The PhoX Proof checker Documentation

Version 0.90

Christophe Raffalli
GAATI, Université de Polynésie Française
Paul Rozière
Equipe PPS, Université Paris VII

June 27, 2024

Contents

Chapter 1

Introduction.

The “PhoX Proof Checker” is an implementation of higher order logic, inspired by Krivine’s type system (see section 1.1), and designed by Christophe Raffalli. PhoX is a Proof assistant based on High Order logic and it is eXtensible.

One of the principle of this proof assistant is to be as user friendly as possible and so to need a minimal learning time. The current version is still experimental but starts to be really usable. It is a good idea to try it and make comments to improve the future releases.

Actually PhoX is mainly a proof editor for higher order logic. It is used this way to teach logic in the mathematic department from “Université de Savoie”.

The implementation uses the Objective-Caml language. You will find in the chapter 10 the instruction to install PhoX.

1.1 Motivation.

The aim of this implementation was first to implement Krivine’s Af2 [1, 2, 3] type system, that is a system which allows to derive programs for proofs of their specifications.

The aim is now also to realize a Proof Checker used for teaching purposes in mathematical logic or even in “usual” mathematics.

The requirements for this new *proof assistant* are (it will be very difficult to reach all of them):

- Most of the “usual” mathematics should be feasible in this system. Actually PhoX is basically higher order classical logic, a more expressive (but not stronger) extension of the theory of simple types due to Ramsey [6][1]. Feasability is probably much more a probleme of “ergonomics” than a probleme of logical strength.

  Anyway it is always possible to represent any first order theory, you can add axioms and first order axiom’s schematas are replaced by second order axioms. You can represent this way set theory ZF in PhoX[2].

- The manipulation of the system should be as intuitive as possible. Thus, we shall try to have a simple syntax and a comprehensive way to build proofs within our system. All of this should be accessible for any mathematician with a minimal learning.

[1]which itself derives from the type system of Russell and Whitehead

[2]For now PhoX does not give the user any mechanical way to control that you use only first order instances of these schematas

- For programs extraction, we already know that PhoX provide enough functions (all functions provably total in higher order arithmetics) but we also need an efficient way to extract programs which should guaranty the fidelity to the specified algorithm and a good efficiency. The system will be credible only after bootstrapping which is the final (and long term) goal of this implementation !

1.2 Actual state.

Like some other systems, the user communicates with PhoX by an interaction loop. The user sends a command to the system. The prover checks it, and sends a response, that can be used by the user to carry on. A sequence of commands can be saved in a file. Such a file can be reevaluated, or compiled. This format is the same for libraries or user's files.

The prover has basically two modes with two sets of commands : the top mode and the proof mode. In the top mode the user can load libraries, describe the theory etc. In the proof mode the user proves a given proposition.

A proof is described by a sequence of commands (called a proof script), always constructed in an interactive way. The proof is constructed top-down : from the goal to "evidences". In case the goal is not proved by the command, responses of the system gives subgoals that should be easier than the initial goal. The system gives names for generated hyptothesis or variables. These names make writing easier, but the proof script cannot be understood without the responses of the system.

The system implements essentially the construction of a natural deduction tree in higher order logic, but can be used without really knowing the formal system of natural deduction.

The originality of the system is that the commands can be enhanced by the user, just declaring that some proved formulas of a particular form have to be interpreted as new rules. That allow the system to use few commands. Each command uses more or less automatic reasoning, and a generic automatic command composes the more basics ones.

A module system allows reusing of theories with renaming, eliminating constants and axioms by replacing them with definitions and theorems.

The existing libraries are almost all very basic ones (integers, lists...), but some examples have been developped that are not completly trivial : infinite version of Ramsey theorem, an abstract version of completness of predicate calculus, proof of Zorn lemma ...

In the current version programs extraction is possible but turned off by default and does not work with all features, see section A.1.8. Extraction is possible for proofs using intuitionistic or classical logic. Programs extraction implements what is described in [1] for intuitionistic functionnal second order arithmetic, but extended to classical logic and $\lambda\mu$-calcul : see [4].

1.3 Other sources of documentation

- The web page of PhoX:

  `https://raffalli.eu/phox/`

- Try PhoX online:

  `https://raffalli.eu/public/phox/`

- The documentation of the library (file `doc/libdoc.pdf`). You can also look at the PhoX files in the `lib` directory

- An article relating a teaching experiment with PhoX [5]. This article gives a short presentation of PhoX giving one commented example and an appendix of the main commands. It is also a good introduction to PhoX.

  It is available from the Internet in french and english:

  `https://raffalli.eu/pdfs/arao-fr.pdf`

  `https://raffalli.eu/pdfs/arao-en.pdf`

- The folder `tutorial/french` : it contains tutorial. It is only in french. A folder `tutorial/english` contains partial translation. Each tutorial comes with two files: `xxx_quest.phx` and `xxx_cor.phx`. In the first one there are questions: "dots" that you need to replace by the proper sequence of commands. The second one contains valid answer to all the questions.

  There are three kinds of tutorials (see the "README" in `tutorial/french` for a more detailed description):

  – Tutorial intended to learn PhoX itself: `tautologie_quest.phx`, `intro_quest.phx` and `sort_quest.phx`.
  – Tutorial intended to learn standard mathematics: `ideal_quest.phx`, `commutation_quest.phx`, `topo_quest.phx`, `analyse_quest.phx` and `group_quest.phx`.
  – Tutorial intended to learn logic: `tautologie_quest.phx` and `minlog_quest.phx` (the latest tutorial is difficult).

- The folder `examples` of the distribution : they contain a lot of examples of proof development. Beware that a lot of these examples were develop for some older version of PhoX and could be improved using recent features.

Chapter 2

Installation

2.1 Opam installation.

Phox is available as an opam package. So typically on a linux debian installation, the following should be enough:

```
apt-get install opam

opam install phox
```

2.2 Installation from source

Phox is available from github:

```
https://github.com/craff/phox
```

If OCaml and dune are installed on your linux machine, the following should work:

```
git clone https://github.com/craff/phox.git
opam install dune js_of_ocaml num camlp5
      #or equivalent if you don't use opam
cd phox
make
make install
```

2.3 Web interface

We plan a "server version" of phox that should make it possible to use the web interface with the local installation of phox, which is around 10 times faster than phox compiled with `js_of_ocaml`.

2.4 Emacs interface.

It is possible to use PhoX directly in a "terminal". But this is far from the best you can do. You can use the PhoX emacs mode developped by C. Raffalli and P. Roziere using D. Aspinall's "Proof-General".

Proof-General is available from

```
https://proofgeneral.github.io
```

2.4.1 Getting things to work.

First you need to have Emacs installed. Then you need to get and install Proof-General version 3.3 or later. Remember where you installed Proof-General.

On debian, this can be done using

```
apt-get install emacs proof-general
```

Then you need to add the following line to the configuration file of Emacs[1]:

```
(setq load-path (cons "PATH-WHERE-PHOX-INSTALLED-EMACS-FILES" load-path))
(require 'phox-mode)
```

If later PhoX fails to start, you can also add a line

```
(set-variable 'phox-prog-name "PATH-TO-PHOX-EXECUTABLE -pg")
```

The `-pg` is essential when PhoX works with Proof-General.

2.4.2 Getting started.

To start PhoX, you only need to open with Emacs a file whose name ends by the extension `.phx`. Try it, you should see a screen similar to the figure 2.1.

When using the interface, you use two “buffers” (division of a window where XEmacs displays text). One buffer represents your PhoX file. The other contains the answer from the system.

You should remark that under XEmacs (not Emacs) some symbols are displayed with a nice mathematical syntax. Moreover, when the mouse pointer moves above such symbol, you can see there ASCII equivalent.

To use it, you simply type in the PhoX file and transmit command to the system using the navigation buttons. The command that have been transmitted are highlighted with a different background color and are locked (you can not edit them anymore).

You should have a Proof-General menu allowing to evaluate your file. The main navigation commands are:

Next-Step (Ctrl-c Ctrl-n sends the next command to the system.

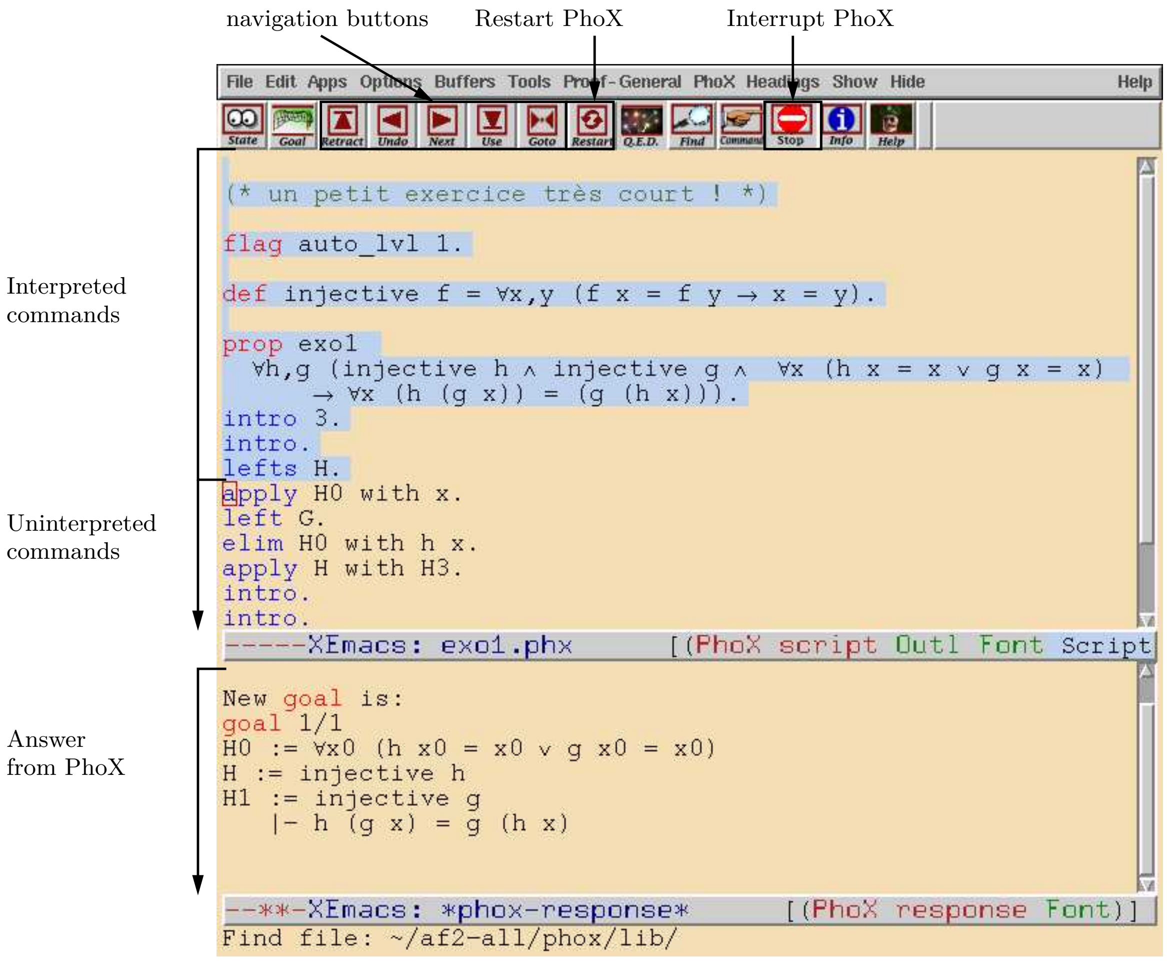
Undo-Step (Ctrl-c Ctrl-u go back from one command.

Goto-Point (Ctrl-c Ctrl-ret enter (or undo) all the commands to move to a specific position in the file.

Restart Scripting restart PhoX (sometimes very useful, because synchronisation between the PhoX system and XEmacs is lost. In this case you to Restart followed by Goto.

All these menus are also associated with a keyboard equivalent (visible in the menu).

[1]This configuration file is (under Unix/Linux) named `.emacs` pr `.emacs.d/init.el` and located in your home directory.
Your system administrator can also add lines in a general startup file to make PhoX available to all users.


navigation buttons
Restart PhoX
Interrupt PhoX
File Edit Apps Options Buffers Tools Proof-General PhoX Headings Show Hide Help
State Goal Retract Undo Next Use Goto Restart Q.E.D. Find Command Stop Info Help
Interpreted commands
Uninterpreted commands
Answer from PhoX

```
(* un petit exercice très court ! *)

flag auto_lvl 1.

def injective f = ∀x,y (f x = f y → x = y).

prop exo1
  ∀h,g (injective h ∧ injective g ∧  ∀x (h x = x ∨ g x = x)
       → ∀x (h (g x)) = (g (h x))).
intro 3.
intro.
lefts H.
apply H0 with x.
left G.
elim H0 with h x.
apply H with H3.
intro.
intro.
-----XEmacs: exo1.phx        [(PhoX script Outl Font Script

New goal is:
goal 1/1
H0 := ∀x0 (h x0 = x0 ∨ g x0 = x0)
H := injective h
H1 := injective g
    |- h (g x) = g (h x)

--**-XEmacs: *phox-response*        [(PhoX response Font)]
Find file: ~/af2-all/phox/lib/
```

Figure 2.1: Sample of a PhoX screen under XEmacs with Proof-General

2.4.3 Mathematical symbols.

Under Emacs, the input method names “TeX” should be activated by default under phox mode. Therefore, you can type any mathematical symbol used by phox by typing its latex equivalent. For instance:

- `\forall` for $\forall$
- `\to` for $\to$
- `\wedge` for $\wedge$
- `\times` for $\times$
- ...

2.4.4 Tips.

Proof-General can only work with one active file at a time. The best is to use the Restart button when switching from one file to another, because command like Import or Use can not be undone (so the Undo or Retract button will not give the expected result).

Sometimes, some information are missing in the answer window (this is very rare). You also may want to see the results of other commands than the last one. In this case, there is a buffer named `*phox*` available from the `Buffers` menu where you can see all the commands and answers since PhoX started.

In some very rare cases, the Restart button may not be sufficient (for instance if you changed your version of PhoX). You can use the menu `PhoX/Exit PhoX` to really stop the system and restart it.

Chapter 3

The primitive symbols of the logic.

3.1 Sorts.

In the /AFD/ formal system, we will manipulate *expressions*. But we will have to consider different kinds of expressions, for instance individual terms which are the objects of the logic (which means that we will prove their properties) and formulas which express properties of these objects. A *sort*[1] is a formal name for a kind of expressions.

There exists one basic sort of objects: `prop` (for formulas or types). New basic sort may be added, and sort may have parameters: for instance typing

```
Sort nat.
Sort list['a].
```

adds two sorts: one is called `nat` and the other is called `list['a]`. The second sort has one parameter allowing to distinguish `list[nat]` and `list[bool]`.

Complex sorts can be constructed using arrows: if τ and τ' are sorts, $\tau \to \tau'$ is also a sort. Intuitively, an expression of sort $\tau \to \tau'$ is an expression of sort τ' with a hole for an expression of sort τ. For instance the sort `nat` $\to$ `prop` is the sort of unary predicates on natural numbers.

Warning: although we use the same notation, the arrow between sorts is not related with the implication that we will introduce bellow.

3.2 The construction of expressions

The basic ways of constructing *expressions* (or simply-typed λ-terms) in a given *context* (a context declares the sorts of some *variables*) are the following:

Variable: A variable is just a name (see the chapter "Expressions, parsing and pretty printing" of the documentation of PhoX for the description of valid variable identifiers in PhoX). In a *context* where a variable x is of sort τ, x is an expression of sort τ

Abstraction: $\lambda x\, e$ is an expression of sort $\tau \to \tau'$ if e is an expression of sort τ' when the context is extended to declare x of sort τ. The variable x is *bound* in the expression

[0]written by: Christophe Raffalli (Paris VII & Paris XII university)

[1]If you know typed λ-calculus, sorts are *simple-types* constructed using atomic types like `prop` or `nat`

$\lambda x\, e$. This means that its name does not matter (renaming x gives an expression that we consider identical to the original one).

Application: $e\, e'$ (`e e'` in PhoX) is an expression of sort τ' if e is an expression of sort $\tau \to \tau'$ and if e' is an expression of sort τ.

It is important to remark that an expression like $\lambda x\, x$ may have many sorts: $\texttt{prop} \to \texttt{prop} or (\texttt{prop} \to \texttt{prop}) \to (\texttt{prop} \to \texttt{prop})$. But any sort has a most general sort using meta-variables written `'a`. One can obtain all the other sorts by replacing the meta-variables. For instance the most general sort of $\lambda x\, x$ is $\texttt{'a} \to \texttt{'a}$.

3.3 The primitive logical constants

Here are the only primitive logical constants:

Implication $\to$ (`$→` in PhoX) is an expression of sort $\texttt{prop} \to \texttt{prop} \to \texttt{prop}$. Thus it is a *binary connective*. We can use it with an Infix notation: if A and B are formulas (expressions of sort F), we can write $A \to B$ (`A → B` in PhoX) for the application of $\to$ to A and B. The meaning of this formula is: $A \to B$ is true if B is true whenever A is true.

Universal quantification $\forall$ (`$∀` in PhoX) is an expression which admits any sort of the form $(\tau \to \texttt{prop}) \to \texttt{prop}$.

As for implication, $\forall$ can be used with a special syntax: $\forall x\, A$ denotes $\forall \lambda x\, A$ (`∀x A` denotes `$∀ λx A` in PhoX). $\forall x\, A$ is true if A is true when x takes any value of sort τ if x is used with the sort τ (which means that $\forall$ is used with the sort $(\tau \to \texttt{prop}) \to \texttt{prop}$).

Equality $=$ (`$=` in PhoX) is an expression of sort $\tau \to \tau \to \texttt{prop}$ for any sort τ (it is a predicate constant of arity 2). As for implication, it can be used with an infix syntax. This equality is equivalent[2] to Leibniz equality: two terms are equal if they can not be distinguished by a predicate: $x = y$ is equivalent to $\forall X\, (X\, x \to X\, y)$.

Notation: We give some *priorities* to the different symbols to be able to omit some parenthesis. We will come back bellow on this point. Moreover, we allow the following abbreviations:

- $\forall x,y,z\, A$ for $\forall x \forall y \forall z A$ (`∀x ∀y ∀z A` in PhoX).
- $\forall x{:}P\, A$ for $\forall x(Px \to A)$ (`∀x:P  A` in PhoX).
- $\forall x,y{:}P\, A$ for $\forall x(Px \to \forall y(Py \to A)$ (`∀x,y:P  A` in PhoX).
- $\forall x{=}y\, A$ for $\forall x(x = y \to A)$ (`∀x = y  A` in PhoX). This abbreviation can be used with any infix predicate symbol.
- $A \to B \to C$ for $A \to (B \to C)$.

When producing LaTeX articles with PhoX one can decide not to use any of these abbreviations by setting some flags (see appendix B). Moreover, one can also decide to use standard mathematical notations for application: $f(x, y)$ instead of $f\, x\, y$. We decided to use here the nearest notation to the PhoX syntax.

[2]The equality is added as a primitive symbol to minimize the size of proofs using equality reasonning.

3.4 Other definitions

To write mathematical formula, you use other connective that just universal quantification ($\forall$) and implication ($\to$). Oher symbols are defined in the library `prop.phx` which is always loaded when you start PhoX. This library and others are described in the “User’s manual of the PhoX library”.

Chapter 4

Examples

4.1 How to read the examples.

We write examples using standard mathematical notation, as it will appear on the screen. To type the mathematical symbols, you need to type their LaTeX equivalent under emacs, terminated by a trailing backslash (
) on the web interface, sometimes with a shortcut.

	Symbol	type in Emacs	type in browser
Universal quantification	$\forall$	`\forall`	`\forall\`
Existential quantification	$\exists$	`\exits`	`\exits\`
Conjunction	$\wedge$	`\wedge`	`\&\` or `\and\`
Disjunction	$\vee$	`\wee`	`\or\`
Less or equal	$\leq$	`\leq`	`\<=\` or `\leq\`
Greater or equal	$\geq$	`\qeg`	`\>=\` or `\qeg`
Different	$\neq$	`\neq`	`\!=\` or `\neq\`

What you have to type to enter a formula, is exactly what is obtained when you replace each mathematical symbol by its ASCII equivalent.

We assume you read the previous section ! Moreover, you should report to the appendix A to get a detailed desciption of each command.

4.2 An example in analysis

The example given below is a typical small standalone proof (using no library).

We prove that two definitions of the continuity of a function are equivalent. We give only one of the directions, the other is similar. We have written it in a rather elaborate way in order to show the possibilities of the system.

- We define the sort of reals.
  ```
  >PhoX> Sort real.
  ```
  `Sort` is the name of the command used to create new sorts, but you can also use it to give name to existing sorts.

- We give a symbol for the distance and the real 0 (denoted by $R0$) as well as predicates for inequalities.
```
>PhoX> Cst d : real -> real -> real.
>PhoX> Cst R0 : real.
>PhoX> Cst Infix[5] x "≤" y : real -> real -> prop.
>PhoX> Cst Infix[5] x "<" y : real -> real -> prop.
>PhoX> def Infix[5] x ">" y = y < x.
>PhoX> def Infix[5] x "≥" y = y <= x.
```
  The command `Cst` introduces new constants of given sorts while `def` is used to give definitions. The commands to define inequalities are quite complex, because we want to use some infix notation with a specific priority.

- Here are the two definitions of the continuity:
  `>PhoX> def continue1 f x =`
  $\forall e{>}R0\,\exists a{>}R0\,\forall y(\mathrm{d}\,x\,y < a \to \mathrm{d}(fx)(fy) < e)$.
  `>PhoX> def continue2 f x =`
  $\forall e{>}R0\,\exists a{>}R0\,\forall y(\mathrm{d}\,x\,y \leq a \to \mathrm{d}(fx)(fy) \leq e)$.

- and the lemmas needed for the proof.
  `>PhoX> claim lemme1` $\forall x, y(x < y \to x \leq y)$.
  `>PhoX> claim lemme2` $\forall x{>}R0\,\exists y{>}R0\forall z(z \leq y \to z < x)$.

  The command `claim` allows to introduce new axioms (or lemmas that you do not want to prove now. You can prove them later using the command `prove_claim`). Beware that there may be a contradiction in your axioms!

- We begin the proof using the command `goal`:
  `>PhoX> goal` $\forall x, f(\mathrm{continue1}\, f\, x \to \mathrm{continue2}\, f\, x)$.
  `goal 1/1`
  `|-` $\forall x, f(\mathrm{continue1}\, f\, x \to \mathrm{continue2} fx)$

- We start with some "introductions".
  `%PhoX% intro 5.`
  `goal 1/1`
  `H :=` $\mathrm{continue1}\, f\, x$
  `H0 :=` $e > R0$
  `|-` $\exists a{>}R0\,\forall y(\mathrm{d}\,x\,y \leq a \to \mathrm{d}(fx)(fy) \leq e)$

  An "introduction" for a given connective, is the natural way to establish the truth of that connective without using other fact or hypothesis. For instance, to prove $A \to B$, we assume A and prove B. Here, PhoX did five introductions:

  – one for $\forall x$ and one for $\forall f$,
  – one for the implication ($\mathrm{continue1}\, f\, x \to \mathrm{continue2} fx$),
  – one for the $\forall e$ inside the definition of continue2
  – and finally, one for the hypothesis $e > R0$.

  Therefore, PhoX created three new objects: x, f, e and two new hypothesis named `H0` and `H1`.

- We use the continuity of f with e, and we remove the hypotheses H and H0 which will not be used anymore.
  ```
  %PhoX% apply H with H0. rmh H H0.
  goal 1/1
  ```
  `G :=` $\exists a{>}R0\,\forall y(\mathrm{d}\,x\,y < a \to \mathrm{d}(fx)(fy) < e)$

  `|-` $\exists a{>}R0\,\forall y(\mathrm{d}\,x\,y \le a \to \mathrm{d}(fx)(fy) \le e)$

  The `apply` command is quite intuitive to use. But it is a complex command, performing unification (more precisely higher-order unification) to guess the value of some variables. Sometimes you do not get the result you expected and you need to add extra information in the proper order.

- We *de-structure* hypothesis G by indicating that we want to consider all the $\exists$ and all the conjunctions (You can also use `lefts G` twice with no more indication).
  ```
  %PhoX% lefts G $∃ $∧.
  goal 1/1
  ```
  `H :=` $a > R0$

  `H0 :=` $\forall y(\mathrm{d}\,x\,y < a \to \mathrm{d}(fx)(fy) < e)$

  `|-` $\exists a_0{>}R0\,\forall y(\mathrm{d}\,x\,y \le a_0 \to \mathrm{d}(fx)(fy) \le e)$

  The `left` and `lefts` are introductions for an hypothesis: that is the way to use an hypothesis in a "standalone" way (not using the conclusion you want to prove or other hypothesis).

  We need to write a "`$`" prefix, because $\exists$ and $\vee$ have a prefix syntax and need other informations. The "`$`" prefix tells PhoX that you just want this symbol and nothing more.

- We use the second lemma with H and we remove it.
  ```
  %PhoX% apply lemme2 with H. rmh H.
  goal 1/1
  ```
  `H0 :=` $\forall y(\mathrm{d}\,x\,y < a \to \mathrm{d}(fx)(fy) < e)$

  `G :=` $\exists y{>}R0\,\forall z{\le}y\ z < a$

  `|-` $\exists a_0{>}R0\,\forall y(\mathrm{d}\,x\,y \le a_0 \to \mathrm{d}(fx)(fy) \le e)$

- We de-structure again G and we rename the variable y created.
  ```
  %PhoX% lefts G $∃ $∧. rename y a'.
  goal 1/1
  ```
  `H0 :=` $\forall y(\mathrm{d}\,x\,y < a \to \mathrm{d}(fx)(fy) < e)$

  `H1 :=` $a' > R0$

  `H2 :=` $\forall z{\le}a'\ z < a$

  `|-` $\exists a_0{>}R0\,\forall y(\mathrm{d}\,x\,y \le a_0 \to \mathrm{d}(fx)(fy) \le e)$

- Now we know what is the a_0 we are looking for. We do the necessary introductions for $\forall$, $\exists$, conjunctions and implications (again, you could use `intros` several times with no more indication). Two goals are created, as well as an existential variable (denoted by `?1`) for which we have to find a value.

  ```
  %PhoX% intros $∀ $∃ $∧ $→.
  goal 1/2
  ```

H0 := $\forall y(\mathrm{d}\,x\,y < a \to \mathrm{d}(fx)(fy) < e)$
H1 := $a' > R0$
H2 := $\exists z{\leq}a'\ z < a$
|- $?1 > R0$
goal 2/2
H0 := $\forall y(\mathrm{d}\,x\,y < a \to \mathrm{d}(fx)(fy) < e)$
H1 := $a' > R0$
H2 := $\forall z{\leq}a'\ z < a$
H3 := $\mathrm{d}\,x\,y \leq ?1$
|- $\mathrm{d}(fx)(fy) \leq e$

- The first goal is solved with the hypothesis H1 indicating this way that `?1` is a'. The second is automatically solved by PhoX by using lemma1, and this finishes the proof.
  `%PhoX% axiom H1. auto +lemme1.`

Remark. Instead of the command `auto +lemme1` one could also say `elim lemme1. elim H0. axiom H3.` or `apply H0 with H3. elim lemme1 with G.` where `G` is an hypothesis produced by the first command. We could also give the value of the existential variable by typing `instance ?1 a'`.
A good exercise for the reader consists in understanding what these commands do. The appendix A should help you !

Chapter 5

Expressions, parsing and pretty printing.

This chapter describes the syntax of PhoX. It is possible to use PhoX without a precise knowledge of the syntax, but for the best use, it is better to read this chapter ... But as any formal definition of a complex syntax, this is hard to read. Therefore, if it is the first time you read BNF-like syntactic rules, you will have problem to understand this chapter.

The layout of this chapter is inspired by the documentation of Caml-light (by Xavier Leroy).

5.1 Notations.

We will use BNF-like notation (the standard notation for grammar) with the following convention:

- Typewriter font for terminal symbols (`like this`). Sequences of terminal symbols are the only thing PhoX reads (by definition). We use range of characters to simplify when needed (like `0...9` for `0123456789`).

- Italic for non-terminal symbols (*like that*). Non-terminal symbols are meta-variables describing a set of sequences of terminal symbols. All non-terminal symbols we use are defined in this section (a := denotes such a definition)

- Square brackets [...] denotes optional components, curly brackets {...} denotes the repetition zero, one or more times of a component, curly brackets with a plus $\{...\}_+$ denotes repetition one or more times of a component and vertical bar denotes ... | ... alternate choices. Parentheses are used for grouping.

- Warning: sometimes, the syntax uses terminal symbol, like square brackets, which we use also with a scpecial meaning to describe the grammar. It is not easy to distinguish for instance the typewriter square brackets (`[]`) and the normal version ([]). When needed, we will clarify this by a remark.

5.2 Lexical analysis.

Blanks

The following characters are blank: space, newline, horizontal tabulation, line feed and form feed. These blanks are ignored, but they will separate adjacent tokens (like identifier, numbers, etc, described bellow) that could be confused as one single token.

Comments

Comments are started by `(*` and ended by `*)`. Nested comments are handled properly. All comments are ignored (except in some special case used for TeX generation, see the chapter 9) but like blank they separate adjacent tokens.

String, numbers, ...

Strings and characters can use the following escape sequences :

Sequence	Character denoted
`\n`	newline (LF)
`\r`	return (CR)
`\t`	tabulation (TAB)
`\`*ddd*	The character of code *ddd* in decimal
`\`*c*	The character *c* when *c* is not in `0...9nbt`

string-character	:=	any character but `"` or an escape sequence.
string	:=	`"` {*string-character*} `"`
char-character	:=	any character but `'` or an escape sequence.
char	:=	`'` *char-character* `'`
natural	:=	$\{$ `0...9` $\}_+$
integer	:=	[`-`] *natural*
float	:=	*integer* [`.` *natural*] [(`e` | `E`) *integer*]

Identifiers

Identifiers are used to give names to mathematical objects. The definition is more complex than for most programming languages. This is because we want to have the maximum freedom to get readable files. So for instance the following are valid identifiers: `A_1'`, `<=`, `<_A`. Moreover, in relation with the module system, identifiers can be prefixed with extension like in `add.assoc` or `prod.assoc`.

letter	:=	A...Z | a...z
end-ident	:=	{*letter* | 0...9 | _ } { ' }
atom-alpha-ident	:=	*letter end-ident*
alpha-ident	:=	*atom-alpha-ident* { . *atom-alpha-ident*}
special-char	:=	! | % | & | * | + | , | - | / | : | ; | < | = | > |
		@ | [|] | \ | # | ^ | ` | \ | | | { | } | ~ |
		Most unicode math symbols
atom-special-ident	:=	{*special-char*}$_+$ [_ *end-ident*]
special-ident	:=	*atom-special-ident* { . *atom-alpha-ident* }
any-ident	:=	*alpha-ident* | *special-ident*
pattern	:=	*any-ident* | (_ {. *atom-alpha-ident* })
unif-var	:=	? {*natural*}
sort-var	:=	' {*letter*}$_+$

Exemples:

- N, add.commutative.N, x0, x0', x_1 are *alpha-idents*.
- <, <<, <_1, +, +_N are *special-idents*.
- ?1 is a *unif-var*.
- 'a is a *sort-var*.
- +, _.N are *patterns* (used only for renaming symbol with the module system).

Special characters

The following characters are token by themselves:

(|) | . | $

5.3 Sorts

sorts-list	:=	*sort*
	|	*sort* , *sorts-list*
sort	:=	*sort-var*
	|	*sort* -> *sort*
	|	(*sort*)
	|	*alpha-ident*
	|	*alpha-ident* [*sorts-list*]

Examples: prop -> prop, ('a -> 'b) -> list['a] -> list['b] are valid *sorts*.

5.4 Syntax

The parsing and pretty printing of expressions are incremental. Thus we will now show the syntax the user can use to specify the syntax of new PhoX symbols.

ass-ident	:=	*alpha-ident* [`::` *sort*]
syntax-arg	:=	*string* | *ass-ident* | (\ *alpha-ident* \)
syntax	:=	*alpha-ident* {*ass-ident*}
	|	`Prefix` [`[` *float* `]`] *string* {*syntax-arg*}
	|	`Infix` [`[`*float*`]`] *ass-ident string* {*syntax-arg*}
	|	`rInfix` [`[`*float*`]`] *ass-ident string* {*syntax-arg*}
	|	`lInfix` [`[`*float*`]`] *ass-ident string* {*syntax-arg*}
	|	`Postfix`| [`[`*float*`]`] *ass-ident*

Moreover, in the rule for *syntax* a *ass-ident* can not be immediately followed by another *ass-ident* or a (\ *alpha-ident* \) because this would lead to ambiguities. Moreover, in the same rule, the *string* must contain a valid identifier (*alpha-ident* or *special-ident*). These constraints are not for LaTeX syntax.

5.5 Expressions

Expressions are not parsed with a context free grammar ! So we will give partial BNF rules and explain "infix" and "prefix" expressions by hand.

Here are the BNF rules with *infix-expr* and *prefix-expr* left undefined.

sort-assignment	:=	`:<` *sort*
alpha-idents-list	:=	*alpha-ident*
	|	*alpha-ident* `,` *alpha-idents-list*
atom-expr	:=	*alpha-ident*
	|	`$` *any-ident*
	|	*unif-var*
	|	`\` *alpha-idents-list sort-assignment atom-expr*
	|	`(` *expr* `)`
	|	*prefix-expr*
app-expr	:=	*atom-expr*
	|	*atom-expr app-expr*
expr	:=	*app-expr*
	|	*prefix-expr*
	|	*infix-expr*

This definition is clear except for two points:

- The juxtaposition of expression if the definition of *app-expr* means function application !
- The keyword `\` introduces abstraction: `\x x` for instance, is the identity function. `\x (x x)` is a strange function taking one argument and applying it to itself. In fact this second expression is syntaxically valid, but it will be rejected by PhoX because it does not admit a sort.

To explain how *infix-expr* and *prefix-expr* works, we first give the following definition:

A syntax definition is a list of items and a priority. The priority is a floating point number between 0 and 10. Each item in the list is either:

- An *alpha-ident*. These items are name for sub-expressions.
- A string containing an *any-ident*, using escape sequences if necessary. These kind of items are keywords.
- A token of the form \ *alpha-ident* \ where the *alpha-ident* is used somewhere else in the list as a sub-expression. These items are "binders".
- The list should obey the following restrictions (except for LaTeX syntax definition):
  - The first of the second item in the list should be a keyword. If the first item is a keyword, then the syntax definition is "prefix" otherwise it is "infix".
  - A name for a sub-expression can not be followed by another name for a sub-expression nor a binder.

Remark: this definition clearly follows the definition of a syntax.

Now we can explain how a syntax definition is parsed using the following principles. It is not very easy to understand, so we will give some examples:

1. The first keyword in the definition is the "name" of the object described by this syntax. This name can be used directly with "normal" syntax prefixed by a `$` sign.

   For instance, if the first keyword is the string `"+"`, then `+` is the name of the object and if this object is defined, `$+` is a valid expression.

2. To define the way *infix-expr* and *prefix-expr* are parsed, we will explain how they are parsed and give the same expression without using this special syntax.

3. The number of sub-expressions in the list is the "arity" of the object defined by the syntax.

4. To parse a syntax defined by a list, PhoX examines each item in the list:

   - If it is the i^{th} sub-expression in the list, then PhoX parses an expression and this expression is the i^{th} argument a_i of the object. At the end, if no binder is used, parsing an object whose name is `N` will be equivalent to parsing `$N` a_1 ...a_n.
   - If it is a keyword, then PhoX parses exactly that keyword.
   - If it is a binder `\x\`, where `x` is the name of the i^{th} sub-expression, then the variable `x` may appear in the i^{th}, and this i^{th} will be prefixed with `\x`. At the end, parsing an object whose name is `N` will be equivalent to parsing `N` $(\backslash x_1, \dots, x_n\ a_1)$ ...$(\backslash y_1, \dots, y_p\ a_n)$.
   - If the first and last item in the syntax definition are sub-expressions, the priority are important: PhoX parses expression at a given priority level, initially 10. If the priority of the syntax definition is strictly greater than the priority level, then this syntax definition can not be parsed.

     When parsing the first item, if it is a sub-expression, the priority level is changed to the priority level of the syntax definition (minus $\epsilon = 1e^{-10}$ if the symbol is not left associative). Left associative symbols are defined using the keyword lInfix of Postfix.

When parsing the last item, if it is a sub-expression, the priority level is changed to the priority level of the syntax definition (minus $\epsilon = 1e^{-10}$ if the symbol is not right associative. Right associative symbols are defined using the keyword rInfix of Prefix.

When parsing other items, the priority is set to 10.

Examples:

- The syntax `lInfix[3] x "+" y` is parsed by parsing a first expression a_1 at priority 3, then parsing the keyword `+` and finally, parsing a second expression a_2 at priority $3 - \epsilon$.

  Therefore, parsing a_1 `+` a_2 is equivalent to `$+` $a_1 a_2$ and parsing a_1 `+` a_2 `+` a_3 is equivalent to `$+ ($+` $a_1 a_2$`)` a_3.

- The syntax `Prefix "{" \P\ "in" y "|" P "}"` is parsed by parsing the keyword `{`, an identifier x, the keyword "in", a fist expression a_1, the keyword `|`, a second expression a_2 that can use the variable x and the keyword `}`.

  Therefore, parsing `{` x `in` a_1 `|` a_2 `}` is equivalent to `${` a_1 `\x` a_2.

- Other examples can be found in the appendix A in the description of the commands `Cst` and `def`

Remark: there are some undocumented black magic in PhoXparser. For instance, to parse $\forall x, y : N \ldots$ (meaning $\forall x(Nx \rightarrow \forall y(Ny \rightarrow \ldots))$ or $\forall x, y < z \ldots$ (meaning $\forall x(x < z \rightarrow \forall y(y < z \rightarrow \ldots))$, there is an obscure extension for binders.

This is really specialized code for universal and existential quantifications ... but advanturous user, looking at the definition of the existential quantifier `\/` in the library file `prop.phx` can try to understand it (though, I think it is not possible).

5.6 Commands

An extensive list of commands can be found in the index A using the same syntax and conventions.

Chapter 6

Natural Commands

PhoX's natural commands are conceived as an intermediate language for a forthcoming natural language interface. But, they are also directly usable with the following advantages and disadvantages compared with the usual tactics:

advantages Proof are readable and more robust (when you modify something in your theorems, less work is necessary to adapt your proofs).

disadvantages The automatic reasoning of PhoX is pushed to the limit and in the current implementation it may be hard to do complex proofs with natural commands. You can greatly help the system by using the `rmh` or `slh` commands to select the hypotheses.

Remark: some of the feature described here are signaled as not yet implemented.

6.1 Examples

Here are two examples:

```
def injective f = /\x,y (f x = f y -> x = y).

prop exo1
  /\h,g (injective h & injective g &  /\x (h x = x or g x = x)
      -> /\x (h (g x)) = (g (h x))).

let h, g assume injective h [H] and injective g [G]
              and /\x (h x = x or g x = x) [C]
  let x show h (g x) = g (h x).
by C with x assume h x = x then assume g x = x.
(* cas h x = x *)
  by C with g x assume h (g x) = g x trivial
           then assume g (g x) = g x [Eq].
  by G with Eq deduce g x = x trivial.
(* cas g x = x *)
  by C with h x assume g (h x) = h x trivial
           then assume h (h x) = h x [Eq].
  by H with Eq deduce h x = x trivial.
save.


def inverse f A = \x (A (f x)).

def ouvert O = /\ x (O x -> \/a > R0 /\y (d x y < a -> O y)).

def continue1 f = /\ x  /\e > R0 \/a > R0
  /\ y (d x y < a -> d (f x) (f y) < e).

def continue2 f = /\ U ((ouvert U) -> (ouvert (inverse f U))).

goal /\f (continue1 f -> continue2 f).
let f assume continue1 f [F]
  let U assume ouvert U [O] show ouvert (inverse f U).
let x assume U (f x) [I] show \/b > R0  /\x' (d x x' < b -> U (f x')).
by O with f x let a assume a > R0 [i] and /\y (d (f x) y < a -> U y) [ii].
by F with x and i let b assume b > R0 [iii] and /\ x' (d x x' < b -> d (f x) (f x') < a) [
let x' assume d x x' < b [v] show U (f x').
by ii with f x' show d (f x) (f x') < a.
by iv with v trivial.
save th1.
```

6.2 The syntax of the command

The command follow the following grammar:

cmd	:=	`let` *idlist cmd* |	
		`assume` *expr naming* {`and` *expr naming*} *cmd* |	
		`deduce` *expr naming* {`and` *expr naming*} *cmd* |	
		`by` *alpha-ident* {`with` *with-args*} *cmd* |	
		`show` *expr cmd* |	
		`trivial` |	
		∅ |	not allowed after `by`
		cmd `then` *cmd* |	
		`begin` *cmd* `end` |	
idlist	:=	*alpha-ident* {, *alpha-ident* }{: *expr* | *infix-symbol expr*} |	
		alpha-ident = *expr* |	not implemented
		idlist `and` *idlist*	
naming	:=	`named` *alpha-ident* | [*alpha-ident*]	
with-args	:=	see the documentation of the `elim` and `apply` commands in the appendix	

Note: In the current implementation, only `trivial` is allowed after `show`. Naming using square brackets wont work if the opening square bracket is defined as a prefix symbol.

6.3 Semantics

Definition 6.1 A natural command is simple if `show` is followed by the empty command.

A simple command in a goal is interpreted as a rule that needs to be proved derivable automatically by PhoX. A natural command can be seen as a tree of simple command and is therefore interpreted as a tree of derivable rule, that is a derivable rule itself.

We will just describe the interpretation of a simple command: let us assume the current goal is $\Gamma \vdash A$ then a simple command is interpreted as a rule whose conclusion is $\Gamma \vdash A$ and whose premises are defined by induction on the structure of the command. Thus, we only need to prove the premises to prove the current goal.

First some syntactic sugar can be elliminated:

- `let` I `and` I' is interpreted as `let` I `let` I'
- `let` $x_1, \dots, x_n \star P$ (where $\star$ is : or an infix symbol is interpreted as `let` $x_1 \star P \dots$ `let` $x_n \star P$
- `let` $x : P$ is interpreted as `let` x `assume` Px
- `let` $x \star P$ is interpreted as `let` x `assume` $x \star P$ where $\star$ is an infix symbol.
- The keyword `deduce` is interpreted as `assume`.
- `assume` A_1 `and` ... `and` A_n is interpreted as `assume` A_1 `assume` ... `assume` A_n

Then the set of premises $|C|$ associated to the simple command C if the current goal is $\Gamma \vdash A$ is defined by

|`let` x C|	=	$\|C\|$ the variable x may be used in $\|C\|$
|`assume` E `named` H C|	=	$\{H := E, \Gamma_1 \vdash B_1, \dots, H := E, \Gamma_n \vdash B_n\}$ if $\|C\| = \{\Gamma_1 \vdash B_1, \dots, \Gamma_n \vdash B_n\}$ if H is not given, it is chosen by PhoX
`by` H `with` $\dots C$	=	$\|C\|$ the indication in by are used as hints by the automated when using H.
|`show` E|	=	$\{\Gamma \vdash E\}$
$\|\emptyset\|$	=	$\{\Gamma \vdash A\}$
|C`then` C'|	=	$\|C\| \cup \|C'\|$
|`begin` C`end` |	=	$\|C\|$

Chapter 7

The module system

This chapter describes the PhoX module system. Its purpose is to allow reusing of theory. For instance you can define the notion of groups and prove some of their properties. Then, you can define fields and reuse your group module twice (for multiplication and addition).

7.1 Basic principles

Our module system strongly uses the notion of names. Any objects (theorems, terms, ...) has a distinct name. Therefore, if you want to merge two PhoX modules which both declare an object with the same name, this two objects must coincide after merging.

Here are the conditions under which two objects can coincide:

- They must have the same sorts. A formula can not coincide with a natural number.
- If both objects are defined expressions, their definitions must be structurally equal.
- If both objects are theorems or axioms, they must have structurally equal conclusions. They do not need to have the same proof.

If one of this condition is not respected, the loading of modules will fail.

These rules allow you to make coincide an axiom with a theorem and a constant with a definition. This is why we can prove axiomatic properties of a structure like groups by adding some constants and axioms and then use this module on a particular group where the axioms may be proven and the constants may already exists.

7.2 Compiling and importing

When you have written a PhoX file `foo.phx`, you can compile it using the command:

```
phox -c foo.phx
```

This compilation generates two files `foo.phi` and `foo.pho` and possibly one or more LaTeX file (see the chapter 9).

The file `foo.pho` is a core image of PhoX just after the compilation. You can use it to restart PhoX in a state equivalent to the state it had after reading the last line of the file `foo.phx`. This is useful when developing to avoid executing each line in the file before continuing it.

The file `foo.phi` is used when you want to reuse the theory developed in the file `foo.phx`. To do so you can use the command:

```
Import foo.
```

This command includes all the objects declared in the file `foo.phx`. The above rules are used to resolve name conflicts.

7.3 Renaming and using

The command `Import` is not sufficient. Indeed, if one wants to use twice the same module, it is necessary to rename the different objects it can contains to have distinct copies of them.

To do this, you can use the command `Use` (see the index of commands for its complete syntax and the definition of renaming). The different possibilities of renaming, and a careful choice of names allow you to transform easily the names declared in the module you want to use.

When you use a module, you sometimes know that you are not extending the theory. For instance, if you prove that a structure satisfies all the group axioms, you can load the group module to use all the theorems about groups and you are not extending the theory. The `-n` option of the `Use` command checks that it is the case and an error will result if you extend the theory.

Important note: there is an important difference between `Use` and `Import` other than the possibility of renaming with `Use`. When you apply a renaming to a module `foo` this renaming does not apply to the module imported by `foo` (with `Import`) but it applies to the module used by `foo` (with `Use`). This allows you to import modules like natural numbers when developing other theories with a default behaviour which is not to rename objects from the natural numbers theory when your module is used. You can override this default behaviour (see the index of commands), but it is very seldom useful.

7.4 Exported or not exported?

By default, anything from a PhoX file is exported and therefore available to any file importing or using it (except the flags values!). However, you can make some theorems of rules local using the `Local` prefix (see the index of commands).

However, constants and axioms are always exported, and a definition appearing in an exported definition or theorem is always exported. So it is only useful to declare local some rules (created using `new_intro`, `new_elim` or `new_equation`), LaTeX syntaxes (created using `tex_syntax`), lemmas or definitions only appearing in local lemmas.

7.5 Multiple modules in one file

Warning: this mode can not be used with XEmacs interface and in general in interactive mode ! To use it, develop the last module in interactive mode as one file and add it at the end of the main file when it works.

This feature will probably disappear soon ....

It is sometimes necessary to develop many small modules. It is possible in this case to group the definitions in the same file using the following syntax:

```
Module name1.
  ...
  ...
end.

Module name2.
  ...
  ...
end.

...
```

If the file containing these modules is named `foo` this is equivalent to having many files `foo.name1.phx`, `foo.name2.phx`, …containing the definitions in each module. Therefore the name of each module (to be used with `Import` or `Use`) will be `foo.name1`, `foo.name2`, ….

Moreover, a module can use the previously defined modules in the same file using only the name of the module (omitting the file name).

Here is an example where we define semi-groups and homomorphisms.

```
Module semigroup.
  Sort g.
  Cst G : g -> prop.
  Cst rInfix[3] x "op" y : g -> g -> g.

  claim op_total /\x,y:G  G (x op y).
  new_intro -t total op_total.

  claim assoc /\x,y,z:G  x op (y op z) = (x op y) op z.
  new_equation -b assoc.

end.

Module homomorphism.

  Use semigroup with
    _ -> _.D
  .

  Use semigroup with
    _ -> _.I
  .

  Cst f : g -> g.
  claim totality.f /\x:G.D  G.I (f x).
  new_intro -t f totality.f.

  claim compatibility.f /\x1,x2:G.D  f (x1 op.D x2) = f x1 op.I f x2.
  new_equation compatibility.f.

end.
```

Chapter 8

Inductive predicates and data-types.

This chapter describes how you can construct predicate and data-types inductively. This correspond traditionnally to the definition of a set as the smallest set such that ...

This kind of definitions are not to difficult to write by hand, but they are not very readable and moreover, you need to prove many lemmas before using them. PhoXwill generate and prove automatically these lemmas (most of the time)

8.1 Inductive predicates.

We will first start with some examples:

```
Use nat.

Inductive Less x y =
  zero : /\x Less N0 x
| succ : /\x,y (Less x y -> Less (S x) (S y))
.

Inductive Less2 x y =
  zero : Less2 x x
| succ : /\y (Less2 x y -> Less2 x (S y))
.
```

This example shows two possible definitions for the predicate less or equal on natural numbers.

The name of the predicates will be `Less` and `Less2` and they take both two arguments. They are the smallest predicates verifying the given properties. The identifier `zero` and `succ` are just given to generate good names for the produced lemmas.

These lemmas, generated and proved by PhoX, are:

```
zero.Less = /\x Less N0 x : theorem
succ.Less = /\x,y (Less x y -> Less (S x) (S y)) : theorem
```

Which are both added as introduction rules for that predicate with `zero` and `succ` as abbreviation (this means you can type `intro zero` or `intro succ` to specify which rule to use when PhoXguesses wrong).

```
rec.Less =
  /\X/\x,y
    (/\x0 X N0 x0 ->
     /\x0,y0 (Less x0 y0 -> X x0 y0 -> X (S x0) (S y0)) ->
     Less x y -> X x y) : theorem

case.Less =
  /\X/\x,y
   ((x = N0 -> X N0 y) ->
    /\x0,y0 (Less x0 y0 -> x = S x0 -> y = S y0 -> X (S x0) (S y0)) ->
    Less x y -> X x y) : theorem
```

The first one: `rec.less` is an induction principle (not very useful ?). It is added as an elimination rule. The second one is to reason by cases. It is added as an invertible left rule: If you want to prove `P x y` with an hypothesis `H := Less x y`, the command `left H` will ask you to prove `P N0 y` with the hypothesis `x = N0` (there may be other occurrences of `x` left) and `P (S x0) (S y0)` with three hypothesis: `Less x0 y0`, `x = S x0` and `y = S y0`.

The general syntax is:

`Inductive` *syntax* [`-ce`] [`-cc`] =
alpha-ident [`-ci`] : *expr*
{ `|` *alpha-ident* [`-ci`] : *expr* }

You will remark that you can give a special syntax to your predicate. The option `-ce` means to claim the elimination rule. The option `-cc` means to claim the case reasonning. The option `-ci` means to claim the introduction rule specific to that property.

8.2 Inductive data-types.

The definition of inductive data-types is similar. Let us start with an example:

```
 Data List A =
  nil : List A nil
| cons x l : A x -> List A l -> List A (cons x l)
.
```

This example will generate a sort `list` with one parameter. It will create two constants `nil : list['a]` and `cons : 'a -> list['a] -> list['a]`.

It will also claim the axiom that these constants are distinct and injective.

Then it will proceed in the same manner as the following inductive definition to define the predicate `List` and the corresponding lemmas:

```
 Inductive List A l =
  nil : List A nil
| cons : /\x,l (A x -> List A l -> List A (cons x l))
.
```

There is also a syntax more similar to ML:

```
type List A =
  nil  List A nil
| cons of A and List A
.
```

The general syntax is (`Data` can be replaced by `type`):

constr-def :=
  alpha-ident {*ass-ident*} |
  `[` *alpha-ident* `]` *syntax*
`Data` *syntax* [`-ce`] [`-cc`] [`-nd`] [`-ty`] =
  constr-def [`-ci`] [`-ni`] `:` *expr* |
  { `|` *constr-def* [`-ci`] [`-ni`] `:` *expr* }   { `|` *constr-def* [`-ci`] [`-ni`] `of` *expr* [`and` *expr* ...]

We can remark three new options: `-nd` to tell PhoX not to generate the axioms claiming that all the constructors are distinct, `-ty` to tell PhoX to generate typed axioms (for instance `/\x:N (N0 != S x)` instead of `/\x (N0 != S x)`) and `-ni` to tell PhoX not to generate the axiom claiming that a specific constructor is injective.

One can also remark that we can give a special syntax to the constructor, but one still need to give an alphanumeric identifier (between square bracket) to generate the name of the theorems.

Here is an example with a special syntax:

```
Data List A =
  nil : List A nil
| [cons] rInfix[3.0] x "::" l : A x -> List A l -> List A (x::l)
.
```

Chapter 9

Generation of LaTeX documents.

When compiling a PhoX file (using the `phox -c` command) you can generate one or more LaTeX documents. This generation is NOT automatic. But PhoX can produce a LaTeX version of any formula available in the current context. This means that when you want to present your proof informally, you can insert easily the current goal or hypothesis in your document. In practice you almost never need to write mathematical formulas in LaTeX yourself. When a formula does not fit on one line, you can tell PhoX to break it automatically for you (this will require two compilations in LaTeX and the use of a small external tool `pretty` to decide where to break).

You can also specify the LaTeX syntax of any PhoX constant or definition so that they look like you wish. In fact using all these possibilities, you can completely hide the fact that your paper comes from a machine assisted proof !

The LaTeX file produced by PhoX can be used as stand-alone articles or inserted in a bigger document (which can be partially written in pure LaTeX).

In this chapter, we assume that the reader as a basic knowledge of LaTeX.

9.1 The LaTeX header.

If you want PhoX to produce one or more LaTeX documents, you need to add a *LaTeX header* at the beginning of your file (only one header should be used in a file even in a multiple modules file). A LaTeX header look like this:

```
tex
  title = "A Short proof of Fermat's last Theorem"
  author = "Donald Duck"
  institute = "University of Dingo-city"
  documents = math slides.
```

The three first fields are self explanatory and the strings can contain any valid LaTeX text which can be used as argument of the `\title` of `\author` commands.

The last field is a list of documents that PhoX will produce. In this case, if this header appears in a file `fermat.phx`, the command `phox -c fermat.phx` will produce two files named `fermat.math.tex` and `fermat.slides.tex`.

The document names `math` and `slides` will be used later in LaTeX comments.

Warning: do not forget the dot at the end of the header.

9.2 LaTeX comments.

A LaTeX comment is started by `(*! doc1 doc2 ...` (on the same line) and ended by `*)`. As far as building the proof is concerned, these comments are ignored. `doc1`, `doc2`, ... must be among the document names declared in the header. Thus, when compiling a PhoX file, the content of these comments are directly outputed to the corresponding LaTeX files (except for the formulas as we will see in the next section).

9.3 Producing formulas

To output a formula (which fits on one line), you use `\[ ... \]` or `\{ ... \}`. The first form will print the formula in a *mathematical version* (like $\forall X(X \to X)$). The second will produce a verbatim version, using the PhoX syntax (like `/\X (X -> X)`). The second form is useful when producing a documentation for a PhoX library, when you have to teach your reader the PhoX syntax you use.

Formulas produced by `\[ ... \]` may be broken by TeX using its usual breaking scheme. Formula produced by `\{ ... \}` will never be broken (because LaTeX do not insert break inside a box produced by `\verb`). We will see later how to produce larger formulas.

LaTeX formulas can use extra goodies:

- They can contain free variables.
- If `A` is a defined symbol in PhoX , `$$A` will refer to the definition of `A` (If this definition is applied to arguments, the result will be normalised before printing). Remember that a single dollar must be used when A as a special syntax and you want just to refer to A (For instance you use `$+` to refer to the addition symbol when it is not applied to two arguments).
- All the hypothesis of the current goal are treated like any defined symbol.
- `$0` refers to the conclusion of the current goal.
- You can use the form `\[n` or `\{n` where `n` is an integer to access the conclusion and the hypothesis of the nth goal to prove (instead of the current goal).
- You can use the following flags (see the index B for a more detailed description) to control how formulas will look like: `binder_tex_space, comma_tex_space, min_tex_space, max_tex_space, tex_indent,`
`tex_lisp_app, tex_type_sugar, tex_margin, tex_max_indent`

A `\[ ... \]` or `\{ ... \}` can be used both in text mode and in math mode. If you are in text mode, `\[ ... \]` is equivalent to `$\[ ... \]$` (idem with curly braces).

WARNING: the closing of a formula: `\]`, `\}` should not be immediately followed by a character such that this closing plus this character is a valid identifier for PhoX. Good practice is always to follow it by a white space. This is a very common error!

9.4 Multi-line formulas

You can produce formulas fitting on more than one line using `\[[ ... \]]` or `\{{ ... \}}`.

The second form produces verbatim formulas similar to those produced by the PhoX pretty printer (with the same breaking scheme) like:

```
lesseq.rec2.N
 = /\X
     /\x,y:N
       (X x -> /\z:N  (x <= z -> z < y -> X z -> X (S z)) ->
          x <= y -> X y)
 : Theorem
```

The first form produces multi-line formulas using the same mathematical syntax than `\[ ... \]` like:

$$\begin{array}{l}\forall X\,\forall x,y{:}\mathbb{N}\\ \quad\left(\begin{array}{l}X\,x \to \forall z{:}\mathbb{N}\ (x \le z \to z < y \to X\,z \to X\,(\mathrm{S}\,z)) \to\\ x \le y \to X\,y\end{array}\right)\end{array}$$

However, breaking formulas in not an easy task. When you compile with LaTeX a file `test.tex` produced from an PhoX file using `\[[ ... \]]`, a file `test.pout` is produced. Then using the command `pretty test` (do not forget to remove the extension in the file name), a file `test.pin` is produced which tells LaTeX where to break lines. Then you can compile one more time your LaTeX file. It may be necessary to do all this one more time to be sure to reach a fix-point.

The formula produced in this way will use no more space than specified by the LaTeX variable `\textwidth`. Therefore, you can change this variable if you want formulas using a given width.

9.5 User defined LaTeX syntax.

You can specify yourself the syntax to be used in the math version of a formula. To do so you can use the `tex_syntax`. This command can have three form:

`tex_syntax` *`symbol`* `"`*`name`*`"` : tells PhoX to use this *name* for this *symbol. name* should be a valid LaTeX expression in text mode and will be included inside an `hbox` in math mode. This form should be used to give names to theorems, lemmas and functions which are to be printed just as a name (like sin or cos).

`tex_syntax` *`symbol`* `Math "`*`name`*`"` : tells PhoX to use this *name* for this *symbol. name* should be a valid LaTeX expression in math mode and will be included directly in math mode.

`tex_syntax` *`symbol syntax`* : tell PhoX to use the given *syntax* for this *symbol.* The *syntax* uses the same convention as for the command `def` of `cst`. When the *symbol* is used without its syntax (using `$symbol`) the first keyword if the syntax is `Prefix` or the second otherwise will be used. Moreover, you can separate tokens with the following spacing information (to change the default spacing):

`!` suppresses all space and disallow breaking (in multi-line formulas).

`<`*n*`>` (where *n* is an integer) uses *n* 100th of `em` for spacing and disallows breaking (in multi-line formulas).

`<`*n i*`>` (where *n* and *i* are integers) uses *n* 100th of `em` for spacing and allows breaking (in multi-line formulas) using *i* 100th of `em` of extra indentation space.

9.6 examples.

```
cst 2 rInfix[4] x "|->" y.
tex_syntax $|-> rInfix[4] x "\\hookrightarrow" y.
```

Will imply the `\[ A |-> B \]` gives $A \hookrightarrow B$ in your LaTeX document. You should note that you have to double the `\` in strings.

```
Cst Prefix[1.5] "Sum" E "for" \E\ "=" a "to" b
  : (Term -> Term) -> Term -> Term -> Term.
tex_syntax $Sum Prefix[1.5]
  "\\Sigma" "_{" ! \E\ "=" a ! "}^{" ! b ! "}" E %as $Sum E a b.
```

Will imply that `\[Sum f i for i = n to p\]` gives $\Sigma_{i=n}^{p} fi$ in your LaTeX document. We have separated the `"\\Sigma"` from the `"_{"` so that `"\[$Sum\]"` just produces a single Σ and we used `"%as"` to modify the order of the arguments (because `E` comes last in the LaTeX syntax and first in the PhoX syntax).

More complete examples can be found by looking at the libraries and examples distributed with PhoX.

Chapter 10

Installation.

You can read up-to-date instructions at the following url :

```
http://www.lama.univ-savoie.fr/~raffalli/phox.html
```

We will explain how to install PhoX on a Unix machine. If you are familiar with Objective-Caml, it should not be difficult to get it work on any machine which can run Objective-Caml.

To install the "PhoX Proof Checker", proceed as follow:

1. Get and install Objective-Caml version 3.0* (at least 3.08). You can get it by ftp:

   ```
   site = ftp.inria.fr
   dir = lang/caml-light
   file = ocaml-3.0*.tar.gz
   ```

2. Get the latest version of PhoX by ftp :

   ```
   site = www.lama.univ-savoie.fr
   or
   site = ftp.logique.jussieu.fr
   dir = pub/distrib/phox/current/
   file = phox-0.xxbx.tar.gz
   ```

3. Uncompress it and detar it (using `gunzip phox-0.xxbx.tar.gz; tar xvf phox-0.xxbx.tar`)

4. Move to the directory phox-0.xxbx which has just been created.

5. Edit the file "./config", to suit you need.

6. Type "make".

7. Type "make install"

8. If you want the program to look for its libraries in more than one directory, you can set the `PHOXPATH` variable, for instance like this (with csh):

   ```
   setenv PHOXPATH /usr/local/lib/phox/lib:$USERS/phox/examples
   ```

9. You are strongly encouraged to use the emacs interface to PhoX. To install an emacs-mode, use Proof-General (release 3.3 or greatest) from:

   ```
   http://www.proofgeneral.org/~proofgen
   ```

   Proof-General works better with xemacs, but pre-releases 3.4 works reasonably well with gnu-emacs 21.

Appendix A

Commands.

In this index we describe all the PhoX commands. The index is divided in two sections: the top-level commands (always accepted) and the proof commands (accepted only when doing a proof).

A.1 Top-level commands.

In what follows curly braces denote an optional argument. You should note type them.

A.1.1 Control commands.

`goal` *`formula`*. Start a proof of the given *`formula`*. See the next section about proof commands.

```
>phox> def fermat =
  /\x,y,z,n:N ((x^n + y^n = z^n) -> n <= S S 0).
/\ x,y,z,n : N (x ^ n + y ^ n = z ^ n -> n <= S S 0) : Form
>phox> goal fermat.
.....

.....
>phox> proved
```

`prove_claim` *`name`* :

Start the proof of an axiom previously introduced by then `claim` command. It is very useful with the module system to prove claims introduced by a module.

`quit.` :

Exit the program.

```
>phox> quit.
Bye
%
```

`restart.` :

Restart the program, does not stop it, process is stil the same.

`{Local} theorem` *`name`* `{`*`"tex_name"`*`}` *`expression`* Identical to goal except you give the name of the theorem and optionally its TeX syntax (this TeX Syntax is used as in `tex_syntax` *`name`* *`"tex_name"`*). Therefore, you do not have to give a name when you use the `save` command.

Instead of `theorem`, you can use the following names: `prop | proposition | lem | lemma| fact | cor | corollary | theo`.

You can give the instruction `Local` to indicate that this theorem should not be exported. This means that if you use the `Import` or `Use` command, only the exported theorem will be added.

A.1.2 Commands modifying the theory.

`claim` *`name`* `{`*`"tex_name"`*`}` *`formula`*`.` Add the *`formula`* to the data-base as a theorem (claim) under the given *`name`*.

You can give an optional TeX syntax (this TeX Syntax is used as in `tex_syntax` *`name`* *`"tex_name"`*).

`Cst` *`syntax`* `:` *`sort`*`.` Defines a constant of any *`sort`*.

```
>phox> Cst map : (nat -> nat) -> nat -> nat.
Constant added.
```

Default syntax is prefix. You can give a prefix, postfix or infix syntax for instance the following declarations allow the usual syntaxes for order $x < y$ and factorial $n!$:

```
Cst Infix x "<" y : nat -> nat -> prop.
$< : d -> d -> prop
Cst Postfix[1.5] x "!" : nat -> nat.
$! : d -> d
```

To avoid too many parenthesis, you can also give a *priority* (a floating number) and, in case of infix notation, you can precise if the symbol associates to the right (`rInfix`) or to the left (`lInfix`).

For instance the following declarations[1]

```
Cst Prefix[2] "S" x : nat -> nat.
Cst rInfix[3.5] x "+" y : nat -> nat -> nat.
Cst lInfix[3.5] x "-" y : nat -> nat -> nat.
Cst Infix[5] x "<" y : nat -> nat -> prop.
```

gives the following :

[1]these declarations are no more exactly the ones used in the PhoX library for integers.

- `S x + y` means `(S x) + y` (parenthesis around the expression with principal symbol of smaller weight) ;
- `x - y < x + y` means `(x - y) < (x + y)` (same reason) ;
- `x + y + z` means `x + (y + z)` (right symbol first) ;
- `x - y - z` means `(x - y) - z` (left symbol first).

More : the two symbols have the same priority and then `x - y + z` is not a valid expression.

Arbitrary priorities are possible but can give a mess. You have ad least to follows these conventions (used in the libraries) :

- connectives : priority > 5 ;
- predicates : priority $= 5$;
- functions : priority < 5.

You can even define more complex syntaxes, for instance :

```
Cst Infix[4.5]  x  "==" y "mod" p : nat -> nat -> nat-> nat.
(* $== : nat -> nat -> nat -> nat *)
print \a,b(a + b == a mod b).
(* \a,b (a + b == a mod b) : nat -> nat -> nat *)
```

you can define syntax for binders :

```
Cst Prefix[4.9] "{" \P\ "in" a "/" P "}"
:   'a -> ('a -> prop) -> prop.
(* ${ : 'a -> ('a -> prop) -> prop *)
print \a \P{ x in a / P}.
(* \a,P {x in a / P } : ?a -> prop -> prop *)
```

`{Local} def` *`syntax`* `=` *`expression`*. Defines an abbreviation using a given *`syntax`* for an *`expression`*.

The prefix `Local` tells that this definition should not be exported. This means that if you use the `Import` or `Use` command, only the exported definitions will be added.

Here are some examples :

```
>phox> def rInfix[7]  X "&" Y = /\K ((X -> Y -> K) -> K).
(\X (\Y /\ K ((X -> Y -> K) -> K))) : Form -> Form -> Form
>phox> def rInfix[8]  X "or" Y =
  /\K ((X -> K) -> (Y -> K) -> K).
(\X (\Y /\ K ((X -> K) -> (Y -> K) -> K))) :
  Form -> Form -> Form
>phox> def Infix [8.5]  X "<->" Y = (X -> Y) & (Y -> X).
(\X (\Y (X -> Y) & (Y -> X))) : Form -> Form -> Form
>phox> def Prefix[5] "mu" \A\ \A\ A "<" t ">" =
  /\X (/\x (A X x -> X x) -> X t).
(\A (\t /\ X (/\ x (A X x -> X x) -> X t))) :
  ((Term -> Form) -> Term -> Form) -> Term -> Form
```

Defintion of the syntax follows the same rules and conventions as for the command `Cst` above.

`{Local} def_thlist` *`name`* `=` *`th1 …thn`*. Defines *`name`* to be the list of theorems *`th1 …thn`*. For the moment list of theorems are useful only with commands `rewrite` and `rewrite_hyp`.

```
>phox> def_thlist demorgan =
  negation.demorgan  disjunction.demorgan
  forall.demorgan    arrow.demorgan
  exists.demorgan    conjunction.demorgan.
```

`del` *`symbol`*`.` Delete the given *symbol* from the data-base. All definitions, theorems and rules using this *symbol* are deleted too.

```
>phox> del lesseq1.
delete lesseq_refl
delete inf_total from ##totality_axioms
delete inf_total
delete sup_total from ##totality_axioms
delete sup_total
delete less_total from ##totality_axioms
delete less_total
delete lesseq_total from ##totality_axioms
delete lesseq_total
delete lesseq1 from ##rewrite_rules
delete lesseq1
```

`del_proof` *`name`*. Delete the proof of the given theorem (the theorem becomes a claim). Useful mainly to undo the `prove_claim` command.

`Sort {['`*`a`*`,'`*`b`*`, …]} {=` *`sort`*`}.` Adds a new sort. The sort may have parameters or may be defined from another sort.

```
>phox> Sort real.
Sort real defined
>phox> Sort tree['a].
Sort tree defined
>phox> Sort bool = prop.
Sort bool defined
```

A.1.3 Commands modifying proof commands.

These commands modify behaviour of the proof commands described in appendix A.2. For instance the commands `new_intro`, `new_elim` and `new_equation` by adding new rules, modify behaviour of the corresponding proof commands `intro`, `elim`, `rewrite` and commands that derive from its.

In particular they can also modify the behaviour of automatic commands like `trivial` and `auto`. They are useful to make proofs of further theorems easier (but can also make them harder if not well used). You can find examples in PhoX libraries, where they are systematically used.

For good understanding recall that the underlying proof system is basically natural deduction, even if it is possible to add rules like lefts rules of sequent calculus, see below.

`{Local} close_def` *`symbol`*. When *symbol* is defined, this "closes" the definition. This means that the definition can no more be open by usual proof commands unless you explicitly ask it by using for instance proof commands `unfold` or `unfold_hyp`. In particular unification does not use the definition anymore. This can in some case increase the efficiency of the unification algorithm and the automatic tactic (or decrease if not well used). When you have add enough properties and rules about a given *`symbol`* with new_...commands, it can be a good thing to "close" it. Note that the first `new_elim` command closes the definition for elimination rules, the first `new_intro` command closes the definition for introduction rules. In case these two commands are used, `close_def` ends it by closing the definition for unification.

For (bad) implementation reasons the prefix `Local` is necessary in case it is used for the definition of the symbol (see `def` command). If not the definition will not be really local.

`edel` *`extension-list item`*. Deletes the given *`item`* from the *`extension-list`*.

Possible extension lists are: `rewrite` (the list of rewriting rules introduced by the `new_equation` command), `elim`, `intro`, (the introduction and elimination rules introduced by the `new_elim` and `new_intro {-t}` commands), `closed` (closed definitions introduced by the `close_def` command) and `tex` (introduced by the `tex_syntax` command). The *items* can be names of theorems (`new_...`), or symbols (`close_def` and `tex_syntax`). Use the `eshow` command for listing extension lists.

```
>phox> edel elim All_rec.
delete All_rec from ##elim_ext
```

See also the `del` command.

`elim_after_intro` *`symbol`*. Warning: this command will disappear soon.

Tells the trivial tactic to try an elimination using an hypothesis starting with the *`symbol`* constructor only if no introduction rule can be applied on the current goal. (This seems to be useful only for the negation).

```
>phox> def Prefix[6.3] "~" X = X -> False.
\X (X -> False) : Form -> Form
```

```
>phox> elim_after_intro $~.
Symbol added to "elim_after_intro" list.
```

`{Local} new_elim {-i} {-n} {-t}` ***`symbol name`*** `{`*`num`*`}` ***`theorem`***. If the *theorem* has the following shape: $\forall\chi_1 ... \forall\chi_n(A_1 \rightarrow ... \rightarrow A_n \rightarrow B \rightarrow C)$ where *symbol* is the head of B (the quantifier can be of any order and intermixed with the implications if you wish). Then this theorem can be added as an elimination rule for this *symbol*. B is the main premise, $A_1, ..., A_n$ are the other premises and C is the conclusion of the rule.

The *name* is used as an abbreviation when you want to precise which rule to apply when using the `elim` command.

The optional *num* tells that the principal premise is the *num*th premise whose head is *symbol*. The default is to take the first so this is useful only when the first premise whose head is *symbol* is not the principal one.

```
>phox> goal /\X /\Y (X & Y -> X).

   |- /\ X,Y (X & Y -> X)
>phox> trivial.
proved
>phox> save and_elim_l.
Building proof .... Done.
Typing proof .... Done.
Verifying proof .... Done.
>phox> goal /\X /\Y (X & Y -> Y).

   |- /\ X,Y (X & Y -> Y)
>>phox> trivial.
proved
>phox> save and_elim_r.
Building proof .... Done.
Typing proof .... Done.
Verifying proof .... Done.
>phox> new_elim $& l and_elim_l.
>phox> new_elim $& r and_elim_r.
```

If the leftmost proposition of the theorem is a propositional variable (and then positively universally quantified), the rule defined by `new_elim` is called a *left* rule, that is like left rules of sequent calculus.

The option [-i] tells the tactic trivial not to backtrack on such a left rule. This option will be refused by the system if the theorem donnot define a left rule. The option should be used for an *invertible* left rule, that is a rule that can commute with other rules. A non sufficient condition is that premises of the rule are equivalent to the conclusion.

A somewhat degenerate (there is no premises) case is :

```
>phox> proposition false.elim
```

```
  /\X (False -> X).
trivial.
save.
%phox% 0 goal created.
proved
%phox% Building proof ....Done
Typing proof ....Done
Verifying proof ....Done
Saving proof ....Done
>phox> new_elim -i False n false.elim.
Theorem added to elimination rules.
```

The option [-n] tells the trivial tactic not to try to use this rule, except if [-i] is also used. In this last case the two options [-i -n] tell the tactic trivial to apply this rule first, and use it as the `left` proof command, that is only once. Recall that in this case the left rule should be invertible. For instance :

```
>phox> proposition conjunction.left
  /\X,Y,Z ((Y -> Z -> X) -> Y & Z -> X).
trivial.
save.
>phox>
   |- /\X,Y,Z ((Y -> Z -> X) -> Y & Z -> X)

%phox% 0 goal created.
proved
%phox% Building proof ....Done
Typing proof ....Done
Verifying proof ....Done
Saving proof ....Done
>phox> new_elim -n -i $& s conjunction.left.
Theorem added to elimination rules.
```

The option [-t] should be used for transitivity theorems. It gives some optimisations for automatic tactics (subject to changes).

The prefix `Local` tells that this rule should not be exported. This means that if you use the `Import` or `Use` command, only the exported rules will be added.

You should also note that once one elimination rule has been introduced, the *symbol* definition is no more expanded by the `elim` tactic. The elim tactic only tries to apply each elimination rule. Thus if a connective needs more that one elimination rules, you should prove all the corresponding theorems and then use the `new_elim` command.

`new_equation {-l|-r|-b}` *name* …. Add the given equations or conditional equations to the equational reasoning used in conjunction with the high order unification algorithm. *name* must be a claim or a theorem with at least one equality as an atomic formula which is reachable from the top of the formula by going under a universal quantifier or a conjunction or to the right of an implication. This means that a theorem like $\forall x(Ax \to f(x) = t \;\&\; g(x) = u)$ can be added as a conditional equation. Moreover equations of the form $x = y$ where x and y are variables are not allowed.

the option "-l" (the default) tells to use the equation from left to right. The option "-r" tells to use the equation from right to left. The option "-b" tells to use the equation in both direction.

```
>phox> claim add_O /\y:N (O + y = y).
>phox> claim add_S /\x,y:N (S x + y = S (x + y)).
>phox> new_requation add_O.
>phox> new_requation add_S.
>phox> goal /\x:N (x = O + x).
trivial.
>phox> proved
```

`{Local} new_intro {-n} {-i} {-t} {-c}` ***name theorem*.** If the *theorem* has the following shape: $\forall\chi_1 ... \forall\chi_n(A_1 \to ... \to A_n \to C)$ (the quantifier can be of any order and intermixed with the implications if you wish), then this theorem can be added as an introduction rule for ***symbol***, where ***symbol*** is the head of C. The formulae $A_1, ..., A_n$ are the premises and C is the conclusion of the rule.

The *name* is used as an abbreviation when you want to precise which rule to apply when using the `intro` command.

The option [-n] tells the trivial tactic not to try to use this rule. The option [-i] tells the trivial tactic this rule is invertible. This implies that the trivial tactic will not try other introduction rules if an invertible one match the current goal, and will not backtrack on these rules.

The option [-t] should be used when this rule is a totality theorem for a function (like $\forall x, y(Nx \to Ny \to N(x+y))$), the option [-c] for a totality theorem for a "constructor" like 0 or successor on natural numbers. It can give some optimisations on automatic tactics (subject to changes). For the flag `auto_type` to work properly we recommend to use the option [-i] together with these two options (totality theorems are in general invertible).

The prefix `Local` tells that this rule should not be exported. This means that if you use the `Import` or `Use` command, only the exported rules will be added.

You should also note that once one introduction rule has been introduced, the ***symbol*** (head of C) definition is no more expanded by the `intro` tactic. The intro tactic only tries to apply each introduction rule. Thus if a connective has more that one introduction rules, you should prove all the corresponding theorems and then use the `new_intro command`.

```
>phox> goal /\X /\Y (X -> X or Y).

   |- /\ X /\ Y (X -> X or Y)
>phox> trivial.
```

```
proved
>phox> save or_intro_l.
Building proof .... Done.
Typing proof .... Done.
Verifying proof .... Done.
>phox> goal /\X /\Y (Y -> X or Y).

   |- /\ X /\ Y (Y -> X or Y)
>phox> trivial.
proved
>phox> save or_intro_r.
Building proof .... Done.
Typing proof .... Done.
Verifying proof .... Done.
>phox> new_intro l or_intro_l.
>phox> new_intro r or_intro_r.
```

A.1.4 Inductive definitions.

These macro-commands defines new theories with new rules.

`{Local} Data ….` Defines an inductive data type. See the dedicated chapter.

```
Data Nat n =
  N0  : Nat N0
| S n : Nat n -> Nat (S n)
.

Data List A l =
  nil : List A nil
| [cons] rInfix[3.0] x "::" l :
    A x -> List A l -> List A (x::l)
.

Data Listn A n l =
  nil : Listn A N0 nil
| [cons] rInfix[3.0] x "::" l :
    /\n (A x -> Listn A n l -> Listn A (S n) (x::l))
.

Data Tree A B t =
  leaf a   : A a -> Tree A B (leaf a)
| node b l :
    B b -> List (Tree A B) l -> Tree A B (node b l)
.
```

`{Local} Inductive ….` Defines an inductive predicate. See the dedicated chapter.

```
Inductive And A B =
  left  : A -> And A B
| right : B -> And A B
.

Use nat.

Inductive Less x y =
  zero : /\x Less N0 x
| succ : /\x,y (Less x y -> Less (S x) (S y))
.

Inductive Less2 x y =
  zero : Less2 x x
| succ : /\y (Less2 x y -> Less2 x (S y))
.

Inductive Add x y z =
  zero : Add N0 y y
| succ : /\x,z (Add x y z -> Add (S x) y (S z))
.

Inductive [Eq] Infix[5] x "==" y =
  zero : N0 == N0
| succ : /\x,y (x == y -> S x == S y)
.
```

A.1.5 Managing files and modules.

`add_path` *`string`*. Add *`string`* to the list of all path. This path list is used to find files when using the `Import, Use and include` commands. You can set the environment variable *PHOXPATH* to set your own path (separating each directory with a column).

```
>phox> add_path "/users/raffalli/phox/examples".
/users/raffalli/phox/examples/

>phox> add_path "/users/raffalli/phox/work".
/users/raffalli/phox/work/
/users/raffalli/phox/examples/
```

`Import` *`module_name`*. Loads the interface file "module_name.afi" (This file is produced by compiling an PhoX file). Everything in this file is directly loaded, no renaming applies and objects of the same name will be merged if this is possible otherwise the command will fail.

A renaming applied to a module will not rename symbols added to the module by the `Import` command (unless the renaming explicitly forces it).

Beware, if `Import` command fails when using PhoX interactively, the file can be partially loaded which can be quite confusing !

`include "filename".` Load an ASCII file as if all the characters in the file were typed at the top-level.

`Use {-n}` *`module_name {renaming}`*`.` Loads the interface file “module_name.afi” (This file is produced by compiling a PhoX file). If given, the renaming is applied. Objects of the same name (after renaming) will be merged if this is possible otherwise the command will fail.

The option `-n` tells `Use` to check that the theory is not extended. That is no new constant or axiom are added and no constant are instantiated by a definition.

The syntax of renaming is the following:

`renaming` := *`renaming_sentence`* `{ |` *`renaming`* `}`

A *`renaming_sentence`* is one of the following (the rule matching explicitly the longest part of the original name applies):

- *`name1`* `->` *`name2`* : the symbol *name1* is renamed to *named2*.
- `_.`*`suffix1`* `-> _.`*`suffix2`* : any symbol of the form *xxx.suffix1* is renamed to *xxx.suffix2* (a suffix can contain some dots).
- `_.`*`suffix1`* `-> _` : any symbol of the form *xxx.suffix1* is renamed to *xxx*.
- `_ -> _.`*`suffix2`* : any symbol of the form *xxx* is renamed to *xxx.suffix2*.
- `from` *`module_name`* `with` *`renaming`*`.` : symbols created using the command `Import` *`module_name`* will be renamed using the given *renaming* (By default they would not have been renamed).

A renaming applied to a module will rename symbols added to the module by the `Use` command.

Beware, if `Use` command fails when using PhoX interactively, the file can be partially imported which can be quite confusing !

A.1.6 TeX.

`{Local} tex_syntax` *`symbol syntax`*`.` Tells PhoX to use the given syntax for this *symbol* when producing TeX formulas.

The prefix `Local` tells that this definition should not be exported. This means that if you use the `Import` or `Use` command, only the exported definitions will be added.

A.1.7 Obtaining some informations on the system.

`depend` *`theorem`*`.` Gives the list of all axioms which have been used to prove the *`theorem`*.

```
>phox> depend add_total.
add_S
add_0
```

`eshow` *`extension-list`*. Shows the given *`extension-list`*. Possible extension lists are (See `edel`): `equation` (the list of equations introduced by the `new_equation` command), `elim`, `intro`, (the introduction and elimination rules introduced by the `new_elim` and `new_intro {-t}` commands), `closed` (closed definitions introduced by the `close_def` command) and `tex` (introduced by the `tex_syntax` command).

```
>phox> eshow elim.
All_rec
and_elim_l
and_elim_r
list_rec
nat_rec
```

`flag` *`name`*. **or** `flag` *`name value`*. Prints the value (in the first form) or modify an internal flags of the system. The different flags are listed in the index B.

```
>phox> flag axiom_does_matching.
axiom_does_matching = true
>phox> flag axiom_does_matching false.
axiom_does_matching = false
```

`path`. Prints the list of all paths. This path list is used to find files when using the `include` command.

```
>phox> path.
/users/raffalli/phox/work/
/users/raffalli/phox/examples/
```

`print` *`expression`*. In case *expression* is a closed expression of the language in use, prints it and gives its sort, gives an (occasionally) informative error message otherwise. In case *expression* is a defined expression (constant, theorem ...) gives the definition.

```
>PhoX> print \x,y (y+x).
\x,y (y + x) : nat -> nat -> nat
>PhoX> print \x (N x).
N : nat -> prop
>PhoX> print N.
N = \x /\X (X N0 -> /\y:X  X (S y) -> X x) : nat -> prop
>PhoX> print equal.extensional.
equal.extensional = /\X,Y (/\x X x = Y x -> X = Y) : theorem
```

`print_sort` *`expression`*. Similar to print, but gives more information on sorts of bounded variable in expressions.

```
>PhoX> print_sort \x,y:<nat (y+x).
\x:<nat,y:<nat (y + x) : nat -> nat -> nat
>PhoX> print_sort N.
N = \x:<nat /\X:<nat -> prop (X N0 -> /\y:<nat X (S y) -> X x)
  : nat -> prop
```

`priority` *`list of symbols`*. Print the priority of the given *`symbols`*. If no symbol are given, print the priority of all infix and prefix symbols.

```
>PhoX> priority N0 $S $+ $*.
S                   Prefix[2]           nat -> nat
*                   rInfix[3]           nat -> nat -> nat
+                   rInfix[3.5]         nat -> nat -> nat
N0                                      nat
```

`search` *`string type`*. Prints the list of all symbols which have the *`type`* and whose name contains the *`string`*. If no *`type`* is given, it prints all symbols whose name contains the *`string`*. If the empty string is given, it prints all symbols which have the *`type`*.

```
>PhoX> Import nat.
...
>PhoX> search "trans"
>PhoX> .
equal.transitive                        theorem
equivalence.transitive                      theorem
lesseq.ltrans.N                         theorem
lesseq.rtrans.N                         theorem
>PhoX> search "" nat -> nat -> prop.
!=                  Infix[5]            'a -> 'a -> prop
<                   Infix[5]            nat -> nat -> prop
<=                  Infix[5]            nat -> nat -> prop
<>                  Infix[5]            nat -> nat -> prop
=                   Infix[5]            'a -> 'a -> prop
>                   Infix[5]            nat -> nat -> prop
>=                  Infix[5]            nat -> nat -> prop
predP                                   nat -> nat -> prop
```

A.1.8 Term-extraction.

Term-extraction is experimental. You need to launch `phox` with option `-f` to use it. At this moment (2001/02) there is a bug that prevents to use correctly command `Import` with option -f.

A $\lambda\mu$-term is extracted from in proof in a way similar to the one explained in Krivine's book of lambda-calcul for system Af2. To summarise rules on universal quantifier and equational reasoning are forgotten by extraction.

`compile` *`theorem_name`*. This command extracts a term from the current proof of the theorem *`theorem_name`*. The extracted term has then the same name as the theorem.

`tdef` *`term_name= term.`* This commands defines *`term_name`* as *`term`*.

`eval [-kvm` *term.*] This command normalises the term in $\lambda\mu$-calcul, and print the result. With `-kvm` option, Krivine's syntax is used for output.

`output [-kvm` {*term_name*$_1$ ...*term_name*$_n$}.] This command prints the given arguments *`term_name`*$_1$...*`term_name`*$_n$, prints all defined terms (by `compile` or `tdef`) if no argument is given. With `-kvm` option, Krivine's syntax is used for output.

`tdel {`*`term_name`*$_1$ ...*`term_name`*$_n$`}.` This commands deletes the terms *`term_name`*$_1$...*`term_name`*$_n$ given as arguments. If no argument is given, the command deletes *all* terms, except `peirce_law`. These terms are the ones defined by the commands `compile` and `tdef`. The term `peirce_law` is predefined, but can be explicitly deleted with `tdel peirce_law`.

A.2 Proof commands.

The command described in this section are available only after starting a new proof using the `goal` command. Moreover, except `save` and `undo` they can't be use after you finished the proof (when the message `proved` has been printed).

A.2.1 Basic proof commands.

All proof commands are complex commands, using unification and equational rewriting. The following ones are extensions of the basic commands of natural deduction, but much more powerful.

`axiom` *`hypname.`* Tries to prove the current goal by identifying it with hypothesis *hypname*, using unification and equational reasoning.

```
...
G := X (?1 + N0)
   |- X (N0 + ?2)
%PhoX% axiom G.
0 goal created.
proved
%
```

```
...
H := N x
H0 := N y
H1 := X (x + S N0)
   |- X (S x)
%PhoX% axiom H1.
0 goal created.
proved
```

`elim` {*num0*} *expr0* `{ with` *opt1* `{and/then ... {and/then` *optn*`}...} .` [2]]

This command corresponds to the following usual tool in natural proof : prove the current goal by applying hypothesis or theorem `expr0`. More formally this command tries to prove the current goal by applying some elimination rules on the formula or theorem *expr0* (modulo unification and equational reasoning). Elimination rules are built in as the ordinary ones for forall quantifier and implication. For other symbols, elimination rules can be defined with the `new_elim`) commands.

After this tactic succeeds, all the new goals (Hypothesis of `expr0` adapted to this particular case) are printed, the first one becoming the new current goal.

```
New goal is:
goal 1/1
H := N x
H0 := N y
H1 := N z
   |- x + y + z = (x + y) + z

%PhoX% elim H.  (* the default elimination rule for predicate N
                   is induction *)
2 goals created.

New goals are:
goal 1/2
H := N x
H0 := N y
H1 := N z
   |- N0 + y + z = (N0 + y) + z

goal 2/2
H := N x
H0 := N y
H1 := N z
H2 := N y0
H3 := y0 + y + z = (y0 + y) + z
   |- S y0 + y + z = (S y0 + y) + z
```

The following example use equational rewriting :

```
H := N x
H0 := N y
H1 := N z
   |- x + y + z = (x + y) + z
```

[2]Curly braces denote an optional argument. You should note type them.

```
%PhoX% elim equal.reflexive.
(* associativity equations are in library nat *)
0 goal created.
```

You have the option to give some more informations *opti*, that can be expressions (individual terms or propositions), or abbreviated name of elimination rules.

Expressions has to be given between parenthesis if they are not variables. They indicate that a for-all quantifier (individual term) or an implication (proposition) occuring (strictly positively) in `expr0` has to be eliminated with this expression. In case there is only one such option, the first usable occurence form left to right is used (regardless the goal).

```
def lInfix[5] R "Transitive" =
  /\x,y,z ( R x y -> R y z -> R x z).
...

H := R Transitive
H0 := R a b
H1 := R b c
   |- R a c
%PhoX% elim H with H0.
1 goal created, with 1 automatically solved.
```

but

```
H := R Transitive
H0 := R a b
H1 := R b c
   |- R a c

%PhoX% elim H with H1.
Error: Proof error: Tactic elim failed.
```

You can pass several options separated by `and` or `then`. In case `opti` is introduced by an `and`, the tactic search the first unused occurrence in `expr0` of forall quantifier, implication or connective usable with `opti`.

```
H := R Transitive
H0 := R a b
H1 := R b c
   |- R a c

%PhoX% elim H with a and b and c.
0 goal created.
```

to skip a variable or hypothesis you can use unification variables (think that `?` match any variable or hypothesis) :

```
H := R Transitive
H0 := R a b
H1 := R b c
   |- R a c

%PhoX% elim H with ? and b.  (* ? will match a *)
0 goal created.
```

In case `opti` is introduced by a `then` : `...` *`opti`* `then` *`opti'`* `...`, the tactic search the first unused occurrence of forall quantifier, implication or connective usable with `opti'` *after* the occurrence used for `opti`.

```
H := R Transitive
H0 := R a b
H1 := R b c
   |- R a c

%PhoX% elim H with H0 and a.
0 goal created.
```

but

```
H := R Transitive
H0 := R a b
H1 := R b c
   |- R a c

%PhoX% elim H with H0 then a.
Error: Proof error: Tactic elim failed.
```

Abbreviated name of elimination rules have to be given between square brackets. The tactic try to uses this elimination rule for the first connective in `expr0` using it.

```
H := N x
   |- x = N0 or \/y:N  x = S y

%PhoX% elim H with [case].
2 goals created.

New goals are:
goal 1/2
H := N x
```

```
H0 := x = N0
   |- N0 = N0 or \/y:N  N0 = S y

goal 2/2
H := N x
H0 := N y
H1 := x = S y
   |- S y = N0 or \/y0:N  S y = S y0
```

You can use abbreviated names and expression, `and` and `then` together. All occurrences matched after a `then` *opti* have to be after the one matched by *opti*. The `and` matches the first unused occurrence with respect to the previous constraint on a possible `then` placed before.

```
H := /\x:N  ((x = N0 -> C) & ((x = N1 -> C) & (x = N2 -> C)))
   |- C

%PhoX% elim H with N1 and [r] then [l].
2 goals created.

New goals are:
goal 1/2
H := /\x:N  ((x = N0 -> C) & ((x = N1 -> C) & (x = N2 -> C)))
   |- N N1

goal 2/2
H := /\x:N  ((x = N0 -> C) & ((x = N1 -> C) & (x = N2 -> C)))
   |- N1 = N1
```

The first option *num0* is not very used. It allows to precise the number of elimination rules to apply.

`elim` {*num0*} {-*num1 opt1*} ... {-*numn optn*} *expr0*.[3] *This syntax is now deprecated* but still used in libraries and examples. Use the syntax above!

Tries to prove the current goal by applying some elimination rules on the formula or theorem *expr0*. You have the option to precise a minimum number of elimination rules (*num0*) or/and give some information *opti* to help in finding the *numi*-th elimination.

- If the *numi*-th elimination acts on a for-all quantifier, *opti* must be an expression which can be substituted to this variable (this expression has to be given between parenthesis if it is not a variable).
- If the *numi*-th elimination acts on an implication, *opti* must be an expression which can be unified with the left formula in the implication (this expression has to be given between parenthesis if it is not a variable).
- If the *numi*-th elimination acts on a connective for which you introduced new elimination rules (using `new_elim`), *opti* has to be the abbreviated name of one of these rules, between square bracket.

Moreover, this tactic expands the definition of a symbol if and only if this symbol has no elimination rules.

After this tactic succeeded, all the new goals are printed, the last one to be printed is the new current goal.

```
>phox> goal /\x/\y/\z (N x -> N y -> N z ->
  x + (y + z) = (x + y) + z).

   |- /\ x /\ y /\ z (N x -> N y -> N z ->
  x + y + z = (x + y) + z)
>phox> intro 6.

H := N x
H0 := N y
H1 := N z
   |- x + y + z = (x + y) + z
>phox> elim -4 x nat_rec.

H := N x
H0 := N y
H1 := N z
   |- /\ y0 (N y0 -> y0 + y + z = (y0 + y) + z ->
  S y0 + y + z = (S y0 + y) + z)

H := N x
H0 := N y
H1 := N z
   |- 0 + y + z = (0 + y) + z

>phox> elim equal_refl.

H := N x
H0 := N y
H1 := N z
   |- /\ y0 (N y0 -> y0 + y + z = (y0 + y) + z ->
  S y0 + y + z = (S y0 + y) + z)
```

`intro` *`num`*. **or** `intro` *`info1 .... infoN`* In the second form, *`infoX`* is either an identifier *`name`*, either an expression of the shape `[`*`name opt`*`]` and *`opt`* is empty or is a "with" option for an `elim` command.

In the first form, apply *`num`* introduction rules on the goal formula. New names are automatically generated for hypothesis and universal variables. In this form, the intro command only uses the last intro rule specified for a given connective by the `new_intro` command.

In the second form, for each *`name`* an intro rule is applied on the goal formula. If the outermost connective is an implication, the *`name`* is used as a new name for the hypothesis. If it is an universal quantification, the *name* is used for the new variable. If it is a connective with introduction rules defined by the `new_intro` command, *name* should be the name of one of these rules and this rule will be applied with the given `elim` option is some where given.

Moreover, this tactic expands definition of a symbol if and only if this symbol has no introduction rules.

```
>phox> goal /\x /\y (N x -> N y -> N (x + y)).

   |- /\ x /\ y (N x -> N y -> N (x + y))
>phox> intro 7.

H := N x
H0 := N y
H1 := X 0
H2 := /\ y0 (X y0 -> X (S y0))
   |- X (x + y)
>phox> abort.
>phox> goal /\X /\Y /\x (X x & Y -> \/x X x or Y).

   |- /\ X /\ Y /\x (X x & Y -> \/x X x or Y)
>phox>  intro A B a H l.

H := A a & B
   |- \/x A x

>phox> intro [n with a].

H := A x & B
   |- A a
```

`intros` {*symbol_list*}`.` Apply as many introductions as possible without expanding a definition. If a *symbol_list* is given only rules for these symbols are applied and only defined symbols in this list are expanded. If no list is given, Definitions are expanded until the head is a symbol with some introduction rules and then only those rules will be applied and those definition will be expanded (if this head symbol is an implication or a universal quantification, introduction rules for both implication and universal quantification will be applied, as showed by the following example).

```
>phox> goal /\x /\y (N x -> N y -> N (x + y)).

   |- /\ x,y (N x -> N y -> N (x + y))
>phox> intros.

H0 := N y
H := N x
   |- N (x + y)
```

A.2.2 More proof commands.

`apply { with` *`opt1`* `{and/then ... { and/then` *`optn`*`}...}` . Equivalent to `use ?. elim ...` . Usage is similar to `elim` (see this entry above for details). The command `apply` adds to the current goal a new hypothesis obtained by applying *expr0* (an hypothesis or a theorem) to one or many hypothesis of the current goal. as for `elim`, if there are unproved hypothesis of *expr0* they are added as new goals. The difference with `elim`, is that `apply` has not to prove the current goal.

```
H0 := /\a0,b (R a0 b -> R b a0)
H1 := /\x \/y R x y
H := /\a0,b,c (R a0 b -> R b c -> R a0 c)
   |- R a a

%PhoX% apply H1 with a.

...
G := \/y R a y
   |- R a a

%PhoX% left G.
...
H2 := R a y
   |- R a a

%PhoX% apply H0 with H2.
...
G := R y a
   |- R a a

[%PhoX% elim H with ? and y and ?. (* concludes *)]
[%Phox% elim H with H2 and G. (* concludes *)]
[%Phox% apply H with H2 and G. (* concludes if auto_lvl=1. *)]

%Phox% apply H with a and y and a. (* does not conclude. *)
...
G0 := R a y -> R y a -> R a a
   |- R a a
...
```

`apply {`*`num0`*`} {-`*`num1 opt1`*`} ... {-`*`numn optn`*`}` *`expr0`*. Old syntax for apply, don't use it ! See `elim`.

`by_absurd.` Equivalent to `elim absurd. intro.`

`by_contradiction.` Equivalent to `elim contradiction. intro.`

`from` *`expr`*. Try to unify *`expr`* (which can be a formula or a theorem) with the current goal. If it succeeds, *`expr`* replace the current goal.

```
>phox> goal /\x/\y/\z (N x -> N y -> N z ->
  x + (y + z) = (x + y) + z).

   |- /\ x /\ y /\ z (N x -> N y -> N z ->
  x + y + z = (x + y) + z)
>phox> intro 6.
....

....

H := N x
H0 := N y
H1 := N z
H2 := N y0
H3 := y0 + y + z = (y0 + y) + z
   |- S y0 + y + z = (S y0 + y) + z
>phox> from S (y0 + y + z) = S (y0 + y) + z.

H := N x
H0 := N y
H1 := N z
H2 := N y0
H3 := y0 + y + z = (y0 + y) + z
   |- S (y0 + y + z) = S (y0 + y) + z
>phox> from S (y0 + y + z) = S ((y0 + y) + z).

H := N x
H0 := N y
H1 := N z
H2 := N y0
H3 := y0 + y + z = (y0 + y) + z
   |- S (y0 + y + z) = S ((y0 + y) + z)
>phox> trivial.
proved
```

`left` ***hypname*** `{`*`num`*` | `*`info1 .... infoN`*`}.` An elimination rule whose conclusion can be any formula is called a left rule. The left command applies left rules to the hypothesis of name *hypname*. If an integer *num* is given, then *num* left rule are applied. The arguments *info1 .... infoN* are used as in the `intro` command.

```
>phox> goal /\X,Y (\/x (X x or Y) -> Y or \/x X x).

   |- /\X /\Y (\/x (X x or Y) -> Y or \/x X x)

%phox% intros.
1 goal created.
New goal is:
```

```
H := \/x (X x or Y)
   |- Y or \/x X x

%phox% left H z.
1 goal created.
New goal is:

H0 := X z or Y
   |- Y or \/x X x

%phox% left H0.
2 goals created.
New goals are:

H1 := X z
   |- Y or \/x X x


H1 := Y
   |- Y or \/x X x

%phox% trivial.
0 goal created.
Current goal now is:

H1 := X z
   |- Y or \/x X x

%phox% trivial.
0 goal created.
proved
```

`lefts` ***hypname*** `{`***symbol_list***`}.` Applies "many" left rules on the hypothesis of name *hypname*. If a *symbol_list* is given only rules for these symbols are applied and only defined symbols in this list are expanded. If no list is given, Definitions are expanded until the head is a symbol with some left rules and then only those rules will be applied and those definitions will be expanded.

```
>phox> goal /\X,Y (\/x (X x or Y) -> Y or \/x X x).

   |- /\X /\Y (\/x (X x or Y) -> Y or \/x X x)

%phox% intros.
1 goal created.
New goal is:
```

```
H := \/x (X x or Y)
   |- Y or \/x X x

%phox% lefts H $\/ $or.
2 goals created.
New goals are:

H1 := X x
   |- Y or \/x0 X x0


H1 := Y
   |- Y or \/x0 X x0

%phox% trivial.
0 goal created.
Current goal now is:

H1 := X x
   |- Y or \/x0 X x0

%phox% trivial.
0 goal created.
proved

...
H := N x
H0 := N y
H1 := N y0
H2 := S y0 <= S y
   |- S y0 <= y or S y0 = S y
%PhoX% print lesseq.S_inj.N.
lesseq.S_inj.N = /\x0,y1:N  (S x0 <= S y1 -> x0 <= y1) : theorem
%PhoX% apply -5 H2 lesseq.S_inj.N.
3 goals created, with 2 automatically solved.

New goal is:
H := N x
H0 := N y
H1 := N y0
H2 := S y0 <= S y
G := y0 <= y
   |- S y0 <= y or S y0 = S y
```

Another example (in combination with `rmh`) :

```
...
```

```
H := List D0 l
H0 := D0 a
H1 := List D0 l'
H2 := /\n0:N  (n0 <= length l' -> List D0 (nthl l' n0))
H4 := N y
G := y <= length l'
   |- List D0 (nthl (a :: l') (S y))

%PhoX% apply -3 G H2 ;; rmh H2.
2 goals created, with 1 automatically solved.
New goal is:

H := List D0 l
H0 := D0 a
H1 := List D0 l'
H4 := N y
G := y <= length l'
G0 := List D0 (nthl l' y)
   |- List D0 (nthl (a :: l') (S y))
```

`prove` ***expr***. Adds ***expr*** to the current hypothesis and adds a new goal with ***expr*** as conclusion, keeping the hypothesis of the current goal (cut rule). ***expr*** may be a theorem, then no new goal is created. The current goal becomes the new statment.

```
>phox> goal /\x1/\y1/\x2/\y2 (pair x1 y1 = pair x2 y2
  -> x1 = x2 or y1 = x2).

   |- /\ x1 /\ y1 /\ x2 /\ y2 (pair x1 y1 = pair x2 y2
  -> x1 = x2 or y1 = x2)
>phox> intro 5.

H := pair x1 y1 = pair x2 y2
   |- x1 = x2 or y1 = x2
>phox> prove pair x2 y2 = pair x1 y1.

H := pair x1 y1 = pair x2 y2
G := pair x2 y2 = pair x1 y1
   |- x1 = x2 or y1 = x2

H := pair x1 y1 = pair x2 y2
   |- pair x2 y2 = pair x1 y1
```

`use` ***expr***. Same as `prove` command, but keeps the current goal, only adding ***expr*** to hypothesis.

A.2.3 Automatic proving.

Almost all proof commands use some kind of automatic proving. The following ones try to prove the formula with no indications on the rules to apply.

`auto` ... Tries `trivial` with bigger and bigger value for the depth limit. It only stops when it succeed or when not enough memory is available. This command uses the same option as `trivial` does.

`trivial {`*`num`*`} {-|= `*`name1 ... namen`*`} {+ `*`theo1 ... theop`*`}.`[4] Try a simple trivial tactic to prove the current goal. The option *num* give a limit to the number of elimination performed by the search. Each elimination cost (1 + number of created goals).

The option {`- `*`name1 ... namen`*} tells `trivial` not to use the hypothesis *name1 ... namen*. The option {`= `*`name1 ... namen`*} tells `trivial` to only use the hypothesis *name1 ... namen*. The option `+ `*`theo1 ... theop`* tells `trivial` to use the given theorem.

```
>phox> goal /\x/\y (y E pair x y).

   |- /\x/\y (y E pair x y)
>phox> trivial + pair_ax.
proved.
```

A.2.4 Rewriting.

`rewrite {-l `*`lim`*` | -p `*`pos`*` | -ortho} {{-r|-nc} `*`eqn1`*`} {{-r|-nc} `*`eqn2`*`} ...` If *eqn1*, *eqn2*, ... are equations (or conditional equations) or list of equations (defined using `def_thlist`), the current goal is rewritten using these equations *as long as possible.* For each equation, the option `-r` indicates to use it from right to left (the default is left to right) and the option `-nc` forces the system not to try to prove automatically the conditions necessary to apply the equation (the default is to try).

```
...
H0 := N y
H1 := N z
H2 := N y0
H3 := y0 * (y + z) = y0 * y + y0 * z
   |- S y0 * (y + z) = S y0 * y + S y0 * z

%PhoX% print mul.lS.N.
mul.lS.N = /\x0,y1:N  S x0 * y1 = y1 + x0 * y1 : theorem
%PhoX% rewrite mul.lS.N.
1 goal created.
New goal is:

H0 := N y
H1 := N z
H2 := N y0
```

```
H3 := y0 * (y + z) = y0 * y + y0 * z
   |- (y + z) + y0 * (y + z) = (y + y0 * y) + z + y0 * z

%PhoX% rewrite H3.
1 goal created.
New goal is:

H0 := N y
H1 := N z
H2 := N y0
H3 := y0 * (y + z) = y0 * y + y0 * z
   |- (y + z) + y0 * y + y0 * z = (y + y0 * y) + z + y0 * z
...
```

If *sym1*, *sym2*, are defined symbol, their definition will be expanded. Do not use `rewrite` just for expansion of definitions, use `unfold` instead.

Note: by default, `rewrite` will unfold a definition if and only if it is needed to do rewriting, while `unfold` will not (this mean you can use `unfold` to do rewriting if you do not want to perform rewriting under some definitions).

The global option `-l` *lim* limits to *lim* steps of rewriting. The option `-p` *pos* indicates to perform only one rewrite step at the *pos*-th possible occurrence (occurrences are numbered from 0). These options allows to use for instance commutativity equations. The option `-ortho` tells the system to apply rewriting from the inner subterms to the root of the term (if a rewrite rule r_2 is applied after another rule r_1, then r_2 is not applied under r_1). This restriction ensures termination, but do not always reach the normal form when it exists.

```
...
H := N x
H0 := N y
H1 := N z
   |- (y + z) * x = y * x + z * x

%PhoX% rewrite -p 0 mul.commutative.N.
1 goal created.
New goal is:

H := N x
H0 := N y
H1 := N z
   |- x * (y + z) = y * x + z * x

%PhoX% rewrite -p 1 mul.commutative.N.
1 goal created.
New goal is:
H := N x
H0 := N y
H1 := N z
   |- x * (y + z) = x * y + z * x
...
```

`rewrite_hyp` *`hyp_name`* `...` Similar to `rewrite` except that it rewrites the hypothesis named *`hyp_name`*. The dots (...) stands for the `rewrite` arguments.

`unfold ...` A synonymous to `rewrite`, use it when you only do expansion of definitions.

`unfold_hyp` *`hyp_name`* `...` A synonymous to `rewrite_hyp`, use it when you only do expansion of definitions.

A.2.5 Managing existential variables.

Existential variables are usually designed in phox by `?x` where `x` is a natural number. They are introduced for instance by applying an `intro` command to an existential formula, or sometimes by applying an `elim H` command where `H` is an universal formula.

You can use existential variables in goals, for instance :

```
>PhoX> goal N3^N2=?1.

Goals left to prove:

   |- N3 ^ N2 = ?1

%PhoX% rewrite calcul.N.
1 goal created.
New goal is:

Goals left to prove:

   |- S S S S S S S S S N0 = ?1

%PhoX% intro.
0 goal created.
proved
%PhoX% save essai.
Building proof ....
Typing proof ....
Verifying proof ....
Saving proof ....
essai = N3 ^ N2 = S S S S S S S S S N0 : theorem
```

`constraints.` Print the constraints which should be fulfilled by unification variables.

```
    >phox> goal /\X (/\x\/y X x y -> \/y/\x X x y).
```

```
   |- /\ X (/\ x \/ y X x y -> \/ y /\ x X x y)
%phox% intro 4.

H := /\ x0 \/ y X x0 y
   |- X x ?32
%phox% constraints.
Unification variable ?32 should not use x
```

`instance` *expr0 expr1.* :

Unify *expr0* and *expr1*. This is useful to instantiate some unification variables. *expr0* must be a variable or an expression between parenthesis.

```
H := N x
H0 := N y
H3 := y = N2 * X + N1
   |- S y = N2 * ?792
>phox> instance ?792 S X.

H := N x
H0 := N y
H3 := y = N2 * X + N1
   |- S y = N2 * S X
```

`lock` *var.* : This command *locks* the existential variable (or meta-variable, or unification variable) *var* for unification. That is for all succeeding commands *var* is seen as a constant, except the command `instance` that makes the existential variable disappear, and the command `unlock` that explicitely unlocks the existential variable. When introduced in a proof, it is possible that you still donnot know the value to replace an existential variable by. As there is no more general unifier in presence of high order logic and equational reasoning, somme commands could instanciate an unlocked existential variable in an unexpected way.

For instance in the following case :

```
%PhoX% local y = x - k.
...
%PhoX% prove N y.
2 goals created.
New goals are:

Goals left to prove:

H := N k
H0 := N n
G := N ?1
H1 := N x
```

```
H2 := x >= ?1
G0 := k <= ?1
   |- N y
...
%PhoX% trivial.
```

if `?1` is not locked, `?1` will be instanciated by `y`, which is not the expected behaviour.

`unlock` *`var`*`.` : This command *unlocks* the existential variable (or existential variable) *var* for unification, in case this variable is locked (see above `lock`). Recall that `instance` unlock automatically the existential variable if necessary.

A.2.6 Managing goals.

`goals.` Prints all the remaining goals, the current goal being the last to be printed, being the first with option `-pg` used for Proof General (cf `next` for an example).

`after {em num}.` Change the current goal. If no *num* is given then the current goal become the last goal. If *num* is given, then the current goal is sent after the *num*th.

`next {em num}.` Change the current goal. If no *num* is given then the current goal becomes the last goal. If a positive *num* is given, then the current goal becomes the *num*th (the 0th being the current goal). If a negative *num* is given, the *num*th goal become the current one (`next -4` is the "inverse" command of `next 4`).

```
>phox> goals.

H := N x
H0 := N y
H1 := N z
   |- /\ y0 (N y0 -> y0 + y + z = (y0 + y) + z ->
  S y0 + y + z = (S y0 + y) + z)

H := N x
H0 := N y
H1 := N z
   |- /\ y0 (N y0 -> 0 + y0 + z = y0 + z ->
  0 + S y0 + z = S y0 + z)

H := N x
H0 := N y
H1 := N z
   |- 0 + 0 + z = 0 + z
>phox> next.

H := N x
H0 := N y
```

```
H1 := N z
   |- /\ y0 (N y0 -> 0 + y0 + z = y0 + z ->
  0 + S y0 + z = S y0 + z)
 ...
```

`select` *`num`*. The *`num`*th goal becomes the current goal.

A.2.7 Managing context.

`local .....` The same syntax as the `def` command but to define symbols local to the current proof (see the `def` (section A.1.2) command for the syntax).

`rename` *`oldname newname`*. Rename a constant or an hypothesis local to this goal (can not be used to rename local definitions).

`rmh` *`name1 ... namen`*. Deletes the hypothesis *`name1, ...,namen`* from the current goal.

```
>phox>  goal /\X /\Y (Y -> X -> X).

   |- /\ X /\ Y (Y -> X -> X)
>phox> intro 3.

H := Y
   |- X -> X
>phox> rmh H.

   |- X -> X
```

`slh` *`name1 ... namen`*. Keeps only the hypothesis *`name1, ...,namen`* in the current goal.

```
>phox> goal /\x,y : N N (x + y).

   |- /\ x,y : N N (x + y)
>phox> intros.

H0 := N y
H := N x
   |- N (x + y)
>phox> slh H.

H := N x
   |- N (x + y)
```

A.2.8 Managing state of the proof.

`abort.` Abort the current proof, so you can start another one !

```
>phox> goal /\X /\Y (X -> Y).

   |- /\ X /\ Y (X -> Y)
>phox> intro 3.

H := H
   |- Y
>phox> goal /\X (X -> X).
Proof error: All ready proving.
>phox> abort.
>phox> goal /\X (X -> X).

   |- /\ X (X -> X)
```

`{Local} save {`*name*`}.` When a proof is finished (the message `proved` has been printed), save the new theorem with the given *name* in the data base. Note: the proof is verified at this step, if an error occurs, please report the bug !

You do not have to give the name if the proof was started with the `theorem` command or a similar one instead of `goal` : the name from the declaration of `theorem` is choosen.

The prefix `Local` tells that this theorem should not be exported. This means that if you use the `Import` or `Use` command, only the exported theorems will be added.

```
>phox>  goal /\x (N x -> N S x).

   |- /\ x (N x -> N (S x))
>phox> trivial.
proved
>phox> save succ_total.
Building proof .... Done.
Typing proof .... Done.
Verifying proof .... Done.
```

`undo {`*num*`}.` Undo the last action (or the last *num* actions).

```
>phox> goal /\X (X -> X).

   |- /\ X (X -> X)
>phox> intro.

   |- X -> X
>phox> undo.

   |- /\ X (X -> X)
```

A.2.9 Tacticals.

This feature is new and has limitations.

`tactic1` `;;` *`tactic2`* Use *`tactic1`* for all goals generated by *`tactic1`*.

`Try` *`tactic`* If *`tactic`* is successful, `Try` *`tactic`* is the same as *`tactic`*. If *`tactic`* fails, `Try` *`tactic`* succeeds and does not modify the current goal. This is useful after a `;;`.

Appendix B

Flags index.

In this index we list all the PhoX flags (see the description of the command `flag` in the index A to learn how to print and modify the value of these flags).

`auto_lvl` (integer, default is 0) : Control the automatic detection of axioms: If it is set to 0 no detection is performed. If it is set to 1, axioms are detected when the goal is structurally equal to an hypothesis. If it is set to 2, axioms are detected when the goal unifies is equal to an hypothesis up to the expansion of some definitions. If it is set to 3, axioms are detected if the goal unifies with an hypothesis (using no equations). We recommend avoiding 3 as it may instantiate variables with the wrong value.

`auto_type` (bool, default is false) : automatically apply all the introduction rule which were introduced with the flags `-i` and `-c` or `-t`. We recommend setting this flag to true and using `auto_lvl` set to 2 to solve automatically all the "typing" goals (like proving that something is an integer).

`binder_tex_space` (integer, default is 3) : set the space after a binder when PhoXis printing TeX formulas.

`comma_tex_space` (integer, default is 5) : set the space after punctuation when PhoXis printing TeX formulas.

`ellipsis_text` (string, default is "...") : the text to be printed when an expression is too deep (used by the pretty printer only).

`eq_breadth` (integer, default is 4) : maximum number of equations used at each step of rewriting.

`eq_flvl` (integer, default is 3) : maximum number of interleaved equations tried without decreasing the distance (the rewriting algorithm uses a distance between first order terms).

`eq_depth` (integer, default is 100) : maximum number of interleaved equations applied by the rewriting algorithm.

`margin` (integer, default is 80) : size of the page (used by the pretty printer only).

`max_indent` (integer, default is 50) : maximum number of indentation (used by the pretty printer only).

`max_boxes` (integer, default is 100) : control the maximum printing "depth". If the expression is too deep, an ellipsis is printed (used by the pretty printer only).

`min_tex_space` (integer, default is 20) : set the minimum space (in 100th of em) between to tokens when PhoX is printing TeX formulas.

`max_tex_space` (integer, default is 40) : set the minimum space (in 100th of em) between to tokens when PhoX is printing TeX formulas.

`tex_indent` (integer, default is 200) : set the indentation space (in 100th of em) used by PhoX when printing multi-lines TeX formulas.

`tex_lisp_app` (boolean, default is true) : If true the syntax $f\ x\ y$ is used for application when producing LaTeX formulas. If false, the syntax $f(x, y)$ is used.

`tex_type_sugar` (boolean, default is true) : If true the syntactic sugar $\forall x : A\ B$ for $\forall x(Ax \rightarrow B)$ is used when producing LaTeX formulas.

`tex_margin` (integer, default is 80) : size of the page used when printing verbatim formulas in TeX.

`tex_max_indent` (integer, default is 50) : maximum number of indentation used when printing verbatim formulas in TeX.

`trivial_depth` (integer, default is 4) : default value for the `trivial` command.

Index

Bibliography

[1] Jean-Louis Krivine. *Lambda-Calcul : Types et Modèles.* Etudes et Recherches en Informatique. Masson, 1990. Available now in english version.

[2] Jean-Louis Krivine and Michel Parigot. Programming with proofs. *Inf. Process. Cybern.*, EIK 26(3):149–167, 1990.

[3] Michel Parigot. Programming with proofs: a second order type theory. *Lecture Notes in Computer Science*, 300, 1988. Communication at ESOP 88.

[4] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. Proc. International Conference on Logic Programming and Automated Deduction, StPetersburg (Russia):190–201, 1992.

[5] Christophe Raffalli and René David. Apprentissage du raisonnement assité par ordinateur. Prépublication 01-09c du LAMA, 2001.

[6] F. P. Ramsey. The foundations of mathematics. The Foundations of Mathematics:1–61, 1925.