New Methods in Language Processing and Computational Language Learning*, pages 151–160. Association for Computational Linguistics, Somerset, New Jersey, 1998.

[18] Sheldon M. Ross. *Stochastic Processes*. John Wiley and Sons; New York, NY, 2nd edition, 1996.

[19] David Barkley Wortman. *A study of language directed computer design*. PhD thesis, 1973.

APPENDIX

A. BACKGROUND

A.1 Asymptotics

Here we recall briefly some facts and notations concerning asymptotic behaviour of functions and series. For a more detailed exposition we suggest [5].

Asymptotic notations. For positive functions $f(n)$ and $g(n)$, we make use of these notations for asymptotic behaviour:

$$f(n) \sim g(n) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1 \tag{13}$$

$$f(n) \prec g(n) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{14}$$

$$f(n) \preceq g(n) \iff \exists c \in \mathbb{R}\,.\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \tag{15}$$

The "big-O" style of notation $f \in o(g)$ is equivalent to $f(n) \prec g(n)$. When we write $h(n) \sim g(n) + o(n^2)$ we mean there exists some function $f \in o(n^2)$ such that $h(n) \sim g(n) + f(n)$.

Series and their convergence. A series $\sum_{i=1}^{\infty} a_i$ is convergent when $\lim_{N\to\infty} \sum_{i=1}^{N} a_i$ exists in the standard reals; otherwise it is divergent. The Harmonic series $h_n = \frac{1}{n}$ is divergent, since $\sum_{i=0}^{\infty} h_i = 1 + \frac{1}{2} + \frac{1}{3} + \cdots$ fails to converge.

We shall make use of the following key fact for bounding convergent sequences.

Fact A.1. *Let a_n, b_n be positive sequences. If $\sum_{n=1}^{\infty} a_n$ converges and $\sum_{n=1}^{\infty} b_n$ diverges, then $a_n \prec b_n$.*

Proposition A.1 is useful to establish a bound on the asymptotic growth of a sequence: for example, if $\sum_{n=1}^{\infty} a_n$ must converge, then $a_n \prec \frac{1}{n}$ since the harmonic series diverges.