Overview of the Inequalitor

Noam Neer

July 11, 2018

Contents

1 Introduction

The Inequalitor is an experimental (and unfinished) proof assistant for the language of real numbers and special functions. It is a standalone Python based tool with its own logic, but its proofs were meant to be translatable to Isabelle. This document concentrates on its rewriting capabilities, in comparison to Isabelle's.

Isabelle has a few tools for rewriting — the simplifier, Sledgehammer etc. All of them have their limitations. for example,

- Using the simplifier often requires the user to choose which lemma to use, avoid, or reverse. This requires familiarity with large lemma libraries.
- If I understand correctly, the simplifier difficulties with conditional rewriting are the reason why Isabelle departs from mathematical conventions in some places. For example, $1/0$ is defined to be 0. This makes laws like

$$a/b \times c = a \times c/b$$

true unconditionally and eases simplification, but the proofs depend upon this special definition and can't be translated to systems that do adhere to mathematical conventions. There are similar problems with powers.

- Sledgehammer is unpredictable, and can fail on very simple problems. The underlying algorithm is very difficult to explain, and is somewhat non-deterministic.

- Other tools seem to be more specialized, for example to linear arithmetic or rational expressions.

The Inequalitor's rewriting algorithm tries, in its very specific domain, to improve upon Isabelle's tools. Its range of solvable problems seems to be larger, and its user is not required to be familiar with the lemma libraries. The lemma libraries follow mathematical conventions more closely. The algorithm itself is deterministic and can be explained, at least in general.

Like many algorithms in this field the Inequalitor's algorithm is a certain combination of enumeration and local search over the space of all expressions, but the basic steps can be more complex than an application of an identity to a sub-expression. For example, the steps contain :

- expansion : multiplying all out, as in

$$(a+b)(a+b)+1 \longrightarrow a^2+2ab+b^2+1$$

- common denominator :

$$a+1/a \longrightarrow (a^2+1)/a$$

- exponent/log pushdown : simplifications of the form

$$((ab)^c)^d \longrightarrow a^{cd}b^{cd}$$
$$\log(ab^2) \longrightarrow \log(a)+2\log(b)$$

Some of the steps remind of Isabelle's simprocs, ao the algorithm can be inaccurately described as a local search with simprocs. To help conditional rewriting, the Inequalitor keeps track of properties of expressions that were rigorously proved, such as sign, and stores them in an accessible database. Using this database enables the "simprocs" to know if their manipulations of the expressions are mathematically valid, and if so to prove them correctly.

The design of the lemma libraries is also important, especially with respect to powers. Isabelle has two power operators, `^` and `powr`. The first

requires the exponent to be a natural number, the second doesn't but behaves well only if the base is positive. Each has different lemmas describing its properties, where `powr`'s lemmas seem to be less developed. As a result many problems either can't be formulated correctly or are difficult to prove. The Inequalitor has a single power operator that (almost) generalizes both.[1] This simplifies many things, but translating the proofs to Isabelle will require defining a new power operator.

The Inequalitor is not finished. The only special functions currently supported are powers, logarithms and absolute values, and the translation of the proofs to Isabelle is yet to be implemented. But it is developed enough to be presented for initial review.

[1] It only almost generalizes both since they disagree on some values. For example 0^0 equals 1 in the first power and 0 in the second. In the Inequalitor 0^0 is unspecified, which is the right convention for real analysis. For abstract algebra 1 could be more a suitable choice.

2 Some Isabelle Examples

Here I present some examples which are not trivial in Isabelle 2017. "Not trivial" doesn't mean that they are impossible to prove, but that they can't be solved with a single application of a standard command. The commands I consider standard are `auto, simp, (simp add: field_simps), (simp add: divide_simps), try0, try` and Sledgehammer. Using the simplifier with other modifiers is not considered standard, since it requires familiarity with Isabelle's lemma libraries.

Since problems that involve only the four basic operations `+,-,*,/` are usually solvable with `(simp add: field_simps)`, the problems here involve special functions. We start with few problems that involve only `^`, the ordinary power operator (that requires natural numbers as exponents.) Wherever I knew how to solve the problem I added the appropriate command, but it **wasn't found** by the standard commands. Otherwise I just wrote the command `oops`.

1. ```
   lemma "((a::real)+b+1)^(c::nat) * (a+b+1)^2 =
          (1+a+b)^c * (a^2+2*a*b+b^2+2*a+2*b+1)"
   oops
   ```

2. ```
   lemma "((a::real)^(c::nat) + (b::real)^c)^2 =
          a^(2*c) + 2*a^c*b^c + b^(2*c)"
   by (simp add: power2_sum power_even_eq)
   ```

   This command only works when the exponent is 2. If it is changed to 3, I am not sure how to do it in a single command,

   ```
   lemma "((a::real)^(c::nat) + b^c)^3 =
          a^(3*c) + 3*a^(2*c)*b^c + 3*a^c*b^(2*c) + b^(3*c)"
   oops
   ```

3. ```
   lemma "(2::real)^a * 3^(3*a) = (2*3)^a * 3^(2*a)"
   oops
   ```

Next we come to problems involving the real power operator `powr`. `powr` allows real numbers as exponents, but behaves correctly only if the base is positive. Because of that one can't prove

4. ```
   lemma "((a::real) powr 2) = ((-a) powr 2)"
   oops
   (* This is probably unprovable since it isn't
      necessarily true in HOL. *)
   ```

Even if we stick to positive bases, many things that work for ^ don't work for powr. For example

5. ```
   lemma "(a::real)>0 ==> (b::real)>0 ==> a-b>0 ==>
          (a - b) powr 2 =
          (a powr 2) - 2*a*b + (b powr 2)"
   oops
   (* The analogous problem with ^ is solvable by
      sledgehammer. *)
   ```

6. ```
   lemma "((2::real) powr (2*x)) =
          ((4::real) powr (x::real))"
   oops
   (* The analogous problem with ^ is solvable by
      sledgehammer. *)
   ```

Using powr for roots we also have problems,

7. ```
   lemma "(2::real) * ((2::real) powr (1/2)) =
          ((8::real) powr (1/2))"
   oops
   ```

8. ```
   lemma "(x::real)>0 ==>
          (((1+x)*(1+x)*(1+x)) powr (1/3)) = 1+x"
   oops
   ```

And the interaction between the two power operators is also problematic,

9. ```
   lemma "(x::real)>0 ==> (((1+x)^2) powr (1/2)) = 1+x"
   oops
   (* Here sledgehammer either didn't finish, or suggested a
      command that didn't work. *)
   ```

Next come few examples with logarithms and absolute values.

10. ```
    lemma "(ca::real)~=0 ==>
           (cb::real)~=0 ==>
           ln(ca^100)*ln(cb^2) = ln(ca^2)*ln(cb^100)"
    oops
    ```

    In this case Sledgehammer was able to solve it when the 100 was replaced by 4 :

```
lemma "(ca::real)~=0 ==>
       (cb::real)~=0 ==>
       ln(ca^4)*ln(cb^2) = ln(ca^2)*ln(cb^4)"
by (smt ln_realpow
        mult.commute numeral_Bit0
        power2_less_eq_zero_iff
        power_add power_even_eq
        semiring_normalization_rules(16)
        semiring_normalization_rules(36))
```

Sledgehammer was also able to solve it when ^ was replaced by `powr` :

```
lemma "(ca::real)~=0 ==>
       (cb::real)~=0 ==>
       ln(ca powr 100)*ln(cb powr   2) =
       ln(ca powr   2)*ln(cb powr 100)"
by (simp add: powr_def)
```

But tracing the simplifier shows that it succeeded by rewriting both to `200*ln(ca)*ln(cb)`, which in ordinary mathematics is ill defined since `ca,cb` are not known to be positive. This was possible since `(?x powr ?a)` is defined as

```
if ?x = 0 then 0 else exp (?a * ln ?x)
```

which again is not true in ordinary mathematics. If `powr` was defined in a more standard way, this proof wouldn't work.

11.
```
lemma "(x::real)>0 ==> y>0 ==>
       (ln(x/y))^2 + (ln(y/x))^2 + 2*(ln(x/y))*(ln(y/x)) =
       0"
oops
(* Here sledgehammer suggested commands thet didn't
   work. *)
```

12.
```
lemma "(x::real)>0 ==> y>0 ==>
       2*(ln(\<bar>x+y\<bar>)) = (ln (x^2+2*x*y+y^2))"
oops
```

3 Some Inequalitor Examples

All the problems presented before can be solved with a single Inequalitor's command, as demonstrated here. The Inequalitor is a Python tool which is loaded from the Python interpreter's command line, and here we assume everything was already installed and loaded properly.

A proof is started by creating a proof object,

```
>>> prf = ie_proof()
```

Proving that one expression equals the other is done by calling the simplification method. We demonstrate it with problem 4, which was

```
lemma "((a::real) powr 2) = ((-a) powr 2)"
```

Here we just call

```
>>> prf.uc_simp( ca**2, (-ca)**2 )
```

`ca,cb,cc,cd` are four predefined logical constants. The Inequalitor has no types, and everything is assumed to be real. It does recognize integers, but as reals satisfying a certain formula. In this case the simplifier prints

```
global, ie_proof_index(89)        : ca**2 = (-ca)**2
ie_proof_index(89)
```

Which means it was able to prove the identity, and it is the 90'th formula in the proof.

Next we go back to problem 2,

```
lemma "((a::real)^(c::nat) + (b::real)^c)^2 =
       a^(2*c) + 2*a^c*b^c + b^(2*c)"
```

Now as it is written this lemma doesn't hold in the Inequalitor, since when `ca,cb,cc` are 0 the powers are unspecified (which is the right convention for real analysis.) Hence we'll add the assumption that `cc` is positive. The logical formulas stating that `cc` is positive integer are created as follows,

```
>>> ie_intP(cc)
(ie_formula)  intP(cc)
>>> ie_gt(cc,0)
(ie_formula)  cc>0
```

In order to use them in the proof we must create a proof object with these as its assumptions, which is done by

```
>>> prf = ie_proof( ie_intP(cc), ie_gt(cc,0) )
global, ie_proof_index('assm0')   : intP(cc)
global, ie_proof_index('assm1')   : cc>0
```

Now we can call the simplifier,

```
>>> prf.uc_simp( (ca**cc+cb**cc)**2,
                 ca**(2*cc) + 2*ca**cc*cb**cc + cb**(2*cc) )
global, ie_proof_index(104)       :
    (ca**cc+cb**cc) ** 2  =
    ca**(2*cc) + 2*ca**cc*cb**cc + cb**(2*cc)
ie_proof_index(104)
```

In this way all the problems can be solved, and sometimes more difficult variants of them. For example, problem 8

```
lemma "(x::real)>0 ==>
       (((1+x)*(1+x)*(1+x)) powr (1/3)) = 1+x"
```

can be proved in a slightly more difficult form,

```
>>> prf = ie_proof( ie_gt(ca,0) )
global, ie_proof_index('assm0')   : ca>0
>>> prf.uc_simp( (1+3*ca+3*ca**2+ca**3)**ie_rat(1,3), 1+ca )
global, ie_proof_index(666)       :
    (1 + 3*ca + 3*ca**2 + ca**3) ** r(1/3)  =  1+ca
ie_proof_index(666)
```

4 The Inequalitor's Power Operator

The expression `ca**cb` is unspecified unless at least one of the following holds :

1. `cb` is a positive integer.
2. `cb` is an integer and `ca` is non-zero.
3. `cb` is positive and `ca` is non-negative.
4. `ca` is positive.

These conditions are not disjoint. If at least one holds and the power is specified, its value is identical to the ordinary mathematical value.

In some mathematical texts expressions like $c_a^{1/3}$ are well defined for every c_a. In the Inequalitor they are specified only if $c_a \geq 0$. So the Inequalitor doesn't follow all mathematical conventions precisely, but since it is "less" specified than ordinary mathematics, Inequalitor proofs are (potentially) translatable to any system that follows them.

In the past I started developing an analogous power operator in Isabelle, based on the definition

```
(* mathematical real power operator *)

definition mrpow :: "real ==> real ==> real"
    (infixr "**" 80)
    where "x ** y ==
        if   x>0
        then (x powr y)
        else (if   x=0
              then (if   y>0
                    then 0
                    else (THE z::real. False))
              else (if   y \<in> Ints
                    then (if   y >= 0
                          then    x ^(nat(  floor y))
                          else (1/x)^(nat(- floor y)))
                    else (THE z::real. False)
                   )
             )"
```

I didn't finish proving all the necessary properties of this operator, but it can be done.