Le théorème de Schroeder-Bernstein dans le Calcul des Constructions

Hugo Herbelin

Mémoire de DEA d'Informatique

sous la direction de Thierry Coquand
et avec l'assistance de Loic Colson
et de Christine Paulin

Laboratoire d'Informatique de l'Ecole Normale Supérieure

INRIA-Rocquencourt

Septembre 1988

Résumé

Notre but est de développer une démonstration du théorème de Schroeder-Bernstein dans le Calcul des Constructions. Partis d'une preuve établie dans la théorie axiomatique des ensembles , nous l'avons transcrite dans une théorie typée des ensembles, plus en adéquation avec le Calcul des Constructions.

Dans le même codage de la théorie des ensembles, nous avons par ailleurs réalisé un codage de la démonstration de l'équipollence entre 2^A et $\mathcal{P}(A)$.

Ces preuves ont été développées interactivement et automatiquement vérifiées dans la machine constructive de Thierry Coquand, Gérard Huet et Christine Paulin.

Table des matières

Chapitre 1

La machine constructive

1.1 Les constructions

1.1.1 Le calcul des constructions et son implémentation

La machine constructive est une implémentation du calcul des constructions permettant de coder du langage mathématique.

Le calcul des constructions, sur lequel se base la machine, est un λ-calcul typé d'ordre supérieur. Il permet de représenter et de vérifier des preuves écrites dans un formalisme de déduction naturelle (ceci par l'isomorphisme de Curry-Howard). Il est ainsi spécialement porté à représenter les raisonnements intuitionnistes même si la logique classique peut y être codée.

Dans ce λ-calcul, les types des termes sont eux-mêmes des termes.

La quantification sur les propositions est distinguée de l'abstraction. Tandis que l'abstraction (notée [M:T]N) a le sens d'abstraire une variable d'une proposition pour en faire une fonction de cette variable, la quantification (noté (M:T)N) abstrait une variable d'une proposition pour généraliser la proposition aux variables du type de la variable abstraite.

On adoptera l'abréviation `A->B` lorsqu'une variable `(p:A)` quantifiant une construction `B` n'intervient pas dans `B`.

Définissons, par exemple, le et intuitionniste :

```
Definition and [A,B:Prop](C:Prop)(A->B->C)->C.
```

Ceci dit que `and` est une fonction des variables `A` et `B` et que (and A B) signifie que pour toute proposition `C` et pour toute preuve de `A->B->C`, on a une preuve de `C`.

Le calcul possède deux types de base : le type des propositions `Prop`, et une hiérarchie de types nommés `Type n` (n entier non nul) tels que, `Type 1` est le type de `Prop` et `Type (n+1)` est le type de `Type n` (n entier non nul).

Dans le calcul des constructions, on peut définir ce qu'est un théorème comme étant une proposition (de type `Prop`) qui poss'ede une preuve, c'est-à-dire un terme bien typé dont le théorème est le type.

La machine constructive intervient pour vérifier le bon typage des termes et donc la vérification des théorèmes. Par ailleurs, elle gère un environnement de démonstration (théorèmes, hypothèses, définitions,...). Le langage qui permet de construire les termes et l'environnement mathématique de la machine s'appelle le vernaculaire.

On pourra se référer à [1] pour avoir une présentation détaillée des règles d'inférences du calcul, de l'implémentation CAML de la machine constructive et des mots-clés du vernaculaire.

1.1.2 Le codage de la logique

Dans le calcul des constructions, nous pouvons définir des connecteurs logiques et des quantificateurs intuitionnistes par :

```
Definition and [A,B:Prop](C:Prop)(A->B->C)->C.
Definition or  [A,B:Prop](C:Prop)(A->C)->(B->C)->C.
Definition pi  [A:Prop][P:A->Prop](x:A)(P x).
Definition sig [A:Prop][P:A->Prop](B:Prop)((x:A)(P x)->B)->B.
Definition sig2
    [A:Prop][P,Q:A->Prop](B:Prop)((x:A)(P x)->(Q x)->B)->B.
```

Pour le développement du théorème de Schroeder-Bernstein où nous manipulons des objets de type `Type`, nous avons défini aussi un `sigt` (il existe), un `sig2t` (il existe vérifiant deux propriétés) :

```
Definition sigt [A:Type][P:A->Prop](B:Prop)((x:A)(P x)->B)->B.
Definition sig2t
    [A:Type][P,Q:A->Prop](B:Prop)((x:A)(P x)->(Q x)->B)->B.
```

Dans la pratique, nous n'utiliserons pas le `pi` puisque sa forme dépliée est tout aussi explicite et plus maniable. C'est souvent aussi que l'on gardera le `and` ou le `sigt` sous forme dépliée pour des raisons de commodité d'utilisation.

Les théorèmes d'introduction de `and`, `or`, `sigt` et `sig2t` sont respectivement `pair`, `inl` et `inr`, `existt` et `exist2t`.

Les théorèmes d'élimination de `and` et `or` sont respectivement `fst` et `snd` et `case`. En synthèse, nous n'utiliserons pas `fst` et `snd`, leur préférant le théorème : `Theorem and_elim A/\B -> (C:Prop)(A->B->C)->C` qui à l'avantage de contenir en lui et l'information du `fst` et celle du `snd`. De même pour `sigt` (et `sig2t`) : `Theorem sigt_elim (sigt U P) -> (C:Prop)((y:U)(P y)->C)->C`

1.2 La synthèse

1.2.1 Aperçu de la synthèse

C'est un outil écrit en CAML qu permet de prouver des théorèmes de façon moins formelle que la preuve écrite en λ-terme dans la partie constructive. En synthèse, on raisonne sur des buts à démontrer sous certaines hypothèses et par l'usage de tactiques, on ramène un but à démontrer sous un contexte d'hypothèse à d'autres buts suffisant sous d'autres contextes d'hypothèses. Les règles de passage d'un but à l'autre sont les règles de typage du calcul des constructions prises dans le sens inverse.

Exemple :

démontrer `(A,B,C:Prop)(A->B)->(B->C)->(A->C)`.

On indique ce que l'on veut démontrer par `goal` :

```
#goal <<(A,B,C:Prop)(A->B)->(B->C)->(A->C)>>;;
<<......(A:Prop)(B:Prop)(C:Prop)(A->B)->(B->C)->A->C>> : message
```

On isole la conclusion à démontrer (résolution Hornale) :

```
#Intros();;
<<1 subgoal
  1......C
  H1 : A
  H0 : B->C
  H : A->B
  C : Prop
  B : Prop
  A : Prop>>
 : message
```

On résoud le but avec l'hypothèse `H0` puis avec l'hypothèse `H` :

```
#by (resolve "H0" THEN resolve "H");;
<<1 subgoal
  1......A
  H1 : A
  H0 : B->C
  H : A->B
  C : Prop
  B : Prop
  A : Prop>>
 : message
```

On montre que le but est dans les hypothèses :

```
#Assumption();;
A est dans les hypotheses
<<goal proved>> : message
```

et le théorème est prouvé, on l'enregistre :

```
#save_thm"o";;
() : void
```

1.2.2 Les tactiques de la synthèse

Des tactiques indispensables, correspondant aux règles de construction des termes bien typés du calcul des constructions :

- `Intro` : correspond à la démarche inverse de l'abstraction : on remet dans le contexte la dernière hypothése abstraite du terme à prouver.

  Le sens intuitif de `Intro` est de poser comme acquis la première condition de l'énoncé que j'ai á démontrer.

- `Resolve` *nom* correspond à la démarche inverse de l'application : cela signifie que le but à prouver provient de l'application du théorème *nom* à un terme qu'il me reste à prouver.

  Au niveau outil de démonstration, cela signifie que pour prouver le but, il faut utiliser le théorème (ou l'hypothèse, ou l'axiome, ...) *nom*.

- `Assumption` correspond à la règle inverse d'introduction d'une variable.

  C'est dire que ce que j'ai à démontrer est évident car c'est dans les hypothèses.

  Des tactiques indispensables, substituant une constante par le terme qu'elle représente.

- `Unfold` *nom* correspond au niveau outil de démonstration à la démarche : je reviens à la définition de *nom*.

- `Unfold_hyp` *hyp nom* : idem pour la constante *nom* dans l'hypothèse *hyp*.

Des tactiques non fondamentales, mais pratiques :

- `Cut` *P* correspond à la démarche : j'admet P que je démontrerai plus tard. C'est poser un jalon dans le parcours qui relie le but aux hypothèses.

- `Exact` *c* donne explicitement le λ-terme prouvant le but. En général peu agréable d'emploi car nécessite de penser au niveau λ-calcul alors qu'il est préférable de penser au niveau résolution.

– `Explicit` *c* fait la même chose que `Resolve` mais l'expression avec laquelle on résoud est le terme `c`.

Et des tactiques à usage plus particulier, qui sont des applications des précédentes :

– `And_intro` ramène la démonstration d'un but de la forme `A/\B` à la démonstration de `A` et à celle de `B`

– `or_intro1` (resp. `or_intro2`) ramène la démonstration d'un but de la forme `A\/B` à la démonstration de `A` (resp de `B`).

– `sigt_intro` *y* ramène la démonstration d'un but de la forme `(sigt U P)` à la démonstration de `(P y)` où *y* est censé être une variable témoin du `(sigt U P)`.

– `And_elim` *hyp* élimine le `/\` dans l'hypothèse *hyp* de la forme `A/\B` en créant les hypothèses prouvant `A`, et `B`.

– `or_elim` *hyp* élimine le `\/` dans l'hypothèse *hyp* de la forme `A\/B` en créant deux sous-buts, l'un avec l'hypothèse `A`, l'autre avec l'hypothèse `B`

– `sigt_elim` *hyp* élimine le `sigt` dans l'hypothèse *hyp* de la forme `(sigt U P)` en rajoutant dans les hypothèses une variable `y` qui vérifie `(P y)`

Ajoutons à cela des constructeurs sur les tactiques :
t `THEN` *t'* applique la tactique *t'* aux sous-buts générés par la tactique *t*.
`REPEAT` *t* répète la tactique *t* tant que c'est possible.
t `ORELSE` *t'* essaie la tactique *t'* si *t* ne réussit pas à s'appliquer.

Chapitre 2

Développement du théorème de Schroeder-Bernstein dans une théorie typée des ensembles

2.1 Le choix du codage : de la tentative de codage de la théorie axiomatique des ensembles dans la machine constructive à un codage dans une théorie typée des ensembles

Notre but est de développer un codage de la théorie des ensembles dans lequel on puisse démontrer agréablemant le théorème de Schroeder-Bernstein . Ce théorème est un théorème classique (utilisant l'axiome du tiers-exclu) et nous nous baserons sur la démonstration de Fraenkel de la théorie axiomatique des ensembles , rapportée par Suppes [2], pour établir ce théorème.

La théorie axiomatique des ensembles se prête mal à un codage dans une machine constructive. Il est certes toujours possible de la coder mais cela accumule les axiomes et par conséquent les risques d'incohérence de la théorie. D'autre part, on est obligé d'adapter un peu la manière de définir les nouveaux concepts, si on ne veut pas rajouter en plus dans la théorie les régles métamathématiques, comme par exemple, pour poser la définition de l'intersection, il faut préalablement démontrer que C tel que $x \in C \Leftrightarrow \; x \in A \wedge \; x \in B$ existe et est unique. Voir en annexe une ébauche de codage de la théorie axiomatique des ensembles (fichier "ast.v").

La philosophie de la théorie axiomatique des ensembles est d'introduire des symboles abstraits (en particulier l'univers fondamental contenant les ensembles, la relation d'appartenance, l'ensemble vide) puis des lois régissant l'utilisation de ces symboles.

Notre démarche, ici, sera autre. Nous *construirons* tant bien que mal les différentes notions de la théorie des ensembles dont nous avons besoin, à partir d'objets déjà définis.

Tout d'abord on définira un ensemble par sa fonction caractéristique, c'est-à-dire la fonction qui dit pour tout élément s'il est dans l'ensemble ou pas. Plus exactement, puisque nous raisonnons en termes intuitionnistes, nous définirons un ensemble par la fonction qui à un élément associe une proposition représentant l'appartenance de cet élément à l'ensemble. Cette proposition peut être selon les cas prouvable, contradictoire ou non décidable.

Si $\mathcal{U}$ est un univers d'objets, on définira donc un ensemble sur ces objets comme un objet de type `U -> Prop` où `U` est le type des objets de $\mathcal{U}$. Un ensemble sera donc typé différemment de ses éléments ce qui conduit à une théorie typée des ensembles. Passer d'un univers d'objets à l'univers des ensembles sur ces objets fait passer à un type d'un niveau plus complexe, on ne pourra pas avoir de "boucle d'appartenance" dans un ensemble, ce qui prévient de manière radicale tous les problèmes du genre le paradoxe de Russell et rend inutile l'axiome de régularité (= axiome de fondation) de la théorie axiomatique des ensembles .

Il va de soit aussi que, ne pouvant définir un ensemble contenant des objets de types différents, comme l'ensemble vide et l'ensemble contenant cet ensemble vide, on ne pourra pas définir les ordinaux dans ce modéle.

2.2 Description du codage choisi

2.2.1 Le codage des ensembles

Le constructeur `Set`

Soit $\mathcal{U}$ un univers d'objets dont les objets sont de type `U`. Un ensemble sur cet univers aura pour type `U -> Prop`. De même, un ensemble sur les ensembles d'objets de type `U` sera de type `(U -> Prop) -> Prop`.

De façon à généraliser le passage des objets de type `U` aux ensembles sur ces objets par un constructeur `Set`, on voudrait que `U` et `U -> Prop` soient de même type et plus généralement que tous les types des objets ensemblistes soient d'un même type.

Le fait que la machine constructive type `U -> Prop` avec `Type`, quelque soit le type de U, nous pousse à définir les objets que nous considérons (qu'ils soient individuels ou ensembles) comme des objets d'univers de type `Type`.

On définit ainsi le constructeur `Set` qui, a un type `U` de type `Type` associe le type des ensembles sur les objets de type `U` :

```
Definition Set [U:Type] U -> Prop.
```

Ainsi, si nous avons des objets de type `U` alors `A:(Set U)` signifie que `A` est un ensemble sur les objets de type `U`.

On définira alors la notion d'appartenance par :

```
Definition appartient [U:Type][A:(Set U)][x:U](A x).
```

Dans la pratique, nous n'utiliserons pas cette définition de l'appartenance et garderons la notation plus simple de `(A x)` pour dire $x \in A$.

Nous avons donc une définition d'un ensemble qui permet de travailler sur des objets et sur des ensembles sur ces objets, peut importe que ces objets soient eux-même des individuels, des ensembles ou, comme nous le verons plus tard, des relations (on verra une application de cela pour le théorème d'équipollence entre 2^A et $\mathcal{P}(A)$).

Le revers de médaille est que dans toutes les définitions et théorèmes , il faudra indiquer le type des objets sur lesquels on travaille. Par exemples on dira `(inter U A B)` pour désigner l'intersection des ensembles `A` et `B` qui opèrent sur des ensembles de type `U`. En fait `U` est redondant dans `(inter U A B)`. La connaissance de `A` entraine celle de `U` et une modification de la machine constructive devrait permettre de ne pas avoir à signaler ce `U` redondant. Une version précédente de la machine constructive évitait d'introduire les hypothèses redondantes. Ce problème n'est donc pas un problème majeur, resolvable facilement (cf [3]).

Comme on l'a vu, cette modélisation des ensembles évite les paradoxes à la Russell et ne rend pas nécessaire l'axiome de régularité dont la théorie axiomatique des ensembles a besoin pour éviter les boucles d'appartenance (du genre $A \in B,\ B \in C,\ C \in A$).

Axiome de régularité : $\forall A,\ \exists x \in A\ tq\ \forall y,\ (y \in x \Rightarrow\ \neg(y \in A))$

On a donc affaire à une modélisation hiérarchisée localement minorée des ensembles. Hiérarchisée car il y a un ordre entre les différents objets ensemblistes, localement minorée car il n'y a de plus petit univers que localement à un énoncé, ce plus petit univers pouvant être substitué par un quelconque autre univers dans un énoncé plus large (cf le théorème d'équipollence entre 2^A et $\mathcal{P}(A)$). Il n'y a pas à proprement parler d'univers des individuels.

Ce que nous voulons dire, c'est qu'il n'y a pas un univers des individuels, puis un univers des ensembles sur les individuels, etc, mais il y a un objet de base sur lequel on travaille qui peut être n'importe quoi (réels, carottes, melange d'ensembles et d'entiers, c'est de toutes façons dans notre esprit seul qu'il a un sens) et un univers des ensembles sur ces objets, etc.

La signification du type d'un objet ensembliste

On peut interpréter de plusieurs façons le fait de dire que l'objet `A` est de type `U`. On peut dire à la manière de la théorie axiomatique des ensembles que U est un univers qui englobe tous les objets de type `U`. De même `A:(Set U)` sera l'univers des ensembles sur les objets de l'univers `U`. Ainsi, tout objet de type `Type` sera un univers de la théorie des ensembles. Mais du fait qu'un ensemble et les objets sur lesquels opère cet ensemble ne sont pas du même type, cela ne coute rien de se représenter l'univers `U` comme l'ensemble des objets de type `U` et la relation 'est de type' comme la relation d'appartenance. C'est une assimilation dans l'esprit de `U` à l'ensemble `[x:U] (A:Prop)A->A` où `(A:Prop)A->A` est un théorème simple utilisé pour dire que cette proposition est demontrable quelque soit `x`.

L'ensemble vide

Avec cette modélisation étagée des ensembles, nous sommes obligés de définir un ensemble vide par étage.

On définira donc un ensemble vide par univers :

```
Global EV : (U:Type)(Set U) = [U:Type][x:U] {}
```

Voici une petite application utilisant cette définition de l'ensemble vide :

```
Theorem inclus_vide (U:Type)(A:(Set U))
           (inclus U A (EV U))->(eqpt (Set U) A (EV U))
 Proof [U:Type][A:(Set U)][h:(inclus U A (EV U))]
       (ext U A (EV U)
            [x:U](iso (A x) (EV U x)
                        (h x)
                        [hV:(EV U x)](hV (A x)))).
```

Dans la preuve, `ext` représente l'axiome d'extensionnalité dont on donne l'énoncé juste ci-dessous.

2.2.2 L'égalite des ensembles

L'égalité fondamentale utilisée sur les objets ensemblistes sera celle de Leibniz, notée `eqpt` (deux objets sont égaux s'ils se comportent de la même façon par rapport à n'importe quelle propriété).

Cependant, l'égalité de deux ensembles sera celle de leurs éléments.

On introduira donc l'axiome d'extensionnalité afin de définir cette égalité.

```
 Axiom ext : (U:Type)(A,B:(Set U))
        ((x:U)(Equiv (A x) (B x))) -> (eqpt (Set U) A B).
```

Notons que c'est le seul axiome que nous avons besoin d'introduire (excepté le tiers-exclu qui intervient spécifiquement pour le théorème de Schroeder-Bernstein)

2.2.3 Commentaires sur les notions générales de la théorie des ensembles

Les diverses notions d'intersection, somme, paire,... se construisent aisément dans le calcul des constructions sans passer par la formulation abstraite de la théorie axiomatique des ensembles qui elle réclame des axiomes d'existence (axiomes de séparabilité, de somme, de paire,...).

En théorie axiomatique des ensembles, par exemple, la paire est définie par l'axiome :

Axiome de paire :
$\forall x, y\ \exists z\ tq\ \forall w\ (w \in z \Leftrightarrow\ w = x\ ou\ w = y)$

A l'aide de l'axiome d'extensionnalité, on établie l'unicité du z tel que $(w \in z\ \Leftrightarrow\ w = x\ ou\ w = y)$, ce qui permet d'établir le théorème de justification de la paire :

Théorème :
$\forall x, y\ \exists! z\ tq\ \forall w\ (w \in z\ \ \Leftrightarrow\ w = x\ ou\ w = y)$

Ce théorème justifie alors la définition de la paire de x et y (notée $\{x, y\}$) :

Définition $w \in \{x, y\}\ \ \Leftrightarrow\ \ w = x\ ou\ w = y)$

Alors qu'en constructif, on peut la définir par :

```
Global paire : (U:Type)U->U->(Set U)
   = [U:Type][x,y:U][z:U](eqpt U x z)/\(eqpt U y z).
```

On peut de même définir l'intersection et la somme d'un ensemble par :

```
Global inter : (U:Type)(Set U)->(Set U)->(Set U)
   = [U:Type][A,B:(Set U)][x:U](A x)/\(B x).

Global somme : (U:Type)(Set (Set U))->(Set U)
   = [U:Type][D: (Set (Set U))][x:U]
             (sig2t (Set U) D [B:(Set U)](B x)).
```

L'existence et l'unicité de l'intersection, de la somme, de la paire,... résultent naturellement de la façon fonctionnelle dont on les définit.

Notons que la somme de `D` peut être définie d'une autre manière en considérant que c'est le plus petit ensemble contenant les ensembles de `D` :

```
Inductive somme' [U:Type;D:(Set (Set U))] : U->Prop =
   somme'_inclus2 : (A:(Set U))(D A)->(x:U)(A x)->(somme' U D x).
```

Cette définition tout comme l'autre permet de définir les deux théorèmes d'encadrement suivants (utilisés par la preuve du théorème de Schroeder-Bernstein), quoique la deuxième définition établie par `Inductive` prouve directement le deuxième théorème :

```
Theorem somme_inclus1
   (U:Type)(D:(Set (Set U)))(A:(Set U))
         ((C:(Set U))(D C)->(inclus U C A))->(inclus U (somme U D) A)

Theorem somme_inclus2
   (U:Type)(D:(Set (Set U)))
         (A:(Set U))(D A)->(inclus U A (somme U D))
```

On peut montrer que la première définition entraine la deuxième :

```
Theorem Sommes (D:(Set (Set U)))(inclus U (somme U D) (somme' U D))
 Proof [D:(Set (Set U))][x:U][h:(somme U D x)]
       (h (somme' U D x)
           [B:(Set U)][hD:(D B)][hx:(B x)]
                              (somme'_inclus2 U D B hD x hx)).
```

En revanche, le sens inverse semble compromis, ne pouvant particulariser à partir de la définition `somme'` un `B` dans `D` et contenant `x` qui permette d'aboutir à la définition de `somme`.

2.2.4 Les relations

De même que nous avons défini un ensemble par sa fonction caractéristique, nous allons définir une relation par sa fonction caractéristique.

Une relation portant sur des objets des univers `U` et `U'` sera de type `U -> U' -> Prop`. On notera que le type du type d'une relation est `Type` ce qui préserve l'homogénéité de typage des univers d'objets que l'on considère.

L'ensemble des relations sur les objets des univers respectifs `U` et `U'` sera noté avec le constructeur `Relation` :

```
Definition Relation [U,U':Type] U -> U' -> Prop.
```

En théorie axiomatique des ensembles , la relation est considéré comme un ensemble sur des paires orientées, définies comme la paire du premier élément et de la paire des deux éléments (la paire orientée de x et y est $\{x, \{x, y\}\}$). Ce n'est pas possible en théorie typée des ensembles puisqu'on ne peut pas prendre la paire de x et $\{x, y\}$ qui sont de niveaux différents.

Contrairement à la théorie axiomatique des ensembles , les relations ne sont donc pas de même nature que les ensembles, mais elles sont de même type. Notons qu'on aurait pu introduire le produit de deux univers par :

```
Definition croix [U,U':Type](U'':Type)(U->U'->U'')->U''.
```

Mais dans ce cas, le type de `(croix U U')` aurait été `Type 2` et aurait été incompatible avec la définition de `Set` sur des objets de type `Type`.

De toutes façons, l'utilisation de la forme curryfiée `U -> U' -> Prop` ne rend pas nécessaire d'introduire explicitement le produit de deux univers.

Notons aussi, qu'il nous faudra introduire un axiome d'extensionnalité pour l'égalité des relations :

```
Axiom ext_R : (U,U':Type)(R,S:(Relation U U'))
              ((x:U)(y:U')(Equiv (R x y) (S x y)))
                     ->(eqpt (Relation U U') R S).
```

A titre d'exemple d'utilisation des relations, nous avons défini ici les propriétés des relations d'équivalence :

```
Definition Rel [U,U':Type][A:(Set U)][B:(Set U')][r:(Relation U U')]
           (x:U)(y:U')(R x y)->((A x)/\(B y)).

Variable U:Type.
Variable A:(Set U).
Variable R:(Relation U U).

Name RelA (Rel U U A A R).

Definition AReflexif
   (RelA)/\(x:U)(A x)->(R x x).

Definition ASymetrique
   (RelA)/\(x,y:U)(A x)->(A y)->(R x y)->(R y x).

Definition ATransitif
   (RelA)/\(x,y,z:U)(A x)->(A y)->(A z)
                 ->(R x y)->(R y z)->(R x z).

Definition AEquivalence
   AReflexif /\ ASymetrique /\ ATransitif.

Discharge U.
```

Et nous avons montré que la relation triviale sur l'ensemble A, $\forall x, y \in A x \Re y \Leftrightarrow x = y$ est une relation d'équivalence sur A (cf listings : Setrel.v).

Faisons à ce sujet quelques remarques :

- Nous imposons à une relation dont on veut montrer par exemple qu'elle est réflexive sur A qu'elle ne soit pas définie ailleurs que sur A. Cela n'est pas nécessaire a priori car seule la restriction de la relation à A nous intéresse. C'est par souci de clarté que nous avons imposé, par la propriété `Rel`, que la relation ne soit définie que sur A. Cela lie intimement la relation et l'ensemble où elle possède telle ou telle propriété. Cela évite le bricolage que l'on serait tenté de faire en jouant avec l'ensemble-crible à travers lequel on voit la relation.

Nous reparlerons de ce problème à propos des fonctions.

- Notons que les hypothèses `(A x)` et `(A y)` sont redondantes avec celle de `(R x y)` du fait de `(Rel U U A A R)` dans la définition de `ASymetrique`. C'est là encore par souci de clarté que nous les avons laissées. Elles insistent sur le fait que l'on prend $x \in A$ et $y \in A$, ce que traduit le `(x:U)(A x)` et le `(y:U)(A y)`. Elles offrent de plus la possibilité de faire une abbréviation `(for x in A)` (par exemple) que la machine traduirait automatiquement par `(x:U)(A x)`.

- Comme nous l'avons vu, le type `U` peut être vu comme l'ensemble des objets de type `U`. On aimerait ainsi définir la réflexivité, la symétrie, la transitivité et l'équivalence sur `U` en tant qu'ensemble sans avoir à reprendre `AReflexif`, `ASymetrique`, `ATransitif` et `AEquivalence` avec l'ensemble `[x:U] (A:Prop)A->A` ce qui alourdirait beaucoup. On établirait alors des définitions plus simples

```
Variable U:Type.

Variable R:(Relation U U).

Definition Reflexif
    (x:U) (R x x).

Definition Symetrique
    (x,y:U) (R x y)->(R y x).

Definition Transitif
    (x,y,z:U) (R x y)->(R y z)->(R x z).

Definition Equivalence
    Reflexif /\ Symetrique /\ Transitif.

Discharge U.
```

C'est ces définitions là par exemple qu'on utiliserait pour démontrer que l'équipollence est une relation d'équivalence sur `(Set U)`.

2.2.5 Les fonctions

La fonction : une relation

Nous définirons la fonction comme étant une relation avec la propriéte en plus que deux éléments ayant même image sont égaux, comme le fait classiquement la théorie des ensembles. Les fonctions seront donc des objets de type `U -> U' -> Prop`.

Il aurait été intéressant de considérer les fonctions comme des objets de type `U -> U'`, insistant ainsi sur l'aspect fonctionnel. Mais cela pose des problèmes pour le théorème de Schroeder-Bernstein , par exemple pour définir
$h : A \rightarrow B$ *tel que*
$h(x) = f(x)$ *si* $x \in J$ *et*
$h(x) = g^{-1}(x)$ *si* $x \in (A \backslash J)$

A priori, il est plus fort de considérer les fonctions comme des relations puisque

si $f_F : U \rightarrow U'$, alors on peut considérer f comme une relation par :

```
Definition fR [x:U][y:U'](eqpt U' y (fF x)).
```

alors qu'il ne semble pas possible de passer de l'aspect relation à l'aspect fonction

Le problème des ensembles source et but d'une fonction-relation

Les fonctions dans la théorie axiomatique des ensembles [2] sont définies par

f *fonction de* $A \rightarrow B \Leftrightarrow (\forall x \in A, \forall y, z \in B,\ (f\ x\ y) \Rightarrow (f\ x\ z) \Rightarrow y = z$ *et* $domaine(f) = A$ *et* $rang(f) = B$

On doit donc passer par l'axiome d'extensionnalité pour prouver les égalités des ensembles $domaine(f) = A$ et $rang(f) = B$.

Nous allons essayer ici un autre codage qui évite de faire intervenir l'axiome d'extensionnalité pour prouver qu'une relation f est une fonction.

Nous avons eu le choix entre définir les ensembles d'arrivée et de départ de la fonction directement liés à la fonction ou bien liés seulement à ses propriétés. A savoir :

1. Une définition qui ne s'occupe pas du domaine de la fonction, mais seulement des ensembles où on la considère comme fonction.

   ```
   Variables U,U':Type.
   Variables A:(Set U).
   Variables B:(Set U').

   Definition fonction [f:(Relation U U')]
         (x:U)(y,z:U')(A x)->(B y)->(B z)->(f x y)->(f x z)->(eqpt U' y z)
       /\(x:U)(A x)->(sig2t U' (f x) B).
   ```

```
Discharge U.
```

Un avantage de cette définition est de manipuler facilement la restriction d'une fonction. Ce sera le même symbole mais vu sous des ensembles différents. Un autre avantage est de définir la propriété `fonction` avec seulement deux hypothèses (au lieu de trois pour les autres définitions envisagées).

On perd cependant en clarté de définition d'une fonction donnée, celle-ci pouvant être n'importe quoi là où on ne la regarde pas, ce qui pose des problèmes, par exemple, pour définir l'image d'un ensemble par une fonction. Il faudrait dans ce cas définir sur quel ensemble B on prend la trace de f(A) :

```
Definition Im [U,U':Type][f:(Relation U U')][A:(Set U)][B:(Set U')]
      [y:U'](sig2t U A [x:U](f x y))/\(B y).
```

ce qui n'est guère satisfaisant.

2. Un autre style de définition est

```
Variables U,U':Type.
Variables A:(Set U).
Variables B:(Set U').

Definition fonction [f:(Relation U U')]
         (x:U)(y:U')(A x)->(f x y)->(B y)
       /\(x:U)(y,z:U')(A x)->(f x y)->(f x z)->(eqpt U y z)
       /\(x:U)(A x)->(sigt U (f x)).

Discharge U.
```

qui introduit une dissymétrie entre A et B et souligne l'aspect fonctionnel. Cette définition résoud le probème de l'image et permet de garder le même symbole pour définir la restriction :

```
Theorem restriction
  (inclus U C A)->(fonction U U' A B f)->(fonction U U' C B f)
```

Cependant, la dissymétrie introduite est désagréable. A partir du moment où la fonction est une relation, on aimerait, par exemple, que

```
  (bijection U U' A B f)->(bijection U' U B A (inverse U U' f))
```

avec

```
Global inverse:(U,U':Type)(Relation U U')->(Relation U' U) =
         [U,U':Type][R:(Relation U U')][y:U'][x:U](R x y).
```

ce qui n'est pas le cas.

3. C'est pourquoi nous nous sommes retranchés derrière la définition suivante :

   ```
   Variables U,U':Type.
   Variables A:(Set U).
   Variables B:(Set U').

   Definition Rel [R:(Relation U U')]
            (x:U)(y:U')(R x y)->((A x)/\(B y)).
   Definition au_plus_une_im [f:(Relation U U')]
            (x:U)(y,z:U')(f x y)->(f x z)->(eqpt U' y z).
   Definition au_moins_une_im [f:(Relation U U')]
            ((x:U)(A x)->(sigt U' (f x))).

   Definition fonct_part (Rel f) /\ (au_plus_une_im f).

   Definition fonction [f:(Relation U U')]
         (fonct_part f) /\ (au_moins_une_im f).

   Discharge U.
   ```

   qui détermine implicitement le domaine sur lequel f est définie (les autres définitions ne le définissaient pas du tout).

   La contre-partie est que nous devons définir un opérateur restr :

   ```
   Global restr:(Set U)->(Set U')->(Relation U U') =
    [C:(Set U)][D:(Set U')][x:U][y:U']((C x)/\(D y))/\(f x y).
   ```

   Ce codage permet d'aboutir à nos fins, mais ce ne sera qu'en essayant de développer plus avant la théorie des fonctions dans celle des ensembles qu'on pourra voir ce que vaut cette définition.

4. Ce qui a posteriori semble le plus clair et le plus pratique, mais qui n'a pas été fait, est de reprendre les définitions précédentes en rajoutant les hypothèses `(A x)`, `(B y)` et `(B z)` dans `au_plus_une_im` et `(B y)` dans `au_moins_une_im`. Ces hypothèses sont certes redondantes, mais aussi plus habituelles d'emploi (on est plus habitué à dire $\forall x \in A$ ce qui se

traduit par `(x:U)(A x)`, que $\forall x : U$. et déduire de `(Rel U U' A B f)` que $x \in A$) On obtiendrait ainsi :

```
Variables U,U':Type.
Variables A:(Set U).
Variables B:(Set U').

Definition Rel [R:(Relation U U')]
          (x:U)(y:U')(R x y)->((A x)/\(B y)).
Definition au_plus_une_im [f:(Relation U U')]
          (x:U)(A x)->(y,z:U')(B y)->(B z)->
            (f x y)->(f x z)->(eqpt U' y z).
Definition au_moins_une_im [f:(Relation U U')]
          ((x:U)(A x)->(sig2t U' (f x) B)).

Definition fonction [f:(Relation U U')]
       (Rel f) /\ (au_plus_une_im f) /\ (au_moins_une_im f).

Discharge U.
```

Un autre avantage de cette dernière définition, c'est qu'elle serait compatible avec une éventuelle abréviation `(for x in A)` pour signifier `(x:U)(A x)`.

Ajoutons aussi que dans la définition effectivement utilisée, on passe par l'intermédiaire de `fonct_part` pour définir `fonction`, ce qui n'est pas très gagnant en raison du peu d'usage de `fonct_part` en général. Il serait donc préférable de définir `fonction` directement à partir des trois propriétés ci-dessus énoncées.

Le détail des définitions et théorèmes traitant de fonctions utilisées par la démonstration du théorème de Schroeder-Bernstein , ainsi que d'autres théorèmes démontrés à titre d'exemple d'utilisation de la synthèse et des objets de la théorie des ensembles définis ici se trouvent dans les fichiers Setrel.v et Setrel.ml (cf listings en annexe).

Notons que tout ce qui concerne la restriction n'a pas d'usage pour la démonstration du théorème de Schroeder-Bernstein ici présentée.

2.3 Le théorème de Schroeder-Bernstein

Le théorème de Schroeder-Bernstein s'énonce ainsi :
f injective de A dans B et g injective de B dans A $\Rightarrow \exists h$ *bijective de A dans B*

Donnons tout d'abord la preuve du théorème de façon mathématique:

La preuve consiste à trouver un sous-ensemble J de A tel que f soit bijective de J dans f(J) et g bijective de $B\setminus f(J)$ dans $A\setminus J$. A partir de là, il suffira de construire h qui vaudra f sur J et g sur $A\setminus J$, pour fournir une bijection entre A et B

Pour construire J, on va se donner l'ensemble D des parties C de A qui vérifient

$$C \subset A\setminus g(B\setminus f(C)).$$

D correspond à l'ensemble des parties C de A tel que f de C dans f(C) et g de $B\setminus f(C)$ dans $g(B\setminus f(C))$ ne se chevauche pas.

L'ensemble cherché J sera alors la réunion de tous les éléments de D.

Montrons que J convient, c'est-à-dire que $g(B\setminus f(J))=A\setminus J$:

Remarquons d'abord que :

> quelque soit $C\subset C'$, on a $f(C)\subset f(C')$, puis $B\setminus f(C')\subset B\setminus f(C)$, puis $g(B\setminus f(C'))\subset g(B\setminus f(C))$ (notons cela (*)).

On en déduit que pour tout $C\subset J$, on a

$$g(B\setminus f(J))\subset g(B\setminus f(C))\subset A\setminus C$$

d'où

$$C\subset A\setminus g(B\setminus f(J))$$

d'où par fermeture de la somme

$$J\subset A\setminus g(B\setminus f(J)).$$

D'où le premier sens de l'égalité :

$$g(B\setminus f(J))\subset A\setminus J.$$

En réinjectant $J\subset A\setminus g(B\setminus f(J))=L$ dans (*) on obtient :

$$g(B\setminus f(L))\subset g(B\setminus f(J))$$

d'où

$$g(B\setminus f(L))\subset A\setminus L$$

et donc L est dans D et donc

$$A\setminus g(B\setminus f(J))\subset J$$

d'où l'égalité :

$$g(B\setminus f(J))=A\setminus J.$$

Et de là, la bijection de g de A dans B puis celle de h.

Nous passerons par le théorème de Tarski pour établir que

$$J = A\backslash g(B\backslash f(J)).$$

En effet, J est un point fixe pour la fonction F croissante qui à C dans A associe $A\backslash g(B\backslash f(C))$. Mais nous aurons besoin de l'axiome du tiers-exclu pour montrer que $A\backslash g(B\backslash f(J)) \subset J$ entraine $(A\backslash J) \subset g(B\backslash f(J))$.

Passons ensuite au codage de la preuve dans le calcul des constructions.

Définir un h qui convient est la partie la plus facile du théorème. h se construit sans problème par :

```
Name F [C:SU](diff U A (Img (diff U' B (Imf C)))).

Name D [C:SU](inclus U C (F C)).

Name J (somme U D).

Definition h [x:U][y:U']((J x)/\(f x y))\/((diff U A J x)/\(g y x)).
```

Il est moins rapide de montrer pourquoi ce h convient.

Il découle rapidement de la définition de h que celui-ci est de A dans B.

L'injectivité de f et de g permet de prouver que h a bien une image au plus sur B et un antécédent au plus sur A.

On va montrer que g est surjective de $(B\backslash f(J))$ sur $A\backslash J$, ce qui entraînera l'existence d'une image par h de tout élément de A.

L'existence d'un antécédent sera simple par le fait que f est surjective sur f(J).

Et c'est là un morceau important de montrer la surjectivité de g sur A moins [AJ. Cela revient à prouver que être dans $(A\backslash J)$, c'est être dans $g(B\backslash f(J))$.

Moyennant l'axiome du tiers-exclu, cela revient à montrer que être dans $(A\backslash g(B\backslash f(J)))$, c'est être dans J, ce qui se montre par le théorème du point fixe de Tarski, via les propriétés de la somme :

$$\forall C \in D,\ C \subset (sommeD)$$

et

$$\forall E,\ si\ (C \in D\ \Rightarrow\ C \subset E)\ alors\ (somme\ D) \subset E$$

et via la croissance de F.

Notons que l'axiome d'extensionnalité introduit de façon générale pour la théorie des ensembles n'est pas nécessaire pour ce théorème où il suffit d'utiliser les inclusions dans un sens et dans l'autre.

cf listings : toute la série des Schroeder : Schroeder1.v, Schroeder1.ml, Schroeder2.v, Schroeder2.ml et Schroeder3.v à lire dans l'ordre indiqué par mkSchroeder.ml .

Une application amusante de la construction de `h` est de définir des entiers tels que le type du type des entiers soit `Type` puis de définir des injections sur les entiers l'une de `A` dans `B` et l'autre de `B` dans `A` et d'en déduire le `h` qui met `A` et `B` en bijection (cf fichier ExSch.v en annexe).

2.4 La bijection entre 2^A et $\mathcal{P}(A)$

Dans la théorie axiomatique des ensembles , ce théorème est établie en définissant 2 comme la paire de l'ensemble vide et du singleton contenant l'ensemble vide.

On le prouve alors en définissant la fonction F définie par

$\forall B \in \mathcal{P}(A)$, F(B)=f où $\forall x \in A$,
f(x) est l'ensemble vide si $x \in B$
f(x) est le singleton contenant l'ensemble vide si $x \in A \backslash B$

et en montrant que F est une bijection.

Dans le calcul des constructions, en l'absence d'ordinaux, nous définirons 2 comme un ensemble ayant deux éléments distincts :

```
Variable U':Type.

Variables y1,y2:U'.

Hypothesis justif_Deux : ((eqpt U' y1 y2)->{}).

Definition Deux (paire U' y1 y2).
```

La fonction F se prête bien à une définition de type fonctionnel. Nous la définirons par :

```
Inductive fn_of [B:(Set U);x:U;y:U'] : Prop =
   fn_ofl_intro : (B x)->(eqpt U' y1 y)->(fn_of B x y)
 | fn_ofr_intro : (diff U A B x)->(eqpt U' y2 y)->(fn_of B x y).
```

et nous la ramènerons après sous une forme relationnelle par

```
Definition F [B:SU][f:FUU'](Part B)/\(eqpt FUU' (fn_of B) f).
```

Nous introduirons aussi

```
Definition B_of [f:FUU'][x:U](f x y1).
```

qui facilitera la démonstration de l'existence d'un antécédant.

Nous aurions pu définir F directement de façon relationnelle par

```
Inductive Fgen [B:(Set U);f:(Relation U U')] : Prop =
   Fgenl_intro : (x:U)(B x)->(f x y1)->(fgen B f)
 | Fgenr_intro : (x:U)(diff U A B x)->(f x y2)->(Fgen B f).

Inductive F [B:(Set U);f:(Relation U U')] : Prop =
   F_intro : (inclus U B A)->(fonction U U' Deux A f)
               ->(Fgen B f)->(F B f).
```

Mais la démonstration de l'existence de l'image et de l'antécédant aurait ramené de toutes façons à construire `fn_of` et `B_of`.

Remarquons que l'axiome d'extensionnalité pour l'égalité des ensembles et l'axiome d'extensionnalité pour l'égalité des relations sont nécessaires à partir du moment où on a à prouver l'égalité de fonction définie par deux biais différents.

Notons aussi que la définition de F bénéficie avantageusement de la constance du type des univers. F est de type `(Relation (Set U) (Relation U U'))` où `U` et `U'` sont les types de base que nous considérons.

cf listings : série des Expon :
Expon1.v, Expon1.ml, Expon2.v, Expon2.ml et Expon3.v

2.5 Le problème des abréviations

Il serait agréable d'avoir une présentation des différentes constantes définies qui soit plus "mathématique". Par exemple, que `A` et `B` de type `U` sont équipollents se note `A~B`, que `(A x)` $(= x \in A)$ se note `(x in A)`, ou même que `(for x in A)` traduise directement `(x:U)(A x)` et réciproquement `(x:U)(A x)` traduise `(for x in A)` lorsque `A` est de type `(Set U)`.

Cela pose plusieurs problèmes :

- Certaines variables sont souvent redondantes dans une expression, par exemple dans `(inclus U A B)`, `U` est superflu puisqu'il se déduit du type de `A` `(A:(Set U))`.

  Ce problème devrait être réglé dans une prochaine version de la machine constructive (les outils théoriques pour résoudre cela sont exposés dans [Colloque85]).

  C'est essentiellement à cause de cela que nous n'avons pas utilisé d'abréviations. En effet, rajouter un type `U` non attendu dans la syntaxe de la plupart de nos constantes aurait alourdi considérablement la notation : un `<U>sigt(P)` pour `(sigt U P)` ou un `(<U>A ~ <U'>B)` pour `(equipollence U U' A B)` auraient été peu avantageux.

- Nous nous trouvons d'autre part limités par un jeu de caractères qui n'accepte pas $\exists$, $\in$, $\subset$, $\simeq$, $^{-1}$, $\cap$, ... Quand aux variables comme `U` pour l'union, leur adoption comme abréviation obligerait à s'interdire la lettre `U` pour désigner un univers. Le symbolisme mathématique dépasse largement les possibilités que nous offre la police de caractères standard. Il reste alors à se rabattre vers les identificateurs pas trop court, et l'abréviation ne serait plus que syntaxe (savoir dans quel ordre donner les arguments et lesquels sont indispensables).

- Il faut d'autre part que l'usage d'abréviations ne perturbe pas le fonctionnement de la machine constructive. Il faut qu'elles soient limitées au parser et au pretty-printer. Mais comment faire pour une constante comme le `and` que l'on aimerait mieux avoir en machine sous sa forme dépliée plus pratique (`(C:Prop)(A->B->C)->C`), mais que l'on aimerait mieux lire sous sa forme logique `and`

  A titre d'exemple, voilà ce que pourrait donner quelques-uns de nos exemples en utilisant des abéviations dans un cadre de rêve (et donc sans se soucier des problèmes de parsing) :

```
Syntax (eqpt U x y) "x=y".
Syntax (inclus U A B) "A incl B".
Syntax (diff U A B) "A\B".
Syntax (somme U D) "Somme(D)".
Syntax (Im U f A) "f(A)". (* Seulement si f:(Relation U U') et A:(Set U) *)
Syntax (injection U U' A B f) "i(f):A->B".
Syntax (bijection U U' A B f) "b(f):A->B".
Syntax (equipollence U U' A B) "A~B".
Syntax (inf_card U U' A B) "A<~B".
Syntax (sig2t U P Q) "exist2 P Q".
(* Theoreme de Schroeder-Bernstein *)

Variables U,U':Type.
Variable A:(Set U).
Variable B:(Set U').
Variable f:(Relation U U').
Variable g:(Relation U' U).

Hypothesis f_inj:i(f)A->B.
Hypothesis g_inj:i(g)B->A.

Name D [C:SU](C incl A)/\((g(B\f(C))) incl (A\C)).
Name J (Somme(D)).
Name L g(B\f(J)).
```

```
Lemma J_dans_A (J incl A)
Lemma L_dans_A (L incl A)
Lemma L_incl_A_moins_J (L incl (A\J))
Lemma A_moins_J_incl_L ((A\J) incl L)

Theorem J_is_good L=(A\J)

Theorem g_surj
 (for x in (A\J))->(exist2 [y:U'](g y x) [y:U'](y in (B\(f(J)))))

Inductive h [x:U][y:U'] : Prop =
   hl_intro : (x in J)->(f x y)->(h x y)
 | hr_intro : (x in (A\J))->(g y x)->(h x y).

Theorem h_bij (b(h):A->B)

Discharge f.

Theorem Schroeder (A <~ B) -> (B <~ A) -> (A ~ B)

Discharge U.
```

Chapitre 3

Les Tactiques

3.1 La résolution avec variables cachées

3.1.1 La résolution avec variables cachées

Il arrive fréquemment qu'un théorème, pour s'appliquer, nécessite une instanciation d'une ou plusieurs variables n'intervenant pas dans la conclusion. C'est le cas typique du `existt` (= `sigt_intro`) :

```
Definition sigt [A:Type][P:A->Prop](B:Prop)((x:A)(P x)->B)->B.

Theorem existt  (A:Type)(P:A->Prop)(x:A)(P x)->(sigt A P)
 Proof     [A:Type][P:A->Prop][x:A][p:(P x)]
           [B:Prop][f:(a:A)(P a)->B](f x p).
```

`x` n'apparait pas dans la conclusion de `existt`. Il est alors nécessaire de donner au `Resolve` la liste des instanciations des variables cachées. Cela est fait par `Resolven` *thname lc* où *lc* est une liste de constructions (syntaxe concrète) qui doit alimenter les "trous" du théorème.
Par exemple :

```
<<1 subgoal
  1......(sigt U A)
  H : (A y)
  y : U
  A : (Set U)
  U : Type>>
 : message
```

```
#Resolven "existt" [<<y>>];;
<<1 subgoal
  1......(A y)
  H : (A y)
  y : U
  A : (Set U)
  U : Type>>
 : message

#Assumptions();;
<<goal proved>> : message
```

On peut se demander si la machine peut elle-même trouver les instanciations qu'il faut. C'est des fois possible, par exemple, à démontrer

```
<<1 subgoal
  1......(sigt U P)
  H : (P x)
  y : U
  x : U
  P : U->Prop
  U : Type>>
 : message
```

où la machine doit pouvoir deviner que résoudre avec la variable `x` prouve le but.

Il y a d'autres cas où il ne faut pas se fier aux apparences. Par exemple :

```
<<1 subgoal
  1......(sigt (Relation U U') (fonction U U' C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  f : (Relation U U')
  ...>>
 : message
```

`f` semble une solution mais en fait c'est `(restr U U' f C B)` qui fait marcher :

```
#Resolven "existt" [<<(restr U U' f C B)>>];;
<<1 subgoal
  1......(fonction U U' C B (restr U U' f C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  f : (Relation U U')>>
 : message
```

Il ne reste plus qu'à appliquer le théorème `restriction` et la preuve est établie :

```
Theorem restriction
        (U,U':Type)(A:(Set U))(B:(Set U'))
        (C:(Set U))(inclus U C A)->(fonction U U' A B f)
             ->(fonction U U' C B (restr U U' f C B))

#by (resolven "restriction" [<<A>>] THEN assumptions);;
<<goal proved>> : message
```

L'absence de généralité oblige donc à prévoir un `Resolve` à paramètres. En revanche, ce que l'on peut imaginer, mais qui n'est pas fait ici, c'est, dans le cadre d'une utilisation interactive de la synthèse, que `Resolve` demande à l'utilisateur des instanciations lorsqu'il rencontre des variables cachées et le cas échéant que l'utilisateur lui donne ou bien lui signale que l'instanciation est évidente.

Dans la version ici présentée, pour indiquer des variables cachées dans une tactique pratiquant la résolution, il faut suffixer le nom de la tactique de la lettre `n` et rajouter en paramètre la liste des instanciations à effectuer.

Par exemple `Introdef` et `Hypelimres` que nous verrons plus loin s'écrierons `Introdefn` et `Hypelimresn` avec variables cachées.

3.1.2 Le source de la résolution avec variables cachées

La tactique centrale de la résolution avec variables cachées est `tac_res_n_pf` *refth lv*. Cette fonction CAML récupère le théorème que représente *refth*, met en correspondance la conclusion de ce théorème avec le but à démontrer, puis complète la liste de substitutions ainsi déterminée avec l'association une à une des variables du théorème non présentes dans sa conclusion (les variables cachées) et des variables de la liste *lv* en vérifiant le typage. Une fois la liste compléte et correctement typée de toutes les substitutions à faire, il fait appel à la fonction validation.

cf listings : introelim.ml .

3.2 Les tactiques d'introduction

3.2.1 Les tactiques et théorèmes d'introduction

De façon très courante dans une démonstration, on a besoin de revenir à la définition d'un symbole pour mener à bien la preuve du théorème.

Par exemple, à démontrer que

si f est une fonction de A dans B et C un sous-ensemble de A alors la restriction de f sur C et B est une fonction

ce qui se formalise par

```
(C:(Set U))(inclus U C A)->(fonction U U' A B f)
                  ->(fonction U U' C B (restr U U' f C B))
```

Pour cela, il faut "revenir à la définition" de fonction ce qu'on fait par unfold (cas où l'on explicite le but) ou unfold_hyp (cas où l'on explicite une hypothèse)

Notre démonstration se déroulera donc ainsi :

```
#goal <<(C:(Set U))(inclus U C A)->(fonction U U' A B f)->(fonction U U' C B (restr U U' f
<<......(C:(Set U))(inclus U C A)->(fonction U U' A B f)->(fonction U U' C B (restr U U' f
 : message

#Intros();;
<<1 subgoal
  1......(fonction U U' C B (restr U U' f C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (unfold_hyp "H0" "fonction" THEN unfold_hyp "H0" "fonct_part");;
<<1 subgoal
  1......(fonction U U' C B (restr U U' f C B))
  H0 : (Rel U U' A B f)/\(au_plus_une_im U U' f)/\(au_moins_une_im U U' A f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (and_elim "H0" THEN and_elim "H0");;
<<1 subgoal
  1......(fonction U U' C B (restr U U' f C B))
  H4 : (au_moins_une_im U U' A f)
```

```
  H3 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H0 : (Rel U U' A B f)/\(au_plus_une_im U U' f)/\(au_moins_une_im U U' A f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (unfold "fonction" THEN unfold "fonct_part");;
<<1 subgoal
  1......(Rel U U' C B (restr U U' f C B))/\(au_plus_une_im U U' (restr U U' f C B))/\(au_mo
  H4 : (au_moins_une_im U U' A f)
  H3 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H0 : (Rel U U' A B f)/\(au_plus_une_im U U' f)/\(au_moins_une_im U U' A f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (and_intro);;
<<2 subgoals
  1......(Rel U U' C B (restr U U' f C B))/\(au_plus_une_im U U' (restr U U' f C B))
  H4 : (au_moins_une_im U U' A f)
  H3 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H0 : (Rel U U' A B f)/\(au_plus_une_im U U' f)/\(au_moins_une_im U U' A f)
  H : (inclus U C A)
  C : (Set U)
  2......(au_moins_une_im U U' C (restr U U' f C B))>>
 : message

#by (and_intro);;
<<3 subgoals
  1......(Rel U U' C B (restr U U' f C B))
  H4 : (au_moins_une_im U U' A f)
  H3 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H0 : (Rel U U' A B f)/\(au_plus_une_im U U' f)/\(au_moins_une_im U U' A f)
  H : (inclus U C A)
  C : (Set U)
  2......(au_plus_une_im U U' (restr U U' f C B))
```

```
  3......(au_moins_une_im U U' C (restr U U' f C B))>>
 : message
```

et cætera

Cela devient vite fastidieux alors que ce ne sont que des trivialités. Le unfold a l'inconvénient de ne déplier une constante que sur l'expression qui la définit. On aimerait parfois, voire souvent, ramener une expression à une autre expression plus pratique ou plus significative que le simple résultat du dépliage. Dans l'exemple ci-dessus, on aimerait de façon immédiate ramener la démonstration de `(fonction U U' A B f)` à celle de `(Rel U U' A B f)`, `(au_moins_une_im U U' A B f)`, `(au_plus_une_im U U' A B f)`. Pour cela, on peut créer un théorème d'introduction de fonction :

```
(U,U':Type)(A:(Set U))(B:(Set U'))(f:(Relation U U'))
(Rel U U' A B f)->(au_moins_une_im U U' A B f)->(au_plus_une_im U U' A B f)
->(fonction U U' A B f)
```

que l'on appliquera au but par un résolve.

Mais à partir du moment où on peut appliquer un quelconque théorème d'introduction à un but, on peut aussi appliquer celui directement associé à la définition de la constante. On peut alors se passer du `Unfold`, en effectuant une résolution avec le théorème d'introduction propre de la constante, ce qui permet d'insérer le dépliage dans le cadre plus général de la résolution.

Voyons un exemple : le dépliage de `fonction` :

```
Theorem fonction_intro
 (U,U':Type)(A:(Set U))(B:(Set U'))(f:(Relation U U'))
      (fonct_part U U' A B f)->(au_moins_un_ant U U' A f)
                 ->(fonction U U' A B f)

<<1 subgoal
  1......(fonction U U' C B (restr U U' f C B))
  H4 : (au_moins_une_im U U' A f)
  H3 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H0 : (Rel U U' A B f)/\(au_plus_une_im U U' f)/\(au_moins_une_im U U' A f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (resolve "fonction_intro");;
<<2 subgoals
  1......(fonct_part U U' C B (restr U U' f C B))
```

```
  H4 : (au_moins_une_im U U' A f)
  H3 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)/\(au_plus_une_im U U' f)
  H0 : (Rel U U' A B f)/\(au_plus_une_im U U' f)/\(au_moins_une_im U U' A f)
  H : (inclus U C A)
  C : (Set U)>>
  2......(au_moins_une_im U U' A (restr U U' f C B))>>
 : message
```

Notons que nous avons établi `fonction_intro` sous sa forme curryfiée qui est plus pratique d'emploi.

Nous allons alors généraliser l'emploi de ces théorèmes d'introduction par la tactique **`introdef name`** qui recherchera un théorème de nom `(name^_intro)` et exécutera une résolution entre ce but courant et ce théorème.

Parallèllement à cela, nous allons donc aussi développer une fonction qui à partir d'une constante crée le théorème d'introduction qui lui est associé. La dite fonction explicitera le `/\`, le `\/`, le `sigt` (le 'il existe') et le `sig2t` intervenant éventuellement dans une définition, ce qui rend le théorème beaucoup plus efficace d'emploi. Cette fonction, appelée `make_intro` sera expliquée dans la section suivante.

Voici un exemple d'application de cet utilitaire :

```
Global somme:(Set (Set U))->(Set U) =
     [U:Type][D:(Set (Set U))][x:U]
         (sig2t (Set U) D [B:(Set U)](B x)).

Theorem somme_intro
     (U:Type)(D:(Set (Set U)))(x:U)
         (B:(Set U))(D B)->(B x)->(somme U D x)
 Proof [U:Type][D:(Set (Set U))][x:U][B:(Set U)]
         (exist2t (Set U) D [C:(Set U)](C x) B).
```

Notons que le cas du `\/` est particulier, car il conduit à deux théorèmes d'introduction que l'on nommera suffixé l'un par `l_intro`, l'autre par `r_intro`. On aura à spécifier dans le `introdef` quel théorème gauche ou droite on veut utiliser.

Exemple :

```
Global union : (U:Type)(Set U)->(Set U)->(Set U)
             = [U:Type][A,B:(Set U)][x:U](A x)/\(B x).
```

```
Theorem unionl_intro (U:Type)(A,B:(Set U))(x:U)(A x)->(union U A B x)

Theorem unionr_intro (U:Type)(A,B:(Set U))(x:U)(B x)->(union U A B x)
```

Indépendamment de la version de la machine avec laquelle on a travaillé a été développé une fonction vernaculaire `Inductive` qui définit les constantes en fonction de leurs propriétés d'introduction. C'est ainsi un moyen simple et efficace de définir simultanément une constante et son(ses) théorème(s) d'introduction.

Par exemple, pour définir `somme` et `union` :

```
Variables U:Type.
Variables D:(Set (Set U)).

Inductive somme [x:U] : Prop =
   somme_intro : (E:(Set U))(D E)->(E x)->(somme x).

Variables A,B:(Set U).

Inductive union [x:U] : Prop =
   unionl_intro : (A x)->(union x)
 | unionr_intro : (B x)->(union x).

Discharge U.

Variables U,U':Type.
Variables A:(Set U).
Variables B:(Set U').
Variables f:(Relation U U').

Inductive fonction : Prop =
   fonction_intro :
   (fonct_part U U' A B f)->(au_moins_une_im U U' A f)->fonction.

Discharge U.
```

Pour bien faire, il faudrait réécrire toute les définitions de constantes faisant intervenir des /\, \/, `sig`, etc avec la commande `Inductive` définissant ainsi simultanément le théorème d'introduction, le théorème d'élimination (comme on le verra au paragraphe suivant), et le sens intuitif de la constante.

Notons aussi que l'utilisation d'un théorème pour déplier une constante (à la place d'un `unfold`) exigera de la place mémoire superflue, lorsqu'on sait qu'un `unfold` n'en consomme pas.

Dans une version plus bavarde de la synthèse on pourra envisager de faire poser la question "quel théorème d'introduction voulez-vous utiliser ?" en réponse à l'instruction `introdef`. Et qu'éventuellement si la résolution se ramène à un dépliage, ne faire que ce dépliage ce qui économisera quelques rédex.

Notons enfin que le `unfold` garde son indispensabilité pour déplier les constantes autre que la constante de tête d'une proposition.

3.2.2 Le `make_intro`

C'est une fonction CAML qui construit le(s) théorème(s) d'introduction lié(s) à une constante en tenant compte du `/\`, `\/`, `sigt` et `sig2t` intervenant potentiellement dans la définition. Pour démontrer le(s) théorème(s), il construit une(des) tactique(s) adéquate(s) qu'il applique à(aux) l'énoncé(s) cherché(s). Il récupère ensuite la preuve et enregistre (de façon sticky) le(s) théorème(s) et sa(leur) démonstration.

Il est facilement extensible, par exemple pour prendre en compte le `/\` 'a trois arguments, le `sig`, etc. Il ne modifie pas les environnements de la machine (excepté qu'il y rajoute le(s) théorème(s) démontré(s)).

Pour créer tous les théorèmes d'introduction non encore énoncés, il y a la commande `all_intro_th()` Et pour créer tous les théorèmes d'introduction non encore énoncés avec question préalable, il y a la commande `make_ask_intro()`

cf listings : introelim.ml .

3.2.3 Le `introdef`

Le `introdef` *nom* est un simple `resolve` avec le théorème lié à la définition de *nom*

Il est utilisable en fait avec n'importe quel théorème se terminant par `_intro`, car `introdef` se borne à rajouter `_intro` au nom qui lui est passé en argument.

Ses variantes sont :

`introdef` *nom*
`introdefc` *nom*
`introdefn` *nom lc*
`introdefcn` *nom lc*

et en application directe sur le but courant :

`Introdef` *nom*
`Introdefc` *nom*

`Introdefn` *nom lc*
`Introdefcn` *nom lc*

où *lc* est une liste de constructions.

cf listings : introelim.ml .

3.3 Les tactiques d'élimination

De la même façon qu'on peut ramener un but à des sous-buts nécessaires par un théorème d'introduction, on aimerait ramener une hypothèse à des hypothèses suffisantes.

Et de la même façon qu'on s'est donné la possibilité d'introduire une constante de plusieurs façons, on va se permettre d'éliminer une constante dans une hypothèse de plusieurs manières.

Une de ces manières sera l'élimination simple de la constante selon sa définition, avec élimination automatique du `/\`, du `\/`, du `sigt` et du `sig2t` qui sont plus pratiques d'emploi sous leur forme dépliée.

Les autres manières sont au choix de l'utilisateur.

Par exemple, soit l'hypothèse `H : (bijection U U' A B f)`,

l'élimination standard conduit à l'hypothèse

```
H' : (C:Prop)((injection U U' A B f)->(au_moins_un_ant U U' B f)->C)->C
```

Mais on peut aussi se donner la possibilité d'aboutir directement à

```
H' : (C:Prop)((fonction U U A B f)->(au_moins_un_ant U U' B f)
                     ->(au_plus_un_ant U U' f)->C)->C
```

On va alors créer une tactique nommée `hypelim` *hyp* qui élimine la constante de tête de la conclusion de l'hypothèse *hyp* et une tactique `hypelim_with` *hyp thnom* qui fait la même chose en forçant le nom du théorème d'élimination.

Cet élimination se fait par coupure avec le théorème d'élimination associé à la constante.

Exemple : soit l'hypothèse

```
Hyp : (inclus U A B)->(inclus U A C)
```

le théorème d'élimination de inclus est :

```
Theorem inclus_elim (U:Type)(A,B:(Set U))(inclus U A B)->(x:U)(A x)->(B x)
```

et le `Hypelim "Hyp"` donne :

```
Hyp' : (inclus U A B)->(x:U)(A x)->(C x)
```

Le **hypelim** est donc une tactique qui réalise une résolution montante (par rapport à la résolution descendante sur le but) sur une hypothèse sous certaines conditions (celles de l'élimination d'une constante : cf le paragraphe sur le source de **hypelim**).

Il serait d'ailleurs intéressant, dans un cadre plus général dans lequel le **hypelim** s'insèrerait, de permettre la coupure systématique entre propositions du contexte en forme clausale.

La syntaxe a ici été choisie différente de celle de **introdef** (on aurait en effet pu écrire **hypelim hyp name** et que **hypelim** applique le théorème d'élimination de nom `(nom^"_elim")` à l'hypothèse `hyp`). Ce qui avait initialement motivé cela était un souci d'économie du nombre d'arguments à fournir. Mais à l'usage, il s'avère qu'une syntaxe comme celle de **introdef** aurait été préférable. Elle rend plus explicite le rôle du **hypelim**, notamment à la relecture de preuves, lorsqu'on ne sait plus ce que recelait le nom de l'hypothèse.

Exemple :

```
#goal <<(C:(Set U))(inclus U C A)->(fonction U U' A B f)->(fonction U U' C B (restr U U' f C
<<.....(C:(Set U))(inclus U C A)->(fonction U U' A B f)->(fonction U U' C B (restr U U' f C
 : message

#Intros();;
<<1 subgoal
  1.....(fonction U U' C B (restr U U' f C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (hypelim_with "H0" "fonction2_elim");;
<<1 subgoal
  1.....(fonction U U' C B (restr U U' f C B))
  H0' : (PR:Prop)((Rel U U' A B f)->(au_plus_une_im U U' f)->(au_moins_une_im U U' A f)->PR
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message
```

Pratiquement, si on déplie une hypothèse, c'est pour l'utiliser ensuite, c'est pourquoi nous avons aussi créer la tactique **hypelimres** *hyp* qui rajoute une résolution avec le but courant à ce que fait **hypelim**.

Notre exemple devient alors

```
#goal <<(C:(Set U))(inclus U C A)->(fonction U U' A B f)->(fonction U U' C B (restr U U' f C
<<.....(C:(Set U))(inclus U C A)->(fonction U U' A B f)->(fonction U U' C B (restr U U' f C

 : message

#Intros();;
<<1 subgoal
  1.....(fonction U U' C B (restr U U' f C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (hypelimres_with "H0" "fonction2_elim");;
<<1 subgoal
  1.....(Rel U U' A B f)->(au_plus_une_im U U' f)->(au_moins_une_im U U' A f)->(fonction U
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message
```

Ou encore, en usant de l'option `c` :

```
#by (hypelimresc_with "H0" "fonction2_elim");;
<<1 subgoal
  1.....(fonction U U' C B (restr U U' f C B))
  H3 : (Rel U U' A B f)
  H2 : (au_plus_une_im U U' f)
  H1 : (au_moins_une_im U U' A f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message
```

Notons que le `hypelim` ne remplace pas l'hypothèse sur laquelle il travaille, mais en crée une autre de même nom indexée par un prime. L'intérêt est de garder la forme originale de l'hypothèse souvent plus expressive non dépliée alors qu'on a besoin aussi de la forme dépliée.

En revanche, le `hypelimres` ne modifie pas l'hypothèse sur laquelle il travaille et n'en crée aucune autre.

De pair avec le `hypelim` et parallélement au `make_intro`, nous avons établi un `make_elim` *nom* qui crée le théorème d'élimination associé à la constante *nom*, tout en dépliant les `/\`, du `\/`, `sigt` et `sig2t`.

Par exemple :

```
Definition somme [U:Type][D:(Set (Set U))][x:U]
                 (sig2t (Set U) D [B:(Set U)](B x)).

Theorem somme_elim (U:Type)(D:(Set (Set U)))(x:U)
           (somme U D x)->(PR:Prop)((B:(Set U))(D B)->(B x)->PR)->PR
```

Rappelons, comme pour l'introduction, que la commande `Inductive` à la caractéristique de définir simultanément une constante et son théorème d'élimination et que son emploi devrait être généralisé et notamment pour les définitions faisant intervenir des /\, \/, `sig`, etc.

Notons comme pour l'introduction que l'utilisation d'un théorème pour déplier une constante exigera de la place mémoire là où il n'y en avait pas besoin. Utiliser un théorème d'élimination (qui est dans le cas standard réductible au λ-terme identité), c'est alourdir la preuve en y rajoutant un rédex.

On pourrait envisager dans une version plus volubile de la synthèse de n'avoir qu'à indiquer qu'on souhaite faire une élimination, que la machine demande quel théorème on souhaite utiliser et que si ce théorème correspond à un dépliage évident, ne pas rajouter de rédex par l'utilisation du théorème.

Notons enfin que le `unfold_hyp` garde son indispensabilité pour déplier les constantes autre que la constante de tête d'une proposition.

3.3.1 Le `make_elim`

C'est une fonction CAML qui construit le théorème d'élimination lié à une constante en tenant compte du /\, \/, `sigt` et `sig2t` intervenant potentiellement dans la définition. A une définition du type :

`Definition A [x1:U1][..][xn:Un](Ac)`
cet utilitaire crée
`Theorem A_elim (x1:U1)(..)(xn:Un)(A x1 .. xn)->(Ac)`,
sauf si `(Ac)` est `A1->..An->(A'1 /\ A'2)` au quel cas il crée

```
Theorem A_elim (x1:U1)(..)(xn:Un)(A x1 .. xn)->
             A1->..An->(C:Prop)(A'1->A'2->C)->C
```

et du même style pour \/, `sig2t` et `sigt`.

Pour démontrer le théorème, il construit une tactique adéquate qu'il applique à l'énoncé cherché. Il récupère ensuite la preuve et enregistre (de façon sticky) le théorème et sa démonstration.

Il est facilement extensible, par exemple pour prendre en compte le /\ à trois arguments, le `sig`, etc. Il ne modifie pas les environnements de la machine (excepté qu'il y rajoute le théorème démontré).

Pour créer tous les théorèmes d'élimination non encore énoncés, il y a la commande `all_elim_th()` Et pour créer tous les théorèmes d'élimination non encore énoncés avec question préalable, il y a la commande `make_ask_elim()`

cf listings : introelim.ml .

3.3.2 Le `hypelim(_with)`

`Hypelim` *hyp* est une fonction CAML qui réalise une coupure entre la conclusion de l'hypothèse *hyp* vue sous sa forme clausale avec le théorème d'élimination associé à la constante de tête de cette conclusion Le résultat de la coupure est posé comme une nouvelle hypothèse de façon à ce que l'arbre de preuve se construise facilement par application du but actuel abstrait avec la nouvelle hypothèse et d'un λ-terme preuve de cette hypothèse

De l'hypothèse

`A1->..->An->(nom x1 .. xn)`

et du théorème

`(x1':U1)..(xn':Un)(nom x1' .. xn')->(Tc)`

il déduit l'hypothèse

`A1->..->An->(Tc)`

où les variables `x1'` à `xn'` ont été substituées par `x1` à `xn`

`Hypelim_with` fait la même chose en forçant le nom du théorème d'élimination à utiliser. Il faut que ce théorème ait le format

`(x1:U1)..(xn:Un)(A x1 .. xn)->(Ac)`

où `A` est le symbole à éliminer.

cf listings : introelim.ml .

3.3.3 Le `hypelimres(c)(n)(_with)`

Le `hypelimres` *hyp* est une élimination dans *hyp* avec `hypelim` suivi d'une résolution avec l'hypothèse qui en découle. Cependant, contrairement à `hypelim`, il ne conserve pas la nouvelle hypothèse dans l'environnement.

Ses options sont `_with` pour l'élimination et `c` et `n` pour la résolution.

Ses variantes sont :

`hypelimres` *hyp*
`hypelimresc` *hyp*
`hypelimresn` *hyp lc*
`hypelimrescn` *hyp lc*
`hypelimres_with` *hyp thnom*
`hypelimresc_with` *hyp thnom*
`hypelimresn_with` *hyp lc thnom*
`hypelimrescn_with` *hyp lc thnom*

et en application directe sur le but courant :

`Hypelimres` *hyp*
`Hypelimresc` *hyp*

`Hypelimresn` *hyp lc*
`Hypelimrescn` *hyp lc*
`Hypelimres_with` *hyp thnom*
`Hypelimresc_with` *hyp thnom*
`Hypelimresn_with` *hyp lc thnom*
`Hypelimrescn_with` *hyp lc thnom*

où *lc* est une liste de constructions.

cf listings : introelim.ml .

3.4 Des abbréviations de tactiques

3.4.1 L'introduction automatique des hypothèses et résolution automatique des buts évidentes

Une tactique qui réalise cela est le `REPEAT (assumption ORELSE intro)`. On lui donnera le nom de `Nettoie_hyp`. Elle est très pratique à la suite d'un `Resolve` pour mettre les nouveaux buts en forme clausale tout en éliminant les sous-buts évidents.

En reprenant l'exemple ci-dessus :

```
<<1 subgoal
  1......(sigt (Relation U U') (fonction U U' C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  f : (Relation U U')>>

#Resolven "existt" [<<(restr U U' f C B)>>];;
<<1 subgoal
  1......(fonction U U' C B (restr U U' f C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  f : (Relation U U')>>
 : message

#Resolven "restriction" [<<A>>];;
<<2 subgoals
  1......(inclus U C A)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
```

```
  C : (Set U)
  f : (Relation U U')
  2.....(fonction U U' A B f)>>
 : message

<<goal proved>> : message
```

On aimerait ne pas avoir à spécifier un `Assumption` supplémentaire.

C'est pourquoi, à chacune des tactiques de résolution, on rajoutera l'option `c` (avant l'option `n` pour "nettoyer" les sous-buts qui s'ensuivront. On aura ainsi `Hypelimresc`, `Hypelimrescn`, `Introdefc`, `Resolvec`, ...

Notre exemple devient alors :

```
<<1 subgoal
  1.....(fonction U U' C B (restr U U' f C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  f : (Relation U U')>>
 : message

#Resolvecn "restriction" [<<A>>];;
<<goal proved>> : message
```

Il n'est toutefois pas satisfaisant d'avoir une hypothèse qui est passée sous silence, même si celle-ci est évidente. C'est pourquoi on rajoutera un message dans `Assumption` à chaque fois qu'un sous-but est éliminé.

Dans la pratique, comme les versions actuelles de résolution ne s'appliquent que sur des buts formés d'une seule proposition, il est pratique de faire automatiquement le maximum d'introduction, ce que fait l'option `c`.

Une extension de l'option `c`

Il serait inéressant d'enrichir le `Nettoie_hyp` afin qu'il s'occupe automatiquement, non pas seulement des buts directement prouvés par une hypothèse, mais aussi des trivialités du genre `(eqpt U A A)` ou `(inclus U A A)`. On pourrait à cet usage concevoir une liste incluant trivialités et hypothès à laquelle le `Nettoie_hyp` comparerait le but, pour savoir s'il est évident.

3.4.2 Pêle-mêle

– Un `intron n` qui introduit les n premières hypothèses.

- Un `By` pour supprimer le message après l'exécution d'une tactique.

3.5 Résumé des tactiques discutées

Les tactiques les plus utiles que nous utilisons sont résumées ici :

- `Intro` correspond à la démarche : je pose comme acquis la première condition de l'énoncé que j'ai à démontrer.
- `Resolve` *nom* correspond à la démarche : pour prouver le but, j'utilise le théorème (ou l'hypothèse, ou l'axiome, ...) *nom*.
- `Assumption` correspond à la démarche : ce que j'ai à démontrer est évident car c'est dans les hypothèses.
- `Unfold` *nom* correspond à la démarche : je reviens à la définition de *nom*.
- `Unfold_hyp` *hyp nom* : idem pour la constante *nom* dans l'hypothèse *hyp*.
- `Cut` *P* correspond à la démarche : j'admet P que je démontrerai plus tard.
- `Nettoie_hyp` correspond à la démarche : je pose les hypothèses non évidentes de mon but.
- `Resolven` *nom lvar* : variante de `Resolve` où le théorème *nom* nécessite les variables de la liste *lvar* non explicites dans le but.
- `Introdef(c)(n)` *nom* (*lvar*) correspond à la démarche : je ramène la constante *nom* à des conditions nécessaires.
- `Hypelimres(c)(n)(_with)` *hyp* correspond à la démarche : j'utilise les informations cachées derrière l'hypothèse *hyp* pour prouver le but.
- `And_intro` ramène la démonstration d'un but de la forme `A/\B` à la démonstration de `A` et à celle de `B` (a le même effet que `Introdef "and"`).
- `or_intro1` (resp. `or_intro2`) ramène la démonstration d'un but de la forme `A\/B` à la démonstration de `A` (resp de `B`). Cette tactique a le même effet que `Introdef "orl"` (resp `introdef "orr"`)
- `sigt_intro` *y* ramène la démonstration d'un but de la forme `(sigt U P)` à la démonstration de `(P y)` où *y* est censé être une variable témoin du `(sigt U P)` (a (presque) le même effet que `introdefn` [*y*]).
- `And_elim` *hyp* élimine le `/\` dans l'hypothèse *hyp* de la forme `A/\B` en créant les hypothèses prouvant `A,  et \verbB`. (a le même effet que `Hypelimresc` *hyp*).

- `or_elim` *hyp* élimine le `\/` dans l'hypothèse *hyp* de la forme `A\/B` en créant deux sous-buts, l'un avec l'hypothèse `A`, l'autre avec l'hypothèse `B` (a le même effet que `hypelimresc` *hyp*).
- `sigt_elim` *hyp* élimine le `sigt` dans l'hypothèse *hyp* de la forme `(sigt U P)` en rajoutant dans les hypothèses une variable `y` qui vérifie `(P y)` (a le même effet que `hypelimresc` *hyp*).

Muni de ces tactiques, notre exemple devient :

Tout d'abord les théorèmes et définitions utilisés dans l'exemple :

```
Variables U,U':Type.
Variable R:(Relation U U').

Global restr:(Set U)->(Set U')->(Relation U U') =
 [C:(Set U)][D:(Set U')][x:U][y:U']((C x)/\(D y))/\(R x y).

Discharge R.

Variable A:(Set U).
Variable B:(Set U').
Variable f:(Relation U U').

Definition Rel (x:U)(y:U')(f x y)->((A x)/\(B y)).
Definition au_plus_une_im (x:U)(y,z:U')(f x y)->(f x z)->(eqpt U' y z).
Definition au_moins_une_im ((x:U)(A x)->(sigt U' (f x))).

Definition fonct_part (Rel U U' A B f) /\ au_plus_une_im.
Definition fonction           fonct_part /\ au_moins_une_im.

Theorem fonct2_elim (fonction U U' A B f)->(C:Prop)
    ((Rel U U' A B f)->(au_plus_une_im U U' f)->(au_moins_une_im U U' A f)->C)->C

Theorem fonct_proj2_elim
    (fonction U U' A B f)->(x:U)(y:U')(f x y)->(B y)

Discharge U.
```

Ensuite, la session de la synthèse qui démontre le théoreme $((C \subset A) \wedge (f : A \to B)) \Rightarrow (f_{|C} : C \to B)$.

```
#goal <<(C:(Set U))(inclus U C A)->(fonction U U' A B f)->(fonction U U' C B (restr U U' f C
<<......(C:(Set U))(inclus U C A)->(fonction U U' A B f)->(fonction U U' C B (restr U U' f C
 : message

#Intros();;
<<1 subgoal
  1......(fonction U U' C B (restr U U' f C B))
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (hypelimresc_with "H0" "fonction2_elim");;
<<1 subgoal
  1......(fonction U U' C B (restr U U' f C B))
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
 : message

#by (introdef "fonction2");;
<<3 subgoals
  1......(Rel U U' C B (restr U U' f C B))
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  2......(au_plus_une_im U U' (restr U U' f C B))
  3......(au_moins_une_im U U' C (restr U U' f C B))>>
 : message

#by (introdefc "Rel");;
<<3 subgoals
  1......(C x)/\(B y)
  H4 : (restr U U' f C B x y)
  y : U'
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
```

```
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  2......(au_plus_une_im U U' (restr U U' f C B))
  3......(au_moins_une_im U U' C (restr U U' f C B))>>
 : message

#by (hypelimresc "H4");;
(C x)/\(B y)est dans les hypotheses
<<2 subgoals
  1......(au_plus_une_im U U' (restr U U' f C B))
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  2......(au_moins_une_im U U' C (restr U U' f C B))>>
 : message

#by (introdefc "au_plus_une_im");;
<<2 subgoals
  1......(eqpt U' y z)
  H5 : (restr U U' f C B x z)
  H4 : (restr U U' f C B x y)
  z : U'
  y : U'
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  2......(au_moins_une_im U U' C (restr U U' f C B))>>
 : message

#by (hypelimrescn "H3" [<<x>>]);;
<<3 subgoals
  1......(f x y)
  H3' : (x0:U)(y0:U')(z0:U')(f x0 y0)->(f x0 z0)->(eqpt U' y0 z0)
  H5 : (restr U U' f C B x z)
  H4 : (restr U U' f C B x y)
```

```
  z : U'
  y : U'
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  2......(f x z)
  3......(au_moins_une_im U U' C (restr U U' f C B))>>
 : message

#by (hypelimresc "H4");;
(f x y)est dans les hypotheses
<<2 subgoals
  1......(f x z)
  H3' : (x0:U)(y0:U')(z0:U')(f x0 y0)->(f x0 z0)->(eqpt U' y0 z0)
  H5 : (restr U U' f C B x z)
  H4 : (restr U U' f C B x y)
  z : U'
  y : U'
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  2......(au_moins_une_im U U' C (restr U U' f C B))>>
 : message

#by (hypelimresc "H5");;
(f x z)est dans les hypotheses
<<1 subgoal
  1......(au_moins_une_im U U' C (restr U U' f C B))
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message
```

```
#by (introdefc "au_moins_une_im");;
<<1 subgoal
  1......(sigt U' (restr U U' f C B x))
  H4 : (C x)
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (hypelimrescn "H2" [<<x>>]);;
<<2 subgoals
  1......(A x)
  H4 : (C x)
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)
  2......(sigt U' (restr U U' f C B x))>>
 : message

#by (hypelimresc "H");;
(C x)est dans les hypotheses
<<1 subgoal
  1......(sigt U' (restr U U' f C B x))
  H5 : (f x x0)
  x0 : U'
  H4 : (C x)
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (sigt_intro <<x0>>);;
```

```
<<1 subgoal
  1......(restr U U' f C B x x0)
  H5 : (f x x0)
  x0 : U'
  H4 : (C x)
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (introdefc "restr");;
(f x x0)est dans les hypotheses
<<1 subgoal
  1......(C x)/\(B x0)
  H5 : (f x x0)
  x0 : U'
  H4 : (C x)
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
 : message

#by (and_intro);;
(C x)est dans les hypotheses
<<1 subgoal
  1......(B x0)
  H5 : (f x x0)
  x0 : U'
  H4 : (C x)
  x : U
  H3 : (au_plus_une_im U U' f)
  H2 : (au_moins_une_im U U' A f)
  H1 : (Rel U U' A B f)
  H0 : (fonction U U' A B f)
  H : (inclus U C A)
  C : (Set U)>>
```

```
 : message

#by (hypelimrescn_with "H0" "fonct_proj2_elim" [<<x>>]);;
(f x x0)est dans les hypotheses
<<goal proved>> : message

#save_thm"restriction";;
```

On obtient donc une preuve qui ne se perd pas dans les trivialités. Un inconvénient cependant de ces fonctions, notamment lorsqu'il y a résolution avec variables cachées, c'est qu'il faut avoir à portée de main la formulation exacte des théorèmes utilisés. Notons qu'une utilisation plus interactive de la synthèse pourrait éviter cela, en rappelant la formulation d'un théorème à la demande, et en indiquant à l'utilisateur lorsqu'il y a des variables cachées à instancier.

Chapitre 4

Conclusion

4.1 Le codage du théorème de Schroeder-Bernstein

Le codage constructif de la théorie des ensembles semble donc prometteur, possédant l'avantage sur la théorie axiomatique des ensembles de n'introduire que le seul axiome d'extensionnalité (et l'axiome du tiers-exclu pour certaines applications qui le nécessitent).

Il est intéressant de noter que contrairement à la théorie axiomatique des ensembles où tout baigne dans l'Univers, les différents objets (ensembles, individuels, relations,...) sont fondamentalement séparés par un typage différents. En revanche, les univers d'objets de même nature sont typés de la même façon, ce qui permet de combiner les différents concepts entre eux.

Contrairement à la théorie axiomatique des ensembles qui montre l'existence des objets ensemblistes par des axiomes (axiome de séparabilité, de paire, de somme, de produit,...), le calcul des constructions la montre par le fait même de construire l'objet. Ne pas introduire d'axiome a l'avantage de ramener la cohérence de la théorie à celle du calcul.

Rappelons enfin que la partie de la théorie des ensembles développée ici, ne représente que la base de la théorie.

4.2 L'usage des tactiques

En démontrant le théorème de Schroeder-Bernstein , nous avons été amené à utiliser deux tactiques `Introdef` et `Hypelimres` qui nous semblent bien utiles pour mener une démonstration mathématique. La première intervient lorsqu'on revient à la définition d'une propriété pour la prouver. L'autre intervient lorsqu'on veut utiliser l'information que recèle une hypothèse. A la base de ces tactiques se trouve la résolution sur le but (avec variables cachées) et la résolution entre deux hypothèses. L'implémentation de ces deux résolutions est fondamentale pour construire d'autres tactiques, notamment le `Introdef` et le `Hypelim`.

D'autres développements seraient aussi souhaitables. Tout d'abord une plus grande communication entre la machine et l'utilisateur, que celui-ci puisse avoir accès facilement à la formulation exacte d'un théorème, qu'il puisse dire quelles hypothèses il souhaite voir s'afficher, qu'il puisse dire si lors d'utilisation d'un théorème nécessitant une instanciation de variables cachées, si ces instanciations sont évidentes ou non.

L'option `c` qui met en forme clause de Horn et élimine les buts évidents, nous semble satisfaisante, à la condition qu'un compte-rendu de son action soit donné à l'utilisateur. Elle peut même s'enrichir pour démontrer toutes les trivialités.

Un autre confort agréable est celui apporté par les abréviations.

Pour ce qui est de l'implémentation du `Introdef` et du `Hypelimres`, il est à revoir afin d'éviter les β-expansions inutiles, utilisant selon les cas les théorèmes engendrés par le `Inductive` du vernaculaire (qui devrait se substituer au `make_intro` et `make_elim`), et des `unfold`.

Bibliographie

[1] Gérard Huet. *The constructive engine, Version 4.5* Presses de l'INRIA.

[2] Patrick Suppes. *Axiomatic Set Theory.* The university series in undergraduate mathematics.

[3] Thierry Coquand et Gérard Huet. *Concepts mathématiques et informatiques formalisés dans le calcul des constructions.* Logic Colloquium 85. Studies in logic and the foundations of mathématics, Vol 122, p 123.