SSReflect/HOL Light v1.1

Alexey Solovyev

October 29, 2012

Contents

1 Getting Started

1.1 References

1. HOL Light home page
`http://www.cl.cam.ac.uk/~jrh13/hol-light`
2. HOL Light reference manual
`http://www.cl.cam.ac.uk/~jrh13/hol-light/reference.html`
3. HOL Light tutorial
`http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial_220.pdf`
4. Mathematical Components project at MS Research–INRIA
`http://www.msr-inria.inria.fr/Projects/math-components`
5. SSReflect tutorial
`http://www.msr-inria.inria.fr/Projects/math-components/tutorial.pdf`
6. SSReflect reference manual
`http://hal.inria.fr/inria-00258384`
7. The Flyspeck project
`http://code.google.com/p/flyspeck/`

1.2 Introduction

SSReflect is a special language for formalizing mathematics in Coq proof assistant. It was developed by Georges Gonthier at Microsoft Research–INRIA Joint Center. In this document I present my implementation of SSReflect language for doing proofs in HOL Light proof assistant. My system (which I will call SSReflect/HOL Light, and the original language will be called SSReflect/Coq) is not a complete reimplementation of SSReflect in HOL Light. I implemented a subset of the language which I personally think is useful for writing proofs in HOL Light in a more efficient way than with a standard HOL Light proof script.

I assume that a reader is familiar with HOL Light proof assistant. Knowledge of SSReflect/Coq is helpful, but it is not required to understand and use my SSReflect/HOL Light system for creating HOL Light proofs.

1.3 Installation

The requirements to run SSReflect/HOL Light are:

- A Java runtime of version 1.5 or higher.
- A checkpointed version of HOL Light.
- A script which can load a checkpointed version of HOL Light. This script must be on the system path.

The latest distribution of SSReflect/HOL Light can be downloaded at

`http://code.google.com/p/flyspeck/downloads/list`

The distribution includes a compiled version of the source code, the source code itself, and all supporting files.

The installation process is simple: extract the distribution archive into any directory. Then SSReflect/HOL Light can be started with the following command

```
java -jar jHOL-SSReflect.jar [name of a HOL Light loading script]
```

By default, it is assumed that a HOL Light loading script has the name `hol_light`. Any other name can be provided as the command argument. A version of HOL Light loaded by the script must contain `"."` in the `load_path` variable (alternatively, `load_path` may contain the path to the directory where `jHOL-SSReflect.jar` is located).

Known issues:
On my computer, SSReflect/HOL Light does not work with a checkpointed version of HOL Light created with Berkeley Checkpoint/Restart (BLCR). I use DMTCP checkpointing to run HOL Light with SSReflect/HOL Light. I didn't test any other checkpointing software.

1.4 Implementation Notes

SSReflect/HOL Light is implemented in Java and OCaml. The Java part includes a user interface, which can understand and print HOL Light terms, and a translator of SSReflect/HOL Light commands and tactics into OCaml. The OCaml part implements special HOL Light tactics.

The project is located at the `jHOLLight` directory of the Flyspeck repository. The directory `jHOLLight/caml` contains OCaml files of the project. This directory contains the following files related to the OCaml part of the system:

- `raw_printer.hl` – a special HOL Light term converter which produces strings of symbols that can be parsed by the Java part of the system.
- `ssreflect.hl` – special HOL Light tactics.
- `sections.hl` – a simple section mechanism in HOL Light.

The main source code packages are:

- `jHOLLight/src/edu/pitt/caml` – classes which correspond to Caml types and objects.
- `jHOLLight/src/edu/pitt/core` – classes which correspond to HOL Light terms and theorems.
- `jHOLLight/src/edu/pitt/core/parser` – a parser of a special representation of HOL Light terms which is produced by `raw_printer.hl`.
- `jHOLLight/src/edu/pitt/core/printer` – a printer of HOL Light terms.

- `jHOLLight/src/edu/pitt/gui` – UI components for displaying terms and theorems.
- `jHOLLight/src/edu/pitt/ssreflect/gui` – SSReflect/HOL Light user interface.
- `jHOLLight/src/edu/pitt/ssreflect/parser` – SSReflect/HOL Light translator.

1.5 Examples

I translated several basic SSReflect/Coq libraries into SSReflect/HOL Light. These libraries are `ssrnat.vhl` and `seq.vhl` (`.vhl` is the extension which I chose for SSReflect/HOL Light script files) and they can be found in the `Examples` directory of the distribution of SSReflect/HOL Light.

I also experimented with group theory in SSReflect/HOL Light and proved Sylow theorems. The proofs are in the file `Examples/group_sylow.vhl`.

Several results about second order Taylor approximations of univariate and multivariate functions was proved with SSReflect/HOL Light. This theory is used in my HOL Light procedure for verification of non-linear inequalities. Theory files are contained in the Flyspeck repository at `svn/trunk/formal_ineqs/taylor/theory` and they are called `taylor_interval.vhl` and `multivariate_taylor.vhl`.

Two Flyspeck theorems were proved with SSReflect/HOL Light. See `svn/trunk/text_formalization/tame/ssreflect`.

2 IDE

2.1 Overview

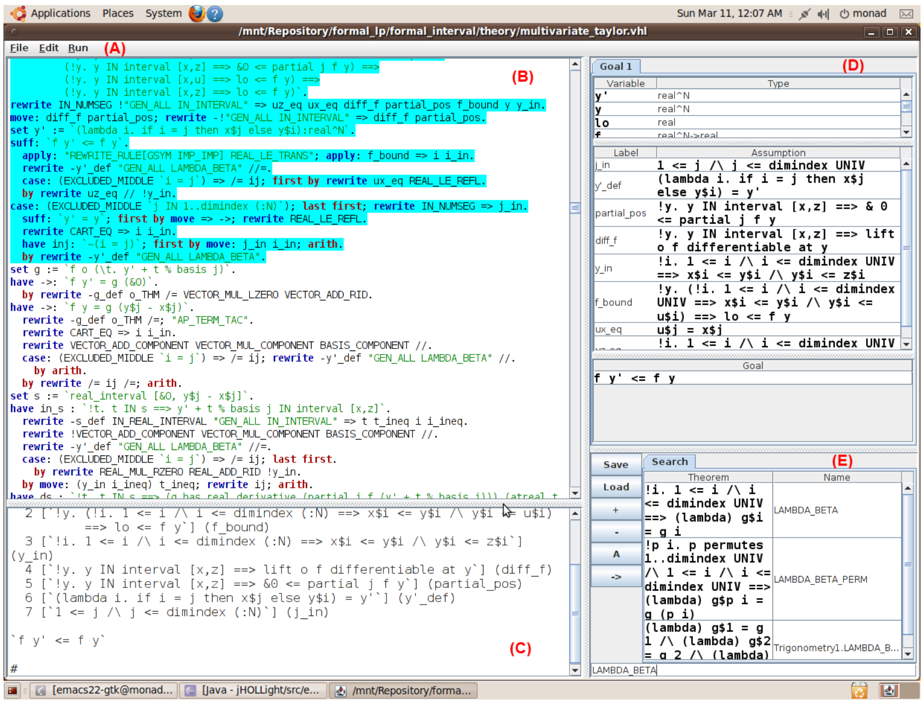
The SSReflect/HOL Light user interface is shown at Figure 1. The main parts of the interface are: the main menu (A), the text editing area (B), the message area (C), the proof status area (D) which includes the list of context variable and the tabbed list of subgoals with assumptions, and the search area (E).

2.2 Main Menu

The menu `File` contains standard commands: create a new file, open an existing file, save changes, save changes in a new file (save as), exit. There is also a list of recently open files.

The menu `Edit` contains the `Highlight` checkbox. If this checkbox is selected, then the source code will be highlighted using SSRefelct/HOL Light rules. Note that the implementation of text highlighting is experimental in the current version of the system and works very slowly for relatively big source files.

The menu `Run` contains the `Compile` command. This command translates an open SSReflect/HOL Light file into a HOL Light file. New HOL Light file will be named as `{name of vhl file}-compiled.hl`.


Applications Places System
Sun Mar 11, 12:07 AM
monad
/mnt/Repository/formal_lp/formal_interval/theory/multivariate_taylor.vhl
File Edit Run
(A)
(B)
(C)
(D)
(E)
Goal 1
Save
Load
+
-
A
->
Search
[emacs22-gtk@monad...
[Java - jHOLLight/src/e...
/mnt/Repository/forma...

```
        (!y. y IN interval [x,z] ==> &0 <= partial j f y) ==>
        (!y. y IN interval [x,u] ==> lo <= f y) ==>
        (!y. y IN interval [x,z] ==> lo <= f y)`.
rewrite IN_NUMSEG !"GEN_ALL IN_INTERVAL" => uz_eq ux_eq diff_f partial_pos f_bound y y_in.
move: diff_f partial_pos; rewrite -!"GEN_ALL IN_INTERVAL" => diff_f partial_pos.
set y' := `(lambda i. if i = j then x$j else y$i):real^N`.
suff: `f y' <= f y`.
  apply: "REWRITE_RULE[GSYM IMP_IMP] REAL_LE_TRANS"; apply: f_bound => i i_in.
  rewrite -y'_def "GEN_ALL LAMBDA_BETA" //=.
  case: (EXCLUDED_MIDDLE `i = j`) => /= ij; first by rewrite ux_eq REAL_LE_REFL.
  by rewrite uz_eq // !y_in.
case: (EXCLUDED_MIDDLE `j IN 1..dimindex (:N)`); last first; rewrite IN_NUMSEG => j_in.
  suff: `y' = y`; first by move => ->; rewrite REAL_LE_REFL.
  rewrite CART_EQ => i i_in.
  have inj: `~(i = j)`; first by move: j_in i_in; arith.
  by rewrite -y'_def "GEN_ALL LAMBDA_BETA".
set g := `f o (\t. y' + t % basis j)`.
have ->: `f y' = g (&0)`.
  by rewrite -g_def o_THM /= VECTOR_MUL_LZERO VECTOR_ADD_RID.
have ->: `f y = g (y$j - x$j)`.
  rewrite -g_def o_THM /=; "AP_TERM_TAC".
  rewrite CART_EQ => i i_in.
  rewrite VECTOR_ADD_COMPONENT VECTOR_MUL_COMPONENT BASIS_COMPONENT //.
  case: (EXCLUDED_MIDDLE `i = j`) => /= ij; rewrite -y'_def "GEN_ALL LAMBDA_BETA" //.
    by arith.
  by rewrite /= ij /=; arith.
set s := `real_interval [&0, y$j - x$j]`.
have in_s : `!t. t IN s ==> y' + t % basis j IN interval [x,z]`.
  rewrite -s_def IN_REAL_INTERVAL "GEN_ALL IN_INTERVAL" => t t_ineq i i_ineq.
  rewrite !VECTOR_ADD_COMPONENT VECTOR_MUL_COMPONENT BASIS_COMPONENT //.
  rewrite -y'_def "GEN_ALL LAMBDA_BETA" //=.
  case: (EXCLUDED_MIDDLE `i = j`) => /= ij; last first.
    by rewrite REAL_MUL_RZERO REAL_ADD_RID !y_in.
  by move: (y_in i_ineq) t_ineq; rewrite ij; arith.
have ds : `!t. t IN s ==> (g has_real_derivative (partial j f (y' + t % basis j))) (atreal t
```

```
  2 [`!y. (!i. 1 <= i /\ i <= dimindex (:N) ==> x$i <= y$i /\ y$i <= u$i)
          ==> lo <= f y`] (f_bound)
  3 [`!i. 1 <= i /\ i <= dimindex (:N) ==> x$i <= y$i /\ y$i <= z$i`]
(y_in)
  4 [`!y. y IN interval [x,z] ==> lift o f differentiable at y`] (diff_f)
  5 [`!y. y IN interval [x,z] ==> &0 <= partial j f y`] (partial_pos)
  6 [`(lambda i. if i = j then x$j else y$i) = y'`] (y'_def)
  7 [`1 <= j /\ j <= dimindex (:N)`] (j_in)

`f y' <= f y`

#
```

Variable	Type
y'	real^N
y	real^N
lo	real
f	real^N->real

Label	Assumption
j_in	1 <= j /\ j <= dimindex UNIV
y'_def	(lambda i. if i = j then x$j else y$i) = y'
partial_pos	!y. y IN interval [x,z] ==> & 0 <= partial j f y
diff_f	!y. y IN interval [x,z] ==> lift o f differentiable at y
y_in	!i. 1 <= i /\ i <= dimindex UNIV ==> x$i <= y$i /\ y$i <= z$i
f_bound	!y. (!i. 1 <= i /\ i <= dimindex UNIV ==> x$i <= y$i /\ y$i <= u$i) ==> lo <= f y
ux_eq	u$j = x$j
uz_eq	!i. 1 <= i /\ i <= dimindex UNIV

Goal
f y' <= f y

Theorem	Name
!i. 1 <= i /\ i <= dimindex UNIV ==> (lambda) g$i = g i	LAMBDA_BETA
!p i. p permutes 1..dimindex UNIV /\ 1 <= i /\ i <= dimindex UNIV ==> (lambda) g$p i = g (p i)	LAMBDA_BETA_PERM
(lambda) g$1 = g 1 /\ (lambda) g$2 = g 2 /\ (lambda)	Trigonometry1.LAMBDA_B...

LAMBDA_BETA

Figure 1: SSReflect/HOL Light user interface.

2.3 Text Editing

SSReflect/HOL Light scripts are simple text file which contain special commands. Each command must be ended with a period symbol. When the period key `.` is pressed during an editing process, then a period symbol will be printed and all unprocessed script text before this period symbol will be processed and the corresponding commands will be executed automatically. The output of the last command will be printed in the message area below the text editing area. If an error occurs during script processing or during command execution, then an error message will be printed in the message area and the process will stop at the command which caused the error. A processed script text will be marked with a special color. This part of the script cannot be modified unless some commands are undone. To undo the last command, press `CTRL + Space`. To undo several processed commands, move the cursor to a desired position inside a processed script text and press `CRTL + Enter`. It will undo all commands up to the command at the cursor position (including this command) and the text corresponding to all these commands will become editable.

If the cursor is located right after a period symbol and the period key is pressed, then the script text will be processed but no additional period symbol will be printed. If you need to print a period symbol without text processing or need to print a period symbol after another period symbol (for example, when you enter a HOL Light term for a numerical segment in the form `a..b`), then use the combination `ALT + .` in order to print a period symbol. In general, it is not necessary to use this combination when entering HOL Light terms containing periods: a single period symbol will be always printed and an incomplete term expression will cause an error which can be ignored.

Known issues:
When a text is processed, then the system will be frozen until the process ends. You need to wait patiently, especially when loading some existing large HOL Light files in the system.

2.4 Theorem Searching

The search area of the user interface helps to find HOL Light theorems which contain special terms or have special names. All search request must be printed in the text field at the bottom of the search area. Each search request has the following form:

search_request	=	**item ... item**
item	=	**'term'** | **sequence_of_symbols**

A search request is a list of search items separated by spaces. Each search item is either a back-quoted HOL Light term expression or a sequence of symbols which does not contain spaces (this sequence of symbols must be unquoted). The search function tries to find all theorems which contain all requested subterms and whose names contain all requested sequences of symbols.

Example: the search request `` `n + m` REAL `` asks for all theorems which contain subterms of the form `` `n + m` `` and whose names contain `REAL`.

Search results can be organized using tabs. To add a new tab, press the `+` button and enter tab's name. To delete an existing tab, select this tab and press the `-` button. To make a tab active, select a tab and press the `A` button. When a tab is active, then search result can be copied to that tab (it is also possible to copy results from one tab to another). To copy a search result, select it in the table of results and then press the `->` button. This action will delete the selected result from the current table and copy it to the active tab table.

It is also possible to save the context of all tabs (excluding the first tab which contains most recent search results). Press the `Save` button and select a file to save the context of tabs. Saved results can be loaded by pressing the `Load` button.

2.5 Producing HOL Light Scripts

An SSReflect/HOL Light script can be translated into a HOL Light script with the `Compile` command in the `Run` menu. This command translates an open SSReflect/HOL Light file into a HOL Light file. New HOL Light file will be named as {`name of vhl file`}`-compiled.hl`.

To load the translated file in HOL Light, it is necessary to load the following files in first: `caml/ssreflect.hl` and `caml/sections.hl`.

3 SSReflect/HOL Light: General Description

There are two main modes in the system: the declaration mode and the proof mode. In the declaration mode, it is possible to enter an arbitrary HOL Light (OCaml) command. In order to do so, a command must be entered in quotes. If a HOL Light command itself contains quotes, then they need to be escaped and written as `\"`. All commands in SSReflect/HOL Light must be ended with a period (`.`) symbol. It includes raw HOL Light commands: a period must be placed after the closing quote. It is not necessary to end quoted HOL Light commands with `;;`.

Examples:

```
"needs \"lib.ml\"".
"let x_def = new_definition `x = 100`".
```

Right now, there is no special mechanism for defining new objects in SSReflect/HOL Light. Standard HOL Light commands for making definitions must be used. Note that it is not possible to cancel or modify a HOL Light definition after it is made. You can go back in SSReflect/HOL Light script but all definitions should be not modified. If you need to modify a definition, then it will be necessary to restart SSReflect/HOL Light.

Standard OCaml comments `(* *)` are available in SSReflect/HOL Light. This comments cannot be nested.

There are two things that can be done in the declaration mode. One thing is to state a lemma and start its proof. Another thing is to use section for organizing lemmas which share common variables and assumptions.

3.1 Lemmas

A simple way to state a lemma and start its proof is the following:

```
Lemma lemma1 : `!x. x + 3 > 2`.
```

This command starts a proof of a lemma with the given HOL Light statement. After a proof is complete, a new lemma will be called `lemma1`. There should be no free variables in the statement of any lemma in SSReflect/HOL Light. The only exception is for variables which are declared as local section variables (more details later). Also, it is allowed to list free variables of a lemma after its name. In this way, the statement could be simpler in many cases:

```
Lemma lemma1 x : `x + 3 > 2`.
```

Right now, it is not possible to specify types of variables which appear after lemma's name. If you need some explicit type for a free variable, then all type information must be entered in the statement:

```
Lemma lemma2 x y : `x > y <=> ~(x = y) /\ x >= (y:real)`.
```

When variables are listed after lemma's name, then they will be automatically generalized in the statement when the lemma is proved.

A proof has the following structure

```
Lemma lemma : ‘statement‘.
Proof.
{tactics}
Qed.
```

The keyword `Proof` may be omitted. The `Qed` keyword ends a proof. This keyword will work only if a lemma is completely proved. It is also possible to abort a proof by entering `Abort` and pressing the period.

3.2 Sections

A new section can be started with the command

```
Section name.
```

Each section must be closed eventually. A section is closed by

```
End name.
```

The `name` should be exactly the same name as the name of open section. Sections may be nested.

Inside a section, it is possible to declare local section variables and assumptions. All these variables and assumptions will be available to all lemmas which appear in that section and in all nested sections (it is true only for lemmas which appear after declarations of local variables and assumptions).

Local variables are declared by

```
Variable x : ‘:num‘.
```

```
Variables x y : ‘:real‘.
```

Types of all local variables must be explicitly given using a back-quoted HOL Light type expression. It is mandatory to put the colon right after the first back quote. The type expression `‘ :real‘` will give an error (this issue will be addressed in next versions of SSReflect/HOL Light). Keywords `Variable` and `Variables` are equivalent.

If a local variable `x` is declared, then it will be possible to use this variable as a free variable in statements of lemmas. Also, its type is fixed, so it is possible to avoid explicit typing. All local variables will be generalized in statements of lemmas when the section where these local variables are declared is closed.

Example:

```
Section Vectors.

Variable x : ‘:real^N‘.

Lemma lemma1 y : ‘x + y = y + x‘.
Abort.

Lemma lemma2 : ‘&0 <= x dot x‘.
Abort.
```

```
(* lemma2 = |- &0 <= x dot (x:real^N) *)

End Vectors.

(* lemma2 = |- !x:real^N. &0 <= x dot x` *)
```

It is also possible to fix a type of a variable with the given name without creating a local variable. Use `Implicit Type` instead of `Variable` to do so.

Example:

```
Section Vectors.

Implicit Type x : `:real^N`.

(* The type of any variable with the name "x" is assumed to be `:real^N` *)
Lemma lemma1 x y : `x + y = y + x`.
Abort.

End Vectors.
```

The keyword `Hypothesis` creates a local section assumption.

```
Hypothesis h1 : `x > 0`.
```

All variables in an assumptions must be bound or they must be local variables declared before. The example above will work only if `x` is a local section variable with the correct type. All lemmas in a section proved after a given local hypothesis share this hypothesis. When the section is closed, then all hypotheses are discharged for all lemmas proved in that section.

Example:

```
Section Sect1.

Variable x : `:num`.
Hypothesis x_gt0 : `x > 0`.

Lemma lemma1 : `~(x = 0)`.
Proof. by move: x_gt0; arith. Qed.

(* lemma1 = x > 0 |- ~(x = 0) *)

Lemma lemma2 y : `y DIV x <= y`.
Proof. by rewrite (DIV_LE lemma1). Qed.

(* lemma2 = x > 0 | !y. y DIV x <= y *)

End Sect1.

(* lemma1 = |- !x. x > 0 ==> ~(x = 0) *)
(* lemma2 = |- !x. x > 0 ==> !y. y DIV x <= y *)
```

4 SSReflect/HOL Light: Proofs

The proof mode is the main mode of SSReflect/HOL Light. As in the declaration mode, it is possible to use raw HOL Light commands and tactics in the proof mode. Again, all such commands must be quoted. The following HOL Light tactics might be helpful in SSReflect/HOL Light: `"ANTS_TAC"`, `"AP_TERM_TAC"`, `"AP_THM_TAC"` (the last two tactics are used because the `congr` tactic of SSReflect/Coq is not completely implemented in SSReflect/HOL Light yet). Sometimes it is also necessary to modify a theorem before it can be used in SSReflect/HOL Light. For instance, it is important that there are no free variables in a theorem statement when it is used as a theorem for rewriting. Most HOL Light theorems satisfy this condition, but in some cases it is wrong. In that situation, the following construction is helpful `"GEN_ALL th"`. SSReflect/HOL Light does not infer types of quoted expressions. It just guesses the type based on a place where a quoted statement is used. This approach could lead to errors, but it is not a big issue. In general, quoted statements are rare inside SSReflect/HOL Light proofs. Eventually, I would like to completely eliminate them by providing appropriate tactics and, probably, special operators (for generalizing theorem variables, etc.).

There are two kinds of objects with which SSReflect/HOL Light tactics work: terms and theorems. Terms can be context variables or raw HOL Light terms. Theorems can be assumptions or HOL Light theorems.

4.1 Terms

All free variables in a proof context (which includes a goal term and all assumptions) are context variables. In general, when a HOL Light term is required, it must be entered as a back-quoted expression. For context variables, it is possible to use their names directly without any quotes. The system will recognize context variables and work with them as with usual terms. When a back-quoted term is used, free variables are not allowed in such a term. The only exception are context variables and special wild cards. Also, if a variable in a term has the same name as some context variable, then it is assumed that these are the same variables and the type of the variable in the term is automatically derived from the context. For example, if the context has a variable `x:real`, then the term `‘x + 2‘` will be rejected since the variable `x` has type `:num` in this term which does not coincide with the type of the context variable `x`. On the other hand, the term `‘x‘` will be accepted and it will be interpreted as `‘x:real‘`. There is no reason to introduce free variables about which nothing is known in the context. In fact, new variables can be introduced with a special tactic `set` which creates abbreviations. This tactic introduces a new context variable and a fact about this variable: its definitional equality.

Back-quoted terms may contain special wild cards. A wild card is just a variable name which starts with `_`. Examples: `_a`, `_1`, or simply `_`. Terms containing wild cards are allowed for several tactics (`rewrite`, `set`, `:`). If a term contains a wild card, then it is treated as a pattern. This pattern is matched against the goal term. In this pattern matching, all wild cards are free variables. If a match is found, then the matched term is used in a tactic. Note that wild cards with the same name must match the same goal subterm. For example, the pattern `` `_ = _` `` matches only subterms of the form `a = a`, and does not match anything like `a = b`.

4.2 Theorems

A proof context contains assumptions. All assumptions in SSReflect/HOL Light must be labelled. In the current version of the system, it is possible to create several assumptions with the same label. This issue will be resolved in next versions of the system. Any assumption can be used as an argument of any tactic when a theorem is required. A labelled assumption behaves in the same way as a named HOL Light theorem.

There are two special ways to construct a new theorem from existing ones. The first construction is creation of conjunctions. If `th1 = |- A` and `th2 = |- B`, then it is possible to write `(th1, th2)`. This expression is a conjunction of two theorems: `(th1, th2) = |- A /\ B`. This construction is not used very often. One example of its application is with the `rewrite` tactic in the following way: `rewrite (th1, th2)`. This tactic first tries to rewrite using the first conjunct (the first theorem) and if it fails, then the second conjunct is tried.

Theorems can be constructed by "applying" a theorem or a term to another theorem. The syntax is the following `(th arg1 ... argn)`. The parentheses are required in most cases. The only exception is the tactic `have` in the form `have a := th arg1 ... argn`.

The "application" works as follows. First of all, `(th arg1 arg2)` is the same as `((th arg1) arg2)`. If a theorem has the form `th = |- P ==> Q` then `arg1` must be a theorem of the form `|- P'` such that it is possible to match `P'` and `P`. The result of `(th arg1)` is `|- Q`. In this case, the "application" is equivalent to `MATCH_MP th arg1`. If a theorem has the form `th = |- !x. P x`, then the argument must be a term `` `y` ``, and `(th arg1)` will be `|- P y` (it is assumed that the type of `` `y` `` is correct). In this case, the "application" is equivalent to ``ISPEC `y` th``. Note that if a theorem has the form `th = |- !x. P x ==> Q x` then it is possible to use "application" with a theorem which can match `P x` directly. It is not required to specialize `x` first.

In fact, the construction `(th arg)` is more general than `MATCH_MP th arg` in many cases. `(th arg)` matches `arg` against all assumptions of `th` until it finds a good match. For instance, suppose `th = |- A /\ B ==> C` and `th_a = |- A`, `th_b = |- B`, then `(th th_a) = |- B ==> C`, `(th th_b) = |- A ==> C`, and `(th th_b th_a) = |- C`. Moreover, it is possible to mix matching of variables and assumptions as shown in examples below.

Examples:

```
th = |- !n. n < 2 ==> n = 0 \/ n = 1
arg = |- 0 < 2
(th arg) = |- 0 = 0 \/ 0 = 1

th = |- !x. &0 < x ==> (!n. &0 < x pow n)
arg = |- &0 < &3
```

```
th `2` = |- !x. &0 < x ==> &0 < x pow 2
th `2` arg = |- &0 < &3 pow 2
th arg `2` = th `2` arg
```

4.3 Chains of Tactics

Tactics can be chained to form a single tactic. Chains of tactics are formed by adding a semicolon between two tactics `tac1; tac2`. In this expression, if `tac1` produces several subgoals, then `tac2` is applied to all produced subgoals. It is possible to apply specific tactics to selected subgoals. It can be done with special tactical `last` and `first` (see the corresponding section) or it can be done with the following construction `tac1; [tac2_1 | ... | tac2_k]`. Here k is the number of subgoals produced by the first tactic. The i-th tactic in square brackets will be applied to the i-th subgoal. In this construction, any tactic may be omitted. For example, the command `tac1; [tac2_1 | | tac2_3]` applies `tac2_1` to the first subgoal and applies `tac2_3` to the third subgoal.

4.4 Introduction `=>`

The introduction tactical `=>` is used for introducing assumptions and for specializing variables.

$$\begin{array}{rcl}
 & & \textbf{tactic} \; => \; \textbf{i-item} \; ... \; \textbf{i-item} \\
\textbf{i-item} & = & \textbf{i-pattern} \mid \textbf{s-item} \mid \textbf{/view} \\
\textbf{s-item} & = & \textbf{/=} \mid \textbf{//} \mid \textbf{//=} \\
\textbf{i-pattern} & = & \textbf{ident} \mid \_ \mid [\textbf{occ-switch}]\textbf{->} \mid [\textbf{occ-switch}]\textbf{<-} \mid \textbf{[i-item} \mid \textbf{...} \mid \textbf{i-item]}
\end{array}$$

The tactic is executed before the introduction. All introductions are processed from left to right.

`s-item` is a simplification operation. `//` is translated into `ASM_REWRITE_TAC[]`, `/=` is translated into `SIMP_TAC[]`, and `//=` is translated into `ASM_SIMP_TAC[]`. Note: `//` is not exactly `ASM_REWRITE_TAC[]`. If `//` cannot solve a subgoal with `ASM_REWRITE_TAC[]`, then it does not change anything in that subgoal.

`/view` is a view which is applied to the top assumption of the goal after performing all previous introduction operations.

`i-pattern` is used for introducing new facts as assumptions and for specialization of generalized variables. It will be explained by examples.

If the goal is `P ==> Q`, then `move => h` will introduce a new assumption `h: P` and the goal becomes `Q`. `move => _` on the same goal will transform the goal into `Q` without introducing a new assumption.

If the goal is `!x. P x ==> Q` then `move => y h` will specialize the variable `x` and call it `y` and then introduce a new assumption `h: P y`. The goal becomes `Q`.

`->` and `<-` can be used for rewriting during the introduction process. The tactic `move: eq_th => ->` is equivalent to `rewrite eq_th`, meanwhile `move: eq_th => <-` is equivalent to `rewrite -eq_th`. More precisely, if the goal is `(a = b) ==> P a` then the tactic `move => ->` transforms the goal into `P b`. The occurrence switch works in the same way as with the `rewrite` tactic.

`[i1 | ... | in]` is a branching `i-pattern` if it is the first `i-item` and the tactic is not `move`. A branching `i-pattern` applies introductions `i1` to the first subgoal generated by the tactic, `in` to the last subgoal generated by the tactic.

`[i1 | ... | in]` is a destructing `i-pattern` if it is not the first `i-item` or if the tactic is `move`. A destructing `i-pattern` executes a case splitting first (in the same way as the `case` tactic) and then behaves like a branching `i-pattern` for subgoals generated by the case splitting.

A tactic of the form `rewrite th => [] []` is equivalent to `rewrite th; case`. It is necessary to use two `i-pattern`'s since the first `i-pattern` is just an empty branching pattern.

If the goal is `A /\ B ==> C` then `move => [h1 h2]` introduces two assumptions `h1: A` and `h2: B` and transforms the goal into `C`. The same effect can be obtained with either `move => [] h1 h2`, `move => [h1] h2`, or `case => h1 h2`.

If the goal is `(?x. P x \/ Q x) ==> R` then `move => [t [hP | hQ]]` produces two goals: the first goal is `R` and it has an assumption `hP: P t`, the second goal is `R` and it has an assumption `hQ: Q t`. This tactic can be written as `case => t; case => [hP | hQ]`.

Known issues:
It is possible to create different variables or assumptions with the same name: `move => h h x x`. This will lead to a situation where one of assumptions or context variables could not be used directly.

4.5 Discharge :

The discharging tactical `:` is used for discharging assumptions and for adding existing theorems as antecedents of a goal.

$$
\begin{array}{rcl}
 & & \textbf{tactic}\,\texttt{:}\,\textbf{d-item}\ \ldots\ \textbf{d-item} \\
\textbf{d-item} & = & [\textbf{occ-switch}](\textbf{term}\ \mid\ \textbf{th})
\end{array}
$$

The tactic before `:` can be any tactic but the most common cases are `move`, `apply`, `case`, `elim`. The tactic is executed after discharging steps.

`move: d1 d2 d3` is equivalent to `move: d3; move: d2; move: d1`. The order of elements is reversed. It is done to make sure that the left-most element `d1` becomes the left-most element in the goal after discharging. In this way, the tactic `move: th1 th2 => h1 h2` creates two new assumptions `h1: conclusion of th1` and `h2: conclusion of th2`.

`move: th` transforms a goal by adding the conclusion of the given theorem as an antecedent of the goal. It is equivalent to `MP_TAC th`.

Assume that `x` is a variable in the context, then `move: x` transforms the goal by generalizing the variable `x`. If the occurrence switch is used, then only specified occurrences of the variable are generalized. All other occurrences are renamed.

Note that `x` will be generalized only in the goal, not in assumptions. If `x` is used in some assumption and need to be generalized in that assumption as well, then this assumption must be discharged before generalization of `x`.

`` move: `term` `` finds all occurrences of the term (which can contain wildcards) in the goal. All these occurrences are replaced with a generated abbreviation, and then the abbreviation is generalized. The occurrence switch specifies which subterms of the goal must be selected.

Examples:

Goal: `a + x = x + 3`

`move: x` transforms the goal into `!x. a + x = x + 3`.

`move: {1}x a` transforms the goal into `!x1 a. a + x1 = x + 3`, where `x1` is an automatically generated name.

`` move: `x + 3` `` transforms the goal into `!x1. a + x = x1`.

4.6 View /

Views in SSReflect/HOL Light provide only a very limited functionality of views in SSReflect/Coq. The primary reason is that there is no difference between Boolean values and propositions in HOL Light.

There are two main forms of views in SSReflect/HOL Light. The first form is `tactic/theorem`. This is equivalent to the following tactic:

`move => top; move: (theorem top); move: top => _; tactic.`

Here `top` is a fresh name. In other words, if a goal is `P ==> Q` and `th = |- P ==> R`, then `move/th` transforms the goal into `R ==> Q`.

The second form of views in SSReflect/HOL Light is `tactic/(_ arg)` where `arg` is either a theorem or a term. This construction can be used if the top goal assumption has the form `P ==> Q` or `!x. P`. If a goal is `(P ==> Q) ==> R` and `th = |- P`, then `move/(_ th)` transforms the goal into `Q ==> R`. If a goal is `(!x. P x) ==> Q`, then `move/(_ y)` transforms the goal into `P y ==> Q` (it is assumed that `y` is a context variable). In the latter case, it is also possible to use raw HOL Light terms: `` move/(_ `y + 2`) ``.

Views can also be used as introduction items after `=>` (see the introduction tactical).

4.7 by

`by` is a tactical which has syntax `by tactic`. Here `tactic` can be a single tactic or a chain of tactics. `by tactic` is equivalent to `tactic; done`. This tactical ensures that a current subgoal is completely proved. It is a good practice to end any subgoal proof with `by` or any other tactic which fails when a goal is not proved (`exact`, `arith`, `done`).

The construction `by []` works in the same way as `done`. The construction `by [tac_1 | ... | tac_k]` is equivalent to `[by tac_1 | ... | by tac_k]`.

4.8 try

`try` tries to apply a tactic and does nothing if the tactic fails.

Syntax: `try tactic`

`tactic` must be a single tactic. If a chain of tactics is required, then all tactics in the chain must be surrounded by parentheses: `try (tac1; ...; tac_n)`.

Example:

`try arith` solves all subgoals which can be solved with `arith`.

4.9 last, first

`last` (`first`) applies a given tactic to the last (first) subgoal in a goal stack.

Syntax: `last tac`, `first tac`

Note that `tac` must be a single tactic or a chain of tactics surrounded by parentheses. For example, the expression

```
case; last rewrite th1; rewrite th2
```

works as follows: `case` is applied, `rewrite th1` is applied to the last subgoal and then `rewrite th2` is applied to all subgoals. On the other hand,

`case; last (rewrite th1; rewrite th2)` applies `rewrite th1; rewrite th2` to the last subgoal only.

When the construction `first by` or `last by` is used, then the behavior is the following:

```
case; last by rewrite th1; rewrite th2
```

applies `rewrite th1; rewrite th2; done` to the last subgoal only (since the chain of these two tactics forms a single argument of the `by` tactical).

4.10 last first, first last

`last first` (`first last`) rotates the goal stack.

Syntax: **last**[**n**] **first**, **first**[**n**] **last**.

By default, the number n is 1.

If a goal stack has subgoals $g_0, g_1, \ldots g_{k-1}$, then `first [n] last` transforms the goal stack to $g_{0+n}, g_{1+n}, \ldots, g_{(k-1)+n}$. Here, $i + n$ is computed modulo k.

If a goal stack has subgoals $g_0, g_1, \ldots, g_{k-1}$, then `last [n] first` transforms the goal stack to $g_{0-n}, g_{1-n}, \ldots, g_{(k-1)-n}$. Here, $i - n$ is computed modulo k.

If there are only two goals in the stack, then `last first` and `first last` are equivalent.

4.11 do

`do` repeats a given tactic.

do [**mult**] [**tactic** | **...** | **tactic**]

mult = [**n**](**!** | **?**)

When a list of parallel tactics is used, then `do [tac1 | tac2 ... | tac_n]` executes the first tactic which is successful.

By default, the tactic is executed only one time. The multiplier `mult` specifies how many times the tactic should be repeated. The key `!` specifies that the tactic must be performed at least one time and then repeated as many times as possible. An explicit number before this key specifies how many times the tactic should be repeated exactly. The key `?` specifies that the tactic must be repeated as many times as possible (even 0 times). An explicit number before this key specifies the maximum number of times the tactic should be repeated. Note: if no explicit number is given then the tactic is repeated at most 10 times for both keys. (This limitation is introduced to prevent infinite loops for some tactics.)

4.12 left, right

`left` transforms a goal `A \/ B` into `A`; `right` transforms a goal `A \/ B` into `B`.

Implementation notes: `left` and `right` correspond to HOL Light tactics `DISJ1_TAC` and `DISJ2_TAC` respectively.

4.13 split

If `split` is applied to a goal `A /\ B`, then two new subgoals are created: `A` and `B`. If `split` is applied to a goal `A <=> B`, then two new subgoals are created: `A ==> B` and `B ==> A`.

Implementation notes: `split` is translated into `split_tac` which is defined as follows

```
let split_tac = FIRST [CONJ_TAC; EQ_TAC];;
```

4.14 exists

`exists` provides witnesses for existential quantifiers.

exists term ... **term**

Examples:

Goal: `?a b. a > SUC b`

`exists ‘3‘ ‘1‘` transforms the goal into `3 > SUC 1`.

`exists x ‘1 + y‘` transforms the goal into `x > SUC (1 + y)` (variables `x` and `y` must appears in the context and have type `:num`).

`exists x` transforms the goal into `?b. x > SUC b`. This will work only if `x` appears in the context and its type is `:num`.

4.15 done

`done` proves trivial goals and fails if it cannot prove a goal.

Implementation notes: `done` is translated into `done_tac` which is defined as follows

```
let done_tac = ASM_REWRITE_TAC[] THEN FAIL_TAC "done";;
```

4.16 arith

`arith` is a tactic for proving basic algebraic statements for natural, real, and integer numbers. This tactics fails if it cannot prove a goal.

Example:

The goal `a + a = &2 * a` is solved by `arith.`

Implementation notes: `arith` is translated into `arith_tac` which is defined as follows

```
let arith_tac = FIRST [ARITH_TAC; REAL_ARITH_TAC; INT_ARITH_TAC];;
```

4.17 move

`move` does virtually nothing. It is used in conjunction with tactical `=>` and `:` for introducing and discharging assumptions.

Implementation notes: `move` is translated into `BETA_TAC`.

4.18 set

`set` introduces an abbreviation.

set name := ‘term‘

The left hand side should be a fresh name, the right hand side is a back-quoted HOL Light term which can contain wild cards `_`. Types of all free variables of the term are instantiated based on the types of context variables.

Wild cards in the term are used for matching the term against the current goal. If a match is found for a subterm in the goal, then the matched term is used in the definition.

If the tactic `set` is successful, then a new equation will be added to the list of assumptions. This equation will have the label `{name}_def` and it will be of the form `term = {name}`.

Examples:

Goal: `SUC (1 + f x y + 2) = 3`

`` set g := `f x y` `` introduces a new assumption `g_def: f x y = g` and changes the goal to `SUC (1 + g + 2) = 3`.

`` set g := `_ + 2` `` introduces a new assumption `g_def: f x y + 2 = g` and changes the goal to `SUC (1 + g) = 3`

4.19 have

`have` introduces a new subgoal.

have [**label**]**: ‘term‘**

This tactic creates a new subgoal defined by the term and adds the same assumption to the original goal with the given label. The label could be omitted. In that case, the statement of the subgoal is added as an antecedent to the original goal.

There is another form of this tactic which can be used for adding an existing theorem into a list of assumption: `have label := thm`

Full syntax of `have` is the following

have (**i-pattern** | **i-item**) [**s-item** | **binder** +] (**: ‘term‘** [**by tac**] | **:= thm**)

A list of binders after the introduction pattern (or item) gives variable names which must be generalized in the subgoal after it is proved. For example, the tactic `` have h1 x: `x = 0 \/ x > 0` `` introduces a new subgoal `` `x = 0 \/ x > 0` `` and adds a new assumption to the original goal `h1: !x. x = 0 \/ x > 0`. The construction `` have ...: `term` by tac `` is equivalent to
`` have ...: `term`; first by tac ``.

Examples:

Goal: `P (1 + x) = r`

`` have: `1 + x = 3` `` creates a new subgoal `1 + x = 3` and changes the original goal to `1 + x = 3 ==> P (1 + x) = r`. Sometimes, it is convenient to use the construction `` have ->: `1 + x = 3` `` which rewrites the original goal with `1 + x = 3`.

`have h1 := ETA_AX f` creates a new assumption with the label `h1` which corresponds to the given theorem. Note that it is not required to put parentheses around `ETA_AX f`.

`have [t [a | b]]: ‘?x. x > 2 \/ x < 3‘` is the same as
`have: ‘?x. x > 2 \/ x < 3‘; last move => [t [a | b]]`.

4.20 suff

`suff` is a variation of `have`.

`suff` switches two goals generated by `have`, i.e., `suff` is equivalent to an application of `first last` after `have`.

4.21 wlog

`wlog` is a variation of `have`.

wlog : id_list / ‘term‘

`id_list` is a list of identifiers (variable names) which can be empty.

`wlog: / ‘Q‘` transforms a goal `‘P‘` into two subgoals `‘(Q ==> P) ==> P‘` and `‘Q‘`.

`wlog: n1 n2 / ‘Q‘` transforms a goal `‘P‘` into two subgoals `‘(!n1 n2. Q ==> P) ==> P‘` and `‘Q‘`.

4.22 case

`case` performs a case analysis on the first antecedent or on the top universally quantified variable of a goal.

The behaviour of `case` depends on the form of a goal.

`case` transforms a goal `A /\ B ==> C` into `A ==> B ==> C`.

`case` transforms a goal `A \/ B ==> C` into `A ==> C` and `B ==> C`.

`case` transforms a goal `(?x. P x) ==> Q` into `!x. (P x ==> Q)`.

`case` transforms a goal `!n:num. P n` into two subgoals `P 0` and `!m. P (SUC m)`.

`case` transforms a goal `!a:(A)list. P a` into two subgoals `P []` and `!h t. P (CONS h t)`.

`case` transforms a goal `!a:A#B. P a` into `!(x:A) (y:B). P (x, y)`.

It is common to combine `case` with `=>` and `:` tacticals.

Examples:

Goal: `(?x. P x) ==> Q`

`case => y hP` changes the goal into `Q` and creates an assumption `hP: P y`.

Goal: `P (x + 2)`

`case: (EXCLUDED_MIDDLE ‘x + 2 = 3‘) => [-> |]` transforms the goal into two subgoals `P 3` and `~(x + 2 = 3) ==> P (x + 2)`.

Implementation notes: `case` is translated into `case_tac`. SSReflect/HOL Light implementation of `case` is not as powerful as the original implementation in Coq/SSReflect. `case_tac` can handle only a predefined number of situations meanwhile the original `case` tactic in Coq/SSReflect works for any inductive type. The current implementation of `case_tac` covers most common situations when this tactic can be applied. It will be pretty easy to extend the current implementation if it becomes necessary.

4.23 elim

`elim` performs an inductive elimination on the top universally quantified variable of a goal.

This tactic works only for variables of `:num` and `:(A)list` types. (It also works for `:bool` and `:A#B` but for these types it is equivalent to `case`.)

`elim` transforms a goal `!n:num. P n` into two subgoals:

`P 0` and `!n. P n ==> P (SUC n)`.

`elim` transforms a goal `!a:(A)list. P a` into two subgoals:

`P []` and `!a0 a1. P a1 ==> P (CONS a0 a1)`.

4.24 apply

`apply` corresponds to `MATCH_MP_TAC`.

`apply` transforms a goal `(P ==> Q') ==> Q` into `P` if `Q'` can be matched to `Q`.

If a goal has the form `(P ==> Q ==> R) ==> X`, then the tactic first tries to match `X` with `Q ==> R`. If this fails, then the antecedent is transformed into `P /\ Q ==> R` and the process is repeated.

There is another form of the tactic: `apply th`. In this case, it is completely equivalent to `MATCH_MP_TAC th`. It was necessary to introduce this form since HOL Light cannot handle assumptions with polymorphic types. Here is an example:

Goal: `!a:(num)list. P a`. Then the tactic `apply list_INDUCT` works and transforms the goal into `P [] /\ (!a0 a1. P a1 ==> P (CONS a0 a1))`. The tactic `apply: list_INDUCT` fails on the goal. The reason is that the tactical `:` adds a new theorem as an antecedent of the goal, and then `apply` tries to use this antecedent to match the original conclusion. Matching is successful, but the matched theorem now has the form `P[:num/:A] |- P[:num/:A]` meanwhile the list of assumptions contains the original polymorphic version of the theorem and the tactic fails.

4.25 exact

This tactic is equivalent (with some limitation) to the construction

```
do [done | by move => top; apply top]
```

where `top` is a fresh name.

`exact` is a common way to finish a subgoal proof.

Implementation notes: `exact` is translated into `exact_tac` which is defined as

```
FIRST [done_tac; DISCH_THEN (fun th -> apply_tac th) THEN done_tac];;
```

4.26 rewrite

`rewrite` uses an equational theorem to change a goal.

		rewrite r-step ... **r-step**
r-step	=	[**r-prefix**] **r-item**
r-prefix	=	[**-**] [**mult**] [**occ-switch**] [[**‘term‘**]]
occ-switch	=	**{**n_1 **...** n_k**}**
r-item	=	**theorem** | **s-item**
s-item	=	/= | // | //=

`rewrite r1 r2` is equivalent to `rewrite r1; rewrite r2`.

The most basic form of the tactic is `rewrite th` where `th` is an equational HOL Light theorem. The tactic tries to find a subterm in the goal which matches the left hand side of the provided theorem. If a matching subterm is found, then the theorem is instantiated such that its left hand side becomes equal to the matching subterm and all instances of the left hand side of the theorem in the goal are replaced with theorem's right hand side. Free variables of the theorem are not instantiated during rewriting.

In fact, `th` could be any theorem. If it is a conjunction of several results, then the tactic tries all conjuncts and it stops after first successful rewriting. If the given theorem is not an equational theorem of the form `|- P` then it is transformed into the theorem `|- P <=> T` before rewriting. A theorem of the form `|- ~P` is transformed into `P <=> F`.

`rewrite` works for theorems with assumptions. If `th = |- P ==> (f = g)`, then `rewrite th` changes all occurrences of `f` to `g` in a goal and introduces a new subgoal `P`.

The `-` switch changes the direction of rewriting. It is equivalent to an application of `GSYM` to a theorem before rewriting.

The occurrence list `occ-switch` specifies which occurrences of a matched term should be rewritten. This list must contain natural numbers separated by spaces.

The term `[‘term‘]` in square brackets must be a back-quoted HOL Light term which may contain wild cards. Types of all free variables in this term are instantiated to the types of the corresponding context variables. If this term is provided, then the tactic finds a matching term in a goal and then searches for rewrites only inside this subterm.

By default, the tactic is executed only one time for the given theorem. It is possible to repeat the rewriting process several times. The exact number of repetitions is controlled by the multiplier `mult` which has the same syntax and behaviour as for the `do` tactic.

It is important to remember that free variables of a given theorem are not instantiated in matching and rewriting processes. If a theorem contains free variables in the left hand side, then they must correspond to the same variables in a goal. Some HOL Light theorems are not generalized, so it is required to use the following construction to work with such theorems

```
rewrite "GEN_ALL REAL_MUL_ASSOC"
```

The fact that free variables of a theorem must coincide with free variables of a goal helps to make rewriting of manually selected subterms. Consider an example:

Goal: `a = &3 * a`

`rewrite -(REAL_MUL_LID ‘&3‘)` transforms the goal into `a = (&1 * &3) * c`, meanwhile `rewrite -REAL_MUL_LID` transforms the goal into `&1 * a = &3 * &1 * a` since the pattern matching follows the left hand side first rule and it finds the first appropriate subterm `‘a‘` and replaces all occurrences of this subterm. Another way to get the first result is to use a pattern: `rewrite -[‘&3‘]REAL_MUL_LID` or `rewrite -[‘& _‘]REAL_MUL_LID`.

`rewrite -{2}REAL_MUL_LID` transforms the goal into `a = &3 * &1 * a` since only the second occurrence of the matched term `‘a‘` is asked to be replaced. `rewrite -{2}(REAL_MUL_LID ‘&3‘)` will not work since there is only one occurrence of the term `‘&3‘` in the goal.

Known issues:
Conditional rewriting could fail when a subterm is rewritten inside an abstraction. This happens because the current implementation of rewriting does not appropriately rename bound variables.

`rewrite ETA_AX` does not simplify subterms of the form `‘\x. f a x‘`. To simplify such a subterm, use `rewr ETA_AX`.

4.27 rewr

`rewr th` is directly translated into `ONCE_REWRITE_TAC[th]`.

This tactic has a limited number of applications. Sometimes `rewrite` fails or does not work in simple cases (e.g., `rewrite ETA_AX`), then `rewr` can be used. `rewr` allows chains of rewrites and it supports multipliers `[n]!` and `[n]?`.

4.28 in

`in` is the localization tactical. Its syntax is

tac in label ... label [*]

The purpose of the `in` tactical is to apply a given tactic to assumptions which names are listed after `in`. If `*` follows the list of assumption names, then the tactic is applied to the goal as well.

More precisely, `tac in h1 h2` works as follows: first of all, the current goal is hidden using an abbreviation (which is mostly equivalent to the tactic `set hidden_goal := ‘_‘`). Then assumptions `h1` and `h2` are discharged and the tactic `tac` is applied. After that, the assumptions are reintroduced and the goal abbreviation is removed. The tactic `tac in h1 h2 *` works in a similar way but it does not hide the goal.

`in` is frequently used with `rewrite`.