Logic for Fun: an online tool for logical modelling

John Slaney
The Australian National University

Abstract

This report describes the development and use of an online teaching tool giving students exercises in logical modelling, or *formalisation* as it is called in the older literature. The original version of the site, 'Logic for Fun', dates from 2001, though it was little used except by small groups of students at the Australian National University. It is currently in the process of being replaced by a new version, free to all Internet users, intended to be promoted widely as a useful addition to both online and traditional logic courses.

Keywords: Logic Teaching, Online Learning, Logical Modelling, First Order Logic.

1 Background: Formalisation

In introducing undergraduates to formal logic, we attempt to impart a range of skills. In a typical "Logic 101" course, the most prominent of these involve manipulation of calculi: devising proofs, usually using some form of natural deduction, constructing semantic tableaux or the like. We also ask students to formalise natural language sentences—often specially constructed to involve awkward nesting of connectives or strings of quantifiers—and may hope that they acquire some facility in critical reasoning and perhaps an appreciation of some wider issues connected to logic, be they mathematical, computational, philosophical or historical. Some of these things we teach better than others. Although devising proofs is traditionally a stumbling block, most students do in fact become tolerably adept at handling the technical details of natural deduction. Where we fail is rather in teaching them to say what they mean in the abstract notation of logic: many students who can mechanically

A short version of this paper was presented in the conference *Tools for Teaching Logic* in Rennes in June 2015. Thanks are due to the participants in that conference for illuminating conversations and ideas.

construct a proof remain depressingly unable to write a well-formed formula to express even a simple claim about a domain of discourse.[1] Barwise and Etchemendy, for instance, comment:

> The real problem, as we see it, is a failure on the part of logicians to find a simple way to explain the relationship between meaning and the laws of logic. In particular, we do not succeed in conveying to students what sentences in FOL mean, or in conveying how the meanings of sentences govern which methods of inference are valid and which are not. [3], p.13

We should find this situation alarming. Mechanical symbol-pushing for the purposes of simple proofs is easy to teach, tolerably easy to learn—at least, if students can be brought to work at it—and almost useless once the course is finished. On the other hand, the ability to read and write in the notation of formal logic, to use this as a medium for knowledge representation, to analyse and to disambiguate, is the most important skill students can take away from an introductory logic course, and it is a skill most of us can claim less success in teaching.

The reasons why students find formalisation so hard are not difficult to discern. The problem of rendering a description into formal notation has no unique solution and there is no easy way to know whether a putative solution is right or wrong. There is also no simple algorithm for treating such problems, so "surface" learning is ineffective. The ony successful method, in fact, is to understand both the natural and formal languages and to match the two. Faced with this challenge, students not infrequently give up.

It was against this background that the tool *Logic for Fun* [9, 10] was devised around 15 years ago. *Logic for Fun* is a website on which users are invited to express a range of logical problems and puzzles in such a way that a black-box solver on the site can produce solutions. It was never tied to a prescribed course, but was intended to be used as an adjunct to undergraduate courses, whether those be in logic, critical reasoning, artificial intelligence or other fields, or by interested individuals outside the context of formal instruction. The language in which problems are to be expressed is that of a many-sorted first order logic, extended slightly with a few built-in expressions and modest support for integers. The solver takes as input a set of formulae in this language and searches for finite models of this set. If it finds a unique model, this is almost certainly a solution to the problem. More usually it

[1] Evidence for this claim is anecdotal, but strong. My own appreciation of it was sharpened in 2011, when analysis of results from a class of 61 students showed grades on proof construction that were on average above their grades for other courses, but after 13 weeks of study more than a third of them were unable to express 'The bigger the burger the better the burger' adequately in the notation of first order logic.

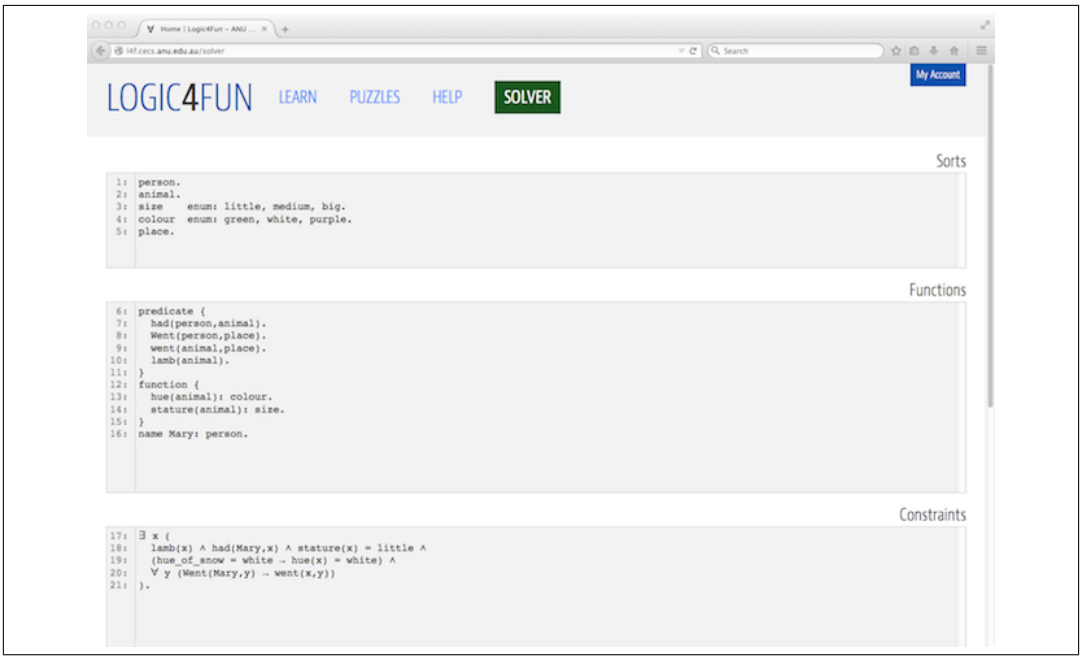

Figure 1: Screenshot of the window for problem input

either reports syntax errors in the encoding or else says that there is no solution. The user (i.e. the student) then debugs the encoding until it is correct. This has several advantages over traditional formalisation exercises:

1. The sentences to be formalised constitute a meaningful problem, rather than looking like isolated examples of things it may be tricky to express formally;
2. There is an easily graspable concept of a *correct* solution, so students know whether they are right or wrong;
3. Feedback is always accurate and *immediate*, rather than coming a week after the homework was handed in;
4. Because there is no feedback latency, because there is a goal (solving the problem) and because the machine is infinitely patient, students will put time and effort into their work, to a degree never seen in the traditional setting;
5. For the problem encoding to count as correct, it must be completely accurate, as the computer will not accept hand-waving, so the value of rigorous attention to detail is constantly emphasised.

An example (not a puzzle in this case, but a first order theory) will help to illustrate the process required of the student. Figure 1 shows the form used for text input. It consists of three boxes: in the first are listed the "sorts" or domains over which variables are to range; in the second is the vocabulary of non-logical symbols (predicates, names, etc) with their types; in the third are the constraints, written as first order formulae.

The logic taught in introductory courses is usually single-sorted rather than many-sorted, but the experience over 15 years has been that students find it easy to adjust to having many sorts available.[2] In the example, there are six sorts (persons, animals, sizes, colours, times and places) of which two (sizes and colours) are given by explicit enumeration while the other four are left for the solver to decide.

```
Sorts:
  person.
  animal.
  size enum:  microscopic, little, medium, big.
  colour enum:  green, white, purple.
  time.
  place.
```

The domains of these sorts are disjoint. We wish to say that Mary had a little lamb, so as vocabulary we declare 'Mary' as a name, 'had' as a relation between persons and animals, 'stature' as a function of animals and 'isLamb' as a predicate:

```
Vocabulary:
  predicate {
    had(person,animal).
    isLamb(animal).
  }
  function {
    hue(animal):  colour.
    stature(animal):  size.
    location_p(person,time):  place.
    location_a(animal,time):  place.
  }
  name Mary:  person.
```

[2]Single-sorted encoding of problems is perfectly possible in *Logic for Fun*, in case anyone really wants or prefers it, but for most problems it is not recommended.

The convenience of using a many-sorted logic for knowledge representation purposes is immediately apparent. It enables us to specify the types of predicates and function symbols separately from the formulae in which they occur. The hue and stature functions, for example, map animals to their colours and sizes; to model the constraints will be to determine which functions to assign to them. This not only removes the need for sort specifications in the antecedents of the constraints, but it allows the type checker to detect many errors which might otherwise lead to nonsense.[3]

Since every user-defined predicate and function symbol is strictly typed, and sorts are disjoint, it is not possible to specify a `location` function which can apply to both people and animals. Hence there are two, one for each sort. We have considered relaxing the syntax rules to allow unions of sorts in argument places, but in fact the ramifications of this would cause more complexity for the user than would be justified by the reduction in artificiality of the syntax.

To finish the example, here is the constraint:

```
∃ x (
    had(Mary,x) ∧
    isLamb(x) ∧
    stature(x) = little ∧
    (hue_of_snow = white → hue(x) = white) ∧
    ∀ t (location_a(x,t) = location_p(Mary,t))
).
```

Below the constraints box are buttons, not visible in Figure 1, for running the solver. There is one for generating solutions and another for running in a lightweight mode to check syntax. Having written the description, the student clicks "Solve" and the back-end reasoner—essentially a SAT solver adapted to finite domain first order problems—starts searching for solutions. Naturally, it finds models of this little theory with *very* small domains, including the expected interpretation in which there is one person (Mary), one animal (the little white lamb) and one place which neither of them visits. Amusingly, the solver also finds unexpected solutions, for instance in which the lamb is green—but it is still just as white as snow because snow is purple!

[3] Other problem representation languages also benefit from this feature. The constraint modelling language Zinc [4, 6] for example has an elaborate type system, whereas its subset Minizinc [7] treats almost everything as a number. In Minizinc, you can add an employee to a day of the week and divide by a truck, and no type error is detected. Many-sorted first order logic is a valuable step towards fully typed languages, with all the advantages they confer.

In the pedagogic context, this provides a good opportunity for the teacher to make some points about interpretations and truth conditions and the semantics of the material conditional. When this kind of situation arises in modelling a puzzle which should admit only one solution, the student must invent more constraints to supply the missing information (e.g. that snow is white and that Mary went to school). In order to do this, they have to think about the semantics of the problem, render it into first order logic and understand the relationship between the formulae they have written and the satisfying formal interpretations.

Of course, some consistent first order formulae have no finite models, and even where finite models do exist, there may be none sufficiently small to be presented. The existence of models and of finite models is undecidable in general. However, this is not a matter of great concern, firstly because the problems on the *Logic for Fun* site are deliberately chosen to have easily discovered small models, and because the solver is in any case set to time out after three seconds (or another short time if the user chooses) so completeness of the search is neither expected nor really desired.

The most significant precursor of *Logic for Fun* was Tarski's World [2, 1] which provides exercises in reading and writing formulae expressing facts about situations. The use of problems rather than simply states of affairs as the basis, and of the user's "freestyle" choice of vocabulary, mark significant differences between *Logic for Fun* and earlier software. More recently, websites have appeared using functionality similar to that of *Logic for Fun* for other purposes. A good example is MiniZinc which has been used to teach aspects of constraint programming—there is even a MOOC based on it [12]—though the emphasis there is partly on efficient encoding and search methods rather than purely on logical modelling.

2 Structure of the exercises

The problems given as exercises on the site are divided into five levels: Beginner, Intermediate, Advanced, Expert and Logician. The boundaries between levels are not really definite, but students like the idea of progressing through levels in the style of a game. "Beginner" problems are fairly trivial to represent and solve, and are designed to get students through the phase of learning to use the site, teaching them how to declare vocabulary, write constraints and read the solver output. Figure 2 shows one of the "Beginner" problems with a suggested encoding. Note that students actually need to learn a good deal about function declarations and other syntax details in order to master this level, but that the logic is not deep. "Intermediate" problems are mainly logic puzzles of the kind found in popular magazines, often calling for bijections between sets of five or six things satisfying a list of clues. There is a long

There are four children, Alice, Boris, Claire and David. Their ages (not in order) are 6, 7, 8 and 9. Each child has either 1, 2, 3 or 4 jellybeans (a different number each).

1. Alice's age plus her beans is the same as Boris's age plus his beans.
2. Claire's age is two more than the number of Alice's beans.
3. Alice's age is equal to the number of beans she and Claire have altogether.

How old is David, and how many beans has he?

Sorts

```
1:  child enum: A, B, C, D.
```

Functions

```
2:  function age(child): int   {all_different hidden}
3:  function beans(child): int {all_different hidden}
4:  name David's_Age: int.
5:  name David's_Beans: int.
```

Constraints

```
6:  ALL x (
7:    age(x) > 5 AND age(x) < 10 AND
8:    beans(x) > 0 AND beans(x) < 5
9:  ).
10:
11: age(A) + beans(A) = age(B) + beans(B).
12: age(C) = beans(A) + 2.
13: age(A) = beans(A) + beans(C).
14:
15: David's_Age = age(D).
16: David's_Beans = beans(D).
```

Solver Output

```
Model 1

David's_Age = 8

David's_Beans = 1
```

Figure 2: A "Beginner" problem, its encoding and its solution

tradition of making these problems gently humorous, and students are usually familiar with the style of problem, so most of them find it fairly easy to get this far with the site. "Advanced" puzzles are not necessarily harder, but have features requiring more sophisticated logical treatment—nested quantifiers and the like. This can cause difficulties for users without a background in logic, so where *Logic for Fun* is used in a logic course, it is worth spending time on several of the "Advanced" problems rather than rushing past them to get to more interesting ones. The "Expert" puzzles are more challenging, and include several state-transition problems from AI planning, for instance. They require students to supply less obvious vocabulary and axiomatisation to represent preconditions, postconditions and frame conditions of actions and so forth. Most of them can be represented in several different ways, using different ontologies and different styles of constraints. This offers opportunities for the teacher to discuss (and for the student to experiment with) non-trivial aspects of formalisation. Finally, the "Logician" section contains problems which hint at applications of logic, for instance to finite combinatorics and to model-based diagnosis.

It is important that all of the suggested problems can be solved quickly by the software behind the site, without requiring coding tricks. This is because the aim is to teach correct logical expression, not constraint programming. Especially for some of the "logician" puzzles, efficiency does matter, as the underlying problems (e.g. minimal hitting set, classical planning, quasigroup completion) are NP-hard, and solver behaviour can be affected by non-obvious things like the order in which functions are declared, but as far as possible the site de-emphasises efficiency and instead lays stress on correctness.

An interesting feature of teaching logical modelling in this way is that concepts are introduced in approximately the opposite order from that in the parallel lectures. The standard structure of a typical logic course is to work from the more abstract levels down to the more detailed ones. We start with the generic idea of inference, then proceed to examine propositional connectives, then move to first order logic with names, predicates and quantifier-variable notation. Then we introduce identity as a special relation symbol and go on finally to deal with function symbols and general terms. We do not usually get as far as many-sorted logic. There are good pedagogic reasons for this order of exposition, as the more intricate parts of logic presuppose the simpler ones, and the details make more sense within a clear framework than they do in isolation. It is undeniably easier to learn to manage boolean operations on sentences than quantification over arbitrary domains, so we aim to give our students facility in manipulating the former before expecting them to tackle the latter.

Logical modelling, by contrast, *starts* with sorts, equations, names and functions.

The very first example in the user guide to *Logic for Fun* is

> Find a number x such that $2 + x = 4$

which of course students without a logical background find entirely trivial. Considered as a first order formula, however, it involves both interpreted and uninterpreted individual constants, a binary function symbol with a built-in interpretation, equality as a special binary relation, and integers as a sort—quite advanced material for Logic 101. The key concept right at the start is that of interpreting a formula by assigning values to its uninterpreted symbols. Generality needs to be introduced next. The universal quantifier is used much more than the existential one: at this early stage, existentials can mostly be replaced by Skolem functions (mostly constants) without making students explicitly aware of this substitution. Only after that do we meet the basic connectives. Introductory logic textbooks, though they vary greatly in emphasis and style, tend to follow the same overall direction as logic courses.[4] My experience of using *Logic for Fun* as part of a standard introductory logic course is that the reversed order of topics does create a certain degree of difficulty. It requires the lecturer to spend time explicitly pointing out the relationship between the logical modelling exercises and the rest of the course, as the two strands do not converge until the end and may seem disjoint to many of the students. I have not yet experimented with the possible strategy of inverting the entire course, making logical modelling the centrepiece and working from the detailed and specific towards the more abstract.

3 The logic of *Logic for Fun*

The fragment of logic underlying *Logic for Fun* is chosen to be useful for expressing simple theories over finite domains without departing too far from standard first order logic. Thus the language includes the usual connectives, $\wedge$, $\vee$, $\neg$, $\rightarrow$, and quantifiers $\forall$ and $\exists$. It also makes heavy use of the identity symbol, $=$, not only to express equations but also to express uniqueness and the like. As noted above, the logic is many-sorted, so that variables are able to range over things of a kind without repetitive antecedents to restrict the constraints, and so that functions can be typed to remove the need for many cumbersome axioms. On interpretation, the sorts correspond to disjoint finite domains. Since the relation of identity makes sense for every domain, the equality symbol is typically ambiguous, though it is a type error to assert an identity between two things of different sorts. The objects of

[4] There are exceptions. One of the most notable is the little introduction by Wilfred Hodges [5] used over 30 years ago as a textbook by the Open University and still in print.

a sort may be enumerated explicitly, giving each a canonical name and defining a total order on them, or they may be left unspecified. Upper and lower bounds may optionally be placed on the cardinality of a non-enumerated sort.

Any non-reserved string of characters not containing punctuation or space may be either a name or a variable. It is a variable if bound by a quantifier; otherwise it is a name. It may not be used both as a name and as a variable in the same problem encoding. Names may be declared along with the rest of the vocabulary, or may be used without declaration.

For the purposes of writing formulae, the logical symbols may be written as the English words `AND`, `OR`, `NOT`, `IMP`, `ALL` and `SOME` (all in upper case). This is helpful for students in the early stages of learning to use the site, who may be unfamiliar with standard notation. There is an option to use pure clause form for writing constraints, avoiding explicit connectives and quantifiers altogether in favour of simple punctuation. This was designed to make *Logic for Fun* independent of notation, so that it could be seen as compatible with absolutely any introduction to elementary formal logic. However, pure clause form proved unpopular with students, so this option is disabled in the current version of the site and "normal" first order notation used instead.

There are two built in sorts, `int` and `bool`. The latter consists of the two truth values with canonical names `FALSE` and `TRUE` (in that order). The former does *not* consist of the integers: since all sorts are finite, and in fact quite small, it consists of the first few natural numbers $0, \ldots,$ `MAX_int`, where `MAX_int` is set to a fairly low value such as 100. For encoding logical puzzles, it is by default set even lower, at 20 or 30, as the exercise only involves logic, not arithmetic, for which purpose very small numbers are sufficient. Note that identity on the boolean sort is material equivalence, so no special symbol besides '=' is needed for the material biconditional.

More built-in operations are provided, as they have been found useful for expressing finite domain problems. Since each sort is totally ordered in a canonical way—either by explicit enumeration or implicitly—it makes sense to refer to the smallest (first) and largest (last) elements of a sort as `MIN` and `MAX` respectively. These symbols may be subscripted with the name of a sort if this is not deducible from the context. The ordering relation on any sort is represented by '<' and '>' as one would expect. Any element may be incremented or decremented by a positive integer, so for instance `foo+2` is the item (if any) which comes two after `foo` in the canonical ordering of its sort. In practice, this is almost always used just to add or subtract one to refer to the successor or predecessor of an object.

Predicate and function symbols, including individual constants (names), may be declared as in the example in Section 1 or in Figure 2. Each has not only a specified type but optionally a list of features. These include the useful feature "hidden",

which suppresses printing of the symbol and its value. Binary operators, whether predicates or function symbols, may optionally be written in infix position between their arguments: there is no need to specify this, as the parser accepts both prefix and infix notation, even for the same symbol in the same formula.

Functions of any type may be partial, lacking values for some arguments. By default, functions are total, but may be declared partial when they are specified in the "Vocabulary" box. Certain other properties, such as being injective or "all_different" may be enforced in the same way. Each sort has an existence predicate `EST` (for "there Exists Such a Thing as...") which returns `TRUE` if the expression to which it applies has a value in the domain of the interpretation, and `FALSE` otherwise. This is extremely useful where partial functions are used. In a domain of persons, for instance, we might have a function `spouse` returning an individual's husband or wife. Then (for the artificial purposes of some problem) we may want to say that there are no same-sex marriages within the domain:

```
ALL x (NOT(female(spouse(x)) = female(x))).
```

However, there may be unmarried people, so we want `spouse` to be a partial function, in which case we can write:

```
ALL x (EST(spouse(x)) IMP NOT(female(spouse(x)) = female(x))).
```

to say that if there exists such a thing as x's spouse, then that individual has opposite gender from that of x.

The list of extensions and restrictions of standard first order vocabulary may may seem complicated, but in fact they serve to adapt "pure" first order logic rather modestly for the purpose of easy applicability to finite domain problems.

One question which arose early in the development of *Logic for Fun* was whether to restrict it to first order logic or to allow higher order constructions. Over finite domains, of course, the distinction between first order and higher order expressions is a little artificial, as sets and functions are just more finite objects which could be referenced in a first order way, but there are clear reasons for avoiding them in general for the purposes of this site. Notably, they cause exponential or hyper-exponential increases in the sizes of domains, thus conflicting with the manner in which everything is internally represented to the solver by explicit enumeration, and with the need to solve problems in seconds at worst (more usually in milliseconds). Some second order features, notably allowing reference to the transitive closure of a binary relation, would be useful for knowledge representation in certain cases and have polynomial time propagators, so they could be added without causing an explosion. It is possible that such features may be included at some stage, but as things currently stand they remain unavailable.

By an *interpretation* of the first order language specified for a given problem we mean an assignment of a domain $\mathcal{D}(s)$ to each sort s and an assignment of a

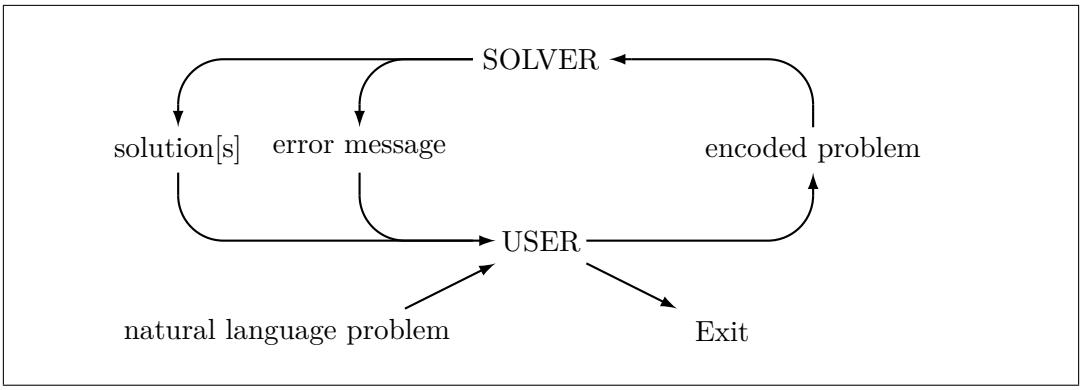

Figure 3: *Logic for Fun* workflow. The main loop is a dialogue between a user and the solver.

function $\mathcal{I}(f)$ of appropriate type to each function symbol f. Each domain is a finite set, and each function symbol of type $s_1, \ldots, s_n \longrightarrow s$ is assigned a function from $\mathcal{D}(s_1) \times \ldots \times \mathcal{D}(s_n)$ to $\mathcal{D}(s)$. A predicate symbol from this perspective is just a function symbol of value type `bool`, and a name is a function symbol as above with $n = 0$. Built in function symbols are given the obvious readings.

As expected, an interpretation is a *model* of the set of constraints specified by the user iff each of them evaluates to `TRUE` in accordance with the standard story about truth for an interpretation. Note that since the language has no free variables, there is no need to treat valuations of variables separately from complete interpretations differing on the values of names. If no model is found by the solver within the time limit, the message "No solution found" is returned. If several models are found, by default up to three are returned. The limit on the number of models may be changed via the "settings" button available to the user. Each model returned is reported by explicitly printing the tables of values of each function for its possible arguments. Students have usually found this style of writing solutions easy to understand.

4 Errors and feedback

The workflow of the site (Figure 3) is one of dialogue between student and machine, whereby the problem in natural language is initially proposed as a challenge to which the student responds by writing formal encodings of all or part of the problem which the solver evaluates. Feedback in the form of error messages or solutions (or lack of solutions) informs the student's next attempt. The cycle is broken when the student decides to terminate or suspend it. Work may be saved at any point for

future reference.

Clearly, the educationally effective part of this process is the correction of errors. To put it simply: the tool is only doing useful work when its users are making mistakes. Feedback is therefore the essence of the process. Errors (apart from accidental slips) are fundamentally of two kinds, syntactic or semantic, evoking very different responses from the system. Errors of syntax are caught by the parser or the type checker and reported with explicit messages. For example, if the user writes

```
had(Mary, ∃x(lamb(x))).
```

(presumably trying to say "Mary had some x such that x is a lamb") the solver replies:

```
Input error on line 32:  had(Mary, SOME x lamb(x)).
Type mismatch with argument of had

Detailed diagnostics:  in the formula
    had(Mary,SOME x lamb(x))
the main operator "had" expects argument 2 to be of type animal
but argument 2 is
    SOME x lamb(x)
which is of type bool.

Hints for this kind of error:  check
  (1) parentheses;
  (2) possible typos (e.g.  misspelling);
  (3) how variables and names get their sorts assigned.
```

The suggested possible causes of this kind of error—misplaced parentheses and wrong names—are not guilty in this case, but the key information that there is a boolean formula where a reference to an animal was expected is clearly present. The solver also lists the vocabulary used in the offending formula and writes out the parse tree as far as the parser was able to get before raising an exception. This kind of detail in error messages is an important feature, but such verbosity can become irritating so a possible enhancement for a future version of the site might be to place detailed drill-down under user control.

Users have frequently reported obscure error messages as a major source of frustration in using *Logic for Fun*. Efforts to make error messages more friendly and informative are continuing, as this is an aspect of the site which is perpetually capable of improvement.

Semantic errors are harder to classify and harder to deal with. There are no "canned" solutions written in, so if the encoded problem gets past the parser and type checker, all the solver can do is search for solutions and report what it finds.

Hence the only symptom of misunderstanding on the semantic level is an unexpected solution or (more often) no solution. This is the case whether the error is due to basic misunderstanding of semantics, such as confusion between the conditional and the biconditional, or whether it is a matter of problem representation—using logic correctly to say the wrong thing.

5 Diagnosis tool

The case in which there is no solution is common, and of course the solver's response "No solution found" provides the user with a minimum of information. Techniques for making the feedback more informative include commenting out lines of the encoding and re-running to see whether solutions exist. This can sometimes be effective in pinpointing incorrectly expressed constraints, though it is laborious and the results are not always helpful.

A diagnosis tool designed to help automate the process of isolating incorrect constraints in cases where the encoded problem globally has no solution has been developed: a prototype exists and has been tested by a range of volunteer users, ranging from beginners to experts, but has not yet been incorporated into the live version of the website. As noted, it cannot tell the user what is wrong with their problem encoding, since there is no way of knowing what solution (if any) is desired, but it makes available two types of information:

Approximate models If there is no model of the set of constraints within the parameters set by the sort and vocabulary specifications, the constraints can be marked as "soft" and the solver asked for an assignment of values satisfying as many of them as possible—that is, to solve the problem as a MAX-CSP. The constraints violated by the approximate model are listed. The user may mark some of them as "hard" and re-solve, finding a new model (if there is one) satisfying the hard constraints and approximately modelling the soft ones. Iterating this procedure partially automates the "commenting-out" dialogue, with additional functionality in that optimal approximations rather than arbitrary models are returned.

The back-end solver can run in two modes to search for approximate models. On the site at present, it always searches by depth-first branch and bound, which has the advantges of a complete search method: it stops when the search space is exhausted—often in a fraction of a second—and when it does it returns either a provably optimal approximation or else the information that the hard constraints are unsatisfiable. There is also an option (not currently

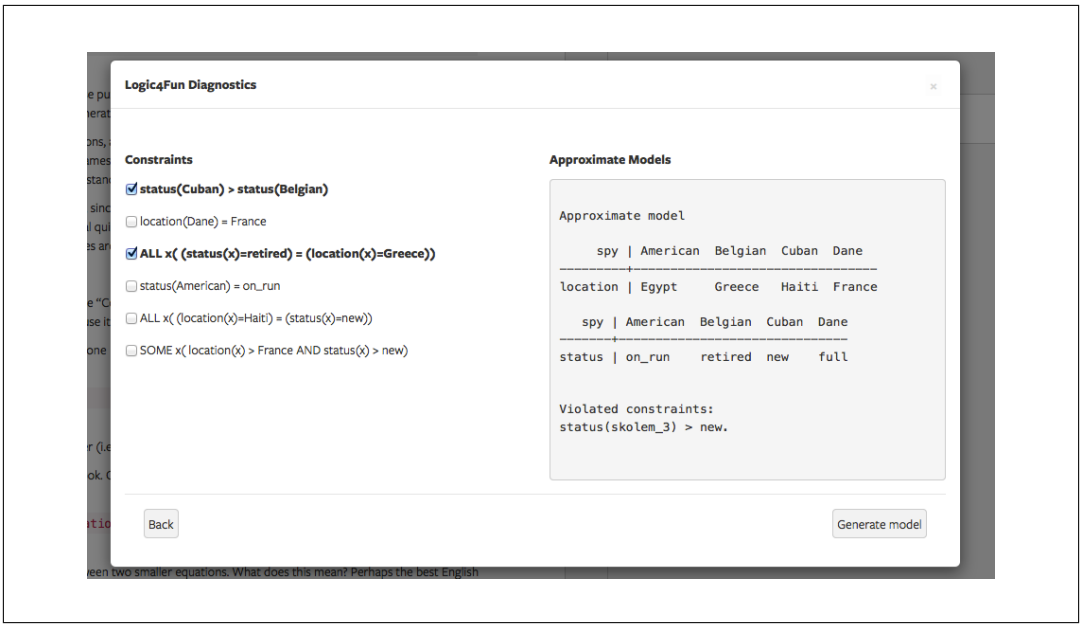

Figure 4: Screenshot of diagosis tool: approximate model

used in *Logic for Fun*) to perform a local search somewhat in the manner of WalkSAT [8]. Since this is an incomplete method, the results it returns come with no guarantee of optimality, and it cannot show unsatisfiability. It does, however, return reasonable models in a reasonable time, even for large or complex problems, so it may have its uses as a fall-back option in cases where the complete search fails. We shall experiment with including it in *Logic for Fun* in future.

Unsatisfiable cores The solver may be asked to identify a minimal subset of the constraints with no solution. Any such subset must contain a contradiction, and so needs correction as at least one of its members is false in the intended model. There may be many unsatisfiable cores in an inconsistent CSP; at present, the diagnosis tool returns an arbitrary one. Finding an optimal (minimal cardinality) unsatisfiable core is computationally difficult: even with an oracle saying whether a subset of the given first order clauses is satisfiable, the optimisation problem would still be NP-hard. However, *every* unsatisfiable core needs to be repaired in order to make the encoding consistent, so *any* core gives potentially useful information. For that reason, it is not obvious that investing time in minimising the size of the core returned is justified. The

Figure 5: Screenshot of diagosis tool: unsatisfiable core

"Find optimal" button visible in Figure 5 provides an optionto seek an optimal core, but in view of the complexity issues, and because with small problem encodings the arbitrary core is quite often optimal anyway, this feature may be omitted from the tool when it is eventually deployed.

At present, the diagnoser works with the problem *after* it has been put into clause form. Consequently, the constraints violated by approximate models and those featuring in unsatisfiable cores are reported as clauses (with a little syntactic sugar such as universal quantifiers binding the variables) rather than as the input first order formulae. This is sometimes a little confusing for logically inexpert users, but is does have its advantages as it results in more precise diagnoses.

The two services provided by the diagnosis tool are in a good sense dual to each other [11]. More strictly, the set of unsatisfied clauses in an approximate model is a *diagnosis* in that it forms a hitting set for the set of all unsatisfiable cores, and dually each unsatisfiable core is *conflict*, which is a hitting set for the set of all such diagnoses. There is no general answer to the question of which is more useful, as it depends on the problem—and to some extent on the user. It may happen that there is no unsatisfiable core much smaller than the entire clause set, in which case the best strategy for the user is to ask repeatedly for approximate models, making constraints hard if they are obviously true. In other cases, the best approximate model may look nothing like the intended solution, and may violate many of the

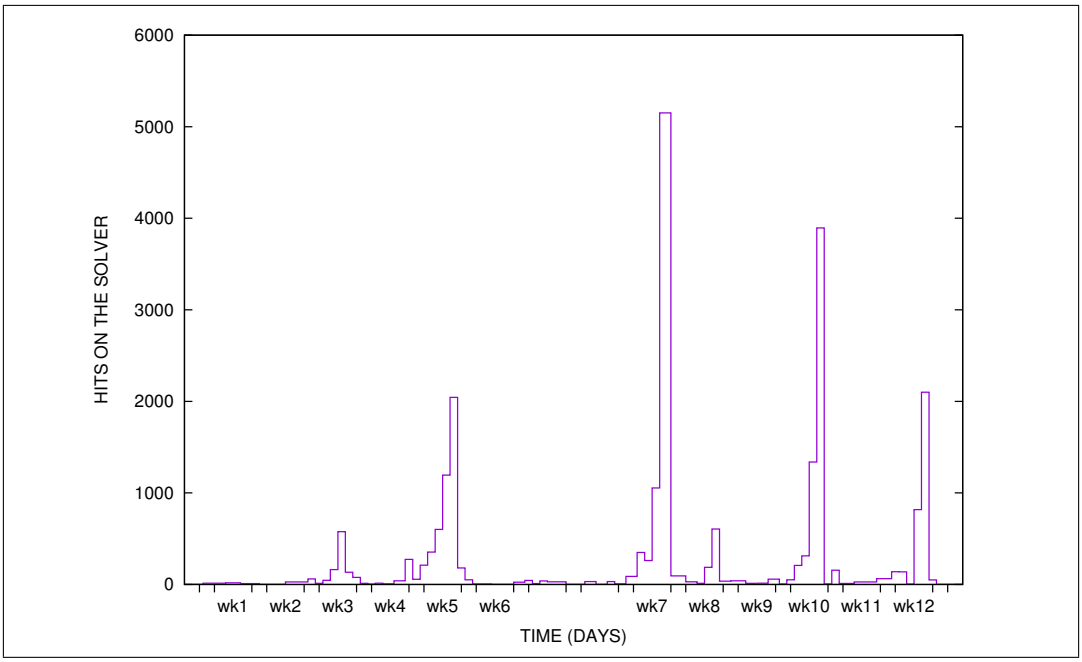

Figure 6: Number of times the solver was run on each day over 12 weeks of a logic course

constraints. In such cases it is likely that there are several bugs in the encoding, so a good technique is to isolate each one, in a small unsatisfiable core if possible, and re-examine the problem after each repair.

The diagnosis tool is not a magic bullet. Sometimes it helps; sometimes it does not. In any case, it provides only information about models or the lack of them: repairing the problem encoding is still a task for the user and still rests on understanding first order logical notation. It does, however, reduce frustration by assisting with the reasoning process rather than letting "No solution found" be a backtracking point in the dialogue between user and machine.

6 Site usage

Logfiles produced by the scripts on the site can be mined for data on usage patterns, and provide some insight into how students set about mastering the web-based tool and using it to solve problems of logic. At the simplest, aggregated statistics for the number of hits on the site allow us to observe class behaviour. Figure 6 shows the number of times the solver was run each day by a class of around 50 undergraduates

Logic Games

Our annual logic competition came to a final showdown between five teams: the Aces, Buccaneers, Cougars, Demons and Eagles. The contest was a round robin, each team meeting each other team once. At the end, the judges announced:

- Every team won at least once, and some team won all its games.
- The Buccaneers beat only the Cougars.
- Exactly one match was drawn, and it didn't involve the Cougars.
- The Aces defeated every team that the Eagles defeated, but they didn't defeat the Demons.
- Not every team that defeated the Aces defeated every team that the Aces defeated.

With that, they left us to work out the full set of results. Well?

Figure 7: The homework problem for week 7

during a 12-week semester in 2013. Note that there was a 2-week break between weeks 6 and 7.

Students had a piece of homework to do each week, and had to hand it in for assessment before midnight each Friday evening. These assignments in weeks 5, 7, 10 and 12 consisted of problems to be solved using *Logic for Fun*, as can clearly be seen from the bursts of activity in those weeks. The assignment in week 8 was a problem concerning the semantics of some first order formulae, which the students were asked to compute on paper, using semantic tableaux, before comparing their answers with models of the same formulae produced by *Logic for Fun*. The small peak in week 3 is associated with the point at which they were introduced to the site and asked to complete some easy exercises to familiarise themselves with it. The homework in weeks 3, 4, 6, 9 and 11 did not call for students to use the site.

The heaviest usage of the semester occurred on the Friday of week 7, when the solver was run on average almost 100 times per student. This represents an extraordinary amount of work by the class on what was a fairly modest piece of logic homework. The problem in question (see Figure 7) was designed to turn on the correct handling of quantifiers, though the most awkward part of the problem is to encode the notion of a drawn match. Since the usage log from that period only records who ran the solver at what time, not the full text of what they sent to the solver, we have no way now of knowing what particular difficulties caused the class

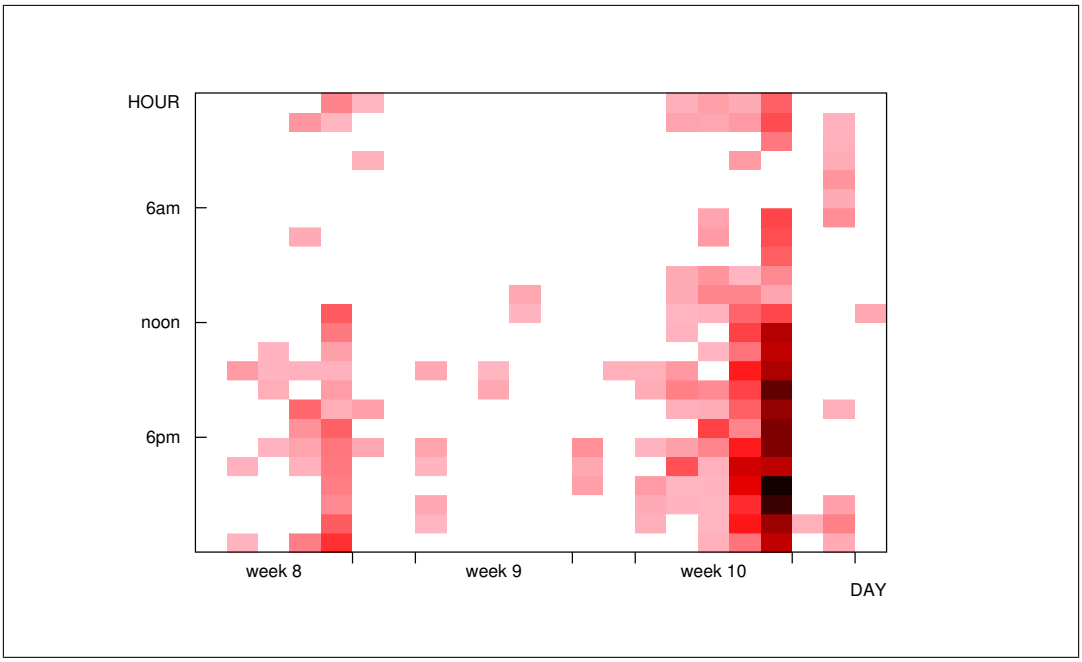

Figure 8: More detailed plot of site usage hour by hour over 3 weeks.

to spend so much more effort on this problem than on the others.

Closer analysis of the aggregate site usage reveals further patterns in students' work. Figure 8 shows the number of runs in each hour over a period of three weeks culminating in an assignment which required them to encode the problem 'Logic Games' (Figure 7). Darker colours indicate heavier usage. Although the amount of work peaks as expected on the day of the assignment deadline (not at the eleventh hour, but about three hours prior to that) there is clearly a substantial number of students whose habit is to work on problems such as these some days in advance. Recall that the "assignment" was only a piece of homework requiring a few hours of work at most, so for most students it was feasible to leave the job until the last day. The activity two days after the deadline was that of students completing the assignment late: they were allowed to submit work up to 48 hours after the deadline, for a marks penalty, and for unknown reasons some of them chose to exercise this option.

Yet more detail is revealed by studying the work patterns of individual students. Some behaviours are quite striking: there are, for example, students who will run the solver 200 times or more on one problem, many of the runs being only seconds apart. Figure 9 shows an example of the activity of one such student over the five

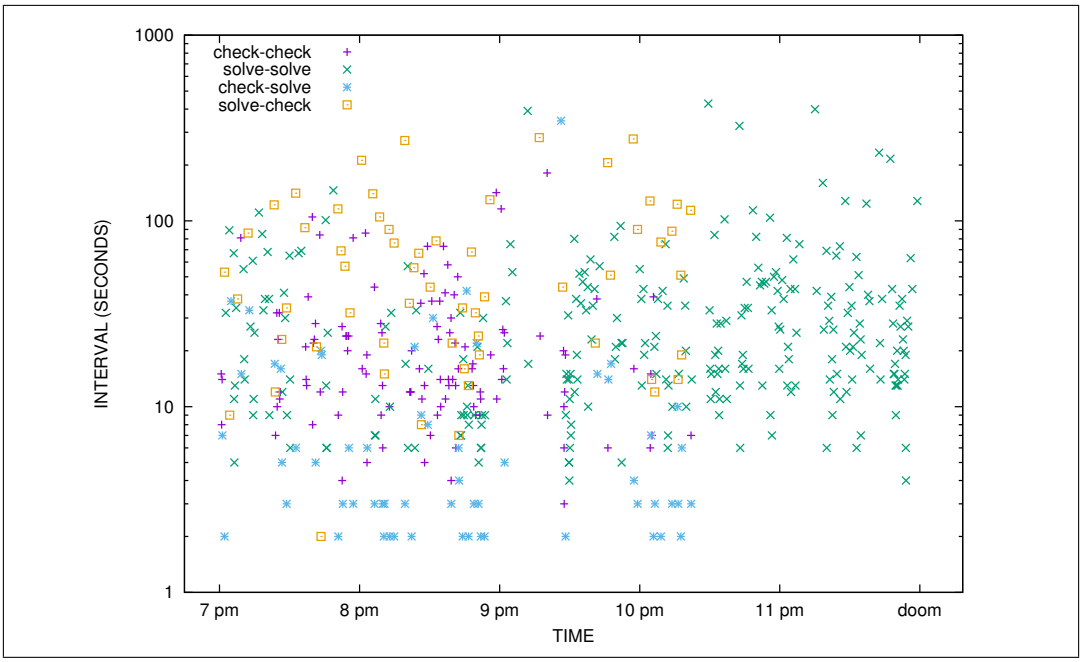

Figure 9: Five hours of work by one student. Each point is a run of the solver, showing the interval since the previous run plotted against the time of day. Different shapes show whether the run was a syntax check or a "solve", and whether the previous run was a check or a solve.

hours before the submission deadline. Note that this is a huge amount of work compared with the few minutes which students normally spend on a hand-written formalisation exercise. At no point during the five hours does this student pause for much more than 5 minutes. Most of the runs which occur within 10 seconds of the previous run are cases in which a syntax check is immediately followed by a "solve", presumably because there was no syntax error. We see some patterns in the record: for instance, at some point (a little before 10:30 pm) this student abandons explicit syntax checks and simply uses the "Solve" button. It is unclear why.

7 Current and future work

Logic for Fun was completely re-scripted in 2013–14, partly because the look and feel of the old site dated from another millennium, partly to extend its functionality in significant ways, and partly to have a version built with modern tools which would be easy to maintain. The new site is scripted entirely in Python, with a little

Javascript for client-side functions such as insertion of logical symbols into the text, though the solver behind it is still the original, written in C some 20 years ago. The beta version of the new site [10] is now freely available to all web users, in contrast to the old site which required them to have accounts and to pay fees. This is in line with the contemporary expansion of free access to educational tools.

The biggest enhancement from the user's viewpoint is a facility to save and reload work, allowing it to be carried over easily from one session to another. Students who join a class (called a "group" on the site) can also submit their work to the group manager for feedback or assessment. Low-level improvements, such as organising the display so that the natural language version of a puzzle can be in the same browser window as the student's logical encoding of it, also do much to enhance the user experience. The syntax for declaring vocabulary has been simplified and made more similar to that of analogous declarations in computational settings. Thus whereas the classic site used mathematical syntax for declarations like

```
function f: s –> t.
function P: s –> bool.
```

the new site uses "computational" syntax for the same thing:

```
function f(s): t.
predicate P(s).
```

There is an ambitious plan for continued upgrading of the site. Tools currently under development include the diagnosis assistant already noted in Section 5 above, for use when a problem as written by the user has no solution. A prototype of this tool looks promising, and has been positively received by users who have tested it, but it will not be incorporated into the live site until 2016. It is also planned that the system will maintain a detailed database of user activity, recording every character of text sent to the solver. This information will be used in a project aimed at deeper understanding of the logic learning process.

An important piece of future work, to be conducted when the new site is stable, is an effectiveness study. It is obvious that doing formalisation exercises online rather than on paper causes students to put much more effort into getting their answers right, but measuring the extent to which they learn more as a result is much harder. There is no naturally occurring control group for an experiment, so it is important that institutions other than the Australian National University begin using the site and generate comparative results between the years when it is used and years when it is not. Such longitudinal data will take time to accumulate, and no such study is available yet.

Acknowledgements The author wishes to thank Matt Gray, Kahlil Hodgson, Daniel Pekevski, Nathan Ryan and Wayne Tsai for help in scripting *Logic for Fun*, Nursulu Kuspanova for her invaluable work on the diagnosis tool, and the many logic students over the last 15 years who have helped by testing the site to destruction.

References

[1] Dave Barker-Plummer, Jon Barwise, and John Etchemendy. Tarski's world, 2008.

[2] Jon Barwise and John Etchemendy. Tarski's world, 1993.

[3] Jon Barwise and John Etchemendy. *Language, Proof and Logic.* CSLI Publications, Stanford, CA, USA, 1999.

[4] Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Principles and Practice of Constraint Programming (CP)*, pages 700–705, 2006.

[5] Wilfred Hodges. *Logic: An introduction to elementary logic, 2nd edn.* Penguin, London, 2005.

[6] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13:229–267, 2008.

[7] Nicholas Nethercote, Peter Stuckey, Ralph Becket, Sebastian Brand, Gregory Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming (CP)*, pages 529–543, 2007.

[8] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *Theoretical Computer Science*, DIMACS Series Volume: Clique, Graph coloring and Satisfiability—Second DIMACS implementation challenge. American Math Soc:290–295, 1996.

[9] John Slaney. Logic for Fun (classic version). `http://logic4fun.rsise.anu.edu.au/`.

[10] John Slaney. Logic for Fun, Version 2 Beta. `http://{L4F}.cecs.anu.edu.au/`.

[11] John Slaney. Set-theoretic duality: A fundamental feature of combinatorial optimisation. In *European Conference on Artificial Intelligence (ECAI)*, pages 843–848, 2014.

[12] Peter Stuckey and Carleton Coffrin. Modelling discrete optimization. `https://www.class-central.com/mooc/3692/coursera-modeling-discrete-optimization`.

Received \jreceived