Towards Formal Proof Metrics

Towards Formal Proof Metrics

David Aspinall[1] Cezary Kaliszyk[2]

[1] LFCS, School of Informatics, University of Edinburgh
Edinburgh EH8 9AB, Scotland
[2] University of Innsbruck Technikerstr. 21a/2 6020, Innsbruck, Austria

Abstract. Recent years have seen increasing success in building large formal proof developments using interactive theorem provers (ITPs). Some proofs have involved many authors, years of effort, and resulted in large, complex interdependent sets of proof "source code" files. Developing these in the first place, and maintaining and extending them afterwards, is a considerable challenge. It has prompted the idea of *Proof Engineering* as a new sub-field, to find methods and tools to help. It is natural to try to borrow ideas from Software Engineering for this.

In this paper we investigate the idea of defining *proof metrics* by analogy with software metrics. We seek metrics that may help to monitor and compare formal proof developments, which might be used to guide good practice, locate likely problem areas, or suggest refactorings. Starting from metrics that have been proposed for object-oriented design, we define analogues for formal proofs. We show that our metrics enjoy reasonable properties, and we demonstrate their behaviour with some practical experiments, showing changes over time as proof developments evolve, and making comparisons across different ITPs.

1 Introduction

Interactive formal proof has advanced to make some impressive achievements, demonstrating that large software and hardware systems can be verified and that large mathematical proofs can be completely captured on machine, giving very high degrees of confidence each case. Some examples are:

- Hales's *FlySpeck* formalisation of his proof of the Kepler Conjecture [13], which includes about 510,000 lines of code proving 27,451 lemmas in the HOL Light interactive theorem prover. This involved a team of 22 people, and an estimated total of 20 person-years of work [14].
- Klein's verification of the seL4 microkernel [20], the core of which consists of almost 400,000 lines of code with 59,000 lemmas, which verifies around 9,000 lines of C and assembler code in the Isabelle ITP. This project involved a team of 13 people, and an estimated total of 20 person-years of work.
- The Compendium of Complex Lattices book formalized in Mizar, performed by a team of 15 people led by Bancerek [3]: it consists of 57 articles with 2,566 theorems and 124,628 lines of proof; it took over 5 person-years.

Draft version. Final publication will appear at `http://link.springer.com`.

In each case, the result consists of instructions written in a dedicated *formal proof language* which direct the proof engine to check a formal proof of some logical statement; these instructions are sometimes called a (formal) *proof script*.

To give further background for those unfamiliar: a *proof script* is somewhat like a program written in an ordinary programming language; like a program it is usually stored in a plain text file. In *interactive* theorem proving, one works intensively with the machine's help to build the proof script; the system checks progress at intermediate points. There are several currently successful ITP systems in which large proofs have been constructed, including the three mentioned above, and others such as HOL4, Coq, ACL2 and PVS. Some ITPs are conceptually similar and use related logics, but each implementation has its own formal proof language so proofs scripts from one cannot be used in another; there are few, if any, useful tools that work for more than one system. This is like the situation with different programming languages, but the user community for each ITP is small. Learning an ITP requires expertise and typically takes months.

Despite the fragmentation, ITPs have continually advanced so that multi-person developments are more common. Leaders of large proof projects have become concerned about the engineering aspects of building and then maintaining large proof scripts, motivating a new sub-field of study: *Proof Engineering* [6, 10, 19]. There are many questions which we do not yet know how to answer. For example: How should a large proof be broken into separate modules? Given a large proof, how can we tell if it is well-structured or in need of improvement, perhaps to improve understanding or maintainability? If a basic definition needs to be changed, how much of the rest of the development will break? These are similar to the concerns of software engineering, so it is natural to ask if software engineering research and practice can provide ideas that transfer.

1.1 From software metrics to proof metrics

In this paper we make some first steps to investigate *proof metrics*, deliberately making a connection to software metrics that have been studied extensively and found utility in practice. Certainly it would be useful to get a handle on the size and complexity of a formal proof and its change over time. It would also be useful to understand how well-structured a formal proof is; one of the most painful proof engineering activities is refactoring existing proofs to change structure [11]; so much so that it is often avoided [12]. So the classic software design goals for modularity of high *coherence* (a module should contain related things) and low *coupling* (connected modules should only have a few connections) are equally applicable to formal proof development.

As our starting point, we take inspiration from the landmark metrics for object-oriented design that were introduced by Chidamber and Kemerer [7] ("C&K"). Although the C&K metrics have since been criticised and modified in a myriad of ways, they still stand as a plausible starting point for a new application area. In particular, they are appropriate because they have simple definitions and motivations and also because there is a rough analogy between the static structure of an object-oriented design and the structure of proof scripts.

OOP	Formal proof
class	proof module
class inheritance	proof module import
instance variable	declaration of a type or constant
method	theorem
method type	theorem statement
method body	theorem proof

Fig. 1. A loose analogy between object-oriented programming and formal proof

Just as with programs, large formal proofs are broken up into *modules*. Various sorts of module have been studied and are adopted in different systems, supporting both in-the-small structuring (e.g., capturing the notion of an algebraic structure with its operations and axioms: *locales* in Isabelle) and in-the-large (e.g., capturing a whole collection of definitions and properties about groups). Here we are concerned with in-the-large modules which form the basic top-level decomposition of a formal proof. In Isabelle modules are called *theories*, in Mizar they are called *articles*; in HOL Light, modules are identified with files.

Top-level modules contain statements to be proved and their proofs, which we (and most ITPs) call *theorems*. A formal proof also needs to define the subject of its concern: whether some mathematics or a proof of software correctness, *declarations* and *definitions* are needed to introduce types and constants of discourse. Although specific mechanisms differ, top-level modules have some *import* mechanism to allow access to other modules. Scoping mechanisms for restricting visibility are currently either primitive or little used; hence visibility of imported theorems usually extends transitively through imported modules, just as class inheritance extends member visibility transitively through the class hierarchy. (Section 2 shows some small example proofs and import graphs, along with the abstraction that we use to define metrics.)

This leads us to the loose analogy shown in Fig. 1. Classes in OO design or programming are like proof modules. Theorems are somewhat like methods: both have complex bodies that describe their implementation. In OO, instance variables capture the nature of what a class models; methods inspect and manipulate the variables. In ITP land, theorem statements describe properties of (immutable) types and constants. Proofs may refer to further types and constants and to other theorems: usually ones proved earlier in some well-founded ordering. This is analogous to methods that invoke other methods in their implementation. (As an aside: ITPs grudgingly admit theorems without proof: *assumptions* or *axioms* taken as given; this allows a form of top-down development, but does not really make up for the lack of any modeling language or technology.)

The plan is now clear. Using the analogy, size metrics that consider the number of methods in a class can be recast as metrics counting the number of theorems in a theory. Metrics based on the class hierarchy relationships can be recast to examine dependencies among proof modules. And so on; how we recast the C&K metrics is defined precisely and discussed in Section 3. We can also

show that our metrics satisfy some analytical properties which have been studied in software metrics; this is covered in Section 4.

There is a risk that the investigation may be futile. Despite good properties, naive translations of software metrics using Fig. 1 could result in functions that are uninteresting or meaningless in the setting of formal proof. Our analogy is rough: the structuring mechanisms differ, and programs have dynamics which is the whole purpose of their construction. But the dynamics of a completed formal proof is a one-shot check and annotation: a black-box operation in which the ITP emits compilation errors if the developer made a mistake or a "QED" acknowledgement in the case of success.

To demonstrate that our metrics may be actually interesting in practice, we examine several large repositories of formal developments, taken from three different ITPs. Our metrics confirm some expected "folklore" aspects of the system differences. Looking at some version control history, we are able to correlate certain changes in formal proof files with changes in the expected metrics, and vice-versa. Our practical experiments are described in Section 5.

Contributions. We believe this is the first attempt to adapt ideas from software metrics to formal proof that goes beyond size-based metrics. We contribute:

1. a simple abstract model definition for formal proofs;
2. a precise definition of a set of *proof metrics* using this model;
3. (informal) proofs that our metrics satisfy some reasonable properties;
4. an implementation of the metrics for three different ITPs;
5. demonstration of metrics for various large proof corpora;
6. demonstration of the historical change of the metrics on 38 versions taken from the version control history of HOL Light.

As well as size measurements, our metrics include *complexity* against relationships and positions within a proof development and estimates of *interdependence* between parts of a formal proof.

Related work. There is a large literature on software metrics concerning their definition and empirical study as well as (often) debating their utility. Despite any debate, metrics continue to be studied and used in practice. For example, metrics are used in cost estimations models (e.g., COCOMO and variants; see Trendowicz and Jeffrey for a recent overview [29]). They can also be used to implement heuristics to detect "bad smells" in code that may suggest when refactorings may be desirable [27] or where they have occured in the past [9].

On the side of formal proof, the topic is quite new. Researchers in the seL4 project have set out an agenda similar to ours [17] and empirically demonstrated a relationship between theorem statement size and proof size [23] in seL4 and, to a restricted extent, other Isabelle proofs (see Sects. 5 and 6 for more remarks). Also for Isabelle, Blanchette et al [5] studied the dependencies and size of whole formalization libraries in the AFP contributed library. They found little entry reuse based on the import graph and showed that the size distribution of the entries follows the power law. Working with Mizar proofs, Pąk investigated ways

of improving proof readability using notions of legibility based on locality of reference, taking inspiration from models of cognitive perception [24]; this is similar to the software engineering idea of cohesion, which is a metric we define. We mention some other related work in the body of the paper.

2 Programming formal proofs

Languages of different ITPs vary considerably, but all provide the user with a way to express theorem statements and give proofs, which the system verifies. There are two predominant styles of proof script: *procedural*, where user instructions (*tactics*) are transformations that successively refine a state backwards from the goal; and *declarative*, where instructions drive the ITP forwards to its target, providing *justifications* "by" that gives the ITP hints. Some systems support both styles. Often, procedural proofs are easier to write and declarative proofs are easier to read. To give a flavour, Fig. 2 shows two formal proof excerpts.

In both styles, instructions used in a theorem's proof take arguments that are theorems themselves, creating dependencies between theorems. Circular dependencies are not allowed: to prove a theorem P using another theorem Q, Q must be provable itself without P. As proof libraries grow larger, theorems are collected together into modules; a well organised library collects related theorems together. Modules also have a cycle-free dependency ordering, as new modules are built from older ones. Fig. 3 shows the dependencies between modules and theorems in the first part of the HOL Light library.

```
let REAL_INV_SGN = prove
  (‘∀x. real_inv(real_sgn x)
    = real_sgn x‘,
  GEN_TAC THEN
  REWRITE_TAC[real_sgn] THEN
  REPEAT COND_CASES_TAC THEN
  REWRITE_TAC[REAL_INV_0;
    REAL_INV_1;
    REAL_INV_NEG]
  );;
```

```
lemma abs_triangle_ineq2:
  "|a| - |b| ≤ |a - b|"
proof -
  have "|a| = |b + (a - b)|"
    by (simp add: algebra_simps)
  then have "|a| ≤ |b| + |a - b|"
    by (simp add: abs_triangle_ineq)
  then show ?thesis
    by (simp add: algebra_simps)
qed
```

Fig. 2. A HOL Light procedural proof (left) and an Isabelle declarative proof (right).

2.1 Formal proof developments, abstractly

We now model this situation. Suppose two sets of identifiers: the *module names* $\mathcal{M}$ and the *theorem names* $\mathcal{T}$. For simplicity (to avoid considering notions of scope) but without loss of generality, we assume that theorem names are globally unique. So each theorem belongs to a module: the mapping $\mathsf{mn} : \mathcal{T} \to \mathcal{M}$ returns

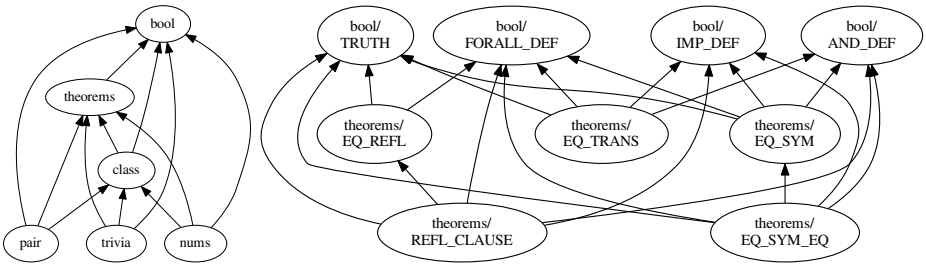

Fig. 3. Module dependencies and theorem dependencies in HOL Light.

the module name of a given theorem. We use "theorem" in a general sense, in reality ITP modules can contain declarations or definitions of various other things (axioms, constants, types, syntax, etc.); we are agnostic whether these are included in the abstract notion of "theorem" or not.

Definition 1 (Proof module and proof development).

- *A* proof module *is a pair* (M, T) *of a module name* $M \in \mathcal{M}$ *and a finite set of theorem names* $T \subset \mathcal{T}$ *such that* $\mathsf{mn}(t) = M$ *for all* $t \in T$.
- *A* proof development $P = \{(M, T_M)\}$ *is given by a finite set of proof modules having distinct module names* M.

Formal mathematics is a well-founded endeavour: later definitions or theorems may only depend on ones that have been given earlier. Theorem dependency relations have been investigated in practice before for real systems (e.g., [1, 26]) but the next definition has not been spelled out before.

Definition 2 (Proof development dependency).

- *A* module dependency (uses) relation $\rightarrow^{\mathrm{M}}$ *is a well-founded relation on a subset of* $\mathcal{M}$. *We write* $\leq_{\mathrm{M}}$ *for the reflexive, transitive closure of* $\rightarrow^{\mathrm{M}}$.
- *A* theorem dependency (uses) relation $\rightarrow^{\mathrm{T}}$ *is a well-founded relation on a subset of* $\mathcal{T}$. *We write* $\leq_{\mathrm{T}}$ *for the reflexive, transitive closure of* $\rightarrow^{\mathrm{M}}$.
- *A* dependency for a proof development P *is given by a module dependency relation* $\rightarrow^{\mathrm{M}}$ *on the module names of* P, *together with a theorem dependency relation* $\rightarrow^{\mathrm{T}}$ *on all of the theorem names in* P, *which respects* $\rightarrow^{\mathrm{M}}$ *in the sense that* $t_1 \rightarrow^{\mathrm{T}} t_2 \implies \mathsf{mn}(t_1) \leq_{\mathrm{M}} \mathsf{mn}(t_2)$.

Thus, a proof development forms a DAG of modules which overlays a set of DAGs of theorems. Note that we distinguish direct or "immediate" dependencies from indirect ones: a theorem t_3 may use a theorem t_2, in its proof which in turn uses t_1 ($t_3 \rightarrow^{\mathrm{T}} t_2$ and $t_2 \rightarrow^{\mathrm{T}} t_1$); but t_3 may have a different proof that uses both t_2 and t_1 directly ($t_3 \rightarrow^{\mathrm{T}} t_1$ and $t_3 \rightarrow^{\mathrm{T}} t_2$). In both cases t_3 ultimately depends on t_1, so $t_3 \leq_{\mathrm{T}} t_1$.

Module dependencies suggest the proof checking (or compilation) order: we suppose that proofs of theorems in each module are checked together, and modules are checked in some defined or inferred order. Then $M_2 \leq_{\mathrm{M}} M_1$ says that

M_2 is the later module that may build on concepts and lemmas, etc., given in M_1. So whenever $t_2 \rightarrow^{\mathrm{T}} t_1$, we require that t_1 has been checked in the same or an earlier module (perhaps transitively earlier) than t_1.

The converse implication need not hold; module dependencies can be "loose" in that they may not reflect any (direct or transitive) theorem dependencies; a bit like "redundant imports" in programming languages. We call a module dependency $M_1 \rightarrow^{\mathrm{M}} M_2$ between different modules *strict* only when there is indeed some $t_1 \leq_{\mathrm{T}} t_2$ for which $\mathsf{mn}(t_1) = M_1$ and $\mathsf{mn}(t_2) = M_2$. In our implementation of the metrics to follow, we take the theorem dependency relation to be primary, and derive $\rightarrow^{\mathrm{M}}$ as the minimal strict relation between modules which respects $\rightarrow^{\mathrm{T}}$. This is an implementation choice that conveniently unifies the treatment between different systems.

The theorem dependency relationship is all that we use to model the bodies of theorems (an axiom has no dependencies). But we need more to capture theorem statements. To avoid any detail of the logical language for statements we suppose that there is an abstract set of *features* $\mathcal{F}$ which capture the constants, types, etc. that a theorem statement may refer to. We suppose, for simplicity again, that every theorem name is associated globally with a statement, so there is a mapping $\mathsf{fea} : \mathcal{T} \rightarrow \mathit{Fin}(\mathcal{F})$. Given a theorem t, $\mathsf{fea}(t) \subset \mathcal{F}$ is the finite set of features used in its statement. For example, the Kepler conjecture, which gives an upper bound on ball packings in $\mathbb{R}^3$ with the formal HOL Light statement:

```
∀V. packing V ⟹ (∃c. ∀r. &1 ≤ r
                 ⟹ &(CARD(V INTER ball(vec 0,r))) ≤
                    pi * r pow 3 / sqrt(&18) + c * r pow 2)
```

can be characterized by the following features based on the constants and types appearing in its formal statement:

$$\begin{aligned}\mathsf{fea}(\mathsf{kepler\_conjecture}) = \{&\mathsf{packing}, \mathsf{sqrt}, \mathsf{ball}, \mathsf{pi}, \mathsf{BIT0}, \mathsf{BIT1}, \mathsf{NUMERAL}, \mathsf{0},\\ &\mathsf{real\_add}, \mathsf{real\_div}, \mathsf{real\_le}, \mathsf{real\_mul}, \mathsf{real\_of\_num}\\ &\mathsf{real\_pow}, \mathsf{CARD}, \mathsf{INTER}, \mathsf{3}, \mathsf{cart}, \mathsf{num}, \mathsf{prod}, \mathsf{real}\}.\end{aligned}$$

3 Six simple proof metrics

In their landmark paper [7], now over 20 years old, Chidamber and Kamerer proposed metrics for object-oriented design which are also applicable to implementations in object-oriented programs. They consolidated earlier work and aimed to set their metrics on a rigorous footing. They designed six metrics for OOD: each metric is a function on a class. The metrics were evaluated by checking analytically that they possess reasonable properties and by examining the result of their application in two real software projects.

Here we revisit C&K's metrics and recast them for formal proof developments. There have been many criticisms, variants and improvements on C&K's work, and empirical studies providing varying degrees of external validation. In this first study, we cannot expect to find perfect metrics for formal proof so we start off with a "straw-man" proposal inspired by this indisputably key work.

3.1 WTM: Weighted Theorems Per Module

WTM is our analogue of C&K's WMC, *Weighted Methods per Class*, which is a basic size assessment of a module. Let the theorems of a proof module M be $T_M = \{t_1, \ldots, t_n\}$. Then WTM is defined by:

$$\mathrm{WTM}(M) = \sum_{i=1}^{n} c(t_i)$$

where c is some complexity function applied to theorems t_i in the theory T.

The idea of the complexity function is that it allows more complex theorems to be given a higher weighting. For example, we could measure the size of the theorem statement (perhaps counting distinct constants unwinding definitions recursively [6, 23]), or, we could instead measure proof size by counting the number of lines-of-code in the proof script for the theorem's proof. The simplest choice (and common in OOP studies) is to take c to be the identity, so WTM=n, the number of theorems in the theory.

3.2 DIT: Depth in Tree

The metric DIT calculates the maximum depth of the proof module in the module dependency graph; it corresponds to the DIT metric for a class in OOP which measures the depth of a class in the inheritance tree. Intuitively, higher DIT values in a proof development suggest modules that are potentially more complex since they rely on more levels of previously constructed proofs. We define:

$$\mathrm{DIT}(M) = \mathit{depth}_P(M)$$

where $\mathit{depth}_P(M_n)$ is the length n of the longest path $M_n \to^{\mathrm{M}} M_{n-1} \cdots \to^{\mathrm{M}} M_0$.

3.3 NOC: Number of Children

The Number of Children, NOC, for a proof module is the number of immediate descendent modules that depend upon it. This is defined as:

$$\mathrm{NOC}(M) = |\{ M' \mid M \to^{\mathrm{M}} M' \}|$$

In OOP, this is a measure of scope of a class: how many other classes immediately depend on this one. Intuitively, modules with higher NOC values may incur greater cost to change, since a local change will have a broader effect. But at the same time, a higher NOC shows a greater reuse, indicating that the module is actually used in many places, demonstrating that it is good or important.

C&K suggest that too many children may indicate "improper" abstraction: superclasses should not be overly general. They found a case of this in a project they examined. In proof developments, we expect the core library modules to have many children. OOP languages often have a separate import mechanism for library functions, independently of subclassing. This is a tension in our analogy: class inheritance is arguably more akin to theories in-the-small, but in-the-small proof modules are less widely used and harder to compare across systems.

3.4 CBM: Coupling between Modules

C&K define a metric called Coupling Between Object classes (CBO) which is a complexity measure on the class hierarchy. Two classes are coupled if one uses member functions or instance variables of the other. Our analogous metric is:

$$\mathrm{CBM}(M) = |\{\, M' \mid M \to^{\mathrm{M}} M' \vee M' \to^{\mathrm{M}} M \}|$$

Intuitively, coupling refers to the degree of interdependence between parts of a design: it means dependency as ancestor or child. Modules with higher coupling values are more closely bound into the proof development hierarchy, meaning that they may be difficult to understand in isolation. The simple definition above doesn't account for a multiplicity of couplings between modules. Later work on OOP coupling metrics addressed this; we might similarly consider a metric which counts the number of strict theorem dependencies that cross module boundaries.

3.5 TDM: Total Dependencies for Module

C&K's next metric is RFC, Response For a Class, which counts the number of methods that could be executed in response to a message received by an object of that class. Intuitively this may estimate the potential (dynamic) complexity of behaviours of objects in the class; high RFC values might suggest classes that are harder to test. We re-interpret this using theorem dependency:

$$\mathrm{TDM}(M) = |\{\, t' \mid t \to^{\mathrm{T}} t' \wedge t \in T_M \}|$$

There is no analogue of "response" for a theorem, but just as invoking a method $m()$ leads to invoking other methods mentioned in the body of m, our metric counts the number of theorems depended on in the definition of a given theorem. Notice that this includes internal dependencies which do not cross the module boundary, as well as external ones.

In the proof setting, we hypothesise that this metric may suggest the overall brittleness of a theory: if (too) many other theorems are used in the construction of a module, it may break easily if the statements of those other theorems change.

3.6 LCOM: Lack of Cohesion in Module

The final metric given by C&K is LCOM, originally Lack of Cohesion of Methods in a class. Cohesion refers to internal consistency within a module; a high LCOM value suggests a module that gathers together many unrelated things. C&K's metric is defined as the difference between the number of pairs of methods which that entirely different instance variables (p), and the number of pairs of methods that access some of the same attributes (q); LCOM was taken to be $p - q$ or zero if $q > p$. Early on, LCOM was criticised for failing to fit empirical data [4], spawning a slew of alternatives (for partial surveys, see e.g. [2, 22]).

As a first proposal for the formal proof setting, we suggest a metric based on theorem statement dissimilarity counted using features: two theorems are

similar if they concern the same concepts, and so mention the same constant names, types, etc.[3] An overall measure of similarity for the module is given by summing up the Jaccard index for each pair of theorem statements:

$$\mathrm{sim}(M) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{|\ \mathsf{fea}(t_i) \cap \mathsf{fea}(t_j)\ |}{|\ \mathsf{fea}(t_i) \cup \mathsf{fea}(t_j)\ |}$$

Then we compute LCOM as the average dissimilarity:

$$\mathrm{LCOM}(M) = 1 - \frac{\mathrm{sim}(M)}{\frac{1}{2}(n^2 - n)}.$$

(this is similar to CC, among others [2]). Unlike the preceding metrics which are on an interval scale with no maximum, LCOM is a ratio in the range 0 to 1.

4 Properties of proof metrics

Weyuker [31] proposed desirable analytical properties for structured program metrics, six of which were adapted by C&K to OOD, again, generating much subsequent discussion and criticism. Here we briefly revisit the properties and their connection to our metrics.

Several properties relate to combinations of programs; for structured programming this meant, essentially simple juxtaposition of source code $P; Q$. Composition in OOP is more complicated. But for our simple model of formal proof languages, we suppose that the operation $M + M'$ stands for (disjoint) combination of modules.

Proposition 1. *Properties of formal proof metrics.*

W1 ***Non-coarseness****. Given a module M and a metric μ, one can always find a module M' st $\mu(M) \neq \mu(M')$.* This is satisfied by all of our metrics.

W2 ***Non-uniqueness****. There can be distinct modules M and M' with $\mu(M) = \mu(M')$.* This is satisfied by all of our metrics.

W3 ***Design is important****. Two modules M and M' may have the same meaning without $\mu(M) = \mu(M')$ holding.* If we take the semantics of an abstract module to be the set of (named) theorems it proves, then this property is not satisfied by the basic size metric WTM metric or the cohesion metric LCOM, which only consider the (number of) theorem statements and don't relate to design-in-the-large. Of course the property holds if we consider the full proof language, which has a complex concrete syntax, so many ways to describe the same module.

W4 ***Monotonicity****. For all M and M', $\mu(M) \leq \mu(M+M')$ and $\mu(M') \leq \mu(M+M')$.* This is true for all of our metrics except LCOM, since it is normalised for comparison between modules; LCOM can be reduced by adding theorems to a module that have an average greater similarity to those already there.

[3] A similar idea was in fact suggested by Matichuk et al [23] as future work.

W5 ***Combination can cause interaction.*** *$\exists M_1, M_2, M_3$ such that $\mu(M_1) = \mu(M_2)$ does not imply $\mu(M_1 + M_3) \neq \mu(M_1 + M_3)$.* This property can be satisfied for all the dependency related metrics, but not the size measure WTM. It can be satisfied by LCOM since this is non-compositional.

W6 ***Interaction can increase complexity.*** *$\exists M_1, M_2$ such that $\mu(M_1)+\mu(M_2) < \mu(M_1 + M_2)$.* This opposite of the triangle inequality fails for all of our metrics except LCOM. C&K argued against it; all of their metrics failed it and it prevents μ being a distance metric in the mathematical sense.

5 Experimental study

To test our metrics, we implemented a tool to make calculations using feature and dependency data exported from (suitably modified) ITPs during proof checking. Then we investigated the metrics on a range of existing formal proof developments and their version histories.

5.1 Large proof development examples

We investigated large developments in three systems:

1. The Kepler formal proof, FlySpeck: we focused on the final version of the text formalization [13] together with the underlying core library of HOL Light and the formalization of Multivariate Analysis [15] (SVN revision 245).
2. The Isabelle HOL Main (core library) theory together with three formalizations: cryptographic protocols, Auth [25]; Java bytecode, Bali; and Probability theory [16] (Isabelle 2015 release version).
3. The Mizar Mathematical Library. We focused on the basic libraries of formalized topology and theory of lattices [3] (Mizar version 7.11.07, MML version 4.156.1112).

In each case, information was exported in a uniform format, containing theorem dependencies and statement features. To be as similar as possible across provers, we used symbol features (i.e., names of constants and types present in the theorem statement, as shown in Sect. 2.1), rather than more complex notions.

For HOL Light, we used the HOLyHammer proof advice tool [18] by Kaliszyk and Urban. For the Isabelle formalizations we used the Blanchette's TPTP/-Mash_Export [21], which can compute dependencies and MaSh features for a given set of Isabelle/HOL theories. For Mizar, used the data available in the MPTP2078 challenge by Urban [30]. The challenge includes the proof dependencies and statements for the selected Mizar articles. We extracted symbol features using standard TPTP tools [28].

Space precludes a complete breakdown of metric values, but the summary in Fig. 4 shows the averages for each development. From theorem count totals we can see, for example, how large Flyspeck is; but metrics give an idea of the form of its structure: it has a large number of modules with comparatively fewer theorems, compared to library code. This likely contributes to the better cohesion score. For module hierarchy, the development is, on average, almost twice as deep as the next deepest, Isabelle's highly structured Main HOL library.

		#mods	#thms	TDM	WTM	NOC	CBM	DIT	LCOM%
HOL Light	Core	21	2618	391	125	7.4	14.8	6.8	75.2
	Multivariate	19	11093	3091	584	7.0	34.9	7.6	75.6
	Flyspeck	237	12999	1582	55	12.6	44.1	27.6	62.0
Isabelle	Main	73	12731	357	174	8.9	17.9	17.6	72.2
	Auth	38	4282	202	209	2.9	12.7	3.9	57.1
	Bali	25	6946	261	502	4.7	16.8	4.2	66.8
	Multivariate	50	7821	245	287	2.1	16.5	3.9	61.0
	Probability	45	5928	460	246	4.7	26.5	7.7	62.7
Mizar	MPTP2078	33	3646	270	110	9.3	26.3	10.2	73.8

Fig. 4. The overall statistics for the considered proof libraries together with the mean values of the proposed proof metrics computed on these libraries.

5.2 Distribution of cohesion and coupling

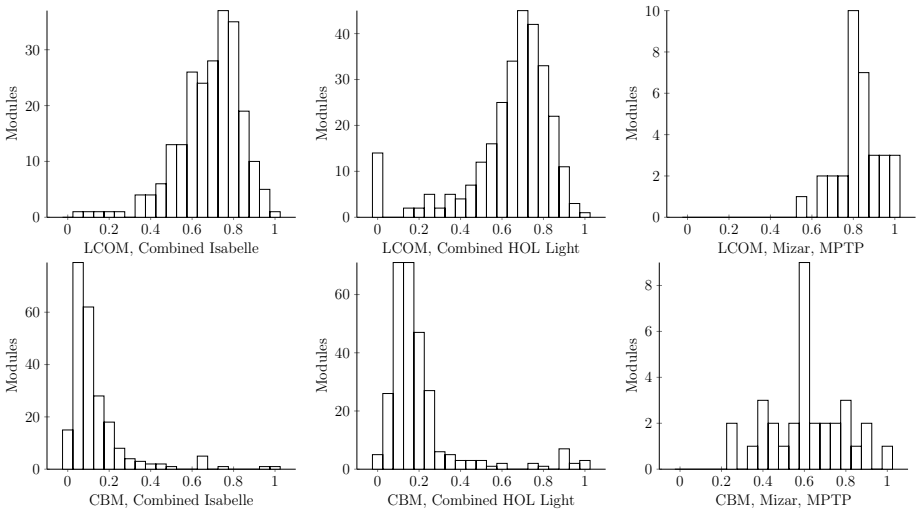

Fig. 5. Cohesion (top) and coupling (bottom) distributions across modules in the three ITPs. Coupling is normalized to $[0; 1]$ for comparison.

In general, the metrics differ widely across modules in the same formal library, so it is interesting to examine their distributions. Fig. 5 shows the distributions of cohesion and coupling across the three considered ITPs. Cohesion shows a similar distribution across the systems, with slightly higher values for Mizar. For Mizar we focused on a more advanced part of the library without including all the foundational modules; perhaps suprisingly we see high dissimilarity scores

within those modules. In HOL Light we see a peak on the histogram for the zero bracket. This is because of a few Flyspeck modules that export precisely one theorem with a large complicated proof. When it comes to coupling, we see that coupling is generally low, but higher for the Mizar case: this is to be expected because we considered a self-contained development in isolation.

5.3 LCOM, TDM, and WTM over time

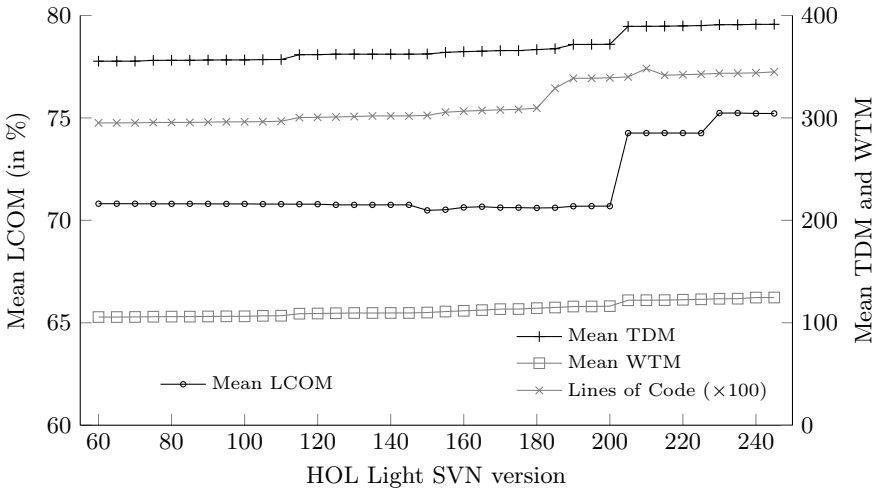

Fig. 6. The values of the metrics for the HOL Light core library compared to the number of lines of code over five years of HOL Light development.

We compared metrics for the HOL Light core library over the last five years of HOL Light development, by exporting the data from 38 selected SVN revisions. The values of selected metrics for these revisions are depicted in Fig 6. In general, the library has grown to prove more theorems over time without changing its modular structure. The average WTM and TDM both increase, showing the growing average number of theorems per module and complexity of the dependency relationship. The jump in LCOM and TDM between revisions 200 and 205 can be traced back to the removal of a module called `ind_defs`, the only change in modular structure that we see. As the library becomes more dense, similar theorems added to the same modules may decrease LCOM, which is seen around version 145. LCOM can also change because of library restructuring; we show an example of this next.

5.4 Case study: HOL Light refactoring

On Dec 1, 2011 John Harrison slightly reorganised the HOL Light library. He moved definitions of supremum and infimum of a set along with all the properties

		real	sets	infsup	misc	Ave.
LCOM (%)	before	68.8	90.5	-	84.7	81.3
	after	68.7	90.3	-	87.4	82.1
	separate	68.7	90.5	49.6	87.4	74.0
TDM	before	286	501	-	503	430.0
	after	292	542	-	482	438.7
	separate	292	501	237	482	376.5
WTM	before	286	468	-	155	303.0
	after	289	517	-	103	303.0
	separate	289	468	49	103	227.3

Fig. 7. The impact of a library reorganization on the proof metrics.

of these concepts from `Multivariate/misc` to `sets`. Three basic Archimedian properties moved from `Multivariate/misc` to `real`. Finding this history in the version control logs, we examined the impact on our metrics.

Fig. 7 shows the results for LCOM, TDM and WTM. The latter metrics reflect moves but averages do not change, since the theorems proved and their proofs stay the same. For LCOM, there is small reduction for `real` and `sets` by the moves, reflecting that relocated theorems enjoyed similarity and/or were similar to those of their destination modules. But the `misc` module was left with a poorer score; less similarity remained among what was left. As an experiment we tested what would happen if a separate module `infsup` was added to hold the relocated theorems about infimum and supremum; the new module introduces few cross-module dependencies decreasing TDM and has a much better cohesion score which brings down the average LCOM value for the whole development.

5.5 Theorem size and the number of dependencies

In a striking recent result, Matichuk et al showed that proof size increases quadratically with statement size (measured by recursively unfolding constant names) in the seL4 verification [23]. This is potentially useful as a predictor of effort, in connection to earlier work that shows a relation between human effort and proof size in seL4.

As a related comparison with a different ITP, and to investigate potential ways of measuring statement size, we compared the number of dependencies with the number of theorem features, the number of subterms in a statement and the size of the theorem statement in MPTP2078. The scatter plots in Fig. 8 show our results. Although increasing numbers of dependencies tend to correspond with larger statement sizes, there are no clear relationships, and plenty of outliers. This is not surprising: we are in a very different setting, with mathematical proofs constructed using an automation strategy, which is rather powerful for the considered domain.

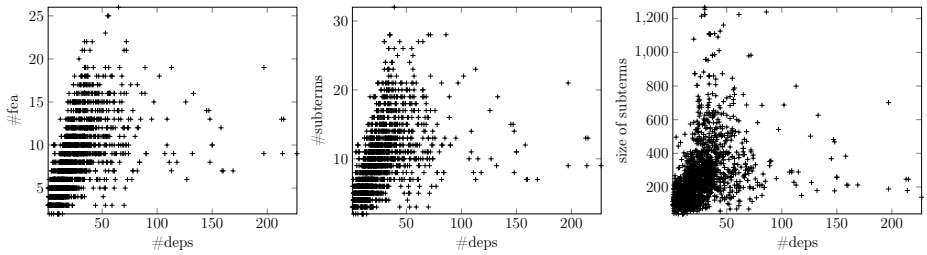

Fig. 8. The number of dependencies compared to three notions of theorem statement size on the Mizar/MPTP2078 proof library.

6 Conclusions

This is the first (to our knowledge) investigation of formal proof metrics which considers both the *modular structure* of a proof development and its size profile. We also gave the first implementation and experimental data for metrics applied to proofs in more than one theorem proving system, raising intriguing questions of whether such measures can be used to compare developments across systems.

There are many caveats for this initial study. It is easy to imagine improvements to our definitions, or to spot potential flaws (e.g., one issue: we count dependencies manifested in final proofs, rather than ones the user mentioned).

Nevertheless, we are believe that our results give evidence of potential value for proof metrics. A central question around metrics — can we show that they actually measure something? — is perhaps even more thorny than for software. Notions of formal proof quality are not yet developed and there are questions over what to assess empirically. Defect prediction is not an obvious aim; bugs as such do not exist in formal proof. If a proof is found by the system, it must be correct! (Saying this, definitions and theorem statements *can* be wrong, even inconsistent, which is serious; dependency metrics might provide hints on that.) Effort prediction in general cannot be feasible: ITPs work in undecidable proof systems, which means that there are profound theorems that have short statements but will need immensely long proofs. The Kepler statement shown in Sect. 2.1 is (almost certainly) just such an example. So it is hard to imagine a mathematical analogue of "function points". Software or hardware verification is a more hopeful domain: where a body of proofs exists about some complex system, we may hope to see correlated scaling effects in proof size or effort as the system evolves, or as more properties are proven; Matichuk et al [23] have demonstrated a case of this, as mentioned above. And a different, perhaps more transferable, use of proof metrics may be for *stability* tracking, to see where a development seems to be proceeding to an optimal design. This was found to be effective in a study of a particular object-oriented framework [8].

Even without general predictive models, we suspect that proof metrics will find a valuable use inside a range of future tools that provide monitoring of proof development progress, and perhaps hints of "bad smells" in a development. We look forward to their further investigation and application.

Acknowledgements and pointers. We're grateful to colleagues Iain Whiteside, Ajitha Rajan and the DReaM group at Edinburgh for discussions. The referees provided useful remarks. We acknowledge financial support from grants from UK EPSRC (EP/J001058/1) and the Austrian Science Fund (P26201). For tools and data, please visit `http://homepages.inf.ed.ac.uk/da/proofmetrics/`.

References

[1] Jesse Alama, Lionel Mamane, and Josef Urban. Dependencies in Formal Mathematics: Applications and Extraction for Coq and Mizar. In: *Intelligent Computer Mathematics, CICM 2012*. 2012, pp. 1–16.

[2] Jehad Al-Dallal and Lionel C. Briand. A Precise Method-Method Interaction-Based Cohesion Metric for Object-Oriented Classes. In: *ACM Trans. Softw. Eng. Methodol.* 21.2 (2012), 8:1–8:34.

[3] Grzegorz Bancerek and Piotr Rudnicki. A Compendium of Continuous Lattices in MIZAR. In: *J. Autom. Reasoning* 29.3-4 (2002), pp. 189–224.

[4] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In: *IEEE Trans. Software Eng.* 22.10 (1996), pp. 751–761.

[5] Jasmin Christian Blanchette et al. Mining the Archive of Formal Proofs. In: *Intelligent Computer Mathematics, CICM 2015*. Vol. 9150. LNCS. Springer, 2015, pp. 3–17.

[6] Timothy Bourke et al. Challenges and Experiences in Managing Large-Scale Proofs. In: *Intelligent Computer Mathematics CICM 2012*. Vol. 7362. LNCS. Springer, 2012, pp. 32–48.

[7] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493.

[8] Serge Demeyer and Stéphane Ducasse. Metrics, Do They Really Help? In: *Proceedings LMO '99 (Languages et Modèles à Objets)*. Ed. by J. Malenfant. 1999, pp. 69–82.

[9] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In: *Object-Oriented Programming Systems, Languages & Applications, OOPSLA 2000*. 2000, pp. 166–177.

[10] Georges Gonthier. Engineering Mathematics: The Odd Order Theorem Proof. In: *Principles of Programming Languages, POPL'13*. ACM, 2013, pp. 1–2.

[11] Georges Gonthier et al. A Machine-Checked Proof of the Odd Order Theorem. In: *Interactive Theorem Proving (ITP)*. Vol. 7998. LNCS. Springer, 2013, pp. 163–179.

[12] Thomas C. Hales. The Jordan Curve Theorem, Formally and Informally. In: *The American Mathematical Monthly* 114.10 (2007), pp. 882–894.

[13] Thomas C. Hales et al. A formal proof of the Kepler conjecture. In: *CoRR* abs/1501.02155 (2015).

[14] Thomas C. Hales et al. A Revision of the Proof of the Kepler Conjecture. In: *Discrete & Computational Geometry* 44.1 (2010), pp. 1–34.
[15] John Harrison. The HOL Light Theory of Euclidean Space. In: *J. Autom. Reasoning* 50.2 (2013), pp. 173–190.
[16] Johannes Hölzl and Armin Heller. Three Chapters of Measure Theory in Isabelle/HOL. In: *Interactive Theorem Proving - Second International Conference (ITP)*. Vol. 6898. LNCS. Springer, 2011, pp. 135–151.
[17] D. Ross Jeffery et al. An empirical research agenda for understanding formal methods productivity. In: *Information & Software Technology* 60 (2015), pp. 102–112.
[18] Cezary Kaliszyk and Josef Urban. Learning-Assisted Automated Reasoning with Flyspeck. In: *J. Autom. Reasoning* 53.2 (2014), pp. 173–213.
[19] Gerwin Klein. Proof Engineering Considered Essential. en. In: *FM 2014: Formal Methods*. Vol. 8442. LNCS. Springer, 2014, pp. 16–21.
[20] Gerwin Klein et al. seL4: formal verification of an OS kernel. In: *Symposium on Operating Systems Principles SOSP 2009*. ACM, 2009, pp. 207–220.
[21] Daniel Kühlwein et al. MaSh: Machine Learning for Sledgehammer. In: *Interactive Theorem Proving, ITP 2007*. Vol. 7998. LNCS. Springer Verlag, 2013, pp. 35–50.
[22] Andrian Marcus and Denys Poshyvanyk. The Conceptual Cohesion of Classes. In: *IEEE International Conference on Software Maintenance ICSM 2005*. 2005, pp. 133–142.
[23] Daniel Matichuk et al. Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification. In: *International Conference on Software Engineering, ICSE 2015*. 2015, pp. 722–732.
[24] Karol Pąk. Automated Improving of Proof Legibility in the Mizar System. In: *Intelligent Computer Mathematics, CICM 2014*. Vol. 8543. LNCS. Springer, 2014, pp. 373–387.
[25] Lawrence C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. In: *J. Comput. Secur.* 6.1-2 (1998), pp. 85–128.
[26] Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In: *User Interfaces for Theorem Provers (UITP)*. 1998.
[27] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics Based Refactoring. In: *Software Maintenance and Reengineering, CSMR 2001*. 2001, pp. 30–38.
[28] Geoff Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In: *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR-16*. Vol. 6355. LNCS. Springer, 2010, pp. 1–12.
[29] Adam Trendowicz and Ross Jeffery. *Software Project Effort Estimation - Foundations and Best Practice Guidelines for Success*. Springer, 2014.
[30] Josef Urban and Geoff Sutcliffe. ATP Cross-Verification of the Mizar MPTP Challenge Problems. In: *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2007*. Vol. 4790. LNCS. Springer, 2007, pp. 546–560.
[31] Elaine J. Weyuker. Evaluating Software Complexity Measures. In: *IEEE Trans. Software Eng.* 14.9 (1988), pp. 1357–1365.