Extracting Test Problems from Real Applications

John Harrison

Intel Corporation

- Applications of theorem proving
- Traditional 'type 1' problems
- Problems from formalization and verification
- Raw logging
- Higher-order to first-order reduction
- Some initial results
- A more sophisticated approach
- Summary

Applications of theorem proving

Automated theorem proving is sometimes pursued just for intellectual enjoyment.

But there are several significant applications, including:

1. Solution of individual mathematical problems (e.g. the Argonne group)
2. Formalization of mathematics (e.g. the Mizar project)
3. Formal verification (e.g. Intel's floating-point work)

Most of the best-known test problems, e.g. the TPTP suite, are heavily biased towards 'type 1 problems'.

In this talk, we consider how to produce more 'type 2 and 3' problems.

Type 1 characteristics

The 'type 1 problems' are often designed specifically as theorem prover test cases.

- Usually pure first order logic or equational logic and often in clause form.
- Often stretch the abilities of older, and sometimes present-day, systems, and may need hours or days to solve.
- Typically slightly 'artificial' (cute algebraic facts, combinatorial curiosities) and generally small.
- Often axiomatized in special ways for feasibility.
- Usually carefully formulated without irrelevant material (`NUM` is one exception).

They quite accurately reflect activity of type 1, but not so much types 2 and 3.

Type 1 examples

Examples include the 'nonobvious' Łoś problem:

$$(\forall x\ y\ z.\ P(x,y) \wedge P(y,z) \Rightarrow P(x,z)) \wedge$$
$$(\forall x\ y\ z.\ Q(x,y) \wedge Q(y,z) \Rightarrow Q(x,z)) \wedge$$
$$(\forall x\ y.\ P(x,y) \Rightarrow P(y,x)) \wedge$$
$$(\forall x\ y.\ P(x,y) \vee Q(x,y))$$
$$\Rightarrow (\forall x\ y.\ P(x,y)) \vee (\forall x\ y.\ Q(x,y))$$

traditional group theory exercises:

$$(\forall x\ y\ z.\ x \cdot (y \cdot z) = (x \cdot y) \cdot z) \wedge$$
$$(\forall x.\ 1 \cdot x = x) \wedge$$
$$(\forall x.\ i(x) \cdot x = 1)$$
$$\Rightarrow \forall x.\ x \cdot i(x) = 1$$

and truly difficult equational problems such as the Robbins problem, $x^n = x$ in a ring implies commutativity, etc.

Type 1: a guarded appreciation

Type 1 problems are representative of some of the most impressive applications of automated theorem proving.

They play a vital experimental role in pushing automated theorem provers to the limit.

Moreover, many of them are particularly striking or memorable.

However, we shouldn't let this blind us that they are not at all representative of more "workaday" type 1 or 2 applications.

It's definitely worth considering type 2 and 3 problems as well.

In fact, from a crudely pragmatic view, they may be *more* important.

A critique of formulation

Sometimes problems are formulated in artificial ways to make them easier. An extreme example is the use of $P(x, y, z)$ instead of $x \cdot y = z$, e.g.

$$
\begin{array}{l}
(\forall x.\ P(1, x, x)) \wedge \\
(\forall x.\ P(x, x, 1)) \wedge \\
(\forall u\ v\ w\ x\ y\ z.\ \ P(x, y, u) \wedge P(y, z, w) \\
\qquad\qquad\qquad\qquad \Rightarrow (P(x, w, v) \Leftrightarrow P(u, z, v))) \\
\Rightarrow \forall a\ b\ c.\ P(a, b, c) \Rightarrow P(b, a, c)
\end{array}
$$

This is not so common nowadays.

However, one unfortunate historical relic survives in the TPTP problem set: the curse of clausal form.

Problems with clausal form

Many traditional theorem proving methods use clausal form internally.

So formulating problems in clausal form allows one to compare underlying algorithms more precisely.

However, not all problems are naturally formulated in clause form. The translation can build in choices that can be difficulty for the underlying prover to reverse or change, e.g.

$$(\exists!x.\ f(g(x)) = x) \Leftrightarrow (\exists!y.\ g(f(y)) = y)$$

Clausifying this directly leads to a problem substantially harder than the two problems obtained by clausifying the two implications separately.

Claim: FOF should be the fundamental TPTP category, not MIX.

Origins of type 2 and 3 problems

Most work in formalization of mathematics and verification is done with *interactive* theorem provers like Mizar, HOL, PVS, Coq.

Full automation of these tasks is not yet feasible, and perhaps never will be.

However, these complication proofs contain many "trivial" subtasks, and so it's natural to exploit automation here.

These subtasks have two connected characteristics:

- Relatively easy
- Prover must solve them *quickly*

Claim: CASC should have a 'blitz' category.

10 seconds? 5 seconds? 1 second? One could afford to try *thousands of problems* in a reasonable time.

Not just first order logic!

Many of the routine tasks in verification are not pure first order logic.

One common category is linear arithmetic (over $\mathbb{R}$, $\mathbb{Z}$ or $\mathbb{N}$). Others are pure algebraic rearrangement.

Among the most tedious to prove manually, but easy and efficient to automate.

Claim: We need other categories beyond pure first order logic

These might belong in TPTP, or in other suites.

Question: Is it feasible to solve these with first order or equational provers, with suitable axioms?

There is currently a consortium connected with FROCOS trying to collect such problems.

Perhaps we should also consider propositional problems and higher order problems?

Non-first-order examples

For example, the following HOL Light problem arises in floating-point verification:

```
REAL_ARITH
  `a <= x /\ b <= y /\
   abs(x - y) < abs(x - a) /\
   abs(x - y) < abs(x - b) /\
   (b <= x ==> abs(x - a) <= abs(x - b)) /\
   (a <= y ==> abs(y - b) <= abs(y - a))
   ==> (a = b)`;;
```

and the following is a lemma when proving that every positive integer is the sum of four squares:

```
let LAGRANGE_IDENTITY = prove
 (`(w1 pow 2 + x1 pow 2 + y1 pow 2 + z1 pow 2) *
   (w2 pow 2 + x2 pow 2 + y2 pow 2 + z2 pow 2) =
   (w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2) pow 2 +
   (w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2) pow 2 +
   (w1 * y2 - x1 * z2 + y1 * w2 + z1 * x2) pow 2 +
   (w1 * z2 + x1 * y2 - y1 * x2 + z1 * w2) pow 2`,
  REWRITE_TAC[REAL_POW_2] THEN INT_ARITH_TAC);;
```

HOL Light's automated subsystems

The various classes of problems are usually solved by fixed HOL functions, e.g. first order logic problems (with equality) by `MESON_TAC` or `ASM_MESON_TAC`.

For example, two calls of `ASM_MESON_TAC` appear in the proof of the wellfounded recursion theorem:

```
let WF_REC = prove
 (`WF(<<) ==> !H. (!f g x. (!z. z << x ==> (f z = g z))
                           ==> (H f x = H g x))
                  ==> ?f:A->B. !x. f x = H f x`,
  let lemma = prove_inductive_relations_exist
    `!f (x:A). (!z:A. z << x ==> R z (f z :B))
               ==> R x (H f x)` in
  REWRITE_TAC[WF_IND] THEN REPEAT STRIP_TAC THEN
  X_CHOOSE_THEN `R:A->B->bool`
    (ASSUME_TAC o last o CONJUNCTS) lemma THEN
  SUBGOAL_THEN `!x:A. ?!y:B. R x y`
    (fun th -> ASM_MESON_TAC[th]) THEN
  FIRST_ASSUM MATCH_MP_TAC THEN ASM_MESON_TAC[]);;
```

Raw logging

We can modify functions like `ASM_MESON_TAC` so that they first record their problem argument in a global variable and then proceed as usual.

We can then run various proof scripts, collect lots of problems, and then finally output them in some appropriate form.

However, all our formulas are HOL formulas, generally not first order.

For example, the first `ASM_MESON_TAC` above results, if we ignore some irrelevant assumptions, in the following:

$$\begin{array}{l}(\forall a_0\ a_1.\quad R\ a_0\ a_1 \Leftrightarrow \\ \qquad\qquad \exists f.\quad a_1 = H\ f\ a_0 \wedge \\ \qquad\qquad\qquad \forall z.\ z \ll a_0 \Rightarrow R\ z\ (f\ z)) \wedge \\ (\forall x.\ \exists! y.\ R\ x\ y) \Rightarrow \exists f.\ \forall x.\ f\ x = H\ f\ x\end{array}$$

Clearly this is a higher order problem.

HOL-FOL translation

If it's a higher order problem, why does `ASM_MESON_TAC` solve it?

After all, it's a standard implementation of first order model elimination à la PTTP.

It includes a preprocessing step that performs simple first-order reduction tricks.

Roughly, it eliminates currying and inserts explicit "application" operations whenever a function f is used both as a function and an argument:

$$\begin{aligned}(\forall a_0\ a_1.\quad & R(a_0, a_1) \Leftrightarrow \\ & \exists f.\quad a_1 = H(f, a_0) \wedge \\ & \qquad \forall z.\ z \ll a_0 \Rightarrow R(z, @(f, z)) \wedge \\ (\forall x.\ \exists! y.\ R(x, y)) & \Rightarrow \exists f.\ \forall x.\ @(f, x) = H(f, x)\end{aligned}$$

This can now be proved (quite easily) in first order logic with equality.

Improved logging

So, instead of tweaking `ASM_MESON_TAC` so that it records the initial problem, we instead tweak it to record the problems resulting from the initial splitting and first order reduction.

On the plus side, this gives us problems in essentially standard clausal form, with equality eliminated, ready to be spat out in TPTP format and then tackled by any first order prover.

On the negative side, we'd really rather avoid the clausal form transformation and splitting to give the raw first order problem.

However, instead of being a trivial tweak, that would require a bit of reprogramming to separate the two, which are currently intertwined in a slightly involved way.

Linear arithmetic too

We also logged calls to the underlying linear arithmetic package for the reals.

In this case, the problems are typically in fairly standard form and no extensive translation is needed.

Sometimes they involve multiplication and alien terms.

However, after expanding out, simplifying, and doing basic normalization, the problems can be solved by linear arithmetic treating nonlinear and alien terms as atomic.

HOL Light code base (1)

We ran a lot of our standard "codebase" of HOL Light proofs.

This codebase consists of 650,000 lines of HOL proofs, including about 8900 calls to first order automation and 7300 to linear arithmetic, e.g.

- HOL's basic theories (like the wellfounded recursion example above)
- Formalization of real analysis and transcendental functions
- Intel floating-point verification proofs including detailed theories of floating-point numbers
- Development of complex numbers and proof of Fundamental Theorem of Algebra
- Metatheory of logic, e.g. compactness for first order logic, Tarski's theorem on the undefinability of truth.

HOL Light code base (2)

As well as the substantial proof efforts, we re-ran dozens of smaller miscellaneous proofs, e.g.

- Proof that exponentiation has a diophantine representation
- Miscellaneous number theory like the trivial $n = 4$ case of Fermat's last theorem and results on sums of squares
- Results about abstract reduction systems like Newman's Lemma
- Wellfoundedness of the multiset ordering
- Formalization of Dijkstra's weakest precondition results.
- Formalization of some compiler dataflow analysis
- Various cute little facts and puzzles like $(\forall n : \mathbb{N}.\ f(n+1) > f(f(n))) \Rightarrow \forall n : \mathbb{N}.\ f(n) = n$.

The fruits of logging

From running the kind of body of proofs mentioned above, we obtained the following:

- 4837 first order logic problems in TPTP format
- 3528 problems of linear arithmetic

We have not used the linear arithmetic problems yet. But anyone who wants a copy is welcome to have one.

We did make a few experiments with the first order problems, trying a few prominent first order provers.

Note that these prover versions are now a couple of years old, and many now do much better.

The next table gives some of the 'harder' problems, with a time limit of 1000 CPU seconds.

Some results (1/2)

Problem	Clauses	Horn?	Vampire	Otter	SETHEO
4501	50	Non	-	-	17
3879	31	Horn	-	-	6
4552	55	Non	124.78	836.78	221
4422	45	Non	-	-	1
4500	50	Non	-	-	1
3901	32	Horn	-	-	1
3878	31	Horn	-	-	1
3830	30	Non	-	-	1
3829	30	Non	-	-	1
3832	30	Non	-	-	1
3831	30	Non	-	-	1
4601	60	Non	209.77	-	1
4608	62	Non	-	156.78	1
4644	73	Non	115.02	-	1
4650	74	Non	114.71	-	1
4652	74	Non	108.21	-	1
4646	73	Non	107.64	-	1
4651	74	Non	89.98	-	1
3820	30	Non	209.85	-	8
4645	73	Non	88.54	-	1

Some results (2/2)

Problem	Clauses	Horn?	Vampire	Otter	SETHEO
4627	69	Non	82.74	-	1
4653	74	Non	82.40	-	1
4626	68	Non	80.99	-	1
4647	73	Non	80.84	-	1
4219	40	Non	130.19	-	1
4218	40	Non	131.50	-	1
4217	40	Non	131.53	-	1
4318	42	Non	41.83	-	1
4216	40	Non	133.82	-	1
4093	36	Horn	22.01	-	1
4091	36	Horn	21.93	-	1
4690	83	Horn	103.12	-	1
4691	83	Horn	103.15	-	1
3759	29	Horn	377.53	11.52	1
3826	30	Non	41.18	-	1
3280	22	Non	150.69	9.55	1
4458	48	Horn	-	-	1
3828	30	Non	20.45	-	1
3827	30	Non	42.25	-	1
4459	48	Horn	66.62	-	1
3825	30	Non	32.88	-	1

Observations

It was surprising that many provers found lots of the problems quite challenging.

This despite the fact that HOL's own first order prover can solve them all in a few seconds (exceptionally a minute).

In some cases, Vampire rejected formulas because they involved terms that were "too big".

But generally, we see that even very sophisticated provers can be beaten by straightforward algorithms for some problems.

SETHEO solved all the problems very quickly (though the results shown use multiple parallel sessions with different settings).

This isn't surprising, since it is also based on model elimination, like `MESON_TAC`.

The problem of self-selection

The relative success of SETHEO indicates why these tests cannot be used to perform an objective comparison between systems.

There is a problem of self-selection, because the problems were exactly those that could be solved by `MESON_TAC`.

Plenty of failed calls to `MESON_TAC` occur in interaction with HOL, but these are not recorded in the final proof scripts (except occasionally in comments).

What can we do about this?

- Record failed `MESON_TAC` attempts
- Produce logs independent of the use of `MESON_TAC`

Unfortunately, starting now, it would take years to build up a decent stock of examples, so the first alternative could only be a long-term project.

Recording HOL proofs

Another alternative is available. HOL is a strictly foundational LCF-style prover:

- All its proofs are done using 10 very simple low-level primitive rules
- Theorems are a type whose only constructors are these primitive rules

Thus, it is only the work of 15 minutes to modify the system so that theorems include a proof tree giving details of how they were derived, and primitive inference rules set up this field.

We can then find subtrees that involve only "essentially first order" reasoning, regardless of the top-level rules that were invoked by the user, and save these as test cases.

Modifying the datatype of theorems

The standard datatype of theorems is just a list of terms and another term, representing the assumptions and conclusion of a single-conclusion sequent:

```
type thm = Sequent of (term list * term);;
```

We just need to change this to include a proof tree, with an inference rule and some structure of hypothesis theorems:

```
type thm = Sequent of (term list * term) *
                      (string * syn list)
and syn = Thm of thm
        | Term of term
        | Type of hol_type
        | Pair of syn * syn
        | List of (syn list);;
```

Modifying the primitive rules

Similarly, it's easy to update the primitive rules. The rule implementing reflexivity of equality is:

```
let REFL tm =
  Sequent([],mk_eq(tm,tm));;
```

We just need to change this to record how the theorem arose:

```
let REFL tm =
  Sequent(([],mk_eq(tm,tm)),
          ("REFL",[Term tm]));;
```

Because theorems are treated as an abstract data type, this change is completely transparent and all HOL proofs now get recorded.

The only difference is that it runs more slowly and requires huge amounts of memory.

Too low-level

Unfortunately, HOL is *too* strictly foundational!

Its primitive rules involve only the notion of equality, and all the usual logical constants are defined rather than primitive:

$$
\begin{array}{rcl}
\top & = & (\lambda x.\ x) = (\lambda x.\ x) \\
\wedge & = & \lambda p.\ \lambda q.\ (\lambda f.\ f\ p\ q) = (\lambda f.\ f\ \top\ \top) \\
\Rightarrow & = & \lambda p.\ \lambda q.\ p \wedge q = p \\
\forall & = & \lambda P.\ P = \lambda x.\ \top \\
\exists & = & \lambda P.\ \forall Q.\ (\forall x.\ P(x) \Rightarrow Q) \Rightarrow Q \\
\vee & = & \lambda p.\ \lambda q.\ \forall r.\ (p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r \\
\bot & = & \forall P.\ P \\
\neg & = & \lambda t.\ t \Rightarrow \bot \\
\exists! & = & \lambda P.\ \exists P \wedge \forall x.\ \forall y.\ P\ x \wedge P\ y \Rightarrow (x = y)
\end{array}
$$

Recording first order inferences

Thus, what may seem like a logical primitive, e.g. `CONJUNCT1`:

$$\frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p}\ \texttt{CONJUNCT1}$$

actually expands into a number of primitive higher-order equality rules. Moreover, `MESON` itself uses higher-order rewriting to implement Skolemization.

It's very difficult to separate these from 'real' higher order steps.

The solution is to modify the next level of logical rules to create proof trees treating them, and applications of `MESON` itself, as atomic steps, replacing the true proof tree.

This requires a bit more work, since there are about 30 such rules, but once done, it is again completely transparent higher up.

Current work

We have implemented all these proof recordings and a way of discriminating between first-order and non-first-order proof steps.

It is now easy to generate test cases whose proof trees involve a chosen number of primitive (or pseudo-primitive) logical rules.

By increasing this number, we can make more and more difficult problems, and so are no longer restricted to generating "easy" ones.

However, we still need to implement a nice first order reduction without splitting and clausification.

Not very much left to do, but we will need to spend a while on the details, and sifting problems.

Summary

- Traditional test suites for theorem provers tend to concentrate on 'type 1' problems.
- Formalization and verification efforts are a gold-mine of test problems, if only they can be suitably extracted.
- It is relatively easy to simply log the problems dealt with by existing automated subsystems.
- More work, but also more useful, is to extract such test cases from actual proofs without regard to the automation used to prove them.
- The advantage of an LCF prover like HOL is that a few local proof-logging changes automatically propagate to the proof system as a whole.
- Of course, look out for the new problems in next year's CASC and the subsequent TPTP update!