Why there is no solution: A diagnosis tool for teaching logic

John Slaney
john.slaney@anu.edu.au

Nursulu Kuspanova
Australian National University

This paper reports our experience in developing an online tool which uses automated reasoning to support the teaching of elementary logic at the undergraduate level. For this particular application, the back-end reasoner searches for satisfying interpretations rather than for proofs, but the key issue of helping users to make sense of automatic reasoning is expected to be common to many inference systems. We have enhanced the user interface by adding functionality to help diagnose errors in the case that the user's expression of a logical problem is syntactically well-formed but inconsistent. In this paper, we describe the teaching tool and the diagnoser, and report preliminary user feedback.

1 The setting

Elementary logic is notoriously harder for many students to learn than to all appearances it ought to be. As teachers, we commonly find ourselves labouring the perfectly obvious; yet every student who infers Fa from $\exists xFx$ in a natural deduction calculus, or who moves from the truth of $p \rightarrow q$ to the truth of q when constructing an analytic tableau, reminds us that the perfectly obvious easily becomes lost in the details of its explicit expression. It is frustrating, and at times even puzzling, that such gross errors persist however often and however carefully we explain the operation of a logical calculus. At their root is the tendency of many students to see superficial learning of the calculus "rules" as an easy option in comparison with any deeper study; as a result, they reject attempts to engage them in depth learning, and their understanding remains limited.

1.1 Logic for use

This tendency to reduce logic to symbol-pushing has an unfortunate side effect. While students' manipulation of the calculi may be error-prone but improves with practice, their use of logical notation to specify situations and problems in many cases remains poor. The reason is not hard to see: faced with a problem and the task of expressing it formally in first order logic, there is no alternative to understanding the semantics of the problem description, understanding the semantics of logic, and matching the two. There is no symbol-pushing algorithm for that. Given that "freestyle" formalisation is too hard for many, we sometimes fall back on setting exercises in which more or less convoluted sentences in stilted English are to be rendered formally by abbreviating words to single letters and writing the well-marked connectives and quantifiers as squiggles according to a look-up table. This makes formalisation easier for the students, and undeniably easier for us when we have to assess their work, but it allows students to emerge from "Logic 101" without the most important skill we want them to acquire.

Logic is indeed a medium for formal proof, and understanding the concepts of validity, derivation and (dually) satisfaction is important, but in practice the *notation* of logic is used far more often for knowledge representation—as a concise and well-defined language framework for describing situations, processes and problems. Those of us who work on theorem-proving software may tend to think of logic primarily in terms of deduction, but the harsh truth is that once Logic 101 is finished, the ability to construct natural deduction proofs is almost useless, whereas the ability to express things logically, to

Submitted to:
UITP 2016

disambiguate, to clarify and to render problems amenable to formal reasoning is of value across many areas of endeavour. If students leave us without having acquired fluency in reading and writing logic to some purpose, we have failed to teach what they really need to know.

1.2 Logic for Fun

The web-based tool *Logic for Fun* [4, 6] is an attempt to address the above issue directly. It asks users to encode a range of finite-domain problems in such a way that a black-box solver on the site can produce solutions. The solver is essentially a SAT solver with a few extra features to make problem encoding easier, and what it gives as a "solution" is actually an interpretation, in the ordinary sense appropriate to first order logic, satisfying the formulae given by the user. The formal language used as input to the solver is a many-sorted first order logic with the usual connectives and quantifiers and with a few built-in predicate and function symbols and (very) limited support for integers. The problems range from the trivial (e.g. "Find a number, x, which when added to 2 gives 4", encoded as "`2 + x = 4.`") through a variety of logic puzzles, to some challenging state-transition problems from AI planning, for instance, and to some problems in combinatorics, diagnosis, etc. which hint at the real uses of logic.

The apparatus of logic is not presented at the start as something to be learned and then applied, but rather introduced when needed, as a collection of techniques to help users express what they want to express. Along the way, there are certainly instances of traditional formalisation exercises such as expressing the constraint "Not every team that defeated the Aces defeated every team the Aces defeated", but by the end the challenges are more open-ended and higher-level: "Find a way to get all the cannibals and missionaries across the river without any unfortunate setback for cultural imperialism." To tackle the advanced problems, users must invent suitable vocabulary (with type specifications) and use it in logical formulae with the right semantics to force its models to be valid plans or whatever; the problem statement in English gives little direct help with this task, and the solver only looks at the logic, returning models of the input or the information that no model exists. Users must then debug their work until the solutions are as desired.

An example will help illustrate the tool. We wish to express not a problem but a first order theory about life in the country and the ways of ruminants:

> *Mary had a little lamb,*
> *Its fleece was white as snow,*
> *And everywhere that Mary went*
> *The lamb was sure to go.*

This is hardly deep as logic goes: as the name of the tool suggests, it's just for fun. There is no unique correct encoding, of course, but a reasonable starting point is to divide the domain of discourse into several disjoint subdomains or sorts. We need to quantify over people, animals, times and places, and it will also be useful to specify some lists of colours and sizes. We therefore have:

```
Sorts:
  person.
  animal.
  time.
  place enum:  home, school, mountain, farm.
  size enum:  microscopic, little, medium, big, huge.
  colour enum:  green, black, white, purple.
```

The vocabulary we need will be declared with (first order) types in terms of these sorts:

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

Since we wish to keep persons and animals as distinct sorts, the `location` function cannot apply to both persons and animals. Hence there are two, one for each sort. The location of persons is made surjective (by annotating its declaration) just to ensure that every place gets visited (by someone) at least once; this is not required by the theory, but it is harmless and makes the story more interesting.

Finally, 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))
).
```

The solver generates some unexpected models along with the expected ones: for instance, there are solutions in which the lamb is green—but it is still just as white as snow because snow is purple!

Logic for Fun is not the only tool of its kind. Its best-known precursor is *Tarski's World* [2, 1] which was originally written (for PC rather than for the Web) in 1991 and which provides a range of machine-checked formalisation exercises. More recently, a web-based tool [7] very much in the style of *Logic for Fun* has been deployed to help in teaching MiniZinc [3], a subset of the Zinc modelling language associated with the constraint programming platform G12 [8]. Like *Logic for Fun*, it asks users to encode problems rather than single statements, though its focus is more on efficient encoding for constraint programming whereas *Logic for Fun* aims purely to teach logic.

2 The problem

The workflow of the dialogue between the user and the site is essentially a loop, in which the user submits a candidate encoding, and the solver replies with solutions or an error message, giving the user information with which to debug the encoding for the next iteration. The loop is broken when the user is satisfied with the solution(s) or gives up. Work can be saved at any stage for later recall. Students

will characteristically put at least an order of magnitude more work into encoding problems this way than they would into doing the exercise on paper, but frustration with the system is a critical problem, experienced by almost all users at some time.

2.1 The customer is always wrong

In a good sense, the system is only doing useful work when the user is making mistakes. The quality of relevant feedback is therefore a key issue. Errors are of two kinds: syntactic and semantic. Syntax errors are relatively straightforward: they are trapped by the parser or the type checker and reported with explanatory messages. For example, if the user types

```
had(Mary, ∃ x isLamb(x))
```

presumnably trying to say Mary had some x such that x is a lamb, then the solver replies:

```
Input error on line 18: had(Mary, SOME x isLamb(x)).
Type mismatch with argument of had

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

Vocabulary used in the formula is:
  predicate had(person,animal).
  name Mary: person.
  quantifier SOME.
  variable x: animal.
  predicate isLamb(animal).

Parsed [sub]formula structure is:

  Mary
  had
  ( SOME x (
      isLamb (
        x
      )
    )
  )
```

While such messages can always be improved and made more user-friendly, they are not the focus of the present report. The more challenging question is how to deal with semantic errors.

2.2 Why is there no solution?

There are no canned answers to the problems in *Logic for Fun*. Even the vocabulary used to express a given problem is provided by the user, not by the system. Clearly, it is not possible in general for the

solver to identify semantic errors with much precision: where logic has been used correctly to say the wrong thing, there is no knowing just what the mistakes are without a description of the right thing. All the solver can do, therefore, if the input is syntactically well-formed, is to evaluate it and report whatever models it finds. Where there are unintended models, as in the case of the little green lamb, the user can inspect them, compare them with the English text to which they should correspond, and generally detect the missing or incorrectly expressed constraints. The case in which there are no solutions at all, however, is more common in practice, and the three-word feedback

```
No solution found
```

is less than ideally informative. Since the user's code is not executed, but only evaluated, there is no possibility of debugging in familiar ways such as stepping through the execution or inserting "print" statements. One is quickly driven to the option of commenting out lines and re-solving, or solving the problem by hand and adding redundant constraints expressing parts of the true solution to detect which formulae are inconsistent with them. These techniques are cumbersome and not always very effective. It would clearly be useful if the dialogue between user and solver could be helped along further in the case where the constraints are inconsistent.

3 The diagnosis tool

A range of possible tools might help. One can imagine a useful role for visualisation, starting with the constraint graph of the input, for instance. Another option would be to discover *proofs* of inconsistency, either by analysing the failed search for models or by applying independent automated reasoning techniques, and to present them in a human-readable form, though preliminary experiments in that direction are not encouraging. The difficulty with proofs is that the problems are essentially based on finite domains, so inconsistency often involves the fact that decision variables must take one of the n values, expressed logically as many ground clauses of length n; such things tend to give rise to ugly proofs, especially for n greater than about 3 or 4. We have also considered reversing the formalisation process by rendering the first order formulae in stilted but maybe comprehensible English, in the hope that this may help explain to users what they have written, but have not experimented with this yet.

What we have implemented, however, is a diagnoser which essentially automates the process of commenting out constraints and observing the results. This requires no judgments about the correctness or incorrectness of any particular formulae, but directs the user's attention to the minimal edits required to restore consistency. It does this in two complementary ways: by generating and presenting unsatisfiable cores and approximate models.

There are three levels of representation of the constraints in a problem encoding. On the highest level are the first order formulae supplied by the user. These contain quantifiers and connectives freely nested as one would expect. After parsing, these are put into clause form, resulting in the second level of problem representation. In general, one first order formula may result in several first order clauses, in which there is no nontrivial nesting of connectives, and from which quantifiers have been removed in the standard way. On the lowest level are the (many) instances of these clauses flattened and grounded on the chosen finite domain and represented to the solver as a single data structure indexed in quite complex ways and used to manipulate the permitted and forbidden tuples of values for the decision variables of a CSP. The diagnoser currently operates on the clause level only when generating unsatisfiable cores, but on the low level when generating approximate models.

An unsatisfiable core, as is familiar, is a minimal subset of the clauses which has no model within the framework provided by the sort and vocabulary declarations. Clearly, *every* unsatisfiable core must

be repaired by removing or editing at least one of its component clauses, so the identification of such a core can furnish the user with valuable information by restricting attention to a part of the encoding in which there must be an error. Our diagnoser gives the user the option of seeing an arbitrary unsatisfiable core, usually generated quite quickly, or one which is guaranteed to be optimal in the sense of being of minimal cardinality. Finding a core requires solving a sequence of NP-hard and co-NP-hard problems, so it is computationally hard in the worst case. Proving optimality makes the problem considerably harder in practice, even if it does not change the complexity class, so the diagnoser sometimes times out before returning an optimal core.

An approximate model is an interpretation of the language satisfying as many as possible of the ground-level constrtaints. That is, since there is no interpretation satisfying all of the constraints, we treat the problem as a MAX-CSP with a fixed penalty of 1 for each constraint violation and solve that by DFBB. In the case of logic puzzles, the problems are usually fairly small, so it is feasible to return provably optimal solutions in most cases, but DFBB can function as an anytime algorithm, so if the solver times out it usually returns a reasonably good solution—often an optimal one without finishing the proof of optimality.

When an approximate model is presented to the user, the clauses which fail in it are listed. The user may decide that some or all of them are correct, in which case they can be marked as hard constraints, meaning that they are no longer allowed to be violated, and another approximate model generated. After a few iterations of this process, it is quite likely that attention is focussed on the real culprit. At least the process allows the user to explore the space of interpretations roughly satisfying their theory, and to gain some feel for which clauses hang together: if a given clause fails, we see what else fails with it.

The set of clauses which are unsatisfied in an approximate model is obviously a hitting set for the set of *all* unsatisfiable cores. We call such a hitting set a *diagnosis*: it could be presented to the user as a suggestion as to which clauses need to be repaired. The set of unsatisfiable subsets and the set of (possibly sub-optimal) diagnoses form a dual pair of monotone problems [5]. One outcome of this is that if there are many diagnoses then there are few unsatisfiable cores, and *vice versa*. Another is that there is a simple method for turning decision procedures into optimisers: in particular, since the existing solver of *Logic for Fun* can decide satisfiability (and hence unsatisfiability) it can be used without modification to generate *optimal* unsatisfiable cores. See the cited paper [5] for details of the dual hitting set minimisation technique, which we use for finding optimal cores.

If there is a small unsatisfiable core, that is usually the best hint as to where to find the error in the encoding. Sometimes, however, there is no small core: the contradiction implicit in the theory may require almost all of the input formulae for its derivation. In that case, approximate models are likely to give more useful information. In the worst case, neither cores nor diagnoses are small, or the problem is so clumsily encoded that the solver times out without producing much information, or perhaps there just is no graspable explanation of the lack of solutions.[1] Hence there will always be cases in which the diagnosis tool gives no real help. However, more usually, it does provide some information which students may use for purposes of self-correction; the educational value of this process is anyway greater than that of copying an answer produced by a more elaborate tool.

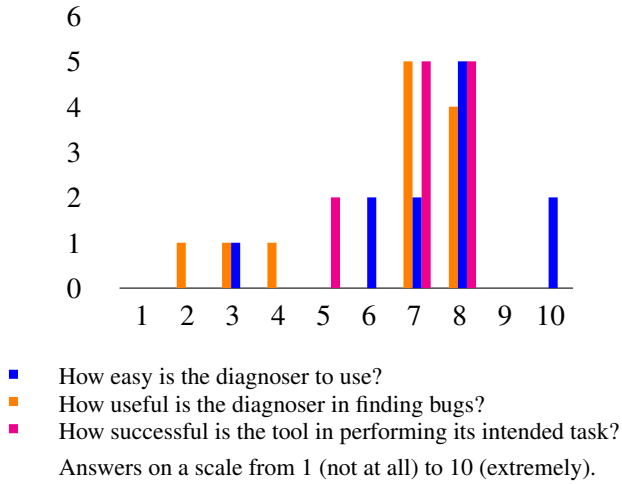

Figure 1: Results of a survey of 12 volunteer users of the prototype

4 User response

The diagnosis tool was incorporated into the live *Logic for Fun* site in February 2016, and is currently (April 2016) undergoing a field trial with a class of 80 students in an introductory logic course at the Australian National University. At the time of writing, the results are not yet available, though the trial will end in early June, so some information concerning usage and user experience will be available by July.

A small experiment was conducted in September 2015, in which 12 volunteers who were all experienced users of the site were shown a prototype of the new tool and asked to use it, without supervision, to help in encoding problems of their own choice. Their sessions with the enhanced site were scheduled to last roughly an hour, and in practice ranged from about 45 minutes to 2 hours. Obviously, given such a small number of users, the experiment did not produce very meaningful statistical results—and nor was it intended to. The users completed a short questionnaire at the end of their session, but their verbal responses were more helpful in informing the further development of the software.

The point of testing the prototype with human subjects was not to evaluate its effectiveness in terms of learning outcomes, but rather to assess its acceptability and to gather reports on whether users *felt* it to be useful. The initial results were reassuringly positive (see Figure 1) though the small sample size and lack of a comparison class mean that quantitative results should be taken as indicative only. All users reported that being able to continue the dialogue with the solver in this way is greatly preferable to being left only with the message "No solution found". Half of them found the unsatisfiable cores more valuable than the approximate models for debugging purposes, though a quarter of them reported the opposite experience and the remaining quarter rated the two types of feedback as equally valuable.

The usage pattern among the current class of students (the first to use the diagnosis tool in actual practice) shows around 50% of students making regular use of it. They typically run it between 5 and 20 times in the course of arriving at an encoding of a puzzle; at this point it is not clear whether they tend to ask mainly for unsatisfiable cores or for approximate models. By comparison, they will run the solver between 100 and 150 times on average in working on one problem, though around half of these runs are just syntax checks.

[1]For example, why is $\forall x \forall y(f(f(f(y,x),y),y) = x)$ unsatisfiable in a domain of size 10? There appears to be no explanation: it is just a brute fact of combinatorics.

5 Conclusion and future work

THe diagnosis tool has now been fully integrated into *Logic for Fun*, where it is providing generally useful feedback and enhancing the user experience of the site. From the technical viewpoint, the principal direction for future work is to incorporate a contemporary high-performance solver in place of the rather limited reasoner currently used. In view of the fact that basic numerical reasoning is needed for many of the more interesting problems, an SMT solver may be more appropriate than a purely logic-based one, though for the purposes of logic teaching it is also important not to stray too far into arithmetic. The back-end solver stands alone, independently of the scripts running the website, and is opaque to the user in any case, so its exact nature is unimportant as long as it deals quickly with small finite-domain problems and does not intrude into the high-level logical syntax.

Meanwhile, another project is currently in progress, mining the usage log of the current site (without the diagnoser, at this point) to identify and analyse strategies used by students when working on encoding logical problems. The record of their activity is a rich source of information, as it provides comprehensive, accurate and highly detailed observations of the learning process obtained non-intrusively with minimal perturbation of the observed behaviour. The first results from this project are expected later in 2016, though it will continue into the future. An obvious extension for the next iteration of data collection will be to examine records from the diagnoser as well as the solver: this is not technically difficult, and will enable us to know in detail in what circumstances students use the diagnoser, how they interact with it and what use they make of the information it provides.

What is already clear is that non-expert users of reasoning software, such as the logic students learning formalisation skills via *Logic for Fun*, need an environment which supports them with relevant information, especially when they make errors. It is useful to think of the human-computer interaction as a dialogue in which the user is guided by the program's responses. "No solution found" is too brusque as a response: if the dialogue cannot be continued past that point, the user easily becomes frustrated or disillusioned and may abandon the process. Our diagnoser does continue the dialogue, helping the user to understand *why* there is no solution by displaying near-solutions and by pinpointing the axioms involved in contradictions. One benefit is that the information is presented on the human-readable level—i.e. in terms of the formulae written by the user and approximate solutions in exactly the style of the full solutions the user wished to see. The response so far has been positive. While the differences between satisfiability solvers and theorem provers make it impossible for automated deduction systems to adopt our diagnosis tool directly, the motivation and the idea of presenting global information (e.g. in the form of models of clause sets) rather than the low-level results of drilling down do carry over to cognate reasoning software including theorem provers.

References

[1] Dave Barker-Plummer, Jon Barwise & John Etchemendy (2008): *Tarski's World*.

[2] Jon Barwise & John Etchemendy (1993): *Tarski's World*.

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

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

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

[6] John Slaney (2015): *Logic considered fun*. In: *Proceedings of the 4th International Conference on Tools for Teaching Logic*, pp. 215–222.

[7] Peter Stuckey & Carleton Coffrin: *Modelling Discrete Optimization*. `https://www.class-central.com/mooc/3692/coursera-modelingdiscrete-optimization`.

[8] Peter J. Stuckey, Maria J. García de la Banda, Michael J. Maher, Kim Marriott, John K. Slaney, Zoltan Somogyi, Mark Wallace & Toby Walsh (2005): *The G12 Project: Mapping Solver Independent Models to Efficient Solutions*. In: *Principles and Practice of Constraint Programming (CP)*, pp. 13–16.