Quelques éléments pour débuter avec le vérificateur de preuves PhoX.

R. David – Université de Savoie, C. Raffalli – Université de Savoie,
P. Rozière – Université Paris 7

4 juin 2003

Table des matières

Le système PhoX [2] permet de vérifier interactivement le développement de théories mathématiques, en particulier de vérifier des *preuves*. Il utilise un petit nombre de commandes. Celles-ci sont *extensibles* par l'utilisateur, c'est à dire qu'il peut en modifier le comportement (un aspect qui ne sera pas étudié dans ce document).

1 Les fondements logiques.

Le système PhoX se fonde sur une *logique d'ordre supérieur multisorte*. Le fait qu'il s'agit de *logique d'ordre supérieur* signifie que l'on peut quantifier non seulement sur les objets – par exemples les entiers – mais aussi sur des propriétés sur ces objets (ou des ensembles d'objets), on peut par exemple exprimer dans le langage de l'arithmétique l'axiome de récurrence qui dit que toute propriété vérifiée pour 0 et close par passage au successeur est vérifiée pour tout entier (dit autrement, tout sous-ensemble des entiers qui contient 0 et qui est clos par application du successeur est l'ensemble des entiers). En *logique du premier ordre* – pour l'essentiel la logique étudiée en cours – on ne peut exprimer la récurrence que comme un *schéma d'axiomes*, c'est à dire énumérer les instances de la récurrence pour chaque propriété sur les entiers exprimable dans le langage de l'arithmétique[1]. Ceci est un cas très simple de propriété exprimable à l'ordre supérieur. Il en existe d'autres qui ne peuvent pas du tout s'exprimer en logique du premier ordre tant que l'on reste dans le même langage de base.

La notion de *sorte* (ou de type) permet de distinguer plusieurs sortes d'objet, par exemple des points et des droites en géométrie plane. On pourra ainsi vérifier syntaxiquement que le prédicat "être situé sur" prend bien comme arguments un point et une droite. Ceci diffère à nouveau de la logique du premier ordre étudiée en cours, qui n'utilise qu'une seule sorte d'objet.

La logique d'ordre supérieur contient bien-sûr la logique du premier ordre. Cependant il est possible d'utiliser des théories du premier ordre comme certaines théorie des ensembles plus expressives et plus "fortes" que la logique d'ordre supérieur de PhoX. On fait le choix de la logique d'ordre supérieur non parce qu'elle est plus "forte", mais parce que les sortes permettent d'éliminer un certain nombre d'écritures incorrectes ; cette vérification syntaxique étant réalisée par le système d'une façon qui ne diverge pas trop avec la pratique mathématique usuelle.

L'utilisateur peut étendre le langage en ajoutant de nouvelles sortes, de nouvelles constantes, de nouvelles fonctions, de nouveaux prédicats ... Il peut également ajouter des axiomes.

Enfin pour développer les preuves l'utilisateur a accès à un langage de commandes inspiré de la *déduction naturelle*, une forme abstraite de la preuve formelle mathématique.

2 L'interface.

2.1 La boucle d'interaction.

La communication du système PhoX proprement dit avec l'utilisateur se fait par l'intermédiaire d'une *boucle d'interaction*, c'est à dire que l'utilisateur envoie des *commandes* au système, à chaque commande correspond une réponse du système ; celle-ci indique si la commande est acceptée ou non – dans ce cas elle a pu modifier l'état du système – et quand elle est acceptée donne le plus souvent des indications sur l'état du système qui en résulte.

On distingue essentiellement deux modes de fonctionnement de PhoX, auxquels correspondent deux jeux de commande du système suivant que l'on a ou non entamé une preuve.

Le mode par défaut est celui des *commandes globales* qui permettent d'étendre la syntaxe, de déclarer des axiomes, d'introduire des définitions, plus généralement de personnaliser le système et bien-sûr de basculer en mode preuve. On peut alors utiliser les *commandes de preuve* pour conduire le système à construire la preuve d'un énoncé donné.

Le prompt `>PhoX>` indique que la machine attend une commande globale. Il se transforme en `%PhoX%` quand le système est en mode preuve.

On peut conserver une suite de commandes dans un fichier dont l'extension doit être "`.phx`".

2.2 L'interface Emacs.

La version du système qui vous est présentée utilise une interface pleine page qui permet à la fois d'éditer et de valider un tel fichier. Cette interface communique avec le système par l'intermédiaire

[1] En fait ces deux façons d'exprimer la récurrence ne sont pas vraiment équivalentes.

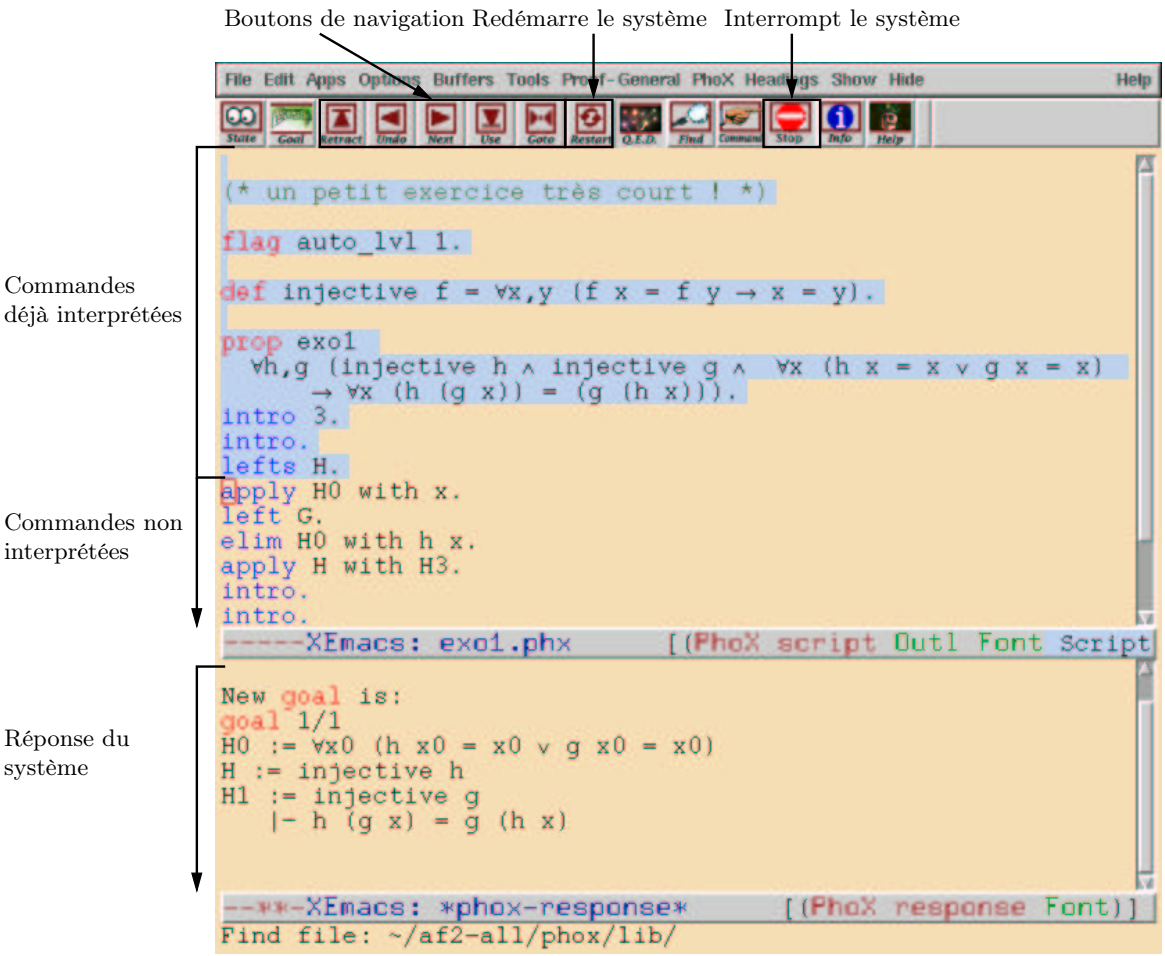

FIG. 1 – Exemple d'écran PhoX

de la boucle d'interaction décrite au paragraphe précédent. Ainsi le fichier est évalué linéairement.

Cette interface a été réalisée à l'aide de XEmacs [1] et ProofGeneral [3] de D. Aspinall. XEmacs est un éditeur de la famille Emacs qui existe sous de nombreux systèmes d'exploitation. La figure 2.2 montre un exemple d'écran PhoX. L'écran est divisé en deux parties : on trouve, dans la partie haute, le *script* des commandes données au système et, dans la partie basse, la réponse de PhoX à la dernière commande.

Les commandes déjà interprétées sont mises en évidence par un fond de couleur différente. Des boutons de navigation permettent d'avancer ou de revenir en arrière au point désiré. Le retour en arrière (en utilisant les boutons de navigations) est la seule façon de modifier une commande déjà interprétée : cela permet de garantir une cohérence entre le script et l'état du système.

Si le système est bien installé il suffit d'éditer sous Xemacs un fichier dont l'extension est `.phx` pour utiliser cette interface.

3 La syntaxe.

PhoX utilise une notation en ligne pour la saisie comme l'affichage. Pour la saisie il utilise les caractères ascii (les lettres usuelles mais pas les lettres accentuées). Cependant, grâce à l'interface Xemacs, certaines suites de caractères peuvent être affichées en utilisant les symboles usuels pour quelques uns d'entre eux $\forall$, $\exists$ ... Les signes `(* *)` délimitent du texte qui ne sera pas évalué par le système (ce que l'on appelle les *commentaires* dans un langage informatique).

3.1 Notation fonctionnelle.

PhoX utilise le plus souvent la *notation fonctionnelle* (utilisée dans les langages informatiques comme LISP ou CAML) : par exemple "$\mathrm{d}\,x\,y$" au lieu de "$\mathrm{d}(x,y)$". Il peut manipuler aussi bien des notations *préfixes*, comme "$\mathrm{d}\,x\,y$", qu'*infixe*, comme "$x < y$" ou plus rarement *postfixe*, comme la notation usuelle pour la factorielle "$n\,!$". L'utilisateur peut choisir l'une de ces notations pour les nouveaux symboles qu'il introduit.

Par ailleurs en logique d'ordre supérieur on exprimera souvent comme une propriété ce qui s'exprime d'habitude en utilisant la notation ensembliste. ainsi on écrira "$\mathrm{Im}\,f\,y$" pour "$y \in \mathrm{Im}(f)$". Voici par exemple comment l'on dira que U est un ouvert : ouvert $U = \forall x(U\,x \to \exists e{>}0\,\forall y(d\,x\,y < e \to U\,y))$.

3.2 Parenthésage et priorité.

Seules les parenthèses usuelles `( )` peuvent être utilisées pour parenthéser les expressions (pas de crochets ni d'accolades). Pour éviter de trop nombreuses parenthèses, PhoX tient compte également de *priorités* entre symboles, autant que possible celles usuelles (souvent un peu plus que celles usuelles). Par exemple si les déclarations convenables ont été faites on peut écrire $x+y*z$ et le système comprendra $x+(y*z)$. Si les déclarations respectent les conventions usuelles[2], les symboles de fonction ou d'opération ont systématiquement priorité sur les symboles de prédicat qui eux-mêmes ont priorité sur les symboles de connecteurs ou de quantificateur, ce qui respecte l'usage. Ainsi dans la définition d'ouvert on a pu écrire $d\,x\,y < e \to U\,y$ plutôt que : $((d\,x\,y) < e) \to (U\,y)$. Il est toujours possible d'ajouter des parenthèses non nécessaires à la saisie, mais elles n'apparaîtront pas dans les réponses.

PhoX utilise aussi des règles d'*association* pour les opérateurs infixes : par exemple $A \wedge B \wedge C$ se lit $A \wedge (B \wedge C)$, et pour les entiers, $x + y + z$ se lit $(x + y) + z$). Ces règles d'association à droite (comme le symbole $\wedge$) ou à gauche (comme le symbole $+$) sont purement syntaxiques et ont peu à voir avec l'associativité. Ainsi il est important de noter que l'implication, qui n'est pas associative, associe à droite : $A \to B \to C$ signifie $A \to (B \to C)$, ce qui est assez usuel en logique, et rappelle la synonymie avec $(A \wedge B) \to C$, mais ne l'est pas dans le reste des mathématiques.

Ces priorités et règles d'association sont systématiquement utilisées pour l'affichage des réponses, ce qui dans quelques cas peut rendre la lecture un peu difficile[3]. Vous pouvez toujours utiliser la commande `print` (décrite dans la suite, voir l'index) pour vérifier que le parenthésage est bien celui que vous pensez.

3.3 Quantifications.

Certaines abréviations sont utilisées pour les suites de quantifications ($\forall$ ou $\exists$) de même nature, $\forall x, y\,A$ pour $\forall x \forall y\,A$, et pour les quantifications bornées $\forall x{\in}A\,B$ pour $\forall x(A\,x \to B)$. Pour un prédicat en notation infixe comme $<$ on aura également $\exists x{<}y\,B$ pour $\exists x(x < y \wedge B)$.

Ces abréviations quand elles sont usuelles, ce qui est le cas le plus fréquent, facilitent la lecture. Mais elles sont utilisées systématiquement à l'affichage, ce qui est parfois troublant – utilisez la commande `print` dans ce cas.

3.4 Récapitulatif des notations.

On donne dans le tableau ci-dessous le code d'entrée des symboles standards de PhoX, la façon dont ils sont affichés par l'interface Xemacs, et leur signification. On ajoute quelques exemples et abréviations usuelles.

[2] dans l'état, un utilisateur peut faire une déclaration qui ne les respecte pas.
[3] mais facilite l'implémentation !

`~`	$\neg$	négation
`&`	$\wedge$	conjonction
`or`	$\vee$	disjonction
`->`	$\rightarrow$	implication
`<->`	$\leftrightarrow$	équivalence
`/\`	$\forall$	quelque soit
`\/`	$\exists$	il existe ... tel que
`=`	$=$	égalité
`!=`	$\neq$	différent
`\`	λ	lambda abstraction
`False`	`False`	l'absurde (la proposition toujours fausse)
`True`	`True`	la tautologie (la proposition toujours vraie)
`/\x,y B x y`	`∀x,y B x y`	$\forall x \forall y\, B\, x\, y$
`\/x,y B x y`	`∃x,y B x y`	$\exists x \exists y\, B\, x\, y$
`/\x:A B x`	`∀x∈A B x`	$\forall x (A\, x \rightarrow B\, x)$
`\/x:A B x`	`∃x∈A B x`	$\exists x (A\, x \wedge B\, x)$
`/\x,y:A B x y`	`∀x,y∈A B x y`	$\forall x (A\, x \rightarrow \forall y (A\, y \rightarrow B\, x\, y))$
`\/x,y:A B x y`	`∃x,y∈A B x y`	$\exists x (A\, x \wedge \exists y (A\, y \wedge B\, x\, y))$

Pour mémoriser la notation des quantificateurs, pensez que le $\forall$ est une sorte de conjonction infinie, et le $\exists$ une sorte de disjonction infinie.

On voit qu'il existe assez peu de symboles prédéfinis. Ils sont presque tous énumérés ci-dessus (`A` et `B` n'en font pas partie !). Il faut noter que l'égalité est *polymorphe*. En particulier cela a un sens d'écrire l'égalité de deux énoncés (sorte `prop`) pour laquelle on utiliserait ordinairement le signe "$\equiv$".

4 Les commandes du système.

Toute commande du système doit se terminer par "`.` " suivi d'un espace ou retour à la ligne. On rappelle que le texte entre `(* *)` n'est pas évalué.

Dans la suite on donnera les exemples sous forme de suite d'entrées (commandes) et de sorties, ce qui ne correspond pas à ce que l'on obtient avec l'interface ProofGeneral. Pour les distinguer les entrées sont préfixée par `>PhoX>` ou `%PhoX%` (ce n'est pas le cas avec votre interface). On donne uniquement la syntaxe de saisie.

Ces commandes peuvent être essayées avec l'interface ProofGeneral. Pour cela écrire la commande et cliquez sur le bouton *Next*. Si la commande est acceptée, il est possible de revenir en arrière en cliquant sur le bouton *Undo*. Vous ne pouvez envoyer une commande que si elle est précédée de commandes déjà acceptées. En particulier il faut effacer les commandes décrites ci-dessous quand elles renvoient une erreur (ou les placer en commentaire), pour poursuivre. Il est parfois utile de réinitialiser le système grâce au bouton *Restart*. On peut ainsi changer de théorie, ou tout simplement sortir de situations – rares espérons le – où l'interface devient inutilisable, typiquement quand l'interface n'est plus synchrone avec le vérificateur de preuves, c'est à dire que certaines des commandes affichées comme acceptées par l'interface n'ont pas été envoyées au vérificateur de preuves ou vice-versa. En bref, dans une situation où l'interface ne semble pas se comporter correctement, n'hésitez pas à réinitialiser par *restart* !

On peut envoyer les commandes une à une grâce au bouton *Next* ou plusieurs à la fois grâce au bouton *Goto* (qui permet également de revenir en arrière pour un groupe de commandes).

Les raccourcis clavier « Ctrl j » («Enter» si vous avez de la chance) pour *Next* et « Ctrl c Ctrl u » pour *Undo* sont disponibles.

Il est parfois utile, pour pouvoir se relire ensuite, de rappeler la sortie dans le fichier d'édition : c'est ce que fait le raccourci clavier « Ctrl C Ctrl ; » qui place cette sortie en commentaire (pour que le fichier d'édition reste correct). Pour la lisibilité, vous avez souvent intérêt à faire du tri dans la sortie insérée en effaçant les lignes redondantes avec une précédente sortie.

4.1 Vérification de la syntaxe.

Il est possible de vérifier la correction syntaxique d'une *expression close* grâce à la commande `print`. On obtiendra par exemple les entrées et sorties suivantes :

```
>PhoX> print /\x x = x.
/\x x = x : prop
>PhoX> print /\x,y,z(x=y -> y=z -> x=z).
/\x,y,z (x = y -> y = z -> x = z) : prop
>PhoX> print /\x/\y/\z(x=y -> (y=z -> x=z)).
/\x,y,z (x = y -> y = z -> x = z) : prop
>PhoX> print (* transitivité *)  /\x,y,z(x=y -> y=z -> x=z).
/\x,y,z (x = y -> y = z -> x = z) : prop
>PhoX> print /\x/\y/\z x=y -> y=z -> x=z.
Syntax Error: Unbound identifier: "z"
>PhoX> print x=x.
Syntax Error: Unbound identifier: "x"
>PhoX> print \x(x=x).
\x (x = x) : ?a -> prop
```

Remarquez qu'à chaque fois que la commande est acceptée la sorte de l'expression est indiquée. Dans le dernier exemple `\x(x=x)`, les parenthèses sont indispensables. Cette expression est un prédicat à une place, le signe `?a` indiquant que ce prédicat peut s'utiliser pour n'importe quelle sorte.

Cette commande permet également d'afficher l'énoncé correspondant à un axiome ou théorème déjà démontré aussi bien par l'utilisateur qu'à l'initialisation du système, comme pour les exemples qui suivent :

```
>PhoX> print equal.transitive.
equal.transitive = /\x,y,z (x = y -> y = z -> x = z) : theorem
>PhoX> print excluded_middle.
excluded_middle = /\X (X or ~ X) : theorem
```

La commande `print` ne modifie pas bien-sûr l'état du système.

4.2 Déclarer une théorie.

4.2.1 Signature d'une théorie.

La signature d'une théorie regroupe les symboles de constantes de fonctions et de prédicats qui permettent d'énoncer la théorie. La notion de signature d'une théorie est un peu plus compliquée qu'en logique du premier ordre, puisqu'il peut y avoir plusieurs sortes d'objets. La seule sorte de base prédéfinie est `prop`, celle des propositions ou énoncés (formules closes).

Supposons que l'on veuille définir la théorie de groupe ordonné. On introduit tout d'abord une sorte pour les élément du groupe grâce à la commande `Sort` :

```
>PhoX> Sort g.
Sort g defined
```

On introduit ensuite la signature de la théorie proprement dite (c'est à dire les constantes non définies que la théorie manipule) : les symboles `+` pour la loi de groupe, `e` pour l'élément neutre, `-` pour l'opposé, `<` pour l'ordre strict. Pour cela on utilise la commande `Cst` :

```
>PhoX> Cst  rInfix[3.5] x "+" y : g -> g -> g.
$+ : g -> g -> g
>PhoX> Cst e : g.
```

```
e : g
>PhoX> Cst Prefix[2] "-" x : g -> g.
$- : g -> g
>PhoX> Cst Infix[5] x "<" y : g -> g -> prop.
$< : g -> g -> prop
```

On peut maintenant vérifier la syntaxe de quelques expressions :

```
>PhoX> print /\x x+e=x.
/\x x + e = x : prop
>PhoX> print - (((-e) + e) + (e + (-e))).
- (- e + e + (e + - e)) : g
>>> print e < e + e.
e < e + e : prop
```

On ne s'attarde pas sur la forme des déclarations, notez que + est en notation infixe et associe à gauche comme l'illustre la dernière expression. Notez également la priorité de - sur + qui ne correspond pas exactement à l'usage et qui est imposée par le système et non par l'utilisateur. Pour le moment vous pouvez considérer les priorités indiquées par les nombres 3.5 et 5 comme standards l'une pour une fonction l'autre pour une relation.

On peut étendre le vocabulaire en ajoutant quelques définitions, par exemple la soustraction et l'ordre large :

```
>PhoX> def lInfix[3.5] x "--" y = x + (-y).
$-- = \x,y (x + - y) : g -> g -> g
>PhoX> def Infix[5] x "<=" y = x < y or x = y.
$<= = \x,y (x < y or x = y) : g -> g -> prop
```

On peut maintenant donner les axiomes de la théorie des groupes en utilisant la commande `claim` :

```
>PhoX> claim associative /\x,y,z x + (y +z) = (x + y) +z.
associative = /\x,y,z x + (y + z) = x + y + z : theorem
>PhoX> claim neutre_droite /\x x + e = x.
neutre_droite = /\x x + e = x : theorem
>PhoX> claim oppose_droite /\x x + (-x) = e.
oppose_droite = /\x x + - x = e : theorem
```

Il faudrait maintenant ajouter les axiomes d'ordre et la compatibilité de l'ordre et de la loi de groupe.

4.2.2 Librairies.

Une autre façon de définir une théorie est de charger une librairie, c'est à dire essentiellement un fichier tel que ceux que vous pouvez écrire, qui le plus souvent ne contient pas seulement la signature d'une théorie, mais également des définitions, des théorèmes déjà démontrés, des extensions implicites des commandes de preuves etc. Pour charger la librairie des entiers naturels on utilise la commande `Import` [4] (il faut utiliser la commande `restart` ou redémarrer PhoX pour supprimer certaines définitions précédentes incompatibles avec la librairie des entiers, de toutes façons, ne saisissez pas ce qui suit si vous voulez continuer le tutoriel !) :

```
>PhoX> Import nat.
Loading nat
*** adding constants: N0 $S nat
*** adding axioms: S_inj.N N0_not_S.N
```

[4] Cette commande ne peut être utilisée directement pour charger un fichier que vous avez défini.

Le message de sortie ne dit rien des nouvelles définitions et des nouveaux théorèmes. Si vous avez entré les précédentes commandes, la commande `Import nat.` ne sera pas acceptée car elle définit `+` que vous avez déjà introduit comme constante. Si cela était utile, il faudrait utiliser d'autres commandes qui permettraient de renommer certains symboles.

On suppose dans la suite que la commande `Import nat` n'a pas été entrée.

4.3 Les preuves.

4.3.1 Entamer une preuve.

Tout d'abord on déclare l'énoncé que l'on veut prouver et on lui donne un nom, qui servira à y faire référence dans la suite si c'est utile. On utilise pour cela l'une quelconque des commandes `theorem` , `proposition` , `lemma` , `fact` , `corollary` , (les abréviations `theo`, `prop`,`lem`, `cor` sont aussi disponibles). Ces commandes ont toutes exactement le même effet : PhoX bascule en mode preuve, les commandes indiquées ci-dessus ne sont plus disponibles sauf la commande `print`. Voici un exemple (qui suppose que vous ayez déjà entré les commandes du paragraphe 4.2.1).

```
>PhoX> fact oppose_gauche /\x (- x) + x = e.
Here is the goal:
goal 1/1
   |- /\x - x + x = e
```

4.3.2 Représentation d'une preuve.

On va décomposer une preuve en un certain nombre d'étapes. On peut supposer qu'à chaque étape correspond un certain *état de preuve*. Un état de preuve est une liste de *buts*. Chaque but se décompose en deux parties :

- *ce que l'on a*, une liste d'énoncés qui représente les hypothèse courantes, les énoncés déjà démontrés, et des déclarations de variables (soit x quelconque ...), ces dernières n'étant pas affichées par PhoX. PhoX nomme tous ces énoncés, en utilisant suivant les cas : `H, H0, H1, ...` ou `G, G0, G1, ....`
- *ce que l'on veut*, un unique énoncé que nous souhaitons donc déduire de *ce que l'on a*, dans le cadre de la théorie développée, donc en utilisant éventuellement les axiomes qui ont été déclarés et les théorèmes déjà démontrés. PhoX préfixe un tel énoncé par `|-`.

Voici un exemple de but en PhoX :

```
G := - x + - - x = e
G0 := - x + x + e = - x + x
   |- - x + x = e
```

A chaque étape (sauf cas exceptionnel que l'on verra plus tard) on ne modifie qu'un but, en le remplaçant par un ou plusieurs buts dont il est à peu près évident qu'ils ont pour conséquence le but de départ. Quand le but est évidemment valide, par exemple *ce que l'on veut* est un énoncé présent dans *ce que l'on a* (c'est ce que l'on appelle un *axiome de la déduction*), la preuve de ce but est terminée, et on supprime le but de l'état de preuve. Quand il n'y a plus de buts dans l'état de preuve, la preuve est terminée.

Le fait d'analyser un but courant comme un énoncé à prouver sous hypothèses une liste de formules et non comme une simple formule peut sembler un peu compliqué à première vue, mais est en fait naturel et indispensable pour une représentation simple des preuves.

On va donner sous cette forme une preuve volontairement détaillée du fait que l'opposé à droite définit également un opposé à gauche. Pour le moment ne faites pas attention au détail des commandes. Dans ce cas l'état de preuve ne comporte toujours qu'un seul but. On ne donne ci-dessous que la partie la plus significative des sorties.

```
>PhoX> fact oppose_gauche /\x (-x) + x = e.
%PhoX% intro.
```

```
    |- - x + x = e
%PhoX% apply oppose_droite with (- x).
 G := - x + - - x = e
    |- - x + x = e
%PhoX% apply neutre_droite with ((- x) + x).
 G := - x + - - x = e
 G0 := - x + x + e = - x + x
    |- - x + x = e
%PhoX% rewrite  -l 1 -r G0.
 G := - x + - - x = e
 G0 := - x + x + e = - x + x
    |- - x + x + e = e
%PhoX% rmh G0.
%PhoX% rewrite  -r G.
 G := - x + - - x = e
    |- - x + x + (- x + - - x) = - x + - - x
%PhoX% rmh  G.
%PhoX% rewrite associative.
    |- - x + x + - x + - - x = - x + - - x
%PhoX% rewrite -p 1 -r associative.
    |- - x + (x + - x) + - - x = - x + - - x
%PhoX% rewrite -p 0 oppose_droite.
    |- - x + e + - - x = - x + - - x
%PhoX% rewrite neutre_droite.
    |- - x + - - x = - x + - - x
%PhoX% intro.
 0 goal created.
%PhoX% save.
 oppose_gauche = /\x - x + x = e : theorem
```

On voit sur cet exemple qu'une preuve est décrite en PhoX par une suite de commandes : les sorties sont utiles pour l'utilisateur afin de comprendre ce qu'il fait et donc de créer cette suite de commandes mais PhoX vérifie que cette suite de commandes constitue une preuve sans avoir besoin des sorties.

Ceci différencie les preuves en PhoX des preuves naturelles : on est souvent moins précis en preuve naturelle dans ce qui correspond aux commandes. Par exemple on dirait "par associativité" à la place de "`rewrite -p 1 -r associative.`" qui indique de plus d'utiliser l'associativité en la deuxième occurrence possible (`-p 1`) et dans le sens inverse de l'énoncé de départ (`-r`) mais l'on donnerait la sortie. Bien-sûr cette preuve est très particulière d'une part parce qu'elle est essentiellement égalitaire, d'autre part parce qu'elle est linéaire (il n'y a toujours qu'un seul but à prouver).

Notons qu'il est possible de faire vérifier une preuve plus courte, d'une part en utilisant les automatismes de PhoX (voir dans le paragraphe 4.4.2 les commandes `auto` et `trivial`), d'autre part en utilisant l'« extensibilité » des commandes de PhoX comme ici `intro`, mais ce dernier point ne rentre pas dans le cadre de cette introduction.

4.3.3 Un autre exemple, le paradoxe de Cantor.

On présente maintenant un exemple un peu plus riche logiquement (et sans raisonnement égalitaire). C'est une version "positive" du paradoxe de Cantor en théorie des ensembles : « il est faux que toute propriété soit équivalente à l'appartenance à un ensemble». Cette proposition se démontre sans aucun axiome de la théorie des ensembles. Là aussi on présente la suite des commandes et la partie la plus significative des réponses. On donne en parallèle une preuve rédigée en suivant la preuve PhoX (et qui ne cherche donc pas à être « élégante »).

Comme pour les groupes on définit la sorte des ensembles (les objets que l'on manipule), et la relation d'appartenance entre les ensembles (une relation binaire) :

```
>PhoX> Sort set.
Sort set defined
>PhoX> Cst (* appartenance *) Infix[5] x "in" y : set -> set -> prop.
$in : set -> set -> prop
```

La propriété annoncée s'énonce :

```
>PhoX> theorem cantor ~ /\P\/x/\y(y in x <-> P y).
```

Supposons que toute propriété soit équivalente à l'appartenance à un ensemble (soit `H` cette hypothèse) :

```
%PhoX% intro.
H := /\P \/x /\y (y in x <-> P y)
   |- False
```

c'est le cas en particulier de la propriété "ne pas appartenir à soi-même" qui définit un ensemble que l'on appelle `x` :

```
%PhoX% elim H with \y(~ y in y).
H0 := /\y (y in x <-> ~ y in y)
   |- False
```

On obtient que les éléments de `x` sont exactement les éléments qui n'appartiennent pas à eux-mêmes.

On détaille la suite, mais celle-ci est "triviale" pour PhoX (commande `auto`).
On a donc si `x` appartient à lui-même alors il n'appartient pas à lui même (`H1`), et si `x` n'appartient pas à lui-même alors il appartient à lui-même (`H2`).

```
%PhoX% apply H0 with x.
G := x in x <-> ~ x in x
%PhoX% left G.
H1 := x in x -> ~ x in x
H2 := ~ x in x -> x in x
```

Distinguons deux cas suivant que `x` appartient ou non à `x`.

```
%PhoX% elim excluded_middle with x in x.
2 goals created.
goal 1/2
...
H3 := x in x
   |- False

goal 2/2
...
H3 := ~ x in x
   |- False
```

Si `x` appartient à lui-même, alors il n'appartient pas à lui-même d'après `H1`, ce qui est absurde.

```
H3 := x in x
%PhoX% elim H1 with H3 and H3.
0 goal created.
```

Si x n'appartient pas à lui-même, alors il appartient à lui-même d'après H2, ce qui est absurde également, et la preuve est terminée.

```
H3 := ~ x in x
%PhoX% apply H2 with H3.
G := x in x
%PhoX% elim H3 with G.
0 goal created.
End of goals.
%PhoX% save.
cantor = ~ /\P \/x /\y (y in x <-> P y) : theorem
```

On voit que si la preuve en PhoX suit le déroulement d'une preuve usuelle, elle utilise des commandes qui ne font pas appel directement au vocabulaire mathématique. Cela peut sembler curieux d'utiliser la même commande dans des situations très différentes. Par exemple `intro`, est utilisée, entre autres pour la réflexivité de l'égalité (à la fin de l'exemple traité au paragraphe 4.3.2) et pour ce que certains appelleraient un raisonnement par l'absurde, au début de l'exemple que l'on vient de traiter[5].

Ces commandes font en fait référence à la structure logique des formules, ce qui permet entre autres d'en limiter le nombre. Ainsi les deux usages de `intro` ci-dessus s'expliquent ainsi. la commande `intro` essaye d'utiliser une façon "standard" de prouver un énoncé ("la" façon standard si cela a un sens, mais ce n'est pas toujours le cas). Par exemple, si cet énoncé est une égalité, on considére que la façon standard est la réflexivité de l'égalité ! Si cet énoncé est une négation on a considéré que la façon standard est de supposer l'énoncé que l'on nie pour aboutir à une contradiction. On reprend et détaille un peu ceci dans le paragraphe suivant.

4.4 Les commandes de preuves.

4.4.1 Quelques principes.

Une commande est composé d'un mot clef comme `rewrite` ou `intro` ci-dessus, suivi ou non, selon les commandes, d'un argument. Comme pour les commandes globales elle peut s'écrire sur plusieurs lignes et doit se terminer par un point "`.`" suivi d'un espace ou d'un retour à la ligne.

Dans le premier exemple ci-dessus, `rewrite associative` a un argument, `intro` n'en a pas. Cet argument peut être une expression du langage ou, comme dans ce cas un nom y faisant référence. Ce nom peut être local : référencé dans le but courant comme `G` et `G0` dans le premier exemple ci-dessus ou global comme `associative` ci-dessus.

Il peut y avoir des arguments optionnels qui modifie le comportement de la commande. Certains d'entre eux sont introduit par un mot qui commence par un tiret "`-`" (voir `-p 0`, `-r` ci-dessus). C'est une tradition héritée du système d'exploitation Unix, et ce tiret n'a rien à voir avec la soustraction ! Cette tradition n'est d'ailleurs pas entièrement respectée par PhoX (voir le paragraphe suivant).

Le comportement des commandes dépend fortement du but courant, par exemple du "constructeur principal" de la formule conclusion (ce que l'on veut). Les noms des commandes sont souvent inspirés du vocabulaire de la théorie de la démonstration. Ainsi quand on parle d'introduction ce mot fait référence à une description de la preuve dans l'ordre inverse de celui que vous pratiquez avec PhoX.

On décrit maintenant les principales commandes qui permettent de développer une preuve.

4.4.2 Principales commandes de preuves.

`axiom` permet de terminer un but lorsque sa conclusion est une des hypothèses. Exemple : dans la situation suivante, la commande `axiom H4` termine le but courant :
`goal 1/2`

[5] Il est important qu'il s'agisse d'une négation, ce n'est pas un "vrai" raisonnement par l'absurde en théorie de la démonstration.

`H :=` continue1 f
`H4 :=` $a_0 > R0$
`|-` $a_0 > R0$

Notez qu'ici "axiome" est utilisé seulement dans le sens "axiome de la déduction", ce qui n'est pas le sens le plus usuel du mot "axiome" en mathématique, voir pour cela la commande `claim` (consultez l'index).

On peut demander à PhoX de détecter automatiquement les axiomes (de la déduction) grâce à la commande `flag auto_lvl 1`.

`intro` et `intros` correspondent aux *règles d'introduction*. Leur action dépend uniquement de la conclusion du but à prouver (on n'utilise pas d'hypothèse). Par règle d'introduction on entend une façon "standard" (souvent la bonne, mais pas forcément suivant le contexte) de prouver un type d'énoncé. Par exemple la façon standard de prouver $A \wedge B$ est ... de prouver A et de prouver B. La façon standard de prouver $A \to B$ est de supposer A pour prouver B. La façon standard de prouver $\forall x\, A\, x$ est de prouver $A\, x$ pour une variable x quelconque, c'est à dire une variable qui n'est pas encore déclarée (ceci peut obliger à choisir un nouveau nom de variable).

La commande `intro` permet de ne faire qu'une seule introduction ou, avec un paramètre qui est un nombre entier, de préciser le nombre d'introductions. La commande `intros` fait toutes les introductions possibles (en fait c'est un peu plus compliqué, cf. commande `lefts`). Voici un premier exemple :

```
goal 1/1
  |- ∀h, g(inj h ∧ inj g ∧ ∀x(h x = x ∨ g x = x) → ∀x(h(gx)) = (g(hx))).
%PhoX% intros.
goal 1/1
H :=  inj h ∧ inj g ∧ ∀x(h x = x ∨ g x = x) .
   |- h(gx) = g(hx) .
```

Un problème se pose lorsqu'il y a un choix possible pour une règle d'introduction. Par exemple, pour montrer $A \vee B$, on peut soit montrer A, soit montrer B. On peut alors préciser le nom de la règle à appliquer en tapant `intro l` (pour « left ») pour montrer A ou `intro r` (pour « right ») pour montrer B.

`apply` et `elim` correspondent aux *règles d'élimination*. Par règle d'élimination on entend une façon standard d'utiliser un énoncé. Par exemple la façon standard d'utiliser $A \vee B$ est de raisonner par cas suivant que l'on suppose A ou que l'on suppose B. La façon standard d'utiliser $A \to B$ et de supposer ou d'avoir démontré A pour en déduire B. La façon standard d'utiliser $\forall x\, A\, x$ est d'en déduire $A\, e$ où e représente un élément du domaine (ce peut être une expression "compliquée").

Les commandes `apply` et `elim` sont les commandes les plus complexes à utiliser. Dans la pratique, comme pour les règles d'introduction, on désire pouvoir appliquer plusieurs règles d'élimination en une seule fois en disant des choses du genre « de $\forall x(A(x) \to B(x))$ et de $A(a)$ je déduis $B(a)$ ». Dans cet exemple, on a indiqué implicitement que l'on désirait donner la valeur a à x.

La commande `apply` permet de faire cela en tapant `apply` $\forall x(A(x) \to B(x))$ `with` $A(a)$. En général, on tape d'ailleurs plutôt `apply H1 with H2` où `H1` et `H2` sont le nom des hypothèses concernées (en fait la première commande est plus lisible, mais plus longue à écrire). On peut aussi donner plusieurs indications à `apply` en les séparant par le mot clef `and`.

La commande `elim` est très similaire à `apply`, mais elle doit conduire immédiatement à démontrer la conclusion du but courant, alors que la commande `apply` ajoute ce qu'elle produit parmi les hypothèses. Il s'agit d'une manière subtile et souvent utile d'indiquer la valeur des variables. Voici deux exemples utilisant ces commandes :

```
goal 1/1
H5 := ∀y0(d x y0 < a' → d(fx)(fy0) < a)
H6 := d x y < a'
```

|- $d(fx)(fy) < a$

`%PhoX% elim H5.`

```
goal 1/1
H5 := ∀y0(d x y0 < a' → d(fx)(fy0) < a)
H6 := d x y < a'
   |- d x y < a'
```

On aurait pu taper `elim H5 with H6` pour éviter la commande `axiom H6`.

```
goal 1/1
H := continue1f
H1 := U(fx)
H0 := ∀x0 ∈ U ∃a>R0 ∀y(d x0 y < a → Uy)
   |- ∃a>R0 ∀y(d x y < a → inverse f U y)
```

`%PhoX% apply H0 with H1.`

```
goal 1/1
H := continue1f
H1 := U(fx)
H0 := ∀x0 ∈ U ∃a>R0 ∀y(d x0 y < a → Uy)
G := ∃a>R0 ∀y(d (fx) y < a → Uy)
   |- ∃a>R0 ∀y(d x y < a → inverse f U y)
```

`prove` et `use` correspondent à l'introduction d'un lemme. `prove A` indique que l'on veut prouver `A`, que l'on pourra ensuite utiliser. La commande `use A` inverse juste l'ordre des buts :

```
goal 1/1
H := bijective(f ∘ f)
   |- bijectivef
```

`%PhoX% prove` injective f.

```
goal 1/2
H := bijective(f ∘ f)
   |- injectivef
goal 2/2
H := bijective(f ∘ f)
H0 := injectivef
   |- bijectivef
```

`left` et `lefts` correspondent à un type de règles dont on n'a pas encore parlé : les règles d'introduction pour les hypothèses (ces règles sont démontrables à partir des autres, mais sont indispensables en pratique). Il s'agit par exemple de remplacer une hypothèse de la forme $A \wedge B$ par deux hypothèses : A et B. La version `lefts` permet d'enchaîner plusieurs de ces règles en indiquant les connecteurs auxquels on veut l'appliquer (on peut utiliser la même syntaxe pour contrôler le comportement de `intros`) :

```
goal 1/1
H2 := ∀z≤e' z < e
G := ∃a>R0 ∀y(d x y ≤ a → d(fx)(fy) ≤ e')
   |- ∃a>R0 ∀y(d x y < a → d(fx)(fy) < e')
```

`%PhoX% lefts G $∃ $∧.`

```
goal 1/1
H2 := ∀z≤e' z < e
H3 := a > R0
H4 := ∀y(d x y ≤ a → d(fx)(fy) ≤ e')
   |- ∃a>R0 ∀y(d x y < a → d(fx)(fy) < e')
```

`by_absurd` permet de faire un raisonnement par l'absurde en ajoutant la négation de la conclusion parmi les hypothèses :

```
goal 1/1
H := ¬∀x(X x)
```

|- $\exists x \neg (X\ x)$

```
%PhoX% by_absurd.
```

```
goal 1/1
```
H := $\neg \forall x (X\ x)$
H := $\neg \exists x \neg (X\ x)$
|- $\exists x \neg (X\ x)$

Plutôt que d'utiliser le raisonnement par l'absurde, il est parfois commode d'utiliser le tiers-exclu : la commande `elim excluded_middle with A`, introduira deux buts, un avec `A` dans les hypothèses, l'autre avec sa négation.

On peut encore utiliser le raisonnement par contraposée et les lois de De Morgan et (voir commande `rewrite`).

`unfold` et `unfold_hyp` permettent de remplacer un symbole par sa définition. La première agit sur la conclusion, la seconde sur une hypothèse :

```
goal 1/1
```
H := continue1 f
H0 := ouvert U
|- ouvert(inverse $f\ U$)

```
%PhoX% unfold ouvert. unfold_hyp H0 ouvert.
```

```
goal 1/1
```
H := continue1 f
H0 := $\forall x \in U\ \exists a{>}R0\ \forall y (d\ x\ y < a \rightarrow U y)$
|- $\forall x \in$ (inverse $f\ U$) $\exists a{>}R0\ \forall y (d\ x\ y < a \rightarrow$ inverse $f\ U y)$

`auto` et `trivial` indiquent à PhoX d'essayer de résoudre seul le but courant. Il ne faut pas en attendre de miracle, et l'on doit souvent interrompre la recherche (en tapant « Ctrl c » deux fois). Ces commandes n'utilisent que ce qui est déclaré dans le but courant pour chercher la preuve. Elles n'utilisent pas les axiomes (introduits par `claim`) ou les théorèmes déjà démontrés. On peut leur demander d'utiliser ceux-ci en utilisant l'argument optionnel + suivi du ou des noms des théorèmes ou axiomes à utiliser, comme indiqué dans la preuve suivante où l'on donne une version plus rapide de l'exemple du paragraphe 4.3.2 :

```
>PhoX> fact oppose_gauche /\x (-x) + x = e.
%PhoX% intro.
    |- - x + x = e
%PhoX% apply oppose_droite with (- x).
 G := - x + - - x = e
    |- - x + x = e
%PhoX% trivial + associative oppose_droite neutre_droite.
 0 goal created.
%PhoX% save.
 oppose_gauche = /\x - x + x = e : theorem
```

`instance` est utile car certaines règles (comme l'introduction d'un $\exists$) nécessitent de trouver la bonne valeur pour une variable. PhoX ne demande pas de trouver cette valeur immédiatement. Dans ce cas, le système introduit des *variables existentielles* dont le nom commence par un point d'interrogation. La commande `instance` permet de choisir la valeur de ce type de variables.

```
goal 1/1
```
H5 := $\forall y_0 (d\ ?1\ y0 < a_0 \rightarrow d(f\ ?1)(f y_0) < a)$
H6 := $d\ x\ y < a_0$
|- $d(fx)(fy) < a$

```
%PhoX% instance ?1 x.
```

```
goal 1/1
```
H5 := $\forall y_0 (d\ x\ y0 < a_0 \rightarrow d(fx)(f y_0) < a)$

H6 := $d\,x\,y < a_0$
|- $d(fx)(fy) < a$

`select` permet de changer de but courant. C'est surtout utile lorsque plusieurs buts contiennent une variable existentielle et que l'on veut commencer par le but qui impose vraiment la valeur de cette variable.

`rewrite`, `rewrite_hyp` correspondent au raisonnement équationnel. Ils prennent en argument une équation ou un nom désignant celle-ci (voir l'exemple du paragraphe 4.3.2) ou un nom désignant une liste d'équations (voir l'exemple qui suit).

La commande `rewrite eq` permet de transformer la conclusion du but courant en utilisant l'égalité `eq`, la commande `rewrite_hyp H eq` fait de même pour l'hypothèse `H`. Chaque équation est lue de la gauche vers la droite, et donc cela n'est pas la même chose de passer en argument une équation ou son écriture symétrique. L'argument optionnel `-r` permet d'utiliser la symétrique de l'équation passée en argument. L'équation est utilisée tant qu'il est possible de le faire ce qui peut ne pas terminer : utilisez « Ctrl c » deux fois pour vous sortir de telles situations. Il existe des options qui vous permettent de restreindre cette utilisation et de la préciser (voir l'usage de `-p` dans l'exemple du paragraphe 4.3.2).

Il ne faut pas oublier que PhoX peut traiter l'équivalence logique ("≡") comme une égalité. Ainsi l'exemple ci-dessous correspond aux lois de De Morgan pour le calcul propositionnel et les quantificateurs. `demorgan` est ici le nom d'une liste de théorèmes équationnels correspondant chacun à une loi de De Morgan précise.

```
goal 1/1
H := Adh (Union A B) x
H0 := ¬(∀e>R0 ∃y ∈ A (d x y) < e ∨ ∀e>R0 ∃y ∈ B (d x y) < e)
   |- False
%PhoX% rewrite_hyp H0 demorgan.
goal 1/1
H := Adh(Union A B)x
H0 := ∃e>R0 ∀y ∈ A ¬(d x y < e) ∧ ∃e>R0 ∀y ∈ B ¬(d x y < e)
   |- False
```

Pour le raisonnement par contraposée, la commande est un peu cryptique : il faut écrire `rewrite -p 0 -r contrapose` (une seule réécriture en la première position possible) pour raisonner par contraposée sur une implication ou une équivalence.

`from` permet de faire du raisonnement égalitaire de façon automatique. La commande `from A.` indique au système de trouver seul les étapes de raisonnement équationnel qui permettent de transformer la conclusion du but courant en `A`. Cela correspond plus à la manière habituelle d'écrire des preuves (on indique les étapes du raisonnement équationnel sans préciser les transformations effectuées).

Références

[1] *The XEmacs editor*. `www.xemacs.org`.

[2] Christophe Raffalli. *PhoX*. `www.lama.univ-savoie.fr/~RAFFALLI/phox.html`.

[3] Kleymann Thomas, David Aspinall, and al. *Proof General*. `zermelo.dcs.ed.ac.uk/~proofgen`.

Index