

# Implicational rewriting for HOL4

## User manual

Vincent Aravantinos

`vincent.aravantinos@fortiss.org`

Analysis and Design of Dependable Systems, fortiss GmbH, Munich, Germany\*\*

### 1 Introduction.

This document is the user manual of the “`impcnv`” HOL4 library. It provides essentially three tactics:

- `IMP_REWRITE_TAC`,
- `TARGET_REWRITE_TAC`,
- `HINT_EXISTS_TAC`

The most useful ones are `IMP_REWRITE_TAC` and `TARGET_REWRITE_TAC`. *These tactics are very powerful and many proofs end up being combinations of these two tactics only.*

The general objective of these tactics is to reduce the distance between the informal reasoning of the user, and the formal reasoning of the theorem prover. As often, this is done through automation. Contrarily to the usual automation developments, we do not target the automation of *complex* reasoning (e.g., arithmetic, inductive, SMT reasoning, etc.), but instead focus on intuitively *simple* reasoning steps which translate into complex tactics sequences. The underlying rationale is to leave complex reasoning to the human, and simple reasoning to the machine (which is not the case currently, since a lot of informally simple reasoning still ends up being formally complex).

More concretely, these tactics have a common point: they avoid writing terms (or expressions related to terms, like paths) explicitly in proof scripts. This happens generally when using `...by ...`, `EXISTS_TAC`, or `GEN_REWRITE_TAC`. The motivation behind this is that such an explicit piece of information is tedious and time-consuming to devise and to write. Another disadvantage of writing explicitly these sorts of information is that it yields very fragile proofs: definition changes, goal changes, name changes, etc., can break the tactic and therefore the whole script. We basically provide here heuristics to automate the generation of such information.

*Note:* This library is ported directly from its original HOL Light counterpart. Unfortunately I do not have any HOL4 code basis to test it exhaustively so the absence of bugs is not guaranteed at all. Feel free to improve.

---

\*\* formerly: Hardware Verification Group, Concordia University, Montreal, Canada

## 2 Installation

*In short:*

1. in the library directory: `.../HOL4-imp-conv> Holmake`
2. in HOL4:

```
- app load ["TargetRewrite"];  
- open ImpRewrite TargetRewrite;
```

*In detail:* First compile the modules using Holmake:

```
.../HOL4-imp-conv> Holmake  
Compiling FoNets.sml  
Compiling ImpConvUtils.sml  
Compiling ImpConv.sml  
Compiling ImpRewrite.sml  
Compiling TargetRewrite.sml  
.../HOL4-imp-conv>
```

Then to use the library, type in HOL4:

```
- app load ["TargetRewrite"];  
<<HOL message: inventing new type variable names: 'a>>  
metis: r[+0+4]#  
> val it = () : unit
```

And finally open the necessary structures:

```
- open ImpRewrite TargetRewrite;
```

which will yield a lot of output, among which the most important are the following:

```
...  
val IMP_REWRITE_TAC = fn :  
  thm list -> term list * term ->  
  (term list * term) list * (thm list -> thm)  
...  
val TARGET_REWRITE_TAC = fn :  
  thm list -> thm -> term list * term ->  
  (term list * term) list * (thm list -> thm)  
...  
val HINT_EXISTS_TAC = fn :  
  term list * term -> (term list * term) list * (thm list -> thm)
```

### 3 IMP\_REWRITE\_TAC

**Informal specification:** given a theorem of the form:

$$\forall x_1 \dots x_n. P \Rightarrow \forall y_1 \dots y_m. l = r$$

*implicational rewriting* replaces any occurrence of  $l$  by  $r$  in the goal, *even if  $P$  does not hold*. This may involve adding some propositional atoms (typically instantiations of  $P$ ) or existentials, but in the end, you are (almost) sure that  $l$  is replaced by  $r$ . Note that  $P$  can be “empty”, in which case implicational rewriting is just rewriting.

*Note:* We use only first-order matching because higher-order matching happens to match “too much”. An improvement would be to define a second version of the tactic using higher-order matching.

*Remark 1.* Contrarily to `REWRITE_TAC` or `FULL_SIMP_TAC`, the goal obtained by using implicational rewriting is generally *not* equivalent to the initial goal. This is actually what makes this tactic so useful: applying only “reversible” reasoning steps is quite a big restriction compared to all the reasoning steps that could be achieved (and often wanted).

**Tactic:** `IMP_REWRITE_TAC : thm list → tactic`

Given a list of theorems  $[\text{th}_1; \dots; \text{th}_k]$  of the form  $\forall x_1 \dots x_n. P \Rightarrow \forall y_1 \dots y_m. l = r$ , `IMP_REWRITE_TAC`  $[\text{th}_1; \dots; \text{th}_k]$  applies implicational rewriting using all theorems.

*Example 1.*

```
- app load ["realTheory"];

- realTheory.REAL_DIV_REFL;
> val it =
  |- !x. x <> 0 ==> (x / x = 1)
    : thm
- g ' !a b c. a < b ==> ((a - b) / (a - b) * c = c) ' ;
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !a b c. a < b ==> ((a - b) / (a - b) * c = c)

    : proofs
- e(IMP_REWRITE_TAC[realTheory.REAL_DIV_REFL]);
OK..
1 subgoal:
```

```

> val it =

    !a b c. a < b ==> (1 * c = c) /\ a - b <> 0

    : proof

```

**Use:** Allows to make some progress when `REWRITE_TAC` or `FULL_SIMP_TAC` cannot. Namely, if the precondition  $P$  cannot be proved automatically, then these classic tactics cannot be used, and one must generally add the precondition explicitly using `...by ...` or `SUBGOAL_THEN`. `IMP_REWRITE_TAC` allows to do this automatically. Additionally, it can add this precondition deep in a term, actually to the deepest where it is meaningful. Thus there is no need to first use `REPEAT STRIP_TAC` (which often forces to decompose the goal into subgoals whereas the user would not want to do so).

`IMP_REWRITE_TAC` can also be used like `MATCH_MP_TAC`, but, again, deep in a term. Therefore you can avoid the common preliminary `REPEAT STRIP_TAC`.

The only disadvantages w.r.t. `REWRITE_TAC`, `FULL_SIMP_TAC` and `MATCH_MP_TAC` are that `IMP_REWRITE_TAC` uses is generally a little bit slower.

#### Bonus features:

- A theorem of the form  $\forall x_1 \dots x_n. l = r$  is turned into  $\forall x_1 \dots x_n. true \Rightarrow l = r$  (this is the reason why `IMP_REWRITE_TAC` can be used as a replacement for `REWRITE_TAC` and `FULL_SIMP_TAC`).

Therefore, in Example 1, we can actually achieve more in one step:

*Example 2.*

```

- g '!a b c. a < b ==> ((a - b) / (a - b) * c = c)';
> val it =
    Proof manager status: 1 proof.
    1. Incomplete goalstack:
        Initial goal:

            !a b c. a < b ==> ((a - b) / (a - b) * c = c)

        : proofs
- e(IMP_REWRITE_TAC[realTheory.REAL_DIV_REFL,
    realTheory.REAL_MUL_LID,realTheory.REAL_SUB_0]);
OK..
1 subgoal:
> val it =

    !a b. a < b ==> a <> b

    : proof

```

And one can easily conclude with:

```

- e(IMP_REWRITE_TAC[realTheory.REAL_LT_IMP_NE]);
OK..

Goal proved.
|- !a b. a < b ==> a <> b
> val it =
    Initial goal proved.
    |- !a b c. a < b ==> ((a - b) / (a - b) * c = c)
    : proof
- A theorem of the form  $\forall x_1 \dots x_n. P \Rightarrow \forall y_1 \dots y_m. Q$  is turned into  $\forall x_1 \dots x_n. P \Rightarrow \forall y_1 \dots y_m. Q = true$  (this is the reason why IMP_REWRITE_TAC can be used as a replacement for MATCH_MP_TAC).
As mentionned, this is actually a deep MATCH_MP_TAC, consider for instance:
Example 3.
- g ' !a b:real. 0 < a - b ==> ~(b = a) ' ;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete goalstack:
        Initial goal:

        !a b. 0 < a - b ==> b <> a

        : proofs
- e(IMP_REWRITE_TAC[realTheory.REAL_LT_IMP_NE]);
OK..
1 subgoal:
> val it =

    !a b. 0 < a - b ==> b < a

    : proof
Actually the goal can be completely proved just by:
- e(IMP_REWRITE_TAC[realTheory.REAL_LT_IMP_NE,
    realTheory.REAL_SUB_LT]);
OK..

Goal proved.
|- !a b. 0 < a - b ==> b < a
> val it =
    Initial goal proved.
    |- !a b. 0 < a - b ==> b <> a
    : proof
(of course on this simple example, it would actually be enough to use
FULL_SIMP_TAC)

```

- A theorem of the form  $\forall x_1 \cdots x_n. P \Rightarrow \forall y_1 \cdots y_m. \neg Q$  is turned into  $\forall x_1 \cdots x_n. P \Rightarrow \forall y_1 \cdots y_m. Q = \text{false}$ ;
- A theorem of the form  $\forall x_1 \cdots x_n. P \Rightarrow \forall y_1 \cdots y_k. Q \cdots \Rightarrow l = r$  is turned into  $\forall x_1 \cdots x_n, y_1 \cdots y_k, \cdots P \wedge Q \wedge \cdots \Rightarrow l = r$ ;
- A theorem of the form  $\forall x_1 \cdots x_n. P \Rightarrow (\forall y_1^1 \cdots y_k^1. Q_1 \cdots \Rightarrow l_1 = r_1 \wedge \forall y_1^2 \cdots y_k^2. Q_2 \cdots \Rightarrow l_2 = r_2 \wedge \cdots)$  is turned into the list of theorems  $\forall x_1 \cdots x_n, y_1^1 \cdots y_k^1, \cdots P \wedge Q_1 \wedge \cdots \Rightarrow l_1 = r_1, \forall x_1 \cdots x_n, y_1^2 \cdots y_k^2, \cdots P \wedge Q_2 \wedge \cdots \Rightarrow l_2 = r_2, \dots$ ;

All these operations are combined. In practice, this entails that *several deduction steps can be applied using IMP\_REWRITE\_TAC with just a big list of theorems*, as show the examples above.

### 3.1 Variant: SEQ\_IMP\_REWRITE\_TAC

**Tactic:** `SEQ_IMP_REWRITE_TAC : thm list → tactic`

Same as `IMP_REWRITE_TAC` but uses the provided theorems *sequentially* instead of simultaneously: given a list of theorems  $[\text{th}_1; \dots; \text{th}_k]$ , `SEQ_IMP_REWRITE_TAC`  $[\text{th}_1; \dots; \text{th}_k]$  applies as many implicational rewriting as it can with  $\text{th}_1$ , then with  $\text{th}_2$ , etc. When  $\text{th}_k$  is reached, start over from  $\text{th}_1$ . Repeat till no more rewrite can be achieved.

*Example 4.* Recall example 2:

```
- g '!a b c. a < b ==> ((a - b) / (a - b) * c = c)';
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !a b c. a < b ==> ((a - b) / (a - b) * c = c)

    : proofs

- e(IMP_REWRITE_TAC[realTheory.REAL_DIV_REFL,
  realTheory.REAL_MUL_LID,realTheory.REAL_SUB_0,
  realTheory.REAL_LT_IMP_NE]);
OK..
1 subgoal:
> val it =

  !a b. ~(a < b)

  : proof
```

But with `SEQ_IMP_REWRITE_TAC`, the same sequence of theorems solves the goal:

```

e(SEQ_IMP_REWRITE_TAC[realTheory.REAL_DIV_REFL,
  realTheory.REAL_MUL_LID,realTheory.REAL_SUB_0,
  realTheory.REAL_LT_IMP_NE]);
OK..
> val it =
  Initial goal proved.
|- !a b c. a < b ==> ((a - b) / (a - b) * c = c)
: proof

```

**Use:** This addresses a problem which happens already with `REWRITE_TAC` or `FULL_SIMP_TAC`: one generally rewrites with one theorem, then with another, etc. and, in the end, once every step is done, (s)he packs everything in a list and provides this list to `IMP_REWRITE_TAC`; but it then happens that some surprises happen at this point because the simultaneous use of all theorems does not yield the same result as their subsequent use. A usual solution is simply to decompose the call into two calls by identifying manually which theorems are the source of the unexpected behavior when used together. Or one can simply use `SEQ_IMP_REWRITE_TAC`. Note that this is however a lot slower than `IMP_REWRITE_TAC`, therefore I advise to first use `IMP_REWRITE_TAC` and if it does not work like the sequential use of single implicational rewrites then use `SEQ_IMP_REWRITE_TAC`.

### 3.2 Variant: CASE\_REWRITE\_TAC

**Informal specification:** given a theorem of the form:

$$\forall x_1 \dots x_n. P \Rightarrow \forall y_1 \dots y_m. l = r$$

*case rewriting* replaces any atom  $A$  containing an occurrence of  $l$  by  $(P \Rightarrow A[l \rightarrow r]) \wedge (\neg P \Rightarrow A)$ .

*Example 5.*

```

- g '!a b c. a < b ==> ((a - b) / (a - b) * c = c)';
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !a b c. a < b ==> ((a - b) / (a - b) * c = c)

    : proofs

- e(CASE_REWRITE_TAC realTheory.REAL_DIV_REFL);
OK..
1 subgoal:

```

```

> val it =

!a b c.
  a < b ==>
    (a - b <> 0 ==> (1 * c = c)) /\
    ((a - b = 0) ==> ((a - b) / (a - b) * c = c))

: proof

```

**Tactic:** `CASE_REWRITE_TAC : thm → tactic`

Same usage as `IMP_REWRITE_TAC` but applies case rewriting instead of implicational rewriting. Note that it takes only one theorem since in practice there is seldom a need to apply this tactic subsequently with several theorems.

**Use:** Similar to `IMP_REWRITE_TAC`, but instead of assuming that a precondition holds, one just wants to make a distinction between the case where this precondition holds, and the one where it does not.

#### 4 TARGET\_REWRITE\_TAC

**Informal specification:** Given a theorem `th` (the “support theorem”), and another theorem `uh` (the “target theorem”), *target rewriting* generates all the goals that can be obtained by rewriting with `th`, until it becomes possible to rewrite with `uh`.

To understand better the difference with `REWRITE_TAC` and the need for a target theorem, consider a goal  $g$  where more than one subterm can be rewritten using `th`: with `REWRITE_TAC`, all such subterms are rewritten simultaneously; whereas, with `TARGET_REWRITE_TAC`, every of these subterms are rewritten *independently*, thus yielding as many goals. If one of these goals can be rewritten (in one position or more) by `uh`, then the tactic returns this goal. Otherwise, the “one-subterm rewriting” is applied again on every of the new goals, iteratively until a goal which can be rewritten by `uh` is obtained.

**Tactic:** `TARGET_REWRITE_TAC : thm list → thm → tactic`

Given a list of theorems  $[th_1; \dots; th_k]$  of the form  $\forall x_1 \dots x_n. P \Rightarrow \forall y_1 \dots y_m. l = r$ , and a theorem `uh`, `TARGET_REWRITE_TAC [th1; ...; thk] uh` applies target rewriting using the whole list of theorems instead of just one support theorem. As one can see from the required form of the theorem, the tactic uses implicational rewriting instead of just rewriting, which makes it even more powerful.

#### Examples:

*Example 6.*

```
- realTheory.REAL_ADD_RINV;
```



```

> val it = |- !x. x + -x = 0 : thm
- g '!x y:real. -y + x + y = 0';
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !x y. -y + x + y = 0

    : proofs
e(TARGET_REWRITE_TAC[realTheory.REAL_ADD_SYM,realTheory.REAL_ADD_ASSOC]
  realTheory.REAL_ADD_RINV);
OK..
1 subgoal:
> val it =

  !x. 0 + x = 0

  : proof

```

*Example 7.* The following example shows how *implicational* rewriting can even be used:

```

- realTheory.REAL_MUL_RINV;
> val it =
  |- !x. x <> 0 ==> (x * inv x = 1)
  : thm
- g '!x y. inv y * x * y = x';
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !x y. inv y * x * y = x

    : proofs
e(TARGET_REWRITE_TAC[realTheory.REAL_MUL_ASSOC,realTheory.REAL_MUL_SYM]
  realTheory.REAL_MUL_RINV);
OK..
1 subgoal:
> val it =

  !x y. (1 * x = x) /\ y <> 0

  : proof

```

*Example 8.* Let us finally consider an example which does not involve associativity and commutativity. Consider the following goal:

```
- app load ["complexTheory"];
> val it = () : unit

- g '!z. modu (conj z) = modu z';
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !z. modu (conj z) = modu z

    : proofs
```

A preliminary step here is to decompose the left-side  $z$  into its polar coordinates, i.e., to turn it into  $|z|.e^{i \cdot \arg(z)}$ . This can be done by applying the following theorem:

```
- GSYM complexTheory.EULER_FORMULE;
> val it =
  |- !z. z = modu z * exp (i * arg z)
  : thm
```

But using standard rewriting would rewrite both sides and would not terminate (or actually, in the current implementation of `REWRITE_TAC`, simply would not apply). Instead we can use `TARGET_REWRITE_TAC` by noting that we actually plan to decompose into polar coordinates with the intention of using `CONJ_SCALAR_LMUL` afterwards, which yields:

```
- complexTheory.CONJ_SCALAR_LMUL;
> val it =
  |- !k z. conj (k * z) = k * conj z
  : thm

- e(TARGET_REWRITE_TAC[GSYM complexTheory.EULER_FORMULE]
  complexTheory.CONJ_SCALAR_LMUL);
OK..
1 subgoal:
> val it =

  !z. modu (modu z * conj (exp (i * arg z))) = modu z

  : proof
```

**Use:** This tactic is useful each time someone does not want to rewrite a theorem everywhere or if a rewriting diverges. Therefore, it can replace most calls to `ONCE_REWRITE_TAC` or `GEN_REWRITE_TAC`: most of the time, these tactics are used to control rewriting more precisely than `REWRITE_TAC`. However, their use is tedious and time-consuming whereas the corresponding reasoning is not complex. In addition, even when the user manages to come out with a working tactic, this tactic is generally very fragile. Instead, with `TARGET_REWRITE_TAC`, the user does not have to think about the low-level control of rewriting but just gives the theorem which corresponds to the next step in the proof (see examples): this is extremely simple and fast to devise. Note in addition that, contrarily to an explicit (and therefore fragile) path, the target theorem represents a reasoning step which has few chances to change in further refinements of the script.

## 5 HINT\_EXISTS\_TAC

**Informal specification:** Given a goal which contains some subformula of the form  $\exists x. \dots \wedge Px \wedge \dots$  in a context where  $P t$  holds for some  $t$ , then instantiates  $x$  with  $t$ .

*Note:* it is enough that just  $P t$  holds, not the complete existentially quantified formula. As the name suggests, we just use the context as a “hint” but it is (in most general uses) not sufficient to solve the existential completely: if this is doable automatically, then other techniques can do the job in a better way (typically `MESON_TAC`).

**Tactic:** `HINT_EXISTS_TAC : tactic`

Given a goal which contains some subformula of the form  $\exists x_1 \dots x_k. P_1 y_1^1 \dots y_{m_1}^1 \wedge \dots \wedge P_n y_1^n \dots y_{m_n}^n$  in a context where  $P_i t_1 \dots t_{m_i}$  holds for some  $t_1, \dots, t_{m_i}$ , then instantiates  $x_1^i, \dots, x_{m_i}^i$  with  $t_1, \dots, t_{m_i}$ . The “context” consists in the assumptions or in the premisses of the implications where the existential subformula occurs.

*Example 9.* - `g ' !P Q R S. P 1 /\ Q 2 /\ R 3 ==> ?x y. P x /\ R y /\ S x y';`  
`<<HOL message: more than one resolution of overloading was possible>>`

`> val it =`

`Proof manager status: 1 proof.`

`1. Incomplete goalstack:`

`Initial goal:`

`!P Q R S. P 1 /\ Q 2 /\ R 3 ==> ?x y. P x /\ R y /\ S x y`

`: proofs`

`- e HINT_EXISTS_TAC;`

`OK..`

`1 subgoal:`

```

> val it =

      !P Q R S. P 1 /\ Q 2 /\ R 3 ==> S 1 3

      : proof

```

**Use:** When facing an existential goal, it happens often that the context “suggests” a candidate to be a witness. In many cases, this is because the existential goal is partly satisfied by a proposition in the context. However, often, the context does not allow to automatically prove completely the existential using this witness. Therefore, usual automation tactics are useless. Usually, in such circumstances, one has to provide the witness explicitly. This is tedious and time-consuming whereas this witness can be found automatically from the context, this is what this tactic allows to do.