5 Applications of Complexity to Formal Languages

5.1 Types of Complexity Involving Formal Languages

As noted in the previous section, complexity is relative to the kind of difficulty that concerns one most. Frequently this is the limiting factor in a task.

For example, if one has already programmed a computer for a certain search task, the remaining difficulty is represented by the time and memory this search will take to complete. If, further, one is in the happy circumstance where the speed and capacity of computers is growing exponentially (as it has been in the last few decades), then a critical factor will be whether the time and memory requirements also grow exponentially with the size of problem, or merely in a polynomial fashion. If the later is true, one will only have to wait, for the problem to become tractable (assuming the current trend in computing power continues), while this is not the case, the program is likely to remain intractable using this algorithm. This sort of question is dealt with by the measures of Computational Complexity (see section 8.47 on page 162).

Similar examples of possible limiting factors include the size of matrix required for discriminating the independence of expressions (section 8.40 on page 158), the ease with which expressions can be compressed (section 8.2 on page 136), the length of proofs (section 8.19 on page 146), the predictive difficulty of modelling using regular languages and a language's position in the Chomsky hierarchy [122].

One I wish to concentrate on is that corresponding to the difficulty in analysing expressions in a formal language, which I will call the "analytic complexity" of an expression. In order to analyse an expression, one tries to trace its roots, i.e. to decompose it according to some system. The language within which one attempts this will be critical. For example some logical theorems may be quite simple to decompose as an expression but difficult to decompose in terms of proof, for other theorems the opposite might be the case.

Such an analytic process can be seen as a possible strategy for *deducing* the overall properties of a model or expression. In this light, analysis by decomposition is complementary to *inducing* the properties by seeking a model for there properties. In other words the ease with which a model can be decomposed reflects that model's accessibility to analysis and in this way obviates the need for a lengthy inductive search process. So in situations where analysis is feasible it becomes the critical factor in the search for an overall formulation of its behaviour.

5.2 Expected Properties of “Analytic Complexity”

Firstly I will look at what properties one might expect of such an “analytic complexity” in the context of formal languages.

5.2.1 Independent of the particular symbols used

A measure of complexity on statements in a formal language should be independent of the particular symbols used; it is the pattern of the expression that encodes its meaning. It should be immaterial whether you write “a∧b” or “statement1 and statement2” if the corresponding formal languages are isomorphic.

5.2.2 The complexity of sub-expressions should be less than the whole

That a sub-system or sub-expression should be no more complex than the system or expression it is part of, is perhaps the most commonly accepted property of complexity. It is rare that a proposed measure of complexity does not obey this rule (but Abstract Computational Complexity is a rare counter-example, section 8.1 on page 136).

If one considered that, in some circumstance, an expression was less complex than one of its parts then, presumably, there would be some mechanism for preventing this “buried” complexity from being fully realised in the whole. In this case, only an abstraction of the full expression is being considered, in which case the complete representation of this abstraction would be different from the complete expression. Sometimes this is because of a confusion between the syntactic content of an expression and some other aspect.

So, for example, in a first-order logic with equality one might consider all substitution instances of identity equally simple because from a proof theoretic view they perform a similar role, despite the fact that some of these instances seem to (syntactically) contain arbitrarily complex sub-expressions. Either for their purposes they are considering all such instances as the *same* as identity, in which case a different language (where this identity is enshrined) is essentially being used, or they are considering the expression from within the language of its proof theory. In either case the language used is not the syntax of the expressions (over which the sub-expressions are being taken) but the syntax of some proof theory, so it is not surprising that it does not have these complex parts when expressed in these ways.

5.2.3 Expressions with no repetitions are simple

Since I am, for the moment, considering complexity over a potentially wide range of languages I need to define what I mean by atomic statements. By atomic statements I mean expressions that have no constituent syntactic parts (in that language). Of course it is possible that in some systems there are no atomic statements (see section 4.1.2 on page 76).

Atomic expressions are as *syntactically* simple as you can get. It is reasonable to allocate them a minimal complexity. Similarly, in a proof theory, a one line proof just listing an axiom would be a minimally complex proof.

One might object that syntactically atomic symbols may have complex meanings, but if we were to attempt to *measure* the semantic complexity of models, we would need to formalise this in a language so as to distinguish the relevant semantic properties. The "semantic complexity" would be relative to this new language.

Expressions with unrepeated atomic symbols, do not *use* the reference denoted by such a symbol, because for there to be any meaning one needs somehow to establish an identity indicated by a symbol and then make some statement about it. If the only reference by a symbol is made once then that reference specifies its sole property (for the expression), there can not be anything more. It is like answering the question "Who wrote the Iliad?" with the name "Homer", but if the *sole* property of the name "Homer" is that it is the author of the Iliad then it does not get you very far. If we knew anything else about Homer (for instance that he was a single historical being[45]), then the appellation could be meaningfully used, and some complexity appear.

Of course, if one is talking about a language which relates *groups* of sub-expressions, then the symbol may be meaningfully repeated elsewhere (in another sub-expression) to establish its purpose. Logical constants in various proof theories have this role. Again, it is necessary carefully to distinguish between the complexity of expressions within a syntax and the complexity of proofs within the syntax of a proof theory.

45.I am told that this is unlikely.

For example, an expression with no repetitions, like “When at a constant pressure, heat causes gases to expand.”, may have a complexity within a language of scientific verification, as the definitions of the words are all established elsewhere. If, on the other hand, this sentence was used where none of the words had a definition or defining context, it could, on its own, perform no complex role.

5.2.4 Small size should limit the possible complexity

In *very* small systems, there might be a limited number of different relationships between its components. In this case, one would expect that there would be a limit to its complexity. However, the complexity of a system can grow extremely quickly with its size, for example there is a Turing machine with only five internal states that halts after exactly 23,554,768 steps [303].

5.2.5 There should be no upper limit to complexity if the language is suitably generative

In most situations one feels that there is a limit to how far one can simplify statements, but it is always possible to complicate them. The whole point of a generative language is that one can construct an expression as complex as necessary. So it would be very surprising if there was an general upper complexity limit for expressions in such a language.

If there were such a limit, what could it mean? One possibility is that although the expressions may appear more complicated they were not “in fact” so, but in this case we are no longer dealing with the expression itself but that expression from within a different language context represented by the meaning of “in fact”. For example, there is an upper limit to how Algorithmically Complex (section 8.2 on page 136) you can prove strings to be from *within* a wide range of formal systems but this does not mean there was a limit to the Algorithmic Complexity of strings. On the contrary, most strings have an Algorithmic Complexity close to their length; it is a limitation on the power of proofs within the systems that is at the root of this limit.

Another possibility could arise if the expression's syntax were limited in some way so as impose an upper limit on its possible complexity. If such a language was generative it would have to generate an infinite number of expressions of limited complexity. Thus most of these expressions would have to not get more complex as they increased in size.

This would make for an odd language indeed, stuffed with unwieldy but simple expressions. They do exist. An example is a language with an infinite supply of atomic symbols and a connective “→”, where only atomic symbols and only expressions of the form x→y are allowed where x and y share no atomic symbols. In most general expressive languages I would not expect this to be the case.

5.2.6 The complexity of irrelevant substitutions

One way of dealing with complex systems is to try and break them down into a number of simpler systems. If they are not broken down into *independent* sub-systems, there is a risk of losing some aspects of the original system in the process. If all the subsystems are mutually irrelevant to each other, then nothing will have been lost in the analysis. In this case the complexity of the whole system would be wholly in these sub-systems, for there is no other interaction between them at the system level.

In particular, if a sub-expression is substituted for an atomic symbol into another expression and this sub-expression is irrelevant to rest of the main expression, then the resulting expression could be completely and successfully analysed into those two parts. Thus we would think of $(a \vee \neg b) \rightarrow (a \vee \neg b)$ as a substitution instance of identity $x \rightarrow x$ with $(a \vee \neg b)$ substituted for x; in some systems the two levels would not syntactically interact. In the example the complexity of $(a \vee \neg b) \rightarrow (a \vee \neg b)$ derives completely from the complexity of $x \rightarrow x$ and the complexity of $(a \vee \neg b)$.

Here we would have to be careful to separate out the complexity of the expression's syntax from the complexity of its proof theory. If, in the logic's proof theory, $(a \vee \neg b)$ and $b \rightarrow a$ were inter-derivable and interchangeable then there would be some interaction between the levels in the sense that it would mean that an implication could imply another implication (itself).

There might well be a close connection between the complexity of an expression's syntax and its derivation (or theory) in a particular logic, but this is not necessarily the case. For example, in some inconsistent systems of Logic, the atomic proposition, *a*, might be derivable after a lot of work and so it might not have a simple derivation.

Another example is a system of differential equations. Some such systems are linearly separable, in which case the solutions can found separately for distinct variables. The variables do not interact, except that they both occur in this set of equations. You can meaningfully talk about each separately without going into the details of their solutions and add the complexity back again by substituting the solutions later, without invalidating any of the previous discussion.

5.2.7 The complexity of relevant relating of expressions

When two relevant expressions are related then this relation is more complex than either of the parts. The sub-expressions A and $(B \to B) \to A$ are relevant to each other. The joined expression $((B \to B) \to A) \to A$ is clearly at least as complex as either of the sub-expressions it contains (by the sub-system property) and the top-level implication could add to this complexity.

In a densely connected system, where every part is relevant to every other, it is difficult to reduce the system to simpler subsets of the original, without losing important information. A popular way of expressing this is by saying that the whole is greater than the sum of its parts.

Imagine a very incestuous party, where everybody has known everybody else for a very long time. Here it is not possible to gain a complete picture of the system of relationships by studying sub-groups. Every time study is restricted to a subgroup of the total information is lost about some of the relationships which effect those in the subgroup. Conversely recombining these subgroups into the whole again is accompanied by a regaining of this lost complexity.

The simplest party is one where nobody interacts with anybody else. Such parties are no fun at all; nothing can come out of them. Everybody understands exactly what is going on.

A normal party is amenable to *some* analysis into expressions for sub-groups, classes, etc. but there are usually enough cross connections to rule out any *complete* understanding of the situation. We are thus sometimes surprised by their outcome (maybe this is why we have them).

5.2.8 Decomposability of expressions

Many of the above criteria are concerned with the decomposability of formula. Thus we will see that such decomposability is strongly related to analytic simplicity, as the easier it is to decompose without loss, the easier it is to analyse.

A statement in a formal language will be called decomposable if some coherent part of it (larger than a single sign) can be substituted for a symbol so that that part is irrelevant to the resulting statement. Thus the decomposability of a statement will be dependant on the syntax of the language and the definition of irrelevance (or relevance). The idea is that if such a substitution is possible then the two parts of the statement can be considered separately, as their only connection is though the symbol that now stands for the substituted part.

Relevance too will be defined relative to the syntax we are considering. I will take two statements to be irrelevant to each other if they do not contain any of the same sub-formula (in that syntax or in sub-syntaxes). This is a conservative definition of irrelevance; no doubt many statements (or examples of reasoning) that do share some of the same sub-structures could also be considered irrelevant to each other.

Thus a formula that can be repeatedly decomposed as described above will be considered simple compared to one that can not. This reflects our intuitive feeling of the complexity of such formula.

For example, the formula $((a \to b) \to c) \to ((a \to b) \to c)$ is considered as identity $x \to x$ with the sub-formula $(a \to b) \to c$ substituted for *x*. On the other hand the formula $((a \to b) \to c) \to ((b \to c) \to a)$ can not be simplified in the same manner.

I have argued that an number of constraints should apply to the complexity of formulas: a formula with no repetition of its sub-formulas is ultimately simple (I will refer to such formulas as “simple”) - it can be completely decomposed into single symbols; if a formula can be irrelevantly decomposed into parts then its complexity is just the sum of these parts; and that a formula must necessarily be at least as complex as any of its sub-formulas.

There will be formulas that can not be decomposed at all in this manner (in the same syntax). I will call such formulas “complex”. Thus I still have the question of how I compare such complexes with respect to their complexity. One way would be to call all such complexes equally complex (producing something akin to a discrete metric space), but this would not reflect our intuitions very well; identity would be as complex as suffixing!

5.3 Measures of Analytic Complexity

The above properties (listed in Section 5.2 immediately above), provide constraints on possible numeric measures of complexity. The characterisation of measures which meet these constraints and some simple results are now considered. I am not concerned with finding a set of axioms that show the existence and uniqueness of a measure but rather the converse – given the above properties I wish to find what measures exist which satisfy them. This reflects my stance that complexity orderings are not innate but are rather models of aspects of our descriptions to reflect the difficulty we have in finding or analysing them.

A practical result of this is that often the assignment of numbers to reflect the complexity comes before a complexity ordering. Occasionally these assignments are intended to give an indication of the exact level of difficulty (as in storage space), sometimes they are intended to merely give an indication of the order of magnitude of the difficulty, but often they are only intended to induce a complexity ordering of expressions. Thus the measures fall somewhere between the ‘ratio’ and ‘ordinal’ measures as categorised by Stevens in [423].

A decision to model the difficulty of finding or analysing a model description by complexity *measure* (i.e. a homomorphism into the reals) relies on the implicit assumption that the domain is sufficiently constrained so that it is meaningful to assume that a complexity ordering relation will be connected (that is for every pair of items x, y either $x \leq y$ or $y \leq x$). Usually this is entailed by the numerical definition of complexity on the whole domain of possible expressions.

5.3.1 Notation

Let $X \equiv_{df} X_0 \cup X_1 \cup X_2$, be a set of symbols, where X_0, X_1 and X_2 are disjoint sets. It is intended that X_0 be a set of constants and variables, X_1 be a set of unary symbols and X_2 be a set of binary connectives[46].

46.Although I only consider up to binary symbols here, the results could easily be extended.

Let $\boldsymbol{L}$ be a context-free language with the obvious production rules:

$c \in X_0 \Rightarrow c \in \boldsymbol{L}$

$u \in X_1, x \in \boldsymbol{L} \Rightarrow (ux) \in \boldsymbol{L}$

$b \in X_{2,}\ x, y \in \boldsymbol{L} \Rightarrow (bxy) \in \boldsymbol{L}$.

The maximum depth of a formula, $\mathrm{depth}(x)$, is defined recursively in the normal way:

$x \in X_0 \Rightarrow \mathrm{depth}(x)=0$

$\mathrm{depth}(ux) = 1+\mathrm{depth}(x)$

$\mathrm{depth}(bxy) = 1+\max\{\mathrm{depth}(x), \mathrm{depth}(y)\}$

Similarly with the size of a formula, $|x|$:

$x \in X_0 \Rightarrow |x|=1$

$|ux| = |x| + 1$

$|bxy| = |x| + |y| + 1$

Let $\wp(x)$ be the set of well formed subformula of x. Note that $x \in \wp(x)$.

Thus x is a subformula of y iff $x \in \wp(y)$. This is a reflexive and transitive relation.

Define $R(x,y)$ to be a minimal syntactic relevance relation on $\boldsymbol{L}$ thus:

$$R(x,y) \ \equiv_{df} \ \wp(x) \cap \wp(y) \neq \varnothing, \qquad \text{(Defn R)}$$

i.e. x and y share a common subformula. This is equivalent to sharing a common variable or constant. This is a symmetric and reflexive relation.

I will assume in all below that there is a sufficient supply of symbols in X_0, so that one can always find a symbol that is irrelevant to any particular formula. Formally:

$$x \in \boldsymbol{L} \Rightarrow \exists c \in X_0, \neg R(x,c). \qquad \text{(Suff Symbols)}$$

If there is not add a suitable sequence of irrelevant symbols to $\boldsymbol{L}$[47].

Let $x^y/_z$ (or sometimes $x(y/z)$) be the notation for the formula x where every instance of y in x is replaced by z. If $y \in X_0 \cap \wp(x)$ and $\neg R(x,z)$, $x^y/_z$ will be called an *irrelevant*

47.There are two ways of doing this: by adding a infinite sequence of symbols into X_0 or a sufficient but finite supply of symbols to $\boldsymbol{L}$ without the production rules applying to them.

substitution, as the formula z is irrelevant to the formula it is being substituted into. If the formula being substituted, z, is not a member of X_0 and the formula being substituted for is a proper subformula of the formula it is being substituted into – I will call this a *non-trivial irrelevant substitution*. This will form an important part of the theory.

A *complex* is a formula that can not be decomposed using a non-trivial irrelevant substitution.

i.e. x is a complex if

$$\neg\exists y\in \wp(x)-\{x\}-X_0;\neg R(x^y/_c,y), \text{ where } c\in X_0-\wp(x). \qquad \text{(Defn } \boldsymbol{Cp}\text{)}$$

where the sufficiency of symbols assumption above (Suff Symbols) ensures that such a c can be found. The idea is that if $x\notin \boldsymbol{Cp}$ then there is a proper subformula, y, (that is not a member of X_0), so that x can be thought of as the join (by substitution) of two, mutually irrelevant parts: $x^y/_c$ and y.

Let the set of all complexes in $\boldsymbol{L}$ be called $\boldsymbol{Cp}$. Note that trivially all members of X_0 are complexes since $\wp(x)-\{x\}$ is empty.

Following Krantz et al. in [271] I take a *measure* on $\boldsymbol{L}$ to be a homomorphism from a structure on $\boldsymbol{L}$ (e.g. $<\boldsymbol{L}, \oplus, \leq>$) into a suitable structure on the reals (e.g. $<\Re^+, +, \leq>$), where $\oplus$ is some concatenation operator defined on $\boldsymbol{L}$, and $\leq$ is an ordering relation on $\boldsymbol{L}$. For the structures given immediately above this is a function $C:\boldsymbol{L}\rightarrow\Re^+$, defined on $\boldsymbol{L}$ into the non-negative real numbers such that, for all $x,y\in \boldsymbol{L}$:

$$C(x\oplus y) = C(x) + C(y)$$

$$x\leq y \Leftrightarrow C(x)\leq C(y).$$

I will use '≤' for both the ordering on $\boldsymbol{L}$ and the normal ordering on the reals. Which I am referring to should be clear from the content. Similarly I will write $x<y$ where $x\leq y$ but not $y\leq x$ and $x\sim y$ if $x\leq y$ and $y\leq x$.

In all below assume $x,y,z\in \boldsymbol{L}$ and $c,k\in X_0$.

5.3.2 Weak complexity measures

There is no natural and general concatenation operator on $\boldsymbol{L}$ since, in general, how and where expressions are joined matters (due to the tree structure of members of $\boldsymbol{L}$). Substitution is a general and natural operation, but this is essentially a ternary operation.

However, there is a special case of substitution that can act in a similar way to a concatenation operator, this is the case of irrelevant substitution. This corresponds to the intuitions discussed in section 5.2.6 on page 90 above. This concatenation would be defined:

$x \oplus y = x^{c}/_{y}$, where $c \in X_0 \cap \wp(x)$, $\neg R(x,y)$

but clearly $x \oplus y$ is not uniquely defined as an operation in $\boldsymbol{L}$ since there can be more than one $c \in X_0 \cap \wp(x)$. However the intention is that the complexity of the result of *any* irrelevant substitution for a member of X_0 will be the sum of the complexities of the original formula and the formula substituted into it. In this way irrelevant substitution performs a similar role to concatenation with respect to a *complexity* measure. For clarity and to ensure that the measures are well defined I will retain the use of explicit substitution (e.g. $x^{c}/_{y}$) within $\boldsymbol{L}$ (instead of $\oplus$, which I will reserve for use as an implicit relation as in $x \oplus y = z$, by which I will mean that z is the result of an irrelevant substitution of y into x). The idea of a measure will remain that of a homomorphism between + in $\Re^{+}$ and any irrelevant substitution in $\boldsymbol{L}$.

Translating the above formalisation of a measure using the substitution operation we get: a *weak complexity measure* is a homomorphism from $<\boldsymbol{L}, \oplus, \leq>$ into $<\Re^{+}, +, \leq>$, that is a function $C:\boldsymbol{L} \rightarrow \Re^{+}$, defined on $\boldsymbol{L}$ to the non-negative real numbers such that, $\forall x,y \in \boldsymbol{L}$:

$c \in X_0 \cap \wp(x), \neg R(x,y) \Rightarrow C(x^{c}/_{y}) = C(x) + C(y)$ (Irrel Subs)

$x \leq y \Leftrightarrow C(x) \leq C(y)$. (Subform)

To start with I will impose no constraints upon the ordering relation on $\boldsymbol{L}$ (other than those implied by Irrel Subs).

Some simple results

Irrel Subs implies that the measure of all nullary symbols (X_0) to be zero, that is:

$c \in X_0 \Rightarrow C(c)=0$ (Nullary Zero)

Proof:

Choose $k \in X_0$, $k \neq c$

$C(k) = C(c^c/_k)$ as $c^c/_k = k$

$= C(c) + C(k)$ by Irrel Subs

....

This means that a complexity measure cannot be a positive measure, i.e. it is *not* true to say that:

$x \oplus y > x.$

An alternative form of irrelevant substitution is sometimes useful:

$y \in \wp(x), \neg R(x^y/_c, y), c \in X_0 - \wp(x) \Rightarrow C(x^y/_c) = C(x) - C(y)$ (Alt Irrel Subs)

Proof:

$C(x) = C(x^y/_c{}^c/_y)$ as $x = x^y/_c{}^c/_y$

$= C(x^y/_c) + C(y)$ by Irrel Subs.

....

If x and y are essentially the same formula but with different X_0 symbols (I will write $x \approx y$) then they have the same complexity (as argued for in section 5.2.1 on page 87 above).

Formally this is equivalent to repeated applications of:

$c \in \wp(x) \cap X_0, k \in X_0 - \wp(x), \neg R(x,k) \Rightarrow C(x^c/_k) = C(x)$ (Symbol Change)

Proof:

$C(x^c/_k) = C(x) + C(k)$ by Irrel Subs

$= C(x)$ by Symbol Zero.

....

Decomposition (Decomp)

The first result, suggested by Irrel Subs, is that any formula can be decomposed into complexes. i.e. for any $x \in \boldsymbol{L}$ there are $a_0,\ldots,a_n \in \boldsymbol{Cp}$ and $c_1,\ldots,c_n \in X_0$ such that

$$x = a_0{}^{c_1}/a_1\ldots{}^{c_n}/a_n$$

and for $i<n$, $\neg R(a_0{}^{c_1}/a_1\ldots{}^{c_i}/a_i, a_{i+1})$ (so the above could be written as the relation: $x = a_0 \oplus \ldots \oplus a_n$

and so

$$C(x) = C(a_0) + \ldots + C(a_n).$$

Note that this decomposition is not necessarily unique.

<u>Proof:</u>

By simple induction on length of x using a non-trivial irrelevant substitution when $x \notin \boldsymbol{Cp}$. See section 9.3 on page 169 for this.

....

The length of a decomposition is defined as the number of irrelevant decompositions that are required (i.e. n in the above).

A weak complexity measure is determined by its values on the complexes, $\boldsymbol{Cp}$. That is, a function, g, on $\boldsymbol{Cp}$ in $\boldsymbol{L}$, $g{:}\boldsymbol{Cp} \to \Re^+$, such that:

$$x \in X_0 \Rightarrow g(x)=0,$$

$$x \approx y \Rightarrow g(x)=g(y)$$

$$x \leq y \Rightarrow g(x) \leq g(y).$$

will generate a unique complexity measure, $C{:}\boldsymbol{L} \to \Re^+$, on $\boldsymbol{L}$ via:

$$C(x) \equiv_{df} g(a_0) + g(a_1) + \ldots + g(a_n),$$

for some decomposition: $x = a_0{}^{c_1}/a_1\ldots{}^{c_n}/a_n$, into complexes: $a_0,\ldots,a_n \in \boldsymbol{Cp}\text{-}X_0$, $n \geq 0$, as above (unless $x \in X_0$, in which case it is its own decomposition) and hence extend the ordering, $\leq$, from $\boldsymbol{Cp}$ to the whole of $\boldsymbol{L}$, via:

$$x \leq y \Leftrightarrow C(x) \leq C(y).$$

<u>Proof:</u>

Proof is by induction on the maximum size of formulas. In the induction step there are two things to prove: that C is well defined and that the $\mathrm{Irrel\ Subs}$ condition holds. The first is shown using a lemma that one of $\neg R(x,y)$, $y \in \wp(z)$ or $z \in \wp(y)$ holds. The second then comes out as a consequence of decomposition into complexes. Details can be found in section 9.4 on page 170.

....

Of, course the reverse is trivially true: a weak complexity measure on ***L*** results in a generating function as above by simple restriction of its domain to ***Cp***. Thus there is a one-to-one correspondence between weak complexity measures and such functions on the complexes of ***L***.

I can now demonstrate the existence of (non-trivial) weak complexity measures by generating such a measure from the function $g{:}\boldsymbol{Cp} \rightarrow \Re^{+}$ defined:

$x \in X_0,\ u \in X_1 \Rightarrow g(ux) \equiv_{df} 1,$

Otherwise $\Rightarrow g(x) \equiv_{df} 0.$

This generates a complexity measure which “counts” the number of instances of unary symbols in a formula that can be decomposed out by a non-trivial irrelevant substitution. So $C(c)=0$, $C(uc)=1$, $C(u(uc))=2$, $C(u(u(uc)))=3$, $C(u(u(u(uc))))=4$ etc. but $C(bc(u(u(u(uc)))))=0$. Thus by simply increasing the size of the formula in this uninteresting way we can increase its complexity – as discussed above (section 3.4.1 on page 57) this is not very satisfactory,

5.3.3 Weak complexity measures where simple repetition does not increase complexity

I now add a condition so as to distinguish more sharply weak complexity measures from those concerned with size, by ruling out measure such as the example above. The extra conditions are:

$c,k \in X_0$, $c \neq k$, $u \in X_1$, $b \in X_2$

$uc \leq c$ (Unary Zero)

$bck \leq c$ (Binary Zero)

Thus

$C(uc) \leq C(c) = 0$ and $C(bck) \leq C(c) = 0$ so $C(uc)=C(bck)= 0$. (Zero)

This has the consequence that the complexity of an irrelevant join of expressions using a binary connective is the sum of the complexities of its parts:.

$\neg R(x,y) \Rightarrow C(bxy) = C(x) + C(y)$ (Irrel Join)

Proof:

Choose $c,k \in X_0$ such that $\neg R(x,c)$, $\neg R(x,k)$, $c \neq k$ (Suff Symbols)

Then

$C(bxy) = C(bx(c^c/_y))$

$= C((bxc)^c/_y)$ as c can not occur in x

$= C(bxc) + C(y)$ by Irrel Subs

$= C(b(k^k/_x)c) + C(y)$

$= C((bkc)^k/_x) + C(y)$ $c \neq k$

$= C(bkc) + C(x) + C(y)$ by Irrel Subs

$= C(x) + C(y)$ by Zero.

....

Similarly

$$C(ux) = C(u({}^{c}/_{x})) = C((uc)^{c}/_{x}) = C(uc) + C(x) = C(x)$$

By simple induction we can see that any formulas where there are no repetitions of the X_0 symbols that occur in it then it is has zero measure. Thus the above conditions meet the requirement that expressions with no repetitions are simple that resulted from the arguments in section 5.2.3 on page 88.

As above, a weak complexity measure under the extra 'zeroing' conditions is determined by its values on the complexes, ***Cp***.

A function, g, on ***Cp*** in ***L***, g:***Cp***$\rightarrow\Re^{+}$, such that:

$x,y\in X_0,\ u\in X_1,\ b\in X_2 \Rightarrow c(x)=0,\ c(ux)=0,\ c(bxy)=0$

$x\approx y \Rightarrow g(x)=g(y)$

$x\leq y \Rightarrow g(x)\leq g(y)$.

will generate a unique complexity measure, C:***L***$\rightarrow\Re^{+}$, on ***L*** via:

$$C(x) \equiv_{df} g(a_0) + g(a_1) + \ldots + g(a_n),$$

for some decomposition: $x = a_0{}^{c_1}/a_1\ldots{}^{c_n}/a_n$, into complexes: $a_0,\ldots,a_n\in$ ***Cp***-X_0, $n\geq 0$, as above (unless $x\in X_0$, in which case it is its own decomposition) and hence extend the ordering, $\leq$, from ***Cp*** to the whole of ***L***, via:

$$x\leq y \Leftrightarrow C(x)\leq C(y).$$

Proof:

Proof is identical to the generation theorem above, except for the extra condition on the generating function, g, which transfers directly to the resulting complexity measure, C – details of the previous proof can be found in section 9.4 on page 170.

....

An example of this type of measure is generated from the function g:***Cp***$\rightarrow\Re^{+}$ thus:

$x\in X_0,\ b\in X_2 \Rightarrow g(bxx) \equiv_{df} 1$.

Otherwise $g(x) \equiv_{df} 0$.

This produces a complexity measure which "counts" the number of instances of complexes of the form bcc that can be decomposed out by a non-trivial irrelevant substitution. Here $C(c)=0$, $C(bcc)=1$, $C(bc(bcc))=0$, but $C(bk(bcc))=1$ (when $c \neq k$).

5.3.4 Weak complexity measures that respect the subformula relation and where simple repetition does not increase complexity

There is a natural constraint on orderings of $\boldsymbol{L}$, namely that of the subformula relation (as discussed in section 5.2.2 on page 87). So next I consider measures from $<\boldsymbol{L}, \oplus, \leq>$ into $<\Re^+, +, \leq>$, that is a function $C:\boldsymbol{L} \rightarrow \Re^+$, defined on $\boldsymbol{L}$ to the non-negative real numbers such that, $\forall x,y \in \boldsymbol{L}$:

$$c \in X_0 \cap \wp(x), \neg R(x,y) \Rightarrow C(x^c/_y) = C(x) + C(y) \qquad \text{(Irrel Subs)}$$

$$x \leq y \Leftrightarrow C(x) \leq C(y). \qquad \text{(Subform)}$$

where $x \in \wp(y) \Rightarrow x \leq y$ as well as $bck, uc \leq c$.

All the above results still hold except the generation of measures from a function on $\boldsymbol{Cp}$, which needs to be adapted.

Such a weak complexity measure is determined by its values on the complexes, but now we do not have as much freedom to allocate values on $\boldsymbol{Cp}$. So now a function, g, on $\boldsymbol{Cp}$ in $\boldsymbol{L}$, $g:\boldsymbol{Cp} \rightarrow \Re^+$, such that:

$$x,y \in X_0,\ u \in X_1,\ b \in X_2 \Rightarrow c(x)=0,\ c(ux)=0,\ c(bxy)=0$$

$$x \approx y \Rightarrow g(x)=g(y)$$

$$p_0{}^{c_1}/p_1 \ldots {}^{c_n}/p_n \in \wp(q)-\{q\}$$
$$\Rightarrow \Sigma_i g(p_i) \leq g(q)$$

will generate a unique complexity measure, $C:\boldsymbol{L} \rightarrow \Re^+$, on $\boldsymbol{L}$ via:

$$C(x) \equiv_{df} g(a_0) + g(a_1) + \ldots + g(a_n),$$

for some decomposition: $x = a_0{}^{c_1}/a_1 \ldots {}^{c_n}/a_n$, into complexes: $a_0,\ldots,a_n \in \boldsymbol{Cp}-X_0$, $n \geq 0$, as above (unless $x \in X_0$, in which case it is its own decomposition) and hence extend the ordering $\leq$ to the whole $\boldsymbol{L}$, via:

$$x \leq y \Leftrightarrow C(x) \leq C(y).$$

Proof:

Again, the proof is identical to that in section 9.4 on page 170, except that we have to check that the subformula constraint on $\leq$ in $\boldsymbol{L}$ holds:

Now if $y \in \wp(q)-\{q\}$, $q \notin \boldsymbol{Cp}$, then by the decomposition theorem above $y=p_0{}^{c_1}/p_1{}^{c_2}/p_2\ldots{}^{c_n}/p_n$ for some $p_0, p_1, \ldots, p_n \in \boldsymbol{Cp}$.

Now, $C(q)$

$= h(q)$ by construction

$\geq \Sigma_i g(p_i)$ by (3)

$= C(y)$ by construction

....

An example of this type of measure is generated from the function $g:\boldsymbol{Cp} \rightarrow \Re^+$ thus:

$x,y \in X_0,\ u \in X_1,\ b \in X_2 \Rightarrow g(x) \equiv_{df} 0,\ g(ux) \equiv_{df} 0,\ g(bxy) \equiv_{df} 0.$

Otherwise $c(x) \equiv_{df} 1$.

This produces a complexity measure which “counts” the number of non-trivial complexes in the decomposition of a formula. Here $C(c)=0$, $C(bck)=0$ where $c \neq k$, $C(u(ux))=0$, $C(bcc)=1$, $C(bc(bcc))=1$, and $C(b(bcc)(bcc))=2$.

The trouble with the above method of generating a complexity measure is that it is not easy to see what generating functions would satisfy the conditions. The following alternative is a little clearer.

Any non-negative function $h:\boldsymbol{Cp} \rightarrow \Re^+$, recursively generates a strong complexity measure on $\boldsymbol{L}$, thus:

$p,q \in X_0,\ u \in X_1,\ b \in X_2 \Rightarrow C(p)=0,\ C(up)=0$ and $C(bpq)=0$,

$x \in \boldsymbol{Cp} \Rightarrow C(x)=C(a_0) + \ldots + C(a_n)$, where x has a decomposition: $a_0{}^{c_1}/a_1\ldots{}^{c_n}/a_n$,

Otherwise $C(x)=h(x) + \max\{C(y) \mid y \in \wp(x)\}$.

In other words g specifies the “extra” complexity assigned to a complex above the greatest complexity of any of its subformula and the complexity of decomposable formulas is built up in the normal way as the sum of the complexities of the complex in a decomposition.

Thus since any such complexity measure generates a non-negative function and a non-negative function generates such a complexity measure in reverse, there is a one-one correspondence between non-negative functions on the Complexes and these complexity measures.

5.3.5 Strong complexity measures

Relevant Join

In the above measures what I call the ‘relevant join’ property does not necessarily hold. This property is:

$R(x,y) \Rightarrow bxy > C(x),\ bxy > C(y)$

This deficiencies will be corrected with the setting up of “strong complexity” measures below. Firstly I prove that the following three conditions are equivalent on a weak complexity measure, in particular the version in (section 5.3.3 on page 100), for $b \in X_2$, $x,y \in \boldsymbol{L}$:

(i) $R(x,y) \Rightarrow C(bxy) > C(x),\ C(bxy) > C(y)$ (Rel Join)

(ii) $y \in \wp(x)-\{x\},\ c \in X_0-\wp(x),\ R(x^y/_c,y) \Rightarrow y < x$ (Rel Subform)

(iii) $y \in \wp(x)-\{x\}-X_0,\ x \in \boldsymbol{Cp} \Rightarrow y < x$ (Subform of Complex)

<u>Proof:</u>

First I show that condition (i) implies the subformula property, then prove in turn (i)⇒(ii); (ii)⇒(i); (ii)⇒(iii); and finally (iii)⇒(ii). This proof can be found in section 9.5 on page 175.

....

Note that (as a result of the proofs in section 9.5 on page 175) any of the three conditions above implies the subformula property, namely:

$y \in P(x) \Rightarrow y \leq x.$

I can now define a *strong complexity measure* as a weak complexity measure together with any of the three constraints above.

Again, in an almost identical way as above, we can generate strong complexity measures from measures on $\boldsymbol{Cp}$, the set of complexes.

A function, g, on ***Cp*** in ***L***, $g:\boldsymbol{Cp}\to\Re^{+}$, such that for all $p,q\in \boldsymbol{Cp}$:

(1) $p,q\in X_0 \Rightarrow g(p)=0$, $g(up)=0$ and $g(bpq)=0$

(2) $p\approx q \Rightarrow g(p)=g(q)$

(3) $p_0{}^{c_1}/p_1\ldots{}^{c_n}/p_n\in \wp(q)-\{q\}$

$\Rightarrow \Sigma_i g(p_i) < g(q)$

will generate a unique complexity measure, $C:\boldsymbol{L}\to\Re^{+}$, on ***L*** via:

$C(x) \equiv_{df} g(a_0) + g(a_1) + \ldots + g(a_n)$,

for some decomposition: $x = a_0{}^{c_1}/a_1\ldots{}^{c_n}/a_n$, into complexes: $a_0,\ldots,a_n\in \boldsymbol{Cp}-X_0$, $n\geq 0$, as above (unless $x\in X_0$, in which case it is its own decomposition) and hence extend the ordering $\leq$ to the whole ***L***, via:

$x\leq y \Leftrightarrow C(x)\leq C(y)$.

Proof:

Again, the proof is identical to that in section 9.4 on page 170, except that we have to check that the subformula constraint on $\leq$ in ***L*** holds:

Now if $y\in \wp(q)-\{q\}$, $q\in \boldsymbol{Cp}$, then by the decomposition theorem above $y=p_0{}^{c_1}/p_1\ldots{}^{c_n}/p_n$ for some $p_0,\ldots,p_n\in \boldsymbol{Cp}$.

Now, $C(q)$

$= g(q)$ by construction

$> \Sigma_i g(p_i)$ by (3)

$= C(y)$ by construction

....

Obviously, any strong complexity measure has a unique restriction to ***Cp*** that obeys (1), (2), and (3) above. Thus there is a one-one correspondence between such functions on ***Cp*** and strong complexity measures.

This time any *strictly positive* function on the non-trivial somplexes $h{:}\boldsymbol{Cp}\text{-}\boldsymbol{X}_0 \to \Re^+$, recursively generates a strong complexity measure on ***L***, thus:

$p,q \in X_0$, $u \in X_1$, $b \in X_2 \Rightarrow C(p)=0$, $C(up)=0$ and $C(bpq)=0$,

$x \notin \boldsymbol{Cp} \Rightarrow C(x)=C(a_0) + \ldots + C(a_n)$, where x has a decomposition: $a_0{}^{c_1}/a_1 \ldots {}^{c_n}/a_n$,

Otherwise $C(x)=h(x) + \max\{C(y) | y \in \wp(x)\}$.

Thus since any strong complexity measure generates a strictly positive function and a strictly positive function generates a strong complexity measure in reverse, there is a one-one correspondence between positive functions on the complexes and strong complexity measures.

Perhaps the most straightforward of such functions, h, mentioned immediately above is the constant function $h(x)=1$. This measure captures the maximum length of a chain of stepwise decompositions by (possibly relevant) non-trivial substitutions. This is analogous to the Krohn-Rhodes measure of complexity based on the decomposition of semi-groups (section 8.23 on page 148).

Other strong complexity measures include: the cylomatic complexity of the minimal directed acyclic graph representation of formula (see section 5.4 on page 106 immediately below), and the logarithm of the number of spanning trees of the same representation (see section 8.31 on page 153).

Thus strong complexity measures have all of the properties argued for analytic complexity above in sections 5.2.1 to 5.2.7. This still leaves a lot of choice, to allow the capturing in such measures of different types of difficulty. I now look at one such in more detail.

5.4 The Cyclomatic Number as a Measure of Analytic Complexity

I will define a possible way of comparing the complexities of complexes. Two ways of breaking a complex up are by substituting for a (non-atomic) repeated sub-formula and by breaking the top level connective into the separate parts. For example in seeking to “break up” the formula $(a \to (a \to b)) \to (a \to b)$ we could substitute for $a \to b$ to make $(a \to x) \to x$ and $a \to b$ or break the top implication to make $a \to (a \to b)$ and $a \to b$. This is not a decomposition in the above sense as both methods lose some of the essential structure of the formula (the whole is more than the sum of the parts).

Call the damage inflicted by such a breaking the number of sub-formula the two parts now have in common. Now keep breaking the formula up until all the parts are simple. Call the total damage of such a process the “analytic loss”. Now call the complexity of the original formula the minimum such loss possible by such an “analysis”. Note that here I am seeking to talk about the analytic complexity of the formula relative to its syntax and above definition of irrelevance; I am not talking about the complexity of any particular analytic process.

For example, we could first break $(a \to (a \to b)) \to (a \to b)$ into $(a \to x) \to x$ and $a \to b$ (with a damage of 1) and then $(a \to x) \to x$ broken into $a \to x$ and x (with a damage of 1). Thus the complexity of the formula would be no greater than 2. The only other possibilities are to first break the formula into $(a \to (a \to b))$ and $a \to b$ (with a damage of 3) or substitute for a (which does not progress the “analysis”. Thus the loop complexity of the formula would be 2.

This, of course is quite a coarse account of the damage inflicted in such an analysis. It says nothing of the *sort* of structural information lost in the process.

It turns out that this is bounded below by the cyclomatic number of a directed graph when the formula is represented as a minimal graph. (see section 9.2 on page 169).

For example the formula $(a \to (a \to b)) \to (a \to b)$ can be represented by the following tree in figure 19.

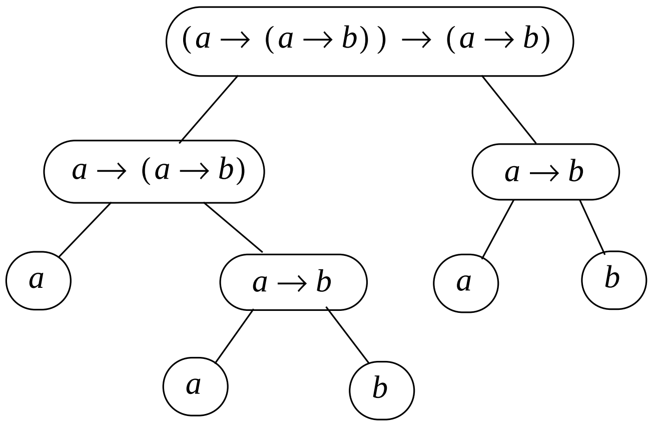


Figure 19. A formula represented as a tree

but this is not minimal. There is a repetition of leaves that can be eliminated by combining the nodes of identical sub-formula to produce the collapsed graph shown in figure 20.

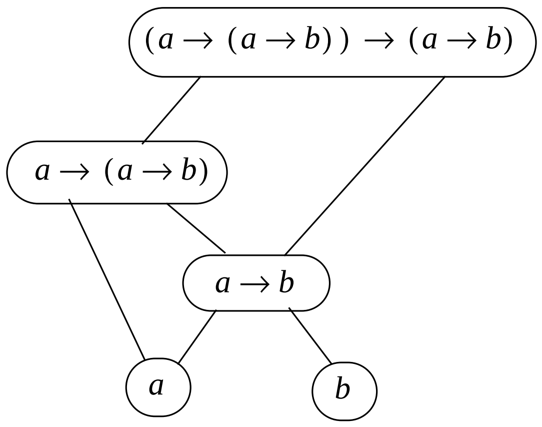


Figure 20. A formula represented as a collapsed tree

The cyclomatic number of this graph is then the number of connecting arcs minus the number of nodes plus one. This gives us a complexity of two. Another way of calculating this is just by counting the number of “independent” closed loops [436].

Thus this definition is consistent with all that I specified above. Simple formula can be broken with no damage and hence have a complexity of zero. If a formula can be decomposed into parts then its complexity is the sum of those parts and the complexity of a formula is at least as great as any sub-formula. See section 5.6.1 on page 113 for a comparison of complexity measures on such formulas.

One advantage of such a measure of complexity is that it is applicable to a wide range of structures and certainly any statement in a wide range of formal syntaxes.

5.5 Layers of Syntax and Complexity

One method frequently used (by humans) to attack such complexes is to change the language they are expressed in. A formula that may be complex in one language might be simple in another. For example, the complexity of a formula in a language’s formula syntax may have no obvious connection with the complexity of a derivation of such a formula as a proof in the proof theory's syntax (see section 5.6 on page 113).

Thus in order to study what is still difficult within this more subtle situation we need to be able to consider varying syntaxes where it is the formula that essentially remains constant. To do this we need a formalism that will encapsulate both the formula syntaxes as well as the syntaxes of proofs. I could do this as a generalised language defined by the production of strings of symbols but this does not clearly preserve for study the distinction between the structure of a syntactic derivation and its result. I want a very clear distinction between the structure of a derivation (like a proof) and its result (a formula) as these belong to *different* syntaxes. Also a linear string is, in many respects, an artificial representation of formal statements, requiring the use of such devices as brackets and parsing. Thus for the purposes of this study I will define a generalised description of a substantial class of syntaxes as tree structures.

I will define a general syntactic structure recursively. Presume there is a set of primitive symbols called P and also a set, V, of variables that will be used in the definition of its generative rules. A syntactic structure, S, is then a pair (X, R) where X is set of sub-syntactic structures (possibly empty) and R is a set of rules of the form:

$$\left((A_i, \ldots, A_n), C, \{ (v_1, X_1), \ldots, (v_m, X_m) \} \right),$$

where the A_i are trees with the nodes labelled with primitive symbols or variables from the syntax, C is the result of the rule, the v_i are variables and the X_i are members of X - the “lower” syntaxes that this one draws on for formulas, symbols etc. In addition there is the restriction that every symbol appearing in C must either have appeared in one of the A_i, be a symbol from P or be one of the v_i.

The idea is that if the $A_1, \ldots, A_n$ occur in the syntax then C will as well, with any occurrences of v_i being replaced with an item from the sub-syntax generated by X_i.

For convenience I will write the rule like this:

$$A_1, \ldots, A_n \gg C, \ (v_1 \in X_1, \ldots, v_m \in X_m) \ .$$

Such a syntactic structure will generate a syntax being the set of all trees recursively constructible from the rules.

I will make this clearer with some examples. In these let the set of primitive symbols, P, include x, →,), ¬ and V include u, v, w.

5.5.1 Example 1 - a supply of variable names

Let S_1 be the syntax $(\varnothing, R_1)$ where R_1 consists of the rules:

N1. $\varnothing \gg x$

N2. $v \gg {}'(v)$, (or $v \gg v'$ when allowing suffix notation).

This generates (in a way that will be specified) the set which could be written as $\{x, x', x'', \ldots\}$ but more pedantically written as $\{N1, N2(N1), N2(N2(N1)), \ldots\}$. This could be used as an infinite supply of pairwise distinct names. There is an important distinction here between $\{N1, N2(N1), N2(N2(N1)), \ldots\}$, which are the structures of the possible derivations and $\{x, x', x'', \ldots\}$ which could be considered as the result of preforming the derivations. This is illustrated in figure 21.

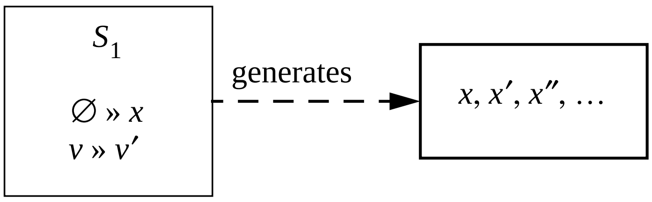


Figure 21. A single layer syntactic structure

In this first example they can be safely conflated as they have a one-to-one structural correspondence, but in the next examples this will not hold.

5.5.2 Example 2 - WFFs of the implication fragment

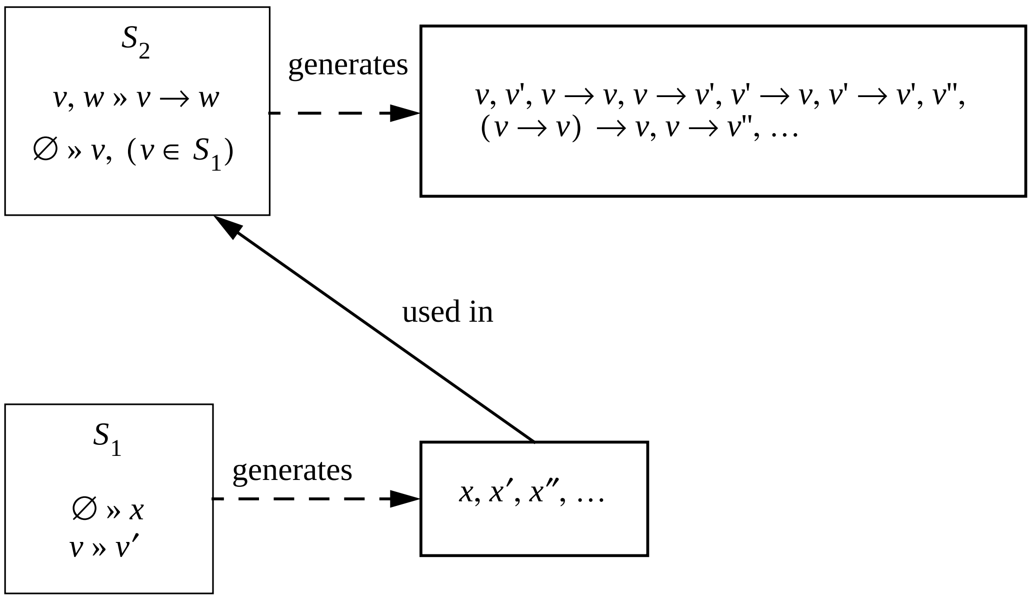
Now let S_2 be the syntax (S_1, R_2), where R_2 consists of the rules

F1. $\varnothing \gg v, \ (v \in S_1)$

F2. $v, w \gg \rightarrow(v, w)$, (or $v, w \gg v \rightarrow w$ using infix notation).

This generates the syntax of pure-implication propositional formulas, which could be written as $\{v, v', v \rightarrow v, v \rightarrow v', v' \rightarrow v, v' \rightarrow v', v'', (v \rightarrow v) \rightarrow v, v \rightarrow v'', \ldots\}$. The first rule allows for the inclusion of trees generated by S_1 and the second the recursive construction of the pure implicational formulas in the normal manner.

Here we have a two-layer system of syntactic structures, illustrated in figure 22.



S_2
$v, w » v \to w$
$\varnothing » v,\ (v \in S_1)$
generates
$v, v', v \to v, v \to v', v' \to v, v' \to v', v'',$
$(v \to v) \to v, v \to v'', \ldots$
used in
S_1
$\varnothing » x$
$v » v'$
generates
$x, x', x'', \ldots$

Figure 22. A two layer syntactic structure

5.5.3 Example 3 - The implicational fragment of E

Now let S_3 be the syntactic structure (S_2, R_3) where R_3 consists of the rules

P1. $\varnothing » (((v \to v) \to w) \to w),\ (v, w \in S_2)$

P2. $\varnothing » ((v \to w) \to ((w \to u) \to (v \to u))),\ (v, w, u \in S_2)$

P3. $\varnothing » (v \to (v \to w)) \to (v \to w),\ (v, w \in S_2)$

P4. $v, v \to w » w$

which will generate a syntax of proofs of pure implicational tautologies. Note that the brackets in rules P1, P2 and P3 are for human convenience only. $((v \to v) \to w) \to w$ really stands for the structure shown in figure 23.

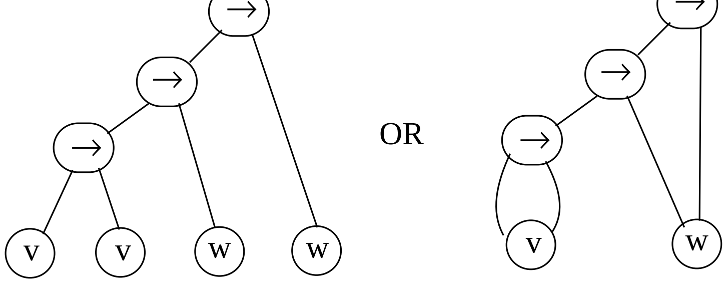


Figure 23. The actual structure of the assertion axiom

In this example we have three levels of syntactic structure for constructing distinct propositional names, formulas and proofs. This is shown in figure 24.

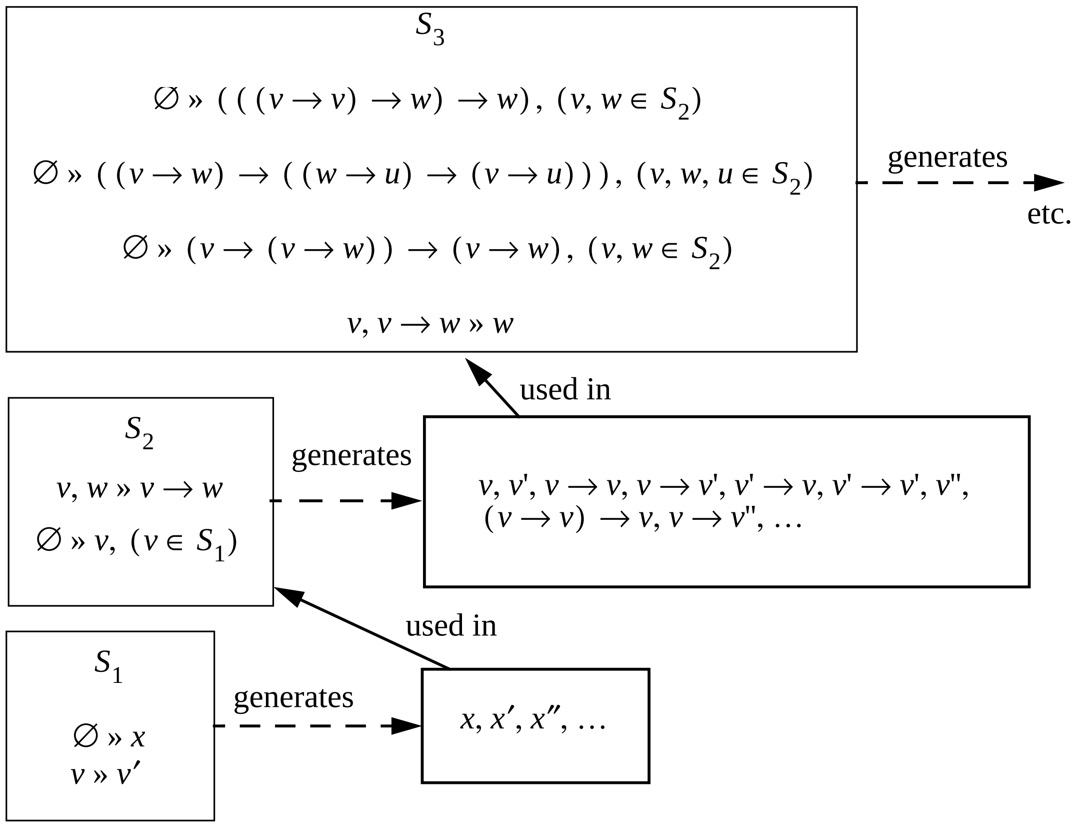

S_3
$\varnothing \gg (((v \to v) \to w) \to w),\ (v, w \in S_2)$
$\varnothing \gg ((v \to w) \to ((w \to u) \to (v \to u))),\ (v, w, u \in S_2)$
$\varnothing \gg (v \to (v \to w)) \to (v \to w),\ (v, w \in S_2)$
$v, v \to w \gg w$
generates
etc.
used in
S_2
$v, w \gg v \to w$
$\varnothing \gg v,\ (v \in S_1)$
generates
$v, v', v \to v, v \to v', v' \to v, v' \to v', v'', (v \to v) \to v, v \to v'', \ldots$
used in
S_1
$\varnothing \gg x$
$v \gg v'$
generates
$x, x', x'', \ldots$

Figure 24. A three layer structure

5.5.4 Discussion of syntactic structures

A syntactic structure generates a syntax, which is the closure under the rules of the empty set. Thus in order to be non-empty at least one rule must have an empty set of antecedents (like the rules N1, P1, F1 or F2 above). I call such rules *leaf nodes*.

There are two kinds of common leaf nodes: those where the consequent is constructed purely from the set of primitive symbols (like rule N1 above) and those whose antecedent is constructed with a substitution from a sub-syntax (like P1, F1 or F2).

Thus the syntax is composed of a set of derivation structures constructed from the rules and specific substitutions of items from the syntaxes of sub-syntactic structures. For example the syntax generated by S_3 would include the proof for $((x \to x) \to x) \to x$ which could be represented as $P1(v \leftarrow F1(v \leftarrow N1), w \leftarrow F1(v \leftarrow N1))$, i.e. the rule

P1 with x substituted for both v and w from the syntax generated from S_2. This x is the rule F1 with the rule N1 substituted from the syntax generated from S_1.

A higher level derivation structure can be 'executed' to produce a result from a lower syntax. Thus the above structure $P1(v \leftarrow F1\,(v \leftarrow N1)\,,\, w \leftarrow F1\,(v \leftarrow N1)\,)$ can be executed to the structure $F2\,(F2\,(\,(F2\,(F1\,(v \leftarrow N1)\,,F1\,(v \leftarrow N1)\,)\,,F1\,(v \leftarrow N1)\,)\,,F1\,(v \leftarrow N1)\,)\,)$ which I would normally write as the string $(\,(x \rightarrow x) \rightarrow x) \rightarrow x$.

In section 10.1 on page 182 in Appendix 3 - Formalisation of Syntactic Structure, I formalise the above notions of syntactic structures and the generation from them. I also show (section 10.2 on page 184) that they have at least the expressive power of a general phrase structured grammar.

In Appendix 4 - A tool for exploring syntactic structures, complexity and simplification, I describe a software tool I developed for exploring such structures. It can deal with a sequence of syntactic structures, where each SS is dependent on the one below. In section 10.3 on page 187 of that appendix I outline a proof that the restriction to such a linear chain does not reduce the overall expressive power of such systems.

5.6 Application to Axioms and Proof Systems

5.6.1 Axiom complexity

Elegance has long been a desirable characteristic in formal mathematics and logic. During the first half of this century this has frequently been associated with parsimony. In particular there was a drive to find formal logical systems with the fewest rules, axioms or variables.

In Hilbert-Frege style axiomatisations [162, 222], the number of rules is reduced to two, substitution (subs) and Modus Ponens (MP)[48], so that the properties of the logic are determined solely by the axioms which thus have a dual role as starting points for proofs and inference tickets (i.e. as the major premiss of an application of an MP rule). This focused attention on the number and length of the axioms.

Many different axiomatisations have been suggested. In 1910, Whitehead and Russell produced an axiomatisation of the disjunction-negation fragment using 5

48.Or just MP if one considers axioms as axiom schemas.

axioms [467] (later one was shown to be a consequence of the others)[49]. All the other connectives were introduced as definitions of these.

This was followed by many other suggestions, many of which favoured parsimony. In 1929 Lukasiewicz produced an axiomatisation of the implication-negation fragment of classical logic using just three axioms[50]:

$$(1)\quad (p \to q) \to ((q \to r) \to (p \to r))\ ;$$
$$(2)\quad (\neg p \to p) \to p\ ;$$
$$(3)\quad p \to (\neg p \to q)\ .$$

In 1949 he exhibited a single axiom system for the pure implicational fragment, proving its minimality [300]:

$$(4)\quad ((p \to q) \to r) \to ((r \to p) \to (s \to p))\ ,$$

followed by several systems by Meredith [320, 321] in the 50’s including a single axiom version of the implication-negation fragment:

$$(5)\quad ((((p \to q) \to (\neg r \to \neg s)) \to r) \to t) \to ((t \to p) \to (s \to p))\ .$$

There was also an investigation into the minimal number of connectives used. In 1918 Nicod exhibited a single axiom with a single connective equivalent to $\neg(p \wedge q)$ involving 11 occurrences of that connective and 5 variables[51].

Similarly there was also a drive to reduce the number of variables used. Lukasiewicz “improved” upon Nicod’s axiom by finding one of the same length but only involving 4 variables[52]. Later Diamond and McKinsey showed that one needed at least one axiom with three variables in it [139] and recently Ulrich proved more about the necessary length of axioms using matrix techniques [448].

Perusing the above it is obvious that mere parsimony does not correspond to any relevant sense of complexity, with respect to such axiomatisations. While Lukasiewicz’s *three* axioms above are amenable to a natural interpretation (versions of transitivity, necessity out and inferring from a contradiction), this is very difficult to do for the single axiom systems. Thus going by the definition of complexity herein the only way in which they could be said to be simpler is due to the difficulty of storage of the axiomatic

49.Frege had already exhibited this type of system [162], but his notation was cumbersome.

50.Listed using the standard infix style notation rather than Lukasiewicz’s.

51.I will not show this, suffice to say it is incomprehensible!

52.As reported by Prior in [362].

description given the logical theory it needs to produce, and the difficulty of storage must be one of the least of one’s problems with such systems.

In appendix 4 (section 12 on page 193) I have tabulated all the propositional formulas in the implication-negation fragment up to those with 6 symbols (not including brackets), sorted by five methods: size, number of variables, depth, breadth and cyclomatic number. All these rankings give credibly low rankings to simple formulas but can differ widely on the rest.

Table 2 on page 193 shows these formulas sorted by size. Here formulas such as $\neg\neg\neg\neg\neg a$ or $a \rightarrow (b \rightarrow c)$ are ranked highly, despite their analytical straightforwardness. In general, as with other size based measures of complexity, size can be important if there are sharp limitations on storage (such as short term memory), but otherwise do not usually pose the most difficult problems.

In Table 3 on page 194, we see the ranking by the number of (distinct) variables. This will always rank formulas of the form $a \rightarrow (b \rightarrow (c \rightarrow \dots z))$ the furthest down the list. In particular here, $(\neg a \rightarrow a) \rightarrow a$ is lower than $a \rightarrow (b \rightarrow c)$. The number of distinct variables has been closely linked to measures of *simplicity.* Kemeny [254] and Goodman [186] elaborate this by also introducing extra criteria concerning symmetry. I argue that complexity is sensibly distinguished from such measures which are perhaps better though of as rough measures on information (see section 6.5 on page 129).

Table 4 on page 195 looks at the formulas in terms of their (maximum) depth. Clearly depth (as to a lesser extent size) can impose difficulties in terms of computation both in inference and induction. Nevertheless formulas like $\neg\neg\neg\neg\neg a$ appear well down the list. Such a ranking may be important if the operator (in this case negation) itself implies a more complex interaction (such as a modal operator might imply between possible worlds).

The breadth of formulas is shown in Table 5 on page 196. This is a weak measure of complexity as the formulas can be indefinitely large, deep and still be fairly interconnected. Primarily this could be seen as a limit on the input to the formulas, or alternatively representing a limit on a hypothetical top-down search on the structure of the formulas. A formula like $((a \rightarrow a) \rightarrow (a \rightarrow a)) \rightarrow ((a \rightarrow a) \rightarrow (a \rightarrow a))$, which intuitively represents merely a multiple use of substitution in identity will have maximal breadth, but it is analytically simple (this would not help you, of course, if you have to process it top-down in ignorance of the content of its leaves).

Finally Table 6 on page 197 shows them ranked by the cyclomatic number of the formula’s minimal directed acyclic graph (as explained in section 5.4 on page 106). Again formulas can be of indefinite size and depth, for a given ranking, but only if a suitable number of distinct variables are introduced (or if you are merely iterating negation). Formulas of the form $a \to (b \to (c \to \dots z))$ are always maximally simple in this way, representing the fact that the formula *in itself* does not encode any complexity of relationship. Non repetitive but involved formulas referring to variables repeatedly but in different ways come out as complex.

The different rankings of a selection of formulas are then directly compared in Table 7 on page 198. The cyclomatic number of the graph of the references in a formula gives a much better measure of an intuitive idea of their *analytic* complexity than the others.

Of course, as I have repeatedly stressed above, the complexity of an expression depends on its context (composed of language, goals and viewpoint). Thus it is important, for example, to keep in mind whether one is talking about the complexity of interaction encoded as facts about the natural world represented in formulas or talking about the properties of the connectives themselves by using axioms and the like. The difference comes out particularly in the appropriate relevance relation: if you are talking about the natural world then the relevance between identically named variables in separate formulas reflects the uniform reference intended by these formulas; if you are implicitly specifying the properties of the connectives (as in formal logic) then the appropriate reference is also intended between identical connectives between the axioms. Thus in these two different situations a different relevant relation needs to be used and this would affect the model of analytic complexity.

5.6.2 Proof complexity

Systems with minimal axioms are very difficult to use to prove anything. The few but flexible rules mean that the form of the desired theorem gives one little indication as to how to proceed. The proof, of identity, using (4) is shown in figure 25[53].

1	CCCpqrCCrpCsp	*Axiom*
2	CCCCpqrCCrpCspCCCCrpCspCpqCrCpq	*Subs. inst. of 1*
3	CCCCrpCspCpqCrCpq	*MP 1, 2*
4	CCCCCrpCspCpqCrCpqCCCrCpqCCrpCspCtCCrpCsp	*Subs. inst. of 1*
5	CCCrCpqCCrpCspCtCCrpCsp	*MP 3, 4*
6	CCCpqCpqCCCpqpCsp	*Subs. inst. of 1*
7	CCCCpqCpqCCCpqpCspCCCCpqrCCrpCspCCCpqpCsp	*Subs. inst. of 5*
8	CCCCpqrCCrpCspCCCpqpCsp	*MP 6, 7*
9	CCCpqpCsp	*MP 1, 8*
10	CCCCpqpCspCCCspCpqCrCpq	*Subs. inst. of 1*
11	CCCspCpqCrCpq	*MP 9, 10*
12	CCCpqpCpp	*Subs. inst. of 9*
13	CCCCpqpCppCCCpqpCspCpp	*Subs. inst. of 11*
14	CCCpqpCspCpp	*MP 12, 13*
15	Cpp	*MP 9, 14*

Figure 25. The proof of identity in Lukasiewicz’s single axiom system

In practice logicians tend to deal with the difficulty of producing proofs in these circumstances by developing an elaborate system of derived rules. Thus they implicitly change both their language of representation of the problem of producing such proofs in terms of these derived rules as well as the original axioms and rules. The greater simplicity of the problem is a result of an implicit change in the language of representation.

For any particular theorem, however complicated, one could trivially invent a proof system where its proof was simple by adding it as an axiom. Likewise for any theorem,

53.The proof is displayed using polish notation and using a substitution rule instead of axiom schemas in order to shorten it.

however simple, one could add intermediate steps necessary to make its proof as difficult as wanted. The complexity of proof production is thus a separate matter from the analytic complexity of individual formulas.

Thus, in this section I concentrate on characterisations of the complexity of the proof tree of a theorem. This is an abstraction of the uses of the axioms and inference rules, where the axioms are the leaves and the inference rules the nodes. For example the proof immediately above (figure 25 on page 117) could be represented as in figure 26 below.

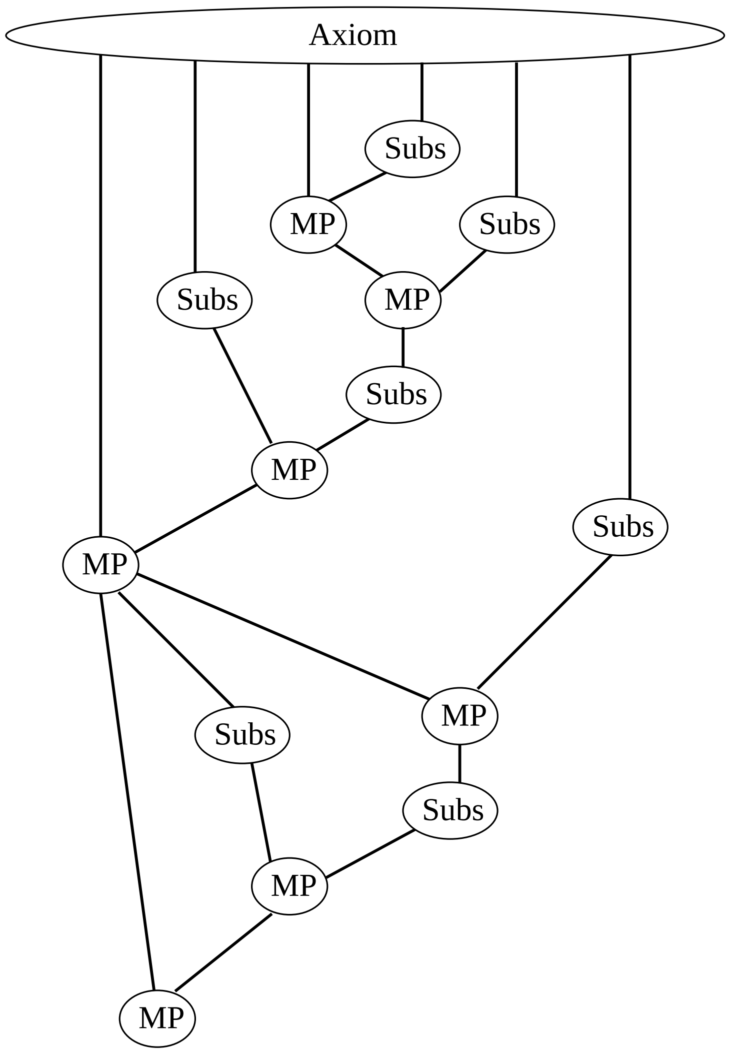


Figure 26. An example proof net

The complexity of the proof would alter dramatically if we included and related the formula that were introduced via the substitution rules. Here we have a case where the complexity of the derivational structures in the proof syntax would be more complex if we also included the derivations of the formulas used in the substitution steps. Of course, for some purposes this would not be relevant to the task at hand, but it would be for others where one was trying to trace the referents as they were manipulated by the proof rules.

In many proof systems attempts at proof construction are made incrementally, in either a top-down fashion, starting at the target theorem or bottom-up fashion, starting with the axioms (or, of course, a combination of these). In this case the search space of potential proof trees grows exponentially with the depth. Thus there is a strong connection between the depth of the proof tree and the computational complexity of a “brute force” automatic search for a such a proof.

An analytic approach to constructing such a tree may, of course, proceed in a more “intelligent” fashion, by analysing the possible proof choices. Thus the analytic complexity of such a proof will, as in the consideration of analytic complexity of axioms etc., depend upon the ability to decompose a proof into separate subtrees. The tractability of such a tree will depend upon the relevance relation between the nodes of such a tree. Different kinds of analysis will lead, as ever, to different measures of its analytic complexity. For example if the relevance relation was expressed just in terms of which axioms were used in the sub-proof tree then one would get a different measure than if one also included what formulas were used to be substituted into those axioms (which would be different again from the situation where every different instance of an axiom schema was considered as different).

Different kinds of proof procedure exhibit different degrees of decomposability in the proof production process. A Hilbert system with the minimal number of axioms will thus produce proofs that are more analytically complex than, for example, those produced using a Fitch based procedure, where the proof of intermediate steps can be recursively constructed to *some extent* independently to the rest of the proof. Of course, such a decomposition of the proof process may, in some circumstances, be illusory in that the sub-task may be as hard (or even harder) to prove than the original. This does not prevent Fitch-style procedures aiding considerably in the decomposition of proof tasks for a wide range of systems and for a wide range of target theorems. Thus the claim that Fitch proofs are simpler is borne out, even if in some cases (e.g. in R due to the undecidability of theoremhood there) it may not help with the most difficult cases.

Even if you restrict yourself to Hilbert style systems the set of axioms chosen can make a difference to the complexity of proof. For example, Robinson [380] exhibits a "nice" set of axioms composed of groups of axioms for each connective, such that you only have to use the axioms that correspond to the connectives in the target theorem. This is in contrast to many systems, for example in many classical axiomatic systems one has to use the properties of negation to prove the theorem $((a \rightarrow b) \rightarrow a) \rightarrow a$, even though it does not mention negation. In this way Robinson's axioms allow the partial partitioning of the proof space by the type of connective. This partitioning represent's an analytic simplification of the proof process.

5.7 Application to Simplification

In this section I briefly look at five possibilities for systematic simplification given the above analyses of complexity: those of searching for equivalent expressions; searching for equivalent derivations; by specialising the syntactic level in the underlying language; by more generally searching over different languages; and by sacrificing accuracy or specificity.

5.7.1 Searching over equivalent expressions within a language

The most obvious way to proceed in simplification is to attempt to search all acceptable expressions in some language for the most simple one. So one might look through all proofs starting with the simplest and working up (in terms of depth) until one arrived at one which proved the desired theorem, which would then be the simplest. Of course a slightly more intelligent procedure would involve a lot of intelligent pruning of the search space before hand, but unless this pruning can be arranged so that it drastically reduces the search space, such a procedure would have limited practical applicability.

In some cases this is not only impractical but impossible. The question is given some formal language, *L*, some equivalence relation, ~, defined on *L*, some measure $C: L \rightarrow \Re^{+}$, and a specific expression, $l \in L$, is there a program that can always find a expression $m \in L$, such that $m \sim l$ with $C(m)$ minimal? In general the answer is, unsurprisingly, "no". For example, it is well known that the problem of finding the minimum sized program for computing a pattern is uncomputable [286] (thus in this case *L* is the space of programs, *C* is their length and $m \sim l$ iff *m* and *l* compute the same pattern).

Even if such a question was computable for some *L*,*C* and ~ there will typically be no way of avoiding searching the space of available expressions L and checking them all (unless P=NP, which is considered unlikely [231]).

5.7.2 Searching over equivalent derivations within a language

Part of the problem with the above approach is the possible ineffectiveness of the equivalence relation, ~. If we have step-wise procedures for computing equivalent expressions the task becomes easier. For example, if one were seeking to simplify an algebraic expression like $2x^2(yz - y^3) + (xy - 3)(1 - xyz)$ one would have a range of possible transformations one could apply to the expression based on the properties of simple arithmetic: factorisation, expansion, association, permutation of commutative operators etc.

This would make the search procedure considerably more tractable and several effective techniques would become available. One such is genetic programming [270]. This is a version of the standard genetic algorithm [228], but where the genes that are evolved according to an evaluation of fitness are tree-structured expressions instead of merely strings of a fixed length. In the case of arithmetic simplification the trees that would be evolved would be possible applications of transformations to the original expression with the fitness of such a gene being decided by the simplicity of the result. The algorithm proceeds by starting with a random population of such compound transformations represented as tree-like structures, then evaluating these expression's fitness (some inverse of complexity) and creating the next population by a combination of propagation and a sexual operator called "crossover"[54] on parents chosen probabilistically in proportion to their fitness. Genetic programming techniques have be used in many domains and there are now many refinements and variations on the technique (see for example [15] and [261]).

5.7.3 Specialising the syntactic level

Consider the following 1-level syntactic structure for generating the propositional implication-negation wffs: $S=(\emptyset, \{R1, R2, R3, R4\})$ where rules $R1, \ldots, R4$ are defined as follows.

54.In this a random subtree is chosen inside each parent and then these are swapped around.

R1: $\emptyset \gg v$

R2: $x \gg x'$

R3: $x \gg \neg x$

R4: $x, y \gg x \to y$

The first two rules produce an indefinite supply of unique variable names $v, v', v'', \ldots$ and the second two are the normal syntactic production rules for negation and infix implication.

Now if one’s definition of relevance is that two expressions are relevant if they share a symbol in the syntax, then everything will be relevant to everything else because all expressions generated by *S* will share the symbol *v*. Clearly, although there is a sense in which all variables *are* related in this system, it might be *intended* that different variables (ones with a different number of primes) not be related. In this case a structure that is closer to what was intended would be $S2{=}(\mathrm{S1},\{\mathrm{V,R3,R4}\})$, where V was defined as follows

V: $\emptyset \gg x\ (x \in S1)$

and $S1 = (\emptyset, \{R1,\mathrm{R2}\})$, where $R1, \ldots, R4$ are defined as above. Now the generation of variable names is done by *S*1 and the building of formulas is done by *S*2. Now by the definition of relevance different variables will not be relevant to each other in the sense above. Thus the formula $v \to v'$ would have a cyclomatic complexity of 1 in *S* but of zero in *S*2, i.e. changing the language from *S* to *S*2 has resulted in a simplification. Of course, this is equivalent to suitably changing the relevance relation in *S*, but here the separation of the syntax allows for a more natural and clearer analysis of the intended situation.

The above is a very artificial example, but such shifts of syntactic level do occur. The sentence “Throughout June I sought her in vain.” will have a different complexity if you are considering the mapping between words and referents than if you look at the correspondence between spelling and phonemes (e.g. the intricacies of the “ou” spelling in English).

In logic, the formula $(a \to b) \to (a \to b)$ is often treated as a special substitution instance of identity, for the purposes of inference, rather than as signifying something in its own right in detail.

Another example comes from chaos theory. It is well known that the ‘tent map’[55] can generate sequences that are indistinguishable from purely random ones. If one models this process by considering all different numbers as irrelevant to each other (i.e just another number), then this seems very surprising - we have what seems to be a simple deterministic process producing random behaviour. Closer inspection reveals that the source of randomness comes from the initialisation of the system, as almost every[56] (abstract) real number has a ‘random’ decimal expansion, and the tent map merely “unpacks” this expansion and returns it digit by digit.

Two other examples from the literature[57] are: seeking to simplify the execution of programming languages by reduction over programming language hierarchies [179] and simplifying the visual presentation of graphs by reduction and abstraction [258].

One final example is arguably this thesis itself; the pushing of much of the detail into appendices is intended to simplify the presentation of the main argument.

5.7.4 Searching over equivalent languages

Perhaps the most radical but also the most natural mechanism for simplification is to change the language of representation. Changing the syntactic level (as described above in section 5.7.2 on page 121) is a particular example of this. Another example is when we enrich our proof language to simplify proofs (as described in section 5.6.2 on page 117).

Part of the problem with formalising complexity is that we humans seem to be very good at “flipping” between sub-languages in order to make things simpler. Since complexity, as I have argued, is relative to the language of representation, the complexity will change with each “flip”. Things can seem very different on the page, because we often fix the formal language (at least for the purposes of display of formal properties) and investigate the properties in that language.

In order to formalise such a procedure for simplification one needs some mechanism for fixing the reference and structure of a formula whilst changing the language it is embedded in. This can be done using the syntactic structures described above (section 5.5 on page 108). The idea is to fix the “lower” structures and vary the top one. For example if

55.The iterated map $x \to 2x$ if $0 \le x \le 0.5$ and $x \to 2(1-x)$ if $0.5 < x \le 1$ see [39] for details.

56.‘almost every’ in the sense that it is true for every number except the rationals

57.Although these are not discussed therein from the same perspective.

one were searching for a proof system that gave as simple derivations as possible (for a target set of theorems) one could define a syntactic structure to generate all the possible expressions and another syntactic structure to select the target set. Then consider a range of equivalent syntactic proof structures that will derive the same target set. The idea is represented below in figure 27 on page 124. The solid lines represent the sub-syntax relation and the dotted lines represent generation.

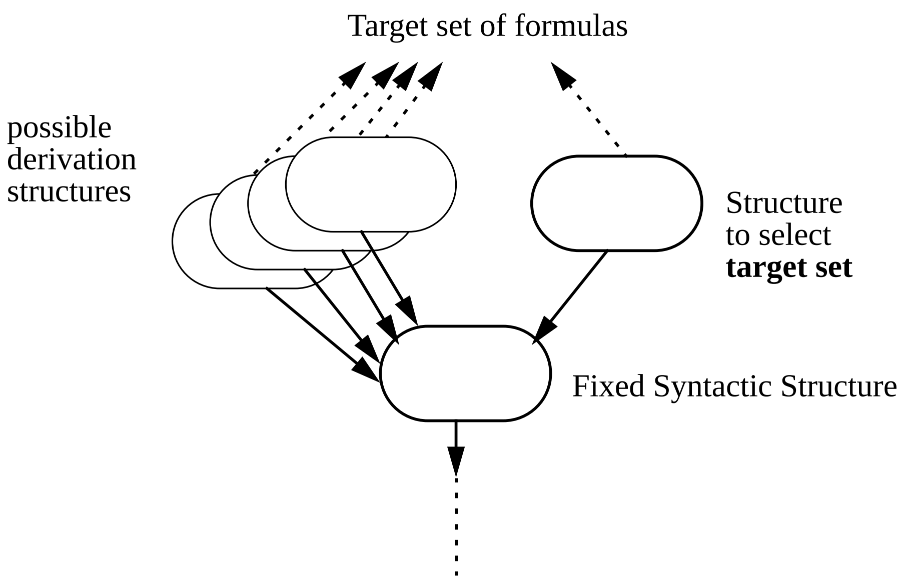

Target set of formulas
possible derivation structures
Structure to select **target set**
Fixed Syntactic Structure

Figure 27. A setup for varying languages

Of course, in general searching over equivalent languages will be at least as hard as searching over equivalent expressions. One could quite well represent the syntactic structures in a tree-like structure upon which a genetic programming algorithm would work, but it would frequently be computationally onerous to check that such a structure generated the same target set (and sometimes this would be totally uncomputable).

If, however, we knew of some acceptable transformations of the syntactic structure which preserved the target set we could search over the possible transformation sequences in a way similar to that outlined in section 5.7.2 on page 121. This, however is a matter well beyond the scope of this thesis.

5.7.5 Simplification via trade-offs with specificity and accuracy

All the above methods of simplification maintain the same model content, which is appropriate in formal systems. However perhaps the most common methods of simplification is where the content of the model is allowed to change so as to trade-off the simplicity of a model with the model’s accuracy or specificity.

Consider first complexity-specificity trade-offs. In some cases, especially when the data can be separated into what might be seen as random noise and an ‘underlying’ signal, a slightly less specific model of the data can be a lot simpler. So, for example the diagram on the right in figure 14 on page 62 representing complete randomness would take a model of high complexity if it were to be modelled in every detail. If on the other hand a model of it as just a random pattern retaining only the granularity, density, and the probabilities of occurrences of black and white dots were used then this might be a lot simpler. Associating the complexity of the pattern with the complexity of its most appropriate model restores our intuitions about them. It is the insistence of an ultimately specific model of these patterns (for example via its reproduction via a Turing Machine) that leads to the unintuitive result that a random pattern holds the most information.

A model can be made less specific in several ways, the simplest being the relaxation of its predictions from a precise value to an range (as is implicit in the notation 12.6cm±1mm); another way is to narrow the conditions of application of the model[58]. Where the distribution of the randomness can be more precisely specified this might be characterised in terms of a probabilities.

Another trade-off is the complexity-accuracy trade-off. This can have a similar effect to the complexity-specificity trade-off: here an increase in error may be acceptable if it results in a sharply simpler model, especially if it is thought that the more elaborate model might not be robust. After all, *some* level of error is deemed acceptable in almost all experimental science – we accept that it is usually not sensible to reject a theory due to the presence of a residual level of error. A famous case of this is the Michelson-Morley experiment, which despite the fact that it seemed to show a positive ether drift of about 5m/s (which was eventually explained in the 1960’s), did not justify the rejection of the special theory of relativity.

An example of an explicit trade-off of accuracy for simplicity in the induction of decision trees is investigated by Bohanec and Bratko in [69].

58.This makes the model less specific overall because outside these conditions of application the model does not restrict what might happen, i.e. it says anything can happen then.