Implicational rewriting
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 "impconv" HOL Light 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 `SUBGOAL_THEN`, `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.

Installation. To make use of these tactics, just type in the following inside a HOL Light session:

```
> needs "target_rewrite.ml";;
```

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

Big example. The library contains in the directory "example" a complete rewriting of the library `https://github.com/aravantv/HOL-functions-spaces` of complex function spaces by Mohammed Yousri Mahmoud, to demonstrate the usability of our tactics on concrete use cases.

2 IMP_REWRITE_TAC

Informal specification: given a theorem of the form:

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

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 instantations 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 `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` $\rightarrow$ `tactic`
Given a list of theorems $[\mathtt{th_1}; \cdots; \mathtt{th_k}]$ of the form $\forall \mathtt{x_1} \cdots \mathtt{x_n}.\ \mathtt{P} \Rightarrow \forall \mathtt{y_1} \cdots \mathtt{y_m}.\ \mathtt{l} = \mathtt{r}$, `IMP_REWRITE_TAC` $[\mathtt{th_1}; \cdots; \mathtt{th_k}]$ applies implicational rewriting using all theorems.

Example 1.
```
# REAL_DIV_REFL;;
val it : thm = |- !x. ~(x = &0) ==> x / x = &1
# g `!a b c. a < b ==> (a - b) / (a - b) * c = c`;;
val it : goalstack = 1 subgoal (1 total)

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

# e(IMP_REWRITE_TAC[REAL_DIV_REFL]);;
val it : goalstack = 1 subgoal (1 total)

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

Use: Allows to make some progress when `REWRITE_TAC` or `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 `SUBGOAL_THEN` or `SUBGOAL_TAC`. `IMP_REWRITE_TAC` allows to do this automatically. Additionnaly, 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`, `SIMP_TAC` and `MATCH_MP_TAC` are that `IMP_REWRITE_TAC` uses only first-order matching and is generally a little bit slower.

Bonus features:

– A theorem of the form $\forall x_1 \cdots x_n.\, l = r$ is turned into $\forall x_1 \cdots x_n.\, \mathit{true} \Rightarrow l = r$ (this is the reason why `IMP_REWRITE_TAC` can be used as a replacement for `REWRITE_TAC` and `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 : goalstack = 1 subgoal (1 total)

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

#  e(IMP_REWRITE_TAC[REAL_DIV_REFL;REAL_MUL_LID;REAL_SUB_0]);;
val it : goalstack = 1 subgoal (1 total)

`!a b. a < b ==> ~(a = b)`
```
And one can easily conclude with:
```
#  e(IMP_REWRITE_TAC[REAL_LT_IMP_NE]);;
val it : goalstack = No subgoals
```

– A theorem of the form $\forall x_1 \cdots x_n.\, P \Rightarrow \forall y_1 \cdots y_m.\, Q$ is turned into $\forall x_1 \cdots x_n.\, P \Rightarrow \forall y_1 \cdots y_m.\, Q = \mathit{true}$ (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. &0 < a - b ==> ~(b = a)`;;
val it : goalstack = 1 subgoal (1 total)

`!a b. &0 < a - b ==> ~(b = a)`

# e(IMP_REWRITE_TAC[REAL_LT_IMP_NE]);;
```

```
val it : goalstack = 1 subgoal (1 total)

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

Actually the goal can be completely proved just by:

```
# e(IMP_REWRITE_TAC[REAL_LT_IMP_NE;REAL_SUB_LT]);;
val it : goalstack = No subgoals
```

(of course on this simple example, it would actually be enough to use `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 $\forall x_1 \cdots x_n.\, P \Rightarrow \forall y_1 \cdots y_m.\, Q = \mathit{false}$;
– 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 $\forall x_1 \cdots x_n, y_1^1 \cdots y_k^1, \cdots P \wedge Q_1 \wedge \cdots \Rightarrow l_1 = r_1$, $\forall x_1 \cdots x_n, y_1^2 \cdots y_k^2, \cdots P \wedge Q_2 \wedge \cdots \Rightarrow l_2 = r_2$, $\ldots$;

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.

2.1 Variant: `SEQ_IMP_REWRITE_TAC`

Tactic: `SEQ_IMP_REWRITE_TAC` : `thm list` $\rightarrow$ `tactic`
Same as `IMP_REWRITE_TAC` but uses the provided theorems *sequentially* instead of simultaneously: given a list of theorems [$\mathtt{th}_1$; $\cdots$; $\mathtt{th}_k$], `SEQ_IMP_REWRITE_TAC` [$\mathtt{th}_1$; $\cdots$; $\mathtt{th}_k$] applies as many implicational rewriting as it can with $\mathtt{th}_1$, then with $\mathtt{th}_2$, etc. When $\mathtt{th}_k$ is reached, start over from $\mathtt{th}_1$. 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 : goalstack = 1 subgoal (1 total)

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

#  e(IMP_REWRITE_TAC[REAL_DIV_REFL;REAL_MUL_LID;REAL_SUB_0;
  REAL_LT_IMP_NE]);;
val it : goalstack = 1 subgoal (1 total)

`!a b. ~(a < b)`
```

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

```
#  e(SEQ_IMP_REWRITE_TAC[REAL_DIV_REFL;REAL_MUL_LID;REAL_SUB_0;
  REAL_LT_IMP_NE]);;
val it : goalstack = No subgoals
```

Use: This addresses a problem which happens already with `REWRITE_TAC` or `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`.

2.2 Variant: `CASES_REWRITE_TAC`

Informal specification: given a theorem of the form:

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

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 : goalstack = 1 subgoal (1 total)

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

# e(CASE_REWRITE_TAC REAL_DIV_REFL);;
val it : goalstack = 1 subgoal (1 total)

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

Tactic: `CASES_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.

3 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 $[\mathtt{th}_1; \cdots; \mathtt{th}_k]$ of the form $\forall \mathtt{x}_1 \cdots \mathtt{x}_n.\ \mathtt{P} \Rightarrow \forall \mathtt{y}_1 \cdots \mathtt{y}_m.\ \mathtt{l} = \mathtt{r}$, and a theorem uh, TARGET_REWRITE_TAC $[\mathtt{th}_1; \cdots; \mathtt{th}_k]$ 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.

```
# REAL_ADD_RINV;;
val it : thm = |- !x. x + --x = &0
# g `!x y z. --y + x + y = &0`;;
Warning: inventing type variables
val it : goalstack = 1 subgoal (1 total)

`!x y z. --y + x + y = &0`

# e(TARGET_REWRITE_TAC[REAL_ADD_AC] REAL_ADD_RINV);;
val it : goalstack = 1 subgoal (1 total)

`!x. x + &0 = &0`
```

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

```
# REAL_MUL_RINV;;
val it : thm = |- !x. ~(x = &0) ==> x * inv x = &1
# g `!x y z. inv y * x * y = x`;;
Warning: inventing type variables
val it : goalstack = 1 subgoal (1 total)
```

```
`!x y z. inv y * x * y = x`

# e(TARGET_REWRITE_TAC[REAL_MUL_AC] REAL_MUL_RINV);;
val it : goalstack = 1 subgoal (1 total)

`!x y. x * &1 = x /\ ~(y = &0)`
```

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

```
# g `!z. norm (cnj z) = norm z`;;
val it : goalstack = 1 subgoal (1 total)

`!z. norm (cnj z) = norm z`
```

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

```
# ARG;;
val it : thm =
  |- !z. &0 <= Arg z /\
         Arg z < &2 * pi /\
         z = Cx (norm z) * cexp (ii * Cx (Arg z))
```

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 CNJ_MUL afterwards, which yields:

```
# e(TARGET_REWRITE_TAC[ARG] CNJ_MUL);;
val it : goalstack = 1 subgoal (1 total)

`!z. norm (cnj (Cx (norm z)) * cnj (cexp (ii * Cx (Arg z))))
  = norm z`
```

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.

4 HINT_EXISTS_TAC

Informal specification: Given a goal which contains some subformula of the form $\exists x. \cdots \wedge Px \wedge \cdots$ 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 $\exists x_1 \cdots x_k. P_1\ y_1^1\ \cdots\ y_{m_1}^1 \wedge \cdots \wedge P_n\ y_1^n\ \cdots\ y_{m_n}^n$ in a context where $P_i\ t_1\ \cdots\ t_{m_i}$ holds for some $t_1, \cdots, t_{m_i}$, then instantiates $x_1^i, \cdots, x_{m_i}^i$ with $t_1, \cdots, t_{m_i}$. 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`;;
val it : goalstack = 1 subgoal (1 total)

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

# e HINT_EXISTS_TAC;;
val it : goalstack = 1 subgoal (1 total)

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

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.