GLEF ATINF, un cadre générique pour la connexion d'outils d'inférence et l'édition graphique de preuves

Michel Herment

THÈSE

présentée par

Michel HERMENT

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(Arrêté ministériel du 30 mars 1992)

en INFORMATIQUE

$GLEF_{ATINF}$, un Cadre Générique pour la Connexion d'Outils d'Inférence et l'Édition Graphique de Preuves

Thèse soutenue le 22 juin 1994 devant la commission d'examen

Composition du jury:

Ricardo Caferra	directeur
Joëlle Despeyroux	rapporteur
Claude Kirchner	examinateur
Hans-Jürgen Ohlbach	rapporteur
Jean-Pierre Verjus	président

Thèse préparée au sein du Laboratoire d'Informatique Fondamentale
et d'Intelligence Artificielle

Remerciements

Je dois tout d'abord remercier Jean-Pierre Verjus, qui a bien voulu nous faire l'honneur de présider le jury.

Je tiens ensuite à remercier Joëlle Despeyroux et Hans-Jürgen Ohlbach, qui ont accepté les contraintes du rôle difficile de rapporteur, et ce malgré des délais relativement courts, apportant ainsi à cette thèse la garantie de leur grande compétence scientifique. Hans-Jürgen Ohlbach s'est de plus astreint à lire le français, langue qui n'est pas la sienne.

Je remercie particulièrement Claude Kirchner qui a eu la gentillesse, en acceptant de faire partie du jury, d'apporter à ce travail le soutient de sa renommée.

Je remercie également les membres du projet ATINF, notamment Thierry Boy de la Tour, qui m'a aidé et soutenu dans ce travail, grâce à sa disponibilité et à sa compétence scientifique.

Je veux aussi remercier Philippe Jorrand, pour avoir su faire du laboratoire qu'il dirige un environnement agréable et efficace, condition indispensable à tout travail de recherche.

Je tiens enfin à remercier Ricardo Caferra, qui s'est révélé un directeur de thèse exceptionnel. Sans jamais rien m'imposer, il a pris le temps de me faire partager sa large connaissance de la littérature et sa vue d'ensemble du sujet, de m'apporter des idées fécondes et des points de vue critiques, et de guider mes premiers pas. Mais je lui suis surtout reconnaissant du soutien moral qu'il m'a apporté tout au long de ce travail et je sais avoir trouvé un ami sincère.

I. Table des matières

II. Présentation

A) L'atelier d'inférence ATINF: motivation, description, état actuel

La déduction automatique a aujourd'hui plus de 40 ans. On peut dire qu'elle est arrivée à maturité et il est possible d'en donner une description abstraite et générale. [Dav83] et [BL84] en présentent une histoire claire et complète jusqu'en 1984. [Wos88] donne une idée assez précise des problèmes ouverts dans l'approche que nous appellerons "classique". Dès la fin des années 70 et le début des années 80, une approche beaucoup plus générale commence à se préciser, parallèlement à l'approche classique. C'est l'approche des *démonstrateurs programmable* fondés sur les notions de *tactiques* ou de *stratégies* exprimées dans un méta-langage. LCF et ML en sont les premiers exemples publiés [GMW79]. Cette approche s'est beaucoup développée, elle est actuellement connue sous le nom de *cadres logiques* ("logical frameworks"). Au lieu d'étudier et d'implémenter des calculs particuliers pour chaque logique elle propose d'employer des sortes de méta-logiques implémentables dans lesquelles toutes les logiques usuelles sont représentables. AUTOMATH [dBr80], Nuprl [C+86] et Coq [D+91] en sont peut-être les exemples les plus significatifs.

On peut considérer que les cadres logiques correspondent aux logiques générales ("general logics") étudiées par les logiciens [Ebb85], [Mes87], [Avr91]. En ce qui concerne cette remarque, il est intéressant de citer [Ebb85] page 26:

> "What is a logic? The answer to this question is a *pragmatic*[1] one: we collect some basic features common to well-known logical systems and use them as defining properties of a logic...In order to escape this dilemma we do not fix a single definition, but leave it to the working logician to choose a suitable notion according to the needs of specific situations..."

Au début des années 80 on pouvait déjà faire une classification des différentes approches en 3 grandes catégories. Cette classification, toujours valable, est faite selon que l'accent est mis sur:

[1] Les italiques sont à nous.

1- L'*efficacité* (approche classique): c'est à dire sur la recherche de bons calculs, bonnes stratégies (parfois heuristiques), bonnes implémentations, etc..

2- La *généralité*: cette approche cherche à éviter le foisonnement de démonstrateurs pour des logiques et calculs différents. C'est l'approche des cadres logiques.

3- La *facilité d'interaction*: l'une des raisons de cette recherche est certainement que l'augmentation du pouvoir des démonstrateurs permettait de s'attaquer de façon automatique (interactive) à des théorèmes difficiles. Dans cette catégorie on peut encore diviser les travaux en deux sous-catégories. La première concerne la *traduction* entre calculs différents pour la même logique (voir par exemple [And80], [And91]). Cette traduction a aussi été tentée entre une logique et une langue naturelle ([Che76], [Hua90]). L'autre sous-catégorie concerne la *présentation structurée* des preuves [Lin90], [LP90], [CHZ91], [CH93], [CH9?].

Les travaux dans la troisième catégorie sont de loin les moins nombreux, mais le besoin d'outils puissants pour la manipulation des preuves est de plus en plus évident.

L'objectif du projet ATINF (ATelier d'INFérence), démarré en 1985, est d'essayer de combiner les avantages des 3 approches et de formaliser, pour les incorporer, des techniques puissantes du raisonnement humain (telles que la recherche simultanée de preuves et contre-exemples, l'utilisation d'analogies, le raisonnement dans des théories, etc.). Il devait être aussi un cadre permettant d'incorporer facilement des démonstrateurs et des outils de calcul formel développés ailleurs.

Les raisons pour initier un tel projet sont très naturelles. On était arrivé à une situation où la reprogrammation systématique d'algorithmes d'unification, de transformateurs en forme clausale, de démonstrateurs, etc. (la plupart étant rapidement abandonnés parce que trop inefficaces ou trop primitifs) était un gaspillage d'énergie manifeste. De plus, les structures de données et les entrées des différents démonstrateurs et outils associés (dont certains sont très efficaces) étant incompatibles, ils ne pouvaient pas coopérer. Notons aussi que des outils très généraux comme Nuprl n'étaient (ne sont) pas des démonstrateurs.

Concernant son implémentation, le projet ATINF a donc été conçu comme un ensemble de modules indépendants qui *communiquent selon des protocoles* communs. Cette stratégie (qui, pour des raisons matérielles était la seule que nous pouvions appliquer) s'est avérée très réaliste pour faire coopérer les très nombreux (et parfois excellents) logiciels d'inférence développés *indépendamment* de par le monde.

Ainsi, un tel atelier d'inférence devait permettre la connexion d'outils d'inférence très diversifiés ce qui imposait l'utilisation d'un formalisme très général pour la communication et la présentation des preuves, formules, etc.. $GLEF_{ATINF}$ est la solution que nous avons proposée pour l'intégration.

B) Les bases de la conception

$GLEF_{ATINF}$ respecte la philosophie d'ATINF, dont la spécification informelle est résumée ci-dessous. Cette spécification informelle tente ainsi de regrouper les caractéristiques que devrait avoir un ensemble d'outils d'inférence (formels) *orienté utilisateur.*

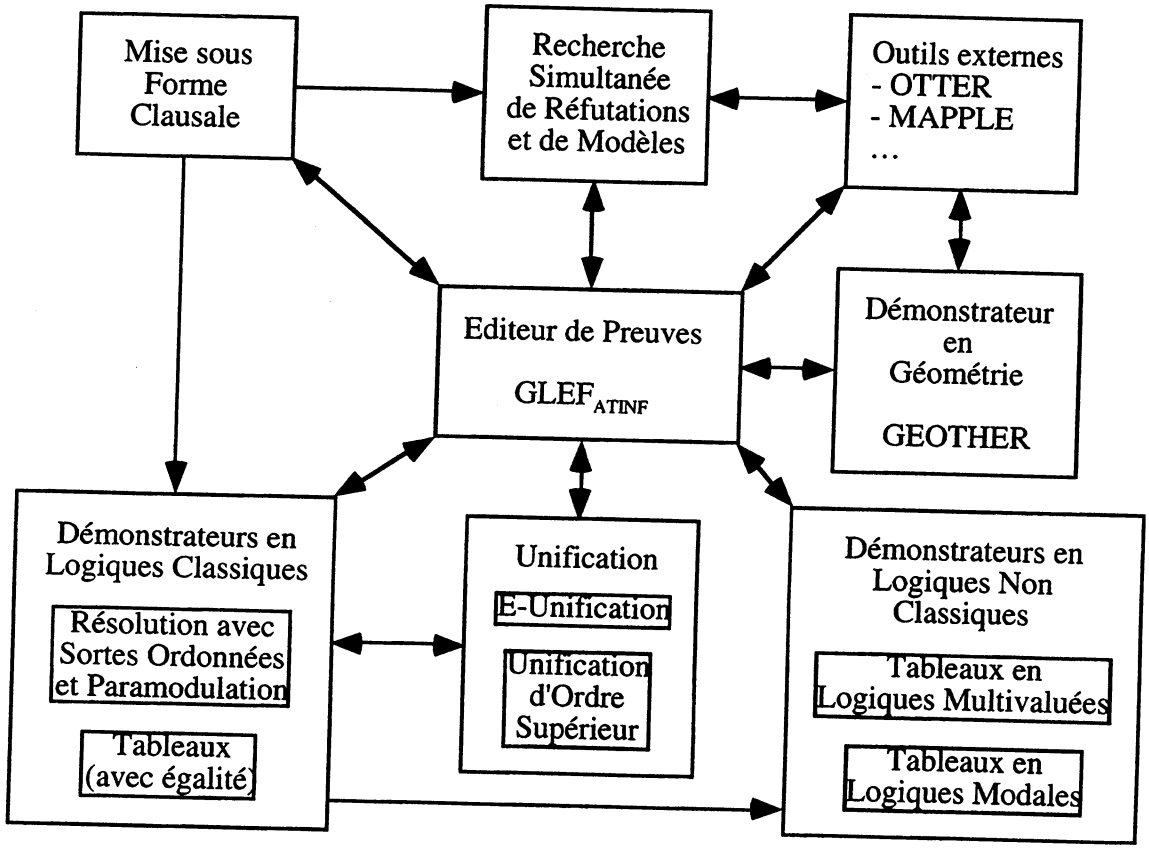
ATINF doit:

- être extensible.
- demander le moins d'expertise possible de l'utilisateur (qui ne doit pas être considéré comme un programmeur). Les preuves doivent donc être présentées d'une façon la plus proche possible de la présentation usuelle.
- permettre de définir la classe la plus large possible de logiques et de calculs.
- éditer et si possible vérifier des preuves dans les logiques définies.
- utiliser un formalisme permettant une définition "incrémentale" des logiques, c'est à dire les caractéristiques communes des différentes logiques doivent être définies une seule fois et réutilisées.
- permettre de présenter des preuves obtenues par des démonstrateurs extérieurs à ATINF.
- permettre d'incorporer dans le même cadre unifié, des systèmes de manipulation symbolique.
- permettre différentes représentations hiérarchiques et graphiques pour une même preuve. La présentation d'une preuve doit être indépendante de la façon dont la preuve a été obtenue.
- être capable d'exhiber et de cacher une partie quelconque d'une preuve, formule, etc..
- donner à l'utilisateur la possibilité d'utiliser des outils tels que le filtrage et l'unification (premier ordre et ordre supérieur), de transformation de formules, indépendamment d'un démonstrateur particulier.
- être capable de mémoriser et de modifier des parties quelconques d'une preuve.

- donner à l'utilisateur la possibilité de nommer et manipuler tout objet ((sous-) preuve, formule, symboles, etc.) de façon homogène.
- permettre de grouper plusieurs pas d'une preuve en pas de taille arbitraire.

Il est clair que le troisième et quatrième point ci-dessus sont des exigences auxquelles on essaye de se rapprocher le plus possible sans perdre les capacités de preuve, mais elles ne peuvent être réellement satisfaites que par des outils très évolués comme Coq.

Le dessin ci-dessous donne l'ensemble des outils actuellement dans ATINF:



Finalement nous devons signaler que la manipulation de "grosses preuves" est un problème capital sur lequel converge la communauté de chercheurs en déduction automatique et celle des mathématiciens se servant d'outils d'inférence (voir par ex. [Lam90], [Lam91], [Hor93]).

III. Polices de caractères

Décrivons ici la signification des différentes polices de caractères et attributs typographiques utilisés dans ce mémoire de thèse.

- Le **Gras** indique, en dehors des titres, un point important.
- Les *Italiques* indiquent un terme qui apparaît pour la première fois.
- Le Souligné indique, en dehors des titres, une référence à un chapitre ou à une annexe.
- La police **Chicago** indique un choix de menu ou un bouton de GLEF (voir chapitre Utilisation section A).
- La police Palatino indique un nom (parfois décliné dans le texte) de manipulations élémentaires (voir Chapitre Réalisation section B.8.a).
- La police Monaco est employée pour les programmes.
- La police Times est employée dans les formules. *Times Italique* indique une variable.

IV. Définition

A) Notions Préliminaires

Cette partie introduit, d'abord de manière intuitive, puis de manière plus rigoureuse, les notions que nous devons traiter dans notre système (logique, preuve, etc.). Moins concise et moins précise, mais peut-être moins abstraite et plus proche de notre implémentation qu'une formulation utilisant les catégories, nous préférons ici une formulation ensembliste.

Notions intuitives

La **logique** est l'étude de la **conséquence** (théorie des preuves). La **conséquence** est une relation entre des **jugements**.

Un **jugement** est un fait qui peut être vérifié ou non. Le plus souvent, un jugement exprime le fait qu'un objet a une certaine propriété. Par exemple, qu'une formule soit un théorème. Cette propriété s'appelle **forme de jugement de base**, et ce type de jugement s'appelle **jugement de base**.

Une **preuve** est un objet qui permet de persuader quelqu'un d'un jugement. En pratique cette définition laisse trop de liberté. Le plus souvent, aussi bien en mathématiques que dans la vie courante, une preuve (ou déduction) est une construction faite à l'aide de **raisonnements** admis, à partir de **jugements** admis.

Un **raisonnement** est une relation entre un certain nombre de jugements, c'est d'ailleurs une forme particulière de jugement. L'un de ces jugements est appelé **conclusion**, les autres sont appelés **prémisses**.

Une **règle d'inférence** est un raisonnement admis. Un **axiome** est un jugement admis, il peut être considéré comme une règle d'inférence sans prémisses. Une **hypothèse** est un jugement admis provisoirement.

Par exemple, considérons le modus-ponens en logique propositionnelle (A et B sont des formules quelconques):

$$\frac{A \quad A \Rightarrow B}{B}$$

"être vrai" est une forme de jugement de base. "A est vrai" et "$A \Rightarrow B$ est vrai" sont les jugements de base prémisses de la règle d'inférence. "B est

vrai" est son jugement de base conclusion. Supposer une proposition P revient à considérer le jugement de base "P est vrai" comme hypothèse.

Notations

Un ensemble est délimité par des accolades, un *multi-ensemble* est délimité par des crochets. Une liste (n-uplet) est délimitée par des parenthèses. Ces symboles peuvent être omis, quand aucune ambiguïté n'est possible. Les ensembles et les multi-ensembles peuvent être décrits en extension ou en compréhension.

On note $\varnothing$ l'ensemble vide et $\dot{\varnothing}$ le multi-ensemble vide. On note $\mathrm{P}(E)$ l'ensemble des sous-ensembles d'un ensemble E et $\mathrm{M}(E)$ l'ensemble de ses sous-multi-ensembles.

Parfois, la virgule dénotera l'union d'ensembles ou de multi-ensembles.

Notion de logique

Un *langage* est un ensemble dont les éléments sont appelés *formules*.

En théorie des preuves, une *relation de conséquence* sur un langage Σ est une relation $\vdash$ entre des multi-ensembles de formules telle que (A dénote une formule, Γ et Δ dénotent des multi-ensembles de formules):

(i) $A \vdash A$ — *réflexivité restreinte*[1] [Avr91]

(ii) $\dfrac{\Gamma_1 \vdash \Delta_1, A \quad A, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}$ — *transitivité (ou règle de coupure)*[2]

Un élément[3] d'une relation de conséquence s'appelle *séquent*.

Cette définition de relation de conséquence a été choisie la plus générale possible (dans la limite du raisonnable, voir citation d'Ebbinghaus dans le chapitre Présentation), on n'impose aucune condition (récursif, récursivement énumérable, etc.) sur Σ et $\vdash$ [Gab91]. La restriction aux multi-ensembles finis correspond à la définition de

---

[1] Par rapport à la réflexivité définie par Scott (i)' définie plus loin.

[2] Le schéma $\dfrac{\varphi_1 \ldots \varphi_n}{\varphi}$, où $\varphi_1, \ldots, \varphi_n, \varphi$ dénotent des propositions, dénote la proposition si φ_1... et φ_n alors φ. Les pointillés servent à différencier cette notation de celle de règle d'inférence, tout en gardant une notation graphique. Notons que $\underline{\underline{\varphi}}$ dénote la proposition φ. (Dans [Sco74], ce schéma de proposition est noté avec une barre simple et les règles d'inférences sont notées avec une double barre).

[3] On peut voir une relation comme une partie d'un produit cartésien.

[Avr91]. Soulignons toutefois les différences avec d'autres définitions de relation de conséquence.

- Contrairement à la relation de conséquence de Tarski (1938) et conformément à celle de Scott (1969) [Sco74], on admet plusieurs conclusions.
- L'une des conditions supplémentaires suivantes, classées ici de la plus forte vers la plus faible, est parfois ajoutée:

(iii) $\dfrac{\Gamma_1 \vdash \Delta_1}{\Gamma_1,\Gamma_2 \vdash \Delta_1,\Delta_2}$ *monotonie* [Sco74]

(iii)' $\dfrac{\Gamma_1 \vdash \Delta}{\Gamma_1,\Gamma_2 \vdash \Delta}$ *affaiblissement* [Gab91], [Avr91]

(iii)'' $\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma, A \vdash B}$ *affaiblissement restreint* [Gab91]

Toutefois, des logiques usuelles (appelées *logiques non monotones*) ne satisfont pas la condition (iii), ni même la condition (iii)' ou (iii)'' (la notion de logique est définie plus loin).

- De même, la condition (i) est souvent remplacée par une condition plus forte:

(i)' $\dfrac{\Gamma \cap \Delta \neq \dot{\varnothing}}{\Gamma \vdash \Delta}$ *réflexivité* [Sco74]

Mais cette condition n'est pas satisfaite par les logiques relevantes ou linéaires.

- Notons que dans [Mes87], une relation de conséquence doit satisfaire les propriétés (iii) et (i)', des relations de conséquences non monotones ou non réflexives sont **considérées comme des calculs** pour des relations de conséquence qui le sont (voir plus loin la définition de calcul).
- Il est plus courant de considérer une relation entre ensembles qu'une relation entre multi-ensembles. Toutefois, les logiques relevantes, linéaires et les logiques de Lukasiewicz à nombre de valeurs finies nécessitent l'emploi de multi-ensembles. Quelques rares logiques amènent à considérer des structures plus complexes (listes [Avr91], ensembles partiellement ordonnés, etc. [Gab91]). Mais cela complique la condition de transitivité: on trouve la notion de *transitivité substitutionnelle* dans [Gab91].

Si on considérait une structure de liste, la règle d'échange permettrait de simuler la structure de multi-ensemble et la règle de contraction permettrait de simuler celle d'ensemble (ces règles sont données ici pour une relation de conséquence mono-conclusion) [Lam93a]:

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}$$ *échange* [Lam93a]

$$\frac{\Gamma, A, A, \Delta \vdash C}{\Gamma, A, \Delta \vdash C}$$ *contraction* [Lam93a]

- Quand on utilise des variables schématiques, on suppose la relation de conséquence *uniforme*, c'est à dire fermée par substitution [Avr91]. Mais cette propriété dépend de la structure des formules et complique l'expression des propriétés.
- Une relation de conséquence est dite *compacte* [Avr91], [Ebb85], ssi quand on a $\Gamma \vdash \Delta$, on peut trouver des ensembles finis de formules $\Gamma_0 \subset \Gamma$ et $\Delta_0 \subset \Delta$ tels que $\Gamma_0 \vdash \Delta_0$. Contrairement à l'usage, nous n'imposons pas la compacité des relations de conséquence. Une relation de conséquence peut même faire intervenir des ensembles qu'on ne puisse pas réduire ainsi à des ensembles récursifs, ou récursivement énumérables. Bien entendu, un calcul pour une relation de conséquence peut ne pas être complet (voir plus loin les définitions de calcul et de calcul complet).

Si Δ est un multi-ensemble de formules, $\vdash \Delta$ et $\Delta \vdash$ dénotent respectivement $\dot{\varnothing} \vdash \Delta$ et $\Delta \vdash \dot{\varnothing}$. Une relation de conséquence est *consistante* [Sco74] ssi pour toute formule A on a:

$$\nvdash A \text{ ou } A \nvdash$$

On de façon plus concise (d'après (ii) et (iii)): $\nvdash$.

Elle est *complète* [Sco74] ssi pour toute formule A on a:

$$\vdash A \text{ ou } A \vdash$$

En théorie des modèles, une *relation de satisfaction* sur un ensemble S, dont les éléments sont appelés *structures*, et un langage Σ, est une relation $\models$ entre une structure et une formule. Une formule φ est *valide* dans S ssi $\forall s \in S \; s \models \varphi$, on le note $S \models \varphi$.

Exemples [FHV92]:

- En logique du premier ordre, on définit la relation de satisfaction par rapport aux structures (D,I,w), où est D un univers, I est une interprétation des prédicats et des fonctions sur l'univers et w est une valuation des variables sur D.
- En logique modale, on définit la relation de satisfaction par rapport aux structures (W,R,π,w), où W est un ensemble de mondes, R est une relation d'accessibilité, π définit la vérité dans chaque monde et w est un monde particulier.

Notons que [Ebb85] propose de représenter les structures par des algèbres. Une signature τ définit une structure algébrique $S(\tau)$. Une logique est alors un couple $(L,\models_L)$, où L associe un langage à une signature, et $\models_L$ associe une relation de satisfaction $\models_L(\tau)$ sur $S(\tau)$ et $L(\tau)$ quand τ est une signature.

A toute relation de satisfaction $\models$, on peut associer les relations $\vdash_t$ et $\vdash_v$ définies par:

$$\sigma \vdash_t \varphi \quad \text{ssi} \quad \forall s \in S \; s \models \sigma \Rightarrow s \models \varphi$$
$$\sigma \vdash_v \varphi \quad \text{ssi} \quad S \models \sigma \Rightarrow S \models \varphi$$

Notons que [FHV92] impose en plus l'uniformité de ces relations en supposant qu'une notion de substitution soit définie pour le langage considéré.

[FHV92] propose aussi de mettre S sous la forme $\{(m,w) \mid m \in M \wedge w \in W_m\}$. Chaque élément de $m \in M$ représente une classe d'équivalence de S, W_m étant alors en correspondance avec celle classe. Pour chaque M, on pose:

$$m \models \varphi \quad \text{ssi} \quad \forall w \in W_m \; (m,w) \models \varphi$$
$$\sigma \vdash_M \varphi \quad \text{ssi} \quad \forall m \in M \; m \models \sigma \Rightarrow m \models \varphi$$

On peut étendre naturellement les relations $\vdash_t$, $\vdash_v$ et $\vdash_M$ aux multi-ensembles de formules:

$$\Gamma \vdash_t \Delta \quad \text{ssi} \quad \forall s \in S \bigwedge_{\sigma \in \Gamma} s \models \sigma \Rightarrow \bigvee_{\varphi \in \Delta} s \models \varphi$$
$$\Gamma \vdash_M \Delta \quad \text{ssi} \quad \forall m \in M \bigwedge_{\sigma \in \Gamma} m \models \sigma \Rightarrow \bigvee_{\varphi \in \Delta} m \models \varphi$$
$$\Gamma \vdash_v \Delta \quad \text{ssi} \quad \bigwedge_{\sigma \in \Gamma} S \models \sigma \Rightarrow \bigvee_{\varphi \in \Delta} S \models \varphi$$

Ensuite, on montre facilement que $\vdash_t \subseteq \vdash_M \subseteq \vdash_v$ et que $\vdash_M$ est une relation de conséquence qui vérifie (iii) et (i)', on l'appelle *M-conséquence*. Si on identifie S et $\{(m,w) \mid m \in S \wedge w \in \{\varepsilon\}\}$ (resp. $\{(m,w) \mid m \in \{\varepsilon\} \wedge w \in S\}$), on

a $\vdash_S = \vdash_t$ (resp. $\vdash_S = \vdash_v$). $\vdash_t$ et $\vdash_v$ sont donc aussi des relations de conséquence vérifiant (iii) et (i)', on les appelle relations de conséquence de *vérité* et de *validité*.

Exemples [FHV92]:

- En logique du premier ordre, on montre la correction de la règle d'inférence (les notions de règle d'inférence et de correction d'une règle d'inférence par rapport à une relation de conséquence sont définies plus loin) de généralisation par rapport à la validité ($\varphi \vdash_v \forall x\, \varphi$) en montrant sa correction par rapport à la M-conséquence, où $W_{(D,I)}$ est l'ensemble des valuations sur D et $M = \{(D,I) \mid \exists w \in W_{(D,I)}\ (D,I,w) \in S\}$.
- En logique modale, on montre la correction de la règle d'inférence de nécessité par rapport à la validité ($\varphi \vdash_v \Box \varphi$) en montrant sa correction par rapport à la M-conséquence, où $M = \{(W,R,\pi) \mid \exists w \in W\ (W,R,\pi,w) \in S\}$ et $W_{(W,R,\pi)}$ est l'ensemble des mondes W.

Ainsi, nous avons vu différentes formes de relations de conséquence. Certaines sont purement syntaxiques, d'autres sont issues d'une approche sémantique. Notons que de nombreuses relations de conséquence issues d'une approche sémantique ne sont pas compactes [Avr91].

Tenant compte de ces considérations, nous adoptons la définition formelle de logique suivante:

Une *logique* est un couple $L=(\Sigma,CR)$ où Σ est un langage et CR est un ensemble **fini** de relations de conséquence sur Σ.

La plupart du temps, la structure d'un langage peut être caractérisée à l'aide de relations de conséquence. Par exemple, les formules bien formées du langage propositionnel peuvent être vues comme les théorèmes du système axiomatique dont les formules bien formées sont les chaînes de symboles, les axiomes sont les variables propositionnelles et les règles d'inférence sont les règles usuelles de formation [Avr91]. Ainsi, on peut considérer des logiques dont **une partie de la structure du langage a ainsi été transférée vers l'ensemble des relations de conséquence.**

Le but des théorèmes suivants est de montrer qu'en augmentant le langage, on peut remplacer l'ensemble des relations de conséquence d'une logique par une seule relation de conséquence. Étant donnée une

logique $L=(\Sigma,CR)$, on introduit un nouveau symbole $j_{\vdash}$ pour chaque relation de conséquence $\vdash$ de CR. On pose ensuite:

$$\Sigma' = \{ j_{\vdash}\ \varphi \mid \vdash \in CR \wedge \varphi \in \Sigma \}$$

Σ' s'appelle l'ensemble des *jugements de base* de la logique. Le symbole $j_{\vdash}$ s'appelle la *forme* du jugement de base $\vdash \varphi$ [HHP89].

Théorème: Etant donnée une logique $L=(\Sigma,CR)$. Soit $\Sigma' = \{ j_{\vdash}\ \varphi \mid \vdash \in CR \wedge \varphi \in \Sigma \}$ l'ensemble des jugements de base de L, et
$\vdash' = \{ ([j_{\vdash}\ \varphi \mid \varphi \in \Gamma],[j_{\vdash}\ \psi \mid \psi \in \Delta]) \mid \vdash \in CR \wedge \Gamma,\Delta \in M(\Sigma) \wedge \Gamma \vdash \Delta \}$.
$\vdash'$ est une relation de conséquence sur Σ'.

Preuve:

Réflexivité:

Soit $A \in \Sigma'$, alors A est de la forme $j_{\vdash}\ \varphi_A$ où $\vdash \in CR$ et $\varphi_A \in \Sigma$
par réflexivité de $\vdash$, on a $\varphi_A \vdash \varphi_A$
par définition de $\vdash'$, $(j_{\vdash}\ \varphi_A) \vdash' (j_{\vdash}\ \varphi_A)$
c'est à dire $A \vdash' A$
□

Transitivité:

Soit $A \in \Sigma'$ et $\Gamma_1,\Gamma_2,\Delta_1,\Delta_2 \in M(\Sigma')$
supposons $\Gamma_1 \vdash' \Delta_1, A$ et $A, \Gamma_2 \vdash' \Delta_2$
alors A est de la forme $j_{\vdash}\ \varphi_A$ où $\vdash \in CR$ et $\varphi_A \in \Sigma$

on pose
$$\begin{cases} \Gamma_1' = [\varphi | j_{\vdash}\ \varphi \in \Gamma_1] & \Gamma_2' = [\varphi | j_{\vdash}\ \varphi \in \Gamma_2] \\ \Delta_1' = [\varphi | j_{\vdash}\ \varphi \in \Delta_1] & \Delta_2' = [\varphi | j_{\vdash}\ \varphi \in \Delta_2] \end{cases}$$

d'où
$$\begin{cases} [j_{\vdash}\ \varphi | \varphi \in \Gamma_1'] \vdash' [j_{\vdash}\ \varphi | \varphi \in \Delta_1'], j_{\vdash}\ \varphi_A \\ j_{\vdash}\ \varphi_A, [j_{\vdash}\ \varphi | \varphi \in \Gamma_2'] \vdash' [j_{\vdash}\ \varphi | \varphi \in \Delta_2'] \end{cases}$$
$$\begin{cases} \Gamma_1' \vdash \Delta_1', \varphi_A \\ \varphi_A, \Gamma_2' \vdash \Delta_2' \end{cases}$$
$$\Gamma_1', \Gamma_2' \vdash \Delta_1', \Delta_2'$$
$$[j_{\vdash}\ \varphi | \varphi \in \Gamma_1', \Gamma_2'] \vdash' [j_{\vdash}\ \varphi | \varphi \in \Delta_1', \Delta_2']$$
$$\Gamma_1, \Gamma_2 \vdash' \Delta_1, \Delta_2$$

□

Théorème: Soit $\begin{cases} J = \{j \mid \forall \varphi \in \Sigma\ j\varphi \in \Sigma'\} \\ \vdash_j = \{([\varphi \mid j\varphi \in \Gamma],[\psi \mid j\psi \in \Delta]) \mid \Gamma \vdash' \Delta\} \\ CR' = \{\vdash_j \mid j \in J\} \end{cases}$

si $\Sigma \neq \varnothing$ alors $CR' = CR$.

Preuve:

On a $\vdash_{j_\vdash} = \{([\varphi \mid j_\vdash\ \varphi \in [j_\vdash\ \varphi \mid \varphi \in \Gamma]],[\psi \mid j_\vdash\ \psi \in [j_\vdash\ \psi \mid \psi \in \Delta]]) \mid \Gamma \vdash \Delta\}$

$\vdash_{j_\vdash} = \{(\Gamma,\Delta) \mid \Gamma \vdash \Delta\}$

$\vdash_{j_\vdash} = \vdash$

d'où $CR' \supset CR$

soit $\vdash' \in CR'$

$\exists j \in J \vdash' = \vdash_j$

comme $\Sigma \neq \varnothing$ $\exists \vdash \in CR \vdash_j = \vdash_{j_\vdash} = \vdash$

d'où $CR' \subset CR$

et $CR' = CR$

□

Corollaire: Toute logique $L = (\Sigma, CR)$ de langage non vide, peut être représentée par une logique à une et une seule relation de conséquence $L' = (\Sigma', \vdash')$. Dans le cas où CR est un singleton, L et L' sont isomorphes.

Nous ne considérerons donc que des logiques à une seule relation de conséquence, sachant que celles ci peuvent être des représentations de logiques à plusieurs relations de conséquence.

Notion de preuve

Étant donné un langage Σ, un *séquent formel* est un couple de multi-ensembles de formules dénoté par $\Gamma \Rightarrow \Delta$ [Avr91]. On note $S(\Sigma)$, l'ensemble des séquents formels. Soit n un entier naturel, une *règle d'inférence d'arité* $n+1\,(n \geq 0)$ est une relation R entre $S(\Sigma)^n$ et $S(\Sigma)$. Le plus souvent, on considère des règles d'inférence décidables, mais considérer des règles d'inférence semi-décidables ne change pas le caractère récursivement énumérable de la classe des théorèmes [McC62].

Une *prémisse* de R est un séquent formel apparaissant dans un ensemble de séquents formels relié par R à un séquent formel. Une *conclusion* de R est un séquent formel relié par R à un ensemble de séquents formels. Un *axiome* est une règle d'inférence d'arité 1 (contrairement à l'usage qui veut qu'un axiome soit une formule).

Étant donné un séquent formel $s{:}(\Gamma \Rightarrow \Delta)$, $\vdash(s)$ dénote le jugement $\Gamma \vdash \Delta$. La notion de correction d'une règle d'inférence établit le rapport entre relation de conséquence et règle d'inférence. Une règle d'inférence R est dite *correcte* pour la relation de conséquence $\vdash$ ssi on a:

$$\forall(s_1,\ldots,s_n,s) \in R \frac{\vdash(s_1) \quad \ldots \quad \vdash(s_n)}{\vdash(s)}$$

Pour la notion de preuve (le mot déduction serait plus adéquat, mais il est moins usuel), nous préférons une représentation **intuitive**, sous forme de graphe à la représentation **usuelle**, plus simple à formaliser, sous forme de liste.

Étant donné un langage Σ et un ensemble RI de règles d'inférence, une *preuve* est un triplet $\pi=(N,E,D)$ où N est un ensemble fini dont les éléments sont appelés nœuds, E est une fonction associant un séquent formel à chaque nœud, D est une relation entre $\mathrm{P}(N)$ et N qui vérifie les deux conditions de correction:

$$\forall(s,n_2) \in D\, \exists R \in RI \left(\left\{E(n_1) \mid n_1 \in s\right\}, E(n_2)\right) \in R \qquad \text{(règles)}$$

$$\neg\exists n_1,\ldots,n_n \in N\, n_1 = n_n \wedge \bigvee_{2<i\leq n} i\, \exists s \in \mathrm{P}(N)\, n_{i-1} \in s \wedge (s,n_i) \in D \quad \text{(cycles)}$$

$\{n \in N \mid \forall s \in \mathrm{P}(N)\,(s,n) \notin D\}$ s'appelle l'ensemble des *hypothèses* de la preuve. $\{n \in N \mid \forall s \in \mathrm{P}(N)\, n \in s \Rightarrow \forall n' \in N\,(s,n') \notin D\}$ s'appelle l'ensemble des *conclusions* de la preuve. Une preuve *complète* (resp. *partielle*) est une preuve dont l'ensemble des hypothèses est vide (resp. non vide). Une preuve *connexe* est une preuve dont l'ensemble des conclusions est un singleton.

La notion de preuve est très générale. Quand on caractérise le langage d'une logique à l'aide de relations de conséquence, elle regroupe un grand nombre d'objets:

constante	fonction	terme
atome	connectif logique	formule
axiome	règle d'inférence	composition de règles d'inférence

En logique du premier ordre, une preuve peut représenter un objet de ce tableau, une variable dénotant un tel objet ou un ensemble de tels objets et variables.

Un éditeur de preuves doit donc permettre d'éditer tous ces objets de manière homogène.

Notion de calcul

Un *calcul*, sur un langage, est un algorithme qui énumère des couples de formules. Il génère ainsi une relation $\Vdash$ [Gab91].

Un *formalisme* est un couple $F=(\Sigma,C)$ où Σ est un langage et C est un calcul sur Σ.

Un *système logique* est un couple $S=(L,C)$ où $L=(\Sigma,\vdash)$ est une logique et C est un calcul sur Σ [Mes87]. Par exemple, la logique propositionnelle munie d'une procédure qui parcourt l'ensemble les formules en déterminant, à l'aide d'une table de vérité, si chaque formule est une tautologie à énumérer, forme un système logique.

C est dit *correct* pour L si $\Vdash \subset \vdash$. Il est dit *complet* pour L si $\vdash \subset \Vdash$ [Sco74], [Mes87], [Avr91].

Un *calcul de preuves* sur un langage Σ est un calcul sur Σ fondé sur les notions d'axiome et de règle d'inférence. C'est à dire, étant donné un ensemble de règles d'inférence, $\Vdash$ est défini par l'ensemble des couples (Γ,Δ) tels qu'il existe une preuve dont l'ensemble des hypothèses est vide et l'ensemble des conclusions est $\{\Gamma \Rightarrow \Delta\}$.

Un *système formel* est un couple $F=(\Sigma,C)$ où Σ est un langage et C est un calcul de preuves sur Σ.

Exemples de représentations de systèmes formels de plus en plus générales [Avr91]:

- systèmes axiomatiques (ou systèmes de Hilbert pour les théorèmes)
- type Hilbert (ce sont des systèmes axiomatiques employés avec la méthode d'extension qui est décrite plus loin)
- déduction naturelle
- type Gentzen

Une *signature* σ est un objet caractérisant un langage noté Σ/σ.

Une *théorie* est un ensemble de formules. Étant donnés un calcul de preuves C et une théorie T, on note $C+T$ le calcul obtenu en ajoutant l'axiome $\left\{\left(\varnothing,\dot{\varnothing} \Rightarrow \varphi\right) \mid \varphi \in T\right\}$ à l'ensemble des règles d'inférence de C.
Remarque: $C+\varnothing=C$.

Certains calculs de preuves, comme les systèmes axiomatiques, ne permettent de prouver que des *théorèmes*, c'est à dire de construire des preuves de séquents formels de la forme $\dot{\varnothing} \Rightarrow \varphi$.

La *méthode d'extension* [Avr91] permet d'employer un tel calcul C pour générer une relation $\Vdash$ de la manière suivante: On a $\Gamma \Vdash \varphi$ si $\emptyset \Vdash_{C+\Gamma} \varphi$ ($\Vdash_{C+\Gamma}$ est la relation générée par $C+\Gamma$, Γ étant considéré comme un ensemble).

Quand la relation de conséquence considérée est à conclusion unique, prise sur des ensembles finis et satisfait l'affaiblissement, cette méthode constitue un calcul correct.

Jugements d'ordre supérieur

Le *jugement hypothétique* $J_1 \succ J_2$ exprime une certaine forme de conséquence: J_2 est prouvable sous l'hypothèse J_1. Étant donné un langage Σ, le *jugement schématique* $\bigwedge_x J(x)$ exprime une certaine forme de généralité: $J(x)$ est uniformément prouvable en x [HHP89]. On considère le nouveau langage Σ' égal à la fermeture de Σ par $\succ$ et $\bigwedge$. Contrairement à [HHP89], on n'impose pas l'appartenance de x à une classe de termes, mais seulement à Σ'. En effet, nous considérons que $\bigwedge_{x \in C} J(x)$ est une représentation abrégée d'un jugement de la forme $\bigwedge_x C(x) \succ J(x)$, ce qui correspond à notre notion de preuve. Toutefois, nous emploierons cette représentation dans le cadre de LF [HHP89] ou CC [CH88].

On considère un système de déduction naturelle uniforme (c'est à dire clos par substitution [Avr91]), pur (voir [Avr91]), à conclusion simple, pris sur des ensembles finis et satisfaisant l'affaiblissement. Sauf pour la règle d'affaiblissement, quand elle est explicite, la forme générale de ses règles d'inférence, qui peut contenir des variables schématiques $(x_1 \ldots x_n)$, est [Avr91]:

$$\frac{\left[\varphi_1^1 \ldots, \varphi_1^{n_1}\right] \quad \ldots \quad \left[\varphi_1^1 \ldots, \varphi_1^{n_n}\right]}{\begin{array}{cc} \psi_1 & \psi_n \end{array}} \qquad \begin{array}{c} \\ \\ \psi \end{array}$$

Les jugements hypothétique et schématique [Avr91], [HHP89] permettent de coder ces règles d'inférence de la manière générale suivante:

$$\bigwedge_{x_1 \ldots x_n} \left(\varphi_1^1 \ldots \succ \varphi_1^{n_1} \succ \psi_1\right) \ldots \succ \left(\varphi_1^1 \ldots \succ \varphi_1^{n_n} \succ \psi_n\right) \succ \psi$$

	Règle d'inférence	Code
règle de transitivité (explicite)	$\dfrac{[A] \quad \begin{matrix} A & B \end{matrix}}{B}$	$A \succ (A \succ B) \succ B$
constante	$Form(true)$	$\succ Form(true)$
règle de formation pour un connectif logique (implication)	$\dfrac{Form(A) \quad Form(B)}{Form(A \Rightarrow B)}$	$\bigwedge_{A,B} Form(A) \succ Form(B) \succ Form(A \Rightarrow B)$
axiome	$\dfrac{Form(A) \quad Form(B)}{True(A \Rightarrow (B \Rightarrow A))}$	$\bigwedge_{A,B} Form(A) \succ Form(B) \succ True(A \Rightarrow (B \Rightarrow A))$
règle d'inférence (modus ponens)	$\dfrac{Form(A) \quad Form(B) \quad True(A) \quad True(A \Rightarrow B)}{True(B)}$	$\bigwedge_{A,B} Form(A) \succ Form(B) \succ True(A) \succ True(A \Rightarrow B) \succ True(B)$
règle d'inférence de déduction naturelle (introduction de l'implication)	$\dfrac{\begin{matrix} [True(A)] \\ Form(A) \quad Form(B) \quad True(B) \end{matrix}}{True(A \Rightarrow B)}$	$\bigwedge_{A,B} Form(A) \succ Form(B) \succ (True(A) \succ True(B)) \succ True(A \Rightarrow B)$

Exemples de codes pour différentes règles d'inférence.

Quand la logique considérée est une représentation d'une logique à plusieurs relations de conséquence, les jugements de base qui constituent les parties droites des séquents d'une règle d'inférence peuvent tous avoir la même forme. Celle-ci peut alors être considérée comme une règle de formation pour une catégorie syntaxique caractérisée par cette forme. Par exemple, un connectif logique peut être considéré comme une règle de formation de la catégorie syntaxique formule. En LF, la forme de jugement de base peut être représentée par un type. Les catégories syntaxiques des variables introduites sont spécifiées au niveau de la représentation du jugement schématique.

Implémentation de systèmes logiques

Par *implémenter*, nous entendons *programmer* ou *spécifier* sur un système informatique.

Étant donné un système logique $S=(L,C)$ où $L=(\Sigma,\vdash)$, on désire implémenter le formalisme $F=(\Sigma,C)$, c'est à dire, son langage et son calcul. Par **abus de langage**, nous parlerons **d'implémenter** (**programmer** ou **spécifier**) le système logique, voire même la logique. Nous avons répertorié trois méthodes principales, de généricité croissante, mais d'efficacité a priori décroissante, du point de vue de la déduction automatique.

1) La *méthode théorie* consiste à programmer F directement. Par exemple, on peut programmer ainsi la théorie des entiers ou des ensembles.

2) Quand C est un calcul de preuves, la *méthode logique* consiste à employer un système formel auxiliaire $F'=(\Sigma',C')$, déjà implémenté. Dans ce système formel, on spécifie une signature σ et une théorie T telles que $\Sigma=\Sigma'/\sigma$ et $C=C'+T$. Parfois, σ est déterminée par T et n'est pas spécifiée (c'est le cas de PROLOG). Par exemple, un système formel pour une logique du premier ordre peut être utilisé pour spécifier une théorie non inductive des entiers.

3) La *méthode cadre logique* consiste à spécifier Σ et C pour un programme, appelé *cadre logique*, capable de les simuler. Souvent, on emploie un formalisme auxiliaire $F'=(\Sigma',C')$, déjà implémenté, Σ et C sont spécifiés dans Σ' et simulés par C'. Ce formalisme doit être le plus simple possible, mais permettre la description d'une classe de formalismes la plus large possible [HHP89]. Il permet de bien distinguer le niveau objet du méta-niveau. Par exemple, LF est un cadre logique capable de représenter de nombreux systèmes formels. Un autre exemple est LDS [Gab91], qui tente de circonscrire les systèmes formels de la manière la plus naturelle possible.

Notons que les **formalismes auxiliaires** des méthodes logique et cadre logique sont eux-mêmes **implémentés par l'une des trois méthodes** décrites.

Les spécifications des méthodes logiques et cadre logique peuvent être combinées à l'aide d'opérateurs, afin d'obtenir de nouvelles spécifications, que l'on appelle *spécifications structurées.*

	Méthode		
	Théorie	Logique	Cadre logique
Programmé	Σ, C	Σ', C'	Σ', C'
Spécifié		σ, T	Σ, C

Hiérarchie

Étant donnée une preuve $\pi=(N,E,D)$, une partie de preuve est un triplet (N',E',D') tel que N' est une partie de N, E' est la restriction de E à N', et D' est une partie de la restriction de D à $P(N')\times N'$. Remarque: toute partie de preuve est une preuve.

On appelle *lemme*, une partie connexe de preuve qui semble "exceptionnelle", soit parce que sa conclusion est "intéressante", soit parce qu'elle se répète telle quelle ou "à quelque chose près", à d'autres endroits de la preuve. La notion d'exception peut être laissée à **l'appréciation** de l'auteur de la preuve ou **formalisée** dans une certaine mesure.

Une personne qui regarde une preuve aimerait d'abord considérer les lemmes comme des pas de preuve élémentaires. Ensuite, elle aimerait examiner les lemmes qui ne lui paraissent pas évidents. Ainsi, quand on choisit un ensemble de lemmes, deux lemmes peuvent être **imbriqués** ($N_1 \subset N_2 \vee N_2 \subset N_1$), mais ils ne peuvent pas se **recouvrir** ($N_1 \cap N_2 \neq \varnothing \wedge N_1 \not\subset N_2 \wedge N_2 \not\subset N_1$) (Bien que leurs nœuds puissent être associés aux mêmes séquents formels).

En particulier, considérer les parties connexes maximales d'une preuve comme des lemmes, correspond à ne regarder d'abord, que ses hypothèses et ses conclusions.

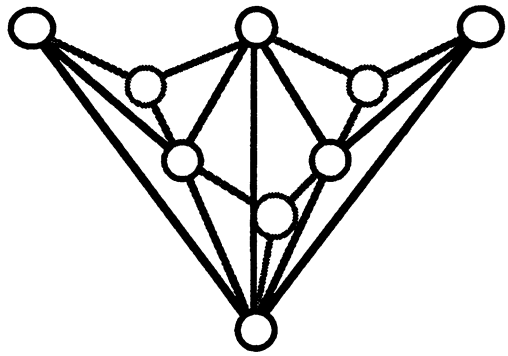


Illustration de la notion de lemme.
On dénombre six lemmes, divisés en trois niveaux,
représentés par des teintes de plus en plus claires.

La possibilité de *nommer* et de *référencer* des preuves connexes réalise cette hiérarchisation des preuves. Notons qu'appliquée systématiquement, cette méthode pourrait redonner aux preuves la structure de liste usuelle, que nous avons écartée initialement.

Conclusion

Un éditeur de preuves est un outil qui doit permettre à l'utilisateur de construire des preuves, de les mémoriser, de les imprimer, de modifier des preuves existantes, et de définir de nouveaux langages et calculs, en vérifiant si possible la correction des opérations réalisées. Dans sa plus simple expression, un traitement de texte ou mieux, un programme de dessin pourrait être considéré comme un éditeur de preuves très général. Mais il faut reconnaître qu'il ne serait pas très pratique. Il faut donc préciser un certain nombre de contraintes, permettant à l'utilisateur de développer ses preuves dans un cadre mieux adapté, prédéfini par un concepteur. Ce cadre peut être basé sur la notion de déduction, d'où l'utilité des notions introduites.

B) Besoins de l'utilisateur d'outils d'inférence

D'après l'analyse de ce qui précède, les besoins d'ordre général de l'utilisateur d'outils d'inférence pourraient vraisemblablement

correspondre à un environnement de travail ayant les possibilités suivantes (ces possibilités sont à rapprocher des bases de la conception d'ATINF données dans la partie B du chapitre Présentation):

Visualisation de preuves

Une preuve (déjà construite) doit être **présentée de manière usuelle**. C'est à dire, en accord avec les habitudes de l'utilisateur (mathématicien, logicien, etc.).

En particulier, cette présentation doit être **graphique**, pour respecter les notations de l'utilisateur, pouvant employer différents symboles, avec différentes dispositions.

Pour une meilleure lisibilité et une meilleure compréhension, elle doit être **structurée**, selon les souhaits de l'utilisateur. Cette notion de structure est fondée sur celle de déduction, qui est, bien entendu, indépendante du calcul utilisé. Notamment, il faut éviter la représentation naïve où chaque pas d'inférence est représenté par une ligne contenant la conclusion et les numéros des lignes contenant les prémisses de la règle appliquée. Ceci n'interdit pas l'utilisation de lemmes quand elle est justifiée.

Pour être adaptée à tout utilisateur, celui-ci doit pouvoir la **diriger**. Au début, il ne voit que la conclusion et les prémisses de la preuve présentée. Ensuite, il peut examiner les parties de preuves qui ne lui semblent pas évidentes, dans l'ordre qu'il désire.

Quand une preuve est longue (beaucoup de formules ou de symboles), il faut pouvoir cacher les pas (règles d'inférences, connectifs logiques, etc.) considérés comme moins importants, afin de diminuer sa taille, ce qui correspond à considérer des pas d'inférence de taille arbitraire. La présentation doit donc être **hiérarchique**, et permettre l'examen récursif des regroupements de pas d'inférence cachés.

[Lam93b] montre l'intérêt de présenter des preuves mathématiques usuelles de manière structurée et hiérarchique, même si elles ne sont pas entièrement formalisées.

Finalement, différents utilisateurs pouvant se servir de différentes notations, une preuve doit pouvoir être **présentée de plusieurs manières**.

Édition de preuves

Cet environnement permet l'édition de preuve, c'est à dire non seulement leur visualisation, leur mémorisation et leur impression, mais aussi leur **création**, **modification** et leur **réutilisation**, directement par l'utilisateur ou par l'intermédiaire d'outils d'inférence. Les outils d'inférence pouvant appliquer des règles d'inférence de manière contrôlée, l'éditeur de preuve peut aussi jouer le rôle d'un démonstrateur interactif.

Vérification de preuves

Il **vérifie** systématiquement la correction des preuves fournies, et des modifications suggérées directement par l'utilisateur ou par l'intermédiaire d'outils d'inférence. Bien entendu, cette vérification ne peut se faire que dans la limite de sa connaissance du système formel considéré.

Définition de systèmes formels

Il propose un cadre naturel pour **définir** des systèmes formels. Étant donnée la définition d'un système formel, il permet immédiatement à l'utilisateur d'**éditer** des preuves, en **vérifiant** ses manipulations. Cette possibilité permet l'expérimentation rapide de systèmes formels en évitant la programmation d'outils d'inférence spécifiques (éditeur, vérificateur, démonstrateur interactif, etc.).

Interface utilisateur

La qualité de son interface vise à mettre l'utilisateur en confiance. Elle est **naturelle**, c'est à dire simple et intuitive. Par exemple, elle peut employer des éléments d'*IUGG* (Interface Utilisateur Graphique Générique) standards (menus, boutons, etc.) répartis et définis de manière judicieuse. Dans la mesure du possible, elle est **homogène** et ne dépend pas de la logique considérée ni des outils associés. Autrement dit, l'environnement n'exige de l'utilisateur ni d'efforts de programmation, ni d'efforts de mémorisation.

L'interface est particulièrement soignée pour la **construction** de preuves pas à pas et pour **l'appel et le réglage des paramètres** des outils d'inférence.

Intégration des outils existants

Il propose un ensemble d'outils d'inférence qui peut être **complété** par l'utilisateur. Il peut utiliser tous **les outils d'inférence existants**, pas seulement ceux qui lui sont particulièrement adaptés. Cette caractéristique est particulièrement importante puisqu'elle permet de présenter des preuves obtenues par n'importe quel démonstrateur (pour plus de détail voir chapitre Utilisation sections B.3 et C.2.c).

Communication entre les outils

Il permet de faire communiquer directement et de manière transparente les outils d'inférence fondés sur le même système formel. Il permet aussi de faire communiquer des d'outils d'inférence fondés sur

des systèmes formels différents, par l'intermédiaire d'outils d'inférence capables de traduire des preuves entre systèmes formels.

Conclusion

L'environnement de travail que nous venons de décrire peut être vu comme un système de **liaison entre l'utilisateur et différents outils d'inférence**. Du côté de l'utilisateur, la liaison se traduit par une interface pratique et la possibilité d'éditer des preuves. Du côté des démonstrateurs, elle se traduit par l'emploi de passerelles mono ou bi-directionnelles.

C) GLEF et ATINF, une réponse possible à ces besoins

Un éditeur de preuves est un outil qui permet à l'utilisateur de construire, visualiser, mémoriser, imprimer et modifier des preuves. Seul, il satisfait certains besoins de l'utilisateur d'outils d'inférence.

Notre idée était de réaliser un éditeur de preuves qui puisse être étendu, afin de répondre à la plupart de ces besoins. Nous avons donc développé *GLEF* (Graphical Logical Edition Framework), un éditeur de preuves graphique, paramétrable par le système formel considéré et sa présentation, qui peut appeler et régler les paramètres d'outils d'inférence, ou d'autres outils. Par ailleurs, il tente de **factoriser** les similitudes de ces outils.

On peut considérer GLEF comme une interface ou un outil d'intégration pour ATINF. Mais il constitue un éditeur de preuves indépendant, qui peut coopérer avec des outils extérieurs à ATINF, ou les connecter entre eux.

Notons que ATINF et GLEF sont des outils **évolutifs**. ATINF regroupe des outils d'inférence de plus en plus nombreux. GLEF permet de connecter ces outils (parmi d'autres), il considère donc de nouveaux systèmes formels et factorise de nouvelles similitudes.

V. État de l'Art

Introduction

Ce chapitre constitue un tour d'horizon des travaux concernant la spécification de systèmes formels, les outils d'inférence et les éditeurs de documents.

L'étude des différents travaux sur la spécification de systèmes formels (première partie) met en évidence des **fondements théoriques possibles** pour GLEF.

Celles des outils d'inférence (deuxième partie) et des éditeurs de documents (troisième partie) établissent une liste de caractéristiques souhaitables pour ATINF. Une discussion termine chaque partie. Bien entendu, réaliser un système regroupant toutes ces caractéristiques n'est pas envisageable. Mais GLEF peut être **potentiellement capable de coopérer** avec des outils d'inférence quelconques et de les **connecter** entre eux, tout en regroupant les caractéristiques communes à la majorité des outils d'inférence et des éditeurs de documents.

A) Spécification de systèmes formels

Étant donné un système logique $S=(L,C)$ où $L=(\Sigma,\vdash)$, on désire implémenter le formalisme $F=(\Sigma,C)$. Dans le chapitre précédent, nous avons vu trois méthodes d'implémentation.

Ces méthodes sont le reflet d'études théoriques de plus en plus poussées. Si leur **efficacité** décroît a priori, leur difficulté **de programmation** est relativement constante et chaque implémentation permet de **spécifier** des classes de systèmes formels de plus en plus larges.

Cette partie ne décrit que la **spécification**, la **programmation** étant étudiée dans la partie suivante, qui concerne les outils d'inférence. Les exemples de formalismes donnés dans cette partie ont un caractère relativement général, ils ont suscité de nombreuses implémentations. Les formalismes spécifiques aux outils d'inférence sont décrits avec ceux-ci, dans la partie suivante.

1. Méthode théorie

La *méthode théorie* consiste à programmer F directement. En pratique, il faut programmer C et une interface pour la lecture et

l'écriture de Σ. Notons que cette méthode permet de considérer des calculs autres que des calculs de preuves.

Par exemple, pour implémenter la théorie des entiers selon cette méthode, on peut programmer les opérateurs arithmétiques (successeur, somme, produit, égalité), et une interface pour la lecture et l'écriture d'expressions arithmétiques. De la même façon, on peut programmer la théorie des ensembles, l'algèbre élémentaire, etc..

Une **calculatrice** ou, de façon plus générale, un **outil de calcul formel** (arithmétique, algébrique, etc.) constituent de telles implémentations.

2. Méthode logique

La *méthode logique* consiste à spécifier une signature et une théorie dans un système formel auxiliaire déjà implémenté.

Souvent, ce système formel auxiliaire correspond à une logique usuelle. Par exemple, citons la logique propositionnelle, le λ-calcul, les logiques de preuves de programmes, la logique des prédicats du premier ordre, la logique des prédicats d'ordre supérieur, etc.. De nombreuses variantes et hybrides de ces logiques usuelles existent aussi.

Notons que des logiques différentes peuvent avoir le même langage. C'est le cas, par exemple, des logiques propositionnelles classiques et intuitionnistes. On peut aussi imaginer des logiques linéaires ou relevantes qui aient même langage.

Rappelons, à l'aide de grammaires, les langages des logiques usuelles:

- Pour la logique propositionnelle:

```
<F> ::= <proposition>
      | ¬<F>                                  /* négation */
      | <F>1 ∧ <F>2                           /* conjonction */
      | <F>1 ∨ <F>2                           /* disjonction */
      | <F>1 ⇒ <F>2 | <F>1 ⇐ <F>2             /* implications */
      | <F>1 ⇔ <F>2                           /* équivalence */
```

Où `<proposition>` est un terminal (de l'analyse syntaxique, mais non-terminal de l'analyse lexicale) dénotant un nom.

- Pour le λ-calcul:

```
<F> ::= <constante>
      | <variable>
      | λ <variable>.<F>                      /* abstraction */
      | <F>1 <F>2                             /* application */
```

Où `<constante>` et `<variable>` sont des terminaux dénotant des noms pris dans deux ensembles différents.

• Pour la logique des prédicats du premier ordre:

```
/* formules */
<F> ::= <A> | ¬<F> | <F>1 ∧ <F>2 | <F>1 ∨ <F>2
      | <F>1 ⇒ <F>2 | <F>1 ⇐ <F>2 | <F>1 ⇔ <F>2
      | ∀ <variable> <F>          /* universalité */
      | ∃ <variable> <F>          /* existence */
/* atomes */
<A> ::= <prédicat> | <prédicat> ( <T>1,…,<T>n)
/* termes */
<T> ::= <variable>
      | <fonction> | <fonction> ( <T>1,…,<T>n)
```

Où <prédicat>, <fonction> et <variable> sont des terminaux dénotant des noms pris dans trois ensembles différents.

• Pour la logique des prédicats d'ordre supérieur:

```
/* formules */
<F> ::= <A> | ¬<F> | <F>1 ∧ <F>2 | <F>1 ∨ <F>2
      | <F>1 ⇒ <F>2 | <F>1 ⇐ <F>2 | <F>1 ⇔ <F>2
      | ∀ <variable> <F>          /* universalité */
      | ∃ <variable> <F>          /* existence */
/* atomes */
<A> ::= <variable> | <variable> ( <T>1,…,<T>n)
      | <prédicat> | <prédicat> ( <T>1,…,<T>n)
/* termes */
<T> ::= <variable> | <variable> ( <T>1,…,<T>n)
      | <fonction> | <fonction> ( <T>1,…,<T>n)
```

Où <prédicat>, <fonction> et <variable> sont des terminaux dénotant des noms pris dans trois ensembles différents.

• Pour les preuves de programmes, on utilise une partie de la grammaire du langage de programmation concerné. Cette partie correspond aux **suites d'instructions**.

En parallèle, on utilise une **logique auxiliaire**, par exemple, la logique des prédicats du premier ordre avec égalité, entiers et fonctions sur les entiers.

La grammaire peut être de la forme:

```
<P> ::= <F>1 <I*> <F>2
/* Formules de la logique auxiliaire */
<F> ::= …
/* Suites d'instructions */
<I*> := …
```

Dans une instance de la première règle, les termes qui correspondent à <F>1 et <F>2 s'appellent respectivement *pré-condition* et *post-condition*.

3. Méthode cadre logique

La *méthode cadre logique* consiste à spécifier le système formel (langage et calcul) pour un programme capable de les simuler.

Les **représentations** de type système axiomatique, Hilbert, déduction naturelle ou Gentzen pourraient être considérées comme des cadres logiques de généricité croissante, si elles étaient entièrement formalisées [Avr91].

Les formalismes de AUTOMATH, LF et CC sont des λ-calculs typés de différentes puissances [Bar92], que l'on peut employer comme des cadres logiques. LDS est un cadre logique qui tente de rester plus "proche" des systèmes formels qu'il permet de définir.

AUTOMATH [dBr80]

Le projet *AUTOMATH* (1967) a été le premier à suivre une telle approche, il a conduit à définir une famille de formalismes dans lesquels la vérification de preuves était réduite à une vérification de types, les règles d'inférence étant définies par des constantes.

L'idée était de développer un système d'écriture permettant d'exprimer toutes les théories mathématiques. Ce système devait être suffisamment précis pour effectuer des vérifications purement syntaxiques, c'est à dire sans interprétation. Des essais de formalisation avaient déjà été tentés par Leibniz, Peano et Hilbert, mais l'apparition de l'ordinateur a permis de standardiser la notion de vérification formelle (dans les limites théoriques déterminées par le théorème d'incomplétude de Gödel).

Les trois motivations pour le développement d'un formalisme permettant d'exprimer toutes les théories mathématiques étaient la vérification des preuves, la compréhension des mathématiques et l'automatisation de calculs.

Bien qu'en général, les mathématiciens vérifient les preuves des théorèmes qu'ils utilisent ou démontrent, la **vérification** purement formelle est trop méticuleuse, et ils sont parfois amenés à faire **confiance** à leur intuition ou a des théorèmes déjà démontrés par d'autres mathématiciens.

Si la théorie mathématique utilisée est très peu intuitive ou si plusieurs théories mathématiques sont mélangées, cette confiance peut être source d'erreurs. Un formalisme permettrait d'automatiser la vérification des preuves.

L'**étude** des systèmes formels a permis de mieux comprendre les mécanismes mathématiques en différenciant les notions de langage, de méta-langage et d'interprétation. Le rôle de l'interprétation est souvent

sous-estimé. Un formalisme permettrait d'isoler le langage et le raisonnement de l'interprétation.

En complément de la vérification automatique, les **ordinateurs** peuvent effectuer différents calculs utiles. Par exemple, ils peuvent calculer quels axiomes et quelles règles d'inférence interviennent dans une preuve de théorème.

De manière générale, on peut écrire les mathématiques sous forme de *livres* constitués d'une suite de *lignes*. Au niveau de chaque ligne, on considère un contexte courant, qui peut être étendu ou restreint par des lignes particulières. Un contexte peut contenir une liste de déclarations de variables, d'hypothèses et de notions mathématiques.

En *SEMIPAL*, le langage de base de AUTOMATH, on utilise une représentation semblable. Cependant, un contexte ne peut contenir que des déclarations de variables et des hypothèses. Les notions mathématiques sont définies par les lignes précédentes et ne peuvent pas être retirées.

Il est possible d'écrire trois sortes de lignes (`x` est un nom, `M` est un terme):

- Les *ouvertures de blocs* (`x`) étendent le contexte courant par de nouvelles variables ou hypothèses.
- Les *définitions* (`x:=M`) introduisent des symboles définis comme des termes construits sur les variables et les hypothèses du contexte courant, et les symboles introduits par les lignes précédentes.
- Les *lignes PN* (`x:=PN`) introduisent des symboles primitifs (constantes, axiomes, etc.).

Après chaque définition, ou ligne PN, le contexte est réinitialisé à la liste vide.

Exemple:

```
/* Chaque variable apparaissant avant le signe * est une
ouverture de bloc */
/* f est une fonction à un argument */
   x * f := PN
/* g est une fonction à deux arguments */
   x,y * g := PN
/* h est définie en fonction de f et g */
   x * h := g(f(x),x)
```

La *δ-réduction* est une règle de réécriture qui consiste à développer les définitions, Elle correspond grosso-modo à l'évaluation des macros dans les langages structurés.

On peut étendre SEMIPAL par le *λ-calcul* (*λ-SEMIPAL*). Rappelons que les termes du λ-calcul (appelés *λ-termes*) sont des termes dont certains sous-termes peuvent manquer. Les sous-termes manquants sont remplacés par des variables muettes introduites par des *abstractions*. L'*application* permet de compléter les λ-termes en remplaçant certaines variables muettes par des termes donnés.

Notations:

- `[x]E` Abstraction sur le terme `E`, `E` peut contenir la variable muette `x` à certaines places de sous-termes.
- `{p}E` Application du terme fonction `E` au terme argument `p`.

La *β-réduction* et la *η-réduction* sont deux règles de réécriture relatives aux notions d'abstraction et d'application. Leur définition est:

$\frac{\{p\}[x]E}{[x/p]E}$	([x/p]E dénote le terme obtenu en substituant p aux occurrences libres de x dans E)
$\frac{[x]\{x\}E}{E}$	(si E ne contient pas x)

Une autre extension de SEMIPAL consiste à utiliser des *types* pour **limiter** la construction de termes (*PAL*). A chaque terme est associé un type, qui est un terme de même nature.

Pour introduire les types, on change la syntaxe des ouvertures de blocs et des lignes PN (`x` est un nom, `T` est un terme):

`x:T`	Ouverture de bloc.
`x:=PN:T`	Ligne PN.

Un terme noté *type* sert de type primitif. On utilise parfois un second type primitif noté *prop*, pour typer les propositions.

Pour construire le terme $p(E_1...,E_n)$ à partir de $p(x_1...,x_n)$, le type de chaque E_i doit être **égal** à celui de x_i où les occurrences de $x_1...x_{i-1}$ ont été remplacées par $E_1...E_{i-1}$. L'égalité considérée est une *égalité définitionnelle*, pour tester si deux termes sont égaux, il faut effectuer toutes les δ-réductions possibles (on ne considère pas encore l'extension par le λ-calcul).

Le type de $p(E_1...,E_n)$ est celui de $p(x_1...,x_n)$ où les occurrences de $x_1...,x_n$ ont été remplacées par $E_1...,E_n$.

Les types des symboles introduits par les **définitions** sont obtenus par extension de cette règle. Toutefois, pour vérifier ce type, on peut écrire les définitions sous la forme:

`x:=M:T`	Définition.

L'utilisation des types permet la mise en œuvre du principe de Curry-Howard qui **définit les propositions comme des types**. Les preuves d'une proposition sont les termes admettant cette proposition pour type.

La **vérification** d'une preuve d'une proposition consiste à vérifier que la proposition est effectivement un type de la preuve. La **recherche** d'une preuve d'une proposition consiste à rechercher un terme admettant cette proposition pour type.

Exemple:

On considère un réel q et deux fonctions Φ et Ψ de l'ensemble des réels dans lui même. On suppose avoir déjà démontré le théorème:

> Théorème 1. Soit x un réel. Supposons $\Psi(x)>1$.
> Soit n un entier. Supposons $\Phi(x)>x^n$. Alors $\Psi(x)>n$.

Et les deux propositions $\Psi(q)>1$ et $\Phi(q)>q^5$. On veut démontrer $\Psi(q)>5$ ...

On dénote respectivement les preuves du théorème, des deux propositions et la proposition à démontrer par `Th1`, `(1)`, `(2)` et `P`.

Démontrer $\Psi(q)>5$ revient à **rechercher** un terme `p` de type `P`. Appliquons le théorème pour $x=q$ et $n=5$:

```
Th2 := Th1(q,(1),5,(2)):P
```

Cette application vérifie que `q` est un réel, `(1)` est une preuve de $\Psi(q)>1$, `5` est un entier, `(2)` est une preuve de $\Phi(q)>q^5$ et `Th1(q,(1),5,(2))` est de type `P`. `Th2` est le nom de la nouvelle preuve.

Définissons la notion de *degré* d'un terme. `type` est un terme de degré 1. Si le terme E est de type F, le degré de E est celui de F plus 1.

Les propositions sont représentées par des termes de degré 2, les preuves sont représentées par des termes de degré 3. En pratique, on décrit **les théories mathématiques à l'aide de termes de degrés inférieurs ou égaux à 3.**

Pour **combiner** ces deux extensions (λ-calcul et typage) de SEMIPAL (*AUT-QE*, *AUT-68*, *AUT-SL*), on type les variables introduites par les abstractions. Il faut ensuite déterminer les types des λ-termes, par des règles de *jugement de typage*[4], dont dépendent le pouvoir d'expression et les propriétés formelles du langage.

---

[4]L'introduction de la notion de jugement de typage permet d'homogénéiser cette description avec celles de LF [HHP89] et CC [CH88], étudiés dans les sections suivantes.

Le jugement de typage "Dans le contexte Γ, le terme E est de type F" se note "$\Gamma \vdash E : F$".

Typage des λ-termes de AUT-QE et AUT-SL:

$$\frac{\Gamma[x{:}A] \vdash B(x){:}C(x)}{\Gamma \vdash [x{:}A]B(x){:}[x{:}A]C(x)}$$

En AUT-68, cette règle est différente si B est de degré 2:

$$\frac{\Gamma[x{:}A] \vdash B(x){:}\mathrm{type}}{\Gamma \vdash [x{:}A]B(x){:}\mathrm{type}}$$

De plus, dans l'abstraction $[x{:}A]B$, seuls certains degrés sont admis pour A et B (({2},{1,2,3}) pour AUT-QE, ({2},{2,3}) pour AUT-68, (N, N) pour AUT-SL[5]).

La règle d'inclusion de type:

$$\frac{\Gamma \vdash T{:}[x{:}A]\mathrm{type}}{\Gamma \vdash T{:}\mathrm{type}}$$

est autorisée en AUT-QE, interdite en AUT-SL et forcée en AUT-68 (qui ignore systématiquement les abstractions sur type).

PAL, AUT-QE et AUT-68 peuvent être décrits comme des PTS (Pure Type System) [Bar92]. Ainsi, on peut les comparer avec précision à d'autres λ-calculs similaires, comme LF [HHP89] ou CC [CH88].

Pour les termes, on utilise une autre *égalité définitionnelle*. Celle-ci est définie par la fermeture réflexive, symétrique et transitive les règles de β, η, et δ-réduction.

Établir la confluence et la terminaison (pour les termes bien typés) de la $\beta\eta\delta$-réduction montre la **décidabilité de l'égalité définitionnelle**. Deux termes sont égaux si et seulement si leurs formes normales sont égales. Cette propriété permet ensuite d'établir la **décidabilité de tous les jugements des théories des types** de la famille AUTOMATH.

Le théorème de fermeture affirme que si un terme A est correct et se réduit en un terme B, alors B est correct. Ce théorème permet **d'éviter** les vérifications de types lors des β-réductions.

---

[5]En AUT-SL, tout livre peut être écrit à l'aide d'une seule ligne, en transformant les lignes PN en ouverture de blocs et en éliminant les définitions. Ce formalisme se rapproche ainsi de CC [CH88], étudié plus loin.

Si les livres devaient préciser chaque pas de réécriture, ils auraient une taille relativement importante et seraient vraisemblablement illisibles. La **décidabilité** de l'égalité définitionnelle dans les formalismes de AUTOMATH permet de réaliser des **vérificateurs automatiques** pour les livres qui omettent certains pas de réécriture, voire tous.

Cependant, le calcul des formes normales peut être assez long, surtout dans les livres de grande taille. L'utilisateur peut alors **aider** un vérificateur automatique en ajoutant des lignes supplémentaires.

Étant donnée une proposition `P`, la recherche d'une preuve consiste à trouver un terme `p` tel que l'on puisse écrire la définition `th := p:P` dans un livre donné. Dans le cas général, c'est un problème très difficile.

Toutefois, un **démonstrateur automatique** pourrait être utilisé pour rechercher de tels termes quand ceux-ci sont **simples** ou pour un livre représentant une **théorie mathématique bien connue.**

Les formalismes de AUTOMATH ont été utilisés pour traduire ou écrire, puis vérifier des livres de mathématiques entiers ("Grundlagen" de Landau (1930) traduit et vérifié en AUT-QE, "Real Analysis" de Zucker (1975) écrit et vérifié en AUT-PI). Lors de la traduction de livres de mathématiques, on a pu noter la constance du rapport entre le nombre de lignes d'une preuve formelle et le nombre de lignes de la preuve informelle dont elle est issue (entre 10 et 20).

Pour des raisons purement contingentes, le projet AUTOMATH a été abandonné, mais il a eu des suites importantes, notamment ses descendants LF et CC.

LF [AHM87], [HHP87], [HHP89]

Les logiques utilisées par les philosophes, les physiciens, les linguistes, les informaticiens, les logiciens ou les mathématiciens sont très diverses. La réalisation d'un outil d'inférence pour une logique particulière est assez longue. Mais **de nombreuses tâches sont similaires pour la réalisation de différents outils d'inférence.**

Au niveau la syntaxe, il faut gérer les opérateurs de liaison, les substitutions, les schémas de formules, de termes, de règles. Il faut ensuite définir la représentation formelle des preuves, et les mécanismes permettant de vérifier leur construction. On peut enfin réaliser un démonstrateur automatique, programmer des tactiques, des compositions de tactiques, l'unification, le filtrage, etc..

LF (Logical Framework) est **un formalisme qui permet de définir une large classe de logiques.** Il permet de spécifier de manière simple et générale le langage, les axiomes, les règles

d'inférence et les preuves, en termes d'un λ-calcul Π-typé (c'est à dire un λ-cacul avec des produits dépendants pour typer les abstractions) très proche des formalismes de AUTOMATH. En résolvant **une fois pour toutes** les problèmes précédents, il peut servir de **fondement** au développement d'outils capables d'éditer, vérifier ou rechercher des preuves, et permettre éventuellement de les **paramétrer** par le système formel objet.

Le principe de Curry-Howard, employé dans les formalismes de AUTOMATH, consiste à représenter les **propositions** par des types. Une proposition est un théorème si et seulement s'il existe un terme admettant sa représentation pour type. Ce terme représente alors une preuve de ce théorème. Les langages de AUTOMATH permettent ainsi de simuler la vérification et la recherche de preuves.

De manière plus générale, une logique possède un langage dans lequel on peut faire des méta-propositions de la forme "φ est vraie" ou "φ est valide" (φ est une formule). Cette forme de proposition s'appelle *jugement de base*. Chaque logique possède un ensemble caractéristique de jugements de base.

Le principe utilisé dans LF consiste à représenter les jugements par des types. Un jugement est vrai si et seulement si sa représentation est le type d'un terme. Ce terme représente alors une preuve du jugement.

En plus des jugements de base, on emploie deux jugements d'ordre supérieur. Le jugement *hypothétique* exprime une forme de conséquence: "Si on suppose J_1, alors J_2" (J_1 et J_2 sont des jugements). Le jugement *schématique* exprime une forme de généralité "Pour tout x de C, on a $J(x)$" (J est un jugement). Ces deux formes de jugements d'ordre supérieur peuvent être représentées en LF.

Une règle d'inférence de la logique objet est représentée par une constante dont le type est un jugement d'ordre supérieur (dans le cas d'un axiome, ce jugement d'ordre supérieur ne contient pas de jugement hypothétique). **Les règles d'inférence constituent ainsi les preuves primitives de jugements d'ordre supérieur**. Ainsi, l'utilisation des jugements d'ordre supérieur **unifie** les notions de règle d'inférence, et de dérivation de règles d'inférence, et de preuve.

Le **langage** de LF est constitué de trois sortes de termes les *objets*, les *familles* et les *genres*. Les **objets** (notés M, N, P) représentent les entités syntaxiques, les preuves et les règles d'inférence. Les **familles** (notées A, B, C) représentent les classes syntaxiques, les jugements et les propositions, elles permettent de classifier les objets. Les **genres** (notés K, L) servent à classifier les familles. Cette **limitation** à trois

niveaux est à rapprocher des limitations sur les **degrés** des termes dans les formalismes de AUTOMATH.

Voici la grammaire du langage. On ne distingue les variables (notées x, y, z) des constantes (notées a, b pour les objets et c, d pour les familles) que pour des raisons de clarté.

- Genres: $K ::= \text{Type} \mid \Pi x{:}A.K$
- Familles: $A ::= a \mid \Pi x{:}A.B \mid \lambda x{:}A.B \mid A\ M$
- Objets: $M ::= c \mid x \mid \lambda x{:}A.M \mid M\ N$

Pour l'exposé des règles de jugement de typage de LF [HHP89], les *contextes* et les *signatures* mémorisent respectivement les environnements de types des variables et des constantes. Ces règles ne sont pas présentées ici, notons toutefois que les produits dépendants (notés $\Pi x{:}\ldots$), qui représentent les espaces fonctionnels, servent intuitivement à typer les abstractions.

- Contextes: $\Gamma ::= () \mid \Gamma, x{:}A$
- Signatures: $\Sigma ::= () \mid \Sigma, a{:}K \mid \Sigma, c{:}A$

A l'instar des formalismes de AUTOMATH, l'étude des **propriétés formelles** de LF, établit la **décidabilité de l'égalité définitionnelle** (existence de formes $\beta\eta$-normales uniques fortes) puis celle **de toutes les assertions de sa théorie de types**. On peut donc écrire un vérificateur automatique pour les livres LF.

En LF, un système formel objet est défini par une signature. Pour justifier cette définition, il faut exhiber une *bijection compositionnelle*, c'est à dire une bijection qui commute avec la substitution, entre le système formel et la classe des termes définie par la signature.

De manière générale, cette signature définit la syntaxe des formules, les jugements de base et les règles d'inférence (jugements d'ordre supérieur). Une constante représente chaque catégorie syntaxique, connectif logique, schéma d'axiome et règle d'inférence du système formel. Ainsi, chacune de ses preuves est représentée par une combinaison de constantes. La preuve est dite correcte ssi le formalisme permet d'écrire cette combinaison à la suite de la signature du système formel.

Exemple:

Catégories syntaxiques:

```
Term: Type
Formula: Type
```

Fonctions:

```
c: Term
f: Term → Term
g: Term → Term → Term
```

Prédicats:

```
P: Term → Formula
```

Connectifs logiques:

```
⇒: Formula → Formula → Formula
...
```

La catégorie syntaxique `variable` est inutile, car les **variables** du système formel objet sont **représentées** par des variables de LF d'un type approprié. Ainsi, des constantes dont le domaine est de type fonctionnel peuvent représenter les opérateurs de **liaison**:

```
...
∀: (Term → Formula) → Formula
...
```

Si ε est la bijection compositionnelle qui code le système formel de l'exemple en LF, on a:

$$\varepsilon(\forall x.\varphi(x)) = \forall(\lambda x{:}\mathrm{Term}\ \varepsilon(\varphi(x)))$$

Grâce à cette représentation, les problèmes des α-conversions (renommage) et des substitutions sans captures sont traitées une fois pour toutes par le noyau LF. Notons que pour des liaisons non standard de variables, une autre représentation est nécessaire. Par exemple, la notion de jugement permet d'exprimer des contraintes contextuelles.

Les **règles d'inférence** correspondant à des jugements hypothétiques et schématiques peuvent être représentées par des constantes de LF. C'est le cas des règles d'inférence des systèmes de Hilbert et de déduction naturelle, mais pas de celles des logiques linéaires ou relevantes. Rappelons qu'en logique *linéaire* (resp. *relevante*) chaque hypothèse doit être utilisée une fois et une seule (resp. au moins une fois).

Exemples de représentations de règles d'inférence en LF :

- La règle d'inférence "modus-ponens" du calcul propositionnel :

$$\frac{A \quad A \Rightarrow B}{B}$$

se code par un terme de la forme :

$$\Pi A{:}o\ \Pi B{:}o\ T(A) \rightarrow T(A \Rightarrow B) \rightarrow T(B)$$

- La règle de déduction naturelle correspondant au théorème de la déduction :

$$\frac{\begin{array}{c}(A)\\ A \quad B\end{array}}{B}$$

se code par un terme de la forme :

$$\Pi A{:}o\ \Pi B{:}o\ T(A) \rightarrow (T(A) \rightarrow T(B)) \rightarrow T(B)$$

On note ici l'utilisation d'un jugement d'ordre supérieur $T(A) \rightarrow T(B)$ comme deuxième argument de la règle d'inférence.

LF permet ainsi de définir tous les systèmes **purs** de **Hilbert** et de **déduction naturelle**.

La notion de **jugement de base** est essentielle pour la définition de logiques modales pour lesquelles on emploie simultanément plusieurs formes de jugement de base (vrai, valide).

La notion de **jugement d'ordre supérieur** est essentielle pour la représentation des systèmes de déduction naturelle. Les règles d'inférence de ces systèmes peuvent en effet admettre des preuves de jugements d'ordre supérieur comme arguments. De façon plus générale, cette notion permet d'exprimer la **dérivation** de règles d'inférence et **l'utilisation** de ces règles d'inférence dérivées.

LF a permis de définir de nombreux systèmes formels, au cours de différents travaux, par exemple:

- Des logiques modales
- Des λ-calculs
- La théorie des types de LF
- Une sémantique opérationnelle
- Des langages de programmation logique

CC [CH85], [CH86], [CH88]

CC (Calculus of Constructions) est un autre formalisme descendant de AUTOMATH, fondé sur la correspondance de Curry-Howard entre les propositions et les types. Il permet d'écrire des preuves dans un style de

déduction naturelle. On peut le considérer comme un **cadre logique** ou comme un **langage de programmation** où les programmes sont vus comme des preuves de leurs spécifications. C'est un λ-calcul Π-typé dont la théorie des types, plus puissante que celle de LF (bien qu'il ne soit destiné qu'à formaliser les mathématiques constructives [HHP89]), il permet d'exprimer des notions imprédicatives.

La structure des termes de CC est fondée sur les constructeurs suivants:

- Univers: $*$
- λ-abstraction: $(\lambda x{:}N)M$
- Produit dépendant: $[x{:}A]M$
- Application: $(M\ N)$
- Variable: x

L'univers joue le rôle des types primitifs `type` et `prop` des formalismes de AUTOMATH.

Pour la syntaxe abstraite des termes, on utilise la notation de de Bruijn [dBr72]. Les variables sont codées par des entiers. L'entier n code la variable liée par le $n^{\text{ième}}$ opérateur de liaison situé au dessus dans le terme.

Exemples:

- $(\lambda x{:}M)((\lambda y{:}N)(x\ y)\ x)$ se code $\lambda(M,\lambda(N,(2\ 1)1))$
- $[x{:}A][y{:}B](x\ y)$ se code $[A][B](2\ 1)$

A cause des opérateurs de liaison, l'algèbre de termes définie par les constructeurs précédents n'est pas libre. On définit donc les termes **biens formés** (notés A, B). Un *contexte* (noté Γ, Λ) est un terme bien formé constitué d'une suite de produits sur $*$. Un *objet* (noté M, N) est un terme bien formé qui n'est pas un contexte.

Une *construction* est un terme bien formé et bien typé relativement à la théorie de types définie par les règles de jugement de typage [CH88]. Pour exposer ces règles, on utilise les deux formes de jugement:

$\Gamma \vdash \Lambda$	"Le contexte Λ est bien typé dans le contexte bien typé Γ"
$\Gamma \vdash M{:}A$	"L'objet M est bien typé et de type A, dans le contexte bien typé Γ"

Exemples:

$\Gamma \vdash *$	"Γ est un contexte bien typé"
$\Gamma \vdash M{:}\Lambda$	"M est une proposition bien typée sur la liste d'arguments déclarés par Λ"
$\Gamma \vdash M{:}N$	"M est une preuve de la proposition N, dans le contexte d'hypothèses Γ"

A l'instar des formalismes de AUTOMATH et de LF, trois degrés de typage suffisent en pratique:

- Un contexte sert à typer des propositions logiques.
- Une proposition est représentée par un type.
- Une preuve est représentée par une construction dont le type est la représentation d'une proposition. En effaçant les types d'une preuve, on obtient un λ-terme pur qui représente l'algorithme de la preuve.

Exemples:

- L'algorithme d'identité polymorphe est construit par la preuve: $(\lambda A{:}*)(\lambda x{:}A)x$ dont le type est défini par le jugement: $[A{:}*][x{:}A]A{:}\ * \vdash (\lambda A{:}*)(\lambda x{:}A)x{:}[A{:}*][x{:}A]A$
  Cette proposition peut se lire comme le schéma d'implication $A \Rightarrow A$.
- Le système est consistant, car il n'est pas possible de construire une preuve pour la proposition $[A{:}*]A$.

Notations:

- $\lambda x.\mathrm{N}$ dénote $(\lambda x{:}\mathrm{M})\mathrm{N}$ quand le type de x est déterminé par le contexte.
- $\forall A.X$ dénote $[A{:}*]X$.
- $A \rightarrow B$ dénote $[x{:}A]B$ quand x n'est pas libre dans B.

L'étude des **propriétés formelles** de CC, établit la **décidabilité de l'égalité définitionnelle** (existence de formes $\beta\eta$-normales uniques fortes) puis celle **de toutes les assertions de sa théorie de types**. On peut donc écrire un vérificateur automatique pour les livres CC.

Le formalisme est **consistant**, les représentations de certaines propositions ne sont types d'aucune construction.

Pour la mécanisation de CC, il est possible de réécrire les règles de jugement de typage en normalisant systématiquement les types. Ainsi,

les réductions ne sont plus faites à chaque test d'égalité définitionnelle, mais une fois pour toutes.

CC a été développé dans l'idée de faciliter la réalisation de **programmes**. On peut considérer qu'une assertion de la forme $\Delta \vdash M{:}P$ exprime que l'algorithme $\nu_\Delta(M)$, dont les α_Δ arguments sont décrits par Δ, obéit aux spécifications P. α et ν sont des fonctions définies dans [CH88]. α_Δ est le nombre des produits de Δ dont le type introduit n'est pas un contexte. $\nu_\Delta(M)$ est l'algorithme obtenu en "retirant" de M toutes les informations de type. C'est un λ-terme pur de λ^{α_Δ} (ensemble des λ-termes à α_Δ variables libres). Il se termine quand ses entrées sont bien typées.

Presque toutes les fonctions partielles récursives peuvent être définies en CC. Par exemple, **toutes** les fonctions **totales récursives**, que l'on peut prouver totales en logique d'ordre supérieur, correspondent aux algorithmes extraits des preuves de la proposition $\mathrm{nat}\rightarrow\mathrm{nat}$ pour le type approprié $\mathrm{nat} = \forall A\ (A\rightarrow A) \rightarrow (A\rightarrow A)$.

Pour réaliser un outil fondé sur CC, un certain nombre d'améliorations d'ordre pratique sont possibles.

L'introduction de **définitions** permettrait de simplifier les termes et d'optimiser l'utilisation de la mémoire et les temps de calcul. Par exemple, `let` $x{=}M_1$ `in` M_2 pourrait dénoter $((\lambda x{:}P)M_2\,M_1)$.

Dans les constantes (polymorphes), la plupart des arguments propositionnels sont redondants, car on peut souvent les inférer comme sous-termes des types des arguments qui les suivent. Ainsi, autoriser la **définition d'arguments implicites, synthétisés** lors des applications, simplifie l'écriture de celles-ci. [CH88] montre comment vérifier automatiquement si un argument peut être considéré comme implicite. L'utilisation d'arguments implicites est courante en mathématiques. Par exemple, en théorie des catégories, le domaine et le co-domaine de f et g ne sont pas indiqués dans la composition $f \circ g$, car ils sont déterminés par f et g.

Des **formats** de la forme suivante permettraient de spécifier la **syntaxe** des constantes définies:

```
{A:*}{B:*}{C:*}[f:A→B] o [g:B→C] ← [x:A](g (f x))
```

(Les accolades mettent les arguments implicites en évidence)

COQ [D+91] est une implémentation de CC qui permet de définir des types inductifs (Calculus of Inductive Constructions). Il a permis d'écrire et de vérifier des programmes tels que division euclidienne, heap sort, merge sort, etc..

LDS [Gab91]

Quand on considère plusieurs logiques, on voit que des logiques d'origines très éloignées peuvent être **formalisées de manière très similaire**. Par exemple, la représentation sous forme de table de vérité de la logique classique est très proche de celles des logiques de Lukasiewicz à valeurs finies mais éloignée de celle de la logique intuitionniste. L'inverse se produit pour la représentation de type Gentzen.

Un cadre logique dans lequel **la plupart** des logiques pourraient être exprimées de manière **similaire** serait très utile. Un tel cadre logique, nommé LDS (Labelled Deductive Systems) a été proposé par D. Gabbay [Gab91].

Dans beaucoup de logiques, on utilise le modus-ponens et les règles de quantification sont les mêmes. Souvent, leurs différences peuvent être vues comme des méta-considérations sur la théorie des preuves ou la sémantique. Pour ramener ces méta-considérations au niveau objet, LDS représente les méta-informations dans des **étiquettes** associées aux formules. Ainsi, $\alpha{:}A$ dénote la formule A étiquetée par α.

On considère qu'une *relation de conséquence* est une relation entre une structure de formules et une formule (mono-conclusion), satisfaisant la réflexivité restreinte et la *transitivité substitutionnelle* ($\Gamma[A]$ signifie que A apparaît quelque part dans Γ et $\Gamma[\Delta]$ signifie que Δ remplace A au sein de Γ):

$$\frac{\Delta \vdash A \quad \Gamma[A] \vdash B}{\Gamma[\Delta] \vdash B}$$

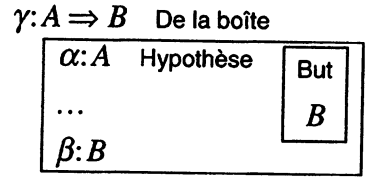
Du point de vue du calcul, les structures de formules sont décrites par les étiquettes. Par exemple, $\{a_1{:}A_1 \ldots, a_n{:}A_n\}, a_1 \ldots < a_n$ représente un ensemble ordonné.

Aux règles d'inférence classiques, on ajoute des règles de propagation des étiquettes. La **structure** des différentes règles est **figée** pour toutes les logiques. Par exemple, l'élimination de l'implication (modus-ponens) a toujours la forme[6]:

$$\frac{\alpha{:}A \qquad \beta{:}A \Rightarrow B}{\beta + \alpha{:}B} \ \text{si} \Psi_{MP}(\alpha,\beta)$$

[6] $+$ et Ψ_{MP} dépendent de la logique objet.

L'introduction de l'implication (théorème de la déduction) a la forme: "pour montrer $t{:}A \Rightarrow B$, ouvrir une boîte, supposer $x{:}A$ et montrer $y{:}B$", représentée par le schéma suivant[7]:



Sortie $\gamma{:}A \Rightarrow B$ si $\Psi_I(\alpha,\beta,\gamma)$

Ψ_{MP} et Ψ_I sont des conditions à l'application des règles, exprimées dans un méta-langage adéquat. Les règles de quantification sont les mêmes pour toutes les logiques.

Toutefois, on a la **liberté** de choisir **l'ensemble d'étiquettes**, les **propriétés de** $+$, les **formules importées** et les **règles autorisées** à l'intérieur des boîtes, et les **conditions** (Ψ_{MP} et Ψ_I) relatives à l'application des règles.

Par exemple, les étiquettes considérées pour la **logique linéaire** sont des multi-ensembles d'étiquettes atomiques. $+$ représente l'union de multi-ensembles. Toute formule peut être importée. Les définitions respectives de $\Psi_{MP}(\alpha,\beta)$ et $\Psi_I(\alpha,\beta,\gamma)$ sont $\alpha \cap \beta = \varnothing \wedge \alpha \neq \varnothing$ et $a \in \beta \wedge \gamma = \beta - [a]$. A est un **théorème** si on peut dériver $\varnothing{:}A$.

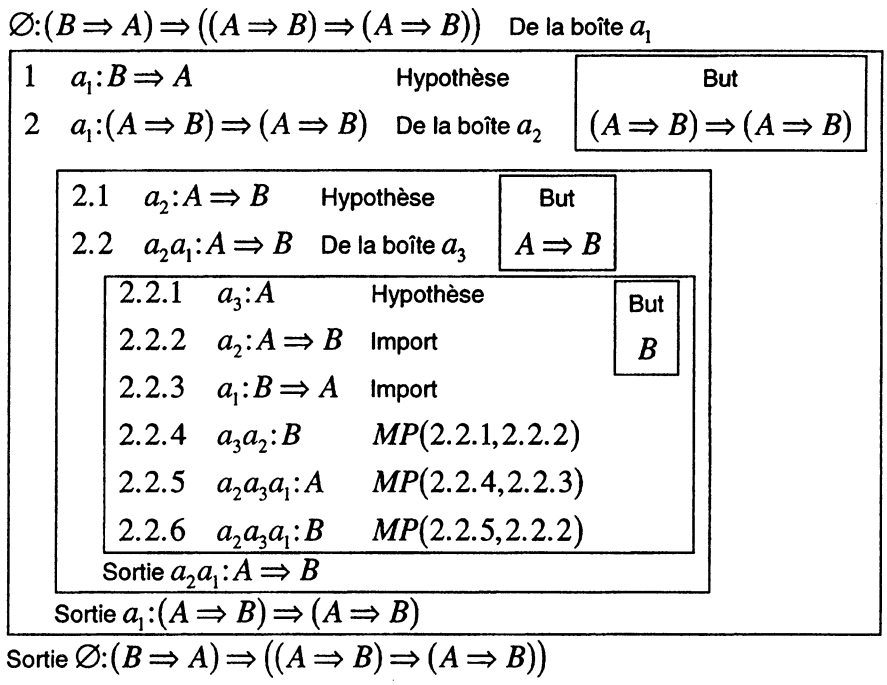
Considérer des ensembles d'étiquettes atomiques, au lieu de multi-ensembles, permet d'obtenir la **logique relevante**.

Ensuite, remplacer $\Psi_I(\alpha,\beta,\gamma)$ par $\gamma = \beta - \{a\}$, donne la **logique intuitionniste**.

Finalement, admettre une boîte dont la dernière formule étiquetée soit ou le but courant, ou un but précédent, donne la **logique classique**.

Voici une preuve de la formule $(B \Rightarrow A) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow B))$ en logique relevante (ou intuitionniste, ou classique). On ne peut pas transformer cette preuve en logique linéaire car la ligne 2.2.6 contiendrait la formule $a_2a_3a_1a_2{:}B$ dont une étiquette a_2 subsisterait jusqu'à la fin:

[7] Ψ_I dépend de la logique objet.



Pour la **logique modale**, les mondes sont représentés par des étiquettes, les règles concernant les connectifs □ et ◊ sont fondées sur la relation d'ordre partiel < entre les mondes.

Pour les **logiques de Lukasiewicz**, le **principe de Curry-Howard** ou les **systèmes experts**, les formules peuvent être respectivement étiquetées par des rationnels, des λ-termes ou des réels. On peut considérer une **logique non monotone** comme un réseau de logiques monotones. C'est à dire, on associe une logique monotone particulière à chaque structure de formules.

4. Structuration de spécifications

Quand on emploie les méthodes logique et cadre logique, il est intéressant de combiner des spécifications, afin d'obtenir des spécifications structurées. Cette section présente différents langages développés pour structurer les combinaisons de spécifications.

Actuellement, peu d'outils d'inférence proposent de tels langages.

Une approche fondée sur la théorie des catégories [HST89]

[HST89] présente une **étude catégorique** des notions de relation de conséquence, logique et théorie.

Une *relation de conséquence* est un couple $(S,\vdash)$, où S est un ensemble de formules (langage) et $\vdash$ une relation binaire entre des ensembles finis de formules et des formules[8].

[8]La notion de relation de conséquence mono-conclusion de [Avr92] correspond à la relation $\vdash$.

Une *logique* L (système formel) est un foncteur $Sig^L \rightarrow CR$ d'une catégorie de signatures dans la catégorie des relations de conséquence. Concernant la partie objet de L, si Σ est une signature de Sig^L, on définit $(|L|_\Sigma, \vdash^L_\Sigma) = L(\Sigma)$. La notion de morphisme σ de signatures est utile pour combiner les théories, d'où la partie flèche.

Une *théorie* est un sous-ensemble de $|L|_\Sigma$ clos par $\vdash^L_\Sigma$.

Etant donnée une logique L, [HST89] propose d'utiliser **un langage pour structurer les théories**. La grammaire de ce langage est la suivante:

```
P ::=   (Σ,Φ)
    |   P1 ∪ P2
    |   translate P along σ
    |   derive P via σ
```

Σ étant une signature et Φ une partie de $|L|_\Sigma$, (Σ,Φ) représente la **théorie** définie par la fermeture de Φ par $\vdash^L_\Sigma$. L'union $\cup$ permet de **combiner** deux théories de même signature. *translate* permet de **renommer** les symboles d'une signature à l'aide d'un morphisme σ de signatures. *derive* permet de **cacher** certains symboles d'une signature à l'aide d'un morphisme σ de signatures.

De plus, ces primitives suffisent pour définir la notion très utile de théorie **générique**.

Par exemple, considérons une logique pour les équations algébriques. La théorie **Groupe** peut être représentée par une signature Σ définissant une loi de composition interne, un élément (neutre) et une fonction (symétrique), et un ensemble d'équations donnant leurs propriétés (caractérisant Φ). De manière similaire, la théorie **Commutatif** peut exprimer la commutativité d'une loi de composition interne.

On peut alors définir la théorie **Groupe Abélien** par:

```
Groupe Abélien = Groupe
   ∪ (translate Commutatif along σCG)
```

où σ_{CG} est le morphisme qui fait coïncider les lois de composition internes.

On peut aussi définir la théorie **Monoïde** par:

```
Monoïde = derive Groupe via σMG
```

où σ_{MG} est le morphisme qui permet "d'oublier" l'existence d'un élément symétrique.

La **recherche de preuves** dans une théorie structurée se ramène naturellement à celle de preuves dans les théories qui la composent.

Ceci est utile quand ces dernières sont plus simples ou mieux connues que la théorie structurée initiale.

Pour rechercher une preuve dans une logique, on emploie souvent une **logique auxiliaire** dont on possède une implémentation. On définit donc la notion de *représentation* d'une logique dans une autre, comme une transformation naturelle permettant de coder uniformément une signature de la première à l'aide d'une signature l'autre, tout en conservant la relation de conséquence image.

Avec certaines précautions, il est possible de **traduire une théorie structurée** de la logique objet en une théorie structurée de la logique auxiliaire, afin de travailler entièrement dans la logique auxiliaire.

LF est une logique particulière, permettant de spécifier une large classe de logiques [HHP89]. Une signature Σ de SIG^{LF} définit une logique $LF_\Sigma : Sig_\Sigma^{LF} \rightarrow CR$.

LF_Σ ne sert pas directement à représenter une logique objet, elle est d'abord restreinte à la portée d'un ensemble J caractérisant les jugements de base de cette logique. Une *présentation* de logique est donc définie comme un couple (Σ,J).

[HST89] étudie comment **structurer** les présentations de **logiques**, à la manière des théories structurées, afin d'utiliser un **langage similaire**.

Une approche ensembliste [LB92]

Pour la structuration de théories, une approche plus récente et plus simple est fondée sur la **théorie des ensembles**. Elle est mieux adaptée à l'implémentation.

Une *logique* ou *frame* est un 5-uplet (Sig, Sen, Hyp, Prf, R). Où Sig est le treillis des signatures possibles. Sen et Hyp sont des fonctions donnant respectivement l'ensemble des formules et le treillis des hypothèses possibles, à partir d'une signature Σ de Sig. En particulier, $Hyp(\Sigma)$ peut être l'ensemble des parties de $Sen(\Sigma)$. Prf est une fonction donnant l'ensemble des preuves d'une formule en fonction d'une signature Σ et d'un ensemble d'hypothèses $\Gamma \in Hyp(\Sigma)$. R est l'ensemble des renommages possibles sur Sig.

Étant donnée une logique L, une *théorie* est un triplet (Σ,Γ,C) où $\Sigma \in Sig$, $\Gamma \in Hyp(\Sigma)$, et C est un ensemble de théorèmes (φ,π) où $\varphi \in Sen(\Sigma)$ et $\pi \in Prf(\Sigma,\Gamma,\varphi)$. Contrairement à l'étude précédente [HST89], une théorie ne **contient** pas l'ensemble des théorèmes, mais seulement **l'ensemble** C des **théorèmes démontrés**.

Cette restriction rend **l'application de théories génériques décidable**, ce qui est important pour l'implémentation d'un langage de structuration de théories. De plus, elle formalise la notion de **bibliothèque de théorèmes démontrés.**

S-CLEAR est un langage de structuration de théories fondé sur la notion de frame. Sa syntaxe est indépendante du frame utilisé. La grammaire pour l'expression des théories est la suivante:

```
<T> ::=
/* Référence à une théorie déjà définie */
     x
/* Spécifie une théorie de base (Σ,Γ,C) */
     | theory(Σ,Γ,C)
/* Opérations décrites dans le texte */
     | join(<T>1,<T>2) | meet (<T>1,<T>2)
     | enrich(<T>,Σ,Γ,C) | select(<T>,Σ,Γ,C)
/* Application d'une théorie générique */
     | ([x1:T1,...,xn:Tn]T)(r,T'1,...,T'n)
/* Renomme les éléments de la signature d'une théorie */
     | rT
```

Avec:

```
Σ ::= sign Σ0 | rΣ
Γ ::= hyps Γ0 | rΓ
C ::= thms C0 | rC
r ::= r0 | ror'
```

join (meet) réalise l'union (l'intersection) de deux théories, en calculant la borne inférieure (supérieure) de l'ensemble des signatures contenant (contenues dans) les signatures de $<T>_1$ et $<T>_2$. *enrich (select)* étend (restreint) une théorie, en ajoutant (sélectionnant) des éléments ne constituant pas eux-mêmes des théories.

Notons que les opérations sur les théories correspondent à des opérations ensemblistes, **plus naturelles et plus faciles** à programmer que les opérations catégoriques de la section précédente [HST89] (morphismes, etc.).

Dans S-CLEAR La définition de théories **génériques** est uniquement syntaxique. Ces théories génériques ont plusieurs applications:

Elles permettent de **raisonner de manière abstraite**: On peut construire des théorèmes dans une théorie et les ajouter à une théorie existante, via un renommage reliant leurs signatures et leurs hypothèses. Par exemple, supposons que la théorie `Group` soit définie, la théorie générique suivante permet de réutiliser ses théorèmes:

```
Generic Inherit_Group(x::Group)
body
   enrich x by thms(Group)
end
```

Ensuite, si on définit une théorie `Int`, dont les théorèmes expriment qu'elle est un groupe relativement à une partie de sa signature. On peut lui ajouter tous les théorèmes démontrés dans `Group` par:

```
Inherit_Group(Int) with X,o,id,-1 as int,+,0,-
```

Elles permettent aussi de **factoriser la construction de théories**. Par exemple, la distributivité peut être définie de manière générique par:

```
Generic Distr(x::sign X,*:X²→X,y::sign X,+:X²→X)
body
   enrich join(x,y) by hyps
         ∀a,b,c∈X.a*(b+c)=(a*b)+(a*c)
         ∀a,b,c∈X.(b+c)*a=(b*a)+(c*a)
end
```

Elles permettent finalement la gestion de **bibliothèques de théories**, grâce à la possibilité de "fermer" une théorie, pour la réutiliser à un autre moment, via un renommage. Une *bibliothèque* est représentée par un ensemble de théories fermées.

```
Generic Close_Group
body
   ... /* Définition de Group */
end
```

Comme le langage de la section précédente, S-CLEAR permet non seulement de structurer les théories pour un système formel, mais aussi de structurer la définition de systèmes formels pour un cadre logique. L'article précédent prenait LF comme exemple, celui-ci prend CC.

5. Notion de Relation de Conséquence [Avr91]

Les cadres logiques décrits dans cette partie ne permettent pas de décrire tous les systèmes formels. Pour développer et employer d'autres cadres logiques, plus généraux, il est important de garder à l'esprit une vision des notions de logique et de calcul de preuve qui ne soient pas dépendantes de tel ou tel cadre logique.

[Avr91] présente clairement les notions de logique et de système formel et les classe selon différents critères.

Une *logique* est un couple constitué d'un ensemble de formules appelé *langage* et d'une relation de conséquence. Une *relation de conséquence* sur un ensemble de formules est une relation binaire $\vdash$ entre des multi-ensembles finis de formules tels que (A dénote une formule, Γ et Δ dénotent des multi-ensembles de formules):

(i) $A \vdash A$ *réflexivité restreinte*

(ii) $$\frac{\Gamma_1 \vdash \Delta_1, A \quad A, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}$$ *transitivité (ou règle de coupure)*

Les multi-ensembles permettent d'exprimer plus de notions que les ensembles (par exemple la "pertinence"). Et leur manipulation reste plus simple que celle des listes [CKB85].

Le plus souvent, les relations de conséquence considérées en théorie des preuves sont *compactes* (si on a $\Gamma \vdash \Delta$, on peut trouver $\Gamma_0 \subset \Gamma$ et $\Delta_0 \subset \Delta$ finis et tels que $\Gamma_0 \vdash \Delta_0$ [Mes87]). On emploie donc des multi-ensembles finis.

Une relation de conséquence est *régulière* si elle peut être considérée entre des ensembles, au lieu de multi-ensembles. C'est à dire $\Gamma, A \vdash \Delta \equiv \Gamma, A, A \vdash \Delta$ et $\Gamma \vdash \Delta, A \equiv \Gamma \vdash \Delta, A, A$.

Une relation de conséquence est *monotone* si elle satisfait la condition de monotonie:

(iii) $$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}$$ *monotonie*

On appelle *connectif*, un connectif logique du langage ou une **combinaison** de connectifs. Étant donnée une relation de conséquence, un *connectif interne* est un connectif que l'on caractérise par une équivalence entre deux séquents de formes particulières. Par exemple, + est une disjonction interne ssi:

$$\Gamma \vdash \Delta, A, B \equiv \Gamma \vdash \Delta, A + B$$

Un *connectif de combinaison* est un connectif que l'on caractérise par une équivalence entre un séquent et une paire de séquents de formes particulières. Par exemple, $\vee$ est une disjonction de combinaison ssi:

$$A \vee B, \Gamma \vdash \Delta \equiv A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta$$

L'équivalence qui caractérise chaque connectif des deux sortes peut être décomposée en deux règles, que l'on peut transformer systématiquement en règles qui ont la propriété de la sous-formule. Une fois transformées, ces règles correspondent aux règles **d'introduction** et **d'élimination** du connectif concerné, dans une représentation du type Gentzen.

En déduction naturelle, on définit souvent les connectifs à l'aide de règles d'introduction. Mais l'alternative qui consiste à considérer les équivalences précédentes est préférable.

Enfin, une **relation de conséquence** peut être **caractérisée** par trois critères: la **régularité**, la **monotonie** et les **connectifs** des deux sortes qu'elle admet.

Par exemple, la relation de conséquence de la logique intuitionniste est la relation de conséquence régulière, monotone, minimale, qui admet une disjonction, une conjonction, une contradiction et une implication internes. En remplaçant dans cette définition, l'implication interne par une implication forte interne, on obtient la logique classique.

Pour développer des preuves pour une relation de conséquence, on emploie la notion de système formel. Un système formel doit être correct, effectif et, si possible, complet.

Les représentations uniformes de type système axiomatique, Hilbert, déduction naturelle et Gentzen (purs ou non) permettent d'exprimer des classes de systèmes formels de plus en plus larges. Notons que LF est un cadre logique pour les systèmes de déduction naturelle réguliers, monotones, purs et mono-conclusion.

Pour ces représentations, les impuretés apparaissent essentiellement sous forme de conditions supplémentaires à l'application des règles. [Avr91] les classe en différents niveaux, selon si elles considèrent la structure des multi-ensembles, celle des formules ou celle des preuves des prémisses.

Parfois, il est utile de considérer des représentations non uniformes de systèmes formels. Par exemple, LF permet de représenter plusieurs relations de conséquence en même temps. Une autre approche intéressante consiste à représenter des séquents de séquents.

6. Discussion

ATINF et GLEF doivent permettre à leurs utilisateurs de travailler dans des logiques variées. Une approche séduisante est d'**employer un cadre logique** pour représenter différents systèmes formels, avec leurs preuves. Ce cadre logique assurerait la correction des preuves considérées et faciliterait la communication entre les outils d'inférence et l'utilisation simultanée de plusieurs logiques.

Bien entendu, on ne doit pas tenter de définir un nouveau cadre logique. Les cadres logiques existants ont des fondements théoriques solides. L'étude des propriétés formelles d'un autre cadre logique est hors du cadre de ce travail.

LDS est un cadre logique très intéressant. Il permet notamment de traiter la relevance. Mais il n'est pas encore totalement formalisé.

Les descendants du projet AUTOMATH [dBr80], LF et CC sont des cadres logiques fondés sur des λ-calculs typés similaires. En fait, le calcul de LF est inclus dans celui de CC [HHP89], [Bar92]. Toutefois, leurs approches sont différentes. En CC, des constantes définissent les

connectifs logiques de manière constructive. En LF, elles définissent le langage et les règles d'inférence grâce aux notions de jugement de base et de jugements d'ordre supérieur (hypothétique et schématique).

Les **règles de jugement de typage** de CC, sa **syntaxe** concise et homogène, et sa technique de synthèse automatique d'arguments **implicites** [CH88] pourraient être employées dans GLEF. D'autre part, la notion de **jugement** introduite pour LF [HHP89] serait particulièrement utile pour représenter simultanément plusieurs relations de conséquence, comme, par exemple, en logique modale.

Nous avons choisi de représenter les systèmes formels selon le principe de LF, en employant le calcul de CC. Comme de toute façon, l'adéquation du codage de chaque système formel doit être vérifiée, ce choix est justifié. Mais employer CC permet de représenter directement des preuves constructives ou des programmes.

L'intégration à la syntaxe d'artifices de présentation semblables aux formats de CC serait inutile. En effet, dans le développement de l'éditeur de preuves, le formalisme employé ne devrait pas servir directement à la présentation. Nous préférons utiliser un langage de **présentation** graphique et mieux adapté. En effet, la représentation des preuves dans les cadres logiques n'est pas très naturelle et leur forme est très éloignée de leur présentation usuelle.

En complément du cadre logique, on pourrait employer un langage de structuration de théories comme ceux décrits dans [HST89] ou [LB92]. Ce langage faciliterait la **spécification de systèmes formels** et la **gestion de bibliothèques** de logiques, de théorèmes et de preuves.

Gardons à l'esprit que les cadres logiques existants ne permettent pas encore de représenter tous les systèmes formels utiles. La réalisation de GLEF devra donc être suffisamment modulaire, afin **d'évoluer vers des cadres logiques plus puissants** lors de leur découverte [Avr91]. En étudiant les représentations des systèmes formels et des preuves dans les cadres logiques, il est envisageable de mettre automatiquement à jour les bibliothèques déjà construites, par exemple, à l'aide de morphismes de signatures [HST89].

B) Différents outils d'inférence

La capacité de calcul des ordinateurs permet une **application efficace** des formalismes décrits dans la section précédente. Cette application est utile, non seulement pour leur **utilisation**, mais aussi lors de leur **développement**.

On pense souvent que tout programme doit être réalisé à partir de spécifications précises. Ceci est vrai, dans un contexte d'utilisation. Les programmes sans spécifications sont toutefois très utiles pour le développement expérimental [Pau88].

Un *outil d'inférence* est un programme capable de manipuler des preuves. La plupart des outils d'inférence existants ont été les objets de développement expérimentaux. Hétérogènes, ils ont, en leur temps, apporté un lot d'idées nouvelles, avant d'être sous-utilisés voire totalement abandonnés [Pau88].

Mais aucun système ne regroupe les capacités des différents outils d'inférence. S'il fallait tout reprogrammer, la réalisation d'un tel système serait un travail colossal.

On pourrait plutôt les utiliser tels quels, dans un **cadre** leur apportant une interface utilisateur simple et homogène, assurant la correction des preuves et leur intercommunication. Ce cadre pourrait être distribué avec les outils d'inférence, pour l'éducation ou une utilisation plus intensive.

Pour développer un tel cadre, nous avons étudié les différentes formes de programmes correspondant à la définition d'outil d'inférence.

Leurs fonctions principales sont la **vérification**, la **recherche** automatique ou interactive, la **mémorisation** et la **transformation** de preuves. Les programmes existants sont souvent des hybrides de ces fonctions. Par exemple, un vérificateur automatique capable de traiter des pas d'inférence non élémentaires ou un démonstrateur interactif capable de compléter automatiquement une preuve partielle, ont des capacités de démonstration automatique. Par ailleurs, de nombreux outils d'inférence intègrent leur propre outil de mémorisation de preuves.

Chaque réalisation correspond à l'une des trois méthodes d'implémentation de systèmes logiques: **théorie**, **logique** ou **cadre logique**. Cette correspondance est **intentionnelle**, car une même réalisation peut servir à implémenter des systèmes logiques selon différentes méthodes. Par exemple, les systèmes formels fondés sur la théorie constructive des types peuvent servir de cadres logiques. **NUPRL** [CKB85] permet ainsi d'implémenter des systèmes logiques selon la méthode cadre logique.

Notons que la réalisation d'un démonstrateur automatique est souvent difficile pour les méthodes théorie et logique et extrêmement difficile pour la méthode cadre logique. Même celle d'un vérificateur automatique ou d'un démonstrateur interactif peut être difficile pour la méthode cadre logique.

Cette section présente différents outils d'inférence, classés selon leur fonction principale. Toutefois, aucun vérificateur purement automatique n'est présenté. En effet, il n'y a pas de différence fonctionnelle entre un vérificateur automatique pur et le formalisme qu'il implémente, on se rapportera donc à la section précédente.

PC [McC62]

En 1962, Mac Carthy envisageait déjà d'utiliser les ordinateurs pour construire, vérifier et transformer des preuves. L'ordinateur pourrait devenir un assistant puissant et peu sujet aux erreurs, pour le **développement de programmes** et la **recherche en mathématiques**.

Pour **réduire** le travail de **programmation**, la recherche et la vérification automatique de programmes remplaceraient leur écriture manuelle et leur test. Rappelons en effet que les programmes peuvent être vus comme des preuves de leurs spécifications.

Pour **aider** le **mathématicien**, les preuves seraient recherchées, complétées ou vérifiées de manière automatique.

Cette utilisation est fondée sur l'existence de systèmes formels dans lesquels les preuves peuvent être exprimées de manière suffisamment brève et naturelle.

Un *système formel* (logique) est composé de deux parties, un langage et une relation.

Le *langage* est un ensemble de chaînes de caractères, caractérisable par un prédicat calculable $g(s)$. En pratique, le calcul de $g(s)$ correspond à une analyse grammaticale.

La *relation* est caractérisable par un prédicat, calculable mais pas toujours totale, $v(s,p)$ où s est une formule et p est une information de forme quelconque. Un théorème est une formule s telle que $\exists p\ v(s,p)$. Une *preuve* est une information p suffisante pour affirmer, en calculant $v(s,p)$, qu'une formule s est un théorème.

Dans la plupart des systèmes formels, $v(s,p)$ exprime que p est une liste de formules terminée par s, telle que chaque formule soit un axiome ou une conséquence immédiate des formules précédentes.

PC (Proof Checker) est un système écrit en LISP, il constitue un environnement pour la réalisation de **vérificateurs de preuves, pour différents systèmes formels**. Un *vérificateur de preuves* est un programme calculant le prédicat $v(s,p)$.

Une règle d'inférence est représentée par un prédicat calculable mais pas toujours total: `rule(premisses, conclusions, parameter)`. L'argument `parameter` donne des informations supplémentaires pour l'application de la règle d'inférence. Par exemple, il peut indiquer quelles sont les formules A et B d'une application du modus-ponens. Admettre les prédicats partiels calculables ne change pas la classe des règles d'inférences pouvant être définies. En effet, toute règle d'inférence partielle peut être rendue totale en limitant le nombre de ses pas de calcul par une borne ajoutée comme argument, dans `parameter`.

Un *système formel* est représenté par un ensemble de règles d'inférence caractérisé par un prédicat `is_rule(r)`.

Une *preuve* n'est pas une simple liste de formules. C'est un 6-uplet `(state, premgen, conclgen, paragen, rulegen, stategen)`. Les éléments `rulegen`, `premgen`, `conclgen`, `paragen` sont des fonctions qui calculent une règle d'inférence et ses différents arguments en fonction d'un état `state`. L'élément `stategen` est une fonction qui calcule une nouvelle valeur de `state`, pour continuer la vérification.

PC est codé comme un prédicat: `proofcheck(prem,concl,proof)`. L'argument `prem` sert à mémoriser les conclusions intermédiaires dans la vérification itérative. La vérification de v(s,p) correspond à un appel de `proofcheck` avec `prem` vide, s dans `concl` et p dans `proof`.

PC constitue un cadre très général, les représentations choisies pour les preuves et les systèmes formels facilitent la réalisation d'un certain nombre de notions:

Les preuves peuvent être considérées comme des programmes générant des applications de règles d'inférence. Ainsi, on pourrait employer des **procédures de décision**, des **démonstrateurs automatiques** ou des **tactiques** et des **opérateurs de tactiques**, ces différentes notions sont présentées plus loin.

Une **règle dérivée** peut être considérée comme un pas de preuve unique: Il suffit d'admettre `proof_check` comme règle d'inférence, en modifiant `is_rule`. Notons que la forme `proof_check` est effectivement celle d'une règle d'inférence.

Certains systèmes formels permettent de **déduire des règles d'inférence**: Il suffit d'autoriser, dans `is_rule`, l'utilisation de ces règles d'inférence, mémorisées dans les prémisses de `proof_check`.

Une certaine forme de **déduction naturelle** peut être incorporée en ajoutant une règle d'inférence correspondant au théorème de la déduction. Cette règle d'inférence permet de déduire $P \Rightarrow Q$ de prémisses qui permettent de déduire Q si on leur ajoute P.

Le principe de PC est assez séduisant. Néanmoins, l'utilisateur doit trouver une représentation interne pour les preuves, programmer des fonctions capables de fournir les applications de règles d'inférence dans l'ordre où elles interviennent et programmer les règles d'inférence et les procédures de décision au plus bas niveau. Cette programmation est à refaire pour chaque système formel utilisé.

VT [DS79]

PC est fondé sur une notion de système formel très générale, mais avec un faible niveau de formalisation. Il oblige l'utilisateur à programmer la plupart des notions usuelles relatives aux systèmes

formels. Son utilisation est sûre, dans la mesure où cette programmation est correcte.

En pratique, on préfère utiliser un système formel capable **d'exprimer de manière sûre et naturelle un maximum de notions mathématiques**.

D'après le théorème d'incomplétude de Gödel, tout système formel "intéressant" est incomplet. Son extension par programmation peut donc toujours être envisagée.

[DS79] ne présente pas une réalisation concrète, mais la possibilité **d'étendre** un système formel par **programmation** tout en assurant la **correction**.

Pour réduire la taille des preuves formelles, ou réduire le temps de calcul, un vérificateur de preuves doit être **extensible** dans plusieurs directions. Cependant, les extensions doivent préserver sa correction, cette propriété s'appelle *stabilité*.

VT est un vérificateur de preuves, non implémenté, qui permet l'ajout de règles méta-théoriques et l'utilisation de programmes externes, tout en assurant la stabilité.

L'ajout d'axiomes, possible dans certains vérificateurs de preuves, peut **réduire la taille des preuves formelles** et augmenter la classe des formules décidables, mais la stabilité est difficile à vérifier. D'autre part, les règles d'inférence dérivées assurent la **stabilité**, mais ne suffisent pas toujours à maintenir la taille des preuves formelles en rapport avec celle des preuves informelles. L'ajout de règles méta-théoriques, au sens de VT permet de rester plus proche des preuves informelles tout en assurant la stabilité.

Notons aussi qu'utiliser des programmes externes accélère la vérification, lorsque certaines théories ont été construites.

VT est fondé sur un système formel *FS* pour la logique du premier ordre, avec notions d'ensembles, d'entiers et de listes et les fonctions associées, de ω l'ensemble des entiers et de fonctions de calcul de la taille des termes, utiles pour définir les inductions.

Le système formel *LFS* est construit comme FS, mais sans ω. Les formules de LFS sont décidables. FS suffit à formaliser tous les calculs finis utilisés pour obtenir les valeurs de vérité des formules de LFS: **FS est complet pour les formules de LFS**.

Selon la méthode de Gödel, les formules de FS peuvent être codées par des entiers ($\overline{\varphi}$ dénote le code de la formule φ). Grâce à ce codage, on peut écrire des formules de LFS décrivant la syntaxe de FS: $\mathrm{TERM}(X)$, $\mathrm{FORM}(X)$ et $\mathrm{SENT}(X)$, indiquant respectivement que X est le code d'un terme, d'une formule ou qu'il est un code valide.

Ces formules permettent de construire deux formules de FS: TRUE(X) et FALSE(X), qui indiquent respectivement que X est le code d'une formule vraie ou fausse de LFS. Notons qu'il est impossible de construire de telles formules pour les formules de FS.

VT est **un vérificateur pour les preuves dans FS**. Il gère un ensemble VA (Verified Assertions) de formules de FS et un ensemble RI (Rules of Inference) de formules de LFS. Si Φ appartient à RI, $\Phi(\alpha)$ vrai exprime que α est de la forme $[\overline{\lambda_1},\ldots,\overline{\lambda_n},\overline{\varphi}]$ et φ est une conséquence de $\lambda_1,\ldots,\lambda_n$ selon la règle $\Phi(X)$. On suppose que $\Phi(\alpha)$ ne change pas de valeur de vérité si on ajoute des $\overline{\lambda_i}$ à α. Initialement VA est vide et RI contient les axiomes et les règles d'inférence de base de FS.

Une *preuve* est une formule de FS de la forme $\pi=[\overline{\varphi_1},\ldots,\overline{\varphi_n}]$telle que pour chaque $0\leq i<n$, φ_{i+1} appartient à VA ou φ_{i+1} est une conséquence de $\varphi_1,\ldots,\varphi_n$, selon une règle de RI.

Ainsi, on peut construire les formules PROVE(X,Y) de LFS, indiquant que Y est une preuve de la formule de code X, et THM(X) de FS, indiquant que X est le code d'un théorème.

VT peut être utilisé selon six modes ($\varphi,\pi \in$ FS, $\Phi \in$ LFS):

- Le mode I (bibliothèque) permet de vérifier qu'une formule φ appartient à VA.
- Le mode II (vérification) permet de vérifier qu'une preuve π prouve une formule φ. Il consiste à tester PROVE$(\overline{\varphi},\pi)$, dont le calcul se termine.
- Le mode III (ajout d'assertion) permet d'ajouter une formule φ à VA,
- Le mode IV (ajout de règle) permet d'ajouter une formule Φ à RI, étant donnés une justification φ et une preuve π. VT vérifie d'abord si φ est bien de la forme:

$$\forall X\Big(\Phi(X)\Rightarrow\big(\forall j_{1\leq j<\mathrm{Len}(X)}\,\mathrm{THM}(X(j))\big)\Rightarrow\mathrm{THM}\big(X(\mathrm{Len}(X))\big)\Big)$$

  Puis teste PROVE$(\overline{\varphi},\pi)$, pour accepter ou rejeter la nouvelle règle Φ.
- Les modes V et VI permettent d'ajouter de nouveaux symboles de fonctions ou de prédicats. Une règle de réécriture (δ-réduction) est alors ajoutée à VA.

On démontre facilement la **stabilité** de VT sous ses différents modes d'utilisation.

Pour utiliser un système de calcul externe M, on peut coder ses programmes et leurs données par des constantes de LFS. Ainsi, on suppose posséder deux formules de LFS: IS_MPROG(X) indiquant que X est un code de programme pour M et GOESBY(*program*,*input*,*output*,*path*) décrivant le fonctionnement de M.

GOESBY permet de construire une formule de FS, YIELDS(*program*,*input*,*output*), décrivant l'appel de programmes.

On désire utiliser des programmes assurant qu'une formule φ de FS est vraie (fausse) si leur sortie est true (false) quand leur entrée est le code $\overline{\varphi}$ de la formule. Notons que ces programmes peuvent boucler ou avoir d'autres sorties. **Avant d'utiliser un programme de code β, on doit vérifier les formules de FS suivantes**:

$$\forall X\ \mathrm{SENT}(X) \wedge \mathrm{YIELDS}(\beta,X,\mathrm{true}) \Rightarrow \mathrm{TRUE}(X)$$
$$\forall X\ \mathrm{SENT}(X) \wedge \mathrm{YIELDS}(\beta,X,\mathrm{false}) \Rightarrow \mathrm{FALSE}(X)$$

Un système de vérification de programmes peut faciliter la vérification de ces formules. Un tel système est décrit par une formule de LFS, IS_VERIF(X,Y,Z), indiquant que X est le code d'une preuve de programme pour le programme de code Y dont les entrées et les sorties doivent vérifier une formule de FS, $\Phi(X,Y)$ de code Z. **Avant d'utiliser un système de vérification** IS_VERIF**, on doit vérifier la formule de FS suivantes**:

$$\begin{aligned}&\forall X,PROG,PROP,INP,OUP,ASRT\\ &\quad[(\mathrm{IS\_VERIF}(X,PROG,PROP) \wedge \mathrm{IS\_MPROG}(PROG)\\ &\qquad\qquad \wedge \mathrm{YIELDS}(PROG,INP,OUP) \wedge \mathrm{SUBST}(PROP,INP,OUP,ASRT))\\ &\qquad \Rightarrow \mathrm{TRUE}(ASRT)]\end{aligned}$$

Néanmoins, après vérification de chaque programme, il reste à vérifier les formules de FS suivantes:

$$\forall X\ \mathrm{SENT}(X) \wedge \Phi(X,\mathrm{true}) \Rightarrow \mathrm{TRUE}(X)$$
$$\forall X\ \mathrm{SENT}(X) \wedge \Phi(X,\mathrm{false}) \Rightarrow \mathrm{FALSE}(X)$$

Ainsi, l'introduction de programmes et de vérificateurs de programmes de plus en plus puissants peut se faire par couches successives.

1. Méthode théorie

Réaliser un outil d'inférence pour une théorie mathématique bien connue est relativement facile. Néanmoins, réaliser un tel outil pour l'ensemble des théories mathématiques, même bien connues, est un travail colossal.

Présentons quelques outils d'inférence développés pour des théories mathématiques.

Une expérience dans l'enseignement [Sup84]

Professeur à l'université de Stanford, l'auteur de cet article est l'un des pionniers de l'utilisation de démonstrateurs automatiques ou interactifs. Dès 1963, il se servait de ces outils pour enseigner la logique et l'algèbre élémentaires.

En 1972, le cours de logique élémentaire a été **porté entièrement** sur ordinateur. Il est maintenant utilisé dans d'autres universités et connaît un grand succès auprès des élèves. En 1974, ce fut le cours sur la théorie des ensembles. Celui-ci est régulièrement utilisé, mais la construction des preuves manque de naturel.

L'enseignement de ces cours est pris entièrement en compte par l'ordinateur. Au fil de son apprentissage, l'élève est conduit à démontrer des théorèmes, en guise d'exercices.

En partant de la conclusion, il construit une preuve pas à pas ("top-down") en appliquant des règles d'inférence ou en utilisant des théorèmes déjà démontrés (règles d'inférence dérivées), de manière interactive.

Un démonstrateur automatique, utilisé comme **une boîte noire**, permet de **vérifier** les buts non développés.

Par exemple, le cours sur la théorie des ensembles est organisé autour de 600 théorèmes, l'élève en démontre en moyenne entre 30 et 50.

Comparer la façon dont une preuve est construite par un élève et sa forme finale est intéressant. Le processus de construction est souvent anarchique et **hautement interactif**, même si la preuve finale est mieux organisée.

Cette expérience a permis à l'auteur de définir les besoins de l'élève et du professeur en matière d'enseignement mathématique assisté par ordinateur.

Du point de vue de l'élève, un cours doit être **accessible** et ne pas nécessiter de connaissances en informatique ou en programmation. Un moyen simple de tester une interface est d'observer comment les élèves construisent les preuves.

Pour cela, une interface de **contrôle** est plus ergonomique, plus simple, plus concise et plus flexible qu'un **langage** mathématique.

Le démonstrateur automatique employé devrait être capable de résoudre les pas de preuve qui paraissent **faciles** aux élèves de différents niveaux, leur vérification pas à pas étant une perte de temps. La connaissance des théories enseignées devrait permettre l'amélioration des démonstrateurs automatiques.

Des **heuristiques** seraient très utiles pour **guider** l'élève bloqué pendant la construction d'une preuve. Mais la réalisation de telles heuristiques est extrêmement difficile.

Pour certains cours, l'utilisation de **graphiques** est aussi très importante.

Le point de vue du professeur est tout aussi important que celui de l'élève, si l'on souhaite le **développement** de tels systèmes d'enseignement.

Une **structure** de cours **générique** et **souple** devrait leur être proposée, sans nécessiter de connaissances en informatique ou en programmation.

Comme pour les élèves, une interface de **contrôle** serait préférable à un **langage** de spécification.

De plus, il serait souhaitable d'étudier comment faciliter la réalisation des **heuristiques** destinées à guider les élèves.

Ces besoins sont justifiés par le fait qu'un certain nombre d'autres cours sont **susceptibles** d'être entièrement informatisés. Toutefois, aucun d'entre eux ne dépasse le niveau du lycée:

- Géométrie élémentaire (à présenter de façon constructive).
- Algèbre linéaire.
- Calcul différentiel et intégral.
- Équations différentielles.
- Introduction à l'analyse.
- Introduction aux probabilités.
- Théorie des automates.

CAP: Tentative de formalisation complète [Hir86]

CAP (Computer Aided Proof) est un projet ambitieux visant à réaliser un vérificateur automatique de preuves, pour l'ensemble des théories mathématiques connues. Un tel outil serait particulièrement utile pour l'enseignement.

Les preuves à vérifier sont des preuves formelles, dont certaines parties peuvent être **omises**. Le vérificateur doit donc avoir des capacités de **démonstration automatique**.

La vérification serait simplifiée par des pas d'inférence inductive, grâce à l'utilisation d'une **base de connaissances**. Cette-ci pourrait être elle-même mise à jour par apprentissage inductif.

L'utilisation d'une base de connaissances constituerait une approche méta-théorique pouvant traiter différentes théories mathématiques simultanément[9].

L'algèbre linéaire a été la première théorie mathématique réalisée (*CAP-LA*). Le langage de description des preuves (*PDL*) est équivalent à la déduction naturelle de Gentzen. Il est cependant présenté sous une forme indentée utilisant un langage naturel (pseudo langue naturelle).

Le système est constitué d'un éditeur structuré pour PDL, d'une base de connaissance et du vérificateur. L'utilisateur écrit une preuve en PDL et la soumet au vérificateur. Celui-ci utilise un **système d'inférence** pour les déductions et un **système de réécriture** pour les égalités. Il peut utiliser ou mémoriser des règles d'inférences ou des égalités dans la base de connaissance.

CAP-LA a permis d'écrire et de vérifier à 90%, le livre "Matrices and Determinants" de S.Furuya.

La poursuite du projet CAP devrait concerner l'amélioration de l'interface, celle de CAP-LA (traitement de l'ordre supérieur, des symétries, etc.), et **l'extension** à d'autres branches des mathématiques, comme *SDG*, une théorie géométrique récente, ou *QJ* un système formel constituant simultanément une logique et un langage de programmation.

2. Démonstrateurs automatiques

Un système formel étant donné, un *démonstrateur automatique* est un programme **partiel** capable de dire si un jugement de base est vrai ou faux, et éventuellement de fournir une preuve. Souvent, on considère la forme de jugement de base "est un théorème".

Quand la forme de jugement de base considérée est décidable, une *procédure de décision* est un démonstrateur automatique **total**.

Cette section décrit un démonstrateur automatique très connu, OTTER. Ce démonstrateur automatique peut considérer des systèmes formels implémentés selon la méthode logique, dans la logique des prédicats du premier ordre.

OTTER [McC90]

OTTER est un démonstrateur automatique fondé sur la résolution pour la logique des prédicats du premier ordre, avec égalité. Il utilise plusieurs règles d'inférence, comme la factorisation, différentes formes de résolution (résolution binaire, hyper-résolution, UR-résolution) et la paramodulation.

---

[9]Attention, employer simultanément différentes axiomatiques peut poser des problèmes!

OTTER met la formule à démontrer, entrée par l'utilisateur, **sous forme clausale**. De nombreuses options servent ensuite à choisir les règles d'inférence à utiliser parmi celles proposées, restreindre leur application, traiter l'égalité, ordonner les termes et les clauses ou contrôler la recherche. Notons que la stratégie obtenue n'est pas toujours complète.

OTTER a été implémenté en C, il est relativement rapide et gère la mémoire de manière efficace. Il a permis de trouver des preuves non guidées de plusieurs problèmes réputés difficiles, et a été utilisé pour trouver, souvent de façon interactive, la solution de certains problèmes ouverts en mathématiques [Wos88].

3. Démonstrateurs interactifs

Pour la logique propositionnelle, qui est décidable, on sait que le problème de la satisfaisabilité est NP-complet. Pour la logique du premier ordre, qui est semi-décidable (la résolution plus la factorisation forment un calcul complet), non seulement on ne peut pas borner le temps de calcul, mais on ne peut pas non plus borner la mémoire utilisée.

Toutefois, on peut réaliser un démonstrateur pratique, si l'on permet le contrôle précis de celui-ci par l'utilisateur. Un tel démonstrateur s'appelle *démonstrateur interactif*. Notons qu'un démonstrateur interactif peut toujours appeler des démonstrateurs automatiques, pour réduire le travail de l'utilisateur.

a) Dans une logique fixe (méthode logique)

EKL [Ket84], [KW83]

EKL est un démonstrateur interactif fondé sur une logique dont le langage est très **expressif**. Tout en restant **correct**, EKL considère les problèmes de **complétude**, de **décidabilité** et de **complexité** comme secondaires. En effet, très peu de théories mathématiques usuelles sont décidables. Laisser les procédures de décision **filtrer les entrées** qu'elles peuvent traiter est préférable.

La logique utilisée est composée de la logique d'ordre fini, munie d'un λ-calcul typé, de notions d'ensembles, d'entiers et de listes, de schémas d'induction, d'une instruction conditionnelle et d'opérateurs méta-théoriques $\uparrow$ et $\downarrow$. L'opérateur $\uparrow$ est une forme de quotation, il permet à EKL de manipuler ses propres formules, en les considérant comme des constantes. L'opérateur inverse $\downarrow$ est une forme d'évaluation. En pratique, ces opérateurs méta-théoriques sont relativement peu utilisés car souvent, l'ordre supérieur peut les simuler. Par exemple, le schéma d'induction:

$$\uparrow P(0) \wedge (\forall n.P(n) \Rightarrow P(n')) \Rightarrow \forall n.P(n)$$

peut aussi s'écrire:

$$\forall P.P(0) \wedge (\forall n.P(n) \Rightarrow P(n')) \Rightarrow \forall n.P(n)$$

Entre chaque commande, l'état de EKL est matérialisé par un contexte courant et une preuve courante. Un *contexte* est un ensemble de déclarations associant chacune un atome à une signification syntaxique. Une *preuve* est un ensemble de lignes inter-dépendantes.

Les commandes de EKL servent à **augmenter le contexte courant** par ajout de nouveaux axiomes, symboles ou faits,... à **créer de nouvelles lignes** avec dépendances, par spécialisation de variables universelles, réécriture ou procédure de décision,... à **ajouter des dépendances** par analyse de cas, introduction conditionnelle ou lien direct,... à **détruire** ou à **copier** des lignes avec leurs dépendances...

Le système est écrit en LISP. Celui-ci sert aussi de langage de programmation sous EKL. LISP n'étant pas un langage fortement typé, des informations de types sont associées à chaque atome:

- Le *type* restreint l'utilisation des atomes pour la construction de termes. L'algèbre des types est générée par les constructeurs produit, disjonction, application et liste. Le constructeur liste sert à écrire des **fonctions d'arité variable**.
- La *sorte* indique si l'atome est défini par l'utilisateur ou s'il s'agit d'un élément du système.
- Le *type syntaxique* indique si l'atome est considéré comme une variable, une constante, un opérateur de liaison, une définition, etc..

Par exemple, le constructeur d'ensemble $\{x|P(x)\}$ est un opérateur de liaison de type $\langle ground \rangle \otimes \text{truthval} \rightarrow \text{ground}$.

Souvent, les discours mathématiques utilisent des manipulations logiques relativement simples, mais des réécritures plus nombreuses et plus complexes. Le noyau de EKL est donc **un puissant système de réécriture**.

Ce système de réécriture utilise l'algorithme d'unification d'ordre supérieur de Huet. D'après [Ket84], "on peut montrer" que cet algorithme converge quand toutes les variables d'ordre supérieur apparaissent dans une seule des deux formules à unifier (ce qui ne semble pas encore été fait).

EKL permet de contrôler les réécritures de manière précise. En effet, $P(x) \Rightarrow A=B$ peut s'interpréter de différentes manières (remplacer

$P(x) \Rightarrow A=B$ par vrai, remplacer B par A quand $P(x)$ est vrai, etc.). Chaque règle de réécriture est donc définie par un terme de type:

<fact>⊗<mode of use>→<rewriting procedure>

<fact> sert à filtrer les règles de réécriture applicables dans un environnement donné. *<mode of use>* spécifie l'une des différentes conditions standard d'application. *<rewriting procedure>* définit les variables à unifier, la partie gauche et la partie droite de la règle, et un ensemble de conditions supplémentaires à son application, sous forme de formules EKL ou de procédures LISP.

Il effectue automatiquement de nombreuses simplifications telles que βη-réduction, élimination de quantificateurs, évaluation de formules, traitement des opérateurs associatifs, appels d'attachements fonctionnels et simplifications méta-théoriques.

L'utilisateur peut en effet utiliser des attachements fonctionnels. Pour assurer sa stabilité, EKL possède une liste standard de fonctions. Quand l'utilisateur définit un symbole dont le nom correspond à celui d'une fonction de cette liste, la fonction est automatiquement attachée au symbole.

Les simplifications méta-théoriques consistent à remplacer $\uparrow\downarrow t$ par t, quand les variables libres de t ne sont pas capturées, ou à remplacer $F(\uparrow t_1, \ldots \uparrow t_n)$ par $\uparrow t$, quand F est attachée à une fonction LISP et t est le résultat de l'application de F à $t_1, \ldots t_n$.

EKL a été utilisé pour prouver de nombreuses propriétés des fonctions de base de LISP (APPEND, REVERSE, etc.), et pour vérifier le livre d'analyse de Landau, pour être comparé à AUTOMATH.

LCF [Mil85], [Pau85], [GMW79]

Initialement destiné à la vérification de programmes et de circuits, *LCF* (Logic for Computable Functions) est l'un des premiers démonstrateurs interactifs.

L'environnement LCF est écrit en *ML* (Meta Language), un langage de programmation fonctionnel interprété, développé pour l'occasion. C'est un langage d'ordre supérieur, une fonction peut être passée en paramètre, retournée comme valeur ou incluse dans une structure de données. Un mécanisme permet de gérer simplement les erreurs et les exceptions.

ML impose surtout une **discipline de types polymorphes** apportant la **sécurité** des types tout en gardant la **souplesse** des langages non typés. Il a été **développé et utilisé plus tard** avec le succès que l'on connaît.

La logique des preuves de LCF, *PPLAMBDA* (Polymorphic Predicate λ-calculus) est composée de la logique du premier ordre avec λ-calcul et égalité. Fondé sur la théorie des fonctions calculables de Scott, elle est particulièrement adaptée à la formulation et à la démonstration de propriétés sur les algorithmes.

Le calcul utilisé est un système de déduction naturelle avec des règles classiques d'introduction et d'élimination. Le langage et le calcul sont directement écrits en ML [CHP86]. ML a donc servi en même temps de **langage de programmation** pour LCF et de **méta-langage** pour PPLAMBDA.

La discipline de types de ML permet de distinguer avec certitude les valeurs représentant des termes, des formules ou des théorèmes. Par exemple, la valeur `1=0` de type formule, ne doit jamais acquérir le type théorème. Pour assurer cette **consistance**, les théorèmes ne peuvent être construits qu'avec les fonctions ML qui représentent des règles d'inférence.

Dans l'environnement LCF, sous l'interprète ML, appliquer des règles d'inférence à des théorèmes, pour obtenir d'autres théorèmes, permet de construire des preuves de façon "bottom-up".

Cependant, les notions de buts et de tactiques de LCF permettent de construire des preuves de façon "top-down", souvent de manière plus naturelle. Un *but* est une formule à démontrer, le but initial est donc le théorème à démontrer. Une *tactique* permet de **décomposer un but en buts plus simples**. Notamment, on peut écrire une tactique correspondant l'application de chaque règle d'inférence à l'envers. Le principe consiste à décomposer chaque but avec une tactique, jusqu'à ce qu'il n'y ait plus de buts. Exemple de décomposition de but en déduction naturelle:

```
Γ ⊢ A ∧ B        and_intro
   1. Γ ⊢ A
   2. Γ ⊢ B
```

Formellement, les buts sont représentés par des objets ML de type `goal`. Ces buts peuvent être *satisfaits* par des événements, représentés par des objets ML de type `event`. Les types ML suivants définissent la notion de tactique[10]:

```
tactic = goal → (goal list # validation)
validation = event list → event
```

La fonction de validation produite par une tactique permet de construire un événement qui satisfait un but à partir d'événements

[10] → représente le type fonctionnel, # représente le produit cartésien.

satisfaisant les buts qui le décomposent. Ainsi, les fonctions de validation produites par les tactiques appliquées à un but de manière "top-down", peuvent être composées pour construire l'événement correspondant de manière "bottom-up".

En pratique, les événements sont des théorèmes et la relation de satisfaction est l'égalité. Les règles d'inférence et les règles dérivées sont des exemples de fonctions de validation. Quand il n'y a plus de buts, la composition des fonctions de validation **construit l'objet de type théorème** qui correspond au but initial.

Les tactiques peuvent être **composées** avec des *opérateurs de tactiques*, pour automatiser leur appel manuel répétitif:

- tac_1 THEN tac_2 applique tac_2 à chaque but généré par tac_1.
- tac_0 THENL $[tac_1;\ldots;tac_n]$ applique respectivement tac_i à chaque but généré par tac_0, si leur nombre est n, il échoue sinon.
- tac_1 ORELSE tac_2 applique tac_1, applique tac_2 quand tac_1 échoue.
- REPEAT tac_1 applique tac_1 jusqu'à son échec sur tous les buts générés.

LCF propose un certain nombre de tactiques standard.

En particulier, un simplificateur permet d'appliquer des règles de réécritures. Une tactique spécifique sert à appeler ce simplificateur. Des opérateurs similaires à ceux des tactiques permettent de combiner les fonctions de réécriture.

LCF permet de sauvegarder les théories développées dans des fichiers. Ces fichiers peuvent être réédités ensuite. Notamment, ils peuvent servir à construire d'autres théories, en structurant celles-ci. Des opérateurs permettent de combiner des théories, d'instancier une théorie ou de cacher des parties d'une théorie.

LCF a permis de construire de nombreuses preuves, notamment pour vérifier:

- La correction de compilateurs, en sémantique dénotationnelle.
- Des programmes, notamment d'analyse grammaticale et d'unification.
- Des circuits, dont certains ont pu être fabriqués.

HOL (Higher Order Logic) [Gor87] est une version de LCF pour la logique d'ordre supérieur, utilisée en particulier pour la vérification de circuits.

NUPRL [CKB85], [C+86]

NUPRL (Nearly Ultimate Proof Refinement Logic) est un démonstrateur interactif destiné aux preuves mathématiques. Son architecture est similaire à celle de LCF, mais son interface est plus moderne (souris, fenêtres).

Comme LCF, il permet de construire des preuves de manière "top-down" en décomposant des buts en buts plus simples. Il gère les bibliothèques de définitions, de théorèmes et de programmes constituant les théories mathématiques développées. ML sert de langage de programmation et de méta-langage à sa logique objet.

La logique de NUPRL est un λ-calcul typé. Selon la correspondance de Curry-Howard, les propositions sont représentées par des types. Les types sont générés par des types primitifs, des variables de types et des constructeurs de types.

Nous avons choisi de présenter NUPRL comme un démonstrateur interactif pour une logique fixe, car il n'était initialement destiné qu'à la formalisation des mathématiques constructives [HHP89]. Toutefois, **la logique de NUPRL peut servir de cadre logique**, à l'instar de LF ou CC.

Types primitifs:

`void`	Type vide.
`int`	Entiers relatifs.
`atom`	Chaînes de caractères.

Constructeurs de types (T_1 et T_2 sont des types):

T_1 `list`	Liste.
$T_1 \rightarrow T_2$	Espace fonctionnel.
$x{:}T_1 \rightarrow T_2$	Espace fonctionnel dépendant.
T_1 # T_2	Produit cartésien.
$x{:}T_1$ # T_2	Produit cartésien dépendant[11].
T_1 | T_2	Union disjointe.
$\{x{:}T_1 \mid T_2\}$	Ensemble, T_2 représente une proposition sur x.
$(x,y){:}T_1$ // T_2	Quotient, T_2 représente une relation d'équivalence sur T_1.

[11] Similaire au $\Pi x{:}\ldots$ de LF et CC.

Un ensemble de constantes et de fonctions pré-définies servent à construire et à manipuler les termes de différents types. Notamment, NUPRL propose deux fonctions pour l'induction sur les entiers et les listes.

Contrairement à la théorie des ensembles, où l'égalité est une notion universelle, **l'égalité**, en NUPRL, est toujours définie relativement à un type. Ainsi, le fait que les termes `a` et `b` soient de type `T` et égaux dans ce type se note `a = b in T`. En plus de l'égalité, cette relation exprime l'appartenance à un **type**: le fait que le terme `a` soit de type `T` peut, en effet, s'écrire `a = a in T`, que l'on abrège `a in T`.

Une notion est dite *prédicative* si elle est introduite par quantification sur un type qui ne la contient pas. Pour obtenir une structure de types prédicative, on considère une hiérarchie infinie d'*univers* (notés U_1, U_2...), fermés par les constructeurs de types. U_1 sert de type aux types primitifs. U_{i+1} ($i \geq 0$) sert de type à U_i et à tous les termes de type U_i.

Un mécanisme de définition permet d'étendre la syntaxe de ce langage à l'aide de formats. Par exemple, si on écrit la fonction ML `abs`, qui calcule la valeur absolue d'un entier, `|<x:int>| == abs(<x>)` définit la notation `|x|` pour la valeur absolue de `x`.

Le fait de représenter les propositions par des types permet une **économie** substantielle du **langage** (T_1 et T_2 sont des types):

T_1 # T_2	Dénote la **conjonction** de T_1 et T_2.
T_1 | T_2	Dénote la **disjonction** de T_1 et T_2.
x:T_1 # T_2	Dénote l'**existence** d'un terme x de type T_1 vérifiant T_2.
x:T_1 → T_2	Dénote que **tout** terme x de type T_1 vérifie T_2.

Par exemple, l'ensemble des entiers naturels peut s'écrire `{x:int|x≥0}` ou `x:int#x≥0`.

La logique de NUPRL sert à construire des **jugements** de la forme ($n \geq 0$, T_1..., T_n, S, s sont des termes):

$$x_1:T_1\ldots,x_n:T_n \vdash S \text{ [ext s]}$$

Un jugement est vrai si le terme `s` admet `S` comme type pour toute substitution correcte des variables $x_1\ldots,x_n$.

Par exemple, le jugement suivant est vrai:

$$\vdash 0 < 1 \text{ [ext axiom]}$$

La notion de jugement sert de relation de **satisfaction** au sens de LCF [GMW79]. Chaque **but** de NUPRL exprime le fait qu'un type soit habité, sous certaines hypothèses. On le représente par un **séquent** de la forme:

$x_1:T_1\ldots,x_n:T_n \vdash S$

Dans LCF, un événement était un terme de type théorème. Dans NUPRL, un **événement** est un terme s qui représente une **preuve**: s satisfait un but $x_1:T_1\ldots,x_n:T_n \vdash S$ si et seulement si le jugement $x_1:T_1\ldots,x_n:T_n \vdash S$ [ext s] est vrai. Quand la décomposition n'est pas complète, la composition des fonctions de validation produit un terme qui représente une **preuve partielle**.

Les tactiques permettent d'automatiser la décomposition de buts dans NUPRL. NUPRL propose un large éventail de tactiques, écrites en ML, extensible par l'utilisateur.

Contrairement à LCF, la construction des preuves constituant les événements **assure la correction** des tactiques utilisées. En effet, ML empêche la construction de preuves incorrectes. Il faut néanmoins **vérifier la relation de satisfaction** après chaque appel de tactique.

Les tactiques correspondant à l'application de règles d'inférence à l'envers ne sont pas écrites en ML. La fonction ML *refine*, prédéfinie par NUPRL, **transforme automatiquement une règle d'inférence en tactique**. Les tactiques produites par refine **vérifient elles-même** la relation de satisfaction.

Les tactiques considérées dans NUPRL sont de deux sortes, les tactiques de raffinement et les tactiques de transformation.

Une *tactique de raffinement* s'applique à un but non décomposé. A l'écran, le nom de la tactique apparaît comme règle de décomposition du but. Mais la validation produite correspond à une règle dérivée pouvant regrouper plusieurs pas d'inférence.

Une *tactique de transformation* peut s'appliquer à n'importe quel but. Elle peut analyser ou transformer la décomposition de ce but (mémorisation, comparaison, optimisation, duplication, décomposition analogue, etc.). Les applications de tactiques produites apparaissent à l'écran. Notons qu'à toute tactique de raffinement correspond une tactique de transformation.

Les types ML suivants sont comparables à ceux définis pour LCF, mais ils permettent d'appliquer des tactiques à des buts non décomposés:

```
tactic = proof → (proof list # validation)
validation = proof list → proof
```

NUPRL propose les opérateurs de tactiques de LCF et les deux nouveaux opérateurs de tactiques:

- COMPLETE tac_1 applique tac_1 sur tous les buts générés, si tac_1 permet de tous les décomposer, il échoue sinon.

- PROGRESS tac_1 applique tac_1 si les buts générés ne contiennent pas le but initial, il échoue sinon. Cet opérateur est utile pour éviter les boucles infinies, par exemple, lors de l'utilisation de REPEAT.

Comme LCF, NUPRL propose des notions de tactiques et d'opérateurs de tactiques similaires pour les règles de réécriture.

Édition interactive de preuves tolérant et mettant les erreurs en évidence [RA83]

Un éditeur de preuves imposant la correction de chaque pas d'inférence n'est pas toujours pratique.

C'est particulièrement vrai pour l'édition de programmes considérés comme preuves de leurs spécifications. Assurer systématiquement la compatibilité des post-conditions et des pré-conditions attachées aux blocs d'instructions assemblés est trop contraignant.

Un éditeur de programmes pratique pourrait laisser l'utilisateur écrire un programme **sous la forme textuelle** usuelle. Il se contenterait de mettre les instructions présumées **responsables** du non respect des spécifications **en évidence**. L'utilisateur pourrait alors choisir entre corriger une erreur particulière et écrire le reste du programme.

En effet, **le fait qu'un programme ne respecte pas ses spécifications n'est pas une propriété locale**. La correction d'une erreur peut en produire ou en corriger d'autres.

[RA83] présente un éditeur de programmes **structuré**, qui tolère et met les erreurs en évidence. Cet éditeur a été réalisé avec CSG [Gri87]. Étant donnée la description d'une classe de termes, sous forme d'une grammaire à attributs, CSG génère automatiquement un éditeur de termes pour cette classe.

Une *grammaire à attributs* est obtenue en attachant des attributs aux symboles d'une grammaire hors contexte. Un ensemble d'équations dites *sémantiques* est associé à chaque règle de production. Chacune de ces équations définit la valeur d'un attribut attaché à un symbole, en fonction des valeurs des autres attributs. Si cet attribut est attaché au symbole non terminal de gauche, il est dit *synthétisé*. S'il est attaché à un symbole de droite, il est dit *hérité*. Dans une grammaire à attributs, les règles de production modélisent la **structure** des termes et les attributs modélisent les **dépendances non locales** entre leurs différents sous-termes.

A chaque sous-terme d'un terme, on peut associer un ensemble de valeurs. Ces valeurs correspondent aux attributs de la règle de production utilisée pour construire le sous-terme. Un terme est dit

correctement attribué lorsque les valeurs associées à tous ses sous-termes vérifient les équations sémantiques.

Quand on modifie un terme avec un éditeur produit par CSG, ces valeurs sont mises à jour de manière **automatique**, **incrémentale** et **optimale en temps**, en respectant les équations sémantiques.

Les grammaires à attributs permettent de spécifier des calculs de preuves. Les **règles d'inférence** (et les axiomes) sont représentées par des **règles de production** munies d'équations sémantiques. Les **valeurs des attributs** considérés sont des **formules**. Les preuves correspondent aux termes correctement attribués.

Pour admettre les preuves incomplètes, on peut ajouter à chaque non-terminal `X`, une règle de la forme `X ::= ⊥`, munie d'équations sémantiques adéquates. Un attribut supplémentaire peut servir à indiquer si une preuve est complète.

Vérifier qu'une preuve démontre un théorème consiste à vérifier une propriété des attributs attachés à la racine du terme édité.

Dans le cadre de l'éditeur de programmes, les formules considérées sont celles d'une logique du premier ordre. Une formule sert à caractériser l'environnement de variables entre deux instructions du programme. Elles forment les valeurs de deux attributs *post* et *pre*, respectivement hérité et synthétisé.

```
<Stmt> ::= if <Cond> Then <StmtList> else <StmtList> fi
    StmtList1.post = Stmt.post
    StmtList2.post = Stmt.post
    Stmt.pre = (Cond ⇒ StmtList1) ∧ (¬Cond ⇒ StmtList2)
```

Initialement, l'utilisateur introduit les deux formules φ_{pre} et φ_{post}, représentant les spécifications du programme à éditer. Quand la valeur de l'attribut `post` du programme final est φ_{post}, φ_{pre} doit vérifier la valeur de l'attribut `pre`.

L'éditeur donne donc la valeur φ_{post} à l'attribut `post` du programme édité, dès la création du programme ⊥ initial. Un *test* est une formule attachée à une règle de production. A chaque modification, un test, attaché à la règle de production principale de la grammaire, indique si φ_{pre} satisfait la valeur synthétisée de `pre`. Quand c'est le cas, le programme respecte ses spécifications.

```
Program ::= {Assertion} <StmtList> {Assertion}
    StmtList.post = Assertion2
    check:IsTheorem(Assertion1 ⇒ StmtList.pre)
```

Pendant l'écriture d'un programme, l'utilisateur est conduit à introduire d'autres **contraintes** (invariants de boucles, spécifications de sous-programmes, etc.). Un test similaire, construit à partir des

attributs et d'expressions locales au programme, exprime chaque contrainte. Il permet une localisation plus précise des erreurs.

A chaque modification, l'éditeur recherche le premier terme (parcours "bottom-up"), dont la règle de production contient un test faux. Le bloc d'instruction correspondant à ce terme est **mis en évidence**.

```
<Stmt> ::= while <Cond> Invariant <Assertion>
                do <StmtList> od
    StmtList.post = Assertion
    Stmt.pre = Assertion
    check:IsTheorem((Assertion ∧ ¬Cond) ⇒ Stmt.post)
    check:IsTheorem((Assertion ∧ Cond) ⇒ StmtList.pre)
```

Pour prouver les tests, l'éditeur de programmes appelle un démonstrateur automatique. En cas d'échec, la vérification du test est laissée à l'utilisateur. Pour construire la preuve, **un éditeur similaire à l'éditeur de programmes** est proposé. Le calcul choisi pour ces contraintes est un système de Gentzen pur.

Les règles d'inférence sont les règles d'introduction et d'élimination des connectifs logiques et des règles traitant l'égalité et l'inégalité. Elles sont représentées par des règles de production.

Un séquent est représenté par deux attributs hérités `ante` et `succ`, reliés par des équations sémantiques.

Les contraintes de cette deuxième grammaire à attributs consistent à vérifier si des formules données apparaissent dans les valeurs des attributs `ante` ou `succ` hérités. Une simple **procédure de décision** permet de **calculer les tests** correspondants.

Notons aussi que des **tactiques** pourraient aider l'utilisateur dans la démonstration des théorèmes du premier ordre. Dans le cadre des grammaires à attributs, ces tactiques pourraient être réalisées à l'aide d'attributs capables de modifier un terme. Il est envisagé d'étendre CSG au traitement de tels attributs.

IPE [Bur86]

IPE (Interactive Proof Editor) est un éditeur de preuves "top-down" pour la logique propositionnelle et la logique des prédicats du premier ordre. Sa **simplicité d'utilisation** est remarquable.

L'utilisation de grammaires à attributs pour maintenir la correction de preuves étant mise en évidence par [RA83], un évaluateur d'attributs incrémental a été écrit en ML. IPE à été réalisé avec cet évaluateur.

Les preuves sont représentées par des termes, un attribut **hérité** contient le **but courant**, un attribut **synthétisé** indique si la **preuve** de ce but est **complète**.

Initialement, IPE demande la formule à démontrer à l'utilisateur. Le but correspondant à cette formule est affiché. Le calcul utilisé par IPE est un système de déduction naturelle. Chaque but représente un séquent affiché sous la forme: Show <*premise*$_1$>, ... <*premise*$_n$> entails <*conclusion*>, s'il a n prémisses ou Show <*conclusion*>, s'il n'a pas de prémisse.

Chaque prémisse (conclusion) d'un séquent **détermine** la règle d'élimination (d'introduction) correspondant à son connectif logique racine. Lorsque l'utilisateur clique sur une prémisse ou une conclusion, cette règle est appliquée (à l'envers) et les buts obtenus sont affichés.

Quand la preuve est complète, IPE indique que la formule initiale est un théorème.

Pendant la construction d'une preuve, l'utilisateur peut faire un mauvais choix, en cliquant sur une formule ne conduisant pas à une preuve complète. Il peut alors **revenir en arrière** en cliquant sur une autre formule du même séquent. La partie de preuve sous ce séquent est effacée et la nouvelle règle est appliquée.

Pour la logique propositionnelle, **la seule action de cliquer suffit pour construire des preuves avec la possibilité de revenir sur tous les choix effectués.**

Pour la logique des prédicats du premier ordre, les actions de construction ne sont guère plus complexes. Pendant l'application d'une règle correspondant à un quantificateur, l'utilisateur est invité à introduire une nouvelle constante ou à spécifier un terme.

Néanmoins, trouver les bons choix pour compléter une preuve est plus délicat, l'ordre de décomposition des quantificateurs étant particulièrement important.

IPE permet de mémoriser les théorèmes démontrés. Ceux-ci peuvent être réutilisés pour construire d'autres preuves. Les théorèmes applicables à un séquent sont obtenus par **filtrage** sur les théorèmes mémorisés.

IPE est délibérément resté un système naïf. Il ne propose pas de recherche automatique de preuves. Sa logique est simple et ne contient pas l'égalité.

Mais en contrepartie, son **interface** est particulièrement **attrayante**.

b) Paramétrables par le système formel utilisé (méthode cadre logique)

EFS [Gri87]

EFS (Environment for Formal Systems) est un démonstrateur interactif **générique**, c'est à dire paramétrable par le système formel

utilisé. Ce système formel objet est défini dans l'un des deux cadres logiques proposés. Le langage de ces cadres logiques est un λ-calcul Π-typé. Leurs calculs de types sont **CC** [CH88] et **LF** [HHP89].

Quand on emploie CC, les constantes du λ-calcul définissent les connectifs logiques de manière constructive. Quand on emploie LF, elles définissent le langage et les règles d'inférence du système formel. **Le λ-calcul traite une fois pour toutes l'analyse syntaxique, le calcul des variables libres, les substitutions et les α-conversions.**

Étant donnée la définition d'un système formel objet, EFS, qui a été réalisé avec CSG [Gri87], se comporte un **éditeur structuré** pour ce système formel. Ainsi, il permet de construire les λ-termes, de leur associer des noms ou des présentations textuelles, d'écrire des règles de raffinement, de regrouper ces informations en théories et de hiérarchiser celles-ci. La structure de **fichier EFS** regroupe l'ensemble de ces notions de manière homogène.

Par exemple, la présentation textuelle de chaque constante du λ-calcul est définie par un format de la forme:

```
<a> => <b> == (implies a b)
```

Contrairement à NUPRL [C+86], **EFS est capable de traiter les variables sans développer les présentations.**

On obtient la forme $\beta\eta\delta$-normale d'un λ-terme en commençant par développer les présentations et les nommages qu'il contient. Un autre algorithme de mise sous forme normale permet de **conserver certaines présentations**, choisies par l'utilisateur.

Comme l'éditeur de programmes de [RA83], EFS tolère les preuves **incorrectes** et met les erreurs **en évidence**, grâce aux grammaires à attributs.

Une preuve peut être construite de manière "bottom-up", par assemblage successif des constantes définissant le système formel objet. Toutefois, des **règles de raffinement**, au sens de NUPRL, permettent une construction "top-down", souvent plus pratique.

Des règles de raffinement standard sont proposées. L'utilisateur peut définir d'autres règles de raffinement. Leur validité, qui dépend du système formel objet, est vérifiée dès leur introduction.

Une règle de raffinement est applicable quand sa conclusion **filtre** le but courant. Pour produire les nouveaux buts et la fonction de validation, il suffit d'appliquer la substitution obtenue, aux hypothèses et à la fonction de validation de la règle de raffinement.

Ajouter un méta-langage comme ML rendrait EFS plus souple. En particulier, ML permettrait de produire automatiquement les règles de

raffinements correspondant à l'application de règles d'inférence à l'envers, comme dans NUPRL.

ISABELLE [Pau88]

ISABELLE est un démonstrateur interactif **générique**, écrit en ML, qui a permis d'implémenter la logique du premier ordre, la logique d'ordre supérieur, la théorie des ensembles et la théorie constructive des types.

Au lieu de représenter les règles d'inférence par des **fonctions**, comme dans LCF, elles sont représentées par des **relations**. Cette représentation facilite la construction de preuves "top-down" et rend les constructions de preuves "top-down" et "bottom-up" homogènes.

Pour appliquer une règle d'inférence de manière "top-down" ("bottom-up"), il suffit **d'unifier** sa conclusion (l'une de ses prémisses) avec un but non décomposé (la conclusion) de la preuve éditée.

Ainsi, **l'unification d'ordre supérieur permet de traiter de manière purement syntaxique les applications de règles d'inférence** et plus généralement la construction de preuves d'une certaine classe de logiques.

Bien que le problème de l'unification d'ordre supérieur soit indécidable, les cas rencontrés en pratique semblent simples, et l'algorithme de Huet fonctionne bien.

Le langage de ISABELLE est un λ-calcul typé avec **deux jugements d'ordre supérieur**, *hypothétique* et *schématique* (notés $\Rightarrow$, et Λ) identiques à ceux considérés dans LF [HHP89], et une facilité de définition (notée $\equiv$). Ces jugements d'ordre supérieur permettent **d'exprimer** les règles d'inférence des systèmes de **déduction naturelle**. Pour isoler la méta-logique ($\Rightarrow$, Λ et $\equiv$) de la logique objet, les formules de celle-ci sont notées entre crochets ([,]).

Par exemple, la règle de généralisation:

$$\frac{A}{\forall x\, A}$$

Peut s'écrire:

$$\Lambda F.\left(\Lambda x.[F(x)]\right) \Rightarrow [\forall x\, F(x)]$$

La règle pure de déduction naturelle:

$$\frac{\begin{array}{c}[A]\\ B\end{array}}{A \supset B}$$

Peut s'écrire:

$$\Lambda AB.([A] \Rightarrow [B]) \Rightarrow [A \supset B]$$

ISABELLE propose des tactiques et des opérateurs de tactiques [GMW79]. Comme dans NUPRL, une tactique standard permet d'appliquer n'importe quelle règle d'inférence de manière "bottom-up". Mais l'écriture des règles d'inférence et de cette tactique standard sont beaucoup plus simples.

λ-PROLOG [FM87]

Les opérations de **recherche** et **d'unification** sont essentielles pour réaliser des démonstrateurs. Une caractéristique des **langages de programmation logique** étant de traiter ces opérations, il est naturel de les employer pour réaliser des démonstrateurs interactifs.

Un langage de programmation logique classique tel que PROLOG ne suffit pas à **décrire les logiques de manière naturelle**. En particulier, les termes du premier ordre ne permettent pas une expression naturelle des problèmes de quantification.

Un langage d'ordre supérieur comme λ-PROLOG est préférable.

λ-PROLOG est fondé sur un λ-calcul simplement typé. Le type o des propositions, les connectifs logiques $\wedge$, $\vee$ et $\Rightarrow$ de type $o \rightarrow o \rightarrow o$ et les constantes Π et Σ de type $(\alpha \rightarrow o) \rightarrow o$ sont **prédéfinis** (α est une variable de type). $\forall x\, A$ et $\exists x\, A$ dénotent respectivement $\Pi\lambda x.A$ et $\Sigma\lambda x.A$.

On considère trois classes de propositions. Les *formules atomiques* (notées A) sont les propositions primitives. Les *formules buts* (notées G) et les *clauses* (notées D) sont définies par la grammaire suivante:

$$G ::= A \mid G \wedge G \mid G \vee G \mid D \Rightarrow G \mid \forall x\, G \mid \exists x\, G$$
$$D ::= A \mid G \Rightarrow A \mid \forall x\, D$$

Un *programme* (noté P) est un ensemble fini de clauses.

Six opérations servent à rechercher si une formule but G donnée est déductible d'un programme P donné.

- Quand $G = G_1 \wedge G_2$, AND tente de déduire G_1 de P et G_2 de P.
- Quand $G = G_1 \vee G_2$, OR tente de déduire G_1 de P ou G_2 de P.
- Quand $G = D \Rightarrow G_1$, AUGMENT tente de déduire G_1 de $P \cup \{D\}$.

- Quand $G = \forall x\, G_1$, `GENERIC` ajoute une nouvelle constante c et tente de déduire $[x/c]G_1$ de P.
- Quand $G = \exists x\, G_1$, `INSTANCE` recherche un λ-terme t et de déduire $[x/t]G_1$ de P.
- Quand $G = A$, `BACKCHAIN` recherche si une instance d'une clause de P est de la forme G ou $G_1 \Rightarrow G$. Dans ces cas respectifs, G est déductible de P, et `BACKCHAIN` tente de déduire G_1 de P, sinon G n'est pas déductible de P.

Bien que le calcul formé par ces six opérations soit **correct** et **complet**, la stratégie de λ-PROLOG n'est pas **complète**.

L'égalité définitionnelle (βη-conversion) de λ-PROLOG et les problèmes d'unification d'ordre supérieur rencontrés sont décidables.

A l'instar de LF [HHP89], un ensemble de constantes typées spécifie le langage d'un système formel:

```
not:                form → form
and,or,implies:     form → form → form
exists,forall:      (i → form ) → form
entails:            (list form) → form → sequent
```

Une règle d'inférence peut être représentée par une clause, à l'aide d'un prédicat *proof* qui joue le rôle d'une forme de jugement de base au sens de LF. `proof` construit aussi la preuve formelle correspondant aux règles d'inférence utilisées.

Par exemple, le modus-ponens est représenté par:

```
(proof A p1) ∧ (proof (implies A B) p2)
   ⇒ (proof B (modus-ponens p1 p2))
```

L'introduction de l'implication en déduction naturelle est représentée par:

```
∀pA ((proof A pA) ⇒(proof B (pB pA)))
   ⇒ (proof (implies A B) (imp_i pB))
```

La règle de généralisation est représentée par:

```
∀y (proof (A y) (p y))
   ⇒ (proof (forall A) (forall_i p))
```

Dans ce contexte, on peut directement utiliser λ-PROLOG comme un **vérificateur de preuves** ou un **démonstrateur automatique**. La formule but `(proof A p)` où `A` est une formule de la logique objet et `p` est une preuve permet de **vérifier** si `p` est une preuve de `A`. La formule but

`(proof A x)` où `A` est une formule de la logique objet et `x` est une variable permet de **démontrer** `A`, `x` sera instanciée par la preuve éventuellement trouvée.

Les tactiques et les opérateurs de tactiques, au sens de LCF, permettent de **contrôler** la recherche de preuve [GMW79]. λ-PROLOG est alors utilisé comme un **démonstrateur interactif**.

Les buts considérés correspondent aux **formules buts**, pas aux formules atomiques. La structure de but pouvant simuler celle de liste, on maintient **un but courant unique**, pas une liste de buts restant à développer, comme dans LCF.

Ainsi, un ensemble de constantes typées définit le langage des expressions buts:

```
truegoal:              goalexp
andgoal,orgoal:        goalexp →goalexp →goalexp
existsgoal,allgoal:    (α →goalexp) →goalexp
impgoal:               α →goalexp →goalexp
```

Une **tactique** est une relation entre deux buts. Le prédicat *maptac* applique récursivement une tactique à un but composé, et construit en retour le but composé résultant. Intuitivement, l'effet de `maptac` est de "compiler" l'application de tactique à un but composé en clauses interprétées par λ-PROLOG. `maptac` permet une écriture simple et rapide des **opérateurs de tactiques**:

```
(T1 g1 g2) ∧ (maptac T2 g2 g3) ⇒ (then T1 T2 g1 g3)
(T1 g1 g2) ∨ (T2 g1 g2) ⇒ (orelse T1 T2 g1 g2)
(idtac g g)
(orelse (then T (repeat T)) idtac g1 g2)
   ⇒ (repeat T g1 g2)
(T g1 g2) ∧ (goalreduce g2 truegoal) ⇒ (complete T g1 g2)
```

Ces opérateurs sont plus généraux que leurs versions LCF, écrites en ML. Par exemple, dans la première expression, si T_2 échoue, l'opérateur `then` tente de prouver $(T_1\ g_1\ g_2)$ d'une autre façon. Toutefois, le prédicat de coupure de λ-PROLOG permet de simuler les versions LCF, en "oubliant" les chemins laissés de côté.

Définir une règle d'inférence de manière **déclarative**, au lieu de **procédurale**, permet de l'employer comme **tactique élémentaire**.

Par exemple, les trois règles d'inférence précédentes sont respectivement redéfinies par:

```
(modus-ponens_tac
   (proofgoal B (modus-ponens p1 p2))
   (andgoal (proofgoal A p1) (proofgoal (implies A B)))

(imp_i_tac
```

```
(proofgoal (implies A B) (imp_i p_B))
(allgoal p_A (impgoal
      (hyp A p_A)
      (proofgoal B (p_B p_A))))

(forall_i_tac
   (proofgoal (forall A) (forall_i p))
   (allgoal y (proofgoal (A y) (p y))))
```

Notons que le terme `(hyp A p_A)` de `imp_i_tac` est ajouté au programme par l'opération `AUGMENT` qui traite `impgoal`.

Il est possible d'écrire d'autres tactiques élémentaires, comme une tactique de simplification de but ou une tactique utilisateur.

Notons que le formalisme de ISABELLE est un sous-formalisme de λ-PROLOG. λ-PROLOG pourrait donc directement simuler ISABELLE. Mais il ne serait probablement pas aussi efficace.

Un algorithme capable de traduire une signature LF en λ-PROLOG a été réalisé, ses propriétés formelles restent à établir.

Coq [D+91]

Coq est un démonstrateur interactif fondé sur *CCind* (Calcul des Constructions inductives). Ce formalisme permet notamment de définir les connectifs logiques en spécifiant leurs règles d'introduction. Leurs règles d'élimination sont alors des théorèmes qu'il faut démontrer.

Coq sert aussi de langage de programmation pour écrire des "programmes certifiés" et permet d'en extraire le contenu algorithmique.

Le formalisme CCind correspond à CC [CH88] étendu par la possibilité de définir des types inductifs, par exemple, comme les types des entiers naturels, des listes, des arbres, ou plus particulièrement des preuves. Dans Coq, on définit un type inductif en donnant la signature de ses constructeurs. Son environnement est alors étendu par ce type inductif, des constantes représentant ses constructeurs, une opération de déstructuration, fondée sur la notion de filtrage, et un principe d'induction.

Au lieu d'avoir une seule sorte `*`, comme dans CC, on emploie trois sortes `Set`, `Prop` et `Type`. `Set` est la sorte des spécifications, en particulier des types de données, qui typent les programmes. `Prop` est la sorte des propositions, qui typent les preuves. `Type` est le type de `Prop`. Les sortes `Set` et `Prop` permettent d'éviter les confusions entre les termes qui représentent les objets logiques et ceux qui représentent les objets mathématiques axiomatisés.

Le schéma général de définition d'un type inductif de nom `X`, de type `A` et de constructeurs $c_1, \ldots, c_p$ est:

```
Inductive Definition X:A =
          c1:C1
    |     c2:C2
          ...
    |     cp:Cp.
```

Par exemple, le type des entiers naturels construits avec `O` et `S` peut être défini par[12]:

```
Inductive Set nat = O:nat | S:nat->nat.
```

Un type inductif peut dépendre de paramètres. Par exemple, le type des listes polymorphes peut être défini par:

```
Inductive Set list [A:Set] =
          nil:(list A)
    |     cons:A->(list A)->(list A].
```

Les définitions inductives ne sont pas nécessairement récursives, l'intérêt dans ce cas étant l'opération de déstructuration et le principe d'induction. Par exemple, le produit et la somme disjointe peuvent être définis par:

```
Inductive Set prod [A,B:Set] = pair:A->B->(A*B).
Inductive Set sum [A,B:Set] =
          inl:A->(A+B)
    |     inr:B->(A+B).
```

On peut aussi définir des relations de manière inductives. Par exemple, l'ordre sur les entiers naturels peut être défini par:

```
Inductive Definition LE:nat->nat->Prop =
          LE_O:(n:nat)(LE O n)
    |     LE_SS:(n,m:nat)(LE n m)->(LE (S n) (S m)).
```

L'opération de déstructuration sert à définir les fonctions primitives récursives sur le type inductif. Ces fonctions sont définies par cas, c'est à dire en donnant leurs valeurs pour les termes de la forme $(c_i\ a_1 \ldots a_p)$ pour chaque constructeur c_i d'arité p, du type inductif.

Ainsi, si `p` est un terme dont le type est inductif et `P` est un type alors l'opération de déstructuration `<P> match p with` est ajoutée à l'environnement.

Par exemple, pour définir une fonction `f` de `A+B` dans `P`, on peut donner ses valeurs t_1 et t_2 pour les termes des formes `(inl A B a)` et `(inr A B b)` où `a` et `b` sont variables. Si `p` est de type `A+B`, le terme `<P> match p with` est de type `(A->P)->(B->P)->P`. Ce qui permet de définir `f` comme `([p:A+B]<P> match p with [a:A]`t_1 `[b:B]`t_2`)`.

---

[12]On note quelques différences de syntaxe par rapport au schéma général.

Le principe d'induction sert au raisonnement par induction sur les éléments du type inductif. Il ne change pas à lui tout seul la classe des théorèmes, mais comme on ne peut pas le démontrer, on doit l'ajouter explicitement à l'environnement. Par exemple, le principe d'induction sur les entiers naturels est représenté par une constante de type
`(P:nat->Prop)(P O)->(n:nat)((P n)->(P (S n)))->(n:nat)(P n).`

En *logique minimale*, on utilise un seul connectif logique, l'implication, et un seul quantificateur, le quantificateur universel. En représentant une proposition par le type de ses preuves, l'*isomorphisme de Curry-Howard*, établit une correspondance entre les propositions et les types, les preuves et les termes. Ainsi, P et Q étant deux propositions, l'espace fonctionnel $P \rightarrow Q$ représente l'implication $P \Rightarrow Q$.

Les *sémantiques de Heyting* suggèrent de représenter une preuve de $\forall x{:}T.P$ par une fonction qui associe une preuve $P[x/t]$ à tout terme t de type T. Les types dépendants, le polymorphisme et les constructeurs de types de CC permettent d'écrire et de typer de tels termes.

En déduction naturelle, les connectifs logiques sont représentés par leurs règles d'introduction et d'élimination. Les règles d'introduction sont analogues aux constructeurs d'un type inductif. Les règles d'élimination sont analogues à l'opération de déstructuration. Représenter un connectif logique par un type inductif revient à noter que la forme de ses règles d'élimination peut être déduits de la spécification de ses règles d'introduction.

La définition et la signification de l'opération de déstructuration (dont on peut déduire les règles d'élimination) de quelques connectifs logiques sont données dans le tableau suivant:

Connectif logique	**Définition**	**Signification de l'opération de déstructuration**
faux	`Inductive Proposition False =.`	De `False`, on peut déduire toute proposition `C`
vrai	`Inductive Proposition True = I:True.`	Pour déduire `C` de `True`, il suffit de prouver `C`
conjonction	`Inductive Proposition and [A,B:Prop] = conj:A->B->(and A B).`	Pour déduire `C` de `(and A B)`, il suffit de prouver `A->B->C`
disjonction	`Inductive Proposition or [A,B:Prop] = or_introl:A->(or A B) | or_intror:B->(or A B).`	Pour déduire `C` de `(or A B)`, il suffit de prouver `A->C` et `B->C`
quantificateur existentiel	`Inductive Proposition ex [A:Set,P:A->Prop] = ex_intro: (x:A)(P x)->(ex A P).`	Pour déduire `C` de `(ex A P)`, il suffit de prouver `(x:A)(P x)->C`

De même, l'égalité de Leibniz est exprimée comme la plus faible relation qui relie tout terme à lui même:

```
Inductive Proposition eq [A:Set,x:A] =
   refl_equal:(eq A x x).
```

Son opération de déstructuration signifie que pour déduire `(P u)` de `(eq A t u)`, il suffit de prouver `(P t)`.

Comme les exemples précédents peuvent le suggérer, le langage manipulé par un utilisateur de Coq n'est pas directement celui de CCind, mais un langage plus naturel appelé *Mathematical Vernacular*. Ce langage propose un mécanisme de sections, pour structurer les définitions et maîtriser la portée des déclarations. Il permet d'introduire des abréviations ou de définir la syntaxe des opérateurs introduits, afin de simplifier la présentation textuelle les termes manipulés.

Le démonstrateur interactif est fondé sur la notion de tactiques, avec les constructeurs de tactiques classiques. Plusieurs groupe de tactiques primitives sont prédéfinies. Les *tactiques d'introduction* permettent de décharger les hypothèses et les variables du but courant vers le contexte local. La *tactique Exact* permet d'introduire directement un terme représentant une preuve du but courant. Les *tactiques de résolution* servent à appliquer les théorèmes déjà démontrés ou les axiomes. Les *tactiques d'élimination* servent à réduire les constantes inductives en leurs constructeurs. Elles correspondent aux raisonnements par cas et par induction. Les *tactiques de conversion* servent à transformer le but courant, les hypothèses ou les paramètres locaux, modulo les $\beta\delta$-conversions et l'élimination des constantes inductives. Les *tactiques automatiques* cherchent à résoudre le but courant de manière automatique.

Coq propose aussi une interface X-WINDOW fondée sur plusieurs fenêtres, avec un mécanisme de couper-coller textuel. Une fenêtre principale permet à l'utilisateur d'entrer les commandes, et en affiche les résultats. Elle est surmontée de plusieurs menus qui permettent d'accéder directement à ces commandes. Une autre fenêtre affiche l'environnement courant de Coq, ses menus permettent d'examiner son contenu ou de le réinitialiser. Deux autres fenêtres apparaissent quand le démonstrateur interactif est activé. Une de ces fenêtre présente le but courant et ses hypothèses locales. Ses menus correspondent aux différents groupes de tactiques primitives. Un bouton permet aussi d'entrer les tactiques composées. L'autre fenêtre présente la liste des sous-buts, elle permet de changer le but courant, qui est mis en évidence, par simple clic.

Une application particulièrement intéressante de Coq est de développer des programmes certifiés, c'est à dire qui respectent leurs

spécifications. Au lieu d'avoir des preuves qui considèrent des programmes, on incorpore les preuves aux programmes. Comme nous l'avons vu, les sortes `Prop` et `Set` permettent de distinguer les propositions et les spécifications, les preuves et les programmes, ces objets étant mutuellement imbriqués. Coq propose ainsi un mécanisme capable d'extraire le contenu algorithmique des programmes certifiés dans un dialecte ML nommé FML. Notons que ce contenu algorithmique est typable dans Fω, un λ-calcul qui correspond à CC, sans les types dépendants.

Coq propose finalement d'optimiser le code obtenu et de le traduire vers les dialectes ML de différents compilateurs (CAML, LML, GAML). Notons que la vérification de type de ces compilateurs doit parfois être désactivée, car elle n'est pas toujours assez puissante pour typer le code, pourtant correct, généré par Coq. Cette traduction permet de bénéficier de toute la technologie des compilateurs existants, et de comparer le code généré à un code écrit manuellement. Elle pourrait aussi suggérer des améliorations pour ces compilateurs, en particulier au niveau logique.

4. Outils de transformation automatique de preuves

Cette section concerne les outils capables de transformer une preuve, sans modifier ses hypothèses et sa conclusion, mais en changeant ses pas d'inférence ou le système formel employé. Des algorithmes destinés à mettre une preuve, ou une partie de preuve (formule, etc.), sous telle ou telle forme, certaines tactiques de transformation de NUPRL [C+86], constituent de tels outils.

La traduction d'une **formule** à démontrer dans le **langage** accepté par un démonstrateur automatique, et la traduction de la **preuve** produite par un démonstrateur automatique dans un autre **système formel** sont des transformations particulièrement utiles. Par exemple, mettre une formule du premier ordre sous forme clausale permet de la réfuter avec un démonstrateur automatique par résolution. OTTER [McC90] effectue une telle traduction.

Décrivons les outils de transformation de formules et de preuves développés pour ATINF.

Transformation de formules [BCC88], [Boy91], [Boy92]

Un transformateur de formules [BCC88] a été réalisé dans le cadre de ATINF. Son noyau et son interface utilisateur **graphique** sont respectivement écrits en LISP et en C. Le noyau peut être utilisé de manière totalement indépendante de l'interface utilisateur. Ainsi, le **démonstrateur par résolution** de ATINF et l'extension d'OTTER décrite dans [BCP94] l'appellent directement.

L'utilisateur introduit une formule du langage des prédicats du premier ordre à transformer. La syntaxe utilisée est très souple: les connectifs logiques $\wedge$ et $\vee$ sont d'arité variable, les notations préfixées et infixées sont admises.

Une **fenêtre** présente cette formule **graphiquement**. L'une des sous-formules, nommée *sélection*, est mise en évidence. Des **boutons** servent à déplacer la sélection ou à lui appliquer **diverses transformations** (forme clausale, forme disjonctive, etc.).

Il est possible de *bloquer* une sous-formule, les transformations la considèrent alors comme atomique et ne la modifient pas. Les sous-formules bloquées sont indiquées entre crochets.

Par rapport à leur application naïve, les transformations connues sont **améliorées**. Par exemple, la forme normale prénexe choisie pour $\forall x P(x) \wedge \forall y Q(y)$ est $\forall x P(x) \wedge Q(x)$, pas $\forall x, y P(x) \wedge Q(y)$.

Notons aussi la disponibilité de la **réduction de la portée des quantificateurs**, qui sert en particulier à réduire les arités des fonctions de skolem, avant skolémisation.

L'utilisateur peut **choisir** de **conserver des propriétés** formelles de la formule initiale (**modèles**, **satisfaisabilité**, **validité**). Le transformateur de formules assure cette conservation en **désactivant** les boutons correspondants aux transformations interdites. Par exemple, si l'utilisateur choisit de conserver la satisfaisabilité, la skolémisation ne peut être appliquée que sur les formules de polarité positive, la skolémisation duale ne peut être appliquée que sur les formules de polarité négative, et sur les formules de polarité nulle, on ne peut appliquer que des transformations conservant les modèles.

Le transformateur de formules sert notamment à mettre des formules sous **forme clausale**. La réfutation d'une formule par un démonstrateur par résolution dépend **essentiellement** de la forme clausale obtenue. La mise sous *forme clausale* standard correspond à une linéarisation, une skolémisation, une mise sous forme normale prénèxe puis conjonctive. Pour améliorer la linéarisation, les équivalences peuvent être linéarisées différemment, selon leurs polarités. Cette amélioration effectue des simplifications et diminue le nombre de clauses obtenues.

Un *renommage* d'une formule consiste à remplacer certaines de ses sous-formules par des littéraux formés de nouveaux prédicats. Après un renommage, si on met la conjonction d'une formule et les définitions des nouveaux prédicats sous forme clausale, l'ensemble de clauses peut **varier** considérablement. On peut donc chercher un renommage favorisant la réfutation. Bien que l'on ne connaisse pas de relation

directe entre la longueur de la réfutation et le nombre de clauses, celui-ci semble être **un bon critère** pour le choix d'un renommage.

La mise sous forme clausale étant déterminée, le nombre de clauses produites pour une formule se calcule très rapidement et **récursivement** (en fonction des nombres de clauses produites pour ses sous-formules directes et leurs négations).

Ce calcul permet d'écrire un algorithme efficace [Boy91], [Boy92] qui, étant donnée une formule, produit un renommage (nommé R_{opt}), pour lequel le nombre de clauses produites est **inférieur** à ceux de la transformation standard améliorée et du renommage *structure preserving* décrit dans [Boy92][PG86] (optimalité restreinte). Dans le cas des formules linéaires, le nombre de clauses est **optimal** dans l'ensemble des renommages possibles.

Cet algorithme a été intégré au transformateur de formules. Les résultats expérimentaux montrent son efficacité, en particulier pour les formules dont la mise sous forme clausale produit un grand nombre de clauses.

Traduction de preuves [CDH93]

Pour la mécanisation des logiques non-classiques on peut classifier les méthodes existantes en deux grandes catégories: les **méthodes directes** et les **méthodes par traduction**. La première consiste à implémenter des calculs particuliers pour chaque logique. La deuxième traduit les logiques non-classiques (logiques sources) en logique classique du premier ordre (logique cible). Dans cette deuxième approche, qui semble devenir la plus employée, il reste un problème essentiel: la preuve est obtenue dans une logique différente de celle où le problème a été spécifié.

Nous avons proposé dans [CDH93] un cadre pour le transfert (partiel) des preuves obtenues dans la logique cible vers la logique source. Ce transfert donne au moins un schéma de la preuve dans la logique cible. Dans un certain sens, la traduction partielle inverse proposée peut être considérée comme une façon de transférer des stratégies de la logique cible vers la logique source, puisque les pas de preuves en logique du premier ordre gardent la "trace" de la stratégie avec laquelle ils ont été obtenus.

5. Traitement de la langue naturelle

Dans cette section, on considère "la" *langue naturelle*. Bien évidemment, cette langue naturelle représente une langue vivante particulière ou un jargon mathématique particulier (on peut même distinguer les formes écrites et parlées, l'origine ou le statut social de

l'utilisateur, etc. [Dyb82]). L'utilisateur est habitué aux preuves en *langue naturelle*. Ces preuves utilisent les **phénomènes très complexes** de la langue naturelle (analogies, commentaires, métaphores, utilisation de pronoms, etc.) usuels à l'être humain.

Hélas, on ne sait pas encore, et on ne saura peut-être jamais **entièrement formaliser** la langue naturelle et ses phénomènes. En particulier, l'ordinateur ne peut pas interpréter directement les preuves en langue naturelle. Dans une certaine mesure, l'utilisateur peut s'adapter aux preuves formelles. Mais leur manque de clarté est une contrainte qui limite la taille des preuves qu'il peut appréhender directement.

Pour faciliter la communication entre l'utilisateur et la machine, on tente de traduire les preuves formelles en langue naturelle, et réciproquement.

Pour traduire une preuve formelle en langue naturelle, on simule celle-ci par une grammaire formelle dont le langage est appelé *langage naturel*. Ce langage peut être considéré comme un système formel dégénéré. La traduction de preuves formelles en langue naturelle est donc, en quelque sorte, un cas particulier de la traduction de preuves entre systèmes formels.

La traduction de preuves en langue naturelle en langage naturel ou en preuves formelles fait intervenir des mécanismes de reconnaissance plus complexes, qui tentent d'extraire la structure formelle de ces preuves.

a) Traduction de preuves formelles en langage naturel

L'enjeu de la traduction automatique de preuves formelles en langue naturelle est l'acceptation des systèmes de déduction automatiques par les mathématiciens.

En pratique, on génère un langage naturel. Le processus de traduction est composé d'une suite d'étapes de transformation et de simplification terminée par une étape de production de messages. Cette suite d'étapes sert à produire la preuve la plus claire possible.

Notons que les deux articles [Che76] et [Lin88] concernent essentiellement la transformation de preuves. Nous avons choisi de les présenter ici car ils décrivent des étapes du processus de traduction de preuves formelles en langue naturelle.

Linéarisation [Che76]

La forme des preuves formelles des systèmes de déduction naturelle est relativement proche de celle des preuves en langue naturelle correspondantes. Cette proximité provient du mécanisme de traitement des hypothèses de ces systèmes. Choisir un système de déduction

naturelle comme point de départ pour traduire les preuves formelles en langage naturel est donc judicieux.

EXPOUND est un programme pour traduire en langage naturel, des preuves d'un système de déduction naturelle pour la logique des prédicats du premier ordre.

Etant donné un système formel, une preuve est souvent représentée par une suite de lignes. Plusieurs ordres sont possibles pour ces lignes. La *linéarisation* consiste à les **ordonner** pour obtenir la représentation la plus **claire** possible.

Dans un premier temps, EXPOUND linéarise donc la preuve formelle. Pour cela, il reconstruit son graphe. Il applique ensuite des règles de regroupement de nœuds voisins, de manière itérative. Par la suite, un nœud peut donc contenir plusieurs lignes. Ces règles considèrent le nombre de prédécesseurs et de successeurs de chaque nœud et le nombre de lignes qu'ils contiennent.

Quand toutes les règles ont été appliquées, les nœuds restants correspondent aux paragraphes de la preuve en langage naturel à produire. Un tri topologique de ces nœuds termine la linéarisation.

Dans un deuxième temps EXPOUND traduit la preuve linéarisée en langage naturel.

Une table fournie par l'utilisateur permet de traduire les instances de prédicats contenus dans les formule. Elle contient des informations lexicales concernant chaque prédicat employé dans la preuve:

- Des *formes verbales* décrivent ses formes positive ou négative, active ou passive.
- Des *prépositions* peuvent précéder chacun des arguments.
- Un *genre* permet de choisir un pronom approprié, quand c'est nécessaire.
- Un *type syntaxique* indique comment compacter l'introduction d'une variable et le fait qu'elle satisfasse le prédicat[13].

La traduction des connectifs logiques $\wedge$, $\vee$, $\Rightarrow$, est directe. Par contre, EXPOUND tente de compacter les formules quantifiées en employant les types syntaxiques.

Dans les nœuds représentant les paragraphes, il produit une phrase pour chaque ligne, en fonction de la ligne précédente et de la règle d'inférence utilisée. Les lignes "évidentes" sont éliminées.

---

[13]Cette notion est à rapprocher de l'utilisation de variables sortées, qui simplifie certaines formules en éliminant les prédicats correspondant aux sortes.

Les différents paragraphes sont enfin reliés par des phrases d'introduction et présentés de manière indentée, pour former la preuve en langage naturel.

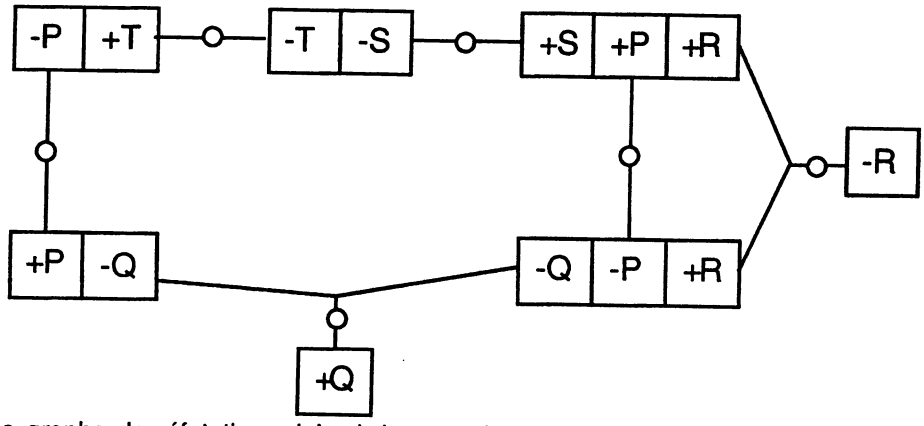
Introduction de lemmes [Lin88]

Les preuves formelles sont souvent obtenues à l'aide de démonstrateurs automatiques. Toutefois, elles ne se présentent pas toujours sous la forme de déductions naturelles. En particulier, un grand nombre de démonstrateurs automatiques produisent des preuves par résolution.

Les preuves par résolution peuvent être **traduites** en déduction naturelle, de manière directe et automatique. Mais pour une preuve par résolution longue ou complexe, cette traduction tend à diminuer sa clarté au lieu de l'améliorer.

Lors du processus de traduction, on peut introduire des **lemmes**, pour éviter les duplications de sous-preuves et structurer la preuve en déduction naturelle obtenue.

Un *graphe de clauses* est un multi-graphe dont les nœuds sont des clauses fermées, et les liens sont des ensembles de littéraux de ces clauses, égaux au signe près. Un *graphe de déduction* est un graphe de clauses sans cycles, en considérant les chemins passant d'une clause à l'autre par deux littéraux de signes contraires et contenus dans le même lien. Un *graphe de réfutation* est un graphe de déduction dont tous les littéraux sont reliés. Un lien est dit *séparant* ssi il est nécessaire à la connexité du graphe considéré.



Exemple de graphe de réfutation minimal. Les carrés contiennent les littéraux, les assemblages de carrés contiennent représentent les clauses. Les ronds matérialisent la séparation entre les littéraux positifs et négatifs des liens.

Une preuve par résolution peut être facilement représentée par un graphe de réfutation minimal. Par exemple, le lien en haut à gauche du graphe précédent représente le pas de résolution qui produit la résolvante $\neg P \vee \neg S$, à partir des clauses $\neg P \vee T$ et $\neg T \vee \neg S$. La traduction

en déduction naturelle consiste à transformer ce graphe pas à pas en utilisant une représentation mixte appelée *preuve de déduction naturelle généralisée*.

A chaque pas de la transformation, on applique un algorithme de restructuration à chaque graphe de réfutation restant. Cet algorithme recherche les *séparants*, et duplique les sous-graphes "triviaux" reliés par ces liens.

Les autres liens séparants définissent une partition du graphe. Un sous-graphe appartenant à cette partition est dit *positif* ssi tous ses littéraux proviennent des axiomes, et *négatif* sinon. Comme un littéral provenant du théorème à démontrer peut être égal à un littéral provenant d'un axiome. On mémorise l'origine de chaque littéral, dans le graphe de réfutation, pendant sa construction.

Suivant la polarisation et la position relative des sous-graphes, l'algorithme introduit des **lemmes**, conduit la preuve par une **analyse de cas**, etc.. Par exemple, pour introduire un lemme, tous les sous-graphes concernés doivent être positifs.

La preuve finalement obtenue peut être linéarisée par la méthode de Chester [Che76] (voir la section précédente).

Toutefois, il est préférable de **linéariser séparément les lemmes** obtenus pendant la recherche de la preuve formelle ou sa traduction en déduction naturelle.

Finalement, on peut essayer de simplifier la preuve linéarisée. Mais cette simplification dépend essentiellement du contexte d'utilisation de la preuve et de son lecteur. Notons que donner un **modèle** de ce lecteur est envisageable.

Choix des formes des références et production des messages [Hua90]

Après linéarisation, une preuve en déduction naturelle peut finalement être traduite automatiquement en langage naturel [Che76].

Pour traduire une ligne, le programme de traduction doit écrire des références aux formules déjà démontrées et à la règle d'inférence utilisées.

La forme d'une référence peut être *explicite*, *omise* ou *implicite*. Elle est explicite si elle dénote littéralement une formule ou une règle d'inférence. Elle est implicite si elle n'indique qu'un moyen de retrouver celle-ci.

Le choix de la forme de référence pour une **formule** dépend de la présence de celle-ci à l'esprit de l'utilisateur. Cette présence est fonction des **distances physique et structurelle** entre la ligne courante et la dernière apparition de la formule.

Pour formaliser la notion de distance *structurelle*, on peut décomposer la preuve de déduction naturelle en *unités*. Ces unités sont définies par l'utilisation des *règles d'inférence structurelles* (un certain sous-ensemble des règles d'inférence du système de déduction naturelle), et celle des définitions et des théorèmes du *niveau courant*. On considère en effet qu'une théorie mathématique est structurée en niveaux, chaque niveau correspond à une ou plusieurs sous-théories.

La valeur de la distance structurelle, *près* ou *loin*, dépend des positions relatives des plus petites unités contenant la formule référencée et la ligne courante.

La distance physique prenant le même type de valeur, on associe une forme de référence à chacune des quatre combinaisons possibles:

Distance Physique	Distance Structurelle	
	près	loin
près	omise	implicite ou omise
loin	implicite ou explicite	explicite

Le Choix de la forme de référence pour une **règle d'inférence** dépend de sa familiarité vis à vis de l'utilisateur. Cette familiarité est essentiellement fonction de la théorie mathématique utilisée: Les règles dérivées des niveaux inférieurs sont omises, celles du niveau courant sont implicites ou explicites.

Les règles d'inférence structurelles sont explicites, les autres sont omises.

Finalement, le programme de traduction en langage naturel combine les choix des formes des différentes références contenues dans chaque ligne, pour produire un message. Toutefois, il peut encore modifier ces choix dans certains cas particuliers.

b) Analyse de preuves en langue naturelle

Comme on ne connaît pas de formalisation de la langue naturelle, ni même des preuves en langue naturelle, on ne sait pas les analyser complètement.

D'autre part, la traduction **manuelle** en preuves formelles, de preuves en langue naturelle, est une **perte de temps et le plus souvent d'informations**.

On sait pourtant analyser les lexèmes et les parties purement formelles, comme certaines formules mathématiques, des preuves en langue naturelle. Pour aller plus loin, on peut raisonnablement penser que leur structure obéit à une certaine **grammaire** et tenter d'extraire celle-ci de manière automatique.

Mais le problème général d'analyse entièrement automatique de preuves en langage naturel est **extrêmement difficile**, et loin d'être résolu.

Une grammaire pour les preuves orales [Dyb82]

Plus proche de la pensée de son auteur, une preuve orale est souvent moins formelle qu'une preuve écrite correspondante. Toutefois, comme l'interlocuteur ne peut pas accéder librement à son contenu, elle doit contenir plus de **connexions**, **références** et **déclarations** explicites. Ces informations supplémentaires facilitent l'analyse de la preuve.

On ne sait pas entièrement formaliser une langue naturelle. Cependant, on peut supposer qu'une preuve soit une *unité de discours* de structure rigoureuse, au même titre qu'une blague ou une histoire.

Afin de capturer cette structure, une analyse de nombreuses preuves orales a été tentée. Cette analyse a été menée de manière purement syntaxique, en évitant d'interpréter les preuves orales par des preuves formelles.

Les données expérimentales regroupent un grand nombre de preuves orales, contenant les interruptions de l'interlocuteur.

Différents locuteurs, choisis pour éviter les divergences de structure dues à la langue naturelle utilisée (Anglais) et à leur pays d'origine, et mis dans des conditions sociales similaires, ont émis ces différentes preuves orales.

L'analyse de ces données a mis en évidence une structure syntaxique générale des preuves orales, malgré les variations dans l'explicitation des hypothèses, des déclarations et des références et dans la taille des pas d'inférence, dues aux différents locuteurs. Cette structure syntaxique se compose de deux niveaux.

La structure de haut niveau décrit l'organisation générale de la preuve orale, interruptions comprises. De nombreuses marques orales la matérialisent: "I want to prove that..." ..., "...hence <THEOREM>.".

Une suite de transformations d'une structure logique arborescente incomplète et d'un pointeur vers un sous-arbre de cette structure permet de la **modéliser**. Ces transformations, invoquées par les marques orales, consistent à compléter pas à pas la structure logique ou à déplacer le pointeur.

[Dyb82] donne une grammaire et un ensemble de règles de réécriture d'arbres formalisant la structure logique et les différentes transformations possibles, pour une certaine classe de preuves orales.

La structure de bas niveau est celle des formules mathématiques, qui correspondent à certains terminaux de la structure logique précédente. Les parenthèses ne sont pas prononcées, mais matérialisées par des pauses et des changements d'intonation.

Exemple:

La formule $(A \cap B) \times (C \cap D) = (A \times C) \cap (B \times D)$ se prononce:

"A intersect B cartesian product with
C intersect D, equals
A cartesian product C intersect with
B cartesian product D."

Cette structure peut être modélisée par le même type de transformations que celles de la structure de haut niveau.

Vérification de preuve en langue naturelle écrite [Sim88]

La formalisation manuelle d'une preuve en langue naturelle peut être assez longue. De plus, elle peut détruire certaines informations utiles au lecteur. [Sim88] décrit un vérificateur de preuves tentant compte d'informations habituellement éliminées. Les preuves considérées sont celles d'un livre d'algèbre élémentaire donné, "Elementary Theory of Numbers" de LeVeque W. J..

La preuve à vérifier est fournie sous forme d'un fichier LATEX. *LATEX* est un langage de description de documents souvent utilisé avec le formatteur de documents TEX, qui permet de produire des documents mathématiques de grande qualité.
LATEX permet donc une représentation très peu destructrice des preuves en langue naturelle écrite.

Dans un premier temps, la preuve est analysée et traduite dans un langage intermédiaire. Chaque phrase est transformée en une liste contenant des formules mathématiques formelles et les mots situés hors de ces formules. L'utilisateur doit **expliciter manuellement les références implicites**, en particulier les pronoms.

PROOF CONNECTOR est un programme capable d'analyser grammaticalement le langage intermédiaire, d'appeler un démonstrateur automatique pour effectuer les mêmes déductions qu'un lecteur de la preuve.
Pendant l'analyse, il gère le but, le contexte d'hypothèses, lemmes et définitions, et le contexte structurel courants. Le contexte structurel indique si l'analyseur considère un pas de récurrence, un cas d'une analyse par cas, etc..

PROOF CONNECTOR est écrit en quelques lignes d'un dialecte PROLOG admettant une certaine forme de variables du second ordre. La simplicité de la grammaire employée suggère que **le système ne s'éloigne pas trop de la "pensée" de l'auteur de la preuve**.

Un démonstrateur automatique est en cours d'expérimentation. Plus généralement, le démonstrateur automatique employé devrait être assez puissant pour déduire le but dans les contextes courants, tout en retournant une preuve formelle simple de cette déduction. Cette preuve formelle permettrait à PROOF CONNECTOR de construire la preuve formelle globale. En cas d'échec, il devrait s'interrompre rapidement pour laisser PROOF CONNECTOR tenter d'autres chemins de la grammaire.

Le vérificateur a permis de vérifier un sixième des preuves du livre considéré. Néanmoins, il a fallu expliciter manuellement la plupart des pronoms.

6. Discussion

Malgré leur difficulté, voire leur incapacité à démontrer la plupart des théorèmes mathématiques connus, **l'apport** des outils d'inférence à la recherche en mathématiques a été substantiel.

Les démonstrateurs automatiques ont permis l'expérimentation et le développement de **systèmes formels** aux propriétés formelles de plus en plus fortes. Les vérificateurs automatiques et les démonstrateurs interactifs ont permis ceux de **formalismes** au pouvoir d'expression de plus en plus grand. Actuellement, il existe des **cadres logiques** capables d'exprimer le langage, le calcul et les preuves de larges classes de systèmes formels.

Quelques anciennes **conjectures**, comme le théorème des quatre couleurs, ont pu être démontrées de manière automatique. Leurs preuves ont une structure simple, mais elles sont de tailles trop importantes pour l'être humain.

Un autre apport non négligeable est la spécification de **ML**, un langage de programmation en même temps polymorphe et fortement typé.

Certains outils d'inférence, comme OTTER, HOL connaissent du **succès** auprès des utilisateurs. Voir des outils d'inférence utilisés par des **non informaticiens**, notamment pour l'enseignement [Sup84], est aussi très encourageant.

Néanmoins, la plupart des outils d'inférence ont été **abandonnés**. Les outils d'inférence modernes **intégrent** plus ou moins bien leurs possibilités. Par exemple, PC [McC62] était capable de vérifier une preuve en effectuant les applications de règles d'inférence programmées par l'utilisateur. Bien que la programmation de la plupart des notions relatives aux systèmes formels fusse laissée à l'utilisateur, la

similarité entre ce principe et les opérateurs de tactiques des démonstrateurs interactifs, beaucoup plus récents, est remarquable.

De nombreux outils d'inférence, anciens ou récents, servent **actuellement** à l'expérimentation. A différents niveaux de formalisation, ils recouvrent les fonctions de vérification automatique, démonstration automatique ou interactive, mémorisation et transformation que nous avons présentées.

Aucun d'entre eux ne possède les capacités de tous les autres. Le projet CAP [Hir86] montre bien la possibilité d'envisager la réalisation d'outils d'inférence pour chaque théorie mathématique connue, selon la méthode théorie. Le fait que des outils d'inférence fondés sur cette méthode soient effectivement utilisés [Sup84] **justifie** ce travail **colossal**. Mais un tel système n'est **pas réalisable** à court terme.

Une autre possibilité est de réaliser un environnement gérant **l'utilisation simultanée** de différents outils d'inférence, en assurant notamment la correction et la stabilité.

Cet environnement aurait des **possibilités** inexistantes dans les systèmes actuels.

Les **démonstrateurs automatiques**, ont été décevants, relativement aux espoirs fondés sur eux. Pourtant, ils peuvent être très utiles au sein de **démonstrateurs interactifs**.

Entre formalismes, les **transformations** permettraient au système de récupérer des théories, des logiques ou des preuves formelles existantes. Entre logiques, elles feraient collaborer différents démonstrateurs automatiques. Au sein d'une logique, elles aideraient les démonstrateurs automatiques (mise sous forme clausale, nommage).

Plus tard, le traitement de la **langue naturelle** permettrait à l'utilisateur d'introduire des preuves ou de récupérer des théories, des logiques ou des preuves informelles qui existent dans les nombreux livres mathématiques. Les preuves ainsi récupérées pourraient être transformées en langage naturel, puis en preuves formelles. On pourrait alors les vérifier automatiquement ou les présenter de manière hiérarchique et structurée. La capacité de générer et de reconnaître les preuves en langue naturelle pourrait être un facteur déterminant pour l'acceptation des outils d'inférence par les mathématiciens.

Vu la généralité de la macro-structure des preuves en langue naturelle [Dyb82], on pourrait même réaliser un **analyseur indépendant de la logique** utilisée. Les parties de preuves spécifiques à la logique utilisée (formules mathématiques) seraient laissées à des analyseurs spécifiques. Bien entendu, ces analyseurs pourraient être différents selon "la" langue naturelle utilisée (langue, jargon, preuve écrite ou parlée, etc.).

On peut envisager d'utiliser de nombreux autres outils. Un formatteur de documents (LATEX) aiderait l'impression de preuves. Un

outil de calcul formel (*MATHEMATICA*, *MAPLE*) vérifierait des théorèmes ou simplifierait des formules dans des théories mathématiques connues, comme l'algèbre élémentaire.

On assemblerait les outils d'inférence pour former des **chaînes**. Par exemple, une chaîne ambitieuse pourrait regrouper les outils de reconnaissance de l'écriture, analyse de la langue naturelle, transformation en logique du premier ordre, nommage, mise sous forme clausale, démonstration automatique, transformation en déduction naturelle, linéarisation, génération de langage naturel.

Dans ces chaînes, les classiques clavier, souris, écran, imprimante ou les moins courants stylet, microphone, scanner, caméra, etc. serviraient d'interfaces utilisateur matérielles.

La plupart des outils d'inférence sont d'un **abord difficile** pour l'utilisateur. Cela provient de leur **théorie sous-jacente** ou de leur **méthode de contrôle**. Notamment, les théories sous-jacentes de certains vérificateurs automatiques, et les méthodes de contrôle de certains démonstrateurs automatiques sont particulièrement difficiles. De plus, leurs interfaces utilisateur sont plus ou moins évoluées et leurs présentations sont plus ou moins riches.

Leur utilisation simultanée pose d'autres problèmes.

Ils sont **hétérogènes**. L'utilisateur doit souvent apprendre la théorie sous-jacente et la méthode de contrôle de chaque outil d'inférence qu'il emploie. Notons que plusieurs démonstrateurs interactifs hétérogènes peuvent être utilisés à partir d'une même interface utilisateur graphique [TBK92].

Ils sont **isolés**. La programmation et la vérification, par l'utilisateur, d'interfaces de communication entre plusieurs outils d'inférence peut être longue et difficile. Pour n outils d'inférence $(n>0)$, jusqu'à $n(n-1)/2$ interfaces (une pour chaque paire d'outil) sont à réaliser.

Malgré leur multiplicité, nous avons pu **classer** les outils d'inférence selon leur fonction principale et mettre en évidence un certain nombre de **similarités**.

On conçoit donc aisément un environnement pour **homogénéiser** et **regrouper** les accès à différents outils d'inférence. Pour la méthode de contrôle, une interface utilisateur unique factoriserait les commandes usuelles comme le réglage des paramètres, la décomposition de buts, l'appel de règles d'inférence ou de tactiques, etc.. Pour la théorie sous-jacente, une représentation **générique** unique factoriserait les notions usuelles telles que formule, système formel, etc.. La majorité de la théorie sous-jacente et de la méthode de contrôle serait donc **apprise une fois pour toutes**.

Concernant **l'abord difficile**, la théorie sous-jacente et la méthode de contrôle de l'environnement ne serait pas toujours plus simple que celles des outils d'inférence employés. Toutefois, **un éditeur structuré, une interface utilisateur graphique moderne et une présentation graphique usuelle** aideraient l'utilisateur à surmonter sa réticence à utiliser des langages de programmation ou des formalismes spécifiques.

Pour rompre **l'isolement**, la **généricité** de l'environnement et l'utilisation de **vues multiples** faciliterait la communication entre outils d'inférence et la possibilité de considérer plusieurs logiques en même temps.

L'utilisation d'un cadre logique, fondé par exemple sur le principe de Curry-Howard, est un fondement solide pour la réalisation d'un tel environnement.

Un cadre logique permet **d'exprimer** le langage, le calcul et les preuves d'une large classe de systèmes formels.

Ainsi, il constitue un support pour la **communication** entre l'interface utilisateur et les divers outils d'inférence (démonstrateurs automatiques, bibliothèques, etc.).

Il assure la **correction** des preuves manipulées et la **stabilité** de l'environnement, lors de l'ajout de nouveaux outils d'inférence. Vérifier la correction des outils d'inférence utilisés n'est plus nécessaire, contrairement à VT [DS79].

L'environnement peut **évoluer**. Si le cadre logique était changé, un transformateur récupérerait automatiquement les informations (systèmes formels, théories et preuves), seules les interfaces entre les outils d'inférence et le cadre logique resteraient à modifier.

Comme le projet CAP [Hir86], la réalisation d'un environnement fondé sur un cadre logique est un pas vers celle d'un **système universel**. Toutefois, la méthode cadre logique ne semble pas imposer un travail aussi **colossal**, elle assure naturellement la **correction** et la **stabilité**, et peut facilement **récupérer** les théories et les preuves développées avec des méthodes théories ou logiques.

C) Édition graphique et structurée

Cette section présente différents types d'éditeurs existants. L'analyse de leurs caractéristiques a permis de définir celles de l'éditeur de preuves. En particulier, l'étude des éditeurs de documents a mis en évidence des **idées** encore **inexploitées** dans les démonstrateurs interactifs et les éditeurs de preuves existants.

Les éditeurs existants peuvent être classés en quatre familles:

- Les éditeurs **textuels**[14].
- Les éditeurs **structurés** qui permettent la manipulation de textes munis d'une structure de haut niveau.
- Les éditeurs **graphiques** qui permettent la manipulation d'éléments graphiques plus divers, avec des dispositions plus souples et plus riches.
- Les éditeurs simultanément **graphiques** et **structurés**.

Les éditeurs de ces quatre familles peuvent aussi être **génériques**. Un éditeur *générique* permet de **spécifier la structure ou la présentation** des informations à éditer.

Nous ne discuterons pas d'interfaces utilisateur graphiques modernes, ni des commandes du genre *annuler*, *couper*, *copier*, *coller*, *effacer* ou *déplacer*, présentes dans la majorité des éditeurs récents. Nous supposerons ces notions connues.

Pour introduire les éditeurs textuels, considérons la notion de programme. Un *programme* possède une **structure** qui obéit à une syntaxe déterminée, avec des **contraintes contextuelles**. Son **aspect visuel** est conçu pour un affichage sur n'importe quel terminal ou imprimante, c'est à dire textuel.

Les programmes sont souvent édités à l'aide d'éditeurs textuels tels que *VI* ou *EMACS*. Il existe aussi quelques rares éditeurs structurés comme celui de *THINK PASCAL*[15] (Macintosh). En effet, EMACS n'est pas un véritable éditeur structuré, il permet bien de compléter des programmes à l'aide de formats prédéfinis, mais il ne vérifie pas la syntaxe de manière interactive.

Pour introduire les éditeurs graphiques, considérons la notion de document. Un *document* est souvent d'**une grande richesse graphique**. Il peut contenir des éléments graphiques de différentes natures, tels que textes, graphiques, tableaux, schémas, formules, photographies, disposés de manière **structurée**. Cette structure est relativement **souple**. Les contraintes contextuelles sont relativement faibles. Elles servent à numéroter les figures ou les sections, et au traitement des références.

Un document peut être édité avec un éditeur textuel pour être imprimé avec un formatteur de document tel que *LATEX*. D'ailleurs, on pourrait facilement écrire un éditeur structuré fondé sur LATEX.

De nombreux éditeurs graphiques (nommés *WYSIWYG*) tels que *MACWRITE*, *MACDRAW*, *MACPAINT*, *WORD* (Macintosh), permettent conjointement de produire des documents de qualité acceptable, de

---

[14]Un *texte* est une suite de lettres et de symboles de positionnement (espaces, tabulations, retours à la ligne etc....).

[15]L'éditeur de THINK PASCAL ne vérifie pas les contraintes contextuelles.

manière rapide et agréable. Ils proposent une aide superficielle à la structuration (paragraphes, styles, mode plan, etc.).

Plusieurs éditeurs de documents graphiques et structurés, tels que *INTERLEAF*, *MENTOR-RAPPORT*, *FRAMEMAKER* ou *TEXTURES* existent aussi. Notons que TEXTURES (Macintosh) est un éditeur graphique et structuré pour les documents LATEX.

Nous présenterons d'autres éditeurs existants, de manière plus approfondie, leurs principes (généricité, etc.) étant particulièrement intéressants.

En résumé, une *information* (programme, document ou autre) peut être graphique ou non. Cependant, postulons que **toute information est structurée**.

A priori, une image est une information graphique non structurée. On montre pourtant (par échantillonnage et dénombrement) que la plupart des images possibles n'existent pas dans l'univers. En pratique, une image possède, le plus souvent, une structure que l'on ne formalise pas (personnages, objets, etc.).

L'utilisation d'éditeurs non structurés consiste à **laisser l'utilisateur vérifier lui-même la structure de l'information éditée**. Notamment, c'est le cas des éditeurs de photographies tels que PHOTOSHOP (Macintosh).

Du point de vue informatique, une **preuve** est une information fortement **structurée**, pouvant avoir des **contraintes contextuelles** (le fait qu'une preuve soit complète, qu'une branche d'un tableau sémantique soit fermée, etc.). Du point de vue humain, c'est une information **graphique** pouvant contenir de nombreux symboles et graphismes.

Présentons donc des éditeurs structurés, dont certaines caractéristiques manquent aux éditeurs graphiques et structurés. Présentons ensuite des éditeurs graphiques et structurés pour les documents, puis les preuves.

1. Édition générique structurée

En informatique, les programmes sont les informations structurées, avec contraintes syntaxiques, les plus manipulées. Les outils les plus **avancés**, notamment génériques, en matière d'édition, sont donc des éditeurs **textuels**.

CSG [Gri87]

CSG (Cornell Synthesiser Generator) est un programme capable de produire un éditeur structuré, à partir d'une **spécification des termes à éditer**.

Cette spécification est composée d'une grammaire hors-contexte, d'informations d'affichage, de règles d'analyse, de contraintes contextuelles et de commandes spécifiques. Les contraintes contextuelles sont spécifiées par des **attributs** ajoutés à la grammaire.

Les éditeurs produits par CSG partagent la même interface textuelle. Chaque éditeur maintient **l'arbre de dérivation** de la grammaire hors-contexte, correspondant au fichier en cours d'édition.

Il propose les commandes spécifiques et des commandes usuelles standard, indépendantes de la spécification. Ces commandes permettent de transformer l'arbre de dérivation. Copier, couper, coller ou déplacer un sous-arbre de dérivation sont des exemples de commandes usuelles standard.

Après chaque transformation, les attributs sont mis à jour de manière **incrémentale**, et les contraintes contextuelles non respectées sont **mises en évidence**.

Lors de sa construction, un arbre de dérivation peut contenir des **trous**, que l'utilisateur peut compléter au clavier ou par les commandes.

CENTAUR [BCDIKLP87], [JR92]

CENTAUR est une version de seconde génération (en LISP) du système *MENTOR* (INRIA) utilisé depuis de nombreuses années comme environnement de développement de programmes.

Il forme un environnement interactif et générique. Étant données la syntaxe et la sémantique d'un langage, il produit automatiquement un éditeur structuré, un interprète et d'autres outils de mise au point pour ce langage.

Pour **spécifier** la syntaxe et la sémantique du langage objet, CENTAUR propose plusieurs langages auxiliaires.

METAL permet de spécifier les **syntaxes concrètes et abstraites**. Il admet des règles de grammaires pour la syntaxe concrète, annotées par des indications pour construire les arbres abstraits.

TYPOL permet de spécifier la **sémantique**. Il admet des règles de sémantique naturelle.

PPML permet de spécifier un pretty-printer pour **présenter** les termes. Il admet des règles constituées chacune d'un *filtre* et d'un *format*. Le filtre détermine les termes auxquels la règle est applicable. Le format décrit leur présentation sous forme d'une boîte pouvant

contenir des terminaux, des variables du filtre ou d'autres boîtes. Le *type* de chaque boîte indique si ses éléments doivent être disposés de manière horizontale, verticale, horizontale avec retour à la ligne, etc.. Les variables indiquent les emplacements des présentations des sous-termes qui leur correspondent. Elles permettent un appel récursif au pretty-printer.

Le **noyau** de CENTAUR représente et manipule les données internes.
Une machine logique PROLOG traite la **sémantique**.
VTP (Virtual Tree Processor) traite la **syntaxe abstraite**. C'est un module **générique** de manipulation de structures arborescentes annotées. Une classe de structures arborescentes annotées étant spécifiée, il fournit des primitives pour manipuler les structures de cette classe.
VTP est **général, fiable** et **stable, indépendant** de CENTAUR. Mais sa **rapidité** et la **compacité** des données manipulées sont encore à **améliorer**.
C'est le **cœur** de l'éditeur structuré de CENTAUR. Les primitives qu'il fournit constituent les manipulations de base de l'éditeur, les manipulations plus complexes étant programmées en LISP à partir de ces primitives.

L'**interface utilisateur** graphique, moderne (menus, boutons, fenêtres, etc.) et portable (fondée sur X-WINDOWS) de CENTAUR, a été réalisée à partir d'une bibliothèque standard *Le_Lisp*.

Les fenêtres de CENTAUR contiennent des **vues textuelles** pouvant être **éditées de manière structurée**. Les textes de ces vues correspondent aux différents formalismes prédéfinis de CENTAUR, ou définis par l'utilisateur, ou à des imbrications de ces formalismes. Ils peuvent dépendre les uns des autres, la modification de l'un entraînant la modification de plusieurs autres.

Bien qu'originalement prévu pour le développement de programmes, CENTAUR permet **l'édition d'autres types de documents structurés**.

CENTAUR-RAPPORT, HYPER-CENTAUR [Ver89]

CENTAUR-RAPPORT est une expérience similaire à *MENTOR-RAPPORT*. Il constitue une extension de CENTAUR pour l'édition de **textes structurés**.

HYPER-CENTAUR est une extension de CENTAUR facilitant la mise en place de liens non hiérarchiques, matérialisant les références à des

parties de documents. Quand on modifie un document contenant des objets référencés, ces liens sont automatiquement mis à jour.

L'application d'HYPER-CENTAUR à CENTAUR-RAPPORT permet de construire des outils de type **hypertexte** [Ver89].

THEO [Des88a], [Des88b]

THEO est un autre démonstrateur interactif générique réalisé sous CENTAUR et écrit en TYPOL. Les logiques objets sont elles aussi représentées en TYPOL. THEO bénéficie de l'IUGG de CENTAUR et de son langage de présentation graphique de termes, PPML.

Après avoir spécifié une logique objet et un séquent à prouver, l'utilisateur peut construire une preuve, de manière bottom-up, en choisissant un but à développer, une tactique à lui appliquer et ses arguments éventuels, avec la souris, jusqu'à ce qu'il n'y ait plus de buts.

Parmi ses tactiques primitives, THEO propose des facilités de démonstration automatique ("prolog", "breadth first", "complete"), d'édition ("cut", "paste") et permet de construire les preuves pas à pas ("user").

Contrairement au démonstrateur interactif décrit dans [Has88], THEO ne mémorise pas les arbres de preuves TYPOL complets, tels qu'ils sont présentés à l'écran. La forme choisie pour mémoriser les preuves est plus compacte et plus proche de la représentation des preuves dans les formalismes issus de AUTOMATH ([CH88], [HHP89]). Une preuve est représentée par un terme défini inductivement par (R est un nom d'un schéma d'axiome, d'un règle d'inférence ou d'un théorème de la logique objet, T est un séquent objet, P_i sont des termes qui représentent des preuves):

$R{:}T$	preuve primitive
T	séquent à prouver
$(R\ P_1 \ldots, P_n)$	application

Avec cette représentation, une seule règle de résolution suffit pour construire les preuves:

$$P \vdash n{:}\frac{p}{c}, c \ \rightarrow \ P[(n\ p)/c]$$

Où P, $n{:}\frac{p}{c}$ et c sont respectivement des instances de la preuve courante, de la règle d'inférence objet de nom n et d'un séquent à prouver de la preuve courante. Ces instances sont déterminées par unification de la conclusion de la règle d'inférence objet choisie et du séquent à prouver choisi dans la preuve courante.

[Des88a] présente ensuite différentes alternatives pour représenter les preuves. Pour chacune de ces alternatives, on peut définir une règle de résolution similaire.

La première alternative, moins compacte, consiste à ajouter un argument à l'application $(R\ P_1 \ldots, P_n)$ pour chaque méta-variable de la règle d'inférence objet de nom R qui apparaît dans sa conclusion, mais pas dans ses prémisses. Cet argument sert à mémoriser les termes substitués à ces méta-variables, au sein de la preuve, ce qui n'était pas possible avec la forme de représentation précédente.

La deuxième alternative consiste à considérer les preuves comme des schémas dont les variables représentent les séquents à prouver. Cette alternative se rapproche de la représentation des preuves dans les formalismes issus de AUTOMATH.

En fait, dans ces formalismes, les preuves et les règles d'inférence objets sont représentées par des schémas dont les variables représentent non seulement les hypothèses (séquents à prouver) et les prémisses, mais aussi toutes les méta-variables de ces hypothèses et prémisses.

Telle quelle, cette représentation est beaucoup moins compacte que celle de THEO. Mais la synthèse automatique des arguments implicites [CH88] permet de représenter les preuves de ces formalismes par des termes plus compacts et plus proches de ceux de THEO.

Application à l'édition de preuves [Has88]

L'utilisation de CENTAUR pour construire des **arbres de preuves** pas à pas est presque **directe**. Tous les systèmes de Gentzen peuvent être définis en TYPOL, sous leur présentation usuelle. [Has88] montre ainsi comment réaliser un démonstrateur interactif fondé sur les notions de but et de tactique.

Ce démonstrateur interactif permet la construction "top-down" de preuves dans deux fenêtres, une fenêtre **preuve** et une fenêtre **règles**. Le processus de construction correspond au cycle principal suivant:

- **Sélectionner** un *but* non résolu dans la fenêtre preuve (avec la souris).
- Demander le **développement** de ce but. C'est à dire faire apparaître la liste des règles d'inférences applicables dans la fenêtre règles.
- **Sélectionner** une règle d'inférence dans la fenêtre règles.
- Demander l'**application** de la règle d'inférence sélectionnée au but sélectionné.

Après application d'une règle d'inférence, ses prémisses forment de nouvelles feuilles de l'arbre de preuves, qui constituent de nouveaux buts à résoudre.

Quelques commandes auxiliaires facilitent l'utilisation du démonstrateur interactif:

- Certains buts peuvent être **évalués** de façon externe. Une évaluation réussie peut générer de nouveaux buts à résoudre.
- Les pas de preuves **intermédiaires** peuvent être **cachés**[16].
- Les différentes transformations d'une preuve peuvent être **annulées**. On peut ramener celle-ci à n'importe quel état précédent.
- L'ouverture et la fermeture d'un niveau de profondeur numérotée permet **d'isoler** la résolution d'un but. Cet isolement correspond à considérer le but comme un lemme.

Notons que la portée de la commande d'annulation est limitée au niveau courant. La mémoire utilisée pour l'annulation au delà du niveau courant peut être **récupérée** sur demande.

Une possibilité particulièrement importante est celle d'automatiser la production d'applications de règles d'inférence, notamment par induction structurelle, à l'aide de **tactiques**.

Les commandes (sélection, développement, application) et les fonctions (affichage, menus) de l'éditeur forment les tactiques primitives. Les règles TYPOL (compilées en programmes PROLOG) permettent de composer ces tactiques primitives.

Mélanger les tactiques automatiques et les décisions de l'utilisateur est très facile. En fait, l'interaction de l'éditeur avec l'utilisateur est considérée comme une tactique particulière:

$$\frac{\text{display}\langle\sigma_1\rightarrow\sigma_2\rangle \quad \text{menu}\langle\sigma_2\rightarrow\text{CMD},\sigma_3\rangle \quad \overset{\text{command}}{\sigma_3 \vdash \text{CMD}: \sigma_4} \quad \text{tactic}\langle\text{TAC}\rangle \quad \sigma_4 \vdash \text{TAC}}{\sigma_1 \vdash \text{"user"}}$$

La description précédente définit la tactique "user", qui consiste à:

- **Afficher** l'état courant.
- **Attendre** une **commande** utilisateur.
- **Exécuter** la commande.
- **Appeler** la tactique courante, qui est le plus souvent la tactique "user".

[16]Cette possibilité ne correspond pas à l'introduction de règles dérivées, tous les pas de preuves sont ici cachés en bloc.

2. Édition générique graphique et structurée de documents

Les éditeurs graphiques de documents les plus évolués sont des éditeurs génériques. Considérer une structure est **indispensable** pour décrire une présentation graphique générique. Présentons donc des éditeurs de documents en même temps graphiques, génériques et structurés.

LILAC [Bro88]

LILAC est fondé sur l'utilisation de deux fenêtres contenant deux *vues* différentes du document édité. Une vue **textuelle**, appelée *vue source*, contient le programme qui décrit le document par sa *structure logique*, sa *présentation* et son *contenu*. Une vue **graphique** WYSIWYG, appelée *vue page*, contient l'image du document, c'est à dire la sortie du programme.

L'utilisateur peut indifféremment éditer les deux vues à l'aide du clavier et de la souris. En pratique, il modifie essentiellement la vue page, ce qui correspond à **modifier le programme en modifiant sa sortie**.

La vue source permet d'effectuer les manipulations difficiles ou impossibles dans la vue page (structure logique, contraintes, comportement conditionnel). Mais le programme sert surtout à **décrire le comportement des manipulations WYSIWYG**.

Une structure de données hiérarchique est associée à chaque vue.

L'arbre abstrait syntaxique du programme correspond à la vue source. LILAC le manipule plus facilement que le texte même de la fenêtre.

Une structure qui décrit le type, la position et les dimensions des éléments constituant l'image du document correspond à la vue page. Cette structure est formée de *boîtes* et de *colle*.

Les notions de boîtes et de colle sont empruntées à TEX. Une boîte est un rectangle invisible contenant une substance visible, l'*encre*. Une page, une ligne, un mot sont des exemples de boîtes. Les boîtes sont reliées entre elles par de la colle, une substance invisible matérialisant les espaces entre les boîtes et l'élasticité de ces espaces. Les boîtes et la colle sont disposées en listes horizontales ou verticales, formant de nouvelles boîtes appelées *boîtes composées*.

Cette *structure de boîtes* joue un rôle fondamental pour la sélection, la mise en évidence et la mise à jour incrémentale de l'image: elle permet de lier l'arbre syntaxique abstrait à son image, de manière bidirectionnelle.

Chaque structure de données peut être traduite vers l'autre par un algorithme de transfert. La **rapidité** de ces algorithmes est vitale:

- Ces algorithmes sont incrémentaux: Si l'on modifie une partie de l'une des structures, l'algorithme recalcule uniquement la partie correspondante dans l'autre structure.
- Le langage de programmation utilisé est de type fonctionnel, mieux encapsulé qu'un langage de macros (comme celui de TEX). Des appels fonctionnels demeurent en effet des unités syntaxiques, alors qu'une structure de macros est développée en une chaîne de caractères.
- La structure des programmes est la plus proche possible de celle des documents, et donc de la structure de boîtes.

L'éditeur WYSIWYG a été étudié pour traiter le plus simplement possible les structures définies par l'utilisateur. C'est la structure logique qui spécifie les manipulations WYSIWYG possibles. En cela, LILAC se comporte comme un éditeur syntaxique indirect.

EDIMATH [Qui87]

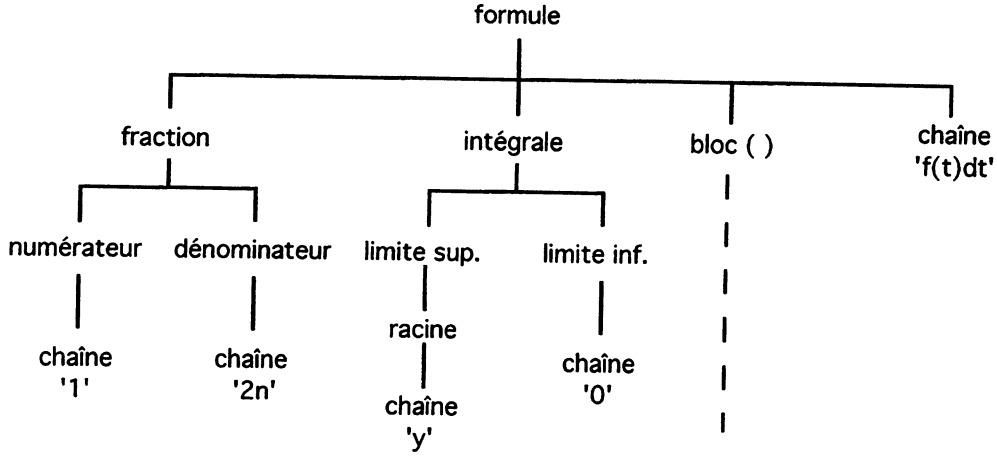
EDIMATH est destiné aux **formules mathématiques**. L'intérêt de ces formules est leur **structure logique figée**. On représente une formule mathématique par un arbre dont les feuilles sont des symboles ou des chaînes de caractères, et les nœuds sont les *constructeurs* de la structure logique.

La formule[17]:

$$\frac{1}{2n}\int_0^{\sqrt{y}}\left(\sum_{k=1}^{n}\sin^2 x_k(t)\right)f(t)dt$$

[17]L'image présentée a été effectivement construite avec EDIMATH.

est ainsi représentée par l'arbre:



Comme LILAC, EDIMATH est fondé sur l'utilisation d'une structure de boîtes, utile pour l'affichage, et la reconnaissance des sélections faites avec la souris.

Mais cette structure de boîte est isomorphe à l'arbre qui représente la formule: Une boîte est associée à chaque constructeur, symbole ou chaîne de caractère utilisé. Les règles de positionnement et de dimensionnement de ces boîtes sont figées.

Toutes les manipulations de formules peuvent donc se faire dans une vue graphique, **EDIMATH est un éditeur à une seule vue.**

L'algorithme de calcul des boîtes effectue un parcours "bottom-up" de l'arbre représentant la formule.

Au niveau de chaque feuille, il calcule les dimensions d'un rectangle appelé boîte, englobant la chaîne de caractères associée.

Au niveau de chaque nœud, il positionne les symboles et les sous-boîtes contenus, avant de calculer les dimensions de la boîte englobante. Ce positionnement est défini par le constructeur associé au nœud, il tient compte de la taille des symboles et des sous-boîtes, et évite les recouvrements.

Liste des constructeurs définis dans EDIMATH:

- *exposant*: un constituant,
- *indice*: un constituant,
- *fraction*: deux constituants, le numérateur et le dénominateur,
- *racine*: un constituant,
- *intégrale*: deux constituants, les deux bornes,
- *triple*: trois constituants, principal, inférieur et supérieur,

- *vecteur*: un nombre variable de constituants, les éléments,
- *bloc*: trois constituants, les deux symboles délimiteurs et le contenu.

En plus de ces huit constructeurs de base, et des chaînes de caractères, EDIMATH utilise la notion de symbole. Un *symbole* est un caractère qui doit subir un traitement particulier. Deux formes de symboles sont utilisées: les symboles de grande taille (union, somme) et les symboles élastiques (radical, barre de fraction).

Notons que l'utilisation de symboles de grande taille pourrait maintenant être évitée en utilisant des polices de caractères définies mathématiquement (Postscript, TrueType), dont les caractères peuvent être agrandis à volonté.

Pour l'édition graphique et structurée de documents, on pourrait tenter de généraliser l'utilisation de constructeurs au sens d'EDIMATH. C'est l'approche choisie dans *MENTOR-RAPPORT*. Cette solution affaiblit toutefois la richesse des documents en spécifiant leurs entités et leur organisation de manière définitive.

GRIF [Qui87]

Contrairement à LILAC, *GRIF* **sépare** la structure logique, la présentation et le contenu des documents.

La structure logique et la présentation sont programmées de manière extérieure à l'éditeur. Le contenu est édité de manière WYSIWYG. Comme EDIMATH, du même auteur, GRIF est **un éditeur de documents à une seule vue**.

Le langage de programmation des structures logiques, appelé *méta-modèle*, permet de décrire de façon homogène des *modèles* représentant des classes de documents. Certains documents peuvent contenir des objets tels que tableaux ou formules mathématiques. Les classes de ces objets peuvent être définies dans le méta-modèle. Leurs instances peuvent être insérées dans tout document. Par exemple, l'auteur a utilisé le méta-modèle pour redéfinir les formules mathématiques telles qu'elles existent dans EDIMATH.

Le méta-modèle est un langage de description qui permet de définir des éléments génériques. Le **nommage** de ces éléments introduit une certaine forme de **typage**.

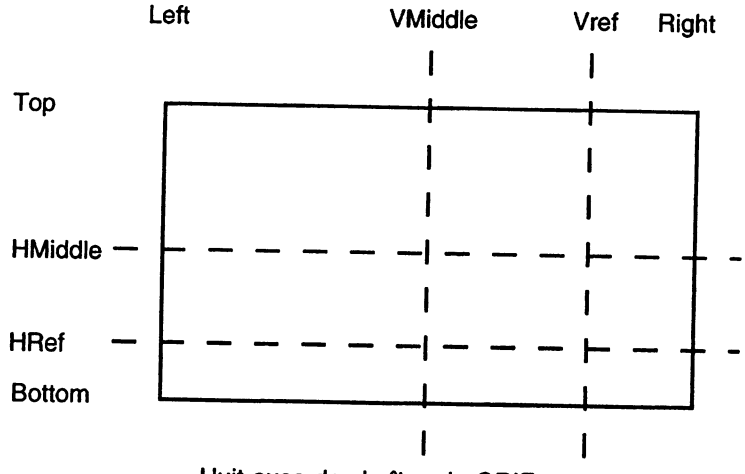
Les éléments génériques primitifs sont la chaîne de caractères, le graphique (formes géométriques simples), l'image et le symbole. Les constructeurs suivants permettent de composer ces éléments:

- L'*agrégat* définit un élément constitué d'une suite d'éléments de types donnés.
- La *liste* définit un élément formé d'une succession d'éléments de type donné.
- Le *choix* définit un élément dont le type appartient à un ensemble donné.
- L'*unité* (extension du choix) représente un élément primitif, un modèle externe ou un élément exportable de la structure logique.
- La *référence* représente un renvoi à un élément de type donné ou à un *élément associé* (note, graphique) dont la position n'est pas figée dans le document.

Finalement, des *attributs* sémantiques peuvent être associés à tout élément, primitif ou composé, pour indiquer s'il constitue une entrée d'index, le titre d'une œuvre, s'il est dans une langue particulière, etc..

Un autre langage de programmation permet de décrire la présentation en termes abstraits, indépendamment de tout appareil. Toutefois, la présentation finale tient compte des limites de l'appareil utilisé.

Une structure de boîtes représente les documents de manière interne. Comme dans EDIMATH, et contrairement à LILAC, il n'y a **pas de colle**. Mais des règles de positionnement et de dimensionnement fondées sur la définition de huit axes, sont associées à chaque boîte.



Huit axes des boîtes de GRIF.

L'objet du langage de présentation est de construire la structure de boîtes, et les instances de règles associées, à partir de la structure logique et du contenu du document édité.

Pour cela, il permet d'introduire de nouvelles boîtes pour la numérotation, les en-têtes et les règles de mise en page, de décrire le contenu des boîtes des images, des éléments graphiques et des

symboles, de décrire la mise en valeur des attributs. Finalement, il permet de spécifier les relations donnant les dimensions et les positions relatives des différentes boîtes utilisées: les boîtes associées aux éléments de la structure logique du document, les boîtes de présentation ajoutées pour les besoins de la présentation et les boîtes de mise en page crées implicitement par les règles de mise en page.

En résumé, GRIF est un éditeur de documents, graphique, structuré, à une seule vue et paramétrable par la structure logique et la présentation des documents. En cela, il constitue une **généralisation d'EDIMATH, de MENTOR-RAPPORT et de LILAC.**

Autres principes de visualisation [SB92]

L'image d'un document étant donnée, plusieurs méthodes permettent sa visualisation sur un écran. La plupart de ces méthodes présentent des inconvénients: Si on réduit l'image à la taille de l'écran, ses **détails** peuvent disparaître. Si on ne visualise qu'une portion de l'image, en la déplaçant à l'aide d'ascenseurs ou d'hyper-liens, la **structure globale** de l'image n'apparaît pas. Si on utilise plusieurs vues, de plus en plus rapprochées, des portions d'écran sont **redondantes**, donc gaspillées.

[SB92] propose une méthode de visualisation résolvant ces trois problèmes. Elle consiste à visualiser l'image, comme si la flèche de la souris était une loupe.

Le programme décrit s'applique aux graphes. De nombreuses méthodes existent, pour calculer une présentation esthétique de graphes. On suppose donc une telle présentation donnée, sous la forme suivante:
A chaque nœud est associée une *boîte* et une *importance* relative. La boîte est définie par sa taille et la position de son centre. Les arêtes du graphe relient les centres de différentes boîtes.

Lorsque l'utilisateur déplace le point de mire, matérialisé par la flèche de la souris, les boîtes **proches** ou **importantes** sont agrandies, les boîtes **éloignées** ou **secondaires** sont réduites. Pour simuler le fonctionnement d'une loupe et éviter les recouvrements, les boîtes autour du point de mire sont dispersées suivant une fonction de distorsion, appliquée aux coordonnées cartésiennes, ou, plus naturellement, à la distance des coordonnées polaires:

$$g(r)=\frac{(d+1)r}{dr+1}$$

d est le paramètre de distorsion, le programme est en effet contrôlé par quelques paramètres de ce type.

Le mouvement est naturel, car la position du point de mire est identique avant et après transformation.

Une **boîte** et son **contenu** peuvent être affichés ou omis, séparément, en fonction de la taille finale de la boîte. Cette taille dépend de sa position et de son importance.

La fonction de distorsion n'est appelée que pour un nombre restreint de points de chaque boîte. Ainsi, **les boîtes et les arêtes** restent respectivement **rectangulaires et linéaires** dans l'image finale. **Des arêtes, initialement disjointes, peuvent donc se croiser**. Mais les calculs à effectuer sont considérablement réduits. En pratique, le temps de calcul des boîtes est **négligeable** devant le temps d'affichage.

[SB92] rappelle aussi l'existence d'autres systèmes de visualisation de graphes:

Par exemple, *PERSPECTIVE WALL* [SB92] applique une image en deux dimensions sur trois panneaux en trois dimensions. Le panneau central montre les détails et les panneaux des cotés, fuyant vers l'infini, montrent la structure globale.

La démarche de *CONE TREE* [SB92] consiste à représenter un graphe en trois dimensions, de manière à ce que chaque nœud soit le sommet d'un cône, ses fils étant disposés en cercle sur le cône. La visualisation se fait en trois dimensions, avec des calculs d'ombre et de transparence. On se déplace "dans" le graphe avec une caméra, dont l'objectif se comporte comme une loupe.

3. Édition graphique et structurée de preuves

Il existe d'une part, des **outils d'inférence paramétrables par la logique** utilisée (partie précédente), d'autre part, des **éditeurs paramétrables par la présentation** (section précédente).

Envisageant **d'étendre la généricité de la présentation** aux outils d'inférence, nous présentons quelques interfaces d'outils d'inférence, capables de présenter les preuves graphiquement.

Une interface spécifique [Bra92]

Cet article présente un outil d'inférence avec son interface graphique. Nous l'avons choisi pour l'intérêt de la logique utilisée. Toutefois, la forme de l'interface utilisateur graphique employée est relativement classique...

En logique temporelle, la vérification de modèles consiste à démontrer une formule donnée, pour un système donné. La méthode traditionnelle parcourt l'ensemble des états du système et échoue lorsque cet ensemble est infini. Une extension de la méthode des

tableaux sémantiques, présentée par l'auteur, permet de traiter des systèmes dont le nombre d'états est infini.

Cette méthode ne permet de démontrer à la main que des formules très simples, pour des systèmes très simples. Au contraire, un démonstrateur interactif est très utile, car de nombreux calculs peuvent être automatisés.

Ainsi, [Bra92] présente un **démonstrateur interactif** avec son **interface utilisateur graphique**, respectivement écrits en ML et en C, sous X-WINDOWS, et reliés par un **pipe** UNIX.

La logique utilisée est le *μ-calcul modal*, dont les formules sont définies par (K est un sous-ensemble de l'ensemble L des étiquettes, Z est une variable):

Z	variable
$\neg\Phi$	négation
$\Phi_1 \wedge \Phi_2$	conjonction
$\Phi_1 \vee \Phi_2$	disjonction
$[K]\Phi$	nécessité
$<K>\Phi$	possibilité
$\nu Z.\Phi$	plus grand point fixe
$\mu Z.\Phi$	plus petit point fixe

Pour traiter les formules de point fixe, on utilise la notion de **définition**.

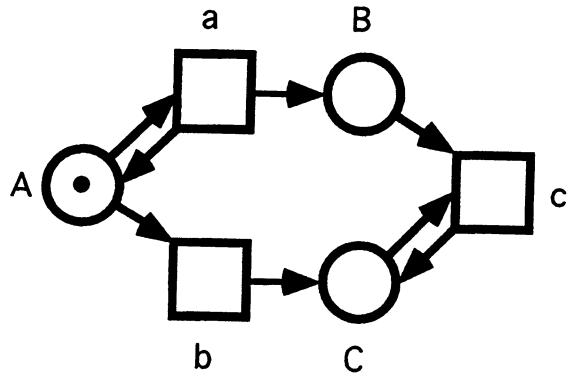
Ainsi, les nœuds des tableaux sémantiques contiennent des séquents de la forme $S \vdash_\Delta \Phi$ où S est un ensemble *d'états*, Φ est une formule du μ-calcul modal, sous forme normale positive, et Δ est une liste de définitions.

Un nœud du tableau est *valide* ssi c'est une prémisse d'une application correcte d'une règle d'inférence ou une feuille qui ferme une branche ou *termine* un motif de branche infinie, c'est à dire, qu'elle est déjà apparue plus haut dans la branche.

L'outil permet de **choisir différents systèmes**. En particulier, il contient un module pour la description de **réseaux de Pétri**. Un *réseau de Pétri* est un triplet `(S,T,F)` où `S` est un ensemble de places pouvant contenir des jetons, `T` est un ensemble de transitions pouvant ajouter ou retirer des jetons aux places et `F` est une fonction décrivant `T` formellement.

Pour appliquer la méthode aux réseaux de Pétri, les **propositions atomiques** sont les (in)égalités linéaires sur les contenus des places et les **états** sont décrits par des combinaisons booléennes de telles (in)égalités.

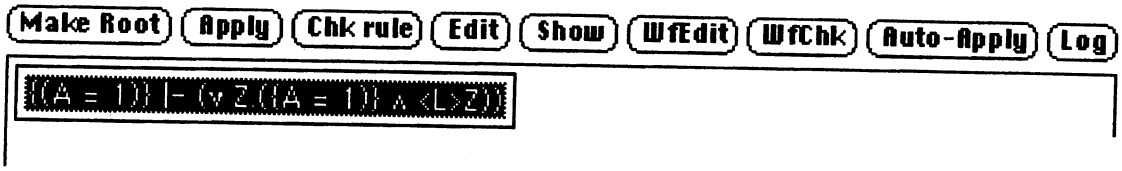
Un système est spécifié par une structure de données ML. Dans le cas des réseaux de Pétri, un simple codage de la définition suffit.



Réseau de Pétri utilisé dans l'exemple.

L'interface utilisateur est composée d'une **fenêtre graphique** contenant le tableau sémantique. L'inversion vidéo indique le nœud **sélectionné**. Un cadre estompé indique un nœud dont la **validité** n'a encore pas été reconnue.

Des **boutons** surmontent cette fenêtre:



Entrée d'une formule

Make Root permet à l'utilisateur d'entrer le séquent initial au clavier. L'analyseur syntaxique utilisé provient de ML-YACC.

Apply applique une règle d'inférence au nœud sélectionné. La règle à utiliser est entièrement déterminée par la formule. Pour la disjonction et la possibilité, elles nécessitent des précisions supplémentaires de la part de l'utilisateur.

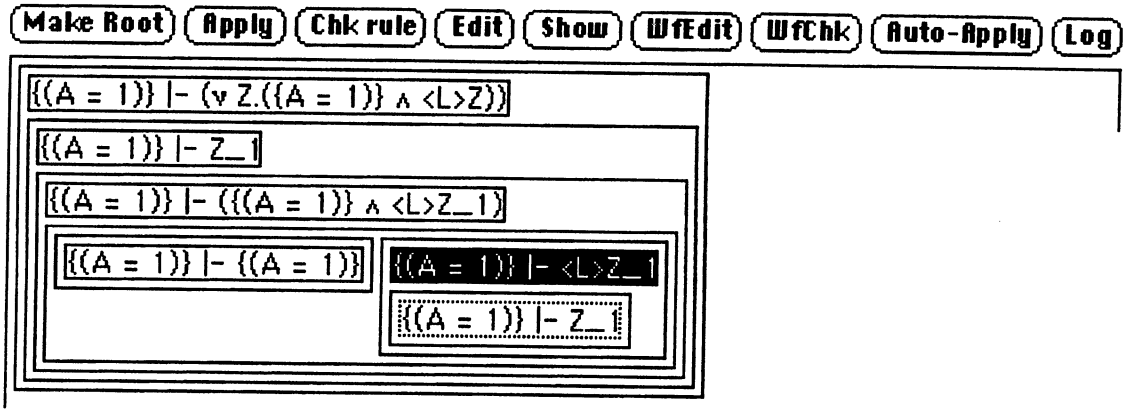
Chk rule vérifie la validité d'un nœud. Il appelle des fonctions spécifiques au système choisi.

Edit permet à l'utilisateur de modifier l'ensemble des états d'un nœud. Bien entendu, le nœud modifié et ses parents ne sont plus marqués valides.

Auto-Apply applique toutes les règles d'inférences ne nécessitant pas d'intervention de l'utilisateur.

Show montre des informations concernant un nœud.

Log montre la séquence de messages produite par l'outil pendant la session.



Etat du tableau après **Auto-Apply** et développement de la possibilité. Il reste à montrer la validité de la feuille de droite. **Chk rule** peut détecter qu'elle termine un motif de branche infinie.

Le fait qu'un point fixe soit vrai pour un état peut dépendre du fait qu'il soit vrai pour d'autres états. Toutefois, il ne doit pas exister de chaîne infinie de dépendances. Parfois, l'existence d'une relation exprimant la convergence des dépendances permet d'affirmer qu'il n'existe pas de telle chaîne.

WfEdit permet à l'utilisateur d'entrer une telle relation. Dans le cas des réseaux de pétri, c'est une combinaison booléenne d'(in)égalités entre les contenus des places aux différents états.

WfChk vérifie si les états de tous nœuds concernés vérifient la relation.

Une interface pour les démonstrateurs interactifs [TBK92]

[TBK92] présente une interface utilisateur destinée aux démonstrateurs interactifs fondés sur les notions de buts et de tactiques.

De nombreux avantages découlent de sa **réalisation indépendante** du démonstrateur interactif employé: On a pu la réaliser dans **un langage de programmation approprié**. Elle permet d'utiliser **différents démonstrateurs interactifs, réalisés dans différents langages de programmations**. On peut la lancer sur une **machine séparée**. Son **portage est indépendant** de celui des démonstrateurs interactifs.

Notamment, l'interface expérimentale, réalisée sous CENTAUR [BCDIKLP87], a été utilisée pour HOL [Gor87], ISABELLE [Pau88] et le démonstrateur interactif en λ-PROLOG [FM87].

Sous l'interface, le démonstrateur interactif employé est caché à l'utilisateur. Ce dernier ne voit constamment que deux fenêtres. Une fenêtre contient la **liste des buts** restant à développer. L'autre contient le ***script* de tactiques** représentant les actions déjà effectuées par l'utilisateur, en termes de tactiques et d'opérateurs de

tactiques. Tant qu'il reste des buts à développer, ce script est incomplet. A chaque but correspond en effet un *"trou"* du script.

A tout moment, on peut afficher une fenêtre contenant **le graphe des théories utilisées**. En cliquant sur un nœud de cette fenêtre, la **liste des théorèmes** de la théorie correspondante apparaît. L'interface permet en effet de manipuler les gestionnaires de théories des différents démonstrateurs interactifs.

L'utilisateur peut **choisir un but et une tactique** à lui appliquer, en entrant le nom et les arguments de la tactique à la place du trou qui correspond au but.

Pour aider l'utilisateur, l'interface propose un **menu des tactiques applicables** à un **trou** sélectionné dans la fenêtre script. Pour calculer ce menu, chaque tactique est associée à une fonction qui vérifie si elle peut être appliquée à un but.

De même, elle propose un **menu des réécritures applicables** à un **terme** sélectionné dans la fenêtre buts. Pour calculer ce menu, l'interface recherche les théorèmes de la forme $\forall x_1 \ldots, x_n\ A=B$ dans la hiérarchie de théories utilisée.

Quand l'utilisateur a besoin d'entrer un théorème comme argument d'une tactique, il peut cliquer sur **la liste des théorèmes** d'une théorie.

Le but de ces manipulations est de remplacer chaque trou par le code d'appel d'une tactique. En cliquant sur le bouton *Expand*, la tactique est appelée et les deux fenêtres principales sont mises à jour. En particulier, les opérateurs de tactiques adéquats sont introduits dans la fenêtre script.

L'interface propose deux fonctions d'annulation de principes différents. L'*annulation historique* consiste à revenir à un script précédent. L'*annulation locale* consiste à remplacer une partie de script par un trou.

L'intégration aux démonstrateurs interactifs, d'un protocole de communication spécifique avec l'interface, est évitée par l'**exploitation** directe de leurs **entrées et sorties standards**. Toutefois, on doit souvent modifier la fonction de sortie d'un démonstrateur interactif pour qu'elle produise des données facilement analysées par l'interface.

Du démonstrateur interactif vers l'interface, le protocole de communication est spécifié par **un ensemble d'actions** possibles. Chaque action est définie par les **marques** délimitant la communication dans la sortie standard du démonstrateur interactif, l'**analyseur** et la **commande** à employer.

Inversement, l'interface **produit des commandes et des termes analysés** par le démonstrateur automatique. Les **termes** entrés directement par l'utilisateur ne sont pas analysés par l'interface, mais transmis au **démonstrateur interactif**.

Pour présenter les buts, les scripts et les termes produits par le démonstrateur interactif, l'interface utilise les modules de présentation textuelle et graphique de CENTAUR. Notamment, la couleur est employée pour mettre les termes de même types en évidence.

En particulier, PPML [BCDIKLP87] permet de présenter les formules en spécifiant les symboles à utiliser, leur espacement et leur décalage vertical, pour chaque constructeur.

Pour une vitesse acceptable de l'interface, les algorithmes de présentation et de communication sont **incrémentaux**.

4. Discussion

Comme un document, une preuve peut être décrite par une **structure logique**, une **présentation** et un **contenu**. Un éditeur de preuves **paramétrable** par la logique et la présentation doit, comme LILAC ou GRIF, et contrairement à EDIMATH ou MENTOR-RAPPORT, permettre de décrire ces trois composantes. Comme dans GRIF, et contrairement à LILAC, il est souhaitable de **séparer** les descriptions de ces trois composantes afin, par exemple, de réutiliser les structures logiques ou de changer leur présentation.

Un **langage de définition** doit permettre de décrire la **structure logique** des systèmes formels et des preuves. Bien entendu, les langages de définition proposés par les éditeurs de documents sont trop faibles. De même, les formalismes de CENTAUR sont moins bien adaptés à la description de systèmes formels qu'à celle de langages de programmation. Bien que TYPOL suffise à spécifier certains systèmes formels [Des88], [Has88], il n'a pas été étudié dans ce but.

Employer un **cadre logique** est donc préférable. Notons que TYPOL pourrait servir de langage de programmation pour implémenter un cadre logique [Des88]. Mais il n'existe pas encore d'éditeur de preuves générique graphique, fondé sur un cadre logique. Par exemple, ALF [ACN90] est un éditeur de preuves graphique fondé sur la théorie constructive des types. Il pourrait donc servir à définir des systèmes formels, comme NUPRL [C+86]. Mais il n'est pas générique du point de vue de la présentation.

Notons que l'interface décrite dans [TBK92] ne propose ni logique ni cadre logique. La construction de preuves dépend donc du démonstrateur interactif employé et l'introduction graphique de formules par l'utilisateur est impossible. Cette interface n'est donc pas générique,

mais elle factorise les similitudes des différents démonstrateurs interactifs qu'elle fédère.

Des outils comme VTP [BCDIKLP87] ou CSG [Gri87] pourraient servir à **réaliser** le noyau d'édition pour le cadre logique. Cependant, nous désirons employer un cadre logique existant. La syntaxe abstraite des cadres logiques existants étant stables et très simple, une implémentation directe est peut être plus facile et vraisemblablement plus efficace. Notons que des attributs, au sens de CSG, pourraient traiter des **contraintes contextuelles** (le fait qu'une preuve soit complète, qu'une branche d'un tableau sémantique soit fermée, etc.) non formalisées dans le cadre logique choisi.

Un **langage de présentation** peut servir à décrire la **présentation** graphique des preuves. Vraisemblablement, il doit être fondé sur la notion de structure de boîtes.

Le langage PPML utilisé sous CENTAUR est simple et concis. L'utilisation du filtrage est particulièrement pratique, pour indiquer les occurrences de termes. Mais les dispositions possibles pour les boîtes sont relativement limitées. Elles correspondent en effet aux capacités d'un pretty-printer.

Le langage de présentation décrit dans GRIF est presque aussi simple. Même si sa syntaxe est plus lourde, la possibilité de réutiliser les présentations le rend plus concis en pratique. Mais surtout, il est plus puissant: Il permet notamment de spécifier la présentation des formules d'EDIMATH. Lui ajouter une notion de filtre ou d'occurrence permettrait de **l'adapter** facilement à la présentation de termes.

Quand une structure de boîtes ne rentre pas dans la surface d'écran allouée, on peut utiliser des **méthodes de visualisation** similaires à celles décrites dans [SB92].

Comme EDIMATH ou GRIF, un éditeur doit permettre d'éditer le **contenu** dans une vue unique, avec la présentation choisie par l'utilisateur. Il ne doit pas, comme LILAC, imposer de vue pour programmer ce contenu. Toutefois, plusieurs parties peuvent composer une vue [TBK92].

Pour la manipulation de texte (CSG) ou de structures de boîtes (GRIF), un éditeur structuré permet de **manipuler un contenu via son image** à l'écran. Soulignons la similitude entre CSG et GRIF en notant que les boîtes sont représentées, de manière interne, par des termes. Ces termes possèdent des attributs (position, dimension, contenu) dont le calcul peut être contextuel et incrémental.

L'objet du projet CENTAUR [BCDIKLP87] est de produire un atelier de programmation comprenant éditeur, interprète, et debugger à partir des spécifications formelles d'un langage de programmation. On peut faire le

rapprochement entre logique et langage de programmation, preuve et programme, correction et respect des spécifications. D'autres similitudes existent certainement. Ainsi, l'activité de démonstration peut être vue comme une activité de programmation. Dans ce sens, des travaux indépendants ont consisté à employer CENTAUR pour construire des preuves [Des88], [Has88].

Ainsi, l'éditeur de preuves pourrait être vu comme **une sorte de CENTAUR adapté aux logiques**. On pourrait donc copier **l'architecture** de CENTAUR. Son interface utilisateur graphique moderne, portable et puissante serait conservée. Le langage de définition remplacerait METAL et TYPOL. Le langage de présentation remplacerait PPML. Le noyau de traitement des termes du cadre logique remplacerait VTP.

Du point de vue de la **communication** l'idée consistant à exploiter les entrées et les sorties standard permet de réutiliser sans modification de nombreux outils d'inférence. Cette idée, décrite dans [TBK92], a été utilisée pour les premières interfaces utilisateur graphiques des outils d'inférence de ATINF [BCC88], [CH93].

VI. Réalisation

Introduction

Ce chapitre décrit la réalisation de GLEF (Graphical Logical Edition Framework), un éditeur de preuves graphique interactif et paramétrable par la logique utilisée. Permettant la communication entre différents outils d'inférence, il réalise l'intégration d'ATINF (ATelier d'INFérence).

La première partie de ce chapitre décrit les notions formelles développées pour GLEF. La deuxième partie décrit sa programmation proprement dite.

A) Réalisation formelle

Dans un environnement de programmation (LISP, CAML, λ-PROLOG), une collection de primitives pour manipuler et présenter des termes pourraient former un éditeur de preuves générique. Mais l'utilisateur devrait apprendre une partie du langage de programmation pour appeler ou paramétrer ces primitives.

Employer un **langage de spécification adapté** aux systèmes formels est préférable. Bien qu'un tel langage impose aussi une certaine forme de programmation à l'utilisateur, ce travail de programmation est minimal.

La **séparation** des spécifications de la **définition** et de la **présentation** d'un système formel est utile. Elle permet notamment d'omettre la présentation et d'utiliser une présentation par défaut, pour spécifier rapidement un système formel. Elle permet aussi de spécifier plusieurs présentations pour un même système formel. Ainsi, l'utilisateur peut employer une présentation respectant ses notations habituelles et adaptée à sa connaissance du système formel objet.

Nous considérons donc un *langage de définition* et un *langage de présentation.*

1. Un formalisme de définition

a) Objet

Fonder GLEF sur un langage de définition le rend **indépendant** de tout outil d'inférence. Il peut ainsi **manipuler** et **présenter** les preuves

de manière autonome. Toutefois, un langage de présentation permet de spécifier leur présentation.

De plus, un langage de définition facilite la **communication** entre différents outils d'inférence, en unifiant les spécifications de systèmes formels.

Associé au langage de définition, un calcul permet à GLEF de vérifier, dans une certaine mesure, les manipulations de l'utilisateur et les preuves fournies par les outils d'inférence. Ainsi, nous considérons plutôt un *formalisme de définition.*

Comme nous l'avons vu dans les chapitres précédents, un *cadre logique* est un formalisme capable de représenter une large classe de systèmes formels. Trouver un cadre logique le plus général et le plus naturel possible est un problème très difficile. LF [HHP89] et CC [CH88], issus de AUTOMATH [dBr80], et LDS [Gab91] sont des cadres logiques résultant de tels travaux.

Employer un **cadre logique existant** comme formalisme de définition paraît donc raisonnable. Nous avons choisi un formalisme proche de LF et CC, en l'adaptant à nos besoins par quelques **extensions** syntaxiques.

b) Rappels sur le Calcul des Constructions [CH88]

CC est un formalisme d'ordre supérieur, capable, à l'instar de LF, de représenter des systèmes formels dans un style de déduction naturelle, et vérifier les preuves de ces systèmes formels.

Les *constructions* sont les termes bien typés d'un λ-calcul Π-typé dont les types sont des termes de même nature. Suivant la correspondance de Curry-Howard entre propositions et types, les propositions sont représentées par des types, et les preuves sont des constructions typées par ces propositions.

c) Langage du Calcul des Constructions

Les différentes règles de formation de ce langage sont:

- Univers: $*$
- λ-abstraction: $\lambda a{:}A.b_a$
- Produit dépendant: $\Pi a{:}A.B_a$
- Application: $(M\ N)$
- Variable: x

L'*univers* est une constante qui joue le rôle de l'univers de tous les types, sans elle-même avoir de type. Si pour tout terme a de type A, il existe un terme b_a de type B_a, l'*abstraction* permet de construire la fonction notée $\lambda a{:}A.b_a$, qui associe b_a à tout a. Le *produit dépendant* permet

de construire son type $\Pi a{:}A.B_a$. Dans le formalisme de définition de GLEF, les syntaxes employées pour $\lambda a{:}A.b_a$ et $\Pi a{:}A.B_a$ sont respectivement $<a{:}A>b_a$ et $[a{:}A]B_a$[18]. Quand a n'apparaît pas dans b, $<A>b$ dénote $<a{:}A>b$, qui est une fonction constante. Quand a n'apparaît pas dans B, $[A]b$ dénote $[a{:}A]B$, ce type représente l'ensemble B^A des fonctions de A vers B. L'*application*, sert à appliquer les fonctions à des arguments. Une abstraction, un produit dépendant ou un contexte introduit chaque *variable*.

Un *contexte* est une liste de couples, notés $[x{:}M]$, associant chacun une variable à un type. Les règles de formation sont toujours appliquées dans un *contexte*, éventuellement vide.

LF et CC sont des cadres logiques fondés sur des λ-calculs typés similaires. Le calcul de LF est en fait un sous-calcul strict de CC [HHP89], [Bar92]. Toutefois, leurs approches sont différentes. En CC, les constantes définissent les connectifs logiques de manière constructive. En LF, elles définissent le langage et les règles d'inférence grâce aux notions de jugement de base et de jugements d'ordre supérieur (hypothétique et schématique).

CCind, le formalisme de Coq [D+91], permet de définir des règles d'inférence comme les constructeurs de types inductifs représentant des connectifs logiques. Cependant, pour des raisons techniques, il ne permet pas de définir directement des règles d'inférence inductives.

Nous avons choisi de représenter les systèmes formels selon le principe de LF, en employant le calcul de CC. Comme de toute façon, l'adéquation du codage de chaque système formel doit être vérifiée, ce choix est justifié. Mais employer CC permet de représenter directement des preuves constructives ou des programmes.

d) Langage de définition

Le langage de CC aurait pu servir tel quel comme langage de définition. Malgré la puissance d'expression de CC, employer son langage pour définir certaines notions qui pourraient être directement introduites au niveau de la syntaxe est parfois difficile. Pour simplifier son utilisation, nous avons donc choisi de lui ajouter des notions de *multi-ensembles* et de *types ensembles*. Ces notions permettent en particulier d'écrire des fonctions **associatives et commutatives d'arité variable, de profils multiples**, et **à arguments de types multiples**.

A moment où cette thèse est terminée, comme Coq est dans le domaine public, on pourrait peut-être tenter d'utiliser CCind pour définir

---

[18]Notons ces syntaxes ne correspondent pas à celles employées dans COQ [DFHHPW91], qui sont respectivement $[a{:}A]b_a$ et $(a{:}A)B_a$. La syntaxe du formalisme de définition de GLEF devrait donc être mise à jour.

les notions que nous avons ajoutées à CC. Toutefois, la représentation des preuves manipulées pourrait être plus complexe. Bien entendu, on pourrait manipuler cette représentation de manière automatique, mais quel serait l'impact sur l'efficacité.

Le langage de définition contient quatre catégories syntaxiques principales, les *termes élémentaires*, les *types* et les *contextes* et les *termes*. Leur définition inductive est donnée par l'ensemble de règles de production suivantes. Chaque règle de production correspond à une catégorie syntaxique élémentaire dont le ou les noms sont donnés à droite.

Terme élém.	::= *	Univers
	| [x:Type] Terme élém.	Produit dépendant
	| Type+	Type de multi-ensemble (Liste)
	| <x:Type>Terme élém.	Abstraction
	| {Terme élém...., Terme élém.}	Multi-ensemble
	| [x=Terme élém.] Terme élém.	Nommage
	| (Terme élém. Terme élém.)	Application
	| x .	Variable
Type	::= {Terme élém...., Terme élém.} .	
Contexte	::= ε	Contexte vide
	| Context [x:Type]	Introduction de constante
	| Context [x=Terme élém.] .	Nommage (Introduction de nom)
Terme	::= Terme élém. | Type .	

Avec la règle de priorité d'opérateurs:

$$[x:T]\,t+ = [x:T](t+)$$

Les symboles '{' et '}' sont employés ici (et dans la description des règles de jugement de typage qui suit) par souci de clarté. Ils sont notés '{' et '}' lors de l'utilisation du langage de définition. En effet, les types et les multi-ensembles ne peuvent pas être confondus au sein d'un terme.

Notons que le langage inclus celui de CC, plus de nouvelles catégories syntaxiques: Multi-ensemble, Type de multi-ensemble et Type. De plus, dans CC certains termes peuvent être considérés comme des types, alors que dans le langage de définition, tout type est un ensemble de types élémentaires, qui ne sont pas des types. Notons que la notion de type ensemble correspond aux **types conjonctifs** [Hue91], pas à une **relation de conséquence à plusieurs conclusions** [Avr91].

Nous utiliserons les abréviations suivantes, pour raccourcir les expressions (T est un type, t et t_i sont des termes élémentaires, x et x_i sont des noms, et Γ_1 et Γ_2 sont des contextes):

$t \equiv \{t\}$ (Dans un sous-terme)

$[x_1 \ldots, x_n : T] \equiv [x_1 : T] \ldots, [x_n : T]$ (Dans les produits dépendants ou les contextes)

$(t_1 \ \ldots \ t_n) \equiv (\ldots (t_1 \ t_2) \ \ldots \ t_n)$

$[T]t \equiv [x:T]t$ (Si x n'apparaît pas dans t)

Dans le formalisme de définition, un système formel objet peut être représenté par un contexte Γ_0. Ce contexte étant donné, tous les contextes considérés prolongent Γ_0, et tous les termes sont considérés dans de tels contextes. Lors de l'édition, le contexte Γ_0 est appelé *contexte d'édition.*

En plus des expressions précédentes, nous utiliserons aussi une expression spéciale appelée *pool*. Cette notation permet à l'utilisateur de GLEF d'introduire de nouvelles constantes de type T au contexte d'édition, pendant l'édition.

$\Gamma_1 [x\text{=pool} : T] \Gamma_2 \equiv \Gamma_1 [x:T] \Gamma_2$ Pool

En effet, pendant la construction d'une preuve, l'utilisateur peut souhaiter étendre Γ_0. Par exemple, en logique propositionnelle, il peut vouloir ajouter des propositions. Sans lui laisser la liberté de modifier complètement le contexte d'édition, on peut l'autoriser à **ajouter des entités de types donnés**. Cette autorisation est spécifiée au niveau du langage de définition, car elle concerne la manière d'employer le système formel objet.

Une *entité* est une variable introduite par Γ_0, par l'intermédiaire éventuel d'un nommage. Une *variable pure* est une variable qui n'est pas une entité. Etant donné un contexte Γ, une *variable libre* de Γ est une variable pure introduite par Γ. Etant donné un terme t considéré dans un contexte Γ, une *variable libre* de t est une variable libre de Γ[19], et une *variable liée* de t est une variable, forcément pure, introduite dans t.

Finalement, le formalisme de définition ne permettant pas de définir toutes les manipulations de λ-termes, on admet dans Γ_0 la définition ***d'entités évaluables***, distinguées par la notation $[\$x:M]$. Une entité évaluable est une entité qui, lorsqu'elle est appliquée à un certain nombre d'arguments, se transforme automatiquement en un terme de type égal à celui de l'application qu'elle remplace. Notons que **cette application n'est pas constructible, puisqu'elle est toujours évaluée**. L'évaluation de cette application est définie de manière totalement externe, par programmation. Cependant, nous avons choisi de laisser une information au niveau du formalisme de définition, en obligeant les nom des entités évaluables à commencer par le caractère $.

[19]Selon cette définition, un terme ne contient pas forcément des instances de toutes ses variables libres.

Par exemple, l'union et le parcours de multi-ensembles sont des entités évaluables très utiles.

Les sections suivantes montrent l'apport des extensions du langage de définition à sa puissance syntaxique.

Des fonctions associatives et commutatives d'arité variable

Considérons le connectif logique "et" à deux arguments, dans CC, son profil peut être défini dans le contexte:

```
[formula:*]
```

Par:

```
[and:          [formula][formula]formula].
```

Le formalisme de définition admet cette écriture comme une simplification de la définition plus complète suivante:

```
[formula:      {*}]
[and:          {[{formula}][{formula}]formula}]
```

Cependant, on peut aussi écrire une version de "et" associative et commutative d'arité variable:

```
[and:          {[{{formula}+}]formula}]
```

La simplification de cette définition donne:

```
[and:          [formula+]formula]
```

Exemple:

Dans le sous-contexte: `[P,Q,R:formula]`

On peut écrire: `(and {P,Q,R})`

Des fonctions de profils multiples

Considérons les deux règles d'inférence:

$$\frac{A \Leftrightarrow B}{A \Rightarrow B} \text{ et } \frac{A \Leftrightarrow B}{A \Leftarrow B}$$

Dans CC, elles peuvent se définir dans le contexte suivant:

```
[formula:                  *]
[equiv:                    [formula][formula]*]
[implies_right:            [formula][formula]*]
[implies_left:       [formula][formula]*]
```

Par:

```
[rule_right:        [A,B:formula][(equiv A B)]
                          (implies_right A B)]
[rule_left:         [A,B:formula][(equiv A B)]
                          (implies_left A B)]
```

Le formalisme de définition admet cette écriture comme une simplification de la définition plus complète suivante:

```
[formula:           {*}]
[equiv:             {[{formula}][{formula}]*}]
[implies_right:     {[{formula}][{formula}]*}]
[implies_left: {[{formula}][{formula}]*}]
[rule_right:        {[A,B:{formula}][{(equiv A B)}]
                          (implies_right A B)}]
[rule_left:         {[A,B:{formula}][{(equiv A B)}]
                          (implies_left A B)}]
```

Cependant, on peut aussi définir ces deux règles d'inférence comme une seule règle de profil multiple:

```
[rule:              {[A,B:{formula}][{(equiv A B)}]
                          (implies_right A B),
                    [A,B:{formula}][{(equiv A B)}]
                          (implies_left A B)}]
```

La simplification de cette définition donne:

```
[rule:              {[A,B:formula][(equiv A B)]
                          (implies_right A B),
                    [A,B:formula][(equiv A B)]
                          (implies_left A B)}]
```

Des arguments de types multiples

Considérons un opérateur `apply` appliquant son premier argument, qui est une fonction de profil multiple, à deux autres arguments de types différents. Le langage de définition permet de définir un tel opérateur:

```
[apply=        <f:{[type1]formula,[type2]formula}>
   <a:type1><b:type2>(and {(f a),(f b)})]
```

Le type de `apply` est:

```
[{[type1]formula,[type2]formula}][type1][type2]formula
```

e) Dépendance du langage de définition et de la présentation:

Dans le langage de définition certains termes peuvent être écrits de plusieurs façons.

Exemples:

`<x:term>(f x)` et `<y:term>(f y)`
`(and {P,Q})` et `(and {Q,P})`

On pourrait éviter ces ambiguïtés de notation:

- Les noms de variables pourraient être éliminés à l'aide des indices de De Bruijn [dBr72], dont le principe est rappelé dans la description de la réalisation pratique (deuxième partie).
- Les éléments des multi-ensembles et des types pourraient être triés selon un ordre total.

Mais à l'instar du λ-calcul, ces ambiguïtés rendent le formalisme de définition beaucoup plus naturel pour l'écriture ou la lecture directe des systèmes formels et des preuves.

D'autre part, elles permettent une liaison naturelle entre la spécification d'une définition et celle de sa présentation, écrite par l'utilisateur.

- L'emploi de **références** est nécessaire pour présenter les entités d'une logique. Ces références pourraient être déterminées à l'aide d'entiers similaires aux indices de De Bruijn, mais il est plus facile d'utiliser les noms des entités.
- Si l'ordre des éléments des multi-ensembles et des types n'a aucune importance pour la définition, il peut en avoir pour la présentation.

Repousser l'élimination des ambiguïtés au niveau de la **syntaxe abstraite** est donc préférable.

Ces considérations montrent ainsi les **limites du principe de séparation entre définition et présentation.**

f) Règles de jugement de typage

Après avoir décrit le langage du formalisme de définition, décrivons son calcul. Comme celui de CC, il est entièrement défini par un ensemble de règles de jugement de typage.

Les *jugements de typage* sont des jugements de base servant à décrire les types des termes de manière formelle. Les types servent à **limiter** la construction **syntaxique** d'applications. En particulier, cette limitation leur permet de représenter des **propositions** objet (principe de Curry-Howard) ou des **jugements** objets [HHP89].

Le langage de définition propose des opérateurs qui n'existent pas dans le langage de CC. Les différences de langage entre le formalisme de définition et CC ne sont donc pas uniquement lexicales, et leurs règles de jugement de typage différent. Le formalisme de définition peut **ne pas avoir toutes les propriétés formelles** de CC.

Établir ces propriétés formelles est très difficile et hors du cadre de cette thèse. Dans un premier temps, nous ne sommes concernés que par son **utilisation** comme cadre logique. Montrer la correction de la représentation de chaque exemple développé serait le plus important [HHP89]. Ensuite, on pourrait étudier la complétude de cette représentation et la décidabilité du calcul, etc.. En attendant, les différents exemples développés donnent un aperçu de sa puissance d'expression.

Les règles de jugement de typage suivantes satisfont donc nos besoins. Notons qu'en éliminant les multi-ensembles, les types de multi-ensembles, les nommages et les nommages partiels, en n'autorisant que des types singletons et en utilisant les abréviations, on retrouve et langage et les règles de jugement de typage de CC [Hue91]. Toutes les classes de termes (catégories syntaxiques objets) pouvant être définies avec CC peuvent donc l'être avec le formalisme de définition.

Dans ces règles:

Le jugement de typage s'écrit sous la forme $\Gamma \vdash t:T$, qui signifie "Γ est un contexte valide, et le terme élémentaire t est du type T dans Γ". (T, T_i sont des types, t et t_i sont des termes élémentaires et Γ est un contexte):

Univers:
$$\varepsilon \vdash *:\{\}$$

Constante:
$$\frac{\Gamma \vdash t_1:T_1 \ \ldots \ \Gamma \vdash t_n:T_n}{\Gamma[x:\{t_1\ldots,t_n\}] \vdash *:\{\}}$$

Nommage:
$$\frac{\Gamma \vdash t:T}{\Gamma[x=t] \vdash *:\{\}}$$

Variable:
$$\frac{\Gamma \vdash *:\{\} \ \mathrm{get}(\Gamma,x)=[x:T]}{\Gamma \vdash x:T}$$
$$\frac{\Gamma \vdash t:T \ \mathrm{get}(\Gamma,x)=[x=t]}{\Gamma \vdash x:T}$$

Produit dépendant:
$$\frac{\Gamma[x:T_1] \vdash t:T_2}{\Gamma \vdash [x:T_1]t:\{*\}}$$

Type de multi-ensemble:
$$\frac{\Gamma \vdash *:\{\} \ \Gamma \vdash t_1:T_1 \ \ldots \ \Gamma \vdash t_n:T_n}{\Gamma \vdash \{t_1\ldots,t_n\}+:\{*\}} \ (n\geq 0)$$

Abstraction:
$$\frac{\Gamma[x:T] \vdash t:\{t_1\ldots,t_n\}}{\Gamma \vdash <x:T>t:\{[x:T]t_1\ldots,[x:T]t_n\}} \ (n\geq 0)$$

Multi-ensemble:
$$\frac{\Gamma \vdash *:\{\}\ \Gamma \vdash t_1:T_1\ \ldots\ \Gamma \vdash t_n:T_n}{\Gamma \vdash \{t_1\ldots,t_n\}:\{\text{type\_union}(T_1\ldots,T_n)+\}}\ (n\geq 0)$$

Application:
$$\frac{\Gamma \vdash t_1:T_1\ \Gamma \vdash t_2:T_2\ \text{type\_apply}(T_1,T_2)\neq\{\}}{\Gamma \vdash (t_1\ t_2):\text{type\_apply}(T_1,T_2)}$$

Où, avec les notations ensemblistes usuelles pour les types:

$\text{type\_union}(T_1\ldots,T_n) = \bigcup_1^n T_i$
$\text{type\_apply}(T_1,T_2) = \{t_3 : \exists t_1\in T_1\ t_1=[x:T_3]\,t_3 \wedge \text{tcp}(T_3,T_2)\}$
$\text{tcp}(T_1,T_2) = (\forall t_1\in T_1\ \exists t_2\in T_2\ \text{to}(t_1,t_2))$
$\text{tcl}(T_1,T_2) = (\forall t_2\in T_2\ \exists t_1\in T_1\ \text{to}(t_1,t_2))$
$\text{to}(t_1,t_2) = (t_1=t_2$
$\quad \vee\ t_1=[x:T_1]\,t'_1 \wedge t_2=[x:T_2]\,t'_2 \wedge \text{to}(t'_1,t'_2) \wedge \text{tcp}(T_2,T_1)$
$\quad \vee\ t_1=T_1+ \wedge t_2=T_2+ \wedge \text{tcl}(T_1,T_2)$

$\text{tcp}(T_1,T_2)$ signifie qu'une constante dont le type est de la forme $\{[x:T_1]\,t,\ldots\}$ peut être appliquée à un argument de type T_2. $\text{tcl}(T_1,T_2)$ signifie qu'un multi-ensemble dont le type est de la forme $\{T_1+,\ldots\}$ peut contenir un élément de type T_2. to sert à la définition récursive mutuelle de tcp et tcl.

get est défini récursivement par:

$\text{get}(\Gamma[y:T],x) = \text{if } x=y \text{ then } [x:T] \text{ else } \text{get}(\Gamma,x)$
$\text{get}(\Gamma[y=t],x) = \text{if } x=y \text{ then } [x=t] \text{ else } \text{get}(\Gamma,x)$
$\text{get}(\varepsilon,x) = \varepsilon$

Remarques et exemples:

- Pour que le formalisme ait de "bonnes" propriétés formelles, il faudrait peut-être restreindre la règle des constantes à $T_i\in\{\{\},\{*\}\}$ et la règle du produit dépendant à $T_2\in\{\{\},\{*\}\}$. En effet, ces restrictions correspondent à limiter la règle de formation du produit aux règles $\{(*,*,*),(*,\square,\square),(\square,*,*),(\square,\square,\square)\}$ dans un PTS [Bar92].
- Quand t est une abstraction ou un multi-ensemble, les jugements de la forme suivante ne sont pas déductibles:

$$\Gamma \vdash [x:T_1]\,t:T_2$$

$$\Gamma \vdash \{t\}+:T$$

- Les règles suivantes donnent les types de {} et des multi-ensemble contenant {}.

$$\frac{\Gamma\vdash *:\{\}}{\Gamma\vdash \{\}:\{\{\}+\}}$$

$$\frac{\Gamma \vdash \{C_1\ldots,C_n\}:\{\{t_1\ldots,t_p\}+\}}{\Gamma \vdash \{C_1\ldots,C_n,\{\}\}:\{\{t_1\ldots,t_p,\{\}+\}+\}}\ (n \geq p \geq 0)$$

- Arguments et fonctions de types multiples:
  Soient a de type T_a et f de type $\{[T_f^1]t_1\ldots,[T_f^n]t_n\}(n \geq 1)$. S'il existe i tel que $T_f^i \subset T_a$ alors f peut être appliquée à a.
- Comparaison de fonctions, de fonctions de fonctions, etc. à arguments de types multiples. Notons "l'inversion" du sens de l'application de tcp à chaque niveau:

  Si
  alors
  $$\begin{array}{c} \mathrm{tcp}(T_f,T_a) \\ \mathrm{tcp}(\{[T_a]t_1\},\{[T_f]t_1\}) \\ \mathrm{tcp}(\{[[T_f]t_1]t_2\},\{[[T_a]t_1]t_2\}) \\ \ldots \end{array}$$
  (par exemple $T_f \subset T_a$)

  Par exemple, on a:
  $$\mathrm{tcp}\left(\left\{\left[\left\{\left[T_1^1\right]t_1,\left[T_1^2\right]t_1\right\}\right]t_2,\left[\left\{\left[T_2^1\right]t_1,\left[T_2^2\right]t_1\right\}\right]t_2\right\},\left\{\left[\left[T_1^i \cup T_2^i\right]t_1\right]t_2\right\}\right)(i=1 \vee i=2)$$
  Donc, si a est de type $\left\{\left[\left[T_1^1 \cup T_2^1\right]t_1\right]t_2\right\}$ ou $\left\{\left[\left[T_1^2 \cup T_2^2\right]t_1\right]t_2\right\}$, et f est de type $\left[\left[\left\{\left[T_1^1\right]t_1,\left[T_1^2\right]t_1\right\}\right]t_2,\left[\left\{\left[T_2^1\right]t_1,\left[T_2^2\right]t_1\right\}\right]t_2\right]t_3$ alors f peut être appliquée à a.
  Bien entendu, $\{[\{t_1,t_2\}]t_3\} \neq \{[\{t_1\}]t_3,[\{t_2\}]t_3\}$.
- Comparaison de types de multi-ensembles, de types de types de listes, etc. d'éléments de types multiples:

  Si
  alors
  $$\begin{array}{c} \mathrm{tcl}(T_f,T_a) \\ \mathrm{tcl}(\{T_f+\},\{T_a+\}) \\ \mathrm{tcl}(\{\{T_f+\}+\},\{\{T_a+\}+\}) \\ \ldots \end{array}$$
  (par exemple $T_f \supset T_a$)

  Par exemple:
  Si a est de type $\{A\}$, et f est de type $\{[\{\{A,B\}+,C\}+]D\}$ alors f peut être appliquée à $\{\{a\}\}$.
  Bien entendu, $\{\{t_1,t_2\}+\} \neq \{\{t_1\}+,\{t_2\}+\}$.

Place du formalisme de définition par rapport à CCind

Avec la récente disponibilité de Coq [D+91] (voir chapitre État de l'Art), il est naturel de se demander si son formalisme CCind (Calcul des Constructions inductives), ne permettrait pas de **simuler** le FD (Formalisme de Définition).

C'est en effet possible, les multi-ensembles peuvent être représentés par un type inductif semblable à celui des listes polymorphes. L'union et l'égalité de multi-ensembles correspondent alors respectivement à la concaténation et à un prédicat inductif, à

définir sur ce type inductif. Toutefois, notons que ces opérations augmentent la complexité des preuves, car chacune de leur utilisations doit être prouvée de façon explicite. Concernant le filtrage et l'unification, bien qu'il soit possible de tenter de les programmer dans CCind, ces programmes ne seraient certainement pas assez efficaces pour être exploités de manière intensive. On peut donc les introduire comme des règles de transformation externes, comme pour le FD (voir partie B).

Quant aux types ensembles, ils sont proches des types inductifs sommes. La différence est que dans le FD, la fonction `type_apply` permet de simplifier leur utilisation. En effet, pour représenter certaines applications du FD par des applications de CCind, il faut transformer leurs arguments par des injections, des fonctions primitives récursives, des abstractions et d'autres applications. Par exemple, le contexte du FD suivant:

```
[f:[{A,B,C}]D][g:[[B]D]E][a:{A,C}]
```

Peut être représenté par contexte de CCind:

```
[f:[A+(B+C)]D][g:[[B]D]E][a:A+C]
```

Dans ces contextes respectifs, les deux applications du FD: `(f a)` et `(g f)` peuvent être représentées par les deux applications de CCind, qui sont plus complexes:

```
(f (match a with <x:A>(inl A B+C x) <x:C>(inr A B+C (inr B C x))))

(g <x:B> (f (inr A B+C (inl B C x))))
```

Ainsi, les représentations dans CCind de preuves qui emploient des multi-ensembles ou des types ensembles sont syntaxiquement plus complexes que leurs représentations dans le FD. Bien entendu, cette complexité additionnelle pourrait être traitée de manière entièrement automatique. Cependant, on peut se demander si les réalisations employant CCind pour simuler le FD, à la place du FD, seraient toujours aussi **efficaces**.

g) Possibilités du formalisme de définition

Le formalisme de définition hérite ses principales possibilités de LF et CC.

Définition d'un système formel et de preuves

Dans le formalisme de définition, on emploie le principe de LF (représenter les jugements par des types [HHP89]) qui est plus général

que le principe de Curry-Howard (représenter les propositions par des types). Nous supposons que ce choix ne pose pas de problèmes.

Un **système formel est représenté par un contexte** du formalisme de définition. Ce contexte définit les entités constituant le langage et le calcul du système formel.

Il commence le plus souvent par les catégories syntaxiques (l'exemple considéré est un système de déduction naturelle pour la logique du premier ordre):

```
[term,formula,judgement:*]
```

Viennent ensuite les connectifs logiques (contrairement au principe de Curry-Howard, on impose pas la logique minimale):

```
[and,or,implies,equiv:
                [formula][formula]formula]
[not:           [formula]formula]
[exists,forall:
                [[term]formula]formula]
```

Les formes de jugement de base (le principe de ELF permet en effet de définir plusieurs formes de jugements, afin de représenter plusieurs relations de conséquence, ce qui permet des représentation relativement simples de certaines logiques modales [HHP89]):

```
[true:          [formula]judgement]
```

Et finalement, les schémas d'axiomes et les règles d'inférence:

$$\text{imp\_i:}\ \frac{\begin{matrix}(A)\\ B\end{matrix}}{A \Rightarrow B} \qquad \text{all\_i:}\ \frac{A[x]}{\forall x\, A[x]}$$

```
[imp_i:  [A,B:formula]
              [[(true A)](true B)]
                   (true (implies A B))]
[all_i:  [A:[term]formula]
              [[x:term](true (A x))]
                   (true (forall A))]
...
```

Une **preuve (connexe) est représentée par un terme** dont le type est le jugement de base démontré, dans le contexte qui définit le système formel employé ou dans un de ses prolongements. On admet en effet de prolonger ce contexte par la définition d'entités **supplémentaires** (fonctions, constantes, prédicats), ou des nommages de termes. Ces nommages servent notamment à **mémoriser** les preuves (connexes) construites, afin de les **réutiliser**. On peut aussi le prolonger par une liste de termes, afin de sauvegarder les fenêtres

d'édition contenant une preuve (connexe ou non) (voir Environnements de Définition section B.e).

```
[f,g: [term]formula]
[P:          [formula]

[p =  (imp_i ...)]
```

Notons que les **multi-ensembles** du FD permettent de représenter des logiques linéaires ou de pertinence, des relations de conséquence multi-conclusions (par exemple avec des calcul de séquents multi-conclusions), ou les branches des tableaux sémantiques (voir chapitre Réalisation). Les **types-ensembles** servent quant à eux à surcharger les constantes, par exemple, pour représenter les deux règles d'introduction de la disjonction dans un système de Gentzen, par une seule constante.

Entités syntaxiques des systèmes formels

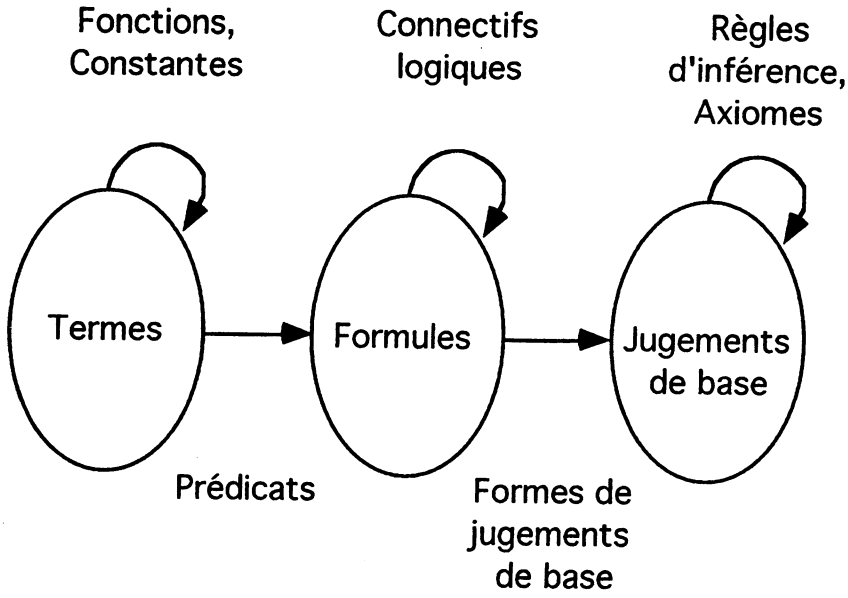
Un système formel est une *structure syntaxique* dont les *catégories syntaxiques* peuvent être définies inductivement à l'aide de constructeurs ou de règles schématiques. Dans l'exemple précédent, nous l'avons comment le formalisme de définition permet, à l'instar du Calcul des Constructions, de représenter et de simuler ces différentes *entités syntaxiques* (catégories syntaxiques, constructeurs et règles schématiques) de manière homogène. En effet, du point de vue du formalisme de définition, il n'y a pas de différence entre les représentations des différentes entités syntaxiques, **seul le calcul de type permet de déterminer le langage et les preuves admissibles**. Notons que les règles schématiques sont représentées par des constructeurs qui admettent leurs variables et des preuves de leurs prémisses pour arguments. La plupart des systèmes formels étant des structures syntaxiques similaires au système formel présenté, cet exemple peut guider la définition d'autres systèmes formels.

En considérant les représentations des entités syntaxiques dans le formalisme de définition, et en omettant les variables schématiques, on voit que la plupart des entités syntaxiques d'un système formel permettent de construire un terme d'une catégorie syntaxique à partir d'un certain nombre, éventuellement nul, de termes d'une catégorie syntaxique unique.

L'omission des variables schématiques est naturelle, en effet, on a tendance à penser qu'une règle d'inférence permet de construire une preuve d'une instance de sa conclusion, en fonction de preuves d'instances de ses prémisses, en omettant, sauf lorsque c'est strictement nécessaire, de préciser les valeurs prises par les variables. Rappelons que [CH88] donne une méthode générale pour déterminer

automatiquement les arguments qui peuvent être omis dans une application, cette méthode s'adapte aussi à notre formalisme de définition.

Dans l'exemple précédent, les entités syntaxiques dont la catégorie syntaxique de départ est égale à celle d'arrivée sont les fonctions et les constantes, les connectifs logiques, les règles d'inférences et les schémas d'axiomes. Celles dont la catégorie syntaxique de départ est différente de celle d'arrivée sont les prédicats et les formes de jugement de base.



Les "bulles" représentent les différentes catégories syntaxiques d'un système formel pour une logique du premier ordre, les flèches indiquent les catégories syntaxiques de départ et d'arrivée des constructeurs et des règles schématiques.

Intérêt du nommage

Le nommage permet d'abord de **hiérarchiser** les définitions, pour les rendre plus claires. En effet, un nommage sert à **séparer la définition des utilisations** d'un terme. En particulier, les utilisations d'un schéma d'application sont ses instances.

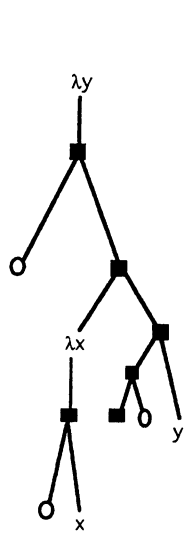
Exemple:

Avant nommage:

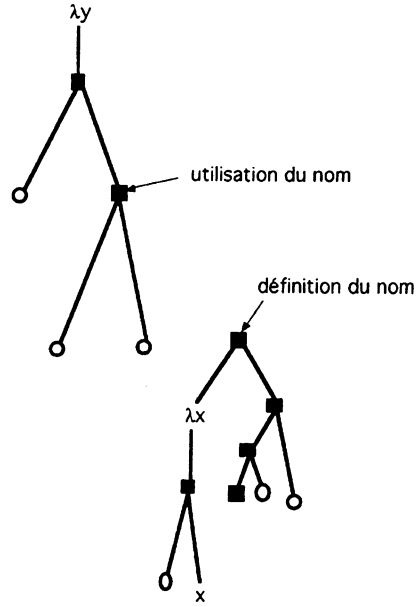
```
...(A ⇒ A)...
```

Après nommage:

```
...[F = <x:formula>(x ⇒ x)]...        (définition)
...(F A)...                           (utilisation)
```



Exemple de terme entièrement développé



A l'aide d'un nommage, ce terme peut être hiérarchisé

■ - Entité (axiome, règle d'inférence, connectif logique, constante, fonction, prédicat, etc.).
O - Variable libre.
x - Variable liée.
λx - Symbole de liaison.

Le **choix** des termes nommés est très important. Si le nommage de certains termes simplifie une définition, le nommage d'autres termes peut, au contraire, la compliquer. De plus, choisir un par un les termes à nommer n'est pas toujours possible.

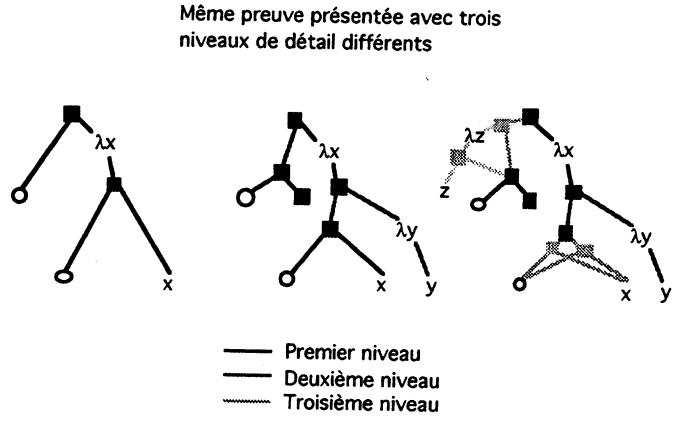
- Le nommage permet de **séparer les démonstrations** des utilisations des **lemmes** mis en évidence dans une preuve.
- Il permet aussi de **factoriser les symétries** "intéressantes" d'une définition. Deux termes sont *symétriques* ssi ils sont instances d'un même terme, qui n'est pas une variable pure. Deux termes sont en effet toujours instances d'un même terme: **ils sont instances d'une variable libre dont le type est l'union de leurs types**. Une *symétrie* est un ensemble de termes deux à deux symétriques. La plupart des définitions comportent un grand nombre de symétries sans intérêt, par exemple, les instances d'une même entité.

  La structure des termes de certaines classes n'est pas significative. Par exemple, on peut souvent transformer une preuve correcte en modifiant la structure de ses pas d'inférence. Par contre, on peut rarement changer la structure de ses formules. Une notion de symétrie plus fine peut prendre cette remarque en compte. Deux termes sont *symétriques* ssi ils sont instances de deux termes de même types, qui ne sont pas des variables pures, et la preuve reste correcte si on remplace un terme factorisant par l'autre. Le nommage permet de **factoriser ces symétries fines**.

Notons que le formalisme de définition peut servir à vérifier a posteriori la correction de la preuve modifiée.

- Finalement, le nommage permet de **regrouper** les pas d'inférence en **pas d'inférence de taille arbitraire.** Quand une preuve formelle contient de très nombreux pas d'inférence, même très simples, l'utilisateur ne peut pas les visualiser tous. En plus du critère de taille, des critères similaires à ceux employés pour introduire les lemmes et factoriser les symétries peuvent **améliorer le choix** de ces regroupements.

Ainsi, des programmes de recherche de lemmes [Lin88] et de symétries (voir travail futur chapitre Travail futur et conclusion section A.13) pourraient hiérarchiser automatiquement les définitions.



La présentation des preuves profite de cette hiérarchie. L'utilisateur peut visualiser séparément un terme nommé et les définitions contenant ses instances. En particulier, une preuve étant introduite via un nommage, l'utilisateur peut la visualiser séparément des preuves qui la réutilisent.

En plus de la hiérarchisation des définitions, le nommage permet de **simplifier certaines démonstrations** automatiques de théorèmes. Par exemple, un algorithme peut nommer les sous-formules d'une formule de la logique du premier ordre, pour minimiser le nombre et la taille des clauses obtenues lors de sa mise sous forme clausale et faciliter ainsi sa réfutation automatique par résolution [Boy92].

h) Limites à la vérification des définitions

Dans le formalisme de définition, la vérification des preuves, des systèmes formels, ou plus généralement des définitions se fait grâce au **calcul de types**. Ce calcul est défini par les règles de jugement de typage.

Un système formel et ses preuves ne sont donc vérifiés que dans la mesure où le langage et le calcul sont entièrement définis dans le

formalisme de définition. S'ils sont définis partiellement, la vérification n'est que partielle.

La vérification suppose la correction de la définition et des entités évaluables. Ces problèmes ne sont évidemment pas présentés dans les systèmes dont le but est la formalisation des mathématiques. Une description claire des solutions possibles, dans les limites théoriques imposées par les résultats de Gödel, est donnée dans [CKB85]

2. Un langage de boîtes

Objet

Si le langage de définition suffit à donner à GLEF une certaine "connaissance" de la logique et des preuves, il ne peut pas servir de langage de communication avec l'utilisateur, car il n'est pas adapté à l'être humain.

Or la première fonction de GLEF est de présenter des preuves dans différentes logiques. En mathématiques l'usage est de les présenter de manière essentiellement graphique. On emploie de nombreux symboles disposés de manières non linéaires, variant suivant les logiques ou les théories utilisées.

Pour cette présentation, nous avons choisi d'employer un langage graphique très général. C'est un langage de boîtes emprunté aux formatteurs et éditeurs de documents tels que TEX et GRIF [Qui87]. En ce qui nous concerne, les documents considérés sont des preuves.

Ce langage permet de présenter la plupart des logiques en respectant l'usage courant. Bien évidemment, il peut présenter toutes les logiques textuelles.

Employer un langage de communication simultanément graphique et très général différencie GLEF des éditeurs de preuves existants.

Notion de boîte

Structure de boîtes d'une formule.

La plupart des documents sont consitués d'éléments graphiques élémentaires: caractères, symboles. Ces éléments graphiques élémentaires sont regroupés en éléments graphiques composés: phrases, formules, et ainsi de suite, jusqu'au document complet.

La *boîte* de chaque élément graphique est le rectangle **invisible** circonscrit dont les côtés sont horizontaux ou verticaux. L'ensemble des boîtes d'un document forme donc une **structure arborescente**, dont des feuilles sont les boîtes des éléments graphiques élémentaires et les nœuds non-terminaux sont les boîtes des éléments graphiques composés.

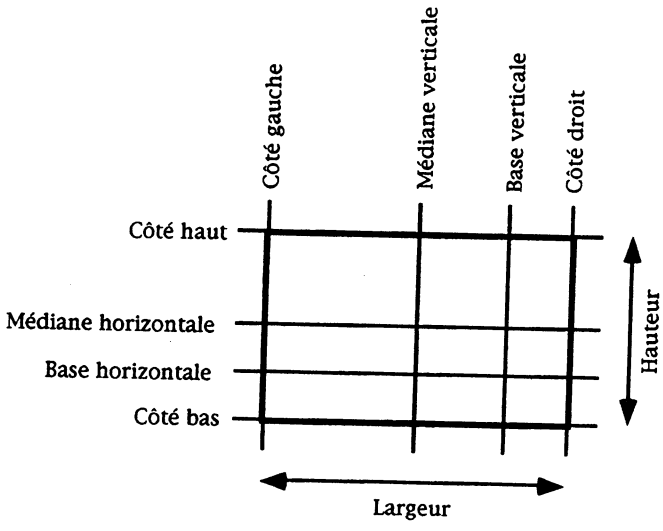
Principe

Chaque boîte détermine l'emplacement occupé par l'élément graphique qui lui correspond. Ainsi, positionner et dimensionner chaque boîte feuille suffit à déterminer la disposition de tout le document.

Toutefois, positionner et dimensionner chaque boîte de la structure arborescente, par rapport à ses boîtes environnantes, est souvent plus naturel et plus pratique.

Formalisation

Dans GLEF, chaque boîte est représentée par dix *champs*: quatre côtés, quatre axes et deux dimensions. On lui associe un certain nombre de relations qui lient ces champs à ceux des boîtes environnantes.



Les boîtes environnantes considérées sont:

- Sa boîte mère.
- Sa boîte sœur suivante ou précédente. On suppose les boîtes sœurs **ordonnées** dans leur boîte mère.
- La *boîte fictive* (dans le sens où elle n'est pas mémorisée), unique, circonscrite à l'ensemble de ses boîtes filles.

Les relations considérées sont des **équations du premier degré** entre ses champs et ceux des boîtes environnantes. Les relations implicites liant les champs d'une même boîte s'ajoutent à ces équations. Ces relations implicites sont les combinaisons linéaires des quatre équations du premier degré suivantes:

```
hauteur = côté bas - côté haut
largeur = côté droit - côté gauche
2 * médiane horizontale = côté bas + côté haut
2 * médiane verticale = côté droit + côté gauche
```

Notons que l'utilisation de boîtes fictives revient à considérer des équations sur les **maximums** et les **minimums** des coordonnées des côtés des boîtes filles concernées.

Les notions de boîte, de champs et de boîtes environnantes ont été empruntées à GRIF [Qui87]. Toutefois, les relations autorisées sont plus générales que celles de GRIF.

3. Un langage de présentation

Pour présenter une définition, GLEF doit **traduire** celle-ci du langage de définition en langage de boîtes. Pour profiter de la richesse du langage de boîtes et satisfaire l'utilisateur, cette traduction ne peut pas être totalement indépendante de la définition.

Une *présentation* spécifie donc la transformation de termes en structures de boîtes. Un *langage de présentation* permet de décrire une présentation pour chaque définition, de manière séparée.

La syntaxe du langage de présentation étant plus complexe que celle du langage de définition, sa description doit être plus formelle. La grammaire complète du langage de présentation est donnée en Annexe B.

Les instructions du langage de présentation sont classées en trois catégories: Les instructions de parcours de termes, les instructions de construction de boîtes et les instructions de structuration.

Les instructions de parcours de termes servent à parcourir les termes du formalisme de définition, pour examiner leur structure et orienter la construction des boîtes. A chaque instruction, on considère le *terme courant* parcouru.

Une *occurrence* est une liste de codes indiquant le chemin allant de la racine d'un terme à l'un de ses sous-termes. Une *occurrence généralisée* est une liste similaire, autorisant en plus, le passage d'un terme à son **type** et d'une application à son $n^{\text{ième}}$ **argument**.

La notion d'*occurrence généralisée* permet de changer provisoirement le terme courant. Par exemple, le code suivant exécute les instructions `<rules>`, en considérant le terme à l'occurrence généralisée `<occ>` du terme courant, comme nouveau terme courant:

```
SELECT <occ>:BEGIN
   <rules>
END;
```

Les instructions de construction de boîtes permettent de construire des boîtes en précisant leurs contenus, positions et dimensions. A chaque instruction, on considère la **boîte courante**, en cours de construction.

On peut créer une boîte fille en changeant provisoirement la boîte courante. Le code suivant crée une boîte fille et exécute les instructions `<rules>` en considérant cette boîte fille comme nouvelle boîte courante:

```
CREATE:BEGIN
   <rules>
END
```

On peut ajouter des équations à la boîte courante. Le code suivant positionne le côté supérieur de la boîte courante par rapport à sa boîte englobante:

```
TOP = ENCLOSING.TOP + 5;
```

On peut donner des attributs graphiques à la boîte courante. Le code suivant permet d'encadrer la boîte courante:

```
GRAPHICS:'R';
```

On peut donner des attributs typographiques à toutes les boîtes suivant la boîte courante. Le code suivant met tous les textes en caractères italiques de taille 9:

```
STYLE:'TiS9';
```

Enfin, on peut créer des boîtes feuilles. Les boîtes feuilles peuvent contenir des images, des symboles, du texte, ou la *présentation externe* d'un terme. La présentation de termes peut en effet être programmée de façon externe, quand le langage de présentation ne suffit pas. Le code suivant crée une boîte feuille contenant le symbole `<name>`. Les instructions `<rules>` permettent de positionner la boîte, ou de lui ajouter des attributs graphiques ou typographiques:

```
SYMBOL '<name>':BEGIN
   <rules>
END;
```

Des instructions associant un nom, un nom de police et un caractère servent à déclarer les symboles. Le choix de polices Postscript pour déclarer les symboles assure la portabilité vers la plupart des ordinateurs (Macintosh, PC, SUN, NEXT, DEC, etc.). Même certains formatteurs TEX emploient ces polices à la place des polices TEX originales. Le code suivant permet d'associer le caractère `<char>` de la police `<font>` au symbole `<name>`.

```
ASSIGN <name> <font> <char>;
```

Finalement, les instructions de structuration servent à décrire la présentation de chaque entité de manière **structurée** et **générique**, car l'utilisateur ne veut et ne peut pas décrire complètement chacun des termes utilisés.

Ces instructions sont fondées sur la notion de parure. Une *parure* est un bloc d'instructions associé à un nom. Le code suivant donne le nom `<name>` aux instructions `<rules>`:

```
<name>:BEGIN
   <rules>
END;
```

Chaque constructeur du formalisme de définition (univers, variable, produit, abstraction, application, liste, multi-ensemble, type) possède une **parure par défaut**. Pour spécifier la présentation des **applications des entités** définies dans le formalisme de définition, l'utilisateur peut aussi écrire des parures portant leurs noms.

Ces parures définissent une présentation générique. Ainsi, le code suivant construit la structure de boîtes correspondant à la présentation générique du terme à l'occurrence généralisée `<occ>` du terme courant. Les instructions `<rules>` permettent de positionner la boîte, ou de lui ajouter des attributs graphiques ou typographiques:

```
PLACE <occ>:BEGIN
   <rules>
END;
```

La notion de parure sert aussi à **factoriser** l'écriture de code, on peut écrire des parures dont les noms ne sont ni ceux des parures par défaut ni ceux des entités introduites par l'environnement de définition. Une instruction permet l'utilisation du bloc d'instructions de la parure:

```
PRESENT:box;
```

Quand plusieurs parures ont le même nom, seule la dernière parure est prise en compte. Cela permet par exemple de redéfinir les parures par défaut, ou de préciser une présentation particulière pour une entité d'une logique dans une théorie particulière.

L'interprétation d'une parure peut produire des erreurs, quand, par exemple, le terme courant n'est pas de la forme attendue. Dans la description de l'algorithme de présentation, nous verrons comment l'utilisation des parures par défaut, supposées correctes, permet **d'éliminer toutes les erreurs**.

Pour comparer le langage de présentation aux langages de programmation, notons que l'instruction *PRESENT* correspond à un **appel de type procédural** et que l'instruction *PLACE* correspond plutôt à un **appel dirigé par les données**.

Dans PPML [BCDIKLP87] l'emploi de filtres remplace le parcours de termes. Cette méthode semble plus naturelle que la méthode des occurrences. Nous envisageons donc d'ajouter la notion de filtre au langage de présentation (voir chapitre Travail futur et Conclusion section A.5). Notons que les filtres correspondent aux procédures qui admettent des arguments, dans les langages de programmation.

L'exemple suivant met en évidence l'intérêt des parures pour la **factorisation** et la **généricité** du code. Soit `(forall <x:term> (or (P x) (Q x)))` le terme du langage de définition à présenter sous la forme $(\forall x\,(P(x) \vee Q(x)))$. Sa présentation pourrait être spécifiée complètement, **sans utiliser de parures auxiliaires**. Vu la richesse du langage de boîtes, le code correspondant est **très long**. En pratique, un tel code n'est jamais employé, mais il peut aider à comprendre le fonctionnement élémentaire du langage de présentation (emploi des occurrences généralisées, positionnement et dimensionnement relatif des boîtes). Il est donné en Annexe C.

En général, les parures servent à **factoriser** les suites d'instructions qui apparaissent fréquemment dans une présentation:

```
box:BEGIN
    WIDTH=ENCLOSED.WIDTH;
    HEIGHT=ENCLOSED.HEIGHT;
END;

inbox:BEGIN
    LEFT=ENCLOSING.LEFT;
    TOP=ENCLOSING.TOP;
END;

line_t:BEGIN
    LEFT=PREVIOUS.RIGHT;
    TOP=PREVIOUS.TOP;
END;

line_t_space:BEGIN
    LEFT=PREVIOUS.RIGHT+5;
    TOP=PREVIOUS.TOP;
END;
```

Elles servent aussi à spécifier la présentation des entités du formalisme définition dont elles portent le nom. La séparation de ces parures permet leur **réutilisation** et la **structuration** des présentations. Par exemple, donnons une parure commune pour les quantificateurs universel et existentiel:

```
exists,forall:BEGIN
    PRESENT:box;
    TEXT '(':BEGIN
          PRESENT:inbox;
    END;
    SYMBOL NAME:BEGIN
          PRESENT:line_t;
    END;
    PLACE A.1.REFERENCE:BEGIN
          PRESENT:line_t;
    END;
    PLACE A.1.BLOC:BEGIN
```

```
            PRESENT:line_t_space;
        END;
        TEXT ')':BEGIN
            PRESENT:line_t;
        END;
    END;
```

Quand la notion de filtre sera intégrée à GLEF, la syntaxe pourra être la suivante:

```
(exists <x:term>b),(forall <x:term>b):BEGIN
    PRESENT:box;
    TEXT '(':BEGIN
        PRESENT:inbox;
    END;
    SYMBOL NAME:BEGIN
        PRESENT:line_t;
    END;
    PLACE x:BEGIN
        PRESENT:line_t;
    END;
    PLACE b:BEGIN
        PRESENT:line_t_space;
    END;
    TEXT ')':BEGIN
        PRESENT:line_t;
    END;
END;
```

Finalement, on voit que l'utilisation de parures rend les spécifications **concises**, **claires** et **génériques**.

4. Conclusion

Nous avons décrit les différents langages développés pour réaliser GLEF. Bien entendu ces langages ne sont pas figés. Le formalisme de définition pourrait être remplacé par un cadre logique mieux adapté. Les langages de boîtes et de présentation pourraient être remplacés par des langages plus naturels ou plus puissants.

La partie suivante décrit l'implémentation et la mise en œuvre de ces langages au sein de GLEF.

B) Réalisation pratique

1. Choix des outils de développement

Nous avions besoin d'outils favorisant la réalisation d'une application:

- Capable d'analyser et d'interpréter des langages.
- Communiquant avec d'autres applications.

- Proposant une *IUG* (Interface Utilisateur Graphique) moderne (menus, dialogues, etc.).
- Composant et affichant des vues graphiques.
- Rapide.
- Gérant sa mémoire de manière précise (voir section B.3.b).
- Pouvant être distribuée à un large public.

Justifier la démarche qui, en dehors de nos goûts personnels, nous a conduits à programmer en C++ sur Macintosh, peut être utile. Examinons donc quelques outils susceptibles de satisfaire nos besoins.

Les environnements de haut niveau

Voyons d'abord les outils les plus puissants, qui permettent souvent les développements les plus rapides. Notons qu'une caractéristique essentielle des environnements de haut niveau est la *méta-programmation*, c'est à dire, la possibilité de construire des programmes de manière dynamique.

CENTAUR

Le système dont le **fonctionnement** est le plus proche de celui d'ATINF est le système CENTAUR [BCDIKLP87], pourtant destiné à l'édition de programmes dans des langages structurés.

CENTAUR utilise en effet la description d'un langage de programmation pour présenter et permettre l'édition de programmes respectant leurs spécifications.

On peut immédiatement faire le rapprochement entre logique et langage de programmation, preuve et programme, correction et respect des spécifications, il y a vraisemblablement beaucoup d'autres similitudes. Un travail a été effectué dans ce sens, il consiste à employer CENTAUR pour construire des preuves [Has88].

Toutefois, les formalismes de description des langages de programmation ne sont pas vraiment adaptés à la définition de systèmes formels. Nous préfèrons employer un formalisme spécialisé comme CC [CH88] ou LF [HHP89].

Les langages de programmation étant textuels, leur présentation est beaucoup plus simple que celle des logiques, qui emploie de nombreux symboles et graphismes. Nous préfèrons donc employer un formalisme plus général de présentation de documents.

Ainsi, CENTAUR ne possède pas le potentiel de base d'un éditeur de preuves. Par contre, il aurait éventuellement pu être utilisé pour produire des éditeurs pour les formalismes de définition et de présentation, mais notre but n'est pas de travailler directement sur ou dans ces formalismes.

Finalement, développer GLEF de manière indépendante est plus facile que d'intégrer ce potentiel de base à CENTAUR.

LISP

Les environnements LISP sont très pratiques pour développer des programmes. Ils proposent une gestion automatique de la mémoire, de grandes bibliothèques de fonctions, des outils de mise au point puissants et leurs interprètes permettent le test individuel des fonctions développées. Des extensions existantes (analyse lexicale et grammaticale, IUG) leur permettent de répondre à nos besoins.

Nous verrons toutefois que la gestion automatique de la mémoire nous est inutile, car nous devons la gérer d'une manière très particulière.

CAML

CAML [CH90] est un langage de programmation structuré simultanément **polymorphe** (C) et **fortement typé** (PASCAL), interfacé avec LEX et YACC. Il est donc particulièrement bien adapté à la définition de notre formalisme de définition de logiques.

Notons que CAML light est une version simplifiée de CAML, qui fonctionne sur les "petites" machines.

Problèmes liés à l'utilisation d'un environnement de haut niveau

En fait, plusieurs raisons nous ont dissuadés d'employer un environnement de haut niveau pour réaliser GLEF. Certaines de ces raisons relèvent de la distribution à un large public, d'autres sont purement techniques. Notons qu'au moins une de ces raisons concerne chaque environnement, ou le concernait, au début de la réalisation de GLEF.

Quand un environnement utilise beaucoup de **mémoire**, il ne peut pas toujours fonctionner sur une "petite" machine, surtout en parallèle avec d'autres applications. Un environnement peut aussi **manquer** ou **coûter cher** sur certaines machines. Et, quand il ne permet pas de réaliser des **applications indépendantes**, chaque utilisateur doit en posséder une version.

Souvent, un environnement est relativement **fermé**, et ne facilite pas son extension par des fonctions hors du langage. Au contraire, on doit modifier directement son code, ce qui rendrait GLEF **dépendant** de son implémentation. Souvent délicates à programmer, ces modifications excluent l'emploi d'un tel environnement comme outil de prototypage.

Les langages de programmation

Employer un langage de programmation classique permet d'éviter les problèmes liés à un environnement existant. L'utilisation de la mémoire peut être optimisée et les applications peuvent fonctionner indépendamment de l'outil de développement, même sur les "petites" machines.

A la méta-programmation près, les langages de programmation satisfont tous nos besoins. La méta-programmation n'est pas toujours facile, mais elle est toujours possible. Par exemple, on peut appeler dynamiquement des fonctions compilées séparément. Mais comme un appel dynamique n'est pas portable, réaliser ou employer un interprète auxiliaire est préférable.

Parmi les langages de programmation classiques, le C++ semble être le standard actuel. Les extensions objets sont d'ailleurs très utiles pour homogénéiser et modulariser les développements. Notamment, elles donnent au langage une certaine forme de polymorphisme multi-sortes fort, ce qui le rapproche un peu de CAML.

L'utilisation d'objets en C++ profite de celle d'un compilateur optimisant correctement les recherches de méthodes. Nous avons choisi un compilateur à l'interface rapide et agréable, mais qui n'optimise pas le code obtenu. Toutefois, GLEF pourra être recompilé avec un meilleur compilateur.

Interface utilisateur graphique

Pour être agréable et pratique à utiliser, GLEF doit proposer une *IUG* (Interface Utilisateur Graphique) moderne. On ne doit pas confondre l'IUG, qui sert à contrôler un programme, et les *primitives graphiques*, qui servent à dessiner et sont analysées plus loin.

Actuellement, la plupart des machines proposent une ou plusieurs *IUGG* natives (Interface Utilisateur Graphique Générique). Du point de vue de l'utilisateur, ces IUGG sont semblables, mais il n'est pas possible de les programmer indépendamment de la machine. Toutefois, l'utilisation de classes d'objets disponibles sur la plupart des machines réduit considérablement les coûts de développement, et minimise ainsi les coûts de portage.

Toutefois, il existe un noyau d'IUGG, *X-WINDOW*, dont l'aspect visuel n'est pas figé. C'est le cœur de différentes IUGG (Motif, OpenWin, MacX, etc.), sur différentes machines. Ainsi, un programme qui emploie X-WINDOW fonctionne sur toutes ces IUGG. X-WINDOW est maintenant très répandu et son emploi est incontournable.

Quand nous avons commencé à réaliser GLEF (première version sur SUN en 88), X-WINDOW n'était pas tout à fait figé, les rares IUGG fondées sur lui étaient très sommaires, et peu de programmes l'employaient. A l'époque, nous avions choisi d'employer une IUGG native (SUNTOOLS, puis MacOS).

Bien entendu, les versions futures de GLEF emploieront X-WINDOW, certainement via BEDROCK. BEDROCK est une bibliothèque de classes d'objets, en phase finale de développement chez Apple, IBM, et SYMANTEC. Elle permet de réaliser des applications indépendantes de la machine employée et de son IUGG (Macintosh, WINDOWS PC, NT, X-WINDOW, etc.). Notons que NEURON DATA développe une bibliothèque similaire, appelée OPENINTERFACE, mais qui est beaucoup plus coûteuse.

Primitives graphiques

Pour dessiner, GLEF a besoin de *primitives graphiques*. La plupart des systèmes d'exploitation et des IUGG (MS-DOS, UNIX, MacOS, X-WINDOW, etc.) en proposent. A quelques détails près (nom, ordre des arguments), beaucoup d'entre elles sont semblables. N'utiliser que ces primitives, en encapsulant leur appel, permet d'obtenir un code relativement portable.

Postscript est un langage de description de documents, reconnu comme standard pour l'impression. Display-Postscript est un ensemble de primitives graphiques, qui pourrait devenir un standard pour l'affichage.

Plusieurs machines (SUN, NEXT) l'emploient déjà dans leurs IUGG. Apple et IBM envisagent aussi de l'employer dans leur futur système d'exploitation commun (PINK). Toutefois, une IUGG peut encapsuler les primitives graphiques qu'elle emploie et les masquer au programmeur.

Matériel

Le Macintosh et le NEXT sont encore les seules machines dont les IUGG obéissent à des normes très précises. En effet, des applications qui emploient des boutons, des menus et des fenêtres de même forme ne sont pas toujours homogènes. En plus de son IUGG, Apple édite des documentations sur la manière d'écrire une application (couleur des icônes, touches de fonctions standard, etc.). Ainsi, les IUGG du Macintosh et du NEXT sont les plus naturelles et les plus homogènes. Cela ne signifie pas qu'elles manquent de puissance (NEXT est fondé sur UNIX). L'alliance entre Apple et IBM laisse supposer que l'IUGG du Macintosh soit destinée à une distribution de plus en plus large. Notons qu'elle sera, elle aussi, fondée sur UNIX.

Finalement, nous avons choisi de programmer GLEF en C++ sur un Macintosh, en employant l'IUGG et les primitives graphiques natives de

cette machine. Bien entendu, nous emploierons BEDROCK pour porter GLEF à d'autres machines. Nous emploierons aussi CAML light, si ce nouvel environnement de programmation se révèle mieux adapté à nos besoins.

2. Implémentation des langages

GLEF doit analyser et interpréter des définitions et des présentations spécifiées dans les langages décrits dans la première partie.

Les syntaxes de ces langages sont simples et bien déterminées. Les compilateurs d'analyseurs lexicaux et grammaticaux, *LEX* et *YACC*, ont donc produit automatiquement leurs **analyseurs syntaxiques**.

Les langages sources de LEX et YACC permettent de programmer des actions complémentaires à effectuer par les analyseurs. Ainsi, le langage source de YACC a permis de programmer la construction des **arbres abstraits** syntaxiques. Pendant cette construction, l'analyseur syntaxique du formalisme de définition **vérifie** les contraintes de **types**. Finalement, GLEF manipule directement les arbres abstraits syntaxiques.

Notons que pour les premières versions de GLEF, nous avions écrit un programme générant automatiquement les sources LEX, YACC et C++, pour l'analyse syntaxique, la construction et la manipulation des arbres abstraits syntaxiques d'un langage, à partir d'une spécification de sa syntaxe. Ce programme a été très utile pendant la phase de développement formel des langages. Mais les structures des arbres abstraits syntaxiques produits par les analyseurs obtenus n'étant pas assez souples, nous avons finalement abandonné ce programme.

3. Implémentation du formalisme de définition

a) Optimisation du temps de calcul

Le formalisme de définition est constitué d'un langage et d'un calcul de types, destiné à vérifier les définitions. Ce calcul de types appelle, sans arrêt, des transformations récursives de termes (recalage, mise sous forme normale) qui durent relativement longtemps.

Décrivons donc la principale méthode employée pour accélérer la vérification de types, dont la rapidité est essentielle.

Calcul des types

Le calcul du type d'un terme peut durer longtemps, surtout si le terme est composé d'un grand nombre de constructeurs. Or le calcul de types calcule au moins une fois, et souvent de **nombreuses** fois, le type de chaque terme:

- Une définition peut contenir plusieurs instances du même terme.
- La présentation peut nécessiter la visualisation du type de certains termes.
- Pendant l'édition, la vérification de contraintes de types autorise certaines manipulations.
- Le calcul du type d'un terme réutilise les types de ses sous-termes.

Pour éviter de calculer plusieurs fois le type de la plupart des termes, **GLEF mémorise tous les termes** rencontrés avec leurs types. Comme nous le verrons, la manière de mémoriser ces couples utilise relativement peu de mémoire et permet un accès rapide aux types des termes.

Calcul des formes

Une définition peut introduire des fonctions de profils multiples. Souvent, l'application d'une telle fonction à une liste d'arguments donnée n'est permise que par certains profils. En particulier, l'application de cette fonction à une liste d'arguments maximale donnée n'est permise, la plupart du temps, que par un seul profil.

Cependant, **la présentation de l'application d'une fonction de profil multiple dépend des profils qui la permettent.**

Pour garder la trace des profils utilisés, on associe à chaque terme `t` un ensemble d'entiers appelée *forme*, et noté *`form(t)`*, de même cardinal n que son type.

Si le terme considéré n'est pas une application, cet ensemble est celui des n premiers entiers.

Sinon, on considère sa fonction `f` de profils *`profil(f)`*. Les profils de `f` ayant permis l'application forment un sous-ensemble de `profil(f)`. `form(t)` est le sous-ensemble de `form(f)` tel que les entiers conservés sont ceux dont la position dans `form(f)` correspond à celle de ces profils dans `profil(f)`. Ceci est **possible** car les types sont **gérés sous forme de listes**. Comme nous l'avons vu, c'est une des limites de l'indépendance entre définition et présentation.

Le calcul des formes ne dure pas très longtemps, mais il est très facile de mémoriser la forme de chaque terme rencontré, à côté de son type.

Autres calculs

Pour les prochaines versions de GLEF, nous envisagerons de mémoriser les recalages et les formes normales de la même manière (voir chapitre Travail futur et conclusion section A.4).

b) Gestion de la mémoire

Mémoriser sans précautions le type et la forme de chaque terme (dans chaque contexte) peut nécessiter une quantité excessive de mémoire[20]. GLEF gère donc l'utilisation de la mémoire de manière plus fine.

Il emploie une méthode très générale, applicable quand un programme doit mémoriser un grand nombre de données avec une forte probabilité de redondance, ce qui est très courant en démonstration automatique.

Ce programme peut éviter de construire plusieurs fois la même donnée, en **répertoriant les données déjà créées**, pour les réutiliser. La table de *hash* est une forme de répertoire efficace.

Parfois, les données concernées sont structurées. Par exemple, ce sont **les termes d'une algèbre**. La fonction de hash peut exploiter cette structure. Pour calculer le code de hash d'une donnée structurée, il suffit de réutiliser les codes de hash des données qui la composent. Comme ce calcul n'est pas récursif, il est très rapide, surtout pour les termes de taille importante.

GLEF emploie cette méthode pour mémoriser les atomes, les termes, les contextes, les jugements de typage, les parures, les occurrences et les systèmes d'équations (avec leurs solutions) associés à chaque boîte (voir section B.6).

Sans cette méthode, la mémoire était principalement utilisée par les termes. Avec cette méthode, l'économie réalisée sur les termes est en pratique **de 90% à 99%**[21], les meilleurs résultats concernent les exemples les plus volumineux, qui sollicitent justement le plus de mémoire.

Notons qu'un environnement LISP mémorise les atomes de cette manière, mais pas les listes. Quand un programme LISP utilise un grand nombre de données avec une forte probabilité de redondance, on pourrait employer un interprète LISP dont les primitives éviteraient de construire plusieurs fois les mêmes données LISP. Toutefois, le programme ne pourrait pas appeler des primitives comme *rplaca*, qui altèrent directement les données structurées.

---

[20] Plusieurs dizaines de méga-octets pour des preuves de quelques centaines de pas d'inférence.

[21] Les statistiques sont calculées par GLEF.

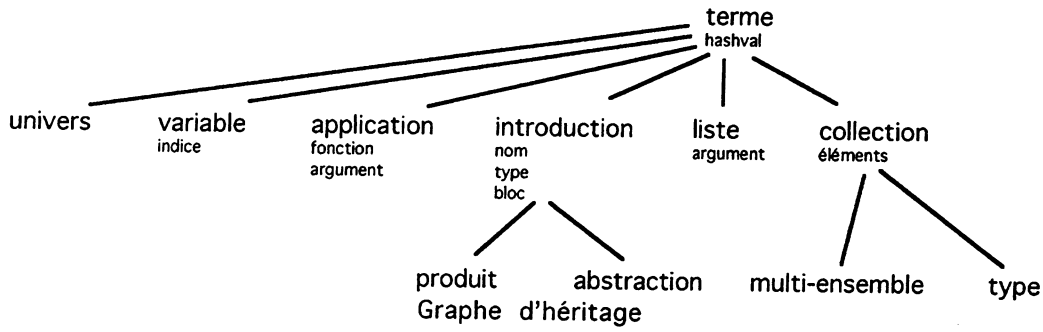
c) Représentation des termes et des types

Cette section décrit les structures de données du formalisme de définition. Ces structures de données sont gérées selon les méthodes décrites juste avant.

Utilisation d'objets

Dans un premier temps, nous avons choisi de représenter les données du formalisme de définition par des *objets*, pour faciliter et accélérer le développement. Par exemple, les termes sont représentés par des objets et leurs manipulations (recalage, forme normale, unification, etc.) sont définies comme des *méthodes*.

Un *graphe d'héritage* peut donc décrire la structure des termes:



Les objets sont représentés par les nœuds du graphe. Sous leurs noms apparaissent les *champs* qui les composent. Notons que les types apparaissent sur la même branche que les multi-ensembles, car ils ont de nombreuses méthodes communes.

Utilisation des indices de De Bruijn [dBr72]

En λ-calcul, les noms des variables peuvent ralentir différentes manipulations. Par exemple, le test d'égalité de deux termes doit tenir compte de **l'indifférence des noms** des variables liées. La substitution doit éviter la **liaison accidentelle** de variables libres du terme substitué avec des variables du terme destination. Pour cela, on pourrait renommer systématiquement les variables libres du terme substitué. En λ-calcul typé, d'autres liaisons accidentelles peuvent survenir lors de l'utilisation d'un type dans certains contextes. De même, on pourrait renommer systématiquement les variables libres du type.

Notons que ces renommages seraient particulièrement gênants pour GLEF, car les noms des variables peuvent apparaître dans la présentation graphique des termes.

Pour éviter ces problèmes, nous avons choisi de représenter les termes par leur *notation de De Bruijn*. Cette notation consiste à

remplacer chaque variable liée par un entier indiquant la distance à son symbole de liaison. Cette distance, appelée *indice de De Bruijn*, est exprimée en nombre de symboles de liaisons rencontrés, lorsqu'on remonte de la variable à son symbole de liaison, en suivant la structure du terme.

Les variables non liées (entités et variables libres) sont considérées comme liées par un contexte extérieur au terme. En λ-calcul typé, la notation de De Bruijn oblige à considérer chaque terme dans son **contexte d'utilisation**. En effet, contrairement à la notation classique, la notation de De Bruijn d'un terme varie en fonction de ce contexte. Les indices des variables non liées augmentent (diminuent) quand on le prolonge (réduit).

d) Manipulations formelles

Cette section décrit les différentes *manipulations formelles* employées par GLEF. Introduisons quelques notions préliminaires pour faciliter cette description.

Définitions et notations

Les symboles [et] délimitent les listes. @ et :: dénotent respectivement la concaténation de listes et l'ajout d'un élément à une liste. $|l|$ dénote la longueur de la liste l.

$\mathbb{Z}$ dénote l'ensemble des entiers, $\mathbb{N}$ dénote l'ensemble des entiers positifs, $\mathbb{N}_n$ dénote l'ensemble $\{1...,n\}$. On définit l'addition d'un entier à un ensemble d'entiers par: $S+n=\{s+n \mid s\in S\}$.

La notation classique d'un terme est appelée *terme concret*. Sa notation de De Bruijn est appelée *terme abstrait*. Les notions de terme concret et abstrait correspondent à celles de syntaxe concrète et abstraite. Dans ce qui suit, le mot terme, sans précision, signifie terme concret.

Un *contexte* est une liste de termes abstraits associés chacun à un nom (voir section A.1.d pour la définition des contextes bien formés). Un contexte Γ_1 *prolonge* un contexte Γ_2 ssi il existe un contexte Γ_3 tel que $\Gamma_1=\Gamma_3@\Gamma_2$. $V_\Gamma=\mathbb{N}_{|\Gamma|}$ représente l'ensemble des variables (entités et variables libres) introduites par le contexte Γ. $V_{\Gamma_1}^{\Gamma_2} = V_{\Gamma_1} \setminus V_{\Gamma_2}$ représente l'ensemble des variables introduites par Γ_1 mais pas par Γ_2, quand Γ_1 prolonge Γ_2.

Comme un terme abstrait est toujours considéré dans un contexte, on définit une *référence* par un couple formé d'un terme abstrait et de son contexte. La fonction $\bar{r}$ sert à construire une référence à partir d'un contexte et d'un terme abstrait. Les fonctions $\bar{\gamma}$ et $\bar{t}$ servent

respectivement à extraire le contexte et le terme abstrait d'une référence.

Une *occurrence* est la position occupée par un sous-terme d'un terme donné. Selon l'usage courant, une occurrence est représentée par une liste de codes indiquant le chemin allant de la racine du terme au sous-terme concerné.

La notion d'*occurrence généralisée* est similaire. Elle autorise en plus le passage d'un terme à son type, d'une application à son $n^{\text{ième}}$ argument et d'une abstraction ou d'un produit dépendant à sa variable liée. Elle est très utile pour la présentation et les manipulations de termes.

Recalage

Dans des contextes différents, un terme concret peut être représenté par des termes abstraits différents. Le *recalage* [Hue91] transforme un terme abstrait, pour qu'il représente le même terme concret, lorsqu'on prolonge ou on réduit son contexte, ou le contexte d'un terme qui le contient.

C'est une fonction récursive très simple, qui ne change pas les indices des variables liées et ajoute aux indices de certaines variables non liées (entités et variables libres) un entier positif ou négatif appelé *valeur de recalage*.

$t{+}(m,n)$ dénote le terme abstrait t recalé de n, les variables non liées d'indices inférieurs ou égaux à m n'étant pas modifiées. $t{+}n$ dénote $t{+}(0,n)$. $t{-}n$ dénote $t{+}(-n)$. Quand t est une variable et $m{=}0$, + correspond à l'addition des entiers.

Notons que le recalage peut échouer. $t{+}(m,n)$ échoue si et seulement si $\{i \in \mathrm{V}_t \mid m < i \wedge i+n < 1+m\} \neq \varnothing$, où V_t est l'ensemble des variables non liées de t.

Diminution de contexte

La *diminution* d'un contexte sert à retirer certaines de ses introductions de variables. Les types des variables introduites après ces introductions (et situées avant dans la liste) doivent être recalés. Notons que la diminution d'un contexte échoue quand certains de ces recalages échouent.

On note $\Gamma \backslash V$ le contexte Γ diminué de l'ensemble V de variables. Etant donné un contexte $\Gamma = \left[(n_1,t_1)\cdots;(n_{|\Gamma|},t_{|\Gamma|})\right]$ on pose:

$$\Gamma \setminus i = \left[(n_1,t_1-(i-2,1))\cdots;(n_j,t_j-(i-1-j,1))\cdots;(n_{i-1},t_{i-1}-1);(n_{i+1},t_{i+1})\cdots(n_{|\Gamma|},t_{|\Gamma|})\right]$$

On définit ensuite:

$$\begin{cases} \Gamma \setminus V = (\Gamma \setminus i) \setminus (V \setminus \{i\}) & (V \subset \mathrm{V}_\Gamma \wedge i = \max(V)) \\ \Gamma \setminus \varnothing = \Gamma \end{cases}$$

Quand aucun recalage n'échoue, le contexte obtenu est bien formé.

Égalité

Nous employons différentes notions d'égalité entre termes abstraits, et avons écrit un test pour chacune de ces notions.

L'*égalité structurelle* permet d'indiquer si deux termes abstraits, dans les mêmes contextes, sont égaux et leur présentation est la même. Comme nous l'avons vu, les termes abstraits sont mémorisés dans des tables de hash et ne sont jamais construits plusieurs fois. Ce test est donc immédiat et se réduit à une comparaison de pointeurs.

L'*égalité de création* est une égalité structurelle, utile quand l'un des deux termes n'est pas encore mémorisé dans la table de hash, et doit justement y être ajouté. Ce test est constitué d'un rapide parcours de profondeur 1, et de tests immédiats d'égalité structurelle. Il suppose en effet que les sous-termes du terme ajouté sont déjà mémorisés dans la table de hash.

L'*égalité abstraite* sert à tester si deux termes abstraits, dans les mêmes contextes, sont identiques au sens du formalisme de définition. A cause des noms des variables liées (qui restent mémorisés au niveau des liaisons pour la présentation) et des multi-ensembles, un parcours complet des terme est obligatoire (voir juste après les remarques concernant les formes normales). Cependant des tests d'égalité structurelle peuvent le court-circuiter.

L'*égalité concrète* sert à tester si deux termes abstraits, dans deux contextes, représentent le même terme. Comme GLEF normalise systématiquement les termes, l'égalité concrète correspond à l'*égalité définitionnelle* des termes concrets. Ce test est écrit de la même façon que celui de l'égalité abstraite, mais il **évite les recalages** en considérant directement les valeurs de recalage.

Formes normales

Nous supposons que le formalisme de définition admette la normalisation forte pour la $\beta\eta\delta$-réduction. Toutefois, pour traiter les multi-ensembles, les types ensembles et les noms des variables liées, les procédures de test d'égalité abstraite et concrète sont **récursives** et particulièrement coûteuses.

Pour des tests d'égalité plus efficaces, les prochaines versions de GLEF calculerons une forme $\beta\eta\delta$-normale, modulo les multi-ensembles, les types ensembles et les noms des variables liées. Par exemple, pour les multi-ensembles et les types ensembles, mémoriser chaque terme

une fois et une seule permet d'employer **l'ordre défini par les pointeurs** au lieu d'un ordre plus long à calculer (lexicographique, etc.). Quant aux noms des variables liées, nous envisageons de les retirer des termes (voir chapitre Travail futur et conclusion section A.4). Les tests d'égalité définitionnelle entre deux termes se réduiront donc à l'égalité structurelle de leurs formes normales (après recalage pour l'égalité concrète), c'est à dire à une comparaison de pointeurs.

Même si la β-réduction reste systématiquement appliquée, les termes ne seront pas systématiquement normalisés. Mais la forme normale de chaque terme sera systématiquement calculée et mémorisée, comme son type et sa forme. Ainsi, elle sera calculée une et une seule fois. Notamment, **la procédure de normalisation des termes ne sera plus récursive** (aux recalages près), car elle réutilisera les formes normales de leurs sous-termes, déjà calculées.

De plus, **mémoriser séparément** la forme normale de chaque terme, dans chaque contexte est **préférable** à normaliser directement chaque terme. En effet, cela permet de laisser les éléments des multi-ensembles et des types ensembles à leurs places, pour la **présentation**.

Notons aussi que pour simplifier encore l'égalité concrète, chaque forme normale pourrait être mémorisée sous la forme d'un **terme** et d'une **valeur** de recalage. Le terme correspondrait au recalage de la forme normale par la valeur de recalage, choisie négative, de valeur absolue maximum.

Fermeture

La *fermeture* consiste à lier certaines variables libres d'un terme abstrait, le contexte associé étant diminué de ces variables. Selon le genre de fermeture choisi, les variables concernées sont liées par des abstractions ou des produits dépendants.

Simplification

La *simplification* consiste à éliminer parmi les abstractions en tête d'un terme abstrait, celles qui introduisent des variables inutilisées dans ce terme abstrait. Comme cette opération ne change pas le contexte du terme abstrait, elle n'en tient pas compte.

Notons qu'éliminer des abstractions à l'intérieur d'un terme est plus délicat, car il faudrait considérer leur environnement d'utilisation.

Substitution

Une *substitution* sur un contexte Γ est un ensemble de couples associant une variable de V_Γ à un terme abstrait, considéré dans Γ. Le type T_2 du terme doit être "compatible" au sens de l'application, avec

celui T_1 de la variable. C'est à dire, on doit avoir $\mathrm{tcp}(T_1,T_2)$ (voir les règles de jugement de typage section A.1.d).

Etant données une référence r et une substitution sur $\bar{\gamma}(r)$, la *substitution* consiste à remplacer dans $\bar{t}(r)$, les variables de la substitution par leurs termes associés (en effectuant les recalages adéquats), et de diminuer $\bar{\gamma}(r)$ de ces variables.

Filtrage et Unification

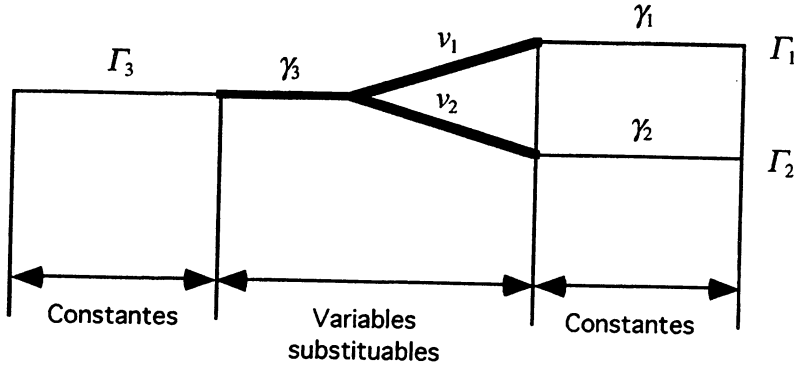
Le *filtrage* est un cas particulier de l'unification, il est traité de manière similaire.

L'*unification* de deux références r_1 et r_2 consiste à tester si remplacer dans $\bar{t}(r_1)$ et $\bar{t}(r_2)$, **certaines** variables de $V_{\bar{\gamma}(r_1)}$ et $V_{\bar{\gamma}(r_2)}$ par des termes de $\bar{\gamma}(r_1)$ et $\bar{\gamma}(r_2)$ permet d'égaliser (concrètement) $\bar{t}(r_1)$ et $\bar{t}(r_2)$, dans $\bar{\gamma}(r_1)$ et $\bar{\gamma}(r_2)$. De plus, on peut **associer** des variables de $V_{\bar{\gamma}(r_1)}$ avec des variables de $V_{\bar{\gamma}(r_2)}$, pour qu'elles soient remplacées par des termes égaux (concrètement). Les *variables substituables* sont les variables de $V_{\bar{\gamma}(r_1)}$ et $V_{\bar{\gamma}(r_2)}$ dont on autorise le remplacement. Les *constantes* sont les autres.

Notre algorithme d'unification suppose par ailleurs que les contextes Γ_1 et Γ_2 des deux termes abstraits soient de la forme:

$$\Gamma_1=\gamma_1@v_1@\gamma_3@\Gamma_3$$
$$\Gamma_2=\gamma_2@v_2@\gamma_3@\Gamma_3$$

Les variables **substituables** sont celles introduites par $v_1@\gamma_3$ et $v_2@\gamma_3$, c'est à dire $V_{v_i@\gamma_3}+|\gamma_i|$, dans Γ_i $(i=1,2)$. Les variables introduites par γ_3 dans Γ_1 et Γ_2 sont **associées**, c'est à dire $k+|\gamma_1|+|v_1|$ dans Γ_1 est associée à $k+|\gamma_2|+|v_2|$ dans Γ_2 $(k=1...,|\gamma_3|)$. Notons que tous les contextes sont déterminés quand on donne Γ_i et $n_i=|\gamma_i|$ $(i=1,2,3)$.



Forme des contextes de deux termes abstraits à unifier.

Les cas rencontrés en pratique sont ceux où $n_1 = n_3 = 0$ (les deux ensembles de variables substituables sont "disjoints") et $|v_1| = |v_2| = 0$ (les deux ensembles de variables substituables sont "confondus"). Notons que même dans le premier cas, v_1 et v_2 peuvent être égaux.

Notons aussi que **plusieurs substitutions** peuvent satisfaire un filtrage ou une unification à cause des multi-ensembles et des types ensembles.

Pour le moment, nous employons un algorithme d'unification du **premier ordre**. Mais l'emploi d'un algorithme d'unification **d'ordre supérieur** améliorerait **l'ergonomie** de GLEF (voir chapitre Travail futur et conclusion section A.8).

Cet algorithme est récursif, ses arguments sont les deux termes abstraits, leurs deux contextes, deux valeurs de recalage associées et un indice de profondeur. Notons que les deux contextes sont donnés sous une **forme particulière**. Cette forme indique les variables substituables et permet de mémoriser les substitutions, en **évitant** leur **construction** séparée. Ces informations sont associées au niveau de chaque élément des deux contextes. Grâce à la forme de ses arguments, l'algorithme **n'effectue ni remplacements, ni recalages**, il est donc très efficace.

Quand une unification réussit, **combiner** les informations associées aux deux contextes permet de calculer les couples de substitutions à appliquer aux deux références. Cette combinaison consiste à ordonner les introductions des variables substituables des deux contextes, en considérant les contraintes de **dépendance de types**. Elle n'effectue pas explicitement les **recalages** de ces introductions et des variables libres des termes unifiés, mais les **code** seulement. Ensuite, l'algorithme de substitution exploite ce codage pour éviter des transformations intermédiaires des termes unifiés.

Du point de vue externe, on considère simplement que l'unification retourne un couple de substitutions.

Renommage

GLEF doit permettre à l'utilisateur de changer les noms des variables libres ou liées, pour améliorer la présentation des preuves. Bien entendu, l'utilisateur ne peut pas changer les noms des entités, mais la manipulation décrite ici est générale.

Étant données une référence et une occurrence généralisée indiquant la position d'une variable dans le terme abstrait de la référence, le *renommage* consiste à changer le nom de la variable concernée, dans la référence.

Grâce aux indices de De Bruijn, changer le nom d'une variable est immédiat, **il suffit de le changer au niveau de sa liaison**, que celle-ci soit située dans le terme abstrait ou dans le contexte.

Cependant, comme chaque terme abstrait est mémorisé une fois et une seule, changer le nom d'une variable au niveau de sa liaison change son nom dans **toutes les liaisons** qui introduisent un contexte structurellement égal, dans le reste de la preuve. De plus, comme les types sont mémorisés séparément des termes, changer le nom d'une variable liée dans le type d'un terme change son nom dans **le type de tous les termes** de types structurellement égaux. C'est une **limite** de l'indépendance entre la représentation interne des termes abstraits et leur présentation.

Pour économiser la mémoire, les futures versions de GLEF mémoriserons vraisemblablement les noms des variables séparément des liaisons (voir chapitre Travail futur et conclusion section A.4). L'égalité structurelle ne tiendra donc plus compte de ces noms et le problème de dépendance concernera les termes égaux de manière abstraite.

Suppression et Remplacement

Intuitivement, le *remplacement* consiste à mettre un terme à la place d'un sous-terme d'un terme donné. La *suppression* consiste à mettre une variable libre à la place du sous-terme. Ces opérations peuvent enlever ou ajouter des variables libres au terme englobant. La suppression est un cas particulier du remplacement, elle est traitée de manière similaire.

En λ-calcul non typé, le remplacement ne pose pas de problèmes. En logique du premier ordre, le terme remplaçant doit seulement être de la sorte du terme remplacé (formule ou terme).

En λ-calcul typé, le remplacement est plus délicat. Pour préserver le typage, il doit modifier certains sous-termes hors du sous-terme remplacé, pour changer leurs types.

Une opération auxiliaire permet donc de modifier un terme pour **changer son type**. Elle pourrait simplement remplacer le terme par une nouvelle variable libre, mais elle doit changer le moins possible la structure du terme modifié. Pour l'instant, elle correspond à l'algorithme suivant:

- Si le terme est une application, on essaie de conserver sa fonction la plus imbriquée, en changeant un par un les types de ses arguments, par un appel récursif à l'opération auxiliaire.
- Sinon, on le remplace par une nouvelle variable libre.

Notons que l'opération auxiliaire généralise l'application du remplacement aux sous-termes situés à des occurrences généralisées.

Un autre algorithme, plus général, est donné en Annexe A, mais il n'est pas encore implémenté.

Profils des manipulations formelles

Avant d'écrire les profils des manipulations formelles, introduisons d'autres notations. $\times$, $\rightarrow$ dénotent respectivement le produit cartésien et l'espace fonctionnel. $x{:}T \times U(x)$ représente un produit cartésien dépendant. T^n dénote $\underbrace{T \ldots \times T}_{n}$. π_i dénote la $i^{\text{ième}}$ projection.

S dénote l'ensemble des chaînes de caractères. context, term et occurrence dénotent respectivement l'ensemble des contextes, l'ensemble des termes et l'ensembles des occurrences généralisées. reference $\subset$ P(context $\times$ term) dénote des références (le terme est typable dans le contexte bien formé). substitution $\subset$ P(N $\times$ term) dénote l'ensemble des substitutions.

Les profils des manipulations formelles peuvent s'écrire (quand la place des arguments est ambiguë, leurs noms sont rappelés):

$$+ \in \text{term} \rightarrow \text{term} \cup \text{Z} \times \text{Z} \rightarrow \text{Z} \cup \text{P(Z)} \times \text{Z} \rightarrow \text{P(Z)}$$

$$\text{close} \in r{:}\text{reference} \times \text{P}\left(\text{V}_{\tilde{\gamma}(r)}\right)$$

$$\text{simplify} \in \text{term} \rightarrow \text{term} \cup \text{reference} \rightarrow \text{reference}$$

$$\text{filter} \in \text{reference}^2 \times \text{context} \times \text{N} \rightarrow \text{P(substitution)}$$

$$\text{unify} \in \text{reference}^2 \times \text{context} \times \text{N}^3 \rightarrow \text{P}\left(\text{substitution}^2\right)$$

$$\left(r_1, r_2, \Gamma_3, n_1, n_2, n_3\right)$$

$$\text{rename} \in \text{reference} \times \text{occurence} \times \text{S} \rightarrow \text{reference}$$

$$\text{delete} \in \text{reference} \times \text{occurence} \rightarrow \text{reference}$$

$$\text{replace} \in \text{reference} \times \text{occurence} \times \text{reference} \rightarrow \text{reference}$$

(référence destination, occurrence, référence remplaçante)

$$\text{substitute} \in \text{reference} \times \text{substitution} \rightarrow \text{reference}$$

$*$ transforme naturellement une fonction de $A \rightarrow B$ en une fonction de P(A)$\rightarrow$P(B), et substitute en une fonction de reference $\times$ P(substitution) $\rightarrow$ P(reference).

Notons que la description des profils ne tient pas compte des **erreurs**. En fait, on suppose qu'une fonction puisse toujours retourner une valeur ε quand elle échoue, et que cette valeur fasse échouer les fonctions qui l'appellent.

e) Environnements de définition

En mémoire externe, les définitions ne contiennent pas directement des termes concrets, mais des contextes qui peuvent introduire des termes concrets avec des nommages. Une primitive du type `#include` sert à relier les différentes parties d'une définition répartie sur plusieurs fichiers. Ces parties forment des contextes de niveau supérieur appelés *environnements de définition.* Les environnements de définition servent à structurer les définitions, sans employer de véritable langage de structuration [HST89], [LB92] (voir chapitre Travail futur et conclusion section A.11).

Ainsi, ils servent à définir des langages et des systèmes formels par **raffinements successifs**. Par exemple, un prolongement de la définition des tableaux sémantiques en logique du premier ordre peut définir les tableaux sémantiques en logique modale, etc. (voir chapitre Utilisation section C.2.d et C.3).

Ils servent à **spécifier des théories**. Par exemple, un prolongement de la définition du langage de la logique du premier ordre peut définir les axiomes d'une théorie des ensembles (voir chapitre Utilisation section C.2.b).

Enfin, ils servent à **séparer les preuves** de leurs systèmes formels. En effet, quand l'utilisateur emploie un système formel, il veut choisir les preuves à considérer.

Pour **permettre la mémorisation de preuves en cours de construction** (non connexes), une liste de termes peut prolonger le contexte spécifié par un environnement de définition. Ces termes sont considérés dans ce contexte. Notons que l'analyseur actuel de GLEF n'admet que des contextes. La procédure de sauvegarde de preuves nomme donc systématiquement les termes construits dans la fenêtre d'édition, pour son édition ultérieure. Cette faiblesse de GLEF devrait être corrigée rapidement.

4. Implémentation du langage de présentation

Représentation interne

La représentation interne des parures du langage de présentation est classique. Toutefois, les parures sont mémorisées dans une table de hash pour un accès rapide et une économie de mémoire.

Présentation des termes

L'algorithme de présentation des termes traduit les termes abstraits du formalisme de définition en structures de boîtes, en obéissant aux parures du langage de présentation.

Comme indiqué dans la description formelle du langage de présentation, cet algorithme considère un *terme courant*, à présenter dans une *boîte courante*.

- Si le terme courant est instance d'un **nommage** qui porte le nom d'une parure, et la boîte courante n'est pas une boîte de plus haut niveau, la parure sert à présenter le terme courant. L'algorithme de **filtrage** permet de déterminer si le terme courant est instance de chaque nommage du contexte de définition. Notons que dans une boîte de plus haut niveau, GLEF ne "compacte" pas ainsi la présentation d'une instance de nommage. Dans une telle boîte, l'utilisateur s'interresse vraisemblablement à la présentation "complète" du terme nommé.
- Si le terme nommé est une entité ou une application d'entité qui porte le nom d'une parure, la parure sert à présenter le terme courant.
- Sinon la parure par défaut, déterminée par la racine du terme courant, permet de présenter celui-ci.

Quand une parure est interprétée, ses instructions sont executées de manière séquentielle.

Elles servent à **créer** de nouvelles boîtes, **changer** provisoirement le terme courant (en l'une de ses occurrences généralisées) ou **appeler** l'algorithme de présentation de manière récursive.

Quand elles produisent des **erreurs**, les effets de **toutes les instructions** de la parure interprétée **sont annulés** et la parure par **défaut** est utilisée. Par exemple, quand le terme courant n'a pas la forme attendue, des occurrences peuvent manquer.

Normalement, les instructions des parures par défaut ne produisent pas d'erreurs. Cependant, comme l'utilisateur peut **redéfinir ces parures par défaut**, un test sur leur utilisation assure **l'arrêt de l'algorithme**.

Contextes de présentation et Environnements de présentation

A l'instar des entités du formalisme de définition, les parures sont gérées aux niveaux **local** et **global** sous forme de piles que l'on peut respectivement appeler *contexte de présentation* et *environnement de présentation*. Au niveau local, cette forme de gestion permet de surcharger la présentation d'entités du contexte de définition. Au niveau global, elle reflète celle de l'environnement de définition.

5. Organisation de la mémoire externe

Pour sauvegarder les logiques ou les preuves créées par l'utilisateur, GLEF produit des fichiers contenant des contextes de définition et de présentation. Ces fichiers doivent être rangés parmi ceux déjà définis pour GLEF.

Employer le système de gestion de fichiers de la machine est plus rapide qu'écrire une gestion de base de données ou un langage de structuration de théories.

Le principe est de créer un dossier pour chaque système formel. Le dossier de chaque système formel contient sa définition, sa présentation et ses preuves. Quand un système formel ou une théorie est défini à partir d'un système formel existant, son dossier peut être placé dans celui de ce système formel. Un dossier racine regroupe tous les dossiers des systèmes formels qui ne sont pas définis à partir d'un autre système formel, il contient aussi les fichiers de présentations par défaut et utilitaires. Quand un même environnement permet d'étendre plusieurs systèmes formels, le mécanisme de lien[22], souvent présent dans le système de gestion de fichier, permet d'éviter de dupliquer son fichier ou son dossier. Ce mécanisme de lien donne une structure de graphe à la hiérarchie de fichiers et de dossiers.

D'autre part, contrairement à une gestion de base de données, cette organisation permet à l'utilisateur d'éditer directement les fichiers de définition ou de présentation avec un simple éditeur de texte. Cette édition est utile et sûre, comme GLEF vérifie systématiquemnt les fichiers utilisés.

6. Implémentation du langage de boîtes

Calcul

La structure de boîtes étant donnée, l'algorithme de calcul des boîtes recherche des valeurs à donner aux champs des boîtes, pour **satisfaire** toutes les relations. Bien entendu, il ne tente pas de résoudre le système d'équations de manière générale.

Tout d'abord, il **"résout" le système** d'équations à dix inconnues **associé à chaque boîte**, pour exprimer ses dix champs en fonction des champs des boîtes environnantes.

Ce système d'équations n'est pas, lui non plus, résolu de manière générale. On le suppose décomposé en deux systèmes d'équations à cinq inconnues, sur les champs "horizontaux" et les champs "verticaux" de la

---

[22]"Soft link" pour UNIX. "Alias" pour Macintosh.

boîte concernée (les champs des boîtes environnantes utilisés sont indifférents). Les équations implicites servent à éliminer les champs `côté bas`, `médiane horizontale`, `côté droit` et `médiane verticale`. De même, deux équations données par l'utilisateur servent à éliminer les champs `base horizontale` et `base verticale`, quand ils apparaissent dans le système. Enfin, il reste deux systèmes à deux inconnues, simples et rapides à résoudre.

Ensuite, il se comporte comme un **évaluateur d'attributs incrémental**. Pour le rendre incrémental, chaque boîte mémorise les valeurs des champs `côté haut`, `côté gauche`, `hauteur` et `largeur`. Les valeurs de `côté haut` et `côté gauche` ne sont pas mémorisées de manière absolue, mais relativement à la boîte mère. Ainsi, elles sont plus "stables" lors des transformations de la structure de boîtes.

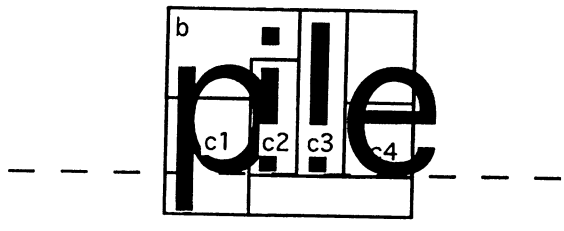
Cet algorithme est **rapide** et utilise **peu de mémoire**. Notons que certaines modifications (constructions) de boîtes, génèrent des systèmes de plusieurs milliers d'équations.

Parfois, la position d'une boîte n'est pas donnée explicitement, mais elle est déterminée par la position de la boîte fictive qui l'englobe, elle et ses boîtes sœurs (voir l'exemple qui suit).

Pour admettre ce type de positionnement, le système d'équations associé à une boîte admet qu'un champ horizontal et un champ vertical soit provisoirement indéterminé. C'est à dire, une équation peut manquer à chaque système d'équations à deux inconnues final. Ainsi, l'évaluateur d'attributs est capable de propager des **fonctions affines d'un des champ** d'une boîte arbitraire, au lieu de valeurs rationnelles. La valeur de ce champ doit être déterminée par d'autres boîtes de la structure.

Notons que lors de l'utilisation de boîtes fictives, un maximum ou un minimum sur des fonctions affines s'évaluent en une fonction affine, quand les coefficients et les champs de chaque fonction affine sont identiques.

Exemple:



L'ensemble d'équations suivantes décrit la disposition verticale de cette structure de boîtes. Les noms en majuscules représentent des

constantes. `b, c1, c2, c3, c4` dénotent les boîtes du dessin, `e` dénotent la boîte fictive englobant `c1, c2, c3` et `c4`.

Positionnement et dimensionnement de la boîte mère:

```
b.top=Y
b.height=e.h
```

Positionnement de la boîte fictive:

```
e.top=b.top
```

Positionnement de l'axe de référence de la boîte mère:

```
b.vref=c1.vref
```

Description des lettres:

```
c1.top=c1.vref-A1
c1.bottom=c1.vref+D1
c2.top=c2.vref-A1
c2.bottom=c2.vref+D1
c3.top=c3.vref-A1
c3.bottom=c3.vref+D1
c4.top=c4.vref-A1
c4.bottom=c4.vref+D1
```

Positionnement relatif des lettres:

```
c2.vref=c1.vref
c3.vref=c2.vref
c4.vref=c3.vref
```

Voyons qu'elle pourrait être la démarche de l'algorithme...

L'équation `b.height=e.h` s'écrit aussi:

```
b.height=max(c1.bottom,c2.bottom,c3.bottom,c4.bottom)
                -min(c1.top,c2.top,c3.top,c4.top)
```

Après élimination par exemple de `c1` puis `c3` puis `c4`:

```
b.height=max(c2.vref+D1,c2.vref+D2,
                              c2.vref+D3,c2.vref+D4)
                -min(c2.vref-A1,c2.vref-A2,
                                 c2.vref-A3,c2.vref-A4)
```

En posant `D=max(D1,D2,D3,D4)` et `A=min(A1,A2,A3,A4)`, on obtient:

```
b.height=D+A
```

etc..

Fin de l'exemple.

L'algorithme se termine toujours, mais il peut échouer pour trois raisons: Soit il n'y a pas de solution, soit il n'en a pas trouvé, soit il y a une infinité de solutions. Dans ces cas, la présentation par défaut sert à présenter l'ensemble des boîtes.

Toutefois, l'algorithme pourrait tenter de propager des fonctions plus générales que les fonctions affines, de fixer les valeurs de variables libres selon des règles "bien" choisies, ou de modifier l'ensemble d'équations en utilisant localement des présentations par défaut. Ces améliorations n'ont pas encore été, ni réalisées, ni étudiées précisément (voir chapitre Travail futur et conclusion section A.6).

Contenu

Les boîtes feuilles de la structure de boîtes peuvent contenir un certain nombre d'éléments graphiques: du texte avec attributs typographiques (police, taille, style), des symboles, des images ou des présentations externes.

Dans le langage de présentation, des noms dénotent les symboles et les images utilisés. Des tables séparées spécifient les symboles et les images associées à ces noms. Comme ces tables dépendent de la logique utilisée, elles sont gérées sous forme d'environnements, comme les environnements de définition et de présentation. En pratique, elles sont mémorisées dans les fichiers de présentation.

7. Intérêt de la structure de boîtes

Quand les équations de la structure de boîtes sont résolues, celle-ci est une représentation interne des données graphiques. Pour **afficher** la preuve complète, il suffit alors de **parcourir** cette structure de boîtes.

Grâce à la notion de **sélection**, elle permet aussi **d'interagir** avec l'utilisateur. Cette interaction sert, entre autres, à diriger la **visualisation hiérarchique**.

a) Sélection et déplacement

Une *boîte terme* est une boîte qui constitue la présentation d'un terme du formalisme de définition.

Quand l'utilisateur visualise ou manipule une preuve, il peut former une sélection avec la souris. Une *sélection* est un ensemble de boîtes termes, que GLEF doit considérer pour une action donnée. La formation de cette sélection respecte l'interface standard Macintosh:

L'inversion video indique la sélection. Quand l'utilisateur clique sur l'image, il sélectionne la plus petite boîte terme qui contient le point cliqué. S'il maintient en même temps la touche majuscule enfoncée, il

ajoute à la sélection la plus petite boîte terme qui contient d'une part, le point cliqué, et d'autre part, toutes les boîtes sélectionnées qui contiennent le point cliqué. Ces boîtes sont alors retirées de la sélection.

En maintenant le bouton de la souris enfoncé, il peut *déplacer* la sélection vers une destination graphique (boîte, icône, etc.).

b) Visualisation hiérarchique

La structure de boîte et la notion de sélection permettent la visualisation hiérarchique des preuves. Ainsi, l'utilisateur peut cacher ou montrer différentes parties de la preuve éditée.

Le bouton **Montrer** (**Show**) montre chaque sous-boîte des boîtes sélectionnées, dont au plus une boîte ancêtre est cachée et possède l'attribut HIDE, défini dans le langage de présentation.

Le bouton **Tout Montrer** (**Show All**) montre toutes les sous-boîtes des boîtes sélectionnées. Son appel correspond à des appels successifs de **Show**, terminés lorsque toutes les sous-boîtes des boîtes sélectionnées sont visibles.

Le bouton **Cacher** (**Hide**) cache chaque sous-boîte des boîtes sélectionnées, qui possède l'attribut HIDE. Bien entendu, les sous-boîtes de boîtes cachées sont cachées.

Le bouton **Attraper** (**Catch**) sert à examiner la sélection, en cachant le reste de la preuve, c'est à dire les boîtes et les parties de boîtes hors de la sélection.

Le bouton **Relâcher** (**Release**) annule le dernier appel de **Attraper**. Des appels successifs de **Relâcher** annulent des appels successifs de **Attraper**. Quand tous les appels de **Attraper** sont annulés, les boîtes principales sont visibles.

Toutefois, la possibilité de cacher les boîtes n'a d'intérêt que dans la mesure où les parties cachées peuvent être contractées pour diminuer la surface d'écran utilisée par la preuve éditée. Décrivons donc comment GLEF contracte les boîtes cachées.

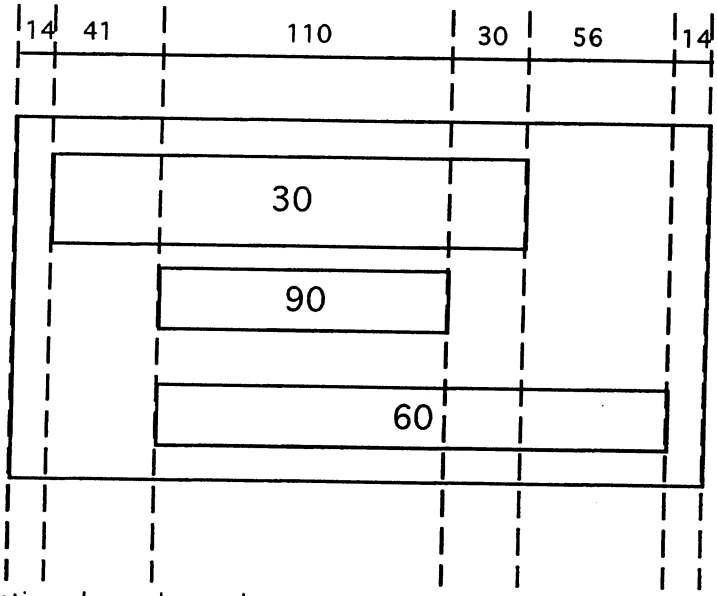
Notons que contracter des parties de preuve falsifie les équations de la structure de boîte. Pour considérer les boîtes cachées, GLEF pourrait tenter de modifier ces équations de façon automatique ou spécifiée dans le langage de présentation. Mais cette méthode ne paraît ni simple, ni naturelle.

L'algorithme proposé **contracte directement l'image** de la preuve, calculée par résolution des équations de la structure de boîtes.

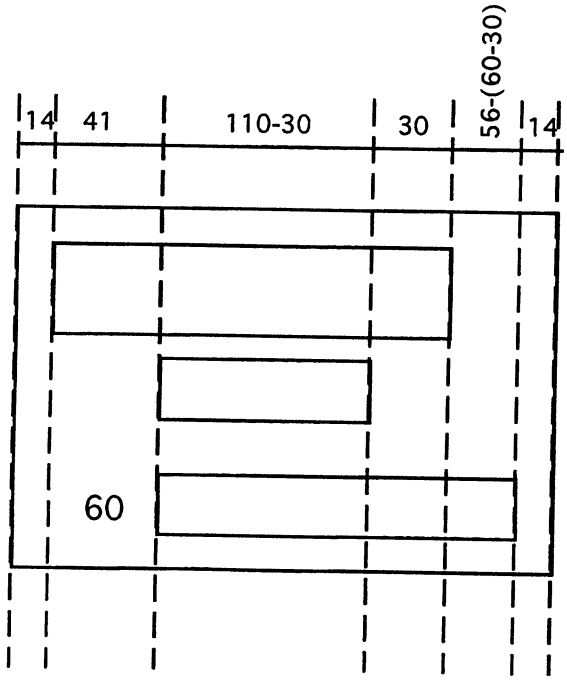
Le principe de cet algorithme est de contracter chaque boîte de la structure en partant des boîtes feuilles. Les *valeurs de contraction* de chaque boîte sont les quantités dont ses dimensions sont réduites.

L'algorithme donne donc des valeurs de contraction nulles (resp. maximales) aux boîtes feuilles visibles (resp. invisibles). A chaque pas, il découpe la boîte courante en bandes verticales et horizontales dont les bords coïncident avec les côtés de ses boîtes filles. Il examine la possibilité de **contracter** chaque bande, en commençant par les bandes qui recouvrent un maximum de boîtes, et en considérant les valeurs de contraction des boîtes recouvertes. Enfin, il repositionne et redimensionne les boîtes filles et détermine la valeur de contraction de la boîte courante.

Exemple de pas de contraction:

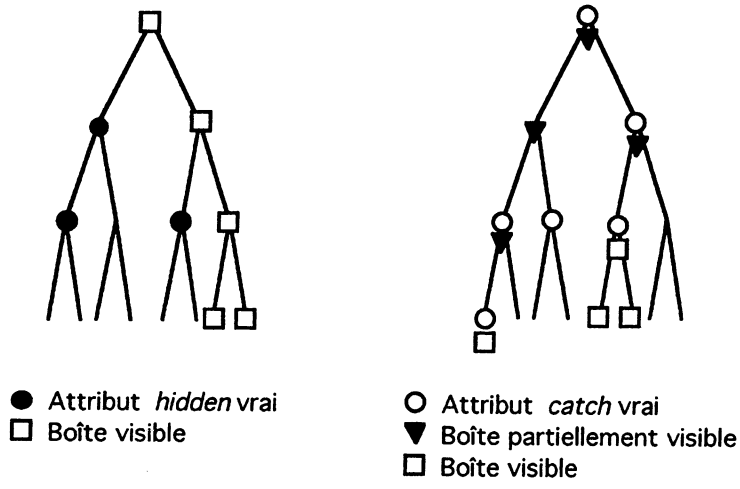


Avant contraction, les valeurs de contraction des trois boîtes filles sont: 30, 90 et 60.



Après contraction, la valeur de contraction de la boîte mère est 60.

Deux champs booléens, `hidden` et `catched`, associés à chaque boîte, mémorisent la vue hiérarchique choisie par l'utilisateur. Une *boîte hidden* (resp. *boîte catched*) est une boîte dont le champ hidden (resp. catched) est vrai. Toute sous-boîte boîte hidden est invisible, GLEF peut contracter la place qu'elle occupe. Toute boîte catched dont le nombre de boîtes catched englobantes est maximum est visible, GLEF peut contracter la place occupée par les parties de boîtes hors de ces boîtes.



8. Manipulations naturelles

Les *manipulations utilisateurs* de GLEF permettent de construire tous les termes du formalisme de définition. Il suffit d'employer directement les constructeurs du langage de définition ou d'appeler des entités évaluables "bien" définies. Mais cette construction n'est pas toujours naturelle.

Même si la notion de naturel est informelle, nous pouvons tenter de caractériser les constructions naturelles. Une manipulation utilisateur est *naturelle* ssi ce n'est pas un appel direct à un constructeur du formalisme de définition. Une construction est *naturelle* ssi elle est uniquement formée de manipulations naturelles et elle n'utilise aucune entité évaluable.

Une structure syntaxique (langage, calcul, etc.) peut être *éditée naturellement* ssi d'une part, on connaît une bijection compositionnelle entre cette structure et une définition du formalisme de définition [HHP89] et d'autre part, GLEF permet de construire les termes de cette définition de manière naturelle. Par *bijection compositionnelle* en entend que les entités de chacune des catégories syntaxiques de la structure soient en correspondance avec des termes déterminés de la définition, ainsi que les compositions de ces entités soient en correspondance avec les applications possibles de ces termes. Par exemple, l'ensemble des mots d'une langue naturelle peut être édité

naturellement car il est fini. Par contre, l'ensemble des phrases d'une langue naturelle ne peut pas être édité naturellement car on ne sait pas le caractériser, ni dans le formalisme de définition, ni dans aucun autre formalisme: Sa représentation par un langage naturel est généralement incomplète et sa représentation par une suite de lettres est incorrecte. Ainsi, même si GLEF contenait un traitement de textes moderne, cet ensemble ne pourrait pas être édité naturellement, selon notre définition.

GLEF doit permettre la construction naturelle de la classe de définitions la plus large possible. Les définitions qui représentent des preuves sont particulièrement importantes. Forcer l'utilisateur à programmer les manipulations naturelles correspondant à chaque système formel n'est pas souhaitable. GLEF propose donc un ensemble de manipulations naturelles fixe. L'environnement de définition, c'est à dire **le système formel et les théories utilisés** définissent l'ensemble des termes constructibles naturellement.

Cette section décrit les manipulations naturelles proposées par GLEF. Pour le moment, ces manipulations naturelles suffisent à construire naturellement les preuves des systèmes formels sans variables. La construction de preuves dans d'autres systèmes formels nécessite parfois l'utilisation de certains constructeurs du langage de définition. Par exemple, l'abstraction est employée pour construire des preuves en logique du premier ordre.

Bien entendu, nous envisageons de développer d'autres manipulations naturelles pour GLEF (voir chapitre Travail futur et conclusion section A.2). Notamment, une manipulation naturelle fondée sur un algorithme d'unification d'ordre supérieur permettrait de simplifer certaines constructions naturelles, **sans changer la classe des termes** constructibles naturellement.

a) Manipulations élémentaires

La programmation de l'effet de chaque manipulation naturelle est indépendante de celle de son interface utilisateur. Cela facilite l'écriture de manipulations naturelles d'effets similaires ou le changement de l'interface utilisateur d'une manipulation naturelle.

Les manipulations élémentaires servent à décrire les effets des manipulations naturelles. En fait, seules quelques manipulations élémentaires (1 ou 2) composent chaque effet. Notons aussi qu'elles sont très peu nombreuses.

Avant de formaliser les manipulations élémentaires, introduisons quelques définitions et notations utiles et décrivons les manipulations de boîtes employées. Notons que la formalisation emploie aussi les manipulations formelles (voir section B.3.d).

Dans la suite, on utilise la police Palatino pour mettre les noms (parfois déclinés dans le texte) des manipulations élémentaires en évidence.

Définitions et notations

GLEF permet à l'utilisateur de choisir l'environnement de définition et l'environnement de présentation dans lesquels il désire éditer les preuves. Quand une fenêtre d'édition est ouverte, ces environnements sont figés. Pour les changer l'utilisateur doit refermer toutes les fenêtres d'édition. Cette limitation permet de considérer les **mêmes environnements** dans toutes les fenêtres d'édition, et de **déplacer des termes** des unes vers les autres.

Quand une fenêtre d'édition est ouverte, l'environnement de définition choisi définit un *contexte d'édition* noté Γ_0. Les notions *d'entité*, de *variable pure* de *variable libre* et de *variable liée* sont définies à partir de ce contexte (voir section A.1.d).

GLEF peut considérer une référence $r=\bar{r}(\Gamma,t)$ dans une fenêtre d'édition si et seulement si Γ est un prolongement de Γ_0. Les *variables libres* d'une référence r sont celles de son contexte Γ. Rappelons que leur ensemble est $V_\Gamma^{\Gamma_0}$.

Considérons maintenant une structure de boîte de racine b_0. Soit b une boîte de cette structure, $\downarrow b$ dénote l'ensemble de ses boîtes filles et $\uparrow b$ dénote la boîte ancêtre de b appartenant à $\downarrow b_0$, quand b est différente de b_0.

La possibilité de présenter plusieurs termes dans une fenêtre d'édition est très utile: L'utilisateur peut construire un terme de manière non connexe, sans jongler avec différentes fenêtres. Des manipulations utilisateur peuvent aussi considérer plusieurs termes arguments, en même temps. Ainsi, b_0 n'est pas la présentation d'un terme unique. Mais les boîtes $\downarrow b_0$, appelées *boîtes principales*, sont les présentations de tous les termes considérés par l'utilisateur.

Une *boîte terme* est une boîte qui est la présentation d'un terme. Par exemple, les boîtes $\downarrow b_0$ sont des boîtes termes. Chaque boîte terme b_t mémorise la référence $\bar{r}(b_t)$ qui lui correspond, et l'occurrence généralisée $\bar{o}(b_t)$, de $\bar{r}(b_t)$ dans $\bar{r}(\uparrow b_t)$. Ces champs permettent les manipulations graphiques. Selon les manipulations effectuées par l'utilisateur, $\bar{r}(b_t)$ peut être réutilisée et $\bar{o}(b_t)$ peut être employée pour modifier $\uparrow b_t$.

On définit:

$$p_1(b_t) = |\bar{\gamma}(b_t)| - |\bar{\gamma}(\uparrow b_t)|$$

$$p_0(b_t) = |\bar{\gamma}(\uparrow b_t)| - |\Gamma_0|$$

$$p(b_t) = p_0(b_t) + p_1(b_t) = |\bar{\gamma}(b_t)| - |\Gamma_0|$$

Les *manipulations de boîtes* dont les profils suivent servent respectivement à ajouter, supprimer ou remplacer une boîte principale:

$$\text{box\_add} \in \text{box} \rightarrow \varnothing$$

$$\text{box\_delete} \in \text{box} \rightarrow \varnothing$$

$$\text{box\_replace} \in \text{box} \rightarrow \varnothing$$

L'algorithme de présentation peut être considéré comme une manipulation de boîte dont le profil suit:

$$\text{present} \in \text{reference} \rightarrow \text{box}$$

Notons que c'est une simplification car l'algorithme de présentation tient compte, en réalité, de la position de la boîte à construire, quand il présente des termes nommés.

Descrition des manipulations élémentaires

Finalement, nous pouvons décrire les manipulations élémentaires de manière formelle, comme suit:

L'ajout d'une référence r sert à ajouter la présentation de la référence r aux boîtes principales. Formellement:

$$\text{box\_add}(\text{present}(\text{simplify}(r)))$$

L'application d'une référence r à une boîte terme b_t sert à remplacer la boîte principale contenant b_t, par les présentations de ses instances les plus générales telles qu'à l'occurrence $\bar{o}(b_t)$, on trouve une instance de r. Pour cette manipulation, les références r et $\bar{r}(b_t)$ sont unifiées dans le contexte Γ_0. Les variables substituables sont $V^{\Gamma_0}_{\bar{\gamma}(r)}$ et $V^{\Gamma_0}_{\bar{\gamma}(\bar{r}(\uparrow b_t))} + p_{1(b_t)}$. Formellement:

$$\text{box\_replace}^*\left(\uparrow b_t, \text{present}^*\left(\text{simplify}^*\left(\text{substitute}^*\left(\bar{r}(\uparrow b_t), \pi_2\left(\text{unify}\left(r, \bar{r}(b_t), \Gamma_0, 0, p_1(b_t), 0\right)\right)\right)\right)\right)\right)$$

L'unification de deux boîtes termes b_s et b_t, telles que $\uparrow b_s = \uparrow b_t = b_p$, sert à remplacer b_p par les présentations de ses instances les plus

générales telles qu'aux occurrences $\bar{o}(b_s)$ et $\bar{o}(b_t)$ on ait respectivement une instance de b_s et b_t. Formellement:

$$\text{box\_replace}^*\left(b_p, \text{present}^*\left(\text{simplify}^*\left(\text{substitute}^*\left(\bar{r}(b_p), \pi_i\left(\text{unify}\left(\bar{r}(b_s), \bar{r}(b_t), \Gamma_0, p_1(b_s), p_1(b_t), p(b_p)\right)\right)\right)\right)\right)\right)$$

Avec, au choix, i=1 ou i=0.

Le remplacement d'une boîte terme b_t par une référence r sert à remplacer b_t par la présentation de r, au sein de la boîte $\uparrow b_t$. On suppose $\bar{\gamma}(b_t) = \bar{\gamma}(r)$. Rappelons que le remplacement peut échouer (voir section B.3.d et Annexe A). Formellement:

$$\text{box\_replace}\left(\uparrow b_t, \text{present}\left(\text{simplify}\left(\text{replace}\left(\bar{r}(\uparrow b_t), \bar{o}(b_t), r\right)\right)\right)\right)$$

La suppression d'une boîte principale b_p sert tout simplement à retirer b_p. Formellement:

$$\text{box\_delete}\left(b_p\right)$$

La suppression d'une boîte terme b_t (non principale) sert à remplacer b_t par la présentation d'une variable libre, au sein de la boîte $\uparrow b_t$. Formellement:

$$\text{box\_replace}\left(\uparrow b_t, \text{present}\left(\text{simplify}\left(\text{delete}\left(\bar{r}(\uparrow b_t), \bar{o}(b_t)\right)\right)\right)\right)$$

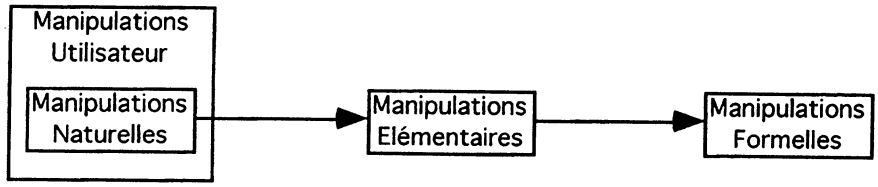
Le renommage d'une boîte terme b_t en un nom s sert à changer le nom de la variable dont b_t est la présentation, au sein de la boîte $\uparrow b_t$. Formellement:

$$\text{box\_replace}\left(\uparrow b_t, \text{present}\left(\text{simplify}\left(\text{rename}\left(\bar{r}(\uparrow b_t), \bar{o}(b_t), s\right)\right)\right)\right)$$

Le renommage a des effets secondaires sur la présentation, comme la description de la manipulation formelle correspondante l'indique (voir section B.3.d).

b) Manipulations naturelles

Les *manipulations naturelles* sont les manipulations effectivement proposées à l'utilisateur. Elle sont composées de manipulations élémentaires.



Hiérarchie des manipulations.

Choix termes et menus de termes

Un choix de menu peut être associé à une référence, on l'appelle alors *choix terme*. Un menu qui ne contient que des choix termes est appelé *menu de termes*. Quand la sélection est vide, l'appel d'un choix terme ajoute la référence associée. Quand une boîte terme est sélectionnée, cet appel lui applique la référence associée. Dans ce cas, seuls les choix dont les termes sont **unifiables** avec le terme de la boîte sélectionnée sont **accessibles**, les autres sont **estompés**.

Les menus de termes de GLEF sont les suivants:

Le menu **Entités** (**Entities**) contient un choix terme `x` pour chaque introduction de constante `[x:T]` ou nommage `[x=t]` du contexte de définition Γ_0. Notons que pour tout nommage `[x=t]` de Γ_0, il existe un type `T` qui vérifie le jugement de type $\Gamma_0 \vdash$ `t:T`. Ainsi, dans les deux cas (`[x:T]` et `[x=t]`) quand `T` est de la forme $[x_1:T_1]\ldots[x_n:T_n]t$, où `t` n'est pas un produit dépendant, la référence associée est $(\Gamma_0[v_1:T_1]\ldots[v_n:T_n],(x\ v_1\ \ldots\ v_n))$. Autrement dit, les v_i sont des variables libres de types T_i. Quand `T` est un ensemble de termes de cette forme, la référence associée est le multi-ensemble des termes de la forme $\langle v_1:T_1\rangle\ldots\langle v_n:T_n\rangle(x\ v_1\ \ldots\ v_n)$, considéré dans le contexte Γ_0. L'application systématique des variables libres aux entités utilisées **minimise les interactions** avec l'utilisateur, en lui permettant de construire des preuves par application et unification.

Le menu **Noms** (**Names**) contient un choix terme `x` associé à la référence (Γ_0,`t`) pour chaque nommage `[x=t]` de Γ_0. Notons que si le type `T` qui vérifie $\Gamma_0 \vdash$ `t:T` n'est pas un produit dépendant, les choix de termes `x` des menus **Entités** et **Noms** sont équivalents.

Les menus **Entités Locales** (**Local Entities**) et **Noms Locaux** (**Local Names**) sont similaires aux menus **Entités** et **Noms**, ils contiennent respectivement les entités et les nommages ajoutés par l'utilisateur en cours d'édition à Γ_0.

Déplacement

L'utilisateur peut déplacer une boîte terme b_s selectionnée vers une boîte terme b_d destination. Si $\uparrow b_s = \uparrow b_d$, les deux boîtes sont unifiées. Sinon, la référence de b_s est appliquée à b_d, comme pour un menu de termes. Pour déplacer la boîte, l'utilisateur clique dessus et maintient le bouton de la souris enfoncé, la boîte destination est déterminée par la position de la flèche, quand l'utilisateur relâche le bouton. Tant que le bouton est

enfoncé, les boîtes **destination possibles** sont **mises en évidence** au passage de la flèche.

Clavier

L'utilisateur peut renommer une variable sélectionnée en éditant simplement son nom au clavier. Après édition, le nom de chaque instance de la variable est changé.

Notons que les noms des variables dans les **autres** contextes structurellement égaux sont changés de manière **interne**, pas dans la **présentation**. Ainsi, ils n'apparaissent que dans les boîtes construites **après** le renommage.

Presse-papiers

Le *presse-papiers* est un objet qui existe dans la plupart des éditeurs. Il peut contenir une information du type l'information éditée. Les commandes usuelles **Couper/Copier/Coller/Effacer** servent à manipuler conjointement le contenu dy presse-papiers et la sélection du document édité. Ainsi, décrire le presse-papiers de GLEF revient à décrire son contenu et ces commandes.

Le presse-papiers de GLEF peut être vide, ou contenir une référence. Les commandes qui le concernent sont:

- **Couper** est un raccourci pour **Copier/Effacer**.
- **Copier** copie la référence d'une boîte terme sélectionnée dans le presse-papiers.
- **Coller** est un choix terme associé à la référence contenue dans le presse-papiers: Quand la sélection est vide, son appel ajoute la référence. Quand une boîte terme est sélectionnée, cet appel lui applique la référence. Contrairement, à l'édition de texte, **Coller** dépend de la sélection effectuée, il n'est pas équivalent à **Effacer/Coller**.
- **Effacer** supprime les boîtes sélectionnées. Les deux cas de suppression interviennent, selon si ces boîtes sont principales ou non. Notons que cette commande ne concerne pas véritablement le presse-papiers, mais elle lui est souvent associée.

A un instant donné, les choix des commandes applicables sont **accessibles**, les autres sont **estompés**.

Annulation

L'*annulation historique*[23] donne à l'utilisateur la possibilité de revenir à n'importe quel état précédent de la fenêtre d'édition et du presse-papiers.

Comme les références construites sont mémorisées de manière définitive, un pointeur peut représenter le contenu du presse-papiers, et une **liste de pointeurs** peut représenter celui de la fenêtre d'édition. Si l'annulation historique ne conservait que ces informations, elle serait très simple, rapide et solliciterait peu de mémoire. Mais elle doit aussi considérer la disposition visuelle de la fenêtre d'édition, l'état de la sélection et les noms des variables.

Poubelle et Photocopieuse

Deux icônes représentant une *poubelle* et une *photocopieuse* apparaissent au bas de chaque fenêtre d'édition.

Déplacer une sélection vers la poubelle provoque la suppression de ses boîtes. Comme pour le menu **Effacer**, dont l'effet est identique, les deux cas de suppression interviennent.

Déplacer une sélection vers la photocopieuse permet de la dupliquer, c'est à dire d'ajouter la référence associée à chaque boîte sélectionnée.

c) Dépendance entre édition et présentation

Quand une manipulation modifie une boîte, GLEF teste la "**compatibilité**" des occurrences, afin de préserver le mieux possible les paramètres de présentation `hidden` et `catched`, et les noms des variables.

9. Liaisons avec l'extérieur

GLEF sert non seulement à construire des preuves, mais aussi à visualiser et à manipuler des preuves existantes, obtenues par d'autres moyens. Cette section décrit les différentes méthodes de communication employées.

Écriture manuelle

L'utilisateur peut écrire **directement** des preuves dans le langage de définition. Mais ce langage est un λ-calcul ni simple, ni usuel.

Néanmoins, les systèmes formels sont souvent définis de cette manière. Le menu **Transformer** (**Transform**) donne accès aux constructeurs du langage de définition et permet bien de construire toutes les définitions.

---

[23]Par opposition à annulation "locale", qui correspond grosso-modo à l'opération effacer du presse-papiers [TBK92].

Mais ces constructions ne sont pas **naturelles** au sens de GLEF. Elles reviennent à employer directement le formalisme de définition.

Bien entendu, GLEF est utile pour **construire des preuves**, car les manipulations employées sont le plus souvent naturelles, et systématiquement vérifiées.

Récupération de preuves produites par un démonstrateur existant

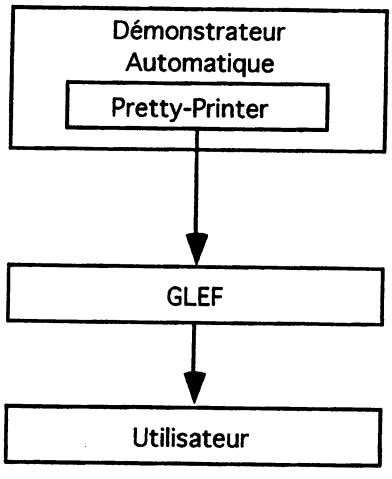
Souvent, la preuve à visualiser provient d'un outil d'inférence capable de produire des preuves (démonstrateur automatique, etc.). Cet outil d'inférence peut **exister indépendamment** de GLEF ou de ATINF.

Pour récupérer les preuves produites, on doit écrire un programme capable de les **traduire**. Deux méthodes sont possibles:

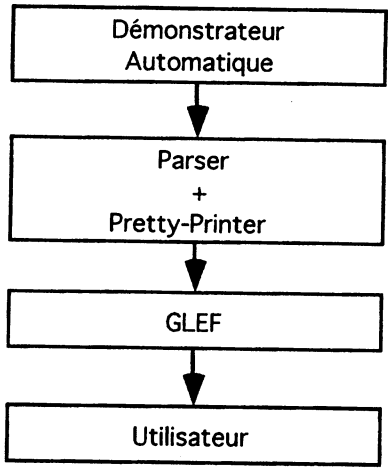
Quand on possède le code source de l'outil d'inférence, on peut directement lui ajouter une procédure d'exportation. Souvent, cette procédure est un simple "*pretty-printer*" qui exploite la structure interne des preuves, son "coût" de programmation est très faible. Sa taille est de l'ordre d'une centaine de lignes. Le temps nécessaire à sa programmation varie de quelques minutes à quelques heures, selon les connaissances du programmeur concernant le code source de l'outil d'inférence et le formalisme de définition.

Sinon, le travail à effectuer est d'une importance comparable. On doit écrire un programme qui traduit la sortie de l'outil d'inférence, sous forme d'écran ou de fichier, dans le formalisme de définition. Ce programme peut se composer d'un "*parser*", souvent facile à réaliser avec des outils comme *LEX* et *YACC*, et d'un "*pretty-printer*".

Quand la structure interne des preuves est trop complexe, la deuxième méthode est parfois plus simple à mettre en œuvre.



1er cas: On possède les sources du démonstrateur automatique



2ème cas: On ne possède pas ces sources ou la structure interne des preuves est trop complexe

Ensuite, le formalisme de définition permet de **définir** le système formel concerné, quand il n'a pas déjà été défini pour éditer des preuves manuellement ou pour utiliser un autre outil d'inférence. Le plus souvent, ce système formel est proche d'un système formel déjà défini. Il suffit alors **d'extraire** dans la définition du système formel déjà défini, la partie utile au système formel à définir, de **compléter** cette partie et de la **factoriser**. Notons que le formalisme de définition est bien adapté à la spécification de définitions par couches successives.

Bien entendu, on ne fait **qu'une seule fois** les opérations décrites pour chaque outil d'inférence intégré.

Liaison directe avec des outils d'inférence existants

La méthode de récupération de preuves précédente est très simple à mettre en œuvre, mais elle n'établit pas de véritable communication entre GLEF et l'outil d'inférence et, par conséquent, entre l'utilisateur et l'outil d'inférence.

Une autre méthode consiste à programmer **l'appel** de l'outil d'inférence dans une entité évaluable. Cette entité évaluable peut établir une **communication à double sens** entre GLEF et l'outil d'inférence. Comme toute entité du contexte de définition, elle est accessible à l'utilisateur.

Le réglage des **paramètres** de l'outil d'inférence peut être lui aussi programmé dans une entité évaluable. S'il est programmé en même temps que son appel, on peut le **fixer** d'avance, le **calculer** dynamiquement en fonction des arguments fournis à l'entité évaluable ou le déterminer par **interaction** avec l'utilisateur. Régler les paramètres de l'outil d'inférence avec une entité évaluable donne à GLEF l'opportunité d'**homogénéiser l'interface utilisateur des différents outils**. En effet, GLEF pourrait proposer des primitives standards pour définir le dialogue avec l'utilisateur, à l'instar de l'IUGG de la machine (voir chapitre Travail futur et conclusion section A.9).

Par exemple, **l'outil d'unification** employé par le calcul de résolution a été intégré de cette manière. Son utilisation est transparente pour l'utilisateur.

VII. Utilisation

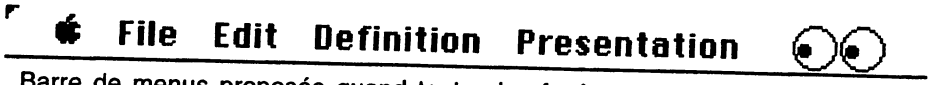
Introduction

Ce chapitre est une description de GLEF (Graphical Logical Edition Framework) vu par l'utilisateur. Il constitue un manuel utilisateur rudimentaire et incomplet.

La première partie décrit l'utilisation de GLEF. La deuxième partie montre comment spécifier de nouveaux systèmes formels et intégrer des outils externes. Enfin, la troisième partie regroupe les exemples développés jusqu'à présent.

A) Utilisation

Rappelons que la philosophie de GLEF est de permettre à un utilisateur de construire, visualiser, mémoriser, imprimer et modifier des preuves, voire des systèmes formels, de façon manuelle ou automatique, en vérifiant si possible chaque manipulation. Il réalise l'intégration d'ATINF (ATelier d'INFérence), en fédérant un ensemble évolutif de spécifications de systèmes formels et d'outils d'inférence.



Barre de menus proposée quand toutes les fenêtres d'édition sont fermées.

1. Manipulation des environnements

Avant d'éditer des preuves, l'utilisateur doit **choisir les environnements de définition** (qui spécifient le langage, le calcul, les théories et les théorèmes à considérer) **et de présentation** (qui spécifient les graphismes associés) dans lesquels l'édition doit se réaliser.

Un environnement est un contexte scindé en parties (voir chapitre Réalisation section B.3.e), mémorisées chacune dans un fichier. **Choisir un environnement, revient donc à choisir une liste ordonnée de fichiers.**

Les choix de l'environnement de définition et de l'environnement de présentation s'effectuent de manière indépendante. En pratique, on choisit un fichier de présentation par fichier de définition. Toutefois, pour chaque fichier de définition, on peut choisir parmi plusieurs

fichiers de présentation, ce qui permet de varier présentation des définitions (langages, calculs, théories, preuves) considérées.

Le menu **Définition** (**Definition**) sert à choisir l'environnement de définition. Il contient un choix **Ouvrir...** et une liste de choix correspondant à la liste des fichiers qui constituent l'environnement de définition choisi à un instant donné.

Le choix **Ouvrir...** (**Open...**) permet de prolonger l'environnement de définition. Il invite l'utilisateur à choisir un fichier de définition dans la mémoire externe et l'ajoute à la liste.

Les autres choix permettent de diminuer l'environnement de définition. Chacun de ces choix retire de la liste: le fichier qui lui correspond et tous les fichiers qui le suivent.

Le menu **Présentation** (**Presentation**) sert à choisir l'environnement de présentation de manière identique.

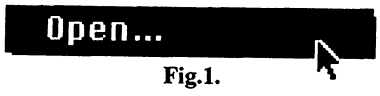


Fig.1.

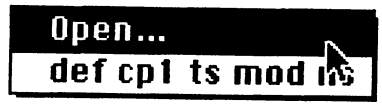


Fig.2.

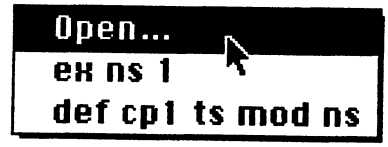


Fig.3.

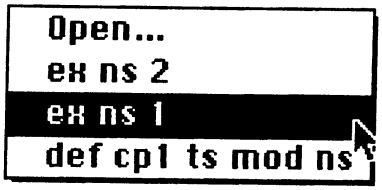


Fig.4.

Fig.5.

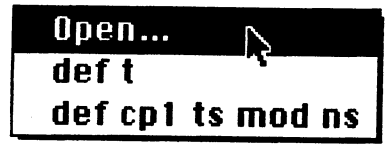


Fig.6.

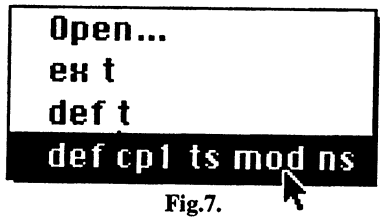


Fig.7.

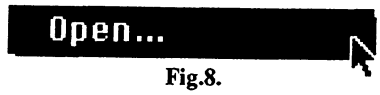


Fig.8.

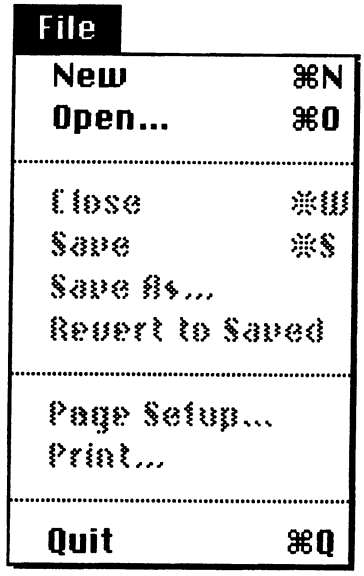
Exemple de manipulation du menu Définition:

1- Choisir le fichier "def cp1 ts mod ns"...
2- Choisir le fichier "ex ns 1"...
3- Choisir le fichier "ex ns 2"...
4- Eliminer les fichiers "ex ns 1" et "ex ns 2".
5- Choisir le fichier "def t"...
6- Choisir le fichier "ex t"...
7- Eliminer tout l'environnement.
8- Choisir un autre fichier... etc..

En plus des systèmes formels et des théories, le menu **Définition** permet de charger les **preuves à réutiliser**. Pour réutiliser une preuve, on ne considère que son nom, ses prémisses et sa conclusion, qui définissent une règle dérivée. Notons qu'une preuve chargée dans l'environnement de définition peut être visualisée, mais pas modifiée.

Le menu **Fichier** (**File**) est un menu standard de gestion de documents. Les environnements de définition et de présentation étant déterminés, il permet d'ouvrir des fenêtres d'édition vides, ou contenant une preuve existante, choisie par l'utilisateur, dans la mémoire externe. Quand la création ou la modification est terminée, il permet de l'imprimer ou de la sauvegarder.

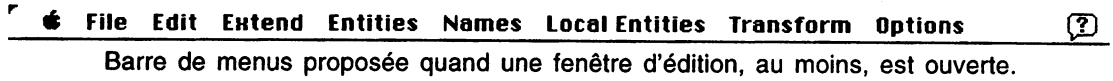
Notons que les environnements de définition et de présentation ne peuvent plus être modifiés, tant qu'une fenêtre d'édition est ouverte. Toutes les fenêtres d'édition considèrent donc les mêmes environnements, et on peut **copier** des parties de preuves d'une fenêtre d'édition vers une autre.



Menu standard de gestion de documents du Macintosh

2. Manipulation des boîtes

Quand une *fenêtre d'édition* est ouverte, le contexte défini par l'environnement de définition choisi est appelé *contexte d'édition* et noté Γ_0 et l'écran est surmonté d'une nouvelle barre de menus:



Barre de menus proposée quand une fenêtre d'édition, au moins, est ouverte.

Chaque fenêtre d'édition est elle-même surmontée de *boutons* et deux *icônes* apparaissent dans son angle inférieur droit. Elle peut contenir les présentations de plusieurs termes.

Boîtes

Un algorithme de construction de boîtes traduit une définition en boîtes, en suivant une spécification dans le langage de présentation. Une *boîte* est un rectangle invisible qui peut contenir un élément graphique primitif (symbole, texte, image) ou d'autres boîtes. Chaque terme est

présenté dans une boîte, appelée *boîte terme*, qui est la racine d'un arbre contenant d'autres boîtes. Dans cet arbre, certaines boîtes présentent ses sous-termes. Une *boîte principale* est une boîte qui n'est incluse dans aucune autre boîte. Une fenêtre d'édition peut contenir plusieurs boîtes principales.

Variables libres

Une *variable libre* est une variable qui apparaît dans le terme associé à une boîte principale, mais qui n'est introduite ni par Γ_0 ni dans le terme. GLEF implémente les variables libres en attachant à chaque boîte principale un prolongement de Γ_0 qui introduit les variables libres. Dans les fenêtres d'édition, les variables libres sont présentées en italiques.

Sélection

Dans une fenêtre d'édition, on peut indiquer un ensemble de boîtes à GLEF. Cet ensemble de boîtes, appelé *sélection*, sert d'argument aux manipulations. Pour indiquer quelle est la sélection, GLEF met ses boîtes **en évidence**. Notons que seules les boîtes associées à des **termes**, appelées boîtes termes, peuvent être sélectionnées.

Pour sélectionner une boîte, il suffit de cliquer dessus. En fait, GLEF sélectionne **la plus petite boîte** qui contient le point cliqué.

Cliquer hors de la sélection, vide celle-ci, sauf si la touche *Majuscule* est enfoncée. Cette touche permet donc la sélection de plusieurs boîtes.

Cliquer dans la sélection n'a pas d'effet, sauf si la touche Majuscule est enfoncée. Dans ce cas, la boîte plus petite boîte terme contenant la boîte sélectionnée qui contient le point cliqué remplace cette boîte dans la sélection.

Cliquer dans la sélection en maintenant le bouton de la souris enfoncé permet de la *déplacer* vers une destination graphique (boîte, icône), déterminée par la position de la flèche quand on relâche le bouton.

Utilisation des Boutons de visualisation

Le plus souvent, on ne souhaite pas visualiser complètement une preuve formelle, surtout si celle-ci est de taille importante[24]. Dans une première approche, on souhaite plutôt faire abstraction des détails, quitte à les examiner ensuite.

GLEF permet donc d'examiner les preuves de manière **hiérarchique**, avec différents niveaux de détail. Lorsqu'on visualise une preuve, on ne

[24]Une preuve formelle peut avoir quelques milliers de formules [Lam91], [Hor93].

voit d'abord que les formules constituant sa conclusion et ses hypothèses. Si la déduction de la conclusion à partir des hypothèses ne paraît pas évidente, cinq **boutons** permettent d'en faire un examen hiérarchique naturel[25]. Ces cinq boutons correspondent à cinq commandes qui, sauf la dernière, nécessitent une sélection préliminaire.

Le bouton **Montrer** (**Show**) montre chaque sous-boîte des boîtes sélectionnées, dont au plus une boîte ancêtre est cachée et possède l'attribut HIDE, défini dans le langage de présentation.

Le bouton **Tout Montrer** (**Show All**) montre toutes les sous-boîtes des boîtes sélectionnées. Son appel correspond à des appels successifs de **Show**, terminés lorsque toutes les sous-boîtes des boîtes sélectionnées sont visibles.

Le bouton **Cacher** (**Hide**) cache chaque sous-boîte des boîtes sélectionnées, qui possède l'attribut HIDE. Bien entendu, les sous-boîtes de boîtes cachées sont cachées.

Le bouton **Attraper** (**Catch**) sert à examiner la sélection, en cachant le reste de la preuve, c'est à dire les boîtes et les parties de boîtes hors de la sélection.

Le bouton **Relâcher** (**Release**) annule le dernier appel de **Attraper**. Des appels successifs de **Relâcher** annulent des appels successifs de **Attraper**. Quand tous les appels de **Attraper** sont annulés, les boîtes principales sont visibles.

On considère une structure de boîtes dont toutes les boîtes peuvent être sélectionnées et possèdent l'attribut HIDE. Ces boutons permettent de visualiser toute partie de cette structure qui vérifie les conditions suivantes:

- Les sous-arbres de mêmes racines que ses sous-parties connexes maximales sont disjoints.
- Si elle contient une boîte, elle contient toutes ses boîtes sœurs.

Souvent, on considère une structure de boîtes dont les boîtes possédant l'attribut HIDE sont les prémisses des pas d'inférence. Ces boutons permettent de visualiser toute partie de l'arbre de déduction dont les sous-parties connexes maximales sont indépendantes du point de vue de la déduction et telle que les prémisses de chaque pas d'inférence soient visibles ou cachées en même temps.

Pour visualiser une telle partie, il suffit de sélectionner les racines de toutes ses sous-parties connexes maximales, de cliquer sur **Attraper**, de sélectionner toutes les feuilles de ces sous-parties et de cliquer sur **Cacher**.

---

[25]On retrouve certains de ces boutons sous des formes différentes, dans les traitement de texte et les traitement "d'idées" (WORD5, MORE, etc.).

Visualisation des nommages

On peut associer une parure (présentation élémentaire) à tout nommage `[x=t]` d'une définition. Cette parure est alors utilisée dans la présentation des instances du nommage, à la place de la présentation du terme nommé (ou de son application, quand il s'agit une abstraction).

Par exemple, quand plusieurs parties de preuves sont identiques, un nommage muni d'une parure permet de **factoriser leurs présentations**. La parure peut afficher le nom et les arguments du terme nommé à la place de la présentation de chaque partie.

On visualise **séparément** le terme nommé, quand il est associé à une boîte principale. Ainsi, si on **attrape une instance de nommage** avec le bouton **Attraper**, on visualise **le terme nommé**. Notons que la commande **Extraire** et les menus **Noms** et **Noms Locaux** (décrits juste après) ont un effet similaire.

3. Manipulation de la preuve

Les différents menus

Les menus de termes

Dans les nouveaux menus proposés lors de l'ouverture d'une fenêtre d'édition, certains choix, appelés *choix termes*, sont associés à des références (une référence est un couple formé d'un contexte et d'un terme). Les menus qui ne contiennent que des choix termes sont appelés *menus de termes*.

Quand la sélection est vide, l'appel d'un choix terme provoque l'ajout de la présentation du terme associé dans la fenêtre d'édition (voir ajout chapitre Réalisation section B.8.a). Quand une boîte terme est sélectionnée, seuls sont accessibles les choix termes dont le terme associé est unifiable avec la sélection, les autres sont estompés. L'appel de l'un de ces choix provoque **l'instanciation** du terme de la boîte principale qui contient la boîte sélectionnée, pour qu'à l'occurrence déterminée par la sélection, on obtienne une instance du terme associé au choix (voir application chapitre Réalisation section B.8.a).

Le menu **Entités (Entities)** contient un choix terme `x` pour chaque introduction de constante `[x:T]` ou nommage `[x=t]` du contexte de définition Γ_0. Notons que pour tout nommage `[x=t]` de Γ_0, il existe un type `T` qui vérifie le jugement de type $\Gamma_0 \vdash$ `t:T`. Ainsi, dans les deux cas (`[x:T]` et `[x=t]`) quand `T` est de la forme $[x_1:T_1]\ldots[x_n:T_n]t$, où `t` n'est pas un produit dépendant, la référence associée est $(\Gamma_0[v_1:T_1]\ldots[v_n:T_n],(x\ v_1\ \ldots\ v_n))$. Autrement dit, les v_i sont des variables libres de types T_i. Quand `T` est un ensemble de termes de cette forme, la référence associée est le multi-ensemble des termes de la forme $\langle v_1:T_1\rangle\ldots\langle v_n:T_n\rangle(x\ v_1\ \ldots\ v_n)$, considéré dans

le contexte Γ_0. L'application systématique des variables libres aux entités utilisées **minimise les interactions** avec l'utilisateur, en lui permettant de construire des preuves par application et unification.

Le menu **Noms (Names)** contient un choix terme `x` associé à la référence (Γ_0,t) pour chaque nommage `[x=t]` de Γ_0. Notons que si le type `T` qui vérifie $\Gamma_0 \vdash t:T$ n'est pas un produit dépendant, les choix de termes `x` des menus **Entités** et **Noms** sont équivalents.

Les menus **Entités Locales (Local Entities)** et **Noms Locaux (Local Names)** sont similaires aux menus **Entités** et **Noms**, ils contiennent respectivement les entités et les nommages ajoutés par l'utilisateur en cours d'édition à Γ_0.

Le menu Étendre

Parfois, pendant la construction d'une preuve, l'utilisateur a besoin d'étendre le contexte de définition. Par exemple, une définition de la logique du premier ordre peut autoriser l'utilisateur à ajouter de nouvelles constantes, fonctions ou prédicats pour construire la formule à démontrer.

Bien entendu, l'utilisateur pourrait introduire les nouvelles entités au niveau des termes qui les utilisent, mais modifier directement le contexte de définition est beaucoup plus **pratique**.

Ainsi, le menu **Étendre (Extend)** permet de prolonger le contexte de définition sans analyser de nouvelle définition. GLEF demande les noms des entités introduites et les ajoute aux menus **Entités Locales** et **Noms Locaux.**

Le choix **Ajouter une Entité... (Add Entity...)** prolonge Γ_0 par l'introduction de constante `[x:T]` où le type `T` est la sélection et `x` est demandé à l'utilisateur. Son utilisation est pratique pour construire des systèmes formels, mais quand on l'utilise pour construire des preuves, on doit ensuite prouver l'existence d'un terme de type `T` (voir chapitre Travail futur et Conclusion section A.12).

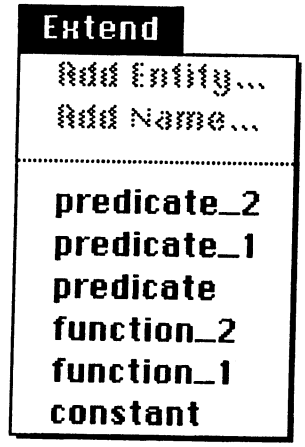
Le choix **Ajouter un Nom... (Add Name...)** prolonge Γ_0 par le nommage `[x=<`x_1`:`T_1`>...<`x_n`:`T_n`>t]` où le terme élémentaire `t` est la sélection, x_i sont les variables libres de `t`, de types T_i et `x` est demandé à l'utilisateur. Par exemple, le terme nommé peut représenter un lemme que l'on souhaite réutiliser plus tard.

Grâce aux contextes de la forme `[x=pool:T]`, une définition peut autoriser l'utilisateur à ajouter à Γ_0, des constantes de types `T`, pendant l'édition. Ainsi le menu **Étendre** contient aussi un choix pour chaque pool `[x=pool:T]` de Γ_0. Ce choix permet de prolonger Γ_0 par l'introduction de constante `[x:T]` où `x` est demandé à l'utilisateur, il agit ensuite comme si `x` était choisi dans le menu **Entités**. Ces choix sont très utiles pour construire des preuves. Par exemple, ils servent à ajouter les nouvelles

constantes, fonctions ou prédicats. De plus, on peut munir chaque type d'une parure spécifiant la **présentation** de chaque entité introduite de cette manière.

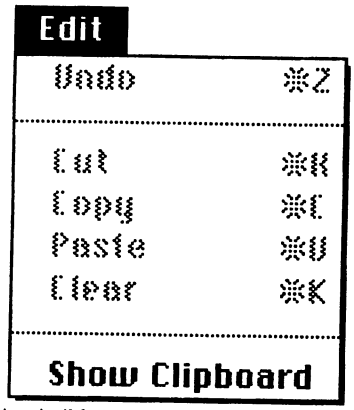
Pour prolonger le contexte de définition après l'appel des deux premiers choix, GLEF doit recaler toutes les références (même celle du presse-papiers), ce qui peut être assez long. Notons que le recalage devrait être beaucoup plus rapide dans les futures versions de GLEF (voir chapitre Travail futur et Conclusion section A.4).

Avec les autres choix, l'ajout d'entités est **immédiat**, car GLEF prévoit leurs places en créant d'avance des "trous" dans le contexte de définition.



Le menu Édition

Le menu **Édition** (**Edit**) est un menu standard d'édition de documents.



Menu standard d'édition de documents du Macintosh

Le choix **Annuler** (**Undo**) annule la dernière modification de la fenêtre d'édition considérée.

Les choix **Couper, Copier, Coller** (**Cut, Copy, Paste**) correspondent aux manipulations standards du *presse-papiers*, qui peut contenir une référence.

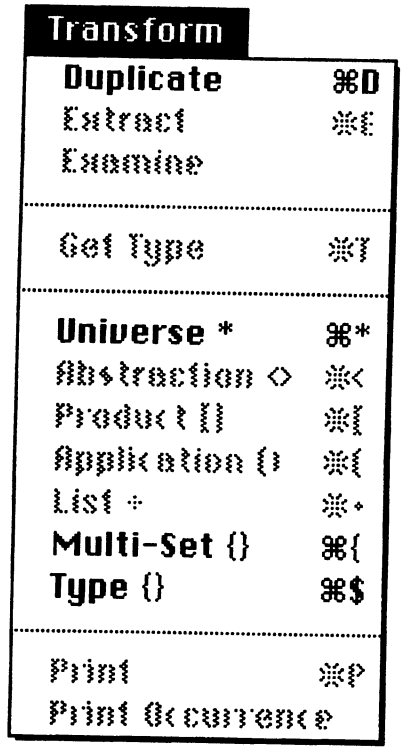
- **Couper** est un raccourci pour **Copier/Effacer.**
- **Copier** copie la référence d'une boîte terme sélectionnée dans le presse-papiers.
- **Coller** est un choix terme associé à la référence contenue dans le presse-papiers.

Le choix **Effacer** (**Clear**) supprime les boîtes de la sélection (voir suppression chapitre Réalisation section B.8.a). Quand la boîte sélectionnée est une boîte principale (boîte de plus haut niveau), elle est simplement retirée. Quand c'est une sous-boîte, le terme de la boîte principale qui la contient est modifié pour qu'à l'occurrence déterminée par la sélection, on ait une variable libre. En λ-calcul typé, la formalisation de cette manipulation n'est pas très simple, notons qu'elle peut modifier le terme "en dehors" de la boîte sélectionnée.

Contrairement à l'édition de texte, la suite de commandes **Couper/Coller** ne correspond **pas** toujours à la commande **Copier**, car elle ne laisse pas toujours la fenêtre d'édition inchangée (voir chapitre Réalisation section B.8.a).

Le menu Transformations

Le menu **Transformations** (**Transform**) est un complément du menu **Édition.**



Le premier groupe de choix de ce menu correspond aux manipulations utilitaires suivantes:

- **Dupliquer** (**Duplicate**) duplique la sélection en ajoutant une nouvelle présentation pour chaque terme sélectionné.
- **Extraire** (**Extract**) remplace les boîtes principales contenant les boîtes sélectionnées, par ces boîtes sélectionnées.
- **Examiner** (**Examine**) ajoute la présentation par défaut des termes sélectionnés, quand leurs présentations ne donnent pas toutes les informations.

Le deuxième groupe de choix permet d'accéder au calcul de types:

- **Obtenir le Type** (**Get Type**) ajoute les présentations des types des termes sélectionnés.

Le troisième groupe de choix correspond aux constructeurs du formalisme de définition, ces choix ajoutent les présentations des termes qu'ils produisent:

- **Univers** (**Universe**) produit le terme *.
- **Abstraction** (**Abstraction**) lie par des abstractions les variables libres sélectionnées dans les termes des boîtes principales qui les contiennent.
- **Produit** (**Product**) est similaire à **Abstraction**, mais la fermeture emploie le produit dépendant.
- **Application** (**Application**) construit les applications ayant pour fonction les termes sélectionnés et pour arguments de nouvelles variables libres.
- **Liste** (**List**) construit un type de multi-ensemble à partir d'un type sélectionné ou de termes sélectionnés pouvant être regroupés dans un type ensemble.
- **Multi-Ensemble** (**Multi-Set**) construit le multi-ensemble des termes sélectionnés.
- **Type** (**Type**) construit un type à partir de termes pouvant former un type.

Le dernier groupe de choix donne des informations au sujet du contenu des boîtes sélectionnées:

- **Imprimer** (**Print**) écrit les termes sélectionnés dans la fenêtre texte de GLEF en employant le langage de définition.
- **Imprimer l'Occurrence** (**Print Occurrence**) écrit les occurrences généralisées des termes sélectionnés par rapport aux termes des boîtes principales qui contiennent leurs boîtes, dans la fenêtre texte.

Déplacements

On peut cliquer sur la sélection, en maintenant le bouton de la souris enfoncé, pour la déplacer vers une destination graphique. Quand on relâche le bouton, la position de la flèche détermine la destination graphique.

Quand la sélection et la destination sont des boîtes termes, le terme sélectionné est **unifié** avec celui de la destination. Si la boîte destination ne fait pas partie de la même boîte principale que la sélection, le terme de la boîte principale qui contient la destination est instancié, pour qu'à l'occurrence déterminée par la destination, on ait une instance du terme sélectionné. L'effet est **identique** à celui d'un choix de menu de termes qui serait associé au terme sélectionné et appelé après avoir sélectionné la destination (voir application chapitre Réalisation section B.8.a). Sinon, le terme de la boîte principale qui contient les deux boîtes est instancié, pour qu'aux occurrences déterminées par chaque boîte, on ait une instance du terme associé à l'autre (voir unification chapitre Réalisation section B.8.a). Ainsi, les **variables** substituables considérées dans l'unification **dépendent** du type de déplacement.

Quand la destination est la *photocopieuse*, (resp. la *poubelle*) le choix **Dupliquer** (resp. **Effacer**) du menu **Transformations** (resp. **Edition**) est appliqué à la sélection.

Tant que le bouton de la souris reste enfoncé, les **destinations** graphiques possibles sont **mises en évidence** au passage de la flèche. Par exemple, une boîte terme est mise en évidence quand la sélection est un singleton et son terme est **unifiable** avec celui de la destination.

Clavier

Pour renommer une variable, il suffit de la sélectionner et d'entrer directement son nom au clavier. Notons que changer le nom d'une variable agit sur la représentation interne de certains termes (voir renommage chapitre Réalisation section B.8.a).

B) Extension de GLEF

1. Spécification d'un système formel

a) Définition

La spécification d'un système formel commence par sa définition dans le formalisme de définition. Le plus souvent, ce système formel est proche d'un système formel déjà défini. Il suffit alors **d'extraire** dans la définition du système formel déjà défini, la partie utile au système formel à définir, de **compléter** cette partie et de la **factoriser**. Notons

que le formalisme de définition est bien adapté à la spécification de définitions par couches successives.

Par exemple, si on désire définir le langage de la logique modale du premier ordre et on possède une définition d'un système formel fondé sur la résolution pour la logique du premier ordre. On peut **extraire** le langage des prédicats de la définition, le **compléter** avec les connectifs logiques modaux pour obtenir la définition recherchée, et le **factoriser** dans les deux définitions.

b) Présentation

Ensuite, le langage de présentation permet de spécifier la présentation de chaque entité définie dans le formalisme de définition.

Pour **faciliter la gestion** des environnements de définition et de présentation, on doit maintenir une certaine **cohérence** entre la structure des fichiers de définition et celle des fichiers de présentation.

Par exemple, un fichier de définition et un seul pourrait correspondre à un fichier de présentation. Toutefois, **plusieurs** fichiers de présentation pourraient correspondre à un seul fichier de définition. Notons que les fichiers contenant les parures par défaut, les parures utilitaires et les définitions des symboles usuels, proposés par GLEF, ne correspondent à aucun fichier de définition.

Symboles

Un *symbole* est un caractère qui n'appartient pas au jeu de caractères standard ASCII. Toutefois, il peut apparaître dans des polices de caractères spécifiques.

Ainsi, un symbole se caractérise par un nom de police et un indice dans cette police. Pour assurer la portabilité, considérer les symboles des **polices Postscript** est préférable, quand c'est possible.

Le langage de présentation permet de *définir des symboles*, c'est à dire de nommer leurs caractérisations. Contrairement aux caractères ASCII, il ne permet pas de changer la police d'un symbole, dans une présentation.

Exemples:

```
ASSIGN and symbol 217;
ASSIGN alpha symbol \a;
```

Notons que `symbol` est le nom d'une police Postscript.

Images

Une *image* est une représentation visuelle, fixée d'avance, que le langage de présentation ne peut pas générer. Par exemple, les flèches

d'un arbre binaire ou un symbole qui n'apparaît dans aucune police connue sont des images. Nous ne considérons que des représentations visuelles fixées d'avance, la construction dynamique de représentations visuelles est présentée juste après.

Pour assurer la portabilité, on devrait décrire les images dans le langage **Postscript**, et mémoriser ces descriptions dans les fichiers de présentation ou des fichiers séparés. Cependant, une telle méthode nécessite l'emploi d'un interprète Postscript. Bien que standards, ces interprètes ne sont pas encore très répandus.

Pour le moment, on emploie le format de description d'images, propre au Macintosh. Cette réalisation n'est pas portable, du moins, pas avant la disponibilité de BEDROCK (voir chapitre Réalisation section B.1), mais elle est très pratique, et son coût de programmation a été quasiment nul. Ajouter une image à un fichier de présentation est très simple, il suffit de la dessiner avec un logiciel de dessin quelconque, de la copier dans le presse-papiers, de la coller dans le fichier de présentation avec l'utilitaire ResEdit, et de la déclarer dans ce fichier. Pour que langage de présentation puisse la reconnaître, on doit lui donner un nom lors de son insertion dans le fichier.

Exemple de déclaration d'une image dans le langage de présentation:

```
PICTURE 'Théorème de Thalès';
```

2. Entités évaluables et présentations externes

Une *entité évaluable* est une entité qui, lorsqu'on l'applique à certains arguments, s'évalue en un terme de même type que l'application.

Une *présentation externe* est une image calculée dynamiquement, en fonction du terme courant.

Pour définir une entité évaluable ou une présentation externe, on doit lui associer une fonction. Cette fonction peut manipuler les données internes du formalisme de définition et, dans le cas d'une présentation externe, celles du langage graphique.

Pour le moment, on ajoute cette fonction directement au code de GLEF. Cette réalisation est portable, mais on doit recompiler une partie de GLEF.

Plus tard, cette fonction sera **incluse dans un fichier** de définition ou de présentation. Le format standard de description de fonctions externes du Macintosh permettrait de réaliser cette inclusion, comme pour les images. Mais pour assurer la portabilité, employer un interprète auxiliaire serait préférable.

Notons que dans les produits existants, l'ajout dynamique de fonctions n'est pas portable, sauf dans les environnements de haut niveau comme LISP, qui permet la méta-programmation.

3. Liaison aux outils externes

Pour lier un outil de production de preuves à GLEF, la méthode la plus **rapide** et la plus simple est d'écrire un programme traduisant les preuves produites dans le formalisme de définition, et de définir le système formel associé (voir chapitre Réalisation section B.9).

Ensuite, on peut écrire des entités évaluables pour contrôler l'appel de l'outil et régler ses paramètres. Ces entités évaluables améliorent **l'ergonomie**. Notons que GLEF peut intégrer directement un outil dont le code est suffisamment court pour constituer une entité évaluable (unification, etc.).

D'ailleurs, écrire une entité évaluable est nécessaire pour lier des outils de transformations de preuves, qui prennent des termes pour arguments.

C) Exemples

Cette partie décrit les différents exemples développés jusqu'à présent avec GLEF. Cette description met en évidence les points délicats de chaque exemple.

1. Logique propositionnelle

Pour la logique propositionnelle, nous avons défini deux calculs représentés par un système de Hilbert et un système de Gentzen. La présentation choisie pour les preuves correspond à une présentation usuelle à chaque système.

a) Langage propositionnel (PL)

Tout d'abord, la définition et la présentation du langage propositionnel qui sont, bien entendu, employées par les deux calculs, ont été **factorisées**.

Définition

Un type représente la catégorie syntaxique des formules:

```
[formula:                    *]
```

Contrairement à LF [HHP89], le type des jugements de base n'est pas `*`, mais un type `jugement`. Ce choix rend les définitions plus lisibles. Bien que la notion de jugement de base concerne toutes les logiques et n'appartient pas au langage, l'introduire au niveau du langage est plus économique que la factoriser avec la primitive `#include`.

```
[judgement:                  *]
```

Une entité représente chaque constructeur du langage. Notons que définir certains constructeurs à partir des autres permet de compléter un calcul qui les ignore.

```
[true,false:                [formula]
[not:                       [formula]formula]
[and,or,implies,equiv:      [formula][formula]formula]
```

Présentation

Le fichier de présentation inclut d'abord les fichiers de présentation standard, qui contiennent les parures prédéfinies:

```
#include ":Sources:utilities.pres"
#include ":Sources:standard.pres"
```

Et les symboles prédéfinis:

```
#include ":Sources:symbols.pres"
```

Ensuite, il définit une parure pour chaque entité de la définition. Cette parure consiste à placer la présentation de chaque argument de l'entité dans la boîte qui la présente.

```
not:BEGIN
    PRESENT:box;
    SYMBOL 'not':BEGIN
          PRESENT:inbox;
    END;
    PLACE A.1:BEGIN
          PRESENT:line_t;
    END;
END;
and,or,implies,equiv:BEGIN
    PRESENT:box;
    TEXT '(':BEGIN
          PRESENT:inbox;
    END;
    PLACE A.1:BEGIN
          PRESENT:line_t;
    END;
    SYMBOL NAME:BEGIN
          PRESENT:line_t;
    END;
    PLACE A.2:BEGIN
          PRESENT:line_t;
    END;
    TEXT ')':BEGIN
          PRESENT:line_t;
    END;
END;
```

Quand la parure d'une entité est omise, sa présentation par défaut est une boîte qui ne contient que son nom. Ainsi, présenter les entités de type non fonctionnel, comme les catégories syntaxiques, `true` et `false` est souvent inutile. Leur parure par défaut équivaut à:

```
formula,judgement,true,false:BEGIN
   PRESENT:st_constant;
END;
```

b) Un système de Hilbert (HS)

Le premier calcul est décrit **intégralement**, il peut servir d'exemple pour spécifier d'autres calculs, dans d'autres langages. Son axiomatisation correspond à un système de Hilbert pour la logique propositionnelle que l'on peut trouver par exemple dans [Men64].

Schémas d'axiomes:

a1: $A \Rightarrow (B \Rightarrow A)$

a2: $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

a3: $(\neg B \Rightarrow \neg A) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B)$

Règle d'inférence:

modus-ponens: $\dfrac{B \quad B \Rightarrow A}{A}$

Définition

La définition du calcul est fondée sur celle du langage propositionnel, mais elle **n'utilise pas** tous les **connectifs logiques**. Un langage de structuration de théories permettrait de masquer les connectifs logiques inutiles. En attendant, on peut les **redéfinir** à partir des autres connectifs logiques:

```
[or=        <a,b:formula>((implies (not a) b)]
[and=       <a,b:formula>(not (or a b))]
[equiv=     <a,b:formula>
                (and (implies a b) (implies b a))]
```

La forme de jugement de base `True` indique si une formule est un théorème (voir chapitre Définition section A). C'est une fonction de `formula` vers `judgement`, le type des jugements de base:

```
[True:          [formula]judgement]
```

Une formule `f` est un *théorème* ssi il existe un terme `p` de type `(True f).p` s'appelle une *preuve* de `f`.

Un **axiome** d'un système de Hilbert est caractérisé par un ensemble de jugements de bases, paramétré par des formules. Une fonction qui admet des formules pour arguments suffit à représenter cet ensemble:

```
[a1:    [a,b:formula]
            (True (implies a (implies b a)))]
```

```
[a2:      [a,b,c:formula]
               (True (implies
                     (implies a (implies b c))
                     (implies (implies a b) (implies a c))))]
[a3:      [a,b:formula]
               (True (implies
                     (implies (not b) (not a))
                     (implies (implies (not b) a) b)))]
```

Une **règle d'inférence** d'un système de Hilbert est caractérisée par une relation entre des jugements de base, paramétrée par des formules. Une fonction qui admet des formules et des jugements de base (prémisses) comme arguments suffit à représenter cette relation:

```
[modus_ponens: [a,b:formula]
                              [(True a)][(True (implies a b))]
                                    (True b)]
```

Finalement, la spécification peut autoriser l'ajout d'entités de types donnés au contexte de définition, par exemple, de nouvelles propositions:

```
[proposition=pool:formula]
```

Présentation

La présentation choisie pour les preuves montre la conclusion de chaque pas d'inférence. L'indentation de ces conclusions met la structure de chaque preuve en évidence. Quand on visualise une preuve, on ne voit d'abord que sa conclusion et ses hypothèses. Ensuite, on peut développer un par un ses pas d'inférence.

Pour indiquer ce choix à GLEF, on doit écrire une parure pour chaque entité introduite dans la définition du calcul:

Pour la forme de jugement de base:

```
True:BEGIN
   PRESENT:st_constant;
   PLACE A.1:BEGIN
         PRESENT:line_t_space;
   END;
END;
```

Pour les axiomes:

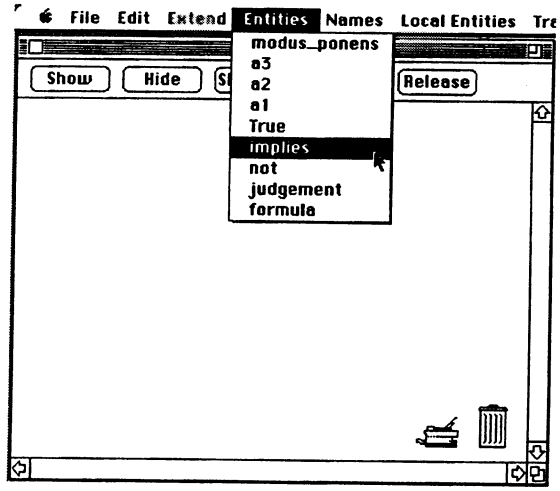
```
a1,a2,a3:BEGIN
   PRESENT:box;
   GRAPHICS:'CF';
   PLACE TYPE1:BEGIN
         PRESENT:inbox;
   END;
END;
```

Pour la règle d'inférence, notons l'utilisation de l'instruction HIDE pour masquer les prémisses:

```
modus_ponens:BEGIN
    PRESENT:frame;
    GRAPHICS:'F';
    PLACE TYPE1:BEGIN
          PRESENT:inframe;
    END;
    PLACE A.3:BEGIN
          HIDE;
          PRESENT:indent;
    END;
    PLACE A.4:BEGIN
          HIDE;
          PRESENT:column_1;
    END;
END;
```

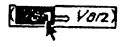
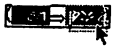
Construction d'une preuve

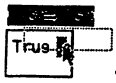
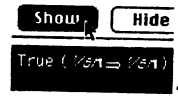
Dans le calcul de Hilbert adopté, décrivons la construction d'une preuve de $A \Rightarrow A$ (I=SKK). Voici l'écran, juste avant la construction de la preuve, la définition et la présentation de la logique ont déjà été analysées.



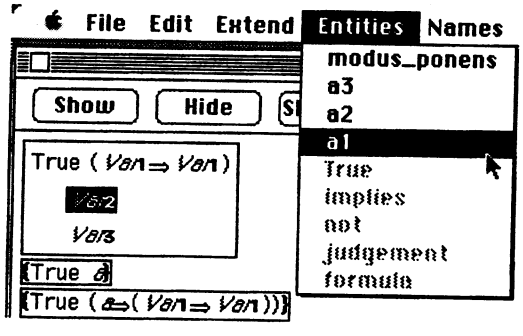
On remarque que les entités définies n'apparaissent pas toutes dans le menu **Entités**. En effet, nous avons développé cet exemple avant de factoriser le langage propositionnel. En fait, nous envisageons à très court terme de permettre le **masquage** de certaines entités du contexte de définition, notamment avec un langage de structuration de théories pour les langages de définition et de présentation.

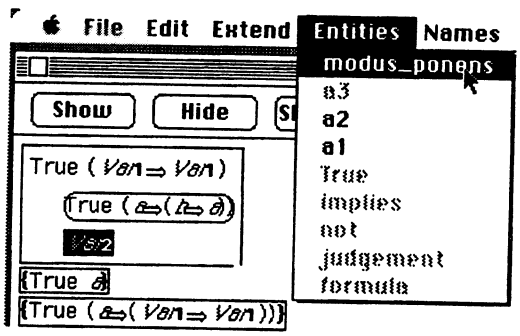
Pour construire la preuve de $A \Rightarrow A$, on peut commencer par appeler le connectif logique $\Rightarrow$ en le choisissant dans le menu **Entités** (les connectifs logiques, les fonctions et les règles d'inférences sont traitées de manière homogène). Il apparaît dans la fenêtre d'édition, ses

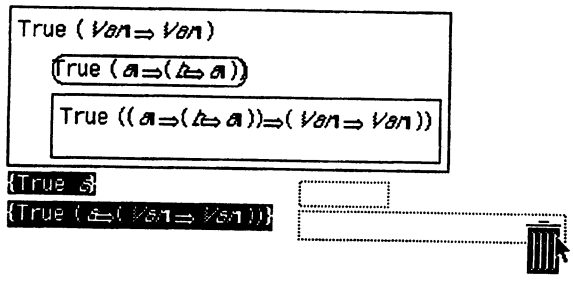
arguments sont matérialisés par deux variables différentes. L'une de ces deux variables peut être sélectionnée  et déplacée vers l'autre  . Pendant un tel déplacement, les destinations possibles sont déterminées par unification et mises en évidence au passage de la flèche. Si on relâche le bouton sur une boîte mise en évidence, la substitution obtenue est "appliquée" au terme de la boîte principale qui la contient  . Rappelons que le comportement de l'unification diffère selon si la sélection et la destination font partie ou non de la même boîte principale (voir chapitre Réalisation section B.8.b).

Pour construire le premier pas d'inférence, on peut appeler la règle `modus_ponens` et déplacer le terme déjà construit vers sa conclusion  . Le bouton **Montrer** permet de montrer la partie cachée de la règle d'inférence, c'est à dire les places de ses prémisses  . Ces places sont matérialisées par des variables libres dont les types représentent les prémisses. Bien que la présentation choisie ne montre pas ces types, le choix **Obtenir le Type** permet de les visualiser.

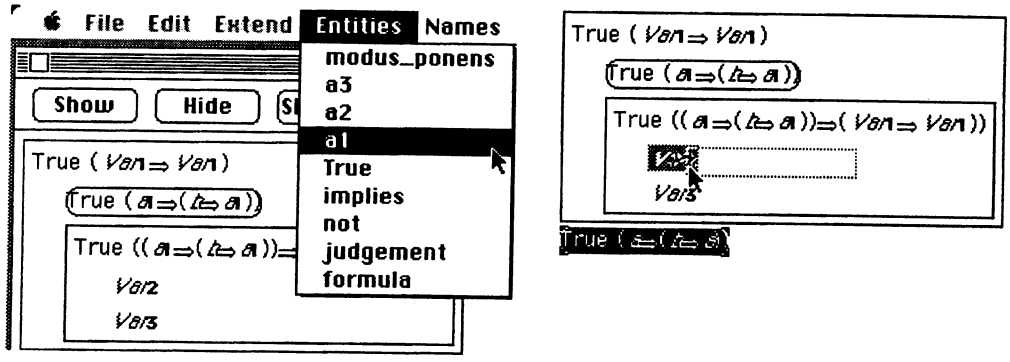
Après avoir examiné les prémisses, on peut appliquer l'axiome `a1` à la première. Pour cela, on pourrait appeler l'axiome et le déplacer vers la place correspondante. Une méthode plus rapide consiste à sélectionner cette place avant d'utiliser le menu **Entités**. Tant que le menu est déroulé, le terme associé à chaque choix est unifié avec le terme sélectionné. Les choix dont les termes ne sont pas unifiables sont estompés. Finalement, la substitution qui correspond au choix de l'utilisateur est appliquée au terme de la boîte principale qui contient la boîte sélectionnée..

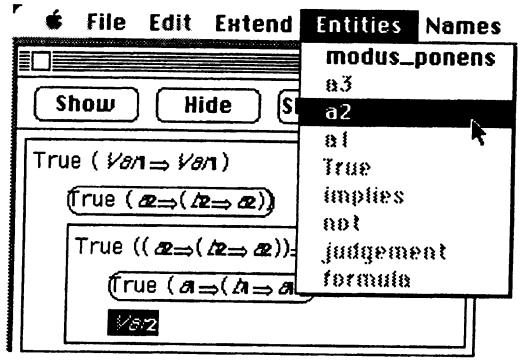




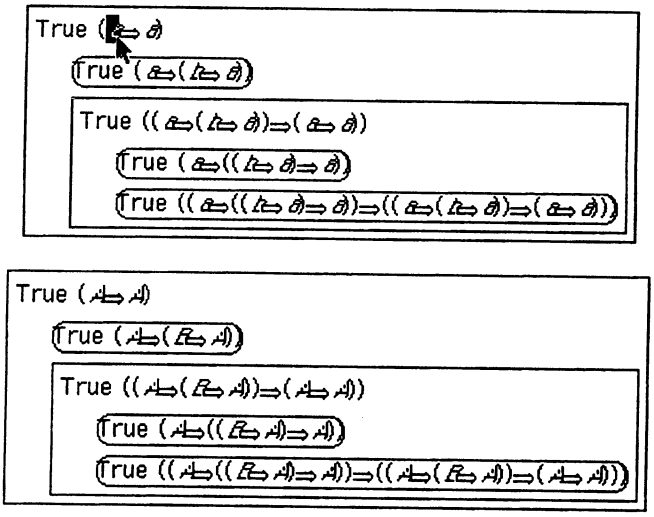


Après avoir jeté les prémisses, devenues inutiles, à la poubelle, on peut continuer la construction de la preuve avec les deux méthodes.

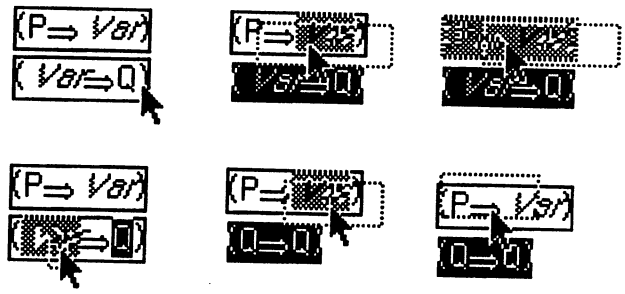




A tout moment, sélectionner une variable libre et éditer son nom au clavier permet de la renommer.



La figure suivante montre des exemples de déplacements en langage propositionnel. Notons la mise en évidence de la source (en noir) et des destinations possibles (en gris).



Pour construire une preuve, on peut commencer par appliquer toutes les règles d'inférence. L'instanciation de leurs hypothèses et de leurs conclusions peut intervenir en dernier, dans la règle dérivée obtenue. Cette méthode permet d'éviter certaines liaisons inutiles de variables. Par exemple, avant d'utiliser GLEF, nous connaissions une preuve de $A \Rightarrow A$ avec les mêmes pas d'inférence, mais où A remplaçait B. Cette méthode a mis en **évidence** la possibilité d'avoir la **variable libre** B indépendante de A.

GLEF a donc permis de trouver une preuve plus générale. Notons que cette preuve est aussi plus facile à lire, car des variables libres différentes aident à reconnaître les égalités entre termes, importantes pour sa compréhension.

Réutilisation de la preuve

La construction de preuves serait limitée si les théorèmes ou les règles dérivées démontrés ne pouvaient pas être réutilisés. Ainsi, GLEF permet de sauvegarder les preuves construites. Par exemple, la sauvegarde de la preuve précédente produit le fichier de définition suivant:

```
[p = <A:formula>(modus_ponens (implies A (implies A A))
    (implies A A)
    (a1 A A)
    (modus_ponens (implies A (implies (implies A A) A))
          (implies (implies A (implies A A)) (implies A A))
          (a1 A (implies A A))
          (a2 A (implies A A) A)
    )
)]
```

Inclure ce fichier dans un autre fichier de définition permet d'utiliser le théorème $A \Rightarrow A$. **L'ouvrir** avec GLEF permet de visualiser ou de modifier sa preuve.

c) Un système de Gentzen (GS)

Comme pour le système de Hilbert, la définition de ce calcul est fondée sur celle du langage propositionnel.

La notion de multi-ensemble du formalisme de définition sert à représenter les contextes qui forment les parties gauches des séquents.

GLEF permet de construire les preuves de ce système formel, en utilisant une seule manipulation **non naturelle**: la création d'un multi-ensemble vide.

Par exemple, on a construit la preuve suivante avec GLEF:

$$\dfrac{\dfrac{\{A,(A\Rightarrow B)\} \vdash A \quad \dfrac{\dfrac{\{A\} \vdash A \quad \{B,A\} \vdash B}{\{(A\Rightarrow B),A\} \vdash B} \quad \{C,A,(A\Rightarrow B)\} \vdash C}{\{(B\Rightarrow C),(A\Rightarrow B),A\} \vdash C}}{\{(A\Rightarrow(B\Rightarrow C)),A,(A\Rightarrow B)\} \vdash C}}{\{(A\Rightarrow(B\Rightarrow C)),(A\Rightarrow B)\} \vdash (A\Rightarrow C)}}{\{(A\Rightarrow(B\Rightarrow C))\} \vdash ((A\Rightarrow B)\Rightarrow(A\Rightarrow C))}}{\{\} \vdash ((A\Rightarrow(B\Rightarrow C))\Rightarrow((A\Rightarrow B)\Rightarrow(A\Rightarrow C)))}$$

2. Logique du premier ordre

a) Langage du premier ordre (FOL)

Le langage propositionnel peut servir à définir le langage du premier ordre. Il suffit de lui ajouter une catégorie syntaxique pour les termes:

```
[term:                  *]
```

Et de définir les connectifs logiques de quantification:

```
[exists,forall:         [[term]formula]formula]
```

Toutefois, comme nous avons choisi d'employer des connectifs logiques de conjonction et de disjonction d'arité variable, la factorisation n'est pas intéressante. Nous avons donc spécifié entièrement le langage du premier ordre, de la même manière que le langage propositionnel. En particulier, la définition des connectifs logiques de conjonction et de disjonction est:

```
[and,or:                [formula+]formula]
```

b) Langage du premier ordre avec sortes ordonnées (FOL*)

La définition du langage du premier ordre avec sortes est fondée sur celle du langage du premier ordre. Elle introduit une catégorie syntaxique pour les sortes et redéfinit les quantificateurs:

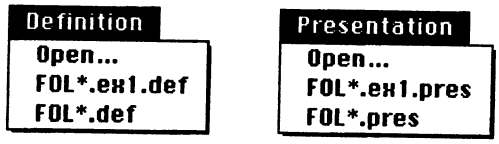
```
[sort:                  *]
[exists,forall:         [s:sort][[s]formula]formula]
```

N'ayant pas défini de calcul, nous n'avons pas construit de preuve sur ce langage. Cependant, nous l'avons employé pour développer deux exemples. Pour chaque exemple, une définition introduit une structure de sortes, des prédicats et une théorie, c'est à dire un ensemble de

formules. Rappelons que GLEF ne distingue pas les notions de preuves et de formules. Et une présentation associe une parure aux prédicats introduits.

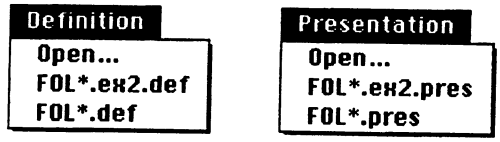
Le premier exemple met en évidence la possibilité de surcharger la présentation. Cette surcharge sert à présenter les formules de la logique dans un langage naturel (pseudo langue naturelle), au lieu d'employer les symboles usuels.

L'utilité de cette possibilité est évidente: il suffit de penser à la présentation de preuves pour un mathématicien (ni logicien, ni informaticien) ou pour enseigner la logique.




```
michel est le pere de jacques
jacques est le pere de robert
soit x un(e) pers,
   soit y un(e) pers,
      soit z un(e) pers,
         si x est le pere de y et y est le pere de z alors x est le grand-pere de z
```

Le deuxième exemple définit une théorie des ensembles, présentée dans une syntaxe et avec des symboles usuels en mathématiques. Les formules définies et présentées sont les axiomes de cette théorie.




```
(∀x:top ¬x∈∅)
(∀x1:set (∀x2:set ((∀u:top (u∈x1⇔u∈x2))⇒x1=x2)))
(∀x1:set (∀x2:set (x1=x2⇔x1⊂x2∧x2⊂x1)))
(∀x1:set (∀x2:set (x1⊂x2⇔(∀u:top (u∈x1⇒u∈x2)))))
(∀x1:set (∀x2:set (x1∈P(x2)⇔x1⊂x2)))
(∀x1:set (∀x2:set (∀u:top (u∈x1∩x2⇔u∈x1∧u∈x2))))
(∀x:set (∀i:top (i∈collection_union(x)⇔(∃u:set i∈u∧u∈x))))
```

c) Calcul de résolution (RES), (OTTER)

Le calcul de *résolution* a été défini pour présenter des preuves issues de différents démonstrateurs par résolution. Le langage employé est celui des clauses. Rappelons qu'une *clause* est une disjonction de

littéraux. C'est donc un sous-langage du langage du premier ordre. Un langage de structuration de théories permettrait de factoriser les définitions de ces langages.

Principe

On considère des hypothèses et une formule de la logique du premier ordre à démontrer. On met la conjonction et la négation de cette formule et des hypothèses sous forme clausale, c'est à dire, sous forme d'un ensemble de clauses.

Cet ensemble est insatisfaisable, et la formule à démontrer est vraie, si et seulement si la clause vide appartient à sa fermeture par deux opérations sur les clauses, la factorisation et la résolution. Ainsi, chaque opération définit une règle d'inférence qui consiste à étendre l'ensemble de clauses par son application.

La factorisation et la résolution emploient respectivement le filtrage et l'unification du premier ordre. La *factorisation* produit une clause formée de littéraux de la clause qui consitue son argument, telle que tout littéral de celle-ci soit instance d'un littéral de la clause produite et aucun littéral de la clause produite ne soit instance d'un autre. La *résolution* considère deux clauses dont deux instances contiennent deux littéraux complémentaires, de positions spécifiées, elle produit la clause formée de la réunion de ces instances, privée des deux littéraux.

Notons que des règles de mise sous forme clausale permettraient d'appliquer directement le calcul de résolution à la logique du premier ordre. Un transformateur de formules suivi d'un démonstrateur automatique pourrait produire des preuves automatiquement

Définition

Les définitions des opérations de factorisation et de résolution emploient le filtrage et l'unification du premier ordre. Le formalisme de définition ne propose pas ces algorithmes, on doit donc les programmer. Pour programmer, le formalisme de définition n'est ni aussi simple, ni aussi puissant qu'un langage de programmation. Ainsi, l'employer pour programmer ces algorithmes semble difficile, voire impossible.

Des fonctions externes, associées à des entités évaluables, implémentent donc les opérations de factorisation et de résolution. Ces fonctions externes exploitent les algorithmes de filtrage et d'unification du premier ordre déjà intégrés à GLEF.

```
[$FACTOR:                    [c1:clause]clause]
[$RESOLVE:                   [c1,c2:clause]clause]
```

Comme dans la plupart des systèmes formels, les preuves par résolution peuvent être représentées de plusieurs manières.

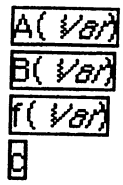
- Par exemple, on peut les représenter par une liste d'ensembles de clauses, correspondant aux états successifs d'un démonstrateur par résolution. Nous avons choisi cette représentation pour présenter les preuves obtenues par **le démonstrateur par résolution d'ATINF** [CHZ91], et pour montrer un exemple de construction de preuve par résolution à l'aide de GLEF (RES).
- Plus naturellement, on peut les représenter par l'arbre des applications des règles de résolution et de factorisation, correspondant à la structure de la preuve. Nous avons choisi cette représentation pour les preuves produites par **un démonstrateur extérieur à ATINF, OTTER** [McC90], qui est certainement devenu l'un des plus populaires parmi les démonstrateurs par résolution (OTTER).

Ces deux exemples sont présentés dans les deux sections suivantes.

Construction d'une preuve (RES)

Dans cet exemple nous avons choisi de représenter la **liste des états d'un démonstrateur automatique fictif**. Chaque état est représenté par un ensemble de clauses. Bien entendu, la forme des preuves n'est pas naturelle, car elle est linéaire, alors que les applications de règles d'inférence forment un arbre. Toutefois, nous avons construit une preuve complète dans cette représentation, afin d'illustrer les capacités de GLEF.

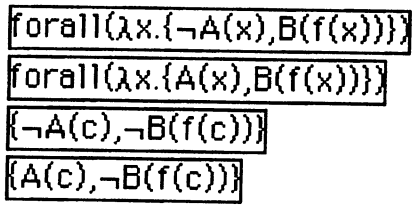
Pour construire la preuve, on commence par construire sa signature avec le menu **Étendre**. Par exemple, on crée deux prédicats, une fonction et une constante.



Ensuite, on construit l'ensemble des clauses initiales. Les entités `plus` et `minus` servent à signer les littéraux. L'entité `forall` permet d'introduire les variables clausales, dissimulées dans la présentation usuelle des clauses.

Moins de 20 manipulations suffisent à construire l'ensemble de clauses suivant, si on évite de construire plusieurs fois des termes identiques. Ces clauses ne sont pas présentées de manière usuelle, car

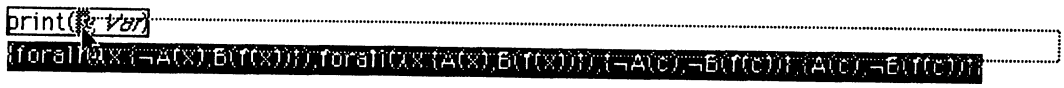
GLEF ne les reconnaît pas encore comme telles. Notons qu'ajouter le filtrage au langage de présentation permettrait à GLEF de présenter ces clauses correctement, sans informations supplémentaires (voir chapitre Travail futur et Conclusion section A.5).

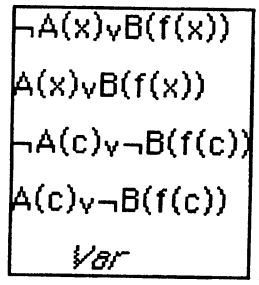


Les règles de factorisation et de résolution utilisées dans cet exemple admettent respectivement une et deux clauses, et l'ensemble des clauses courant comme arguments. Elles produisent un nouvel ensemble de clauses.

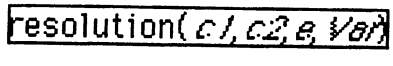
Dans leur présentation, nous avons choisi de masquer l'ensemble de clauses courant. La parure d'une règle d'inférence `print`, qui n'a aucun effet, sert à afficher cet ensemble. L'utilisation de cette règle introduit une dépendance supplémentaire entre la définition et la présentation. Bien que pratique, cette dépendance n'est pas souhaitable. Espérons que l'utilisation de filtres dans le langage de présentation permette d'éviter cette règle.

On appelle donc la règle `print`, pour commencer la preuve. Dans cette règle et dans le reste de la preuve, les clauses sont présentées de manière usuelle.

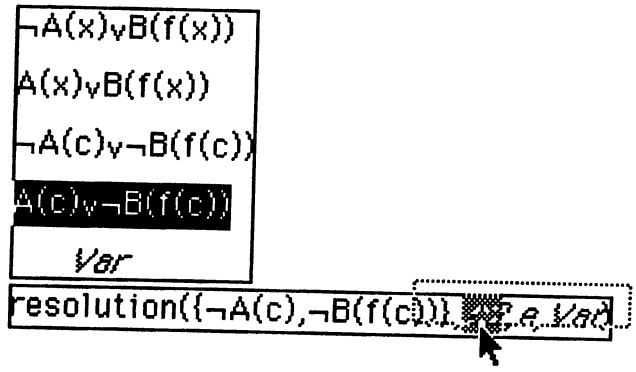




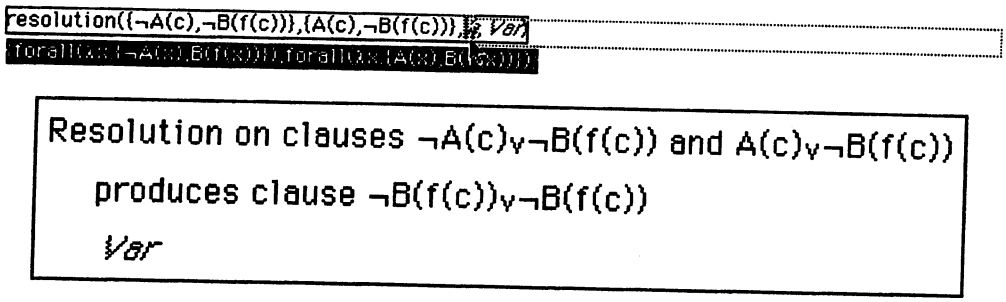
Pour construire le premier pas d'inférence, on appelle la règle de résolution. Comme ses arguments ne sont pas encore déterminés, sa parure échoue et la présentation par défaut est employée.



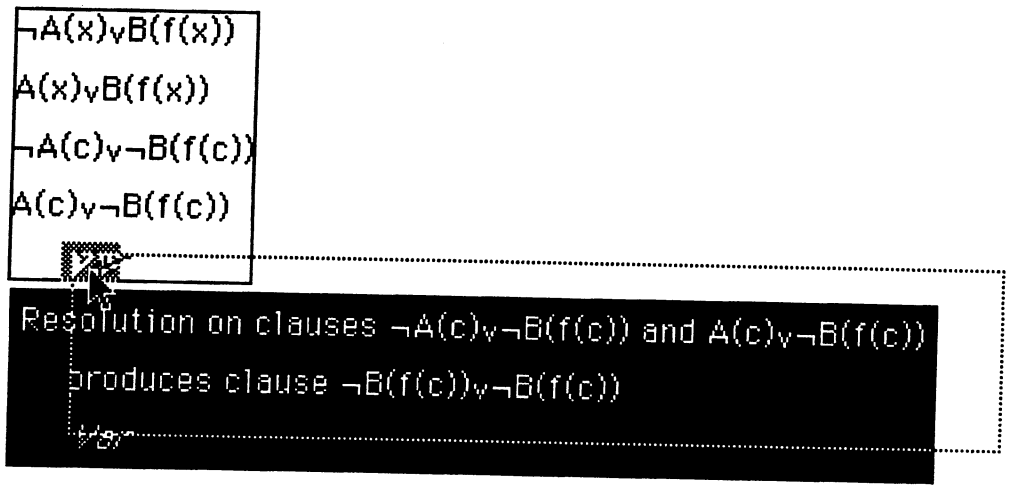
Les deux dernières clauses sont déplacées vers les deux premières places, c_1 et c_2:

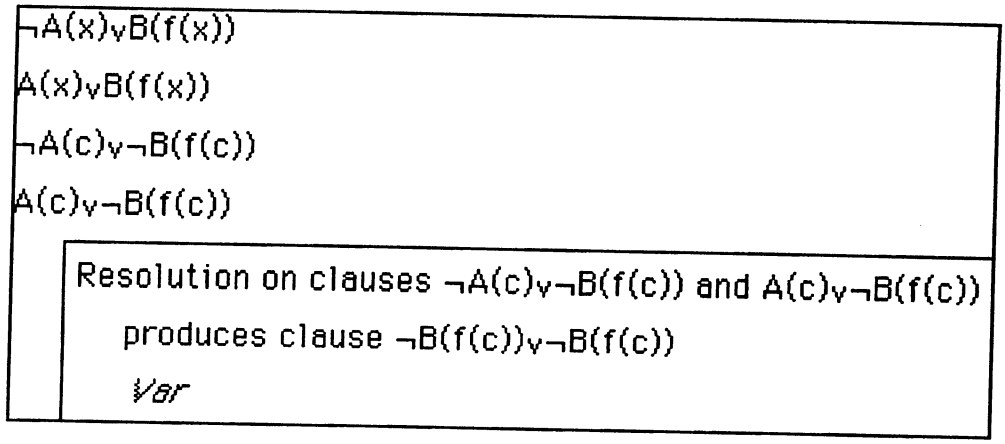


L'ensemble des clauses restantes est déplacé vers la place e pour préciser l'état du démonstrateur avant l'appel de la règle de résolution:



Le pas d'inférence peut être déplacé vers une place libre de la preuve:

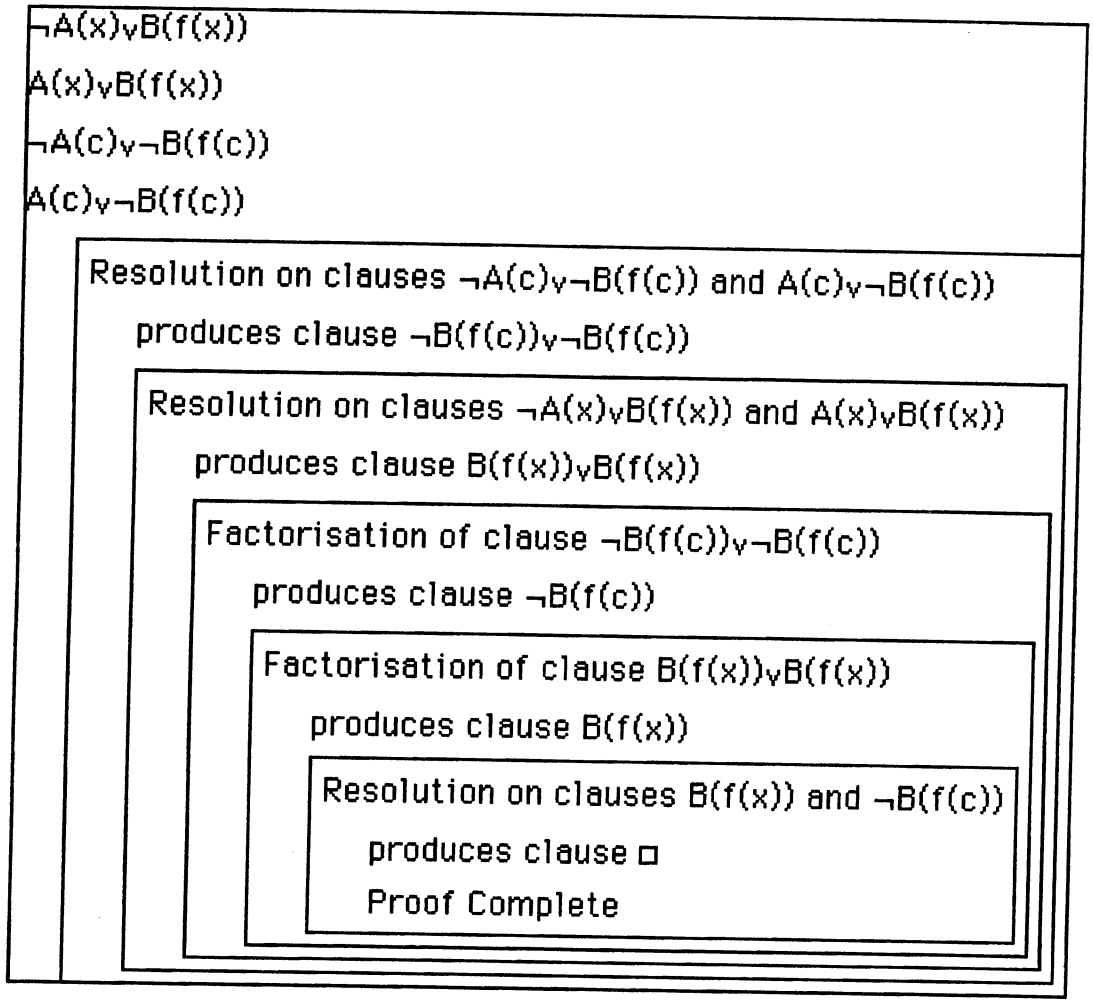




Ensuite, on construit et on insère les autres pas d'inférence (un pas de factorisation, deux pas de résolution et un pas de reconnaissance de la clause vide) de la même façon, c'est à dire:

- Appel de la règle.
- Déplacement des clauses concernées (0, 1 ou 2 clauses).
- Déplacement de l'ensemble des clauses restantes.
- Déplacement du pas d'inférence vers une place libre de la preuve.

La présentation de la preuve finale est:



Avec une structure de pas d'inférence plus naturelle, la construction de preuves serait plus naturelle, voire plus rapide. Notons que la **présentation** d'une preuve ne **respecte pas** forcément sa **structure**. Ainsi, on pourrait tenter de présenter une preuve par résolution de structure arborescente sous cette forme linéaire.

Présentation d'une preuve obtenue en dehors d'ATINF (OTTER) [McC90].

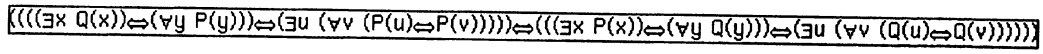
Pour illustrer le fait que GLEF puisse être connecté à différents outils d'inférence existants, nous avons choisi de présenter les preuves produites par un démonstrateur automatique, en l'occurrence OTTER, dont nous ne sommes pas les auteurs.

La définition du système formel emploie celle de la logique du premier ordre et les règles de factorisation et de résolution. Elle représente les preuves par **l'arbre** d'application de ces règles. Contrairement à l'exemple précédent, ces règles ne constituent pas un calcul de preuves, mais le formalisme de définition suffit à les définir. La présentation dispose ces arbres de manière **indentée**, elle montre **la**

clause vide en premier. Notons qu'elle aurait pu montrer les clauses initiales en premier.

Nous aurions pu considérer OTTER comme une **boîte noire** et écrire un programme d'analyse et de traduction des preuves produites. Mais comme son code source est public, ajouter un code générant un fichier de définition contenant ces preuves, au niveau de leur affichage, a été plus facile.

La preuve présentée est celle du fameux problème d'Andrews.



La preuve produite par OTTER contient 100 clauses, parmi plus de 500 clauses générées. Écrite par OTTER ou par la plupart des démonstrateurs automatiques, une preuve par résolution aussi longue est **très difficile à lire**. Dans cette écriture, chaque clause est représentée par une ligne numérotée, qui peut référencer 0, 1 ou 2 clauses précédentes, via les numéros de lignes.

La représentation d'une preuve dans le formalisme de définition reflète cette écriture. Les numéros de ligne sont simplement matérialisés par des **nommages**. Le langage de présentation de GLEF permet d'éviter ces références et de **présenter** directement **l'arbre** de résolution, de manière **hiérarchique**. Rappelons aussi qu'il présente les formules avec les symboles et la syntaxe **usuels**.

Bien entendu, quand une clause est utilisée plusieurs fois, son nom peut servir à factoriser la présentation de la preuve. Le code chargé d'écrire le fichier de définition des preuves produites repère automatiquement de telles clauses et génère un fichier de présentation adéquat.

D'autre part, on peut remarquer un certain nombre de **symétries** dans l'énoncé du théorème initial. Cette observation nous a conduit à rechercher des symétries dans la preuve elle même. Pour repérer les parties de preuves symétriques, nous n'avons considéré que leurs **conclusions** et leurs **hypothèses**, pas la structure de leurs pas d'inférence. La factorisation de parties de preuves symétriques dont les structures de pas d'inférence sont différentes ne conserve que la structure de l'une des parties. Elle modifie donc la structure des pas d'inférence de la preuve. Conserver la structure la plus "simple" assure que cette modification soit toujours une **simplification** de la preuve, même quand on développe les factorisations.

Ensuite le nommage a permis de factoriser les symétries repérées (voir chapitre Réalisation section A.1.g et chapitre Travail futur et Conclusion section A.13). La présentation permet de visualiser les symétries factorisées **indépendamment** du reste de la preuve.

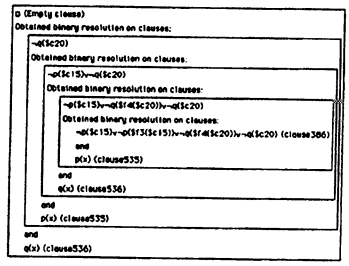
Ainsi, la preuve produite par OTTER contient quatre *symétries intéressantes* imbriquées. Comme ces symétries recouvrent presque toute la preuve, visualiser un "petit" sous-ensemble des pas d'inférence suffit pour **l'appréhender** entièrement.

On voit donc l'intérêt d'un outil général de recherche automatique et systématique des symétries intéressantes, dans des preuves déjà obtenues et décrites dans le formalisme de définition (voir chapitre Travail futur et Conclusion section A.13). Actuellement, un travail sur la recherche automatique de régularités dans preuves déjà obtenues est mené au sein d'ATINF.

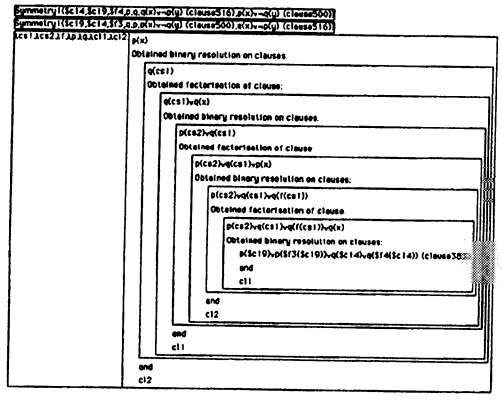
Notons que factoriser la présentation d'une clause utilisée plusieurs fois est un cas particulier de la factorisation de symétrie. La symétrie considérée est une égalité, elle n'a donc pas d'arguments.

Voici la présentation de la preuve, elle factorise les symétries trouvées manuellement:

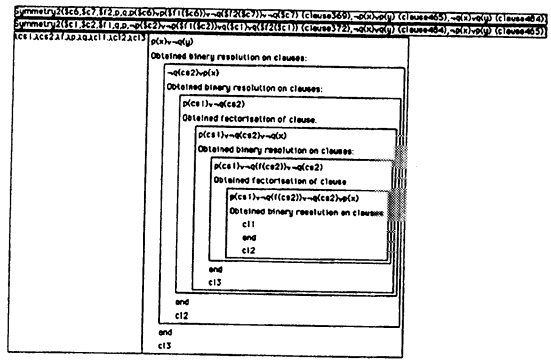
Racine:



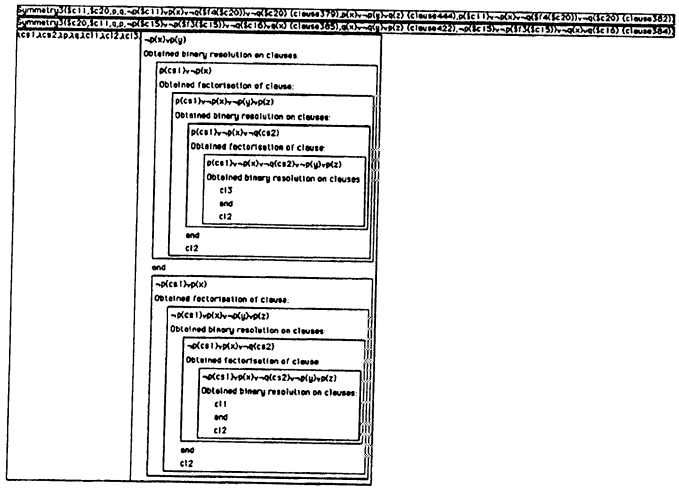
Clauses 535, 536 et première symétrie:



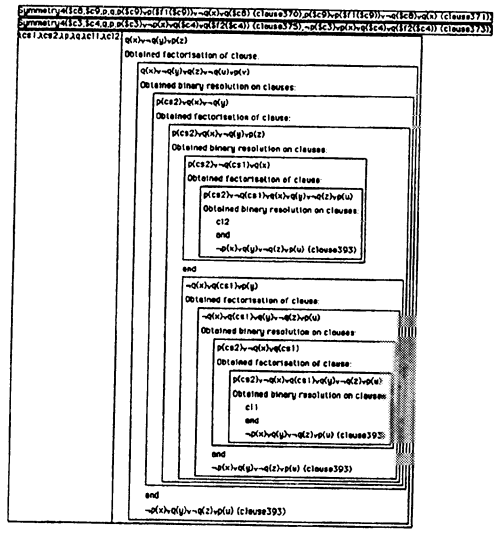
Clauses 500, 516 et deuxième symétrie:



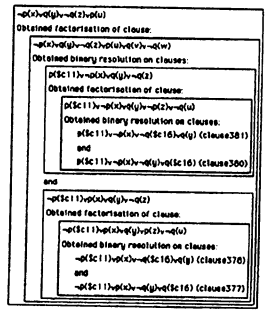
Clauses 465, 484 et troisième symétrie:



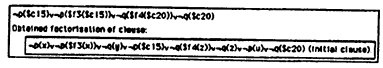
Clauses 422, 444 et quatrième symétrie:



Fin de la preuve, clause 393:



Les clauses restant à développer sont les factorisations des clauses initiales. Par exemple, voici la présentation de la clause 386:



d) Tableaux sémantiques (TS)

Ce calcul a été implémenté pour présenter les preuves produites par le démonstrateur par tableau sémantiques en logique du premier ordre de ATINF [CHZ91]. Bien entendu, son implémentation pourrait servir à construire des preuves manuellement ou à présenter des preuves produites par d'autres démonstrateurs.

Sa spécification est fondée sur celle du langage du premier ordre. En particulier, les connectifs logiques de conjonction et de disjonction sont d'arité variable.

GLEF a permis de présenter de nombreuses preuves produites automatiquement. Toutefois, ce document ne donne aucun exemple, car le calcul considéré est une restriction des tableaux sémantiques en logique modale. L'exemple donné pour cet autre calcul peut donc servir de référence.

Principe

La méthode des *tableaux sémantiques* construit un arbre de manière "top-down". Un ensemble de formules étiquette chacun de ses nœuds. Initialement, les hypothèses et la négation de la formule à démontrer étiquettent la racine. Ensuite, les règles servent à ajouter des nœuds à l'arbre courant.

Dans l'arbre courant, on choisi une formule qui n'a pas encore été décomposée. On peut appliquer une règle et une seule à cette formule. Cette règle indique comment décomposer la formule en plusieurs ensembles de formules. A chaque feuille de l'arbre courant, située "sous" la formule choisie, on ajoute des fils étiquetés par chacun de ces ensembles.

Une branche est *fermée* ssi elle contient un littéral et son complémentaire. Un arbre est *fermé* ssi toutes ses branches le sont.

Quand l'arbre de preuve est fermé, l'ensemble des formules qui étiquette la racine est insatisfaisable.

Notons que le calcul des tableaux sémantiques tel quel n'est pas un calcul de preuves. En effet, dans notre définition de preuve (voir chapitre Définition section A) l'application d'une règle d'inférence ne peut dépendre que de séquents formels, pas de l'existence d'une formule dans une branche de l'arbre de preuve. Toutefois, la description de sa définition montre comment le représenter par un tel calcul (voir la section simulation des branches qui suit).

On peut regrouper les règles selon leurs formes, en deux classes principales: Les règles de type α et les règles de type β.

Règle de type α

$$\frac{\alpha}{\begin{array}{l}\alpha_1\\ \vdots\\ \alpha_n\end{array}}$$

Règle de type β

$$\frac{\beta}{\beta_1 \quad \cdots \quad | \beta_n}$$

Définition

La définition de ce calcul, pose trois problèmes majeurs: la simulation des branches, le regroupement des règles selon leur classe et les règles d'inférence à nombre variable de prémisses.

Décrivons ces problème et leurs solutions, de manière plus précise.

Simulation des branches

Du point de vue de l'utilisateur, la représentation graphique des tableaux sémantiques est très pratique. Les formules obtenues par décomposition sont écrites au niveau des feuilles de l'arbre courant et ne sont jamais modifiées ni dupliquées.

Affirmer qu'une branche soit fermée fait intervenir l'ensemble de ses nœuds, de la racine jusqu'à sa feuille. En général, les sous-arbres d'un arbre fermé ne sont pas fermés.

Pour rendre la propriété de fermeture locale, on remplace l'étiquette de chaque nœud par la réunion des ensembles qui étiquettent le nœud et tous ses ancêtres. Un nœud est dit *fermé* ssi le nouvel ensemble qui l'étiquette contient un littéral et son complémentaire ou tous ses fils sont fermés. Notons qu'un **arbre est fermé** (avec les anciennes étiquettes) si et seulement si sa **racine est fermée** (avec les nouvelles étiquettes).

Pour chaque règle, on considère la règle d'inférence qui combine les nœuds à l'envers. On considère l'axiome caractérisé par l'ensemble des nœuds dont l'étiquette contient un littéral et son complémentaire. Ces

règles d'inférence et cet axiome génèrent l'ensemble des nœuds fermés. Ils forment donc un **calcul de preuves**. On étend la notion de classe de règles aux règles d'inférence qui leur correspondent.

Nous avons implémenté ce calcul de preuves, qui simule le fonctionnement des branches des tableaux sémantiques, dans le formalisme de définition.

Regroupement des règles d'inférence selon leur classe

Le formalisme de définition permet de définir des entités de profils multiples. Cette possibilité peut servir à définir directement les classes des règles d'inférence, au lieu de définir séparément chaque règle d'inférence. Par exemple, voici la définition de la classe des règles d'inférence de type α:

```
[alpha:  {[f:formula+][b:formula+]
              [(close ($union formula f b))]
                   (close ($union formula {(and f)} b)),
              [f:formula+][b:formula+]
              [(close ($union formula
                   ($map formula formula not f) b))]
                        (close ($union formula {(not (or
f))} b)),
         [f,g:formula][b:formula+]
              [(close ($union formula {f,(not g)} b))]
                   (close ($union formula
                        {(not (implies f g))} b))}]
```

Les trois profils de la classe correspondent à ses trois règles d'inférence, qui servent respectivement à construire les conjonctions, les négations de disjonctions et les négations d'implications.

Décrivons le premier profil, qui sert à construire les conjonctions:

- Dans `[f:formula+],f` représente les formules de la conjonction.
- Dans `[b:formula+],b` représente les formules non concernées par la règle d'inférence. Ces formules sont propagées des prémisses vers la conclusion.
- `[(close ...)]` est la prémisse de la règle, elle s'applique à certains nœuds. Bien entendu, une règle d'inférence peut avoir plusieurs prémisses.
- `(close ...)` est la conclusion de la règle, elle construit un nœud.

On peut interpréter ce premier profil comme la règle d'inférence informelle suivante:

Si le nœud contenant les formules:

$$\left.\begin{matrix} f_1 \\ \vdots \\ f_n \end{matrix}\right\} f$$
$$\left.\begin{matrix} g_1 \\ \vdots \\ g_p \end{matrix}\right\} b$$

est fermé alors le nœud contenant les formules:

$$f_1 \ldots \wedge f_n \} f$$
$$\left.\begin{matrix} g_1 \\ \vdots \\ g_p \end{matrix}\right\} b$$

l'est aussi.

Règles d'inférence à nombre variable de prémisses

Dans une application de la règle d'inférence de type β qui correspond à la disjonction, le nombre de prémisses de la règle est égal au nombre de formules de la disjonction concernée. Comme la disjonction est **d'arité variable**, le nombre de prémisses de cette règle n'est pas connu d'avance.

Une entité évaluable `$make-pi-list` permet de définir une règle d'inférence dont le nombre de prémisses dépend de ses premiers arguments.

Pour définir des règles d'inférence dont le nombre de prémisses dépend de ses premiers arguments, on peut utiliser une entité évaluable, ici `$make-pi-list`. Lorsque la règle d'inférence est appliquée à ses premiers arguments, l'application de `$make-pi-list` est évaluée et **produit** ainsi dynamiquement **une règle d'inférence spécifique** à appliquer aux autres arguments.

```
[beta: {[f:formula+][b:formula+]
   ($make-pi-list judgement
         ($map formula judgement
               <g:formula>(close ($union formula {g} b)) f)
                     (close ($union formula {(or f)} b))),
[f:formula+][b:formula+]
   ($make-pi-list judgement
         ($map formula judgement
               <g:formula>(close ($union formula {(not g)} b)) f)
                     (close ($union formula {(not (and f))} b))),
[f,g:formula][b:formula+]
   [(close ($union formula {(not f)} b))]
   [(close ($union formula {g} b))]
```

```
                (close ($union formula {(implies f g)} b)),
[f,g:formula][b:formula+]
  [(close ($union formula {f,g} b))]
  [(close ($union formula {(not f),(not g)} b))]
        (close ($union formula {(equiv f g)} b)),
[f,g:formula][b:formula+]
  [(close ($union formula {f,(not g)} b))]
  [(close ($union formula {(not f),g} b))]
        (close ($union formula {(not (equiv f g))} b))}]
```

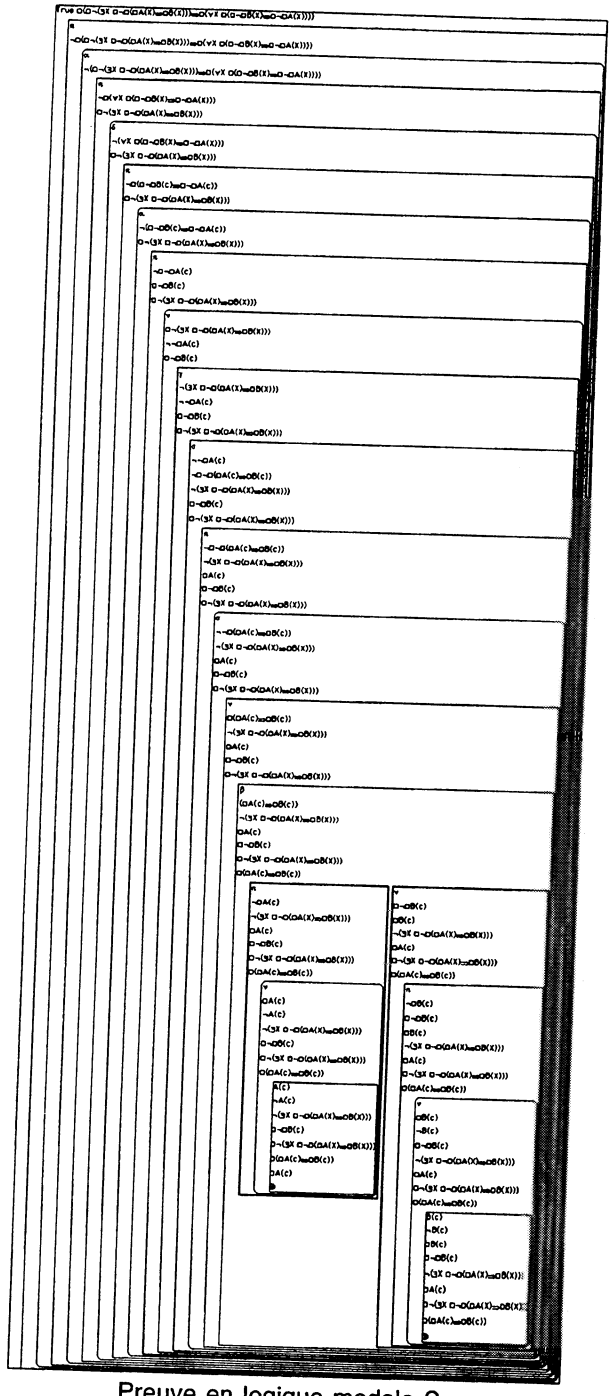
Si nous avions décidé d'employer une conjonction et une disjonction à deux arguments, les profils des règles d'inférence de type β correspondant à la disjonction et à la négation de conjonction auraient la forme des profils correspondant à l'implication, à l'équivalence et à la négation d'équivalence.

3. Tableaux sémantiques en logique modale (TSM)

Des calculs ont été implémentés dans les logiques modales T et S_4 pour présenter les preuves produites par le démonstrateur par tableaux sémantiques pour logiques modales de ATINF [CDH91], [CD92], [CDH93].

La spécification des tableaux sémantiques en logique modale est fondée sur celle des tableaux sémantiques en logique du premier ordre. Elle n'inclut pas directement de règle d'inférence de type π, liée au connectif logique de possibilité, et qui diffère dans les logiques modales T et S_4. Ainsi, les spécifications des tableaux sémantiques pour ces logiques modales sont fondées sur cette spécification. Elles définissent chacune une règle d'inférence de type π.

Voici un exemple de preuve produite par le démonstrateur par tableaux sémantiques pour logiques modales et présentée avec GLEF:



Preuve en logique modale S_4

Examinons les différentes couches constituant la spécification des tableaux sémantiques en logique modale S_4. **Elles reflètent les factorisations effectuées entre les définitions et les présentations de différents langages, calculs ou systèmes formels**:

- Langage du premier ordre
- Tableaux sémantiques en logique du premier ordre
- Tableaux sémantiques en logique modale

- Tableaux sémantiques en logique modale S_4

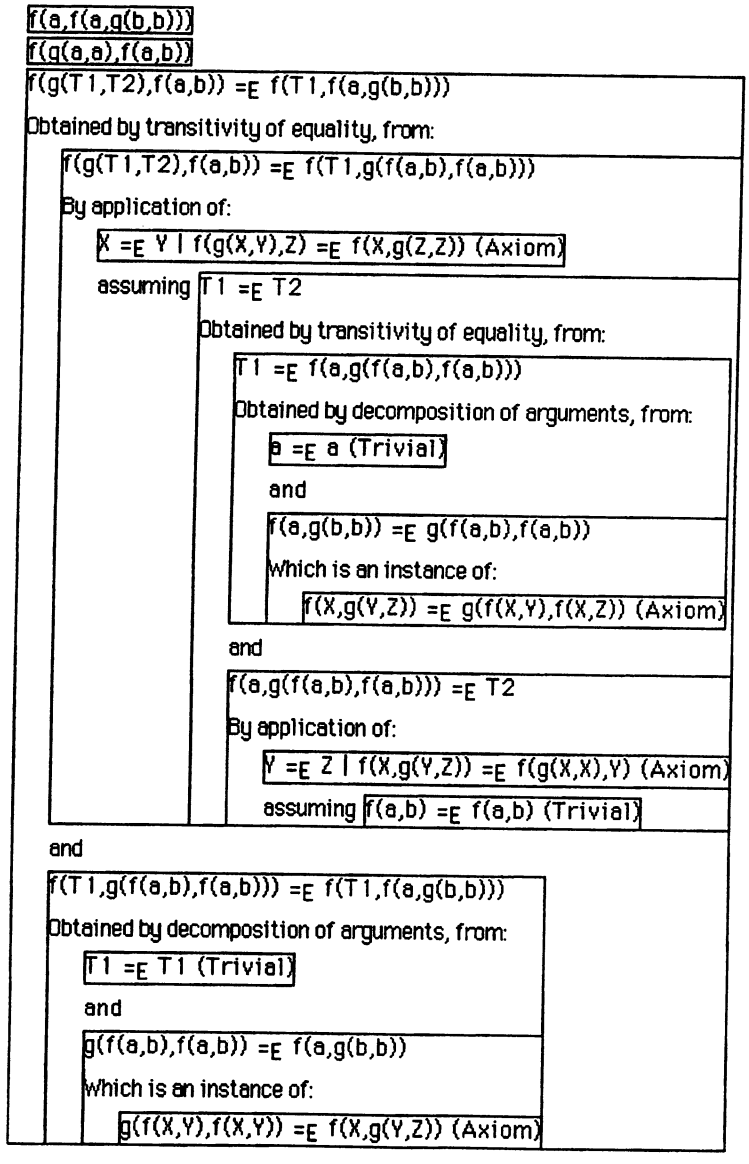
4. Preuves équationnelles (EQP)

Ce système formel a été implémenté pour présenter les preuves obtenues par le démonstrateur en logique équationnelle de ATINF [CHZ91], [CH93], [Del92].

Éditer les preuves équationnelles ne pose pas de problèmes particuliers. L'égalité est définie, présentée et manipulée comme un connectif logique classique.

```
[equals:          [term][term]equality]
```

Voici un exemple de preuve produite par le démonstrateur en logique équationnelle et présentée avec GLEF. Les deux premières boîtes présentent les termes `T1` et `T2`.



En améliorant le langage de présentation, une parure pourrait modifier la présentation de sous-termes de la partie de terme dont elle décrit la présentation (voir chapitre Travail futur et Conclusion section A.5). Par exemple, la parure associée à la règle de transitivité de l'égalité pourrait mettre le terme auxiliaire introduit en évidence.

5. Transformation de preuves: Traduction en pseudo-langue naturelle (NL)

GLEF permet d'employer des outils de transformation de preuves. Par exemple, nous considérons une preuve en langage naturel produite par un traducteur de preuves formelles en langue naturelle [Hua90]. Ce genre d'outils [Che76], [Lin88], peut servir présenter des preuves produites par différents moyens, de manière plus "lisible".

En effet, la capacité de GLEF de présenter des preuves de manière naturelle, par exemple en langue naturelle, est extrêmement importante.

- C'est l'habitude en mathématiques.
- Ce n'est pas celle des outils d'inférence.

Comme on ne sait pas entièrement formaliser la langue naturelle, on simule celle-ci par une grammaire formelle dont le langage est appelé *langage naturel*. Le formalisme de définition et le langage de présentation ont permis très facilement de spécifier entièrement le langage naturel employé et la syntaxe des formules utilisées dans la preuve [Hua90].

Voici une preuve en langage naturel, présentée avec GLEF:

Proof:

Because 1_u is the unit element of U, $1_u \in U$.

Thus $\exists x\ x \in U$.

Now Suppose u is an arbitrary element of U.

By the definition of unit, $u1_u = u$.

Because U is a subgroup of F, $U \subset F$.

Thus $u \in F$.

Similarly, $1_u \in F$.

In addition, F is a semigroup because it is a group.

Now We have proved that 1_u is a solution of $ux = u$ in F.

On the other hand, Because 1 is the unit element of F and $u \in F$, $u1 = u$ and $1 \in F$.

Thus 1 is also a solution of $ux = u$ in F.

By the uniqueness of solution in a group, $1 = 1_u$.

This conclusion is independent of the choice of the element u.

VIII. Travail futur et conclusion

A) Travail futur

Par nature, ATINF et GLEF sont des systèmes évolutifs. ATINF regroupe des outils d'inférence de plus en plus nombreux. GLEF fédère des outils d'inférence, internes ou externes à ATINF. Il tente de factoriser les similitudes suffisamment générales des outils d'inférence existants.

Cette partie donne un aperçu non exhaustif des extensions et des directions de développement possibles et intéressantes pour GLEF. Certains points sont des problèmes d'implémentation, fondés parfois sur des études théoriques existantes, d'autres sont des problèmes essentiellement théoriques. Nous les avons approximativement classés du court terme vers le long terme.

1. Vers un produit de qualité, à large distribution

Pour qu'un maximum d'utilisateurs soit tenté et satisfait par GLEF, nous cherchons encore à améliorer sa **maniabilité**. En particulier, nous envisageons d'optimiser l'utilisation de la mémoire et la vitesse, de simplifier la syntaxe des langages, d'élargir la classe des termes que l'on peut construire de manière naturelle, d'améliorer les manipulations naturelles, etc.. Cette partie présente d'ailleurs ces différentes idées.

Bien entendu, nous devrons rédiger le plus tôt possible un **manuel utilisateur** et considérer rapidement les **suggestions des différents utilisateurs**, dès le début de la distribution.

2. Interface

On envisage plusieurs optimisations de l'interface:

Les menus de termes pourraient être remplacés par des **palettes flottantes**, comme celles proposées par certains éditeurs d'équations comme EXPRESSIONIST. Une *palette flottante* est une fenêtre découpée en cases qui forment des boutons. L'accès à une case de palette flottante (un click) est donc beaucoup plus **rapide** que celui d'un choix de menu (un click et un déplacement).

Ces palettes flottantes pourraient être **organisées** comme des **menus détachables**. Un *menu détachable* est un menu que l'utilisateur peut transformer en palette flottante, en le déplaçant.

Pour un accès plus **intuitif**, chaque choix (ou case) associé à un terme pourrait contenir sa **présentation**, au lieu de son nom.

Après l'appel d'un constructeur, GLEF pourrait **sélectionner** la "première" variable libre de la preuve partielle, pour **accélérer** sa construction. L'utilisateur pourrait employer la touche de tabulation (standard Macintosh) pour changer la variable libre sélectionnée, etc..

Notons que GLEF ne permet pas de **saisir** des λ-termes au clavier, **directement** dans le langage de définition. En effet, une telle possibilité suppose que l'interface graphique puisse être moins pratique que l'interface textuelle. Quand c'est le cas, cela provient plutôt d'une étude **insuffisante** de l'interface graphique.

3. Expérimentations

L'un des buts de GLEF est de réaliser l'intégration d'ATINF. C'est à dire, de faire cohabiter et communiquer différents outils d'inférence sous une interface utilisateur homogène. Dans le chapitre Etat de l'Art section B, nous avons vu les nombreuses formes d'outils d'inférence existantes.

GLEF communique déjà avec cinq démonstrateurs automatiques (différents) et un traducteur de preuves formelles en langage naturel [Hua90]. Cette communication est superficielle, mais donne des résultats intéressants (voir chapitre Utilisation section C). Il intègre aussi des outils de manipulation de multi-ensembles, de filtrage et d'unification.

Bien entendu, nous intégrerons d'autres outils à GLEF. Il serait particulièrement intéressant **d'intégrer** des outils d'inférence de natures plus diverses (démonstrateurs interactifs, traitement de la langue naturelle, introduction de lemmes, transformation de formules ou de preuves, nommage, etc.) (voir chapitre Etat de l'Art section B), des outils non "logiques" (outils de calcul formel, de présentation de graphes, etc.), et surtout de les faire **communiquer** entre eux, notamment pour former des chaînes d'outils (voir chapitre Etat de l'Art section B.6). Notons que la réalisation d'un analyseur de preuves en langue naturelle indépendant de la logique utilisée (voir chapitre Etat de l'Art section B.5) correspond avec la capacité de GLEF à **traiter des preuves décrites à différents niveaux de formalisation, et de faire communiquer des outils travaillant à ces différents niveaux**.

4. Structure des termes du formalisme de définition

GLEF mémorise tous les termes rencontrés une fois et une seule. A priori, les aspects négatifs de ce principe sont la dépendance avec la présentation, la mémorisation de tous les termes et la lenteur de création des termes. Ces idées sont erronées, car la plupart des termes

créés existent déjà, la duplication de termes n'est plus récursive, etc.. Ce principe accélère au contraire de nombreux calculs (égalité, présentation, etc.) (voir chapitre Réalisation section B.3.d). En pratique, on constate une amélioration considérable de l'utilisation de la **mémoire** et de la **rapidité** de GLEF.

Pour simplifier et améliorer encore GLEF, nous envisageons de retirer les **noms des liaisons** pour les mémoriser dans des tables séparées, puis de calculer et répertorier les **formes normales**, comme les types et les formes des termes. De même, nous pourrions essayer de répertorier les **recalages** de termes.

Pour diminuer la taille des **répertoires**, nous pourrions aussi ne mémoriser que les types, les formes et les formes normales des termes minimalement recalés, dans les contextes diminués correspondants. Pour éviter les recalages, il faudrait associer à chaque terme qui n'est pas une variable, le terme minimalement recalé et l'indice de recalage qui lui correspondent, via un répertoire, une mémorisation directe dans le terme, etc..

Nous pourrions même envisager de ne considérer que **des termes minimalement recalés**, munis d'un indice de recalage. Cela diminuerait la taille des répertoires et rendrait le recalage immédiat. Néanmoins, la mémoire employée par chaque terme serait plus importante.

Mais le **gain** (ou la perte) de mémoire ou de temps apporté par ces idées n'est pas évident à **évaluer**. Des expérimentations ou une étude statistique seraient utiles.

5. Langage de présentation

Le langage de présentation est puissant, mais sa syntaxe est encore un peu lourde. Remplacer des mots clés comme `BEGIN` ou `END` par des mots clés plus courts, comme `{` ou `}`, ne simplifierait pas véritablement sa syntaxe (bien entendu, on pourrait autoriser les deux écritures, pour la compatibilité).

Par contre, nous pensons qu'employer les occurrences pour parcourir les termes a été un mauvais choix. Employer des filtres, comme PPML [BCDIKLP87], serait beaucoup plus naturel. Pour un langage donné, un *filtre* est une expression incomplète (avec des variables qui n'appartiennent pas au langage) qui détermine une classe d'expressions par filtrage. Heureusement, cette erreur est relativement facile à corriger, et le sera sous peu.

D'autre part, une parure décrit la présentation d'un terme, de sa racine, jusqu'aux places de certains sous-termes. D'autres parures décrivent ailleurs, les présentations de ces sous-termes. Pouvoir modifier à distance ces présentations serait très pratique. Par exemple,

dans la présentation de la preuve équationnelle du chapitre Utilisation (voir chapitre Utilisation section C.4), la parure associée à la règle de transitivité de l'égalité pourrait mettre le terme auxiliaire introduit en évidence.

Les sous-termes considérés pourraient être indiqués à l'aide d'occurrences généralisées ou de filtres. De plus, la portée des filtres pourrait être limitée à certaines profondeurs de sous-termes.

6. Correction de la structure de boîtes

Pendant la construction de la structure de boîte, l'algorithme de présentation **détecte** et **contourne** les erreurs de présentation en minimisant les déformations de l'image finale (voir chapitre Réalisation section B.6).

De même, l'algorithme de calcul des boîtes devrait non seulement détecter, mais aussi contourner les erreurs de placement, en minimisant ces déformations.

En particulier, des erreurs de placement se produisent quand le système d'équations de boîtes a plusieurs solutions ou n'en a pas. Une méthode de récupération pourrait consister à ajouter, retirer ou modifier automatiquement des équations du système. Pour minimiser les déformations de l'image finale, il faudrait étudier les dépendances entre les systèmes d'équations locaux, associés à chaque boîte.

7. Tactiques

GLEF permet la construction de preuves en partie manuelle et en partie automatique. Lors de la construction manuelle d'une preuve ou de l'appel de procédures automatiques, l'utilisateur peut être conduit à faire des opérations répétitives.

De telles opérations peuvent être considérablement simplifiées et accélérées à l'aide de tactiques et d'opérateurs de tactiques [GMW79].

Intégrer un mécanisme de tactiques à GLEF serait donc souhaitable. Pour cela, nous pourrions prendre les entités du contexte de définition comme tactiques de base, et proposer des opérateurs de tactiques à l'utilisateur. De plus, GLEF pourrait proposer un langage de programmation auxiliaire, par exemple ML. Ce langage servirait d'ailleurs à programmer les entités évaluables.

Grâce au traitement homogène du formalisme de définition, les tactiques pourraient non seulement servir à automatiser des recherches de preuves mais aussi toutes les opérations manuelles. Par exemple, elles pourraient servir à transformer des formules ou à traduire des preuves.

Dans une certaine mesure, l'intégration de tactiques transformerait GLEF en un **démonstrateur interactif** (graphique et générique).

8. Unification d'ordre supérieur

Avec les constructeurs du langage de définition, GLEF permet de construire tous les termes de manière bottom-up. Comme les menus de termes contiennent des entités appliquées à un nombre maximal de variables, l'ajout et une manipulation élémentaire de substitution suffiraient pour construire tous les termes formés d'applications d'entités et de variables, de manière top-down. Spécifier et réaliser une interface utilisateur naturelle pour la manipulation élémentaire de substitution n'est pas évident. Pour déterminer une substitution, l'utilisateur pourrait instancier successivement les variables libres du terme concerné.

La manipulation formelle d'unification **facilite** et **simplifie** ce type de construction, en déterminant automatiquement une substitution à appliquer à un terme destination, pour qu'il soit instance d'un terme source. Ainsi, GLEF propose deux manipulations élémentaires fondées sur l'unification, l'application et l'unification. Notons qu'il ne propose pas la substitution et l'instanciation de variables libres comme manipulations élémentaires car l'application peut les simuler.

De même, l'unification d'ordre supérieur n'augmenterait pas la classe des termes constructibles avec l'ajout et la substitution, mais elle **faciliterait** et **accélérerait** encore ce type de construction.

9. Homogénéité des interfaces utilisateur de réglage des paramètres

Souvent, un outil possède des paramètres réglables par l'utilisateur. Pour ces réglages, il peut proposer une interface utilisateur graphique ou textuelle, ou, comme OTTER [McC90], admettre des entrées qui les contiennent.

Nous pensons intégrer à GLEF un service facilitant la création de **dialogues utilisateur**. Il ne s'agit pas de créer une nouvelle IUGG du type X-WINDOW. Ce service serait seulement capable de créer et de gérer automatiquement un dialogue utilisateur, à partir, par exemple, d'une **spécification** de celui-ci dans **un langage très simple**.

Dans une expérience précédente, nous avions réalisé **un éditeur graphique et générique de données structurées**. L'utilisateur pouvait éditer les données structurées correspondant à une spécification qui décrivait leur structure, les contraintes sur les valeurs des constituants de cette structure, et les règles de placement relatif de ces constituants. Cet outil correspond à peu près au service désiré ici.

Il permettrait à l'utilisateur de régler les **paramètres** de chaque outil lié à GLEF, ces paramètres étant décrits par des données structurées. Ainsi, un outil sans interface utilisateur, ne proposant

qu'une interface utilisateur textuelle ou nouvellement développé **bénéficierait d'une IUG** pour régler ses paramètres, pour un faible coût de programmation. De plus, les interfaces utilisateur de réglage de paramètres de tous les outils liés à GLEF auraient un **aspect homogène**. Contrairement à l'utilisation d'une IUGG native, le langage de spécification serait **indépendant de la machine** utilisée.

10. Méta-programmation

Quand on utilise un langage de programmation produisant des applications indépendantes, comme C++, permettre à l'utilisateur d'ajouter dynamiquement de nouvelles fonctions au système est difficile et rarement portable.

GLEF pourrait donc proposer un interprète auxiliaire, pour l'utilisateur désirant programmer de manière interactive. Bien entendu, au lieu de développer un nouvel interprète, on emploierait un interprète existant, un interprète ML par exemple.

Ajouter ML à GLEF ne sous-entend pas que C++ ait été un mauvais choix pour sa programmation, et qu'il eusse été préférable d'employer ML. Rappelons que CENTAUR [BCDIKLP87] a été réalisé par assemblage de nombreux langages de programmation, le but de chacun étant bien déterminé.

11. Structuration de théories

Les langages de définition et de présentation permettent la spécification de systèmes formels par couches successives. Mais pour le moment, ils ne proposent que l'opération `#include` pour structurer les spécifications. Cette opération correspond à peu près à l'opération d'extension de S-CLEAR [LB92], elle est relativement limitée.

Employer un véritable langage de structuration de théories, comme S-CLEAR, serait préférable pour **mieux structurer** les spécifications de définitions et de présentations.

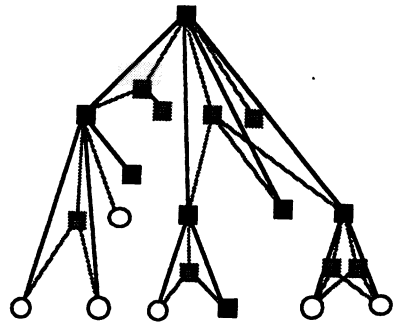
12. Supposition et planification

Pour construire une preuve avec GLEF, l'utilisateur peut **combiner** des termes avec les constructeurs du formalisme de définition, ou les **instancier** avec d'autres termes. Les termes initiaux sont les entités du contexte de définition appliquées à un nombre maximal de variables libres. Ainsi, on ne considère que des termes construits avec les **constructeurs** du formalisme de définition, à partir de **variables libres** et **d'entités du contexte de définition**.

GLEF pourrait aussi permettre à l'utilisateur de faire des **suppositions**. Une *supposition* est un lemme que l'utilisateur emploie avant de le démontrer. Contrairement à une hypothèse, une supposition

n'est pas "argument" des preuves qui l'utilisent. Par exemple, la *planification* est une méthode qui consiste à construire une preuve en faisant des suppositions, à démontrer celles-ci en faisant d'autres suppositions, etc. jusqu'à la complétion de la preuve.

En pratique, faire une supposition revient à considérer l'existence d'un terme de type donné. Cela correspond à **ajouter** au contexte de définition une nouvelle entité qui soit de ce type. Par la suite, démontrer la supposition correspond à **transformer** cette entité en nommage.



13. Recherche automatique de symétries

Comme nous l'avons vu, le nommage sert à hiérarchiser les définitions pour les simplifier et simplifier leurs présentations. En particulier, il permet de factoriser les symétries "intéressantes" (voir chapitre Réalisation section A.1.g).

Le repérage **manuel** des symétries est **difficile**, notamment quand les variables libres à introduire dans le terme à factoriser sont de types divers. Par exemple, pour reconnaître une symétrie entre deux parties d'une preuve par résolution, on doit parfois changer des fonctions, des prédicats et même des clauses.

Décrivons le fonctionnement d'un **algorithme général de recherche** de symétries. Comme les algorithmes de recherche de lemmes [Lin88] ou de nommages optimaux de sous-formules [Boy92], il pourrait être intégré à GLEF.

Au lieu de rechercher directement des symétries au sein des définitions, l'algorithme peut **parcourir systématiquement** l'ensemble des termes mémorisés par GLEF pour repérer les termes deux à deux symétriques. Ce parcours est quadratique en fonction du nombre de termes.

A chaque pas, tester la symétrie de deux termes revient à tester l'égalité (définitionnelle) de leur structure de leur racine jusqu'à une certaine profondeur. Au-delà de cette profondeur, les variables libres des sous-termes non parcourus doivent être libres dans les termes.

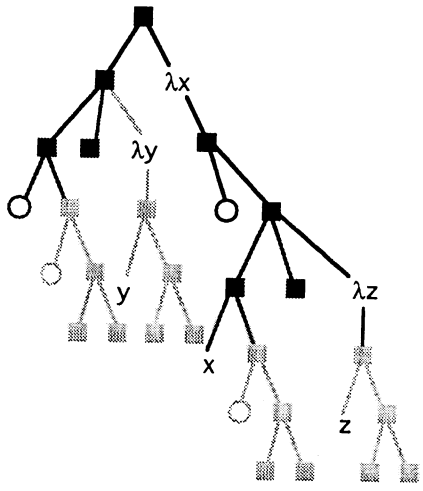
Dans le cas des symétries "fines", le test est similaire. Mais automatiser la recherche de ces symétries, comme éditer des preuves avec GLEF, est dangereux si le système formel employé n'est pas décrit entièrement, c'est à dire, quand sa définition est complète, mais incorrecte.

Le plus souvent, une définition contient un grand nombre de symétries. La caractérisation des symétries "intéressantes" peut considérer la taille du terme factorisant, le fait qu'il soit maximal, le nombre de ses variables, l'homogénéité des types de ces variables, la fréquence d'utilisation du terme factorisant ou des termes symétriques, etc..

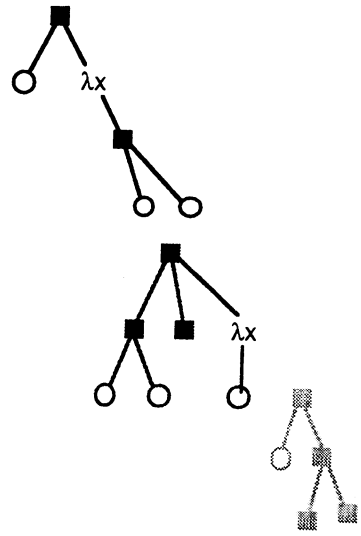
Le **nommage** servirait à factoriser les symétries "intéressantes" découvertes par l'algorithme. Rappelons que lorsqu'une symétrie n'est pas une égalité, le terme nommé est une λ-abstraction. Une **parure** pourrait être associée à certains de ces nommages.

L'utilisateur pourrait intervenir dans le choix des symétries finalement factorisées, des noms à leur donner et des éventuelles parures à leur associer.

Dans sa présentation, un terme appartenant à une symétrie factorisée est considéré comme une application du terme factorisant la symétrie. Quand une parure est associée au nommage qui introduit ce terme, l'utilisateur peut visualiser séparément **les termes symétriques**, **le terme factorisant** et **les applications de ce terme**.



Définition avant
factorisation des symétries



Même définition après
factorisation des symétries

14. Construction de preuves par analogie

Toujours dans l'idée de manipuler des preuves dans un cadre logique général, on peut tenter de découvrir des preuves par transformation de preuves connues. Cette méthode vient de la pratique mathématique où l'on constate que les différentes habilités que l'on peut avoir à développer les preuves dépendent fortement de la connaissance acquise sur une théorie particulière. Cette connaissance ne concerne pas seulement les théorèmes de la théorie, mais surtout leurs preuves. On emploie souvent les mêmes techniques de preuves, pour obtenir des résultats similaires ou analogues, dans une même théorie.

Considérons donc qu'il s'agit d'un processus de transformation de preuves que l'on peut essayer de rendre formel au sein de GLEF. La transformation peut être établie par une règle de transformation que l'on peut exprimer comme une règle d'inférence dont les prémisses et la conclusion feraient référence à des preuves, pas à des formules. Ce qui est d'ailleurs proche de ce que l'on pratique en déduction naturelle. Cette "règle d'inférence" serait appliquée grâce à un algorithme de filtrage d'ordre trois [BK92].

15. Utilisation simultanée de plusieurs logiques

Pour clore cette liste, non limitative, de projets futurs, citons la possibilité de transférer une preuve d'un système formel à un autre, tout en conservant sa structure[26].

Pour réaliser ce transfert, on pourrait, par exemple, déterminer une correspondance entre les entités syntaxiques des deux systèmes formels. Ensuite, on pourrait remplacer dans la preuve les constructeurs qui représentent les entités syntaxiques du système formel source par les constructeurs qui représentent les entités syntaxiques correspondantes du système formel destination, en passant par une représentation intermédiaire non typée de la preuve, c'est à dire un λ-terme pur.

Une autre méthode, plus ambitieuse, pourrait être de considérer cette forme de transfert comme une généralisation, au niveau de la définition des systèmes formels, de la construction de preuves par analogie.

B) Conclusion

L'étude du travail futur montre que GLEF n'est pas encore un système fini. Notamment, il lui manque quelques caractéristiques, comme les tactiques, qui existent dans de nombreux autres systèmes, et pourraient lui être ajoutées relativement rapidement. Toutefois, GLEF

---

[26]Cette idée nous a été suggérée par C.Morgan de l'université de Victoria, CANADA.

est caractérisé par de nombreuses propriétés et possibilités intéressantes et inexistantes dans les autres systèmes, et il a déjà servi à quelques utilisateurs.

En guise de conclusion, nous parlerons de ces utilisations et de celles prévues à différents termes.

Ainsi, GLEF a d'abord servi à présenter de manière lisible, c'est à dire graphique, structurée et hiérarchique, des preuves obtenues par des démonstrateurs automatiques [CDH91], [CHZ91], [CDH93]. Le chapitre Utilisation montre quelques exemples de preuves obtenues avec des démonstrateurs automatiques et présentées par GLEF.

GLEF a aussi servi à saisir, vérifier et présenter des preuves écrites à la main, toujours afin de les insérer dans une publication [HO94]. Notons que dans ces preuves, il a découvert une erreur qui était passée inaperçue au logicien.

Actuellement, dans le cadre de GLEF, une personne travaille sur la recherche automatique de symétries, une autre travaille sur la construction de preuves par analogie et une troisième envisage de le compléter par une interface utilisateur générique et homogène pour régler les paramètres des différents outils d'inférence. Personnellement, j'aimerai prendre le temps de rendre GLEF très facile à utiliser, tant au niveau de l'interface utilisateur qu'au niveau des manipulations formelles.

Pas à pas, nous espérons pouvoir aborder les autres évolutions décrites en travail futur.

Annexe A

Un Algorithme d'Effacement Original

Introduction

On appelle *terme généralisé*, un terme abstrait t ou un type ensemble T, dans la notation de De Bruijn (Voir chapitre Réalisation). Un terme généralisé doit donc toujours être considéré dans un contexte Γ. Ainsi, on appelle *référence généralisée* un couple formé d'un terme généralisé et d'un contexte.

Un *sélecteur d'occurrence* est un mot clef qui désigne un des arguments d'un des constructeurs de terme abstrait. Un *sélecteur d'occurrence généralisée* est soit un sélecteur d'occurrence, soit le mot clef $\mathtt{T}_i$ qui désigne le $i^{\text{ième}}$ élément d'un type (bien qu'un type soit un ensemble, il est mémorisé comme une liste), soit le mot clef $\mathtt{T}$ qui désigne le type d'un terme abstrait. Une *occurrence* (resp. *occurrence généralisée*) est une liste de sélecteurs d'occurrences (resp. sélecteurs d'occurrences généralisées).

Etant donnés une référence généralisée r_0 et un ensemble L_0 d'occurrences généralisées de r_0, le **but** de l'effacement est de trouver une référence généralisée, la moins générale possible, dont r_0 soit une instance, et dont les occurrences généralisées contenues dans L_0 existent et correspondent à des variables libres différentes de celles de r_0.

A) Définitions et notations

[] dénote la liste vide. :: dénote la liste de tête e et de queue l. $l_1 @ l_2$ dénote la concaténation des listes l_1 et l_2.

Dans un programme, l'instruction $\exists \uparrow x \in X$ crée une variable locale x contenant un élément de X, et renvoie la valeur booléenne de la formule $\exists x \in X$. L'instruction `erreur` interrompt le programme, qui renvoie aussitôt une erreur. L'instruction `attrape` permet de protéger une suite d'instructions, elle renvoie `vrai` si une erreur survient, `faux` sinon.

Etant donnés deux ensembles E_1 et E_2 et un opérateur binaire $*$ sur $E_1 \times E_2$, on définit les opérateurs binaires *extensions* de $*$, notés $\bar{*}, *\bar{}$ et $\bar{*}$, et définis respectivement sur:

$$P(E_1)\times E_2,\ E_1\times P(E_2),\ \text{et}\ P(E_1)\times P(E_2)$$

Pour tout $o_1\in E_1, o_2\in E_2, E_1'\in \mathrm{P}(E_1), E_2'\in \mathrm{P}(E_2)$:

$$E_1'\bar{*}o_2=\{o_1*o_2 \mid o_1\in E_1' \wedge \texttt{attrape}(o_1*o_2)=\texttt{faux}\}$$

$$o_1\vec{*}E_2'=\{o_1*o_2 \mid o_2\in E_2' \wedge \texttt{attrape}(o_1*o_2)=\texttt{faux}\}$$

$$E_1'\bar{\vec{*}}E_2'=\{o_1*o_2 \mid o_1\in E_1' \wedge o_2\in E_2' \wedge \texttt{attrape}(o_1*o_2)=\texttt{faux}\}$$

En particulier, nous emploierons les opérateurs binaires extensions des opérateurs ::, @, / (/ est défini plus loin). Quand aucun autre opérateur $*$ n'est défini sur $P(E_1)\times E_2, E_1\times P(E_2)$, ou $P(E_1)\times P(E_2)$, le symbole $*$ peut respectivement dénoter $\bar{*}, \vec{*}$ ou $\bar{\vec{*}}$.

Racines des noms de variables utilisés:

t	terme abstrait ou généralisé.
T	type.
Γ	contexte.
r	référence généralisée.
s	sélecteur d'occurrence.
S	sélecteur d'occurrence généralisée.
o	occurrence.
O	occurrence généralisée.
l	ensemble d'occurrences.
L	ensemble d'occurrences généralisées.

$S_{OG}=\{\texttt{T},\texttt{PI},\texttt{PB},\texttt{L},\texttt{AI},\texttt{AB},\texttt{M}_i,\texttt{F},\texttt{A},\texttt{T}_i\}$ dénote l'ensemble des sélecteurs d'occurrences généralisées.

$t{:}T$ dénote le terme abstrait t, et indique que le jugement $\Gamma\vdash t{:}T$ est vrai, Γ étant sous-entendu.

Dans la suite, tous les termes, les références et les occurrences considérés seront **sous-entendus généralisés**, sauf indication contraire.

Occurrences d'un terme:

Constructeur du terme	Forme générale du terme. Quand c'est un terme abstrait, son type est indiqué.	Sélecteurs d'occurrences possibles	Sous-termes "directs" correspondants
variable	$x{:}T$	T	T
produit	$[x{:}T_1]t{:}T_2$	PI, PB, T	T_1, t, T_2
liste	$T_1{+}{:}T_2$	L, T	T_1, T_2
abstraction	$<x{:}T_1>t{:}T_2$	AI, AB, T	T_1, t, T_2
multi-ensemble	$\{t_1\ldots,t_n\}{:}T$	$M_1\ldots, M_n$, T	$t_1\ldots, t_n, T$
application	$(t_1\ t_2){:}T$	F, A, T	t_1, t_2, T
type ensemble	$\{t_1\ldots,t_n\}$	$T_1\ldots, T_n$	$t_1\ldots, t_n$

gterm, gref, gocc, context dénotent respectivement les ensembles des termes généralisés, des références généralisées, des occurrences généralisées, et des contextes.

La référence située à l'occurrence O de la référence r, notée r/O est définie récursivement par (notons que pour définir ou appeler une fonction dont un argument est une référence, on peut utiliser un terme et un contexte comme arguments):

$$
\begin{aligned}
&(t,\Gamma)/[] = (t,\Gamma)\\
&([x{:}T]t,\Gamma)/\mathrm{PI}::O = (T,\Gamma)/O\\
&([x{:}T]t,\Gamma)/\mathrm{PB}::O = (t,\Gamma[x{:}T])/O\\
&(T{+},\Gamma)/\mathrm{L}::O = (T,\Gamma)/O\\
&(<x{:}T>t,\Gamma)/\mathrm{AI}::O = (T,\Gamma)/O\\
&(<x{:}T>t,\Gamma)/\mathrm{AB}::O = (t,\Gamma[x{:}T])/O\\
&(\{t_1\ldots,t_n\},\Gamma)/\mathrm{M}_i::O = (t_i,\Gamma)/O\\
&((t_1\ t_2),\Gamma)/\mathrm{F}::O = (t_1,\Gamma)/O\\
&((t_1\ t_2),\Gamma)/\mathrm{A}::O = (t_2,\Gamma)/O\\
&(t{:}T,\Gamma)/\mathrm{T}::O = (T,\Gamma)/O\\
&(\{t_1\ldots,t_n\},\Gamma)/\mathrm{T}_i::O = (t_i,\Gamma)/O\\
&(\_,\_)/O = \mathtt{erreur}
\end{aligned}
$$

Où _ dénote une expression quelconque.

Réduction O/i de l'occurrence O par l'entier i.

$$\begin{cases} O / 0 = O \\ S::O / i+1 = O / i \\ [\,] / i = \texttt{erreur} \quad (i \neq 0) \end{cases}$$

Réduction O_1 / O_2 de l'occurrence O_1 par l'occurrence O_2:

$$\begin{cases} O / [\,] = O \\ S::O_1 / S::O_2 = O_1 / O_2 \\ \_ / \_ = \texttt{erreur} \end{cases}$$

Calcul des occurrences $\mathrm{L}(r) = \{O \mid \texttt{attrape}(r / O) = \texttt{faux}\}$ de la référence r:

$$\begin{cases} \mathrm{L}(k{:}T,\Gamma) = \{[\,]\} \cup \mathrm{T}::\mathrm{L}(T,\Gamma) \\ \mathrm{L}([x{:}T_1]t{:}T_2,\Gamma) = \mathrm{PI}::\mathrm{L}(T_1,\Gamma) \cup \mathrm{PB}::\mathrm{L}(t,\Gamma[x{:}T_1]) \cup \mathrm{T}::\mathrm{L}(T_2,\Gamma) \\ \mathrm{L}(T_1{+}{:}T_2,\Gamma) = \mathrm{L}::\mathrm{L}(T_1,\Gamma) \cup \mathrm{T}::\mathrm{L}(T_2,\Gamma) \\ \mathrm{L}(<x{:}T_1>t{:}T_2,\Gamma) = \mathrm{AI}::\mathrm{L}(T_1,\Gamma) \cup \mathrm{AB}::\mathrm{L}(t,\Gamma[x{:}T_1]) \cup \mathrm{T}::\mathrm{L}(T_2,\Gamma) \\ \mathrm{L}(\{t_1\ldots,t_n\}{:}T,\Gamma) = \bigcup_{i=1}^{n} \mathrm{M}_i::\mathrm{L}(t_i,\Gamma) \cup \mathrm{T}::\mathrm{L}(T,\Gamma) \\ \mathrm{L}((t_1\ t_2){:}T,\Gamma) = \mathrm{F}::\mathrm{L}(t_1,\Gamma) \cup \mathrm{A}::\mathrm{L}(t_2,\Gamma) \cup \mathrm{T}::\mathrm{L}(T,\Gamma) \\ \mathrm{L}(\{t_1\ldots,t_n\},\Gamma) = \bigcup_{i=1}^{n} \mathrm{T}_i::\mathrm{L}(t_i,\Gamma) \end{cases}$$

On définit aussi:

$$\mathrm{L}_n(r) = \{O \in \mathrm{L}(r) \mid |O| = n\}$$

Calcul des occurrences $\mathrm{L}(k,r) = \{O \mid \texttt{attrape}(r / O) = \texttt{faux} \wedge r / O = k\}$ de la variable k, dans la référence r:

$$
\begin{cases}
\mathrm{L}(k,k{:}T,\Gamma) = \{[]\} \cup \mathrm{T}::\mathrm{L}(k,T,\Gamma) \\
\mathrm{L}(k_1,k_2{:}T,\Gamma) = \mathrm{T}::\mathrm{L}(k_1,T,\Gamma) \quad (k_1 \neq k_2) \\
\mathrm{L}(k,[x{:}T_1]t{:}T_2,\Gamma) = \mathrm{PI}::\mathrm{L}(k,T_1,\Gamma) \cup \mathrm{PB}::\mathrm{L}(k+1,t,\Gamma[x{:}T_1]) \cup \mathrm{T}::\mathrm{L}(k,T_2,\Gamma) \\
\mathrm{L}(k,T_1{+}{:}T_2,\Gamma) = \mathrm{L}::\mathrm{L}(k,T_1) \cup \mathrm{T}::\mathrm{L}(k,T_2,\Gamma) \\
\mathrm{L}(k,<x{:}T_1>t{:}T_2,\Gamma) = \mathrm{AI}::\mathrm{L}(k,T_1,\Gamma) \cup \mathrm{AB}::\mathrm{L}(k+1,t,\Gamma[x{:}T_1]) \cup \mathrm{T}::\mathrm{L}(k,T_2,\Gamma) \\
\mathrm{L}(k,\{t_1\ldots,t_n\}{:}T,\Gamma) = \bigcup_{i=1}^{n} \mathrm{M}_i::\mathrm{L}(k,t_i,\Gamma) \cup \mathrm{T}::\mathrm{L}(k,T,\Gamma) \\
\mathrm{L}(k,(t_1\ t_2){:}T,\Gamma) = \mathrm{F}::\mathrm{L}(k,t_1,\Gamma) \cup \mathrm{A}::\mathrm{L}(k,t_2,\Gamma) \cup \mathrm{T}::\mathrm{L}(k,T,\Gamma) \\
\mathrm{L}(k,\{t_1\ldots,t_n\},\Gamma) = \bigcup_{i=1}^{n} \mathrm{T}_i::\mathrm{L}(k,t_i,\Gamma)
\end{cases}
$$

On définit aussi:

$$\mathrm{LT}(k,r) = \mathrm{L}(k,r)@_{[\mathrm{T}]}$$

Fermeture et ouverture d'une référence r par des produits. Notons que la fermeture de références dont les termes sont des multi-ensembles, des abstractions ou des produits n'est **pas légale** dans le formalisme de définition. Toutefois, pour ne pas introduire de nouveau constructeur et garder une expression simple de l'algorithme d'effacement, on suppose que c'est possible au sein de la fonction `erase`.

$$
\begin{cases}
\texttt{close\_pi}(t,\Gamma_0) = (t,\Gamma_0) \\
\texttt{close\_pi}(t,\Gamma[x{:}T]) = \texttt{close\_pi}([x{:}T]t,\Gamma) \\
\texttt{close\_pi}(\_,\_) = \texttt{erreur}
\end{cases}
$$

$$
\begin{cases}
\texttt{open\_pi}(t,\Gamma,0) = (t,\Gamma) \\
\texttt{open\_pi}([x{:}T]t,\Gamma,i+1) = \texttt{open\_pi}(t,\Gamma[x{:}T],i) \\
\texttt{open\_pi}(\_,\_,\_) = \texttt{erreur}
\end{cases}
$$

B) Algorithme d'effacement:

Le principe de l'algorithme d'effacement est de propager dans r_0, les occurrences à effacer, en tenant compte des contraintes locales et non locales dans r_0. Parmi ces occurrences on distingue différentes classes d'équivalence, qui correspondent chacune à une variable à introduire. Ces classes d'équivalence sont représentées par des ensembles que l'on réunit si leur intersection ne peut pas être vide. Quand le premier argument d'une application, des occurrences d'une constante ou des occurrences mal ciblées (le sélecteur d'occurrence ne correspond pas au constructeur du terme) doivent être effacés, on doit effacer le terme

courant. Pour cela, il faut ajouter un nouvel ensemble, qui contient son occurrence.

La solution la plus simple aurait été de traiter les variables libres de r_0 comme les constantes. Mais il est intéressant de pouvoir généraliser leurs types, comme ceux des variables liées. Pour homogénéiser les traitements des variables libres et des variables liées, on effectue des traitements supplémentaires avant et après la propagation. Dans la fonction `erase`, ces traitements correpondent aux lignes contenant les appels des fonctions `open_pi` et `close_pi`. Pour que les variables libres soient traitées comme des constantes, il suffit d'ôter ces lignes.

Constantes:

Γ_0	Contexte d'édition.
r_0	Référence objet.
L_0	Ensemble des occurrences à effacer.

Variables globales, pour les fonctions, le symbole $\cdot$ dénote l'emplacement d'un argument:

Λ	Ensemble des sous-termes "directs" du terme courant. Cet ensemble dépend du constructeur de plus haut niveau de ce terme.
$S_\cdot \in \Lambda \to \text{gocc}$	S_λ: Occurrence du sous-terme "direct" λ dans le terme courant.
$C_\cdot \in \Lambda \to \text{context}$	C_λ: Contexte dont on doit prolonger le contexte du terme courant pour obtenir le contexte de son sous-terme "direct" λ.
$M_{\cdot\cdot} \in \Lambda \to \Lambda \to \text{P}(\text{gocc}) \to \text{P}(\text{gocc})$	$M_{\lambda\mu}$: Fonction permettant de transformer un ensemble d'occurrences du terme μ en un ensemble d'occurrences du terme λ. Cette fonction représente les dépendances entre deux sous-termes "directs" du terme courant. Elle est donnée sous la forme d'un tableau.
$max \in \text{N}$	Indice maximal des variables à effacer.
$v_\cdot \in \text{N}_{max} \to \text{N}_{max}$	v_i: Pointeur vers la $i^{\text{ième}}$ variable à effacer.

Appel de l'agorithme:

$r := \texttt{erase}(r_0, L_0)$;

Définition de `erase`:

```
/* Fonction principale d'effacement */
erase(r ∈ gref, L ∈ P(gocc)) ∈ gref{
   /* Variables locales */
    i ∈ N                          Compteur d'occurrences.
```

$O \in \text{gocc}$

$t' \in \text{gterm}$ — Terme de la référence calculée.

$\Gamma' \in \text{gterm}$ — Contexte de la référence calculée.

$n \in \mathrm{N}$ — Compteur des variables introduites.

$r_T \in \text{gref}$ — Référence correspondant au type de la variable considérée.

$u \in \mathrm{N}$ — Identifiant de variable introduite.

$L^{\cdot} \in \mathrm{N}_{max} \rightarrow \mathrm{P}(\text{gocc})$ — Ensemble d'occurrences correspondant à chaque pointeur de variable introduite.

$L^{*} \in \mathrm{N}_{max} \rightarrow \mathrm{P}(\text{gocc})$ — Ensemble d'occurrences correspondant à chaque variable introduite.

$v \in \mathrm{N}$ — Nombre de variables libres de la référence.

```
/* Calcul du nombre de variables libres */
v := |γ̄(r0)| - |Γ0|;
/* Elimination des variables libres, pour pouvoir
   généraliser leurs types */
r := close_pi(r);
/* Construction de l'ensemble initial des variables
   à effacer */
i := 1;
tant que ∃↑ O ∈ L {
      v_i := i;
      L^i := {O};
      i++;
}
max := i - 1;
/* Erreur d'occurrence */
si ∃i ∈ N_max |r / L^i| ≠ |L^i| alors
      erreur;
/* Propagation des occurrences */
L· := propagate(r, L·);
Γ' := γ̄(r);
t' := t̄(r);
n := 1;
L^u := ⋃_{i∈N_max ∧ v_i=u} L^{v_i};  (u ∈ N_max)
/* Tri topologique des variables introduites */
tant que ∃↑u ∈ N_max ∃↑O ∈ L^u (L^u \ O) / O = ∅ {
      /* Comme:
            (∃O ∈ L^u (L^u \ O) / O = ∅)
            ⇒ (∀O ∈ L^u (L^u \ O) / O = ∅)
      cette condition equivaut à:
            ∃u ∈ N_max L^u ≠ ∅ ∧ (∀O ∈ L^u (L^u \ O) / O = ∅)
      */
```

$r_T := r / (O@[\mathrm{T}])$;

$\Gamma' := \Gamma'\left[\mathrm{var}:\bar{\mathrm{t}}(r_T) + \left(|\Gamma'| - |\bar{\gamma}(\mathrm{r}_0)|\right)\right]$;

$t' := t'[n / O]$; $\left(O \in L^n\right)$

$n{+}{+}$;

$L^n := \varnothing$;

}

$(t', \Gamma') :=$ open_pi(t', Γ', v);

retourner $\bar{\mathrm{r}}(t', \Gamma')$;

}

Définition de `propagate`:

```
/* Calcul des occurrences à remplacer par des
   variables libres. */
```

propagate$(t \in \mathrm{gterm}, \Gamma \in \mathrm{context},$

$$L^{\cdot} \in \mathrm{N}_{max} \to \mathrm{P}(\mathrm{gocc})) \in \mathrm{N}_{max} \to \mathrm{P}(\mathrm{gocc})\{$$

```
/* Variables locales */
```

$L^{\cdot}_{\cdot}, L^{\cdot}_{\cdot} \in \mathrm{N}_{max} \to \Lambda \to \mathrm{P}(\mathrm{gocc})$

$L \in \mathrm{P}(\mathrm{gocc})$

$erased \in \mathrm{Booleen}$

```
/* Initialisation */
```

fill(t, Γ);

$L^i_\lambda := L^i / S_\lambda$; $\left(i \in \mathrm{N}_{max} \wedge \lambda \in \Lambda\right)$

$erased :=$ faux;

répéter {

$L := \bigcup_{i \in \mathrm{N}_{max}} L^i$;

```
/* Test: application, constante
ou occurrence mal ciblée */
```

si $\neg erased \wedge \left(\begin{array}{l} [\mathrm{F}] \in L \\ \vee \left(t = k \wedge \left(|\Gamma| - |\Gamma_0|\right) \le k \wedge L \neq \varnothing\right) \\ \vee \bigcup_{i \in \mathrm{N}_{max}} \left(L^i \setminus \bigcup_{\lambda \in \Lambda} S_\lambda @ L^i_\lambda \right) \neq \varnothing \end{array} \right)$

alors {

```
/* Ajout d'une nouvelle variable */
```

$L^i := [\mathrm{T}]@\left(L^i / [\mathrm{T}]\right);\ \ (i \in \mathrm{N}_{max})$

$max++;$

$v_{max} := max;$

$L^{max} := \{[]\};$

fill(1,Γ);

$L^i_\lambda := L^i / S_\lambda;\ \ (i \in \mathrm{N}_{max} \wedge \lambda \in \Lambda)$

$erased := \mathrm{vrai};$

}

/* Propagation */

$L^\bullet_\bullet := L^\bullet_\bullet;$

$L^i_\lambda := \bigcup_{\mu \in \Lambda} M_{\lambda\mu}\left(L^i_\lambda\right);\ \ (i \in \mathrm{N}_{max} \wedge \lambda \in \Lambda)$

wrap($L^\bullet_\bullet$);

/* Appel récursif */

$L^\bullet_\lambda := \mathrm{propagate}(\lambda, \Gamma C_\lambda, L^\bullet_\lambda);\ \ \left(\lambda \in \Lambda \wedge L^\bullet_\lambda \neq L^{\bullet\bullet}_\lambda\right)$

/* Reconstruction de $L^\bullet$ */

$L^i := \bigcup_{\lambda \in \Lambda} S_\lambda @ L^i_\lambda;\ \ (i \in \mathrm{N}_{max})$

} jusqu'à $L^\bullet_\bullet = L^{\bullet\bullet}_\bullet;$

retourner $L^\bullet;$

}

Définition de `wrap`:

/* Egalisation de variables */

wrap($L^\bullet_\bullet \in \mathrm{N}_{max} \to \Lambda \to \mathrm{P}(\mathrm{gocc})$) {

$i, j \in \mathrm{N}$ Indices de pointeurs de variables.

tant que $\exists i, j \in \mathrm{N}_{max}\ \exists \lambda \in \Lambda\ L^i_\lambda \cap L^j_\lambda \neq \varnothing \wedge v_i \neq v_j$

$v_i := v_j;$

}

Définition de `fill`:

/* Remplissage des tableaux
en fonction du constructeur de plus
haut niveau du terme courant */

fill($t \in \mathrm{gterm}, \Gamma \in \mathrm{context}$) {

au cas

• où $t = [x{:}T_0]t_0$ {

$\Lambda := \{t_0, T_0\}$

S_λ:

λ	t_0	T_0
S_λ	[PB]	[PI]

$M_{\lambda\mu}$:

$\lambda \setminus \mu$	t_0	T_0
t_0	x	$\mathrm{LT}(1,t_0)@x$
T_0	$x / \mathrm{LT}(1,t_0)$	x

}

• où $t = T_0 +$ {

$\Lambda := \{T_0\}$

S_λ:

λ	T_0
S_λ	[L]

$M_{\lambda\mu}$:

$\lambda \setminus \mu$	T_0
T_0	x

}

• où $t = \langle x{:}T_0 \rangle t_0 : \{[x{:}T_1]t_1 \ldots, [x{:}T_n]t_n\}$ {

$\Lambda := \{t_0, T_0, t_1, T_1 \ldots, t_n, T_n\}$

S_λ:

λ	t_0	T_0	t_i	T_i
S_λ	[PB]	[PI]	[T;T_i;PB]	[T;T_i;PI]

$(i \in \mathrm{N}_n)$

$M_{\lambda\mu}$:

$\lambda \setminus \mu$	t_0	t_i	t_j	T_0, T_i, T_j
t_0	x	[T;T_i]@x	[T;T_j]@x	$\mathrm{LT}(1,t_0)@x$
t_i	x / [T;T_i]	x	$\mathrm{LT}(1,t_i)@(x / \mathrm{LT}(1,t_j))$	$\mathrm{LT}(1,t_i)@x$
t_j	x / [T;T_j]	$\mathrm{LT}(1,t_j)@(x / \mathrm{LT}(1,t_i))$	x	$\mathrm{LT}(1,t_j)@x$
T_0, T_i, T_j	$x / \mathrm{LT}(1,t_0)$	$x / \mathrm{LT}(1,t_i)$	$x / \mathrm{LT}(1,t_j)$	x

$(i, j \in \mathrm{N}_n \wedge i \neq j)$

}

• où $t = \{a_1 \ldots, a_n\} : \{\{A_1 \ldots, A_p\}+\}$ {

soit $j_i := j \in \mathrm{N}_p \mid a_i : A_j$; $(i \in \mathrm{N}_n)$

$\Lambda := \{a_1 \ldots, a_n, A_1 \ldots, A_p\}$

S_λ:

λ	a_i	A_j
S_λ	[M_i]	[T;T_1;L;T_j]

$(i \in \mathrm{N}_n \wedge j \in \mathrm{N}_p)$

$M_{\lambda\mu}$:

$\lambda \setminus \mu$	a_i	a_j	a_k	A_{j_i}, A_{j_j}	A_k
a_i	x	[T]@$(x$ / [T])	$\varnothing$	[T]@x	$\varnothing$
a_j	[T]@$(x$ / [T])	x	$\varnothing$	[T]@x	$\varnothing$
a_k	$\varnothing$	$\varnothing$	x	$\varnothing$	[T]@x
A_{j_i}, A_{j_j}	x / [T]	x / [T]	$\varnothing$	x	$\varnothing$
A_k	$\varnothing$	$\varnothing$	x / [T]	$\varnothing$	x

$(i, j, k \in \mathrm{N}_n \wedge (i \neq j \wedge i \neq k \wedge i \neq k) \wedge (j_i = j_j \wedge j_i \neq j_k))$

• où $t = \{t_1 \ldots, t_n\}$ {

$\Lambda := \{t_1 \ldots, t_n\}$

S_λ:

λ	t_i
S_λ	[T_i]

$(i \in \mathrm{N}_n)$

$M_{\lambda\mu}$:

$\lambda \setminus \mu$	t_i	t_j
t_i	x	$\varnothing$
t_j	$\varnothing$	x

$(i, j \in \mathrm{N}_n \wedge i \neq j)$

}

• où $t = k{:}T_0$ {

/* Variable (liée, libre ou constante) */

$\Lambda := \{T_0\}$

S_λ:

λ	T_0
S_λ	[T]

$M_{\lambda\mu}$:

$\lambda \setminus \mu$	T_0
T_0	x

}

}

• où $t = (f\ a){:}type\_apply(T_f, T_a)$ {

$\Lambda := \{f, a, T_f, T_a\}$

S_λ:

λ	f	a	T_f	T_a
S_λ	[F]	[A]	[F;T]	[F;A]

$M_{\lambda\mu}$:

$\lambda \setminus \mu$	f	a	T_f	T_a
f	x	[T]@ta_a2f(T_f,T_a,x/[T])	[T]@x	[T]@ta_a2f(T_f,T_a,x)
a	[T]@ta_f2a(T_f,T_a,x/[T])	x	[T]@ta_f2a(T_f,T_a,x)	[T]@x
T_f	x/[T]	ta_a2f(T_f,T_a,x/[T])	x	ta_a2f(T_f,T_a,x)
T_a	ta_f2a(T_f,T_a,x/[T])	x/[T]	ta_f2a(T_f,T_a,x)	x

}

C_λ := si S_λ = [AI] $\vee$ S_λ = [PI] alors $[x{:}T_0]$ sinon *; $(\lambda \in \Lambda)$

}

Pour les définitions de ta_f2a(T_1, T_2, L), et ta_a2f(T_1, T_2, L), on suppose que type_apply(T_1, T_2, L) soit déjà vérifié, et que $|T_1 / L| = |L|$.

Définition de ta_f2a(T_1, T_2, L):

$$\mathtt{ta\_f2a}(T_1, T_2, L) = \bigcup_{i=1}^{|T_1|} \mathtt{tc\_f2a}(T_1/[\mathrm{T}_i;\mathtt{PI}], T_2, L/[\mathrm{T}_i;\mathtt{PI}])$$

$$\mathtt{tc\_f2a}(T_1, T_2, L) = \bigcup_{(i,j)=(1,1)}^{(|T_1|,|T_2|)} \mathrm{T}_j{::}\mathtt{to\_f2a}(T_1, T_2, L/[\mathrm{T}_i])$$

$$\texttt{tc'\_f2a}(T_1, T_2, L) = \bigcup_{(i,j)=(1,1)}^{(|T_1|,|T_2|)} \texttt{T}_j\texttt{::to\_f2a}(T_1, T_2, L/[\texttt{T}_i])$$

$$\begin{cases} \texttt{to\_f2a}(t,t,L) = L \\ \texttt{to\_f2a}([x{:}T_1]t_1', [x{:}T_2]t_2', L) = \begin{matrix}\texttt{PI::tc\_f2a}(T_1, T_2, L/[\texttt{PI}]) \\ \cup\ \texttt{PB::to\_f2a}(T_1, T_2, L/[\texttt{PB}])\end{matrix} \\ \texttt{to\_f2a}(T_1+, T_2+, L) = \texttt{L::tc\_f2a}(T_1, T_2, L/[\texttt{L}]) \\ \texttt{to\_f2a}(\_,\_,\_) = \varnothing \end{cases}$$

Définition de $\texttt{ta\_a2f}(T_1, T_2, L)$:

$$\texttt{ta\_a2f}(T_1, T_2, L) = \bigcup_{i=1}^{|T_1|} \texttt{T}_i\texttt{::PI::tc\_a2f}(T_1/[\texttt{T}_i;\texttt{PI}], T_2, L)$$

$$\texttt{tc\_a2f}(T_1, T_2, L) = \bigcup_{(i,j)=(1,1)}^{(|T_1|,|T_2|)} \texttt{T}_i\texttt{::to\_a2f}(T_1, T_2, L/[\texttt{T}_j])$$

$$\texttt{tc'\_a2f}(T_1, T_2, L) = \bigcup_{(i,j)=(1,1)}^{(|T_1|,|T_2|)} \texttt{T}_i\texttt{::to\_a2f}(T_1, T_2, L/[\texttt{T}_j])$$

$$\begin{cases} \texttt{to\_a2f}(t,t,L) = L \\ \texttt{to\_a2f}([x{:}T_1]t_1', [x{:}T_2]t_2', L) = \begin{matrix}\texttt{PI::tc\_a2f}(T_1, T_2, L/[\texttt{PI}]) \\ \cup\ \texttt{PB::to\_a2f}(T_1, T_2, L/[\texttt{PB}])\end{matrix} \\ \texttt{to\_a2f}(T_1+, T_2+, L) = \texttt{L::tc\_a2f}(T_1, T_2, L/[\texttt{L}]) \\ \texttt{to\_a2f}(\_,\_,\_) = \varnothing \end{cases}$$

C) Exemples

Dans les exemples qui suivent, les indices indiquent les sous-termes effacés et l'ordre dans lequel ils le sont. L'indice 1 correspond au sous-terme dont l'effacement est demandé. Ainsi, le premier exemple est l'effacement de `A⇒A` dans la preuve suivante:

```
(mp A⇒A⇒A A⇒A1
    (a1 A A)
    (mp A⇒(A⇒A)⇒A4  (A⇒A⇒A)⇒A⇒A2
          (a1 A A⇒A)5
          (a2 A A⇒A A3)))
```

Le deuxième exemple est l'effacement de la première instance de `formula` dans le terme suivant:

```
(map formula1 formula8 <a:formula4>a5 {P,Q}4.)
```

Dans cet exemple, des occurrences généralisées doivent être effacées, au sein même du type d'un constructeur `map`, qui est:

$$[A_2, B_7 : *][f : [A_3]B_6][A_{+3\cdot}]B_{+7\cdot}]$$

Annexe B

Carte syntaxique du langage de présentation

```
<file>              ::=  <presentations>
<presentations>     ::=  ε
                    |    <presentations> ';' <presentation>
                    |    <presentations> ';' <assignment>
<assignment>        ::=  ASSIGN <name> <name> <integer>
                    |    ASSIGN <name> <name> \<char>.
<presentation> ::=  <names> ':' <rules_block>
<rules_block>       ::=  'BEGIN' <rules> 'END'.
<cases_block>       ::=  'BEGIN' <cases> 'END'.
<rules>             ::=  ε | <rules> <rule> ';'.
<rules>             ::=  'PRESENT' ':' <name>
                    |    <ivalue> '=' <ivalue>
                    |    'HIDE'
                    |    <content_type> <value_type>.
                    |    <attr_type> ':' <string>.
                    |    'ON' <integers> ':' <rules_block>
                    |    'DEFINITION' ':' <rules_block>
                    |    'SELECT' <occ> ':' <rules_block>
                    |    'PLACE' <occ> ':' <rules_block>
                    |    'IF' <bvalue> ':' <rules_block>
                    |    reverse 'MAP' ':' <cases_block>
                    |    reverse 'WHILE' <bvalue> <occ>
                              ':' <cases_block>.
                    |    'CREATE' ':' <rules_block>.
<cases>             ::=  ε | <cases> <case> ';'.
<case>              ::=  'CASE' <bvalue> <case_type>
                         ':' <rules_block>.
<occ>               ::=  <occ_type> | <occ_type> . <occ>.
<reverse>           ::=  ε | 'REVERSE'.
<field>             ::=  'LEFT' | 'HMIDDLE' | 'HREF' |
'RIGHT'
                    |    'TOP' | 'VMIDDLE' | 'VREF' |
'BOTTOM'
                    |    'HEIGTH' | 'WIDTH'.
<ivalue>            ::=  <integer>
                    |    <field>
                    |    <reference> '.' <field>
                    |    '-' <ivalue>
                    |    <ivalue> <isop> <ivalue>
                    |    <ivalue> <imop> <integer>
                    |    '(' <ivalue> ')'.
<isop>              ::=  '+' | '-'.
```

```
<imop>              ::=   '*' | '/' | '%'.
<bvalue>            ::= 'ISA' '(' <cons_id> ')'
                    |     'ISA' '(' <occ> ',' <cons_id> ')'
                    |     <bvalue> <bop> <bvalue>
                    |     '!' <bvalue>
                    |     '(' <bvalue> ')'.
<bop>               ::=   '|' | '&'.
<names>             ::=   <name> | <names> ',' <name>.
<integers>          ::=   <integer>
                    |     <integers> ',' <integer>.
<case_type>         ::= 'FIRST' | 'BETWEEN' | 'LAST'
                    |     'ALL'  |  'NONE'  |  'BEFORE'  |
'AFTER'.
<occ_type>          ::= 'IT' | 'TYPE' | 'TYPE1'
                    |     'REFERENCE_TYPE'        |
'REFERENCE_TYPE1'
                    |     'BLOC' | 'REFERENCE'
                    |     'FUNCTION' | 'ARGUMENT' | 'A'
                    |     'L_ARGUMENT' | 'E'
                    |     <integer>.
<content_type> ::=  'TEXT' | 'SYMBOL'
                    | 'PICTURE' | 'EXTERN'.
<value_type>        ::=   'NAME'
                    |     'REFERENCE_NAME' | 'POOL_NAME'.
<attr_type>         ::=   'STYLE' | 'GRAPHICS'.
<reference>         ::=   'ENCLOSING' | 'ENCLOSED'
                    | 'PREVIOUS' | 'SELF' | 'NEXT'.
<cons_id>           ::=   'term'
                    |     'universe'
                    |     'variable'
                    |     'introduction'
                    |     'product'
                    |     'abstraction'
                    |     'application'
                    |     'list'
                    |     'collection'
                    |     'mset'
                    |     'type'.
```

Avec les expressions régulières usuelles suivantes, décrivant les non terminaux `<name>`, `<integer>`, `<string>` et `<char>` dans la syntaxe de *LEX*:

```
DIGIT               [0-9]
LETTER              [_$\-a-zA-Z]
NAME                {LETTER}({LETTER}|{DIGIT})*
INTEGER             {DIGIT}+
STRING              '[^']*'
CHAR                .
```

Annexe C

Présentation d'un terme entièrement développée

Le code suivant est une présentation complètement développée du terme `(forall <x:term> (or (P x) (Q x)))` a présenter sous la forme $\boxed{(\forall x\,(P(x) \vee Q(x)))}$. En pratique, un tel code n'est jamais employé, mais il peut aider à comprendre le fonctionnement élémentaire du langage de présentation (emploi des occurrences généralisées, positionnement et dimensionnement relatif des boîtes). Pour une présentation plus concise du même terme, à l'aide de parures, voir chapitre Réalisation section A.3.

```
term:BEGIN
    HEIGHT=ENCLOSED.HEIGHT;
    WIDTH=ENCLOSED.WIDTH;
    TEXT '(':BEGIN
          LEFT=ENCLOSING.LEFT;
          TOP=ENCLOSING.TOP;
    END;
    SYMBOL NAME:BEGIN
          LEFT=PREVIOUS.RIGHT;
          TOP=PREVIOUS.TOP;
    END;
    SELECT A.1.REFERENCE:BEGIN
          LEFT=PREVIOUS.RIGHT;
          TOP=PREVIOUS.TOP;
          HEIGHT=ENCLOSED.HEIGHT;
          WIDTH=ENCLOSED.WIDTH;
          TEXT NAME:BEGIN
                LEFT=ENCLOSING.LEFT;
                TOP=ENCLOSING.TOP;
          END;
    END;
    SELECT A.1.BLOC:BEGIN
          LEFT=PREVIOUS.RIGHT+5;
          TOP=PREVIOUS.TOP;
          HEIGHT=ENCLOSED.HEIGHT;
          WIDTH=ENCLOSED.WIDTH;
          TEXT '(':BEGIN
                LEFT=ENCLOSING.LEFT;
                TOP=ENCLOSING.TOP;
          END;
          SELECT A.1:BEGIN
```

```
        LEFT=PREVIOUS.RIGHT;
        TOP=PREVIOUS.TOP;
        HEIGHT=ENCLOSED.HEIGHT;
        WIDTH=ENCLOSED.WIDTH;
        TEXT NAME:BEGIN
              LEFT=ENCLOSING.LEFT;
              TOP=ENCLOSING.TOP;
        END;
        TEXT '(':BEGIN
              LEFT=PREVIOUS.RIGHT;
              TOP=PREVIOUS.TOP;
        END;
        SELECT ARGUMENT:BEGIN
              LEFT=PREVIOUS.RIGHT;
              TOP=PREVIOUS.TOP;
              WIDTH=ENCLOSED.WIDTH;
              HEIGHT=ENCLOSED.HEIGHT;
              TEXT NAME:BEGIN
                    LEFT=ENCLOSING.LEFT;
                    TOP=ENCLOSING.TOP;
              END;
        END;
        TEXT ')':BEGIN
              LEFT=PREVIOUS.RIGHT;
              TOP=PREVIOUS.TOP;
        END;
END:
SYMBOL 'or':BEGIN
      LEFT=PREVIOUS.RIGHT;
      TOP=PREVIOUS.TOP;
END;
SELECT A.2:BEGIN
      LEFT=PREVIOUS.RIGHT;
      TOP=PREVIOUS.TOP;
      WIDTH=ENCLOSED.WIDTH;
      HEIGHT=ENCLOSED.HEIGHT;
      TEXT NAME:BEGIN
            LEFT=ENCLOSING.LEFT;
            TOP=ENCLOSING.TOP;
      END;
      TEXT '(':BEGIN
            LEFT=PREVIOUS.RIGHT;
            TOP=PREVIOUS.TOP;
      END;
      SELECT ARGUMENT:BEGIN
            LEFT=PREVIOUS.RIGHT;
            TOP=PREVIOUS.TOP;
            WIDTH=ENCLOSED.WIDTH;
            HEIGHT=ENCLOSED.HEIGHT;
            TEXT NAME:BEGIN
                  LEFT=ENCLOSING.LEFT;
                  TOP=ENCLOSING.TOP;
            END;
      END;
      TEXT ')':BEGIN
            LEFT=PREVIOUS.RIGHT;
```

```
                        TOP=PREVIOUS.TOP;
                    END;
                END:
                TEXT ')':BEGIN
                    LEFT=PREVIOUS.RIGHT;
                    TOP=PREVIOUS.TOP;
                END;
        END;
        TEXT ')':BEGIN
            LEFT=PREVIOUS.RIGHT;
            TOP=PREVIOUS.TOP;
        END;
END;
```

Index

Références Bibliographiques

[ACN90] **Augustsson L., Coquand T., Nordström B.**, 'A short description of Another Logical Framework' Preliminary Proceedings of the First Annual Workshop on Logical Frameworks, Antibes, May 1990.

[AHM87] **Avron A., Honsell F. A., Mason L. A.**, 'Using Typed Lambda Calculus to Implement Formal Systems on a Machine', LFCS Report Series, University of Edinburgh 1987.

[AHMP92] **Avron A., Honsell F. A., Mason L. A., Pollack R.**, 'Using typed lambda calculus to implement formal systems on a machine', Journal of Automated Reasoning **9** N°3, December 1992, pp. 309-354.

[And80] **Andrews P.B.**, 'Transforming matings into natural deduction proofs', in CADE-5, Springer-Verlag 1980, LNCS 87, pp. 281-292.

[And91] **Andrews P.B.**, 'More on the problem of finding a mapping between clause representation and natural deduction representation', Journal of Automated Reasoning **7**, 1991, pp. 285-286.

[And81] **Andrews P.B.**, 'Theorem Proving via General Matings', *JACM Vol 28 No 2*, 1981, pp. 193-214.

[Avr91] **Avron A.**, 'Simple Consequence Relations', Information And Computation 92, 1991, pp. 105-139.

[Bar92] **Barendregt H.**, 'Lambda Calculi with Types', Gabbay, Abramsky and Maibaum, editors, Handbook of Logic in Computer Science, Vol 2, Oxford University Press 1992.

[BCC88] **Boy de la Tour T., Caferra R., Chaminade G.**, 'Some Tools for an Inference Laboratory (ATINF)', CADE-9, *LNCS 310*, Springer-Verlag 1988, pp. 744-745.

[BCDIKLP87] **Borras P., Clément D., Despeyroux T., Incerpi J., Kahn G., Lang B., Pascual V.**, 'CENTAUR: The system', INRIA Sophia-Antipolis 1987.

[BCP94] **Bourely C., Caferra R., Peltier N.**, 'Building models automatically. Experiments with an extension of OTTER.', Proc. CADE-12, Springer-Verlag 1994 (a paraître).

[Bib81] **Bibel W.**, 'On Matrices with Connections', *JACM Vol 28 N°4*, 1981, pp. 633-645.

[BK92] **Boy de la Tour T., Kreitz C.**, 'Building proofs by analogy via the Curry-Howard isomorphism', Proc. of the Conference on Logic Programming and Automated Reasoning, LPAR'92, LNAI 624, Springer-Verlag 1992, pp. 202-213.

[BL84] **Bledsoe W.W., Loveland D.W.** (Eds.), 'Automated Theorem Proving after 25 years', vol. 29 of "Contemporary Mathematics", American Mathematical Society, Providence, RI, 1984.

[Ble77] **Bledsoe W.W.**, 'Non-resolution Theorem Proving', Artificial Intelligence 9, 1977, pp. 1-35.

[Boy91] **Boy de la Tour T.**, 'Optimisations par renommage dans la méthode de résolution', PhD Thesis, INPG Grenoble 1991.

[Boy92] **Boy de la Tour T.**, 'An Optimality Result for Clause Form Translation', Journal of Symbolic Computation Vol 14 1992, pp. 283-301.

[Bra92] **Bradfield J.C.**, 'A Proof Assistant for Symbolic Model-Checking', LFCS Report Series, ECS-LFCS-92-199, University of Edinburgh, March 1992.

[Bro88] **Brooks K. P.**, 'A Two-view Document Editor with User-definable Document Structure', Rapport de Recherche N°33, Digital 1988.

[Bur86] **Burstall R.M.**, 'Research in Interactive Theorem Proving at Edinburgh University', LFCS Report Series, University of Edinburgh 1986.

[Caf86] **Caferra R.**, 'Notes sur la logique, sa mécanisation, la programmation logique', Notes de cours, ENSIMAG Grenoble 1986.

[CD92] **Caferra R., Demri S.**, 'Semantic Entailment in Non Classical Logics Based on Proofs in Classical Logic', CADE-11, *LNAI 607*, Springer-Verlag 1992, pp. 385-399.

[CDH91] **Caferra R., Demri S., Herment M.**, 'Logic morphisms as a framework for backward transfer of lemmas and strategies in some modal and epistemic logics', Proc. AAAI-9, MIT Press 1991, pp. 421-426.

[CDH93] **Caferra R., Demri S., Herment M.**, 'A framework for the transfer of proofs, lemmas and strategies from classical to non-classical logics', Studia Logica, Vol. 52 n°2, 1993, pp. 197-232.

[CKB85] **Constable R.L., Knoblock T. B., Bates L. J.**, 'Writing Programs that Construct Proofs', *Journal of Automated Reasoning Vol 1*, 1985, pp. 285-326.

[CH85] **Coquand T., Huet G.**, 'Constructions: A Higher Order Proof System for Mechanizing Mathematics.', EUROCAL85, *LNCS 203,* Sringer-Verlag 1985.

[CH85] **Coquand T., Huet G.**, 'Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions', Rapport de Recherche N°463, INRIA Sophia-Antipolis 1985.

[CH86] **Coquand T., Huet G.**, 'The Calculus of constructions', Rapport de Recherche N°530, INRIA Sophia-Antipolis 1986.

[CH88] **Coquand T., Huet G.**, 'The Calculus of constructions', *Information and Computation 76*, 1988, pp. 95-120.

[CH90] **Cousineau G., Huet G.**, 'The CAML primer', Rapport technique n°122, INRIA 1990.

[CH93] **Caferra R., Herment M.**, '$GLEF_{ATINF}$ A graphic framework for combining provers and editing proofs for different logics', Proc. DISCO'93, LNCS 722, Springer-Verlag 1993, pp. 229-240.

[CH9?] **Caferra R., Herment M.**, 'A Generic graphic Framework for Combining Inference Tools and editing Proofs and formulas', Submitted.

[Che76] **Chester D.**, 'The Translation of Formal Proofs into English', *Artificial Intelligence 7*, 1976.

[CHP86] **Cousineau G., Huet G., Paulson L.**, 'The ML Handbook', INRIA 1986.

[CHZ91] **Caferra R., Herment M., Zabel N.**, 'User-oriented theorem proving with the ATINF graphic proof editor', Proc. FAIR'91, LNAI 535, Springer-Verlag 1991, pp. 2-10.

[CZ92] **Clarke E., Zhao X.**, 'Analytica-An experiment in combinig theorem proving and symbolic computation', RR CMU-CS-92-117, September 1992.

[C+86] **Constable R.L.** et al.: *Implementing Mathematics with the Nuprl development system*, Prentice-Hall 1986.

[dBr72] **de Bruijn N.G.**, 'Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem', 1972.

[dBr80] **de Bruijn N.G.**, 'A Survey of the Project AUTOMATH.', in To H.B. Curry:: Essays on Combinatory logic, Lambda Calculus and Formalism (J.P. Seldin and J.R. Hindley, Eds.), Academic Press 1980, pp. 579-606.

[Dav83] **Davis M.**, 'The Prehistory and Early History of Automated Deduction', in Automation of Reasoning 1 (J. Siekmann and G. Wrighston Eds.), Springer-Verlag 1983, pp. 1-28.

[Del92] **Delsart B.**, 'A new approach to E-unification based on conditional term rewriting' Springer-Verlag, *LNCS 656*, pp. 468-483.

[Des88a] **Despeyroux J.**, 'THEO: An Interactive Proof Development System', Rapport de Recherche N°887, INRIA Sophia-Antipolis 1988.

[Des88b] **Despeyroux J.**, 'THEO: An Interactive Proof Development System', Manuel utilisateur, INRIA Sophia-Antipolis 1988.

[DS79] **Davis M., Schwartz T.**, 'Metamathematical Extensibility for Theorem Verifiers and Proof-Checkers', *Comp. & Maths. with Appls.*, Vol 5, 1979, pp. 217-230.

[Dyb82] **Dybjer P.**, 'Mathematical proof in Natural Language', Report 5, Programming Methodology Group, ASSN 0282-2083, University of Goteborg 1982.

[D+91] **Dowek G.** et al., 'The Coq Proof Assitant User's Guide, version 5.6', Rapport Technique INRIA N° 134, INRIA Le Chesnay 1991.

[Ebb85] **Ebbinghaus H.D.**, 'Extended Logics: the General Framework', in Model Theoretic Logics, J. Barwise and S. Feferman eds, Springer-Verlag 1985, pp. 25-76.

[FHV92] **Fagin R., Halpern J.Y., Vardi M.I.**, 'What is an inference rule', Journal of Symbolic Logic, Vol. 57, n° 3, 1992.

[FM87] **Felty A., Miller D.**, 'Specifying Theorem Provers in a Higher-Order Logic Programming Language', PA 19104-6389, University of Pennsylvania 1987.

[Gab91] **Gabbay D.**, 'Labelled Deductive Systems, Part 1', Centrum für Informations und Sprachverarbeitung, Universität München 1991.

[GGJLZ88] **Gilbert P., Graver J., Johnson R., Loyall J., Zurawski L.**, 'Typed Smalltalk Working Papers', Rapport de Recherche N°1457, University of Illinois 1988.

[GMW79] **Gordon M., Milner R., Wadsworth C.**, 'Edinburgh LCF: A Mechanized Logic of Computation', *LNCS 78*, Springer-Verlag 1979.

[Gol81] **Goldfrab W.D.**, 'Note the Undecidability of the Second-Order Unification Problem', *Theoretical Computer Science 13*, 1981, pp. 225-230.

[Gor87] **Gordon M.**, 'HOL: A Proof Generating System for Higher Order Logic', dans VLSI Specification, Verification and Synthesis (G.Birtwistle and P.A. Subrahmanyam eds.), Kluwer 1987.

[Gri87] **Griffin T. G.**, 'An Environment for Formal Systems', LFCS Report Series, University of Edinburgh 1987.

[Has88] **Hascoët L.**, 'A tactic-driven system for building proofs', INRIA Sophia-Antipolis 1988.

[Has89] **Hascoët L.**, 'Theory meets Efficiency: a new Implementation for Proof Trees', Article N°1109, INRIA Sophia-Antipolis 1989.

[HHP87] **Harper R., Honsell F. A., Plotkin G.**, 'A Framework for Defining Logics', LFCS Report Series, University of Edinburgh 1987.

[HHP89] **Harper R., Honsell F. A., Plotkin G.**, 'A Framework for Defining Logics', CMU-CS-89-173, Carnegie Mellon University 1989.

[HHP93] **Harper R., Honsell F. A., Plotkin G.**, 'A Framework for Defining Logics', *JACM Vol 40 N°1*, January 1993, pp.143-184.

[Hir86] **Hirose K.**, 'An approach to Proof Checker', Department of Mathematics, Waseda University 1986.

[HO94] **Herment M., Orlowska E.**, 'Handling Information Logics In A Graphical Proof Editor', Computational Intelligence, 1994, *à paraître.*

[Hor93] **Horgan J.**, 'L'ordinateur en Mathématiques', Pour la Science, N° 194, décembre 1993, pp. 84-93.

[HST89] **Harper R., Sannella S., Tarlecki A.**, 'Structure and Representation in LF', University of Edinburgh 1989.

[Hua90] **Huang X.**, 'Reference Choices in Mathematical Proofs', Proc. ECAI'90, Pitman 1990, pp. 720-725.

[Hue75] **Huet G.**, 'A Unification Algorithm for Typed λ-calculus', *Theoretical Computer Science 1*, pp. 25-57, IRIA Rocquencourt 1975.

[Hue91] **Huet G.**, 'Initiation au λ-calcul', Notes de cours, Université de Paris VII 1991.

[JR92] **Jacobs I., Rideau-Gallot L.**, 'A CENTAUR tutorial', Rapport Technique n° 140, INRIA Sophia-Antipolis 1992.

[Ket84] **Ketonen J.**, 'EKL-A mathematically Oriented Proof Checker', CADE-7, *LNCS 170*, Springer-Verlag 1984, pp. 65-79.

[KW83] **Ketonen J., Weening J.**, 'The langage of an Interactive Proof Checker', Standford University, CS Report STAN-CS-83-992, 1983.

[Lam90] **Lam C.W.H.**, 'How Reliable is a Computer-Based Proof?', The Mathematical Intelligencer, Vol. 12, N° 1, 1990, pp. 8-12.

[Lam91] **Lam C.W.H.**, 'The Search for a Finite Projective Plane of order 10', The American Mathematical Society Monthly, Vol. 98, number 4, April 1991, pp. 305-318.

[Lam93a] **Lambek J.**, 'What is a deductive system', Algebraic and Categorical Methods in Computer Science, TEMPUS Summer School, Brno 1993.

[Lam93b] **Lamport L.**, 'How to Write a Proof', Research Report n°94, DEC SRC 1993.

[LB92] **Luo Z., Burstall R.**, 'A Set-theoretic Setting for Structuring Theories in Proof Development', LFCS Report Series, University of Edinburgh 1992.

[Lin88] **Lingenfelder C.**, 'Structuring Computer Generated Proofs', SEKI-Report SR-88-19, 1988.

[Lin90] **Lingenfelder C.**, 'Transformation and structuring of computer generated proofs', SEKI Report SR-90-26, University of Kaiserslautern 1990.

[McC62] **McCarthy J.**, 'Computer programs for checking mathematical proofs', *Proceedings of the Symposia* in *Pure Mathematics*, vol. V. Recursive Function Theory, American Mathematics Society, Providence, R.I., 1962, pp. 219-228.

[McC90] **McCune W.**, 'Otter 2.0', CADE10, LNCS 449, Springer-Verlag 1990, pp. 663-664.

[McC90] **McCune W.**, 'Otter 2.0 Users Guide', Technical Report ANL-90/9, Argonne National Laboratory 1990.

[Men64] **Mendelson E.**, 'Introduction to Mathematical Logic', D. Van Nostrand Co. 1964.

[Mes87] **Meseguer J.**, 'General Logics', Logic Colloquium '87, H-D Ebbinghaus (Ed), North-Holland 1987, pp. 275-330.

[Mil84] **Miller D.A.**, 'Expansion tree proofs and their conversion to natural deduction', Proc. CADE-7, LNCS 170, Springer-Verlag 1984, pp. 375-393.

[Mil85] **Milner R.**, 'The use of machines to assist in rigorous proof', dans 'Mathematical Logic and Programming Languages' (C.A.R Hoare and J.C. Shepherdson, eds.), Prentice Hall 1985, pp. 77-88.

[MLRC88] **Madany P.W., Leyens D.E., Russo V.F., Campbell R.H.**, 'A C++ Class Hierarchy for Building UNIX-Like File Systems', Rapport de Recherche N°1462, University of Illinois 1988.

[Pau85] **Paulson L.C.**, 'Lessons learned from LCF: A Survey of Natural Deduction Proofs', The Computer Journal, vol. 28, no. 5, 1985, pp. 474-479.

[Pau88] **Paulson L.C.**, 'Experience with Isabelle: A Generic Theorem Prover', Proc. COLOG '88 Tallin, URSS, University of Cambridge 1988.

[Pau89] **Paulson L.C.**, 'The foundation of a generic theorem prover', Journal of Automated Reasoning **5**, 1989, pp. 363-397.

[PG86] **Plaisted D. A., Greenbaum S.**, 'A structure-preserving clause form translation', Journal of Symbolic Computation N°2, 1986, pp. 293-304.

[Qui87] **Quint V.**, 'Une approche de l'Edition Structurée des documents', Thèse d'Etat, Université Sc. Tech. et Méd. de Grenoble 1987.

[RA83] **Reps T. W., Alpern B.**, 'Interactive Proof Checking', Cornell University, *POPL11*, N.Y. 1983, pp. 36-45.

[RT85] **Reps T. W., Teitelbaum T.**, 'The Synthesizer Generator Reference Manual, Dept. of Computer Science, Cornell University, Ithaca, N.Y. 1985.

[Sco74] **Scott D.**, 'Completeness and Axiomatizability in Many-valued Logic', Proceedings of the Tarski Symposium, pp. 411-435, Proceedings of Symposia in Pure Mathematics, Vol XXV, American Mathematical Society, Providence, Rhode Island 1974.

[SB92] **Sarkar M., Brown M.H.**, 'Graphical Fisheye Views of Graphs', Research Report n°84, DEC SRC 1992.

[Sim88] **Simon D.**, 'Checking Natural Language Proofs', CADE-9, *LNCS 310*, Springer-Verlag 1988, pp. 141-150, .

[Sup84] **Suppes P.**, 'The next generation of Interactive Theorem Provers', CADE-7, *LNCS 170*, Springer-Verlag 1984, pp. 303-315.

[TBK92] **Théry L., Bertot Y., Kahn G.** 'Real Theorem Proverss Deserve Real User-Interfaces', Rapport de Recherche N°1684, INRIA Sophia-Antipolis 1992.

[Ver89] **Vercoustre A. M.**, 'Edition structurée - Approche hypertexte - Coopération et complémentarité', Rapport de Recherche N°1052, INRIA Sophia-Antipolis 1989.

[Wos88] **Wos L.**, *Automated Reasoning: 33 Basic Research Problems*, Prentice Hall 1988.

Résumé

Après un historique bref et général de la déduction automatique, on analyse les tendances actuelles et les besoins en présentation de preuves et communication d'outils d'inférence. Les notions théoriques concernées sont présentées et étudiées en détail. On donne ensuite une synthèse comparative critique et exhaustive de l'état de l'art. Cette synthèse manquait dans la littérature.

L'analyse des notions fondamentales en logique et la synthèse sur l'état de l'art permettent d'établir les caractéristiques retenues pour le système **GLEF** (**G**raphical & **L**ogical **E**dition **F**ramework).

La conception et la réalisation de deux langages ont permis de rendre GLEF *générique* (c'est à dire paramétrable par le système formel employé et par la présentation de ses preuves). Un formalisme de définition, fondé sur le Calcul des Construction (dû à Coquand et Huet), sert à représenter et à vérifier les systèmes formels et les preuves dans ses systèmes formels. Un langage de présentation, fondé sur la notion de "boîte", sert à décrire leur présentation.

En annexe nous donnons un algorithme original pour l'opération d'effacement, particulièrement difficile en λ-calcul typé, qui sert à réaliser la commande "couper" de GLEF.

GLEF a été développé au sein du projet ATINF (ATelier d'INFérence). Un manuel utilisateur rudimentaire et de nombreux exemples d'utilisation en sont donnés. Certains exemples montrent comment, après avoir spécifié la définition et la présentation d'un système formel objet, un utilisateur de GLEF peut construire ou visualiser des preuves en manipulant directement les objets (formules, preuves partielles, etc.) à l'écran, avec la souris. D'autres illustrent comment GLEF présente les preuves produites par les démonstrateurs d'ATINF ou extérieurs à ATINF.

Les principales lignes de recherche future concluent ce travail.

Mots Clés

Démonstration automatique et interactive. Cadres logiques. ("logical frameworks"). Présentation graphique de preuves. Édition structurée de preuves. Coopération d'outils d'inférence.