Introduction

« Le bon coin pour le Snark ! » cria l'homme à la cloche,
Tandis qu'avec soin il débarquait l'équipage,
En maintenant, sur le vif de l'onde, ses hommes,
Chacun par les cheveux suspendu à un doigt.

Si le problème de la définition d'un dessin (2D) ou d'un objet (3D) à partir d'une cotation (cohérente) de celui-ci est un sujet très étudié actuellement dans le monde de la Conception Assistée par Ordinateur, son équivalent en Enseignement Assisté par Ordinateur, l'automatisation de la résolution d'exercices de construction géométrique fait l'objet de beaucoup moins de recherches.

1. Des problèmes de construction

Le problème général des constructions, géométriques ou non, peut s'exprimer par :

Dans un certain univers U*, étant donnés* n *objets* $d_1, d_2 \ldots, d_n$ *et une formule logique* F *(n+m)-aire s'interprétant dans* U*, trouver* m *objets,* $r_1, r_2 \ldots, r_m$ *de* U *tels que l'interprétation de* $F(d_1, d_2, \ldots d_n, r_1, r_2, \ldots, r_m)$ *soit vraie dans* U.

Cet énoncé général, assez vague finalement, doit être précisé en répondant, par exemple aux questions suivantes :
- de quel univers et de quels types d'objets s'agit-il ?
- doit-on trouver une solution, quelconque ou privilégiée, au problème ou s'agit-il de les trouver toutes ?
- faut-il expliquer, le cas échéant, pourquoi un problème particulier n'a pas de solution ?
- s'agit-il de donner des valeurs aux objets $r_1, r_2, \ldots, r_m$ à partir de valeurs données à $d_1, d_2 \ldots d_n$ (résolution numérique) ou d'expliciter un procédé (aussi général que possible ?) permettant de trouver ces valeurs à partir de valeurs attribuées à $d_1, d_2 \ldots, d_n$ (résolution formelle) ?
- que signifie précisément le terme "trouver" dans l'énoncé ci-dessus : faut-il trouver les r_i exactement ou approximativement, une marge d'erreur étant fixée ? Tous les procédés sont-ils permis ou seuls quelques "instruments" de base sont-ils autorisés ?

Par exemple, dans le domaine de la CAO, plusieurs univers peuvent être considérés, comme la représentation par les bords ou la géométrie constructive du solide [Requicha 80]. Il s'agit alors souvent de trouver des valeurs numériques, aussi précises que possible, pour les objets r_i à partir de valeurs données aux d_i. On ne cherche alors qu'une seule solution qui doit être la plus vraisemblable possible. A cet égard, on utilise souvent une esquisse, c'est-à-dire, une figure numérique approximative fournie par l'utilisateur avant la résolution et guidant celle-ci à l'aide de certains critères, comme la topologie ou le positionnement. Les procédés de

recherche de solution ne sous-entendent, en général, aucune restriction en ce qui concerne les outils autorisés : il s'agit finalement, à l'aide de diverses méthodes, purement géométriques ou non, de résoudre un système d'équations, en principe polynomiales.

Les manières d'appréhender un problème de construction géométrique peuvent se ranger principalement dans deux catégories. D'une part, dans les approches algébriques, on choisit un repère, aussi simplificateur que possible, on traduit l'énoncé du problème en équations, généralement polynomiales, et l'on résout le système trouvé. D'autre part, dans les approches purement géométriques, ces équations restent sous-jacentes et leur résolution se fait, en quelque sorte, "sur la figure". Il faut noter que ces deux approches ne présument en rien de la méthode de résolution qui peut être, dans les deux cas, soit numérique, soit formelle. Citons la méthode de Lebesgue [Lebesgue 50] comme exemple d'approche algébrique formelle, et le travail de G. Sunde [Sunde 87] et de A.Verroust et F. Schonek comme exemples d'approches géométriques numériques [Schonek & Verroust 88]. Parmi les méthodes géométriques formelles, on peut citer [Scandura & al 74], [Aldefeld 88], [Koch & al 89] et [Barz & Holland 89]. Par ailleurs, l'important travail de M. Buthion, que nous avons pris comme point de départ, semble toujours la méthode géométrique formelle la plus générale ayant effectivement donné des résultats [Buthion 79].

Le problème du choix des instruments de construction n'est en général pas discuté dans les méthodes numériques. Dans le cadre des méthodes formelles, on précise, d'habitude, quels instruments sont autorisés. Dans l'enseignement de la géométrie, on considère souvent les constructions à la règle et au compas, mais d'autres modes de construction comme celles utilisant la règle seule, le compas seul, la règle et le rapporteur de distance, la règle et le traceur de parallèles, sont également étudiés. On pourra trouver dans [Lebesgue 50] et [Carrega 89] une discussion intéressante à ce sujet. Précisons encore que le choix de l'approche, algébrique ou géométrique, ne préjuge pas du choix des instruments et que les constructions à la règle et au compas sont à l'origine de travaux célèbres en algèbre (Cf op. cités).

2. Progé

Notre travail de recherche a pour objet la résolution automatique des problèmes de construction à la règle et au compas en géométrie élémentaire plane. Notre objectif est de concevoir et de réaliser un logiciel qui, à partir de l'énoncé déclaratif d'un exercice type lycée-collège, engendre un programme procédural de construction dans le cadre d'une approche purement géométrique. Notre prototype, appelé Progé[1], préfigure ce que peut être un tel logiciel. Il fonctionne en deux phases. Dans la première, il résout formellement un problème de construction à la règle et au compas à l'aide d'une figure, que nous appelons formelle. Dans la deuxième, il permet d'interpréter numériquement le programme de construction obtenu en une figure, que nous appelons numérique, ainsi que de tracer le ou les dessins correspondants, que nous appelons figures graphiques.

Progé est conçu comme un système à base de connaissances gérant une figure formelle et un historique des déductions. Il s'appuie en partie sur les idées développées par M.Buthion dans

---

1C'est-à-dire, Petit Résolveur de cOnstructions en Géométrie Elementaire.

le logiciel Geom3 [Buthion 79], notamment celles d'historique des déductions, de figure formelle et d'unification modulo. Mais notre travail s'en distingue par :

- la clarification de certains concepts ;
- l'ajout des notions d'énoncé et de programme de construction ;
- une méthode d'unification modulo plus précise ;
- l'interprétation numérique d'une solution formelle ;
- la gestion des solutions multiples ;
- la prise en compte des différents cas de figure dégénérés ou non ;
- le tracé des figures correspondantes ;
- et surtout, la possibilité donnée à l'utilisateur expert de définir l'univers géométrique.

Considérer l'univers géométrique de base de Progé comme une donnée du système résulte d'un point de vue pragmatique : ne sachant pas trop comment faire évoluer nos recherches, nous avons essayé de concevoir un système aussi ouvert et évolutif que possible. Nous avons, par conséquent, opté pour une architecture en forme de système à base de règles avec un moteur d'inférence d'ordre au moins 1 ! Cependant, les premières maquettes que nous avons réalisées présentaient trop de cas particuliers à traiter : comment permuter les arguments des termes employés pour avoir d'autres termes équivalents, traitements particuliers pour les sortes géométriques utilisées, traitements particuliers en ce qui concerne les rapports entre les relations d'incidence et certains symboles fonctionnels, propagation des contraintes ... Nous avons donc essayé de poser un cadre sur ces notions en modélisant le concept d'univers géométrique.

Dans notre approche, l'univers géométrique est un système formel au sens de la logique. Nous donnons ainsi un cadre axiomatique à la géométrie élémentaire : ceci n'est plus vraiment neuf depuis Euclide ! Cependant, au contraire de systèmes axiomatiques comme celui de Hilbert [Hilbert 71], notre approche est essentiellement pratique : les axiomes que nous avons retenus n'ont qu'un rapport lointain avec ceux d'Hilbert. Au reste, nous ne cherchons pas à avoir un système minimal, et bien que cela soit désirable, nous n'avons pas un système adéquat et complet (en partie parce que nous nous sommes limités à la logique du premier ordre).

Ce système formel est décrit au moyen d'une spécification algébrique. Tandis que la structure de données implantant un univers géométrique est décrite par une méta-spécification détaillant la mise en oeuvre de la spécification de bas niveau. Nous y avons gagné une implantation relativement extensible et un moyen de description rigoureux des mécanismes mis en oeuvre dans notre système.

3. Plan de ce rapport

Le chapitre 1 présente le problème des constructions géométriques à la règle et au compas. Après avoir défini les termes de *constructibilité* et de *construction à la règle et au compas*, nous exposons une palette d'exemples pour illustrer la richesse du domaine et la complexité qui se cache sous une formulation très simple.

Le chapitre 2 expose une approche algébrique des constructions à la règle et au compas. Cette approche formelle résulte d'une étude et d'une mise en application de l'algorithme de H.

Lebesgue [Lebesgue 50], faite lors d'un stage de DEA par G. Chen [Chen 92]. Nous y présentons l'algorithme de décomposition de Ritt-Wu [Chou 88], utilisé par G. Chen.

Au chapitre 3, nous faisons une présentation informelle de Progé. Nous y exposons les différentes fonctionnalités offertes tant à l'utilisateur qu'à l'expert. Nous y introduisons les concepts centraux de Progé comme ceux d'univers géométrique, de programme de construction, de figure et de raisonnement formels. Ceux-ci seront repris, en détail, dans les chapitres suivants.

Le long chapitre 4 explique la notion d'*univers géométrique*. Nous y détaillons les axiomes qui nous paraissent utiles dans le cadre des constructions à la règle et au compas et nous montrons comment ces axiomes se retrouvent dans notre spécification de la sorte univers géométrique. Nous y donnons également les sortes et opérations de base permettant d'utiliser un tel univers géométrique. La sorte substitution et l'opération d'unification classique sont spécifiés algébriquement. Quelques exemples simples de mise en oeuvre sont donnés.

Au chapitre 5, nous présentons les entrées/sorties de Progé du point de vue de l'utilisateur. Nous détaillons les entrées/sorties de Progé. Nous décrivons le *langage d'énoncé* utilisé. Nous précisons le rôle de l'univers géométrique dans l'interprétation d'un énoncé géométrique. Nous abordons ensuite la notion de *programme de construction géométrique*. Nous décrivons les structures de contrôle retenues pour modéliser certains phénomènes géométriques comme la multiplicité des solutions ou les cas de dégénérescence. Nous approfondissons le problème de l'interprétation d'un programme de construction, les liens avec l'univers géométrique.

Le chapitre 6 expose la notion de *figure formelle*. Une figure formelle mémorise tous les "éléments constructifs" découverts par Progé. Les *objets géométriques* utilisés par la construction et leur *état de construction* y sont consignés ainsi que les *relations d'égalité* et d'*incidence* entre objets géométriques. Diverses fonctions de recherche dans une figure sont détaillées pour finir par l'opération cruciale d'*unification modulo une figure*. La description de la traduction d'un énoncé géométrique en une figure initiale vient conclure ce chapitre.

Le chapitre 7 détaille le fonctionnement du moteur d'inférence employé par Progé. A cette occasion, nous exposons la syntaxe des *règles simples*. Nous donnons une spécification algébrique de la sorte *raisonnement formel* et nous montrons comment Progé utilise tous ces éléments, notamment au moyen de l'unification modulo la figure décrite au chapitre précédent, pour faire progresser la résolution d'un problème de construction.

Le chapitre 8 fait le point sur les phénomènes d'*exceptions* et de *cas de figures* souvent rencontrés dans les problèmes de construction géométrique. Nous exposons les mécanismes de traitement des cas dégénérés et des règles disjonctives. Nous détaillons l'incidence de ces traitements sur le programme de construction géométrique élaboré par Progé.

Le chapitre 9 expose rapidement le passage de la spécification algébrique décrite dans les chapitres précédents, à une *implantation en Prolog* de Delphia. Nous explicitons nos choix sur l'implantation de chaque structure de données et nous montrons comment l'exploitation des mécanismes de Prolog comme l'unification et le retour arrière, permettent de simplifier cette traduction.

L'annexe 1 détaille quelques exemples de résolution de problèmes de construction géométriques par Progé. L'annexe 2 présente les énoncés d'exercices résolus actuellement par notre programme.

4. La chasse au Snark

Les citations introduisant chaque chapitre sont extraites du poème La Chasse au Snark de Lewis Carroll et plus particulièrement de la Huitième Crise [Carroll 89].

Chapitre 1

Constructions à la règle et au compas

« Le bon coin pour le Snark ! Je vous l'ai dit deux fois :
Cela devrait suffire à vous encourager.
Le bon coin pour le Snark ! Je vous l'ai dit trois fois :
Ce que je dis trois fois est absolument vrai. »

Les problèmes de construction géométrique sont en premier lieu des problèmes pratiques pour lesquels il faut déterminer certains points et/ou tracer certaines figures. Il se pose donc naturellement la question des intruments utilisables pour effectuer ces tracés. Ces instruments sont différents suivant les circonstances. Ainsi les jardiniers se servent essentiellement de cordeaux (pour tracer des droites, des cercles, mais aussi des ellipses) et les géomètres utilisent des instruments de visée tels que le théodolite. Lorsqu'on peut travailler sur une surface très plane et suffisamment petite comme en tôlerie ou en cartographie, on utilise habituellement la règle et le compas, mais on se sert aussi de divers gabarits et pistolets propres à l'activité considérée.

Ce sont les constructions à la règle et au compas qui sont habituellement considérées en géométrie élémentaire et surtout dans l'enseignement de celle-ci. Les constructions à la règle et au compas ont une longue et riche histoire que nous ne développerons pas ici, mais dont le lecteur pourra avoir un aperçu dans [Colette 73], [Carrega 89], [Lebesgue 50] et [Lehmann & Bkouche 88]. C'est essentiellement ce genre de constructions que nous considérons dans ce chapitre mais nous donnons un aperçu d'autres moyens de construction également utilisés en géométrie élémentaire.

1. Présentation du problème du point de vue géométrique

Historiquement, les géométres se sont d'abord intéressés à la résolution effective de problèmes de construction, à la règle et au compas ou avec d'autres instruments. L'impossibilité de résoudre à la règle et au compas des problèmes célèbres tels que la quadrature du cercle, la duplication du cube ou la trisection de l'angle a conduit les mathématiciens à s'interroger sur la caractérisation de la classe des problèmes constructibles à la règle et au compas. Cette notion de constructibilité formalise et cerne bien le problème, c'est pourquoi nous la présentons en premier.

1.1. Définition de la constructibilité

Dans [Lebesgue 50], on trouve cette définition :

Des points étant donnés, on se propose de construire un point qu'ils déterminent en n'usant que de la règle et du compas, la règle ne devant servir qu'à joindre deux points donnés ou précédemment obtenus, le compas ne devant servir qu'à tracer une circonférence dont le

centre est un point donné ou déjà obtenu et dont le rayon est la distance de deux points donnés ou déjà obtenus.

Bien que très claire, nous la précisons par les définitions plus formelles, mais aussi plus lourdes, suivantes :

Définition (Constructibilité d'un point à partir d'un ensemble de points [Carrega 89]). Soit P un plan euclidien et B un sous ensemble fini de P ayant au moins deux éléments.
Un *point* M de P est dit *constructible* à la règle et au compas à partir de B s'il existe une suite finie de points de P se terminant par $M : M_1, M_2, ... M_n = M$ telle que

pour tout i, $1 \leq i \leq n$, M_i est un point d'intersection soit de deux droites, soit d'une droite et d'un cercle, soit de deux cercles, obtenus à l'aide de l'ensemble $E_i = B \cup \{M_1, ..., M_{i-1}\}$ de la manière suivante :

- chaque droite passe par deux points distincts de E_i,
- chaque cercle est centré en un point de E_i et a pour rayon la distance entre deux points de E_i.

Une *droite* passant par deux points constructibles à la règle et au compas est dite *constructible* à la règle et au compas.
Un *cercle* centré en un point constructible à la règle et au compas et ayant pour rayon la distance entre deux points constructibles à la règle et au compas est dit *constructible* à la règle et au compas.
Les points de B sont appelés *points de base*.
□

Dans la suite, pour éviter les lourdeurs, le mot constructible signifiera constructible à la règle et au compas.

Définition (réel constructible). Un *réel* α est dit *constructible* à partir d'un ensemble de base B si et seulement si il existe un point M dont une des coordonnées est α dans un repère euclidien construit à l'aide de points de B.

Traditionnellement, en algèbre, on considère les réels constructibles à partir des points de base O et I de coordonnées respectives (0, 0) et (1, 0), et on les appelle simplement les *réels constructibles*. Un des grands problèmes de l'algèbre a été de caractériser ces nombres constructibles (Cf. [Carrega 89] par exemple). Mais très vite, on s'aperçoit que ces définitions ne constituent pas un cadre théorique suffisant pour les constructions géométriques.

Exemple. L'énoncé:

Enoncé 0. *Construire une droite* Δ *perpendiculaire à une droite* D *donnée et passant par un point* A *donné.*

montre qu'il faudrait pouvoir prendre en compte la donnée d'une droite et pas seulement de points.

C'est pourquoi on complète cette définition de la manière suivante [Carrega 89] :

Définition (Points remarquables d'une figure, constructibilité à partir d'une figure). Soit, dans le plan euclidien, une figure F constituée par des points isolés, des droites et des cercles en nombre fini. On appelle *ensemble des points remarquables* de F l'ensemble B_1 constitué des points isolés, des points d'intersections entre les droites et les cercles de F et des centres des cercles de F. Si F n'est pas *entièrement déterminée* par B_1, i.e. s'il existe une droite de F contenant moins de deux points de B_1 ou un cercle de F ne contenant pas de points de B_1, on complète B_1 pour obtenir un ensemble B_2 en ajoutant des points arbitraires pris sur les droites et cercles de F afin que F soit déterminée par B_2.

Les éléments *constructibles à partir de F* sont ceux constructibles à partir de B_2 et qui ne dépendent pas effectivement des points arbitraires ajouté à B_1 pour obtenir B_2.

□

Exemple. On peut résoudre l'exercice correspondant à l'énoncé 0 de la manière suivante. On choisit arbitrairement deux points B et C sur la droite D, distincts entre eux et distincts de A, la figure est ainsi entièrement déterminée par les points A, B et C (Cf. figure 1). On trace le cercle Γ de centre A et passant par B, si ce cercle est tangent à D alors la droite AB convient, sinon, ce cercle recoupe D en un point M. Les deux cercles respectivement de centre B et passant par M et de centre M passant par B se coupent en deux points E et F, la droite EF répond à la question : elle est perpendiculaire à D puisque c'est la médiatrice de [B, M] et elle passe par A par construction du point M. Corollaire : puisqu'il n'y a qu'une droite dans ce cas, celle-ci ne dépend pas des points B et C utilisés pour la construction. Faisons une remarque d'ordre pratique : on a intérêt à faire la construction avec le point, parmi B ou C, le plus éloigné de A, car il donne la meilleure précision.

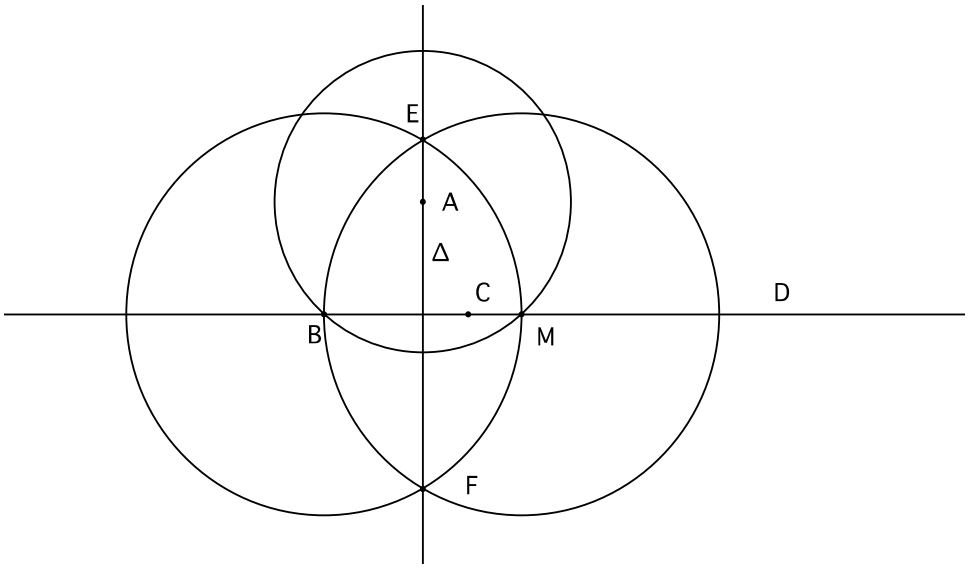

Figure 1 : Construction d'une perpendiculaire

Remarquons que cette définition de la constructibilité, relativement imprécise, se prête plus difficilement à la formalisation que la définition 1. Dans le cadre d'une formalisation de telles

constructions, il faudrait répondre nettement aux questions : que signifient "choisir des points arbitraires" et "dépendre effectivement de" ?

D'autres instruments de dessin ont été envisagés, soit, pour résoudre des problèmes réputés insolubles à la règle et au compas, soit pour montrer qu'on pouvait avoir des instruments imparfaits comme un compas rouillé, i.e. à ouverture fixe, ou un compas "glissant", i.e. incapable de reporter des distances (Cf. [Lebesgue 50], [Carrega 89] et [Toussaint 90] sur le cinquième axiome d'Euclide).

1.2. Définition de "construire"

1.2.1. Construire.

Le but des définitions précédentes n'est certes pas d'expliciter des procédés de construction. On peut cependant en tirer un mode opératoire systématique du style "engendrer et tester" pour décider si un point est constructible à la règle et au compas à partir de points donnés : soit B_0 l'ensemble des points de base. On trace tous les cercles et toutes les droites possibles à partir de B_0 en suivant les règles de la définition des points constructibles. Soit B_1 l'ensemble de tous les points de B_0 auxquels on ajoute les points obtenus par intersection des droites et cercles précédents, on définit ainsi, par récurrence, une suite d'ensembles de points (B_n). Il suffit alors de calculer les ensembles de cette suite jusqu'au rang n_o tel que le point demandé appartienne à l'ensemble B_{n^o} ... si cela se produit un jour et si on sait le détecter. Nous revenons sur cette question au chapitre 2.

Outre son inefficacité, cette méthode de "construction au hasard" ne correspond pas à l'idée que l'on se fait d'une construction géométrique. Dans [Lebesgue 50], le sens du verbe construire est défini comme suit :

> *Supposer que le point inconnu est constructible avec la règle et le compas, c'est pouvoir l'obtenir grâce à un tracé précisé dans un programme tel que celui-ci : tracer d'abord la droite joignant les points donnés A et B ; puis la circonférence de centre donné C ayant pour rayon la distance des points donnés D e E [...] ; et cela,* quelles que soient les valeurs données aux variables indépendantes dont dépendent les données, *au moins pour un certain champ de variation de ces variables.*

Résoudre un problème de construction géométrique c'est donc fabriquer un programme, que nous qualifions de *programme de construction géométrique* (en abrégé `pcg`), qui peut être interprété en attribuant des valeurs numériques aux objets figurant comme données dans l'énoncé et qui fournit alors une ou des solutions particulières à cette instance de problème. Pour formaliser cette notion, il faut répondre aux questions suivantes :
- quelle forme donner à un langage de construction ? Quelles structures faut-il mettre en place dans ce langage ?
- est-ce que les solutions numériques trouvées en interprétant un programme de construction géométrique sont correctes ? Les a-t-on toutes ?

- quel est le domaine de validité d'une construction ? Que se passe-t-il en dehors de ce domaine de validité ?
Nous montrons par la suite comment nous avons résolu ces questions dans Progé.

1.2.2. Méthode des lieux

Une autre remarque concerne la manière de progresser dans la résolution d'un problème de construction : la détermination d'un objet géométrique se fait progressivement. Par exemple pour la détermination d'un point, on peut connaître d'abord une droite à laquelle il doit appartenir, puis une autre droite. Le point est alors déterminé graphiquement (on "voit" l'intersection) ou algébriquement par résolution d'un système linéaire portant sur les coordonnées du point dans un repère fixé à l'avance. Cet état de fait est à l'origine de la méthode appelée *méthode des lieux* qui consiste, lorsqu'on a un point à déterminer, à transformer les contraintes portant sur ce point en lieu géométrique (c'est-à-dire ensemble de points vérifiant certaines conditions et facilement traçable). Dans notre cas, il s'agit essentiellement de droites et de cercles, mais on peut aussi considérer des coniques, car on sait construire l'intersection d'une droite et d'une conique à la règle et au compas.

Cette progressivité soulève quelques problèmes lorsqu'on veut automatiser la méthode des lieux. Le plus important est celui de la redéfinition d'un objet géométrique : si un système ne sait pas détecter l'*égalité* entre deux définitions d'objets, il peut faire l'erreur de définir un point comme étant l'intersection d'une droite avec elle-même : s'il n'est pas faux de dire qu'un tel point appartient à cette intersection, il est faux de croire que ce point est déterminé ainsi et le reste de la construction est d'habitude une suite d'absurdités. La détection et la gestion de l'égalité entre des objets géométriques dans une configuration donnée est un problème important et très difficile à résoudre. La difficulté de la détection formelle des cas d'égalité est du même ordre que le problème initial (qui est de construire une figure). En outre, on peut avoir des objets généralement distincts et qui dans un cas de figure particulier sont confondus. Si le géomètre "voit" sur la figure des objets distincts, un système automatique de construction devra vérifier le mieux possible que des objets cruciaux sont distincts.

1.2.3. Réflexions sur le degré de liberté

Un second problème, lié au précédent, est de savoir quand un objet géométrique est *déterminé*. Un point est déterminé par une définition précise (par exemple, comme le milieu de deux points connus) ou par l'intersection de deux lieux. Une droite est déterminée par une définition précise ou par la connaissance de deux points distincts connus lui appartenant ou par la connaissance d'un point et d'une direction, etc. Une mesure du degré de construction d'un objet géométrique réside dans les notions intuitives de degré de liberté maximal d'un type géométrique, de degré de restriction d'une relation et du degré de liberté d'une figure et d'un objet dans une figure.

La notion de *degré de liberté* a une connotation cinématique : le déplacement d'un point du plan, libre de toute contrainte, peut se décomposer suivant deux axes, un point possède donc deux degrés de liberté en translation. Une droite a un degré de liberté en translation et un degré de liberté en rotation. Fixer sa direction supprime le degré de rotation et fixer un point supprime le degré de translation ou le degré de rotation si un autre point de la droite est fixé.

On peut préciser un peu en disant que le degré de liberté d'un type géométrique est le nombre minimal de paramètres qu'il faut connaitre pour déterminer raisonnablement[2] un objet de ce type.

La notion de *degré de restriction* d'une contrainte est liée à celle de degré de liberté : le degré de restriction d'une contrainte est le nombre de degrés de liberté supprimés pour des objets soumis à cette contrainte. Intuitivement, ceci correspond au nombre d'équations, sur les paramètres des objets mis en cause, introduites par la contrainte. Par exemple, la relation d'incidence "appartenir à une droite" a un degré de restriction 1, et un point dont on sait simplement qu'il appartient à une droite connue à un degré de liberté égale à 1 dans cette figure. Un objet qui a un degré de liberté nul dans une figure est en principe connu. Mais ceci est à moduler. Ainsi un point devant appartenir à la fois à un cercle et à une droite donnés n'est généralement pas uniquement déterminé par cette intersection qui contient parfois deux points. Ce phénomène peut s'aggraver de la manière suivante : intuitivement, la relation appartenir à une courbe "simple" (non fractale ou de dimension de Hausdorff égale à 1) a un degré de restriction égal à 1. Ceci entraîne qu'un point appartenant à la fois à une spirale logarithmique (ou à une sinusoïde) et à une droite possède, dans cette figure, un degré de liberté nul alors que cette intersection n'est pas finie mais dénombrable. Rassurons-nous : dans le cadre des constructions à la règle et au compas, et plus généralement lorsque toutes les contraintes se traduisent par des équations polynomiales, tout objet de degré de liberté nul dans une figure est à choisir dans un ensemble fini d'objets déterminés.

2. Exemples divers

Les exercices de construction géométrique présentent une grande diversité dans les contraintes que doivent vérifier les objets et dans les techniques mises en oeuvre pour les résoudre. Nous donnons dans cette section quelques exemples de résolution informelle d'exercices classiques afin d'illustrer cette richesse. Une large palette d'exercices et d'heuristiques pourra être trouvée dans [Petersen 90].

Afin de ne pas compliquer inutilement les constructions développées plus bas, nous nous servons du lemme suivant que nous demandons d'admettre :

Lemme.
On peut construire à la règle et au compas :
- la parallèle à une droite passant par un point donné,
- les parallèles à une droite à une distance donnée de celle-ci,
- le milieu de deux points,
- la médiatrice de deux points,
- les bissectrices de deux droites,
- le symétrique d'un point, d'une droite ou d'un cercle par rapport à un point ou une droite,
- l'image d'un point, d'une droite ou d'un cercle par une homothétie, une translation, une rotation ou une inversion,
- la somme, la différence, le produit et le rapport de deux distances,
- le sinus, le cosinus et la tangente d'un angle.

2C'est-à-dire de manière calculable.

2.1. Raccord

Voici un exercice de construction très simple typique de ce qu'on peut trouver en Dessin Assisté par Ordinateur. Cet exercice qui illustre la méthode des lieux ferait sans doute l'objet d'une seule règle dans le logiciel de B. Aldefeld [Aldefeld 88], [Aldefeld & al 91].

Énoncé 1. *Construire un cercle* Ω *de rayon* r *donné tangent à une droite* D *et à un cercle* C *donnés*.

L'ensemble D_r = { o ∈ P / le cercle de centre o et de rayon r est tangent à D}, appelé le lieu géométrique des centres des cercles de rayon r tangents à D, est la réunion des deux droites Dr1 et Dr2, parallèles à D est à distance r de celle-ci. Le lieu géométrique C_r des centres des cercles de rayon r tangents à C est la réunion des deux cercles Cr1 et Cr2, de centre O, centre de C, et de rayons respectifs R+r et |R-r|, où R est le rayon de C. Le centre du cercle cherché appartient nécessairement à la fois à D_r et à C_r. L'intersection de ces deux ensembles donne de zéro à huit points dont on vérifie facilement qu'ils sont tous solutions. La figure 2 montre un exemple où il y a huit solutions nommées Ω1, ..., Ω8 sur le dessin.

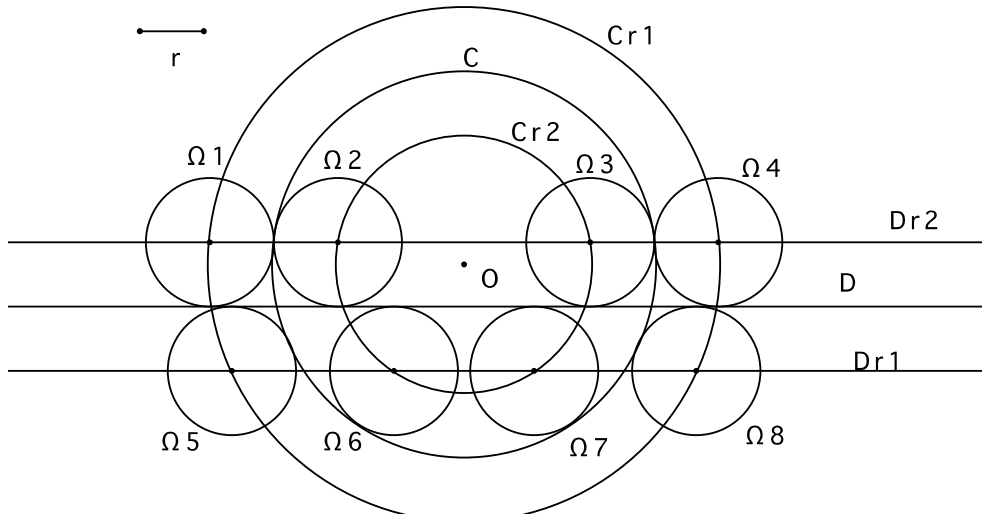

Figure 2 : Huit solutions à l'énoncé 1

2.2. Utilisation d'éléments cachés

Dans cet exercice, on doit mettre en évidence un objet non donné par l'énoncé et qui permet de résoudre le problème. Nous appelons de tels objets des *éléments cachés* ou *objets passerelles*.

Énoncé 2. *Construire un quadrilatère* ABCD *dont on connaît la longueur des côtés et l'angle* α *formé par les droites (*AB*) et (*CD*)*.

Après avoir, par exemple, fixé les points C et D, on peut considérer le point E tel que ABCE soit un parallélogramme : ce point E se construit à partir de C et D car on connaît l'angle formé

par les droites (CE) et (CD) (égal à α) et la distance EC (égale à AB). On construit ensuite le point A comme intersection des cercles c1, de centre E et de rayon BC, et c2, de centre D et de rayon AD. Puis le point B est construit en complétant le parallélogramme (Cf. figure 3). On a, en général, quatre solutions distinctes à isométrie près.

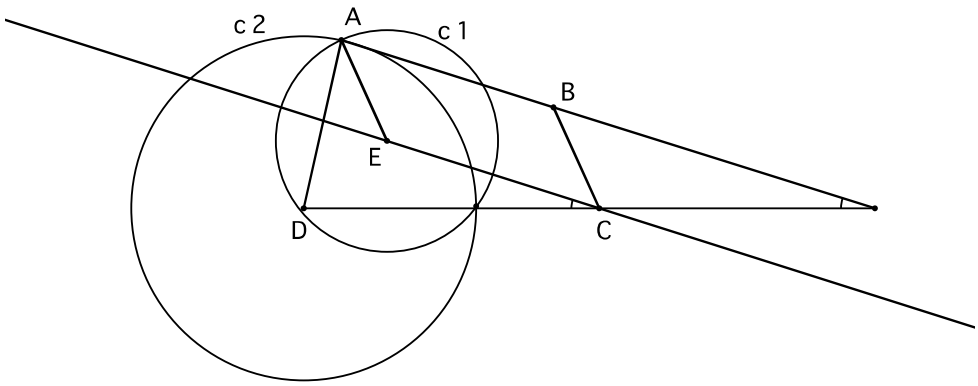

Figure 3 : Une solution pour l'énoncé 2.

On fait ici intervenir un élément non présent dans l'énoncé, et qui ne résulte pas de la transformation directe d'une contrainte, cet élément permettant de matérialiser, dans un certain sens, une contrainte. Cet élément peut être un point, une droite ou un cercle, mais aussi une transformation géométrique comme une homothétie, une translation (comme dans cet exemple) ou une rotation.

2.3. Transformation de problème

On utilise une *transformation géométrique* qui permet de passer du problème à un autre qu'on sait résoudre. On revient au problème initial par une transformation inverse. On peut ainsi utiliser des homothéties, des translations ou des inversions. Voici un exemple assez classique.

Énoncé 3. *Construire un cercle* Ω *passant par les points* A *et* B *donnés et tangent à la droite* D *donnée également.*

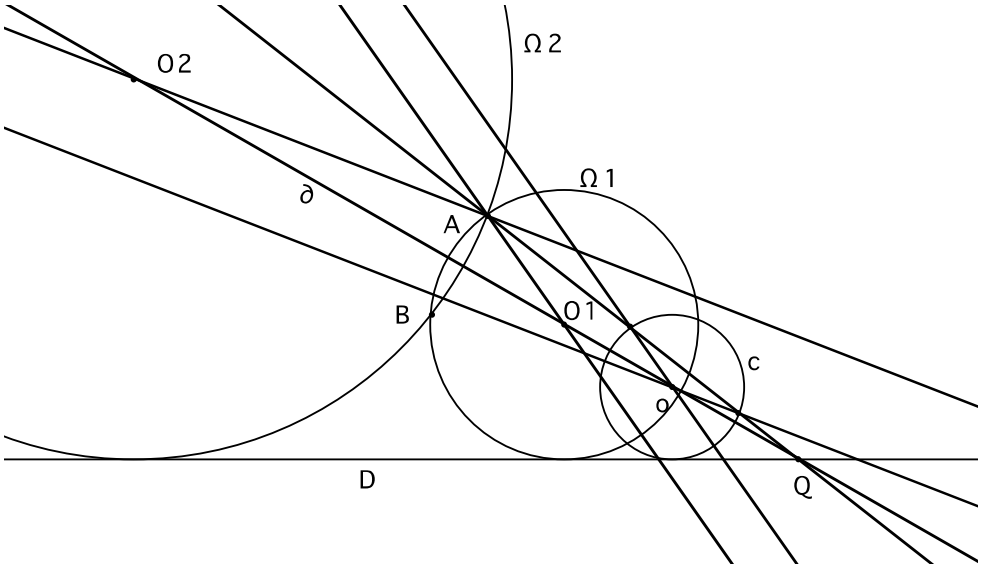

Figure 4 : Deux solutions à l'énoncé 3 lorsque ∂ et D sont concourantes

Soit O le centre de Ω, il suffit de construire O pour résoudre le problème. Comme O est sur la médiatrice ∂ de (A, B), on est ramené au problème de construire un cercle tangent à une droite donnée, dont le centre est sur une droite donnée et qui passe par un point donné. Si D et ∂ sont concourantes en un point Q, on remarque que toute homothétie de centre Q conserve les deux premières contraintes, mais pas la troisième. D'où la construction suivante : on construit un cercle c tangent à D et dont le centre est sur ∂, en choisissant arbitrairement son centre o par exemple. Il existe deux homothéties de centre Q transformant chacune c en un cercle passant par A qui est donc une solution. Si au contraire les droites D et ∂ sont parallèles, on se sert de translations de direction la direction commune de D et ∂ et on opère de la même manière.

La clé de la construction présentée est de transformer le problème en un problème sous-contraint dans lequel on peut choisir un élément qui rende la construction facile. Pour pouvoir utiliser cette méthode, il faut que toutes les contraintes soient conservées par la transformation envisagée sauf une.

Dans d'autres méthodes, l'utilisation de transformations change complétement le problème. Ces méthodes sont d'habitude spécifiques à un type de problèmes. Par exemple, lorsqu'il est question de tangence entre deux cercles, il est parfois utile de considérer une inversion.

2.4. Utilisation de calculs

Pour certains problèmes, il faut parfois faire des *calculs intermédiaires* comme par exemple des triangulations. L'exemple suivant est inspiré de [Roller & al].

Énoncé 4 : *Construire un hexagone* ABCDEF *tel que* A *et* B *sont donnés, les distances* BC, CD, DE, EF *et* FA *et les angles* (,), (,) *et* (,) *sont connus*.

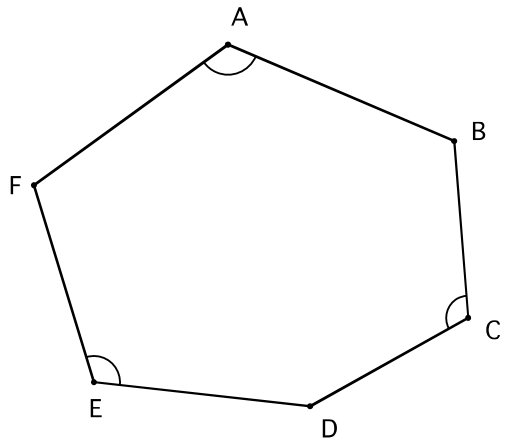

Figure 5 : hexagone de l'énoncé 4

Connaissant la distance AF et l'angle (,), on construit facilement le point F à partir de A et de B. On calcule les distances BD et FD par les formules :

BD =

FD =

D est alors construit par l'intersection des deux cercles de centre B et F et de rayons respectifs BD et FD. Les constructions de C et E s'en déduisent immédiatement.

Les calculs donnés par les formules déterminant BD et FD sont effectuables à la règle et au compas, mais on a un moyen graphique plus simple de calculer ces distances. Par exemple, on construit un triangle BC'D' tel que BC' = BC, C'D' = CD et (,) = (,), on aura alors CD = C'D'.

La figure 6 montre ainsi une construction de D obtenue à partir comme intersection des cercles ∑3 et ∑6 de centres respectifs B et F et passant respectivement par D' et D".

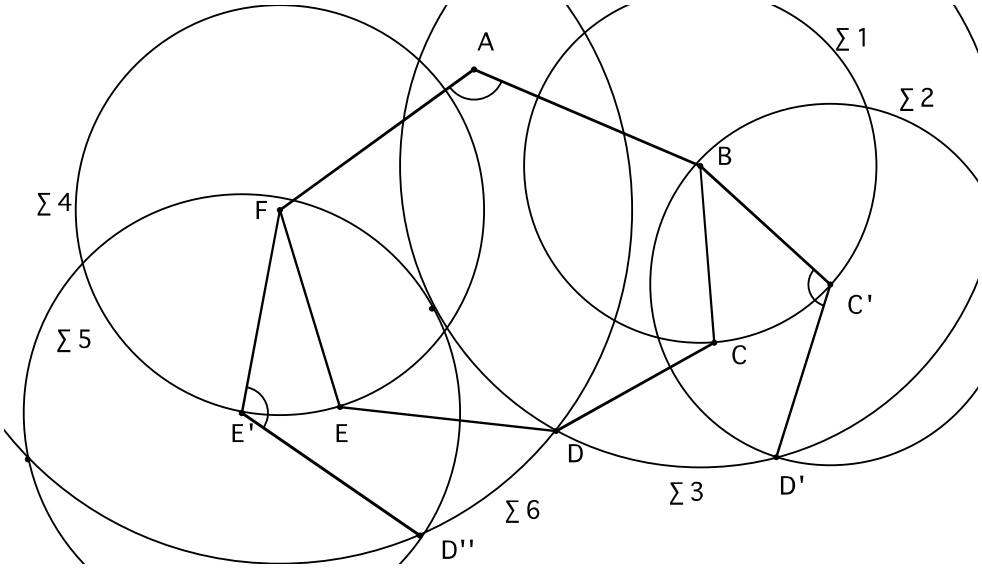

Figure 6 : Une construction pour l'énoncé 4.

Le point C' a été arbitrairement choisi sur le cercle ∑1 de centre B et de rayon la distance BC (donnée) et le point E' a été arbitrairement choisi sur ∑4 de centre F et de rayon la distance FE (donnée). Sur cette figure, on ne montre pas la construction de E et de C.

2.5. Où il faut choisir

On ne peut habituellement pas décider à la lecture d'un énoncé quel *type de méthode* est applicable. En fait, il arrive parfois que toutes ces méthodes soient applicables, comme dans l'exemple suivant.

Énoncé 5. *Construire un triangle* ABC *de périmètre* p *donné et dont les angles sont connus*.

Nous allons dans toutes les méthodes suivantes fixer le point A et la direction de la demi-droite [AB). On peut alors fixer la direction de la demi-droite [AC). Sur la demi-droite [AB), on place le point P tel que AP = p.

Première méthode.
On utilise le cercle ∑ exinscrit opposé à A. En effet, si M est le point de contact de ∑ avec la droite AB, on a AM = . On sait par ailleurs construire la droite AC, puisqu'on connaît l'angle en A, et les bissectrices Δ1 et Δ2 du couple de droites (AB, AC). On peut donc construire ∑ : son centre est à l'intersection de la perpendiculaire à AB passant par M et la bissectrice intérieure de (AB, AC). Pour les points B et C, il suffit alors de construire une tangente à ∑ faisant un angle donné avec la droite AB et telle que A et ∑ soient des deux côtés de cette droite. Ceci se fait facilement à l'aide d'une translation par exemple : on construit une droite t' avec la direction voulue, et on fait une translation pour l'amener au contact de ∑ (le point de contact étant un point de l'intersection de ∑ avec la perpendiculaire en o, centre de ∑, à t').

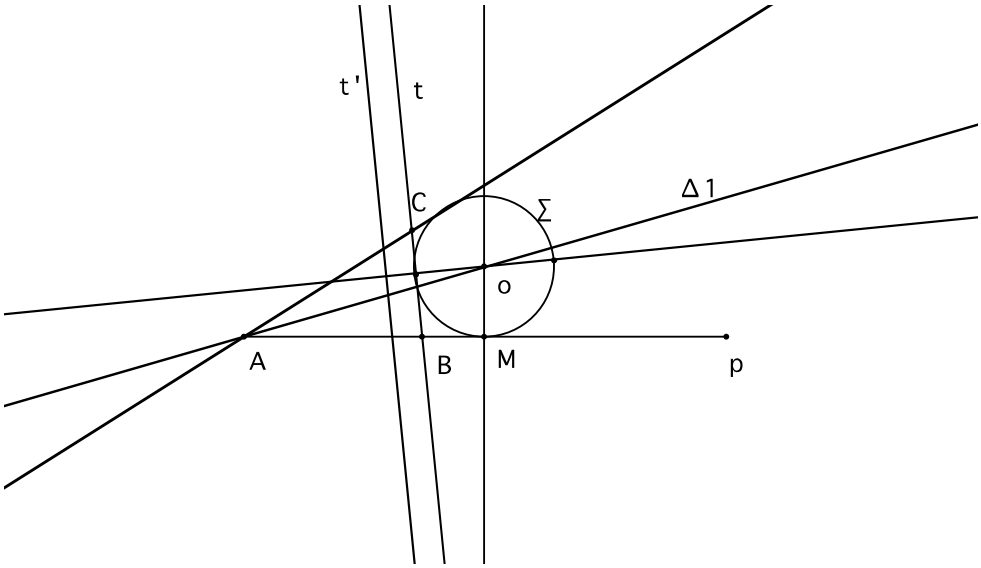

Figure 7 : construction pour l'énoncé 5 avec un cercle exinscrit.

Deuxième méthode.
Puisque les homothéties conservent les angles mais pas les distances, on peut construire un triangle vérifiant les contraintes d'angle et de périmètre quelconque et on utilise une homothétie pour ajuster le périmètre. On peut alors opérer de la manière suivante : les points A et P étant fixés, on prend un point A' arbitraire en dehors de la droite AP, et un point B' tel que la droite A'B' soit parallèle à AB. On construit le point C' de sorte que le triangle A'B'C' satisfasse les contraintes d'angle de l'énoncé, puis, par report de distances, le point A'' sur la droite A'B' tel que la distance A'A'' soit égale au périmètre du triangle A'B'C'. Le point O intersection des droites AA' et PA'' est le centre de l'homothétie cherchée qui par ailleurs transforme A' en A, on en déduit le point B image de B' par cette homothétie, puis le point C. Si les droites AA' et PA'' étaient par hasard parallèles, alors le triangle A'B'C' répondrait à la question, et il suffirait de considérer la translation de vecteur pour amener cette solution au point A fixé au départ.

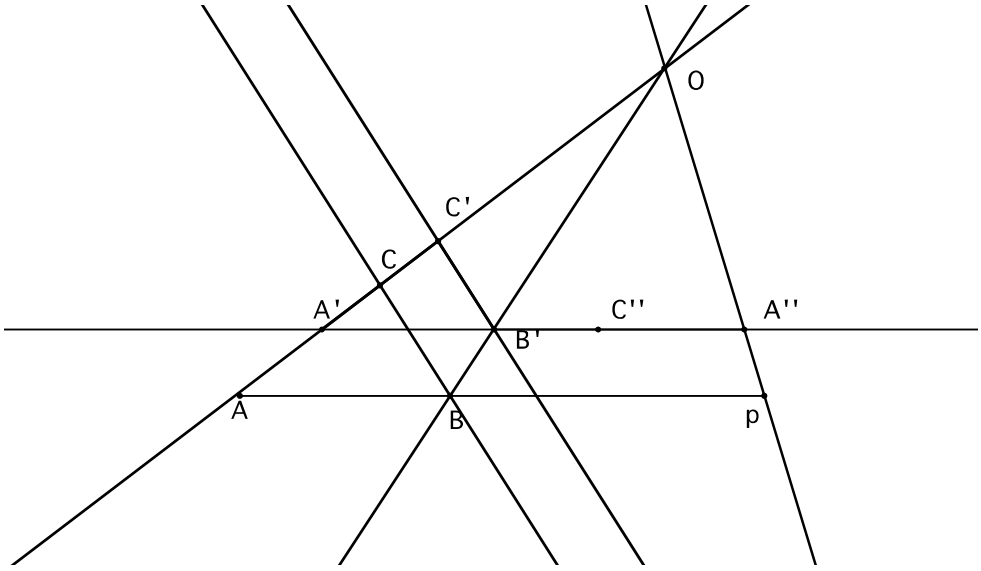

Figure 8 : construction pour l'énoncé 5 avec une homothétie.

Troisième méthode.
On utilise la relation des sinus pour calculer chacun des côtés : si a, b et c sont les longueurs des côtés opposés respectivement aux sommets A, B et C, on a :

= = =

d'où :

a = 6 sin()
b = 6 sin()
c = 6 sin()

Comme nous le verrons plus bas, ces calculs peuvent se faire à la règle et au compas. Une fois la longueur c calculée avec la dernière formule, la construction est évidente.

2.6. Limites physiques

La tradition des constructions géométriques offre parfois des *contraintes d'ordre pratique* comme dans l'exemple suivant :

Énoncé 6. *Construire une bissectrice ∂ de deux droites données* D1 *et* D2 *qui se coupent hors de la feuille de dessin.*

Il suffit de transformer le problème à l'aide d'une translation de vecteur un vecteur directeur de l'une des deux droites, disons D1, et telle que l'intersection de la droite D'2, image de D2 par cette translation, et de D1 soit sur la feuille. Construire D'2, puis une bissectrice ∂' de D1 et D'2 est immédiat. On retourne au problème initial en utilisant la translation inverse. Sur la figure 9, on a choisi un point M sur D1 et dans les limites de la feuille, et on utilise la translation de vecteur = = , A' et B' étant des points choisis sur D'2 et ∂'.

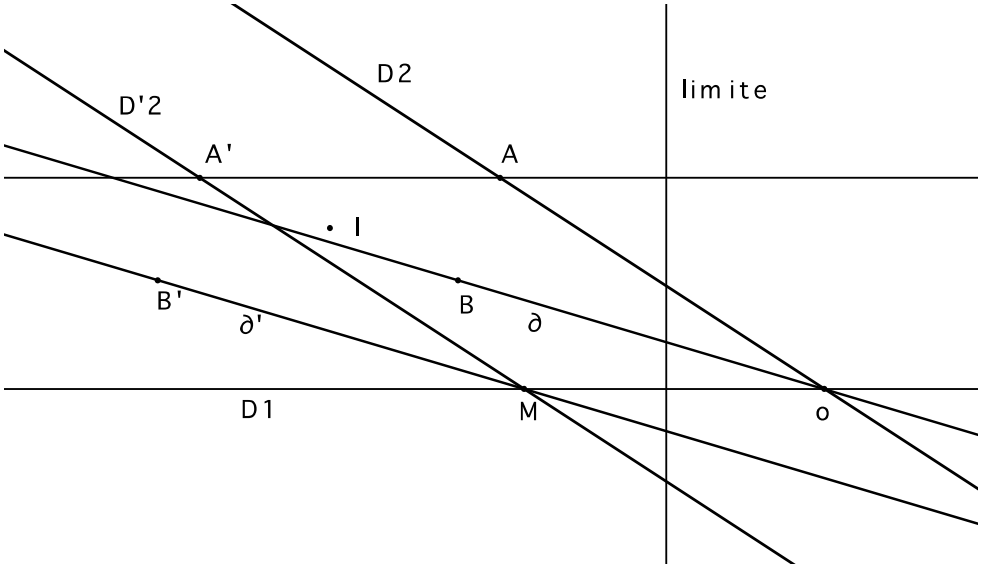

Figure 9 : Bissectrice de deux droites se coupant hors de la feuille de dessin

Sur la figure, le point I, milieu de [A, B'] permet de construire facilement le point B sur ∂.

2.7. Optimisations

Il existe un certain nombre d'exercices de constructions géométriques où il est demandé de construire un des objets minimisant une certaine contrainte numérique, par exemple, minimiser la longueur d'une ligne brisée. Si ces exercices sortent du cadre des constructions à la règle et au compas et font plutôt appel à des techniques d'analyse, on peut parfois les résoudre à l'aide d'un raisonnement géométrique. C'est le cas bien connu de la construction de triangle orthique d'un triangle [Sortais 89], ou encore de l'exercice suivant (problème du tuyau de descente).

Énoncé 7. *étant donné un rectangle* ABCD, *construire un point* M *à l'intérieur de ce rectangle tel que la somme des distances* AM, BM *et de la distance* MH *de* M *à la droite* CD *soit minimale* (Cf. figure 10).

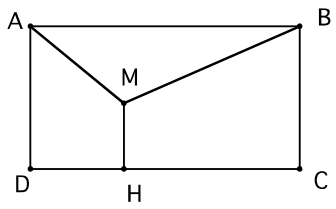

Figure 10 : exemple de figure non optimale

L'ensemble des points M tels que AM + BM soit égale à une constante k est une ellipse de foyers A et B. Pour chaque ellipse de cette famille d'ellipses homofocales, il n'y a qu'un point minimisant la distance de ce point à la droite CD et il se trouve sur la médiatrice de (A, B). Le point M doit donc se trouver sur cette médiatrice. Si on considère maintenant la figure image par une rotation de centre A et d'angle , on obtient les points B', C', D', M' et H' (Cf. figure 11).

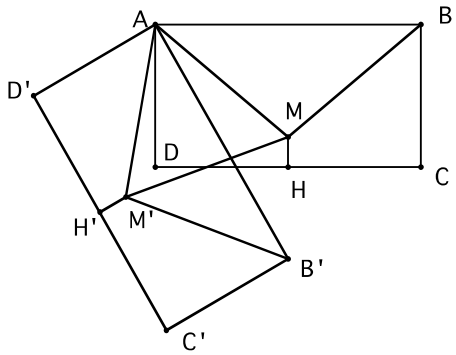

Figure 11: utilisation de la rotation de centre A et d'angle

Puisque AM = BM, on a : AM + BM + MH = BM + MM' + MH'. Or, parmi toutes les lignes brisées joignant B à la droite C'D', le segment joignant B à son projeté orthogonal Hb sur C'D' est la plus courte, donc le point M doit être sur la droite passant par B, faisant un angle de .

3. Conclusion

A travers les définitions et les exemples de ce chapitre, nous avons esquissé la problématique des constructions à la règle et au compas. Après une *phase d'analyse* où des propriétés intéressantes de la configuration donnée par l'énoncé sont exprimées, suit une *phase de synthèse* où l'on élabore un programme de construction à l'aide des propriétés précédemment déduites. Habituellement, la construction trouvée est discutée pour examiner le nombre de solutions et le domaine de validité de la construction pour chaque solution. Éventuellement, on essaye de trouver d'autres méthodes de construction en dehors de ce domaine. Cette phase de discussion est pénible et souvent négligée. Elle fait cependant apparaître deux phénomènes caractéristiques qui sont l'existence de cas dégénérés et la possibilité de solutions multiples.

A partir d'une formulation trompeusement simple, les problèmes de constructions à la règle et au compas offrent une grande diversité des contraintes envisagées et des méthodes de résolution. Au contraire des problèmes de construction géométrique en DAO/CAO, ceux proposés dans l'enseignement secondaire ne font habituellement intervenir que peu d'objets géométriques mais mettent en oeuvre des raisonnements subtils comme dans l'élimination d'une contrainte par transformation de problème. En particulier, les constructions faisant intervenir des points arbitraires nous semblent difficiles à automatiser. Elles sont pourtant présentes dans nombre de constructions de base comme la construction d'une parallèle ou d'une perpendiculaire à une droite donnée ou la construction des tangentes communes à deux cercles.

Chapitre 2

Point de vue algébrique

Ils le traquaient, armés d'espoir, de dés à coudre,
De fourchettes, de soin ; ils tentaient de l'occire
Avec une action de chemin de fer ; ou de
Le charmer avec du savon et des sourires.

Il paraît plus facile de dérouler des calculs que de raisonner sur une figure. Le cadre algébrique fournit en effet de puissants outils pour explorer le domaine des problèmes des constructions géométriques. Outre les résultats théoriques concernant les constructions à la règle et au compas[3], des méthodes pratiques ont été tirée de l'algébrisation des problèmes de construction. Ces méthodes sont soit itératives comme dans [Light & Gossard 82], soit à base de propagation de contrainte comme dans [Owen 91]. Ces méthodes sont numériques et posent parfois certains problèmes [Roller & al 88]. A notre connaissance, une seule méthode formelle a été décrite, et ce dans les années 40, par H. Lebesgue [Lebesgue 50]. Cette méthode a été étudiée par G. Chen dans le cadre d'un stage de DEA co-encadré par le professeur J. F. Dufourd et par l'auteur de ce rapport et qui a eu lieu au Département d'Informatique de l'Université Louis Pasteur à Strasbourg [Chen 92] et c'est son travail que nous allons exposer dans ce chapitre.

La section 1 présente les résultats classiques sur la caractérisation des nombres constructibles. Ces résultats permettent d'envisager un procédé algébrique permettant de résoudre formellement des exercices de construction à la règle et au compas. Nous présentons ce procédé sur un exemple à la section 2. Les sections 3 et 4 reprennent plus en détail deux points cruciaux de ce procédé. La section 3 présente un algorithme de triangulation d'un système polynomial (méthode de Ritt-Wu). La section 4 présente l'algorithme de Lebesgue permettant de résoudre une équation polynomiale, à coefficient dans un corps adéquat, par radicaux carrés. La section 5 explique pourquoi nous n'avons pas utilisé cette méthode.

1. Principaux résultats sur la constructibilité

En ce qui concerne les nombres constructibles, on a le théorème suivant.

Théorème (corps des nombres constructibles). L'ensemble des nombres constructibles forme un corps $\mathbb{C}$ qui est le plus petit sous-corps de $\mathbb{R}$ stable par racine carrée.

Preuve

La démonstration que $\mathbb{C}$ soit un corps stable par racine carrée est assez simple. En effet, 0 et 1 sont dans $\mathbb{C}$, et

3Ces résultats ont répondu à des conjectures qui ont résisté plus de vingt siècles à la sagacité des mathématiciens et non des moindres.

- si $u \in \mathbb{C}$ alors $-u \in \mathbb{C}$, car on a la construction suivante.

Figure 1 : construction de l'opposé

Soit A le point de l'axe Ox d'abscisse u. Le cercle de centre O et passant par A recoupe l'axe Ox en un autre point A' d'abscisse $-u$.

- si u et v appartiennent à $\mathbb{C}$ alors $u+v$ et $u.v$ aussi, car pour la somme, on construit facilement les points A et B sur Ox tels que = u et = v. On a alors = $u+v$.

Pour le produit, on considère les points A et B respectivement sur Ox et Oy, d'abscisse u pour A et d'ordonnée v pour B (Cf. figure 2). La parallèle à la droite (JA) passant par B coupe l'axe Ox en C d'abscisse uv. Il suffit de considérer l'homothétie h de centre O et de rapport v pour s'en convaincre.

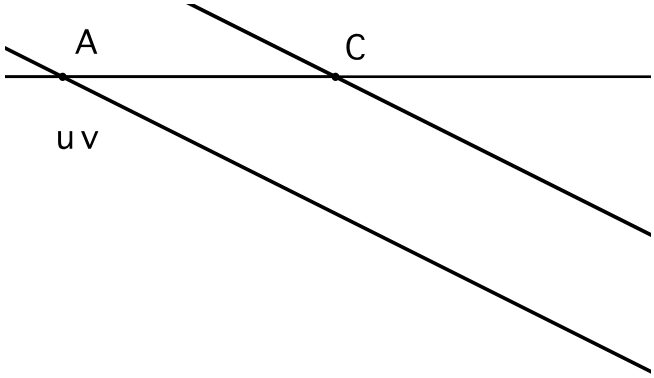

Figure 2 : construction du produit

- si $u \in \mathbb{C}^*$, alors $\in \mathbb{C}$. En effet, si A est le point d'abscisse u de l'axe Ox, la parallèle à la droite (AJ) passant par I coupe l'axe Oy en le point B d'ordonnée .

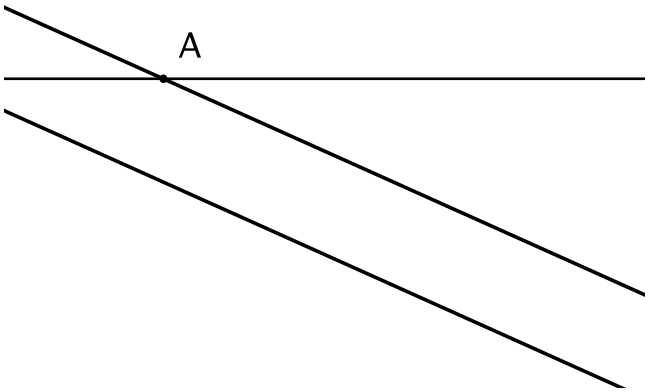

Figure 3 : construction de l'inverse

- si $u \in \mathbb{C}$ et $u > 0$ alors $\in \mathbb{C}$. Ceci résulte de la construction suivante : à partir du point A sur l'axe Ox d'abscisse $1+u$, on trace le cercle de diamètre [OA], ce qu'on sait faire à la règle et au compas d'après un lemme du chapitre 1. L'intersection de ce cercle avec la perpendiculaire à l'axe Ox passant par I donne deux points : l'un, noté B sur la figure 4, est d'ordonnée et l'autre

d'ordonnée -. En effet, le triangle OAB étant rectangle en B et I étant le pied de la hauteur issue de B, on a $BI^2 = 1.u$, d'où le résultat.

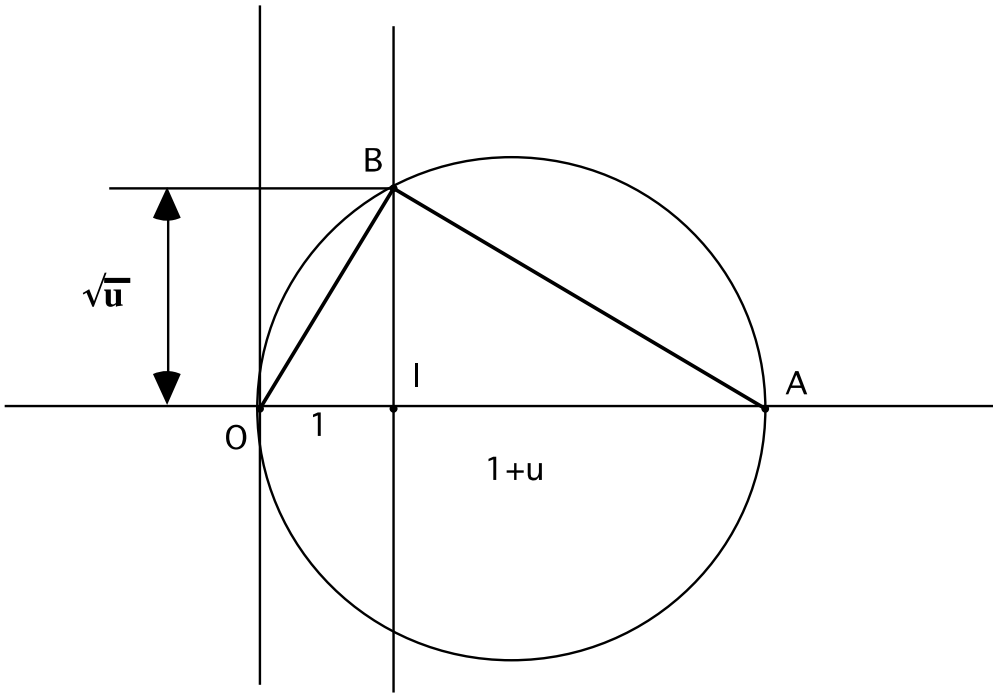

Figure 4 : construction de la racine carrée

La preuve de la minimalité est un corollaire du théorème fondamental suivant dont la démonstration peut être trouvée par exemple dans [Carrega 89].

Théorème (caractérisation des nombres constructibles). Soient α un nombre algébrique, $P(X)$ son polynôme minimal sur $\mathbb{Q}$, et K le corps de dislocation de $P(X)$. Pour que α soit constructible à la règle et au compas il faut et il suffit que le degré $[K : \mathbb{Q}]$ soit une puissance de 2.

Corollaire 1.
Un nombre réel est constructible si et seulement si il s'exprime à l'aide de radicaux carrés à partir de $\mathbb{Q}$.

Corollaire 2.
$\mathbb{C}$ est le plus petit sous-corps de $\mathbb{R}$ stable par racine carrée.
□

Ces résultats se généralisent lorsque l'ensemble de base contient un nombre fini de points dont les coordonnées sont soit des rationnels, soit des indéterminées p_1, p_2 ... p_n. Un problème se posera cependant sur la détermination du signe des expressions polynomiales en p_1, p_2 ... p_n lorsqu'on tentera d'en considérer la racine carrée.

2. Algébrisation et résolution de systèmes polynomiaux

Les exercices courants de construction ne se réduisent en général pas immédiatement à des problèmes de constructibilité d'un réel à partir des points de coordonnées (0, 0) et (1, 0) :
- on doit construire d'autres types d'objets que des points,
- on part d'une figure donnée (et non pas de deux points) éventuellement à compléter,
- et l'on a des paramètres et non des valeurs numériques.

Par ailleurs, il faut savoir traduire en toute généralité un énoncé géométrique en un énoncé algébrique c'est-à-dire en un système polynomial. Nous exposons notamment à l'aide d'un exemple les différentes étapes d'une résolution algébrique formelle d'un problème de construction. Les méthodes de Ritt-Wu et de Lebesgue avec les simplifications dues à G. Chen seront exposées plus en détail dans les sections 3 et 4.

2.1. Traduction d'un énoncé géométrique en un énoncé algébrique

En premier lieu, il faut décider d'une représentation à l'aide de coordonnées pour chaque type d'objet. Cette représentation n'est en général pas intrinsèque et se fait via le choix d'un système de représentation. Plus précisément, on doit avoir pour chaque type d'objet O une bijection φ de O dans $K_1 \times K_2 \times ... \times K_n$ où K_i désigne un intervalle de . Etant donné un objet o de type O, chaque membre du n-uplet $\varphi(o)$ est une coordonnée de l'objet on dit que φ est un système de coordonnées pour le type d'objet choisi. En géométrie, ces correspondances se font à travers le choix d'un repère. On peut par exemple, après avoir fixé un repère du plan, utiliser les coordonnées suivantes :

- pour le type point, on utilise les coordonnées cartésiennes dans le repère fixé.

- pour le type droite, on associe à chaque droite un couple de $\times$ $\mathbb{R}$:
  (a, b) si la droite n'est pas parallèle à l'axe Oy et ceci représentera la droite d'équation $y = ax + b$ dans le repère fixé,
  ($\perp$, c) pour représenter les droites d'équation $x = c$

- pour le type cercle, on caractérise un cercle par son centre et son rayon, on lui associe donc un triplet de $\mathbb{R}^2 \times \mathbb{R}_+$ où les deux premières coordonnées sont celles du centre dans le repère et la troisième, le rayon.

Il existe bien entendu d'autres systèmes de représentation comme coordonnées polaires pour un point, podaire de l'origine pour une droite, coefficients de l'équation cartésienne pour un cercle. Si des instruments sont imposés, il faut vérifier qu'on peut tracer les objets à partir des systèmes de cordonnées décrits à l'aide des intruments autorisés uniquement.

Les contraintes sont des relations imposées ou déduites entre des objets. Ces relations se traduisent par des relations algébriques entre les coordonnées des objets ainsi contraints. Généralement, une contrainte r est représentée par un système polynomial (S) à plusieurs variables : on a $r(o_1, o_2, ..., o_n)$ si et seulement si les coordonnées de $o_1, o_2, ... o_n$ vérifient le système (S). La représentation algébrique des contraintes dépend donc des systèmes de coordonnées choisis pour les types d'objets considérés. Dans toute la suite, nous ne considérons que des contraintes se traduisant en systèmes polynomiaux en les coordonnées.

Exemple.

Énonce : *Étant donné un cercle* Γ, *construire un triangle* ABC *équilatéral inscrit dans* Γ.

Ce problème possède une infinité de solutions se déduisant de l'une d'elle par une rotation de centre le centre de Γ ou par la composition d'une symétrie axiale d'axe passant par le centre de Γ et une rotation. Nous allons donc construire une solution particulière. Par ailleurs, pour simplifier, nous faisons l'hypothèse que Γ est de rayon 1 : nous aurons les solutions du cas général à une homothétie prés. Nous considérons alors le repère orthonormé (O, I, J) tel que O est le centre de Γ. J est donc sur Γ et nous chercherons la solution telle que C = J. Dans ce repère, les points A et B sont de coordonnées respectives (x_1, x_2) et (x_3, x_4), et il faut déterminer les variables x_1, x_2, x_3, x_4.

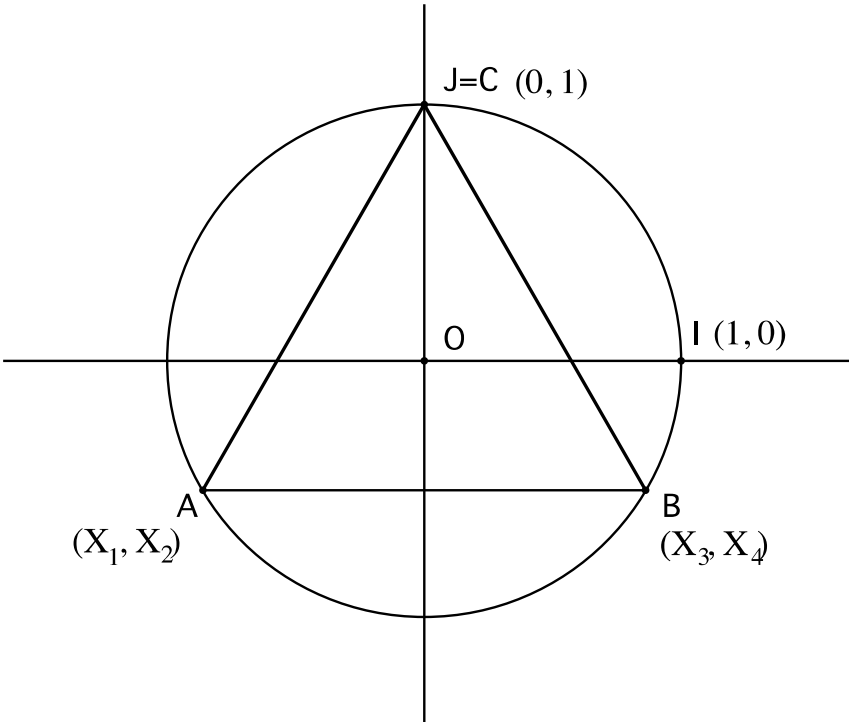

Figure 5 : triangle équilatéral inscrit dans le cercle unité

L'énoncé se traduit par le système *S* d'équations suivantes :

$f_1 = x + x - 1 = 0$ (A est sur Γ)

$f_2 = x + x - 1 = 0$ (B est sur Γ)

$f_3 = x + (x_2 - 1)^2 - (x_3 - x_1)^2 - (x_4 - x_2)^2 = 0$ ($JA^2 = AB^2$)

$f_4 = x + (x_4 - 1)^2 - (x_3 - x_1)^2 - (x_4 - x_2)^2 = 0$ ($JB^2 = AB^2$)

Il faut maintenant résoudre ce système par radicaux carrés.

2.2. Triangulation du système obtenu

Pour résoudre un tel système, il faut commencer par le trianguler, puis résoudre les équations les unes après les autres. Si dans le cas où toutes les équations sont linéaires, il existe des algorithmes simples pour trouver un système triangulaire équivalent, il se pose plusieurs problèmes avec des équations polynomiales générales. En particulier, il n'y a pas qu'un seul système triangulaire équivalent au premier : il faut discuter de la nullité des coefficients dominants et de l'irréductibilité des polynômes traités. Ceci fait l'objet de la théorie de l'élimination [Kalkbrener 91]. La méthode de Ritt-Wu que nous présenterons plus loin permet de calculer ces systèmes triangulaires.

Exemple.

Dans le cas particulièrement simple constitué par notre exemple, on n'obtient qu'un seul système triangulaire :

$g_1 = 4x - 3 = 0$

$g_2 = 2x_2 + 1 = 0$

$g_3 = -128x_3x_1 - 96 = 0$

$g_4 = -294912 - 589824x_4 = 0$

2.3. Résolution par radicaux carrés

Une fois les systèmes triangulaires obtenus, il reste à résoudre des équations polynomiales irréductibles à une seule inconnue. D'après le théorème caractérisant les nombres constructibles, ces équations doivent avoir pour degré une puissance de 2. On ne sait pas résoudre de telles équations formellement dans le *cas général*. Mais, dans le cadre de la résolubilité par radicaux carrés, H. Lebesgue a exposé une méthode permettant de dire si une équation irréductible a des solutions exprimables à l'aide de radicaux carrés et dans ce cas de calculer formellement ces solutions [Lebesgue 50]. Dans notre exemple, il est inutile d'employer cette méthode. Les polynômes étant de degré au plus 2, on connait l'expression formelle de leurs racines, et on trouve les deux solutions :

$$x_1 = \varepsilon,\ x_2 = -\ ,\ x_3 = -\varepsilon,\ x_4 = -\ ;\ \text{avec } \varepsilon = \pm 1$$

2.3. Traduction en constructions géométriques

L'expression formelle des solutions peut être traduite en construction géométrique en utilisant la première partie de la démonstration du théorème 1 : on transforme ainsi géométriquement les sommes, différences, produits, quotients et extraction de racines carrées.

Exemple.
Dans l'exemple que nous avons pris, il nous faut construire de cette manière x_1, x_2, x_3 et x_4.
Pour x_1, on construit tout d'abord . Le point M_1 sur Ox d'abscisse 4 est obtenue en reportant trois fois la distance OI sur Ox à partir de I. On trace le cercle C de diamètre $[OM_1]$ (construction supposée connue pour ne pas alourdir), puis M_2, point d'intersection d'ordonnée positive de C et de la perpendiculaire à Ox passant par I. Soit M_3 le projeté orthogonal de M_2 sur Oy, M_3 est d'ordonnée .

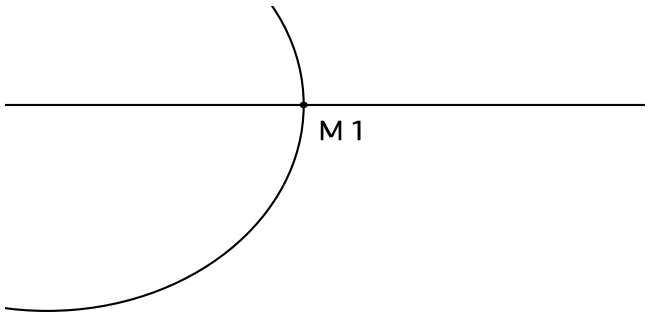

Figure 6 : construction du point M_3 d'ordonnée

On construit facilement N_1 sur Oy d'ordonnée 2, puis N_2 intersection de la parallèle à (N_1I) passant par J avec l'axe Ox. N_2 est d'abscisse .

Figure 7 : construction de $\mathbf{N}_2$ d'abscisse

La parallèle à (N_2J) passant par M_3 coupe l'axe Ox en M_4 d'abscisse . On construit ensuite M_5 sur Ox d'abscisse - ; puis M_6 sur Oy d'ordonnée - . On obtient ainsi la figure 8.

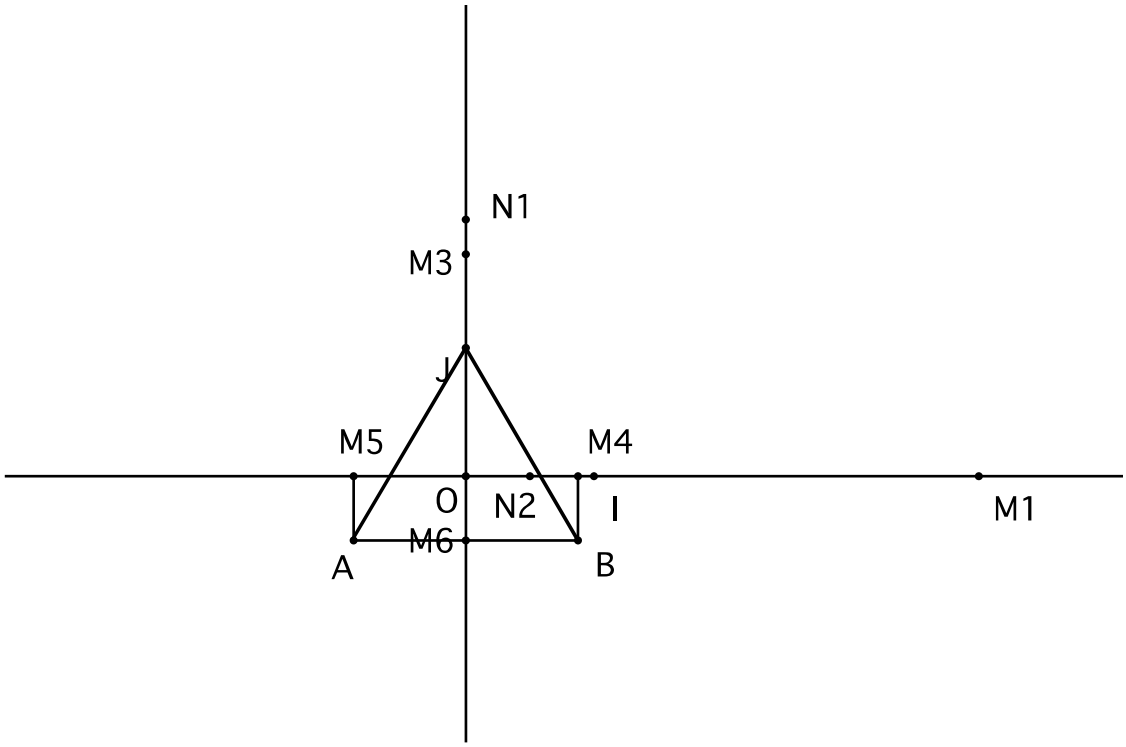

Figure 8 : fin de la construction.

Le cercle unité a été ajouté à la figure pour vérifier que les points A et B lui appartiennent bien, mais il n'a pas été utilisé pour la construction.

3. Méthode de Ritt-Wu

La méthode de Ritt-Wu propose un algorithme permettant de trouver les ensembles caractéristiques d'un ensemble fini *S* de polynômes [Chou 88], [Kalkbrenner 91]. En d'autres termes, elle permet de trouver à partir de *S* des systèmes polynomiaux triangulaires ayant "presque" les mêmes racines communes que *S*. Nous présentons les grandes lignes de cette méthode (pour plus de détails, le lecteur pourra avantageusement se référer à [Chou 88], [Kalkbrenner 91] et [Chen 92]).

3.1. Notations et définitions

Soient *K* un corps de caractéristique nulle, $x_1, \ldots x_m$ des indéterminées. On note

$$A = K[x_1, \ldots x_m] = K[x]$$

l'anneau des polynômes à coefficients dans *K* en les indéterminées $x_1, \ldots x_m$. Les variables sont ordonnées par leur indice : $x_1 < x_2 \ldots < x_m$.

Soit *f* un polynôme dans *A*. On note *deg*(*f*, x_i) le degré de *f* par rapport à x_i, *classe*(*f*) le plus petit entier *c* tel que *f* appartienne à $K[x_1, \ldots, x_c]$, $vp(f) = x_{classe(f)}$ la *variable principale* de *f* et *ci*(*f*) le *coefficient initial* de *f*, i.e. le coefficient du terme de degré *deg*(*f*, *vp*(*f*)) en *vp*(*f*) de *f*.

Point de vue algébrique

On pose *classe*(*f*) = 0 si *f* est un polynôme constant. On dit qu'un polynôme *g* est *réduit* par rapport à *f* non constant si *deg*(*g*, *vp*(*f*)) < *deg*(*f*, vp(*f*)).

L'anneau *A* n'est pas euclidien, on a cependant une *pseudo-division* suivant une variable, définie de la manière suivante.

Théorème et définition (pseudo-division). Soit *f* et *g* deux polynômes de *A*. Il existe un entier *j* et deux polynômes uniques *q* et *r* tels que

$$(ci(f))^j \, g = qf + r$$

avec $0 \le j \le deg(g, vp(f)) - deg(f, vp(f)) + 1$ et $deg(r, vp(f)) < deg(f, vp(f))$ (autrement dit, *r* est réduit par rapport à *f*).
On appelle *r* le *pseudo-reste* de *g* par *f* par rapport à *vp*(*f*) et on le note *prem*(*g*, *f*).
□

On peut calculer le pseudo-reste en utilisant un algorithme issu directement de celui d'Euclide. Cette pseudo-division est exploitée dans les mécanismes d'élimination utilisés dans la démonstration du principe de Ritt.

On définit un *ordre partiel* sur K[*x*] par : *f* < *g* si et seulement si
- soit *classe*(*f*) < *classe*(*g*),
- soit *classe*(*f*) = *classe*(*g*) et *deg*(*f*, vp(*f*)) < *deg*(*g*, vp(*f*)).

On dit alors que *f* est de rang inférieur à *g*. Si on n'a ni *f* < *g*, ni *g* < *f*, on a *classe*(*f*) = *classe*(*g*) et *deg*(*f*, *vp*(*f*)) = *deg*(*g*, *vp*(*f*)). On dit alors que *f* et *g* sont de même rang, ce que l'on note par *f* ~ *g*.

Définition (suite ascendante). Une suite (f_1, f_2, ... f_r) de polynômes de *A* est une *suite ascendante* si
soit *r* = 1 et *f*1 n'est pas le polynôme nul,
soit *r* > 1 et 0 < *classe*(f_1) < *classe*(f_2) ... < *classe*(f_r) et pour tout couple (*i*, *j*) tel que *i* < *j* , f_j est réduit par rapport à f_i.

□

Ainsi, une suite ascendante induit un système polynomial triangulaire et réduit. On définit *le pseudo-reste prem*(*g*, *C*) d'un polynôme *g* par rapport à une suite ascendante *C* = (f_1, ... f_r) par *prem*(*g*, *C*) = *prem*(... *prem*(*prem*(*g*, f_r), f_{r-1}) ..., f_1). On a alors la relation suivante :

$$bg = q_1 f_1 + ... + q_r f_r + prem(g, C)$$

où *b* est un produit de puissances des coefficients initiaux I_i des f_i : *b* =*II*... *I*
Le pseudo-reste *prem*(*g*, *C*) est réduit par rapport à chacun des f_i, on dit qu'il est réduit par rapport à *C*.

Définition et théorème (suite ascendante minimale). Soient *C* = (f_1, ..., f_r) et *C*' = (g_1, ..., g_t) deux suites ascendantes. On dit que *C* est *inférieure* à *C*' si et seulement si l'on a

- soit l'existence d'un entier $s \leq min(r, t)$ tel que $f_i \sim g_i$ pour $i < s$ et $f_s < g_s$,

- soit $t < r$ et $f_i \sim g_i$ pour $i \leq t$.

Avec cet ordre partiel, chaque ensemble S de polynômes, non vide et non réduit au polynôme nul, contient une suite ascendante minimale. Dans le cas où S est fini, il existe un algorithme fournissant une telle suite.

□

Proposition. La relation d'ordre partielle ainsi définie sur l'ensemble des suites ascendantes est bien fondée, c'est-à-dire qu'il n'existe pas de suite infinie de suites ascendantes qui soit strictement décroissante.

□

Soit C une suite ascendante minimale d'un ensemble fini de polynômes S, ne contenant pas de constante. C possède les deux propriétés importantes suivantes :
- chaque zéro de S est un zéro de C (car $C \, 1 \, S$),
- C est triangulaire.

Cependant, l'ensemble des racines communes de C que l'on note $V(C)$ est généralement plus grand que l'ensemble des racines communes de S, $V(S)$, i.e. $V(S)$ est strictement inclus dans $V(C)$. Ce résultat était prévisible : on ne peut pas d'habitude extraire directement un système triangulaire de S, il faut transformer S.

3.2. Principe de Ritt

La notion d'ensemble caractéristique apporte une première réponse à cette question d'équivalence entre systèmes.

Définition (ensemble caractéristique). Soit S un sous-ensemble de A non vide et non réduit au polynôme nul. Une suite ascendante C (non forcément incluse dans S) est un *ensemble caractéristique* de S si
i) chaque zéro de S est un zéro de C,
ii) chaque zéro de C qui n'est zéro d'aucun des coefficients initiaux des polynômes de C est zéro de S.

□

Le théorème suivant indique l'existence d'un algorithme pour obtenir un ensemble caractéristique d'un ensemble fini de polynômes.

Théorème (Principe de Ritt). Soient $S = \{f_1, ..., f_n\}$ un ensemble fini non vide de polynômes non nuls de A et I l'idéal engendré par $f_1, ..., f_n$. Il existe un algorithme pour obtenir une suite ascendante C telle que :
- soit C contient un seul polynôme constant,
- soit $C = (h_1, ..., h_r)$ avec classe$(h_i) > 0$ et telle que pour tout $i = 1, ..., r$ et $j = 1, ..., n$ on ait $h_i \quad I$ (i) et prem$(f_j, h_1, ... h_r) = 0$ (ii)

La suite ascendante C est donc un ensemble caractéristique de S.

Ebauche de preuve.

Point de vue algébrique

On calcule une suite d'ensemble S, S_1, S_2, ... S_n obtenue en ajoutant successivement à chaque S_i l'ensemble des pseudo-restes non nuls des polynômes de S_i par rapport à une suite minimale ascendante C_i de S_i (on a vu qu'on sait en trouver un). On obtient une suite de suites ascendantes $C_1 > C_2 ... > C_n$ qui doit être finie car l'ordre donné plus haut sur les suites ascendantes est bien fondé.
Au rang n, soit S_n contient une constante et on arrête, soit $S_n = S_{n+1}$. Dans ce cas, comme tous les polynômes ajoutés étaient dans I, les h_i finaux sont aussi dans I (i), et, comme $S_n = S_{n+1}$, tous les pseudo-restes des polynômes de S_n par rapport à C_n sont nuls (ii).

□

On peut ainsi obtenir tous les zéros de S qui ne sont pas des zéros des coefficients initiaux. On peut aller encore plus loin et obtenir exactement l'ensemble $V(S)$ des zéros de S sous forme d'une réunion d'ensemble de zéros d'ensembles caractéristiques.

Définition (irréductibilité). Une suite ascendante (f_1, ... f_r) est dite *irréductible* si chaque f_i est irréductible dans l'anneau des polynômes $F_{i-1}[x_i]$ où les F_i sont définis par :

$F_0 = K$, $F_1 = F_0[x_1]/(f_1)$, ..., $F_r = F_{r-1}[x_r]/(f_r) = F_0[x_1, ... x_r]/(f_1, ... f_r)$.

□

Théorème (algorithme de décomposition de Ritt). Il existe un algorithme pour décider si, pour chaque ensemble S fini non vide de polynômes de A, l'idéal engendré par S est A, ou, dans le cas contraire, décomposer $V(S)$

$$V(S) = V(P_1) \cup V(P_2) ... \cup V(P_s)$$

où les P_i sont des idéaux premiers engendrés par un ensemble caractéristique irréductible.

□

L'idée de cet algorithme -que nous ne détaillerons pas ici- est de factoriser les polynômes réductibles rencontrés dans les ensembles caractéristiques et d'ajouter à ces mêmes ensembles les coefficients initiaux pour obtenir d'autres ensembles caractéristiques.

Le problème de la triangularisation d'un système polynomial reçoit donc une solution complète si on sait factoriser les polynômes à coefficient dans un corps de la forme $\mathbb{Q}(p_1, p_2, ... p_m, \mu_1, ..., \mu_t)$ où les p_i sont des *indeterminées*,i.e. des *éléments transcendants* sur $\mathbb{Q}$, et chaque μ_j est un nombre algébrique sur $\mathbb{Q}(p_1, p_2, ... p_m, \mu_1, ..., \mu_{j-1})$.

Exemple ([Chen 92]).
L'ensemble $S = \{f_1, f_2, f_3, f_4\}$ est un ensemble des polynômes obtenus en traduisant algébriquement l'exemple donné au chapitre 3, avec

$f_1 = x + x - 1$

$f_2 = x+ x- 1$

$f_3 = x- 2x_3p_1 + x- x+ 2x_1p_1 - x$

$f_4 = x_1x_4 - x_1p_3 - p_2x_4 - x_2x_3 + x_2p_2 + p_3x_3$

L'application de l'algorithme de décomposition de Ritt à l'ensemble *S* conduit à la décomposition

$$V(S) = V(P_1) \text{ " } V(P_2)$$

P_1 et P_2 étant engendrés par les ensembles caractéristiques C_1 et C_2 suivants

pour C_1, obtenu directement à partir de S en appliquant le principe de Ritt :

$g_1 = x+ x- 1$

$g_2 = (x_1 - p_2)^2(x_1 - x_3)(2x_1p_2 - p+ 2p_3x_2 - 1 - p)$

$g_3 =(x_1 - p_2)^3(2p_3x+ 2x_1p_2x_4 - 2x_2p_2x_1 - px_4 - x_4 + p- px_4 + px_2 + x_2 - 2p_3 + 2p_3x_2x_4)$

pour C_2, obtenu en considérant le coefficient initial de g_2 :

$g= x_1 - p_2$

$g= x- 1 + p$

$g= (p_3 - x_2)(p_2 - x_3)$

$g= (x) - 1 + p)(p+ 2p_3x_2 - p- 1)$

Autrement dit, pour résoudre le système *S*, il faut résoudre les deux systèmes C_1 et C_2.

3.3. Problème de la factorisation de polynômes.

L'étude de la factorisation est un sujet trés étudié [Kartofeln 83] depuis de nombreuses années. Dans le cas de polynômes à coefficients dans $K = \mathbb{Q}(p_1, ..., p_m)$ où les p_i sont des indéterminées, ces études ont mené à des algorithmes maintenant classiques ([Viry 78], [Wang 76], [Weinberger & Rothschild 76], etc) qui sont utilisés dans des systèmes de calcul formel.

Dans le cas d'un polynôme $P(X)$ à coefficients dans $K(\mu)$, où μ est un nombre algébrique sur *K* défini par son polynôme minimal, H. Lebesgue utilise une méthode de recherche systématique des facteurs de degré *r* donné en résolvant un système polynomial dans *K* obtenu par division euclidienne de *P* par l'hypothétique facteur [Lebesgue 50]. Pour résoudre ce système, on peut utiliser une version simplifiée de l'algorithme de décomposition de Ritt.

Dans le cas d'un polynôme $P(X)$ de $K(\mu_1, \mu_2, ..., \mu_m)[X]$ où chaque μ_i est algébrique sur $K(\mu_1, \mu_2, ..., \mu_{i-1})$, on peut se ramener au cas précédent en calculant un *élément primitif* μ de l'extension $K(\mu_1, \mu_2, ..., \mu_m)$ qui est alors égale à $K[\mu]$. La recherche d'un élément primitif peut se faire par des méthodes présentées dans [Lebesgue 50] ou [Yokoyama & al 89]. Cependant, on peut aussi rechercher directement les facteurs de P par une méthode semblable à celle de l'alinéa précédent comme le suggère G. Chen dans [Chen 92] (Note : l'algorithme décrit n'a pas été implanté à ce jour (septembre 92)).

4. Méthode de Lebesgue

Nous exposons dans cette section une méthode issue de celle exposée par H. Lebesgue au cours de leçons données au Collège de France [Lebesgue 50]. La méthode originale est extrêmement fastidieuse et utilise une résolvante de Galois difficile à calculer. Le théorème suivant proposé par G. Chen et démontré avec l'aide du professeur H. Carayol de l'Université Louis Pasteur à Strasbourg [Chen 92], la simplifie considérablement.

Théorème (Décomposabibilité par radicaux). Soit K un corps de caractéristique nulle. Si une équation polynomiale $P(X) = 0$, où P est un polynôme de $K[X]$, est irréductible et résoluble par radicaux carrés sur K, alors il existe un élément r de K tel que P soit réductible dans $K()$.

La preuve de ce théorème utilise la théorie de Galois et exploite les particularités des 2-groupes obtenus. Nous ne donnons pas ici cette démonstration qui peut être trouvée dans [Chen 92].
□

Notons tout d'abord que le polynôme P est nécessairement de degré 2^n avec $n \geq 1$. L'objectif est donc de découvrir des réels $r_1, r_2, ... r_k$ de sorte que le polynôme P dont on cherche les zéros, soit scindé sur $K(, ...)$ ceci pouvant se faire de manière progressive si le polynôme est résoluble par radicaux carrés d'après le théorème précédent. La méthode utilisée pour cette décomposition est celle décrite par H. Lebesgue dans [Lebesgue 50]. On cherche en premier lieu une extension quadratique K_1 de K telle que P, irréductible sur K, soit décomposable sur K_1. Puis on recommence l'opération sur chacun des facteurs de degré supérieur à 1.

On essaye donc de trouver un élément r de K de sorte que P soit décomposable dans $K()$. Pour cela, on tente tout d'abord d'avoir un facteur de degré 1 : dans ce cas, il doit exister deux réels a et r tels que $P(a +) = 0$. Or $P(a +)$ s'écrit sous la forme $A + B$où A et B sont deux polynômes à coefficients dans K en a et r. Comme n'est pas dans K, on doit donc avoir

$A = 0$
$B = 0$

Ce système, que l'on doit résoudre dans K, est un sujet idéal pour la méthode de Ritt Wu.

Si l'on ne trouve aucune solution dans K, il nous faut essayer un facteur de degré 2. Un tel diviseur de $P(X)$ peut s'écrire sous la forme

$$X^2 + (m + a)X + n + b = X^2 + mX + n +$$

En égalant à 0, le reste de la division de $P(X)$ par ce polynôme, on a 4 équations algébriques à coefficients dans K :

$$A_1(m, n, a, b, r) = 0$$
$$A_2(m, n, a, b, r) = 0$$
$$A_3(m, n, a, b, r) = 0$$
$$A_4(m, n, a, b, r) = 0$$

Comme on aurait pu passer l'un des coefficients a ou b sous le radical, on peut imposer la condition supplémentaire :

$$(a - 1)(b - 1) = 0$$

La recherche des solutions de ce système de 5 équations à 5 inconues dans K se fait en utilisant la méthode de Ritt-Wu.

On procéde de la même manière pour un facteur de degré quelconque. Si, ayant examiné tous les degrés possibles jusqu'à 2^{n-1}, on n'a découvert aucun facteur de la forme requise, cela signifie, en vertu du théorème précédent, que le polynôme ne possède pas de zéros exprimables à l'aide de radicaux carrés. Dans le cas contraire, on continue la décomposition de chacun des facteurs dans $K()$ et ainsi de suite.

5. Conclusion

Transcrire le problème des constructions géométriques dans le cadre algébrique du plan euclidien a permis de résoudre des questions théoriques importantes en donnant un critère puissant de constructibilité constitué par le théorème 2. En outre, les travaux présentés ici permettent de se faire une idée de la décidabilité des problèmes de construction. Il faut signaler qu'une partie des algorithmes indiqués a été implantée dans un prototype en Maple sur station Sparc de Sun. Quelques exemples simples ont été traités [Chen 92], mais des problèmes de saturation de la mémoire surviennent rapidement.

En outre, résoudre brutalement -pour ne pas dire naïvement - de manière algébrique un exercice de construction peut se révéler d'une complexité désastreuse. Un exemple classique en est la construction d'un pentagone régulier inscrit dans le cercle unité : une algébrisation comme celle présentée en exemple conduit à considérer un système de 8 équations de degré 2 à 8 inconnues, alors que la considération des racines cinquièmes de l'unité fournit un exercice de Bac C résolvable à la main en 30 minutes au plus.

Un autre reproche peut être fait à la résolution algébrique des problèmes de construction, comme en démonstration géométrique automatique par méthode algébrique [Chou 88] : c'est l'absence de signification géométrique des étapes de calcul et le problème de l'interprétation des résultats [Chou 88]. Le lecteur a pu s'en rendre compte avec la construction effective d'un triangle équilatéral. Dans certains domaines comme celui de l'Enseignement Assisté par Ordinateur, les méthodes algébriques du type de celle présentée ici semblent donc à proscrire.

Point de vue algébrique

Notre objectif étant de produire des programmes de construction géométrique "naturels", nous nous intéresserons à l'automatisation des méthodes de constructions par voie purement géométrique.

Chapitre 3

Présentation informelle de Progé

Ils tremblaient à l'idée de revenir bredouilles,
Et messire Castor, enfin intéressé,
Se mit à sautiller sur le bout de sa queue,
Car le couchant déjà était couleur de rouille.

1.Présentation générale

Progé est un système résolvant des problèmes de construction géométrique de manière formelle puis numérique. A partir d'un énoncé, Progé produit un programme de construction géométrique qui peut, ensuite, être interprété pour donner lieu à des figures numériques, ensembles d'objets géométriques associés à leur coordonnées exprimées de manière numérique, et des figures graphiques, ensembles de pixels colorés .

Du point de vue de son architecture, Progé est un système à base de connaissances composée d'un univers géométrique et d'un ensemble de règles. La figure 1 propose un aperçu de l'architecture de Progé : les cadres rectangulaires représentent des structures de données tandis que les cadres aux coins arrondis figurent le code du programme proprement dit.

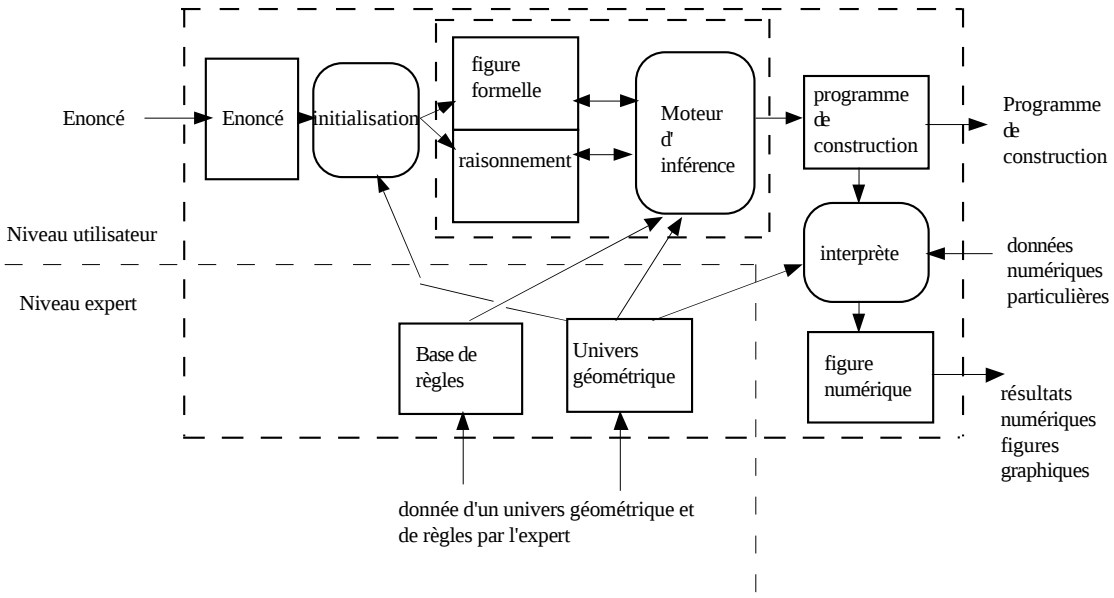

Figure 1 : Schéma général simplifié de Progé

A travers le moteur d'inférence, les règles de Progé utilisent et agissent sur une base de faits issue à l'origine d'un énoncé de problème, que nous appelons *configuration*, dans laquelle nous distinguons une *figure formelle* et un historique des déductions, que nous appelons un *raisonnement formel*.

Progé peut être décrit à deux niveaux séparés par une ligne pointillée à la figure 1 :
- un niveau expert où l'on met au point un *univers géométrique* et une *base de règles* dans lequel est précisée la syntaxe des termes employés dans les règles ;
- un niveau utilisateur où un problème décrit par un *énoncé* est résolu sous forme d'un *programme de construction* et/ou des *figures numériques* et *graphiques*.

Ces deux niveaux sont présentés dans les sections 2 et 3. A la section 4 nous précisons les notions de figure formelle et de raisonnement. La section 5 expose brièvement les opérations fondamentales comme la sélection et l'application des règles, l'unification et l'atomisation.

2. Progé du point de vue de l'expert

Comme on peut le voir à la figure 1, un expert géomètre peut modifier les capacités de Progé, non seulement en faisant évoluer la base de règle, mais aussi en changeant l'univers géométrique.

2.1. Description et modification de l'univers géométrique de base

L'univers géométrique utilisé par Progé résulte d'une tentative de modélisation des entités de base de la géométrie élémentaire. Nous avons codifié et attaché certains renseignements aux sortes géométriques, symboles fonctionnels et symboles relationnels reconnus de Progé. Cet univers est décrit par une théorie logique au sens habituel. Cette théorie logique est manipulé par Progé lors de la résolution d'un problème de construction. Ainsi, dans Progé, il ne suffit pas de signaler quels sont les symboles fonctionnels et relationnels utilisés : il faut aussi indiquer, par exemple, comment on peut permuter leurs arguments pour avoir des termes équivalents et quels sont les liaisons existant entre les relations d'incidence et certains symboles fonctionnels. Ceci est détaillé au chapitre 4.

En outre, pour pouvoir produire une sortie graphique des constructions trouvées, notre univers géométrique de base intègre une interprétation, définie par l'expert, dans un cadre algébrique. Ainsi, nous avons assigné :
- à chaque sorte géométrique, une représentation algébrique et des traitements particuliers comme la manière de saisir ou de dessiner un objet de cette sorte ;
- à chaque symbole fonctionnel non constant, une fonction d'évaluation qui est la représentation algébrique de celui-ci ;
- à chaque symbole relationnel, une fonction booléenne.
Cette interprétation est complétée lors de l'évaluation d'un programme de construction par l'attribuation de valeurs numériques aux constantes.

Les modifications de l'univers géométriques sont possibles et même facilement réalisables. Cependant, pour l'instant, il n'existe pas d'interface conviviale ou assurant un certain contrôle sur ces remaniements : il faut éditer les fichiers concernés et ajouter ou retrancher les faits qui

codent les attributs, et donc une bonne connaissance de la syntaxe interne de Progé est nécessaire.

2.2. Forme de règles et modification de la base de règles

Pour faire progresser une construction, Progé s'appuie essentiellement sur des règles. Ces règles peuvent être des *règles simples* ou des *règles disjonctives* (Cf. plus bas et chapitre 8). La syntaxe d'une règle simple est la suivante :

<nom> # `si` <liste de faits> `et` <liste de vérifications> `alors` <liste de faits>

Exemple.
Dans la base de règles couramment utilisée par Progé, nous avons la règle :

```
1 # si [dist(A, B) '=l=' K ] et [connu A, connu K, pas_connu B]
   alors [B est_sur ccr(A, K) : 1].
```

Cette règle exprime que si la distance `K` entre deux points `A` et `B` est connue, i.e. soit donnée par l'énoncé, soit construite à ce moment de la résolution, que le point `A` est connu, i.e. construit, et que le point `B` n'est pas encore construit, alors `B` appartient au cercle -construit- de centre `A` et de rayon la distance de `A` à `B` et l'on peut restreindre de 1 son degré de liberté. Une telle règle est qualifiée d'*active* par M. Buthion [Buthion 75] car elle participe directement à la détermination d'un élément de la figure. A cet effet, nous avons introduit la liste des vérifications pour que la conclusion soit effectivement déterminante.

La syntaxe indiquée ainsi que l'exemple donné sont équivalents à des clauses de Horn. En fait, nous avons besoin d'un autre type de règles pour résoudre complètement des problèmes de construction.

Exemple.
Si dans une construction nous avons à considérer le lieu `L` des points équidistant à deux droites `d1` et `d2`, il nous faudra tenir compte des cas suivants : soit `d1` et `d2` sont concourantes, alors `L` est la réunion des deux bissectrices de `d1` et `d2`, soit `d1` et `d2` sont strictement parallèles, alors `L` est alors la droite parallèle et à égale distance de `d1` et `d2`, soit `d1` et `d2` sont confondues, et alors `L` est égal au plan. La distinction entre ces cas ne peut se faire en général que lorsque `d1` et `d2` ont reçu une valeur numérique : on ne peut pas décider d'un cas de figure lors de la phase formelle car tous ceux-ci sont également envisageables.

Nous avons donc introduit des règles disjonctives ou à conclusions multiples de syntaxe :

```
<nom> # si <liste de faits> et <liste de vérifications>
        alors soit <liste de faits> et <liste de faits>
              ou soit <liste de faits> et <liste de faits>
              ...
              ou soit <liste de faits> et <liste de faits>
```

Chaque liste de faits aprés le mot réservé `soit` est une liste de *prémisses secondaires* servant à départager les différents cas qui doivent s'exclurent mutuellement. Chaque liste de la partie conclusion et aprés le mot réservé `et` correspond à la conclusion de la règle dans le cas de figure précisé par les prémises secondaires. Si Progé peut prouver formellement, en examinant les prémisses secondaires, qu'un seul cas de figure est possible, alors seul celui-ci est envisagé. Sinon, les cas de figure possibles apparaissent dans le programme de construction

sous forme de structures conditionnelles où les prémisses secondaires seront testées numériquement au moment de l'interprétation. Nous détaillons les implications de ces *règles disjonctives* au chapitre 8.

Exemple.
La règle "des bissectrices" évoquée plus haut est ainsi codée :

```
7 #  si [did(A, D1) '=l=' did(A, D2)]
     et[differents [D1, D2],connu D1, connu D2, pas_connu A]
     alors
        soit [dird(D1) diff dird(D2)] et [A est_sur bis(D1, D2) : 1]
        ou
        soit [dird(D1) eg dird(D2), D1 diff D2] et [A est_sur dmd(D1, D2) : 1]
        ou
        soit [D1 eg D2] et [].
```

On peut soit modifier la base de règles standard (dans le fichier `regles`) soit utiliser une autre base de règles. La modification de la base de règles se fait en éditant celle-ci avec un éditeur de texte standard en suivant la syntaxe décrite précédemment. Pour que cette nouvelle base soit prise en compte, il suffit de la charger dans Progé. Comme le lecteur peut s'en apercevoir, nous n'avons pas prévu d'interface sophistiquée en ce qui concerne la base de règles : il serait peut-être souhaitable d'en étudier une vérifiant les erreurs de syntaxe, les redondances, les incohérences éventuelles, le fractionnement ou les regroupements possibles (Cf. [Wilson 88], [Pintado 91]).

3. Niveau utilisateur

Pour un utilisateur, il s'agit de résoudre des problèmes de construction tels que ceux que nous avons présentés à la section 2 du chapitre 1, ou tels que le suivant [Buthion 79], qui va nous servir de guide pour présenter l'utilisation de Progé.

Enoncé. *Etant donnés deux points* a *et* b, *et un cercle* c, *trouver une droite* d *passant par* a *et coupant* c *en deux points distincts* e *et* f *équidistants de* b.

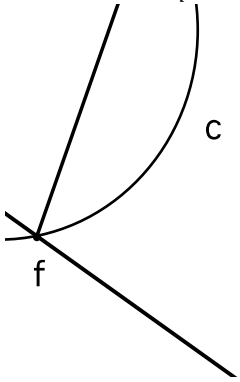

Figure 2 : Tracé correspondant à l'énoncé de la section 3.

Les points a et b et le cercle c étant des paramètres non précisés, il est nécessaire de trouver une solution formelle où ces paramètres ne sont pas évalués. Puis éventuellement, on peut tracer une figure ou plusieurs figures illustrant la construction en donnant des valeurs à a, b et c.

Ceci se traduit par la distinction de deux phase dans le fonctionnement de Progé :
- dans une première phase, Progé produit à partir d'un énoncé entré par l'utilisateur, un programme de construction géométrique.
- dans une deuxième phase, Progé permet d'exploiter le programme de construction. Un interprète permet de l'exécuter sur des jeux de données fournies par l'utilisateur et d'obtenir les figures numériques correspondantes. Un traceur permet d'en déduire des figures graphiques.

Le traitement de l'énoncé donné en exemple va nous servir à introduire les différents éléments qui interviennent dans notre approche des notions d'énoncé et de programme de construction. Pour bien préciser ce que Progé va produire, rappelons que l'on fournit souvent comme solution à un tel problème un plan de construction "brut". Pour l'énoncé ci-dessus, on aurait par exemple :

Tracer la droite d' *joignant* b *au centre du cercle* c, *la droite* d *est la perpendiculaire à* d' *passant par* a.

Ce plan de construction, souvent admis comme convenable, suscite cependant bien des questions. On peut, d'abord, se demander :
- si la droite ainsi définie répond à la question (et pourquoi) ;
- si elle est la seule à y répondre ;
- si elle répond à l'énoncé quelles que soient les positions respectives de a, b et c ;
- si l'on souscrit bien à la condition de n'utiliser que la règle et le compas : comment obtenir le centre du cercle c à la règle et au compas, comment tracer une perpendiculaire à une droite passant par un point donné, à la règle et au compas ?

Dans notre prototype Progé, les réponses à ces questions correspondant à ce que l'on appelle souvent *discussion*, sont parties intégrantes du programme de construction comme nous allons le voir maintenant. En Progé, l'énoncé donné en exemple s'écrit :

```
0 'dec:' point :: a donne.
1 'dec:' point :: b donne.
2 'dec:' cercle :: c donne.
3 'dec:' point :: e mentionne.
4 'dec:' point :: f mentionne.
5 'dec:' droite :: d cherche.

0 'cont:' a est_sur d.
1 'cont:' e est_sur d.
2 'cont:' f est_sur d.
3 'cont:' e est_sur c.
4 'cont:' f est_sur c.
5 'cont:' dist(b,e) '=l=' dist(b,f).
```

Dans cet énoncé, `'dec:'` indique une déclaration d'objet géométrique et `'cont:'` une contrainte entre des objets déclarés plus haut. Les numéros présents avant `'dec:'` ou `'cont:'` n'ont d'utilité que dans la saisie d'un énoncé à l'aide d'un éditeur interne. Le motif d'une déclaration est `donne` pour un objet donné, `cherche` pour un objet cherché et `mentionne` pour un objet auxiliaire de l'énoncé, comme `e` ou `f`. Le symbole `est_sur` désigne une relation d'incidence et le symbole `'=l='` impose l'égalité entre les deux distances `dist(b, e)` et `dist(b, f)`. De telles égalités typées sont utilisées par Progé afin d'améliorer l'efficacité.

Après résolution formelle, Progé propose le programme de construction suivant :

```
a := donne
b := donne
c := donne
point00 := centre(c)
long00 := rayon(c)
cercle00 := cdiam(b,a)                          % cercle de diamètre (b, a)
point02 := centre(cercle00)
long02 := rayon(cercle00)
long03 := dist(a,b)
si [a diff b] alors
 droite01 := dro(a,b)
 dir02 := dird(droite01)                 % direction de la droite droite01
 long04 := dist(point00,b)
 si [point00 diff b] alors               % b n'est pas le centre de c
       droite00 := dro(point00,b)              % droite passant par point00 et b
       dir01 := dird(droite00)
       list00 := intercd(cercle00,droite00) % intersection cercle droite
       pour point01 dans list00 faire    % point01 est le milieu de (e, f)
             long05 := dist(point01,a)
             si [point01 diff a] alors         %i.e. a est sur la droite (b, point00)
                   d := dro(point01,a)
                   e := intercd(c,d)
                   f := intercd(c,d)
                   fin
             sinon                             % a n'est pas sur dro(b, point00)
                   dir04 := diro(droite00)  % direction orthogonale
                   d := dpd(a, dir04)
                   e := intercd(c,d)
                   f := intercd(c,d)
                   fin
             fin
       fin
 sinon                                         % b est égal au centre de c
       echec                                   % toute droite passant par a et
       fin                                     % coupant c convient
 fin
sinon                                          % a = b
 echec                                         % cf. remarque plus bas
 fin
verifier(e est_sur c)                          % vérifications pour la solution
verifier(f est_sur c)                          % numérique obtenue
verifier(e est_sur d)
verifier(f est_sur d)
verifier(a est_sur d)
verifier(dist(e,b) =l= dist(f,b))
```

```
fin
```

Le texte en caractères "courier" correspond à ce qui est affiché par Progé, les commentaires, en caractères "times" et introduits par %, ont été ajoutés à la main. Ce programme comporte des structures de contrôle courantes, comme des instructions conditionnelles ou des itérations, de signification intuitive, mais dont les implications seront expliquées au chapitre 5. Il comporte aussi des primitives de l'univers choisi dont la signification, si elle n'est pas évidente, est indiquée en commentaires.

Ce programme fournit toujours une construction de la droite d dans le cas où tous les points sont distincts, ce qui est satisfaisant. Dans le cas dégénéré où le point a est égal au point b, la droite perpendiculaire à la droite dro(b,centre(c)) passant par a est encore solution, du moins lorsque a est à l'intérieur du cercle c. Cette construction n'est pas trouvée par Progé : l'instruction echec dans un programme de construction ne signifie pas que la construction n'est pas possible, mais, plus prosaïquement, que Progé ne sait pas faire cette construction.

Le programme de construction géométrique répondant à l'énoncé est formé conjointement à l'élaboration de la figure formelle par ajout de faits dans la configuration. Ces faits sont récupérés à la fin de la phase de construction formelle pour donner le programme de construction proprement dit.

Ce programme est ensuite interprété numériquement en donnant des valeurs à a, b et c. Si ces valeurs correspondent au cas général, Progé fournit alors un dessin comme à la figure 2.

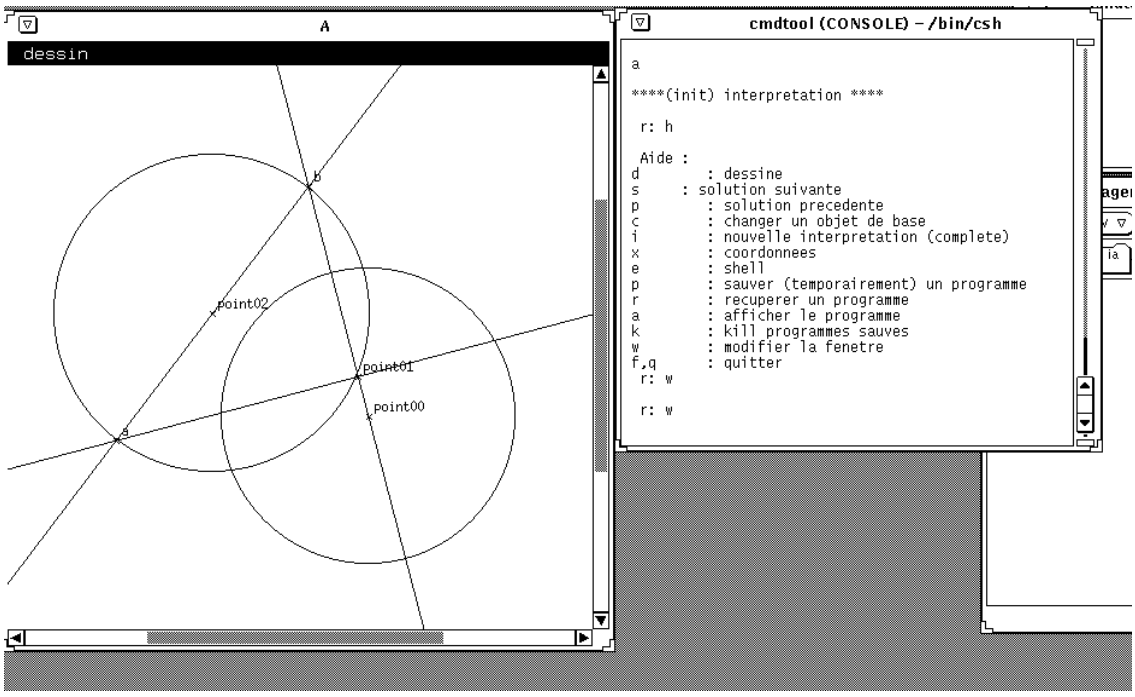

Figure 2 : copie d'écran correspondant à une solution de Progé à l'énoncé 1

Progé propose aussi quelques outils d'exploration d'une figure numérique et le moyen de réinterpréter le programme de construction en changeant une ou toutes les valeurs des objets donnés par l'énoncé.

4. Description de la figure et du raisonnement formels

Progé travaille en chaînage avant : les faits découverts tant à l'aide des règles que par l'exploitation de l'univers géométrique sont ajoutés à la base de faits. Les faits découverts par Progé sont de différentes espèces : il peut s'agir de l'égalité entre deux termes, d'une relation "constructive", c'est-à-dire, intuitivement, qui "fait avancer la construction", ou d'une relation qui n'est ni une égalité, ni une relation constructive. La base de faits est séparée en deux structures pour mémoriser ces différents types de faits : une figure formelle thésaurise les égalités et les relations constructives, et un raisonnement formel mémorise tous les faits, constructifs ou non, découverts à l'aide des règles. Cette section présente succinctement ces deux concepts importants qui seront décrits plus précisément au chapitre 6.

4.1. Figure formelle

Intuitivement, la *figure formelle* est la modélisation de la figure graphique que pourrait faire quelqu'un résolvant à la main un problème de construction. A travers la notion d'objet géométrique, on y consigne la *définition* de tous les objets qui apparaissent dans l'énoncé et au cours de la résolution : milieux, intersections utiles, médiatrices, bissectrices, etc. On y note les relations d'incidence et l'état d'avancement de la construction des différents objets notamment leur *degré de liberté* qui évolue lors de la résolution.

Plus précisément, une figure formelle peut être vue comme étant composée d'un graphe d'incidence et d'un hypergraphe de dépendance fonctionnelle. Dans le *graphe d'incidence*, les sommets sont les objets géométriques concernés par la construction et les arêtes les relations d'incidence entre les objets. Nous traduisons l'existence d'une arête (o_1, o_2) étiquetée par un symbole relationnel `p` dans le graphe d'incidence, en disant que, dans la figure en cours, o_1 est un *participant* de o_2 au titre de la relation `p`.

Exemple.
L'énoncé proposé plus haut est traduit en une figure formelle initiale qu'on peut représenter graphiquement comme aux figures 4 et 5. La figure 4 représente le graphe des relations d'incidence existant entre les objets géométriques après traduction de l'énoncé. Les cercles représentent les objets géométriques dont le nom est inscrit à l'intérieur et les flèches les relations d'incidence. On y voit, par exemple, que `e` est un participant de `d` au titre de la relation `incidentpd`, `d` est un participant de `a` au titre de la relation `incidentdp`, ou dit autrement que `a` est sur `d`, `d` passe par `a`, `e` est sur `d` et `d` passe par `e`.

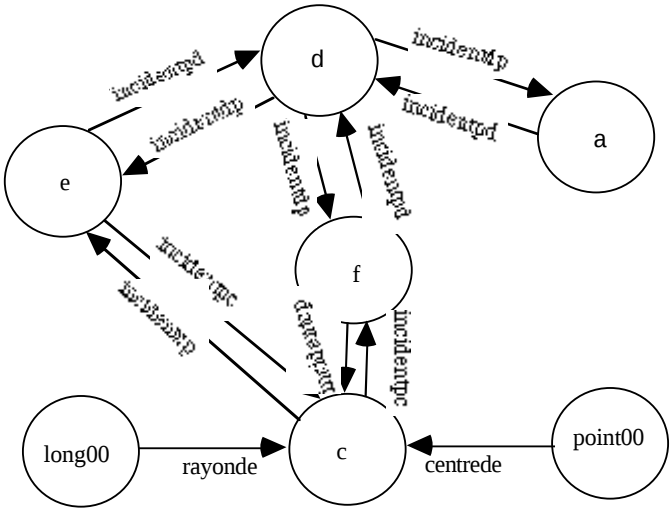

figure 4 : graphe d'incidence correspondant à l'énoncé.

Dans l'*hypergraphe de dépendance fonctionnelle*, les sommets sont encore les objets géométriques et les hyperarêtes les dépendances fonctionnelles entre ces objets. Une hyperarête ordonnée $\{o_1, o_2, \ldots, o_n\}$ étiquetée par un symbole fonctionnel `f` de l'hypergraphe fonctionnel représente l'égalité $o_1 = f(o_2, \ldots o_n)$. Nous disons que $f(o_2, \ldots, o_n)$ est un *représentant* de o_1 dans cette figure.

Exemple.
La figure 5 représente l'hypergraphe des dépendances fonctionnelles correspondant à l'énoncé de la section 3. Les hyper-arêtes sont ordonnées et etiquetées par les noms de symboles fonctionnels à l'intérieur des rectangles. Nous y voyons ainsi que `dist(e, b)` et `dist(f, b)` sont des représentants de `long00`.

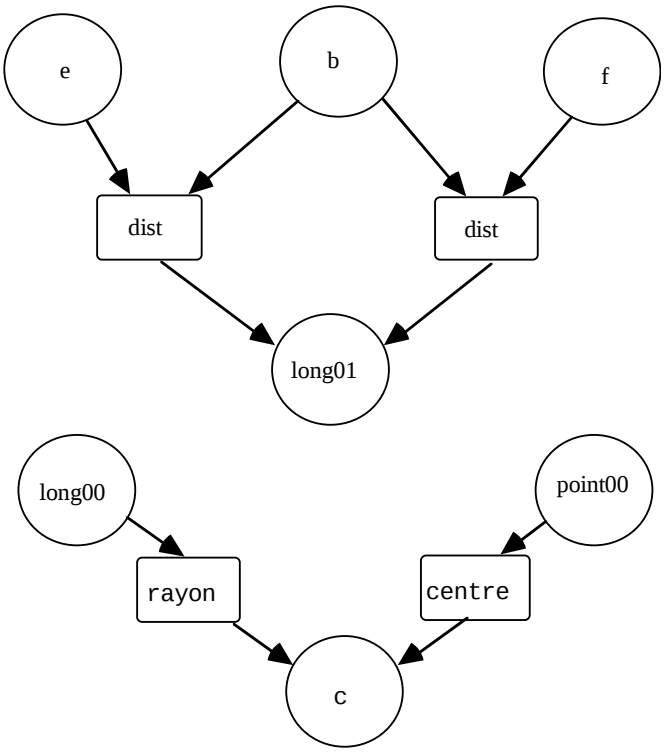

Figure 5 : hypergraphe des dépendances fonctionnelles.

On notera que les symboles de point et de longueur `point00`, `long00` et `long01` ont été engendrés automatiquement par Progé.

4.2. Raisonnement formel

Intuitivement, le *raisonnement* consigne à chaque instant l'historique des règles appliquées et des propriétés déduites de la figure et des propriétés précédentes.

Plus précisément, il peut être vu comme un *hypergraphe de raisonnement* où les sommets sont les propriétés déduites par Progé et où les hyperarêtes sont les occurrences des règles ayant permis la déduction.

Exemple.
L'énoncé donné plus haut fournit, après résolution formelle dans le cas général, le raisonnement donné à la figure 6. Les rectangles représentent les sommets de l'hypergraphe et contiennent les propriétés déduites par Progé. Dans ce schéma, qui correspond à un cas simple, on n'a que des arêtes. Elles sont étiquetées par deux entiers. Le premier correspond au numéro de l'inférence et le deuxième au numéro de la règle appliquée (le lecteur peut ainsi suivre la chronologie des événements). Les trois sommets en haut du schéma correspondent aux contraintes fournies par l'énoncé. Nous n'avons pas représenté celles non utilisées (`e est_sur d` et `f est_sur d`). On peut y lire par exemple que lors de l'inférence n° 2, l'application de la règle n° 105 à la propriété `iso(b,e,f)` a engendré deux nouvelles propriétés : `droite00 ortho d` et `droite00 = dro(b, point01)= med(e,f)`. Les autres propriétés de `droite00, droite00 = dro(point00, point01) = med(e, f)` ont été engendrées par l'inférence n° 11 avec la règle n° 105.

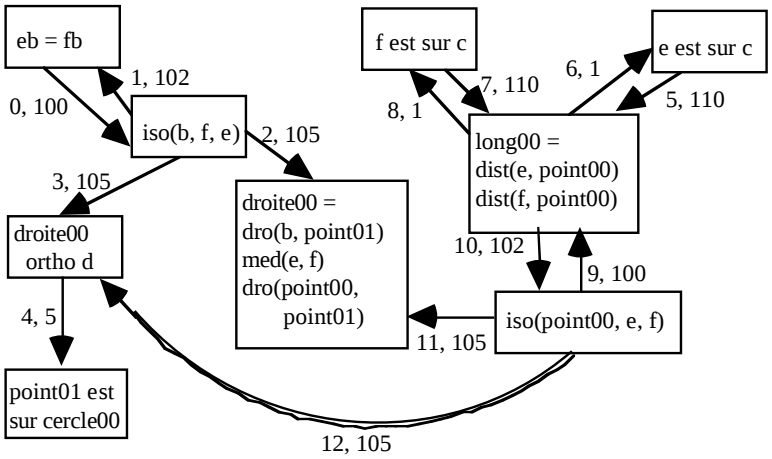

Figure 6 : Raisonnement du cas général de l'énoncé 1.

La configuration initiale, c'est-à-dire la figure formelle et le raisonnement initiaux, résultent de la traduction de l'énoncé par un module d'initialisation inclus dans Progé. A l'heure actuelle, il n'y a qu'une vérification sommaire de la cohérence d'un énoncé. Aussi, l'une des conditions de réussite de cette initialisation est que l'énoncé soit correctement posé.

5. Gestion d'une configuration.

Une configuration évolue à partir d'un état initial obtenu après lecture d'un énoncé jusqu'à un état final où tous les objets cherchés sont déterminés. Cette évolution se fait grâce à l'application des règles et de mécanismes de gestion de la figure courante comme la propagation des degrés de liberté ou l'application d'axiomes "automatiques" de l'univers géométrique.

5.1. Filtrage

Classiquement, lors de l'opération de *filtrage*, i.e. la recherche des règles applicables, se fait en comparant les prémisses de chaque règle avec les propriétés déjà obtenues et mémorisées dans le raisonnement. Suit une étape de résolution des conflits où des heuristiques sont mises en oeuvre pour n'appliquer que la ou les règles jugées les plus intéressantes.

Dans Progé, nous n'avons pas recherché des heuristiques originales : il y a deux stratégies possibles. L'une consiste à essayer d'appliquer systématiquement les règles dans l'ordre où elles sont écrites, l'autre consiste à exploiter les derniers faits trouvés pour chercher une règle applicable. Dans les deux cas, la première règle applicable trouvée est appliquée.

Rappelons qu'une règle normale est de la forme :

`<nom> # si <liste de faits> et <liste de vérifications> alors <liste de faits>`

La recherche d'une règle applicable se fait d'une part en essayant d'unifier la première liste de faits à l'aide des faits contenus dans le raisonnement, d'autre part en vérifiant sur la figure des informations -énumérées par la deuxième liste- comme le degré de liberté des objets dans la figure courante ou la différence syntaxique de deux objets.

La figure ne sert pas seulement à fournir les informations demandées par la liste de vérification, elle est également utilisée au moment de l'unification. En effet, comme l'a montré M. Buthion [Buthion 79], l'unification classique est insuffisante dans le cas de la géométrie. Aussi avons-nous mis en oeuvre une méthode d'unification particulière, `unif`, que nous appelons *unification modulo une figure* :

```
unif : figure terme terme -> substitution*
```

Alors que l'unification classique entre un nom d'objet `o` et un terme $f(t_1, t_2, ..., t_n)$ échoue, l'unification modulo une figure réussit, c'est-à-dire `unif` rend une liste non vide de substitutions, si on peut trouver dans cette figure un représentant de `o` de la forme $f(a_1, a_2, ..., a_n)$ tel que pour i = 1, 2, ..., n, t_i s'unifie à a_i modulo la figure. La substitution qui résulte de l'unification entre $f(t_1, t_2, ..., t_n)$ et $f(a_1, a_2, ..., a_n)$, si elle réussit, est ajoutée à la liste des substitutions rendue par `unif`. Si le symbole fonctionnel est décomposable, le représentant est reconstitué à partir des participants.

Exemple.
Considérons la règle

```
110 # si [ M est_sur ccr(O, R)] et [ ]
        alors [dist(O, M) '=l=' R ]
```

qui signifie : si `M` est sur le cercle de centre `O` et de rayon `R`, alors la distance de `O` à `M` est égale à `R`. Si nous voulons unifier cette règle avec la propriété

```
e est_sur c
```

l'unification classique de `c` avec `ccr(O, R)` échoue. Dans l'unification modulo une figure, on recherche, ou on reconstruit à l'aide des participants, un représentant adéquat de manière que l'unification puisse continuer. Ainsi, `c` sera ici remplacé par le terme `ccr(point00,long00)`, par exemple, et les variables `O` et `R` et `unif` rend la liste réduite à une substitution [{`O` = `point00`, `R` = `long00`}]. Bien entendu, si un tel représentant n'existe pas, l'unification échoue et la fonction `unif` rend une liste de substitutions vide.

5.2. Application d'une règle

Lors de l'application d'une règle, les faits en conclusions -qui sont alors instanciés par des termes clos- servent à compléter la figure et le raisonnement.

Avant d'être mémorisé dans le raisonnement, un fait est *atomisé* : c'est à dire qu'on le remplace par un terme de profondeur 1 construit avec un symbole relationnel, ou encore tous les termes fonctionnels contenus dans le fait sont remplacés par des constantes.

Exemple.
Ainsi si le fait

```
m est_sur med(mil(a, b), c)
```

a été dérivé, on le mémorisera sous la forme

```
m est_sur droite02
```

par exemple.

Ce mécanisme d'atomisation de profil

```
atomise : figure terme -> figure nom
```

est un des mécanismes essentiels dans la gestion d'une figure. Par le biais de cette opération, Progé contrôle d'éventuelles redondances d'objets géométriques et officialise de nouveaux objets en leur donnant une existence dans la figure. L'atomisation est utilisée par ailleurs pour assurer que les représentants d'objets géométriques sont bien des termes clos de profondeur 1. On cherche, lors de l'atomisation d'un terme `t` dans une figure `Fig`, à remplacer `t` par un nom d'objet géométrique soit existant dans `Fig`, si `t` correspond à une hyperarête de `Fig`, soit engendré par Progé, si `t` désigne un objet géométrique nouveau. Dans le deuxième cas la figure est mise à jour en ajoutant ce nouvel objet ayant `t` comme représentant.

Exemple.
Avec l'exemple précédent, il faut atomiser `med(mil(a, b), c)`, médiatrice du milieu de `[a, b]` et de `c`. Progé va nommer les termes concernés en leur attribuant éventuellement une existence dans la figure. Supposons qu'il existe dans la figure courante un objet géométrique de nom `i` dont un représentant est `mil(a, b)`. Ce terme est alors remplacé par `i` et l'atomisation se poursuit avec `med(i, c)`. Supposons maintenant qu'il n'existe aucun objet géométrique dans la figure dont `med(i, c)` soit un représentant. Progé crée un nom de droite non encore utilisé dans la figure, par exemple `droite02`, et complète la figure courante en ajoutant l'hyper-arête correspondant à l'égalité `droite02 = med(i, c)`. Le degré de liberté et une définition de `droite02` sont calculés en conséquence. Finalement, la fonction `atomise` retourne la figure ainsi complétée et le nom `droite02`.

Mais la mise à jour du raisonnement n'est pas suffisante. Il faut également faire progresser la figure en tenant compte de toutes les propriétés constructives déduites à l'aide de la règle appliquée. Ces propriétés constructives concernent les égalités et les relations d'incidence :

-pour une relation d'incidence de la forme `r(a, o)`, si `a` est un objet construit dans la figure et pas `o`, alors le participant `(a, r)` est ajouté à la définition partielle de `o` (transformant éventuellement celle-ci en définition complète) ;

- pour une relation d'égalité entre deux objets `o1` et `o2`, on réalise la fusion des deux objets, i.e. on met en commun tous les représentants et les participants des deux objets, on calcule une nouvelle définition et un nouveau degré de liberté pour les deux objets et on les identifie dans le reste de la configuration à travers l'utilisation d'une *table de synonymes*.

Si lors de l'application d'une règle, un objet est construit, c'est-à-dire si le nouveau degré de liberté attribué à cet objet dans la figure vaut 0, alors on propage classiquement cette information à travers le graphe et l'hypergraphe qui constituent la figure.

Aprés cet exposé rapide des grandes fonctions et caractéristiques de Progé, nous en détaillons les différents aspects.

Chapitre 4

Spécification d'un univers géométrique

L'homme à la cloche dit : « C'est lui, Machinchouette !
Comme un dément, il crie, oh ! écoutez-le donc !
Il agite les mains, il balance la tête :
Sans nul doute il aura découvert quelque Snark ! »

Ce chapitre aborde un élément essentiel de notre travail. Il s'agit de la description des entités de base manipulées par notre système expert. Ce chapitre va permettre de définir clairement la forme de la base de faits et la syntaxe des règles utilisées par Progé.

La section 1 donne un aperçu du formalisme que nous allons utiliser pour décrire la sorte univers géométrique. La section 2 précise quelques notations concernant les spécifications algébriques. La section 3 fournit un exemple d'univers géométrique et son utilisation pour un exercice de construction simple. La section 4 "passe au niveau méta" en décrivant les entités de base et les propriétés que nous en avons retenu. La section 5 décrit les notions bien connues de substitutions et d'unification à l'aide de spécifications algébriques. Des exemples d'utilisation des propriétés d'un univers géométrique sont donnés. La section 6 montre comment le passage au niveau méta peut s'exprimer à l'aide de spécifications algébriques.

1. Introduction

Dans la résolution d'un problème de construction géométrique, on peut distinguer deux catégories de connaissances :
- celles générales qui décrivent les types, les relations, les définitions et les théorèmes du cadre géométrique ;
- et celles particulières à l'énoncé du problème comme le nommage d'objet et les relations imposées entre les objets nommés.

Les premières sont regroupées dans ce que nous avons appelé un univers géométrique *de base* et les secondes dans un énoncé. La traduction de l'énoncé permet d'avoir un univers géométrique *particularisé* contenant les axiomes propres au problème de construction traité : ceux-ci sont regroupés dans deux structures, une figure formelle et un raisonnement formel que nous avons déjà évoqué et que nous formaliserons au chapitre 6. Nous ne décrivons dans ce chapitre que l'univers géométrique de base et quelques opérations utilisant cette notion comme le filtrage modulo les permutations possibles d'arguments.

Un univers géométrique de base peut être vu comme un système formel en logique du premier ordre typé avec égalité décrit par des spécifications algébriques. Notre objectif d'avoir un système équationnel conditionnel qui garantit l'existence d'une l'algèbre initiale. Mais, dans Progé, l'univers géométrique est une donnée pour le moteur d'inférence. Il doit donc être emmagasiné dans une structure de données et manipulé comme une donnée ordinaire. Aussi pour bien préciser les opérations d'accès et de manipulation, nous tentons dans ce chapitre

d'en faire une spécification algébrique [Wirsing 90]. On a ainsi une spécification de deuxième niveau, une sorte de méta-spécification d'une théorie géométrique décrite elle-même comme une spécification du premier niveau. Actuellement cette "donnée" univers géométrique est figée, mais une des extensions futures sera de pouvoir facilement la mettre à jour à travers une interface expert plus élaborée.

Pour ne pas trop alourdir, nos spécifications algébriques ne seront pas complètement formelles. En particulier, nous exprimerons souvent les préconditions en langage naturel lorsqu'il n'y aura pas d'ambiguïté. Nous savons que notre démarche est dangereuse : les puristes ne reconnaitront pas cette spécification comme telle, les réfractaires se poseront la question de l'utilité d'un tel formalisme. En ce qui concerne les premiers, nous serions enchantés qu'un spécialiste des spécifications algébriques trouve ce travail suffisamment intéressant pour le reprendre proprement. Pour les seconds, nous leur recommandons de les oublier en première lecture et de se reporter aux explications informelles et aux exemples, en espérant qu'ils s'y intéressent en seconde lecture pour vérifier le bien fondé de tout ce que nous racontons.

Remarques.

1) Les commentaires sont écrits à l'intérieur d'une spécification en caractères "times" et sont introduits par le signe %.

2) Dans la suite les variables commencent par une majuscule et sont quantifiées universellement.

3) Les langages de spécification algébrique actuels prévoient des mécanismes sophistiqués de paramétrisation, modularisation et enrichissement [Wirsing 90]. Ces mécanismes ne sont pas utilisés ici. Nous supposons que nous avons une seule spécification que nous complétons tout au long des chapitres suivants.

2. Préliminaires

2.1. Intersections et autres

En géométrie de la règle et du compas, on considère le plan euclidien réel Π muni d'un repère orthonormé (O, I, J). Parmi les sous-ensembles de Π, deux types sont caractérisés et utilisés dans la géométrie de la règle et du compas : il s'agit des droites et des cercles. Nous notons d l'ensemble des droites et c l'ensemble des cercles. Nous avons $d \lceil p(\Pi)$ et $c \lceil p(\Pi)$. Les opérations d'union, d'intersection et de complémentation sont donc envisageables entre cercles et droites bien que, pour nous, l'opération la plus utilisée soit l'intersection. Puisque nous avons distingué des types parmi les sous-ensembles de Π, l'opération $\cap\leftrightarrow$ est polymorphe avec les profils suivant :

$$\_ \cap\leftrightarrow \_ : d \times d \varnothing p(\Pi)$$
$$\_ \cap\leftrightarrow \_ : c \times d \varnothing p(\Pi)$$
$$\_ \cap\leftrightarrow \_ : d \times c \varnothing p(\Pi)$$
$$\_ \cap\leftrightarrow \_ : c \times c \varnothing p(\Pi)$$

L'intersection peut donner un ensemble fini de points ou un ensemble infini. Dans le premier cas, nous dirons qu'il s'agit d'une intersection définissante et dans le second, qu'il s'agit d'un cas dégénéré. En géométrie de la règle et du compas, seules les intersections définissantes sont intéressantes. Notons que les cas dégénérés se produisent ici lorsqu'on fait l'intersection d'un ensemble avec lui-même. Il est donc important de pouvoir dire si deux droites ou deux cercles sont égaux. Si cela semble facile dans chaque cas particulier, cela se complique lorsqu'on agit formellement, nous verrons plus bas comment nous avons tenté de contourner la difficulté dans Progé.

De manière semblable, on considère la fonction `dro` de profil :

$$\texttt{dro} : \Pi \times \Pi \varnothing \mathcal{P}(\mathcal{D})$$

Cette opération associe à deux points `x` et `y` l'ensemble des droites passant par `x` et `y`, c'est-à-dire l'intersection de l'ensemble[4] des droites passant par `x` avec l'ensemble des droites passant par `y`. Comme précédemment, cette intersection est définissante lorsqu'elle contient un nombre fini de droites : c'est le cas où `x` et `y` sont différents et `dro(x, y)` est un singleton ne contenant qu'une seule droite. Le cas où `x=y` définit une infinité de droites, c'est un cas dégénéré.

En ce qui concerne les cercles, la définition du chapitre 1 suggère de considérer la fonction `ccr` de profil :

$$\texttt{ccr} : \Pi \times \Pi \times \Pi \varnothing \mathcal{C}$$

Cette fonction associe à tout triplet de points `(x, y, z)` le cercle de centre `x` et de rayon la distance de `y` à `z`. Le cas normal est celui où les points `y` et `z` sont différents. Lorsque `y=z`, le cercle est réduit à `{x}` et on dit qu'on a un cas dégénéré.

De manière pratique, pour distinguer les points d'une intersection définissante on peut introduire une relation d'ordre sur Π. On pourra ainsi parler du premier point d'intersection entre un cercle `x` et une droite `y`. Dans la suite, nous prendrons l'ordre lexicographique sur les coordonnées dans le repère `(O, I, J)`.

2.2. Spécifications algébriques : types de base

Un de nos objectifs est de modéliser la géométrie des constructions géométriques à la règle et au compas en utilisant le formalisme des spécifications algébriques de type abstrait : la présentation de ce formalisme pourra être trouvée dans [Goguen & al 92], [Ehrig & Mahr 85], [Bergstra & al 89] ou [Wirsing 90]. Dans la suite, une spécification se compose d'une description syntaxique, appelée *signature*, et d'un ensemble d'*axiomes*.

La signature précise les noms des *sortes*[5] et des *fonctions* ou *opérations* définies sur ces sortes. La syntaxe des fonctions est précisée en donnant leur *profil*, qui est composé de deux listes de sortes, l'*arité* et la *coarité* de la fonction, séparées par le signe ∅ :

4on parle alors du faisceau des droites passant par `x`.

5Le mot sorte est utilisé plutôt que le mot type, car ce dernier a trop de connotations. Une sorte est juste un nom symbolique et ne représente rien de plus.

$f : sorte_1\ sorte_2 \ldots sorte_n \varnothing sorte_{n+1}\ sorte_{n+2} \ldots sorte_{n+k}$

L'arité et la coarité d'une fonctions sont simplement notées en séparant les sortes par des espaces. Un profil peut être noté de manière infixe, on emploie alors le signe _ pour indiquer la place des arguments.

Cette syntaxe est complétée en donnant des axiomes censés formaliser le comportement des objets des sortes envisagées. On peut imposer plusieurs formes à cet ensemble d'axiomes. On peut se restreindre à des équations, des équations conditionnels, des clauses de Horn ou des clauses de la logique du premier ordre par exemple [Wirsing 90]. Dans la suite, nous n'envisageons que des *équations conditionnelles* ce qui nous permet de rester en terrain connu.

Afin de préciser les notations utilisées, nous donnons la spécification de quelques types de base.

2.2.1. Booléens.

Dans la spécification suivante (cf. figure 1), la sorte `bool` représente le type booléen. Les quatres fonctions envisagées comprennent les deux définitions de constante : `vrai` et `faux` et les deux opérations classiques de conjonction, notée de manière infixe `_ & _`, et de négation, `non`. Les relations entre ces opérations sont précisées à l'aide des axiomes classiques donnés à la figure 1.

Note : Les variables ne sont ni déclarées, ni typées dans la suite. Typographiquement elles se distinguent par le fait qu'elles commencent par une majuscule. Par ailleurs, elles sont toujours quantifiées universellement.

```
sorte
     bool

fonctions
     vrai, faux : ∅ bool
     _ & _ : bool bool ∅ bool
     non : bool ∅ bool

axiomes
     non(vrai) = faux
     non(faux) = vrai
     vrai & vrai = vrai
     faux & X = faux
     X & faux = faux
```

figure 1 : une spécification des booléens

2.2.2. Entiers

De la même manière, on peut donner la spécification suivantes des entiers naturels pour laquelle on considére l'addition.

```
sorte
     naturel
```

```
fonctions
     0 : ∅ naturel
     s : naturel ∅ naturel
     _ + _ : naturel naturel ∅ naturel

axiomes
     s(X) + Y = s(X + Y)
     0 + X = Y
```

figure 2 : une spécification des entiers naturels

Nous préférerons cependant employer une sorte que nous appellerons `entier` et utilisant les opérations et les notations classiques sur les entiers relatifs, leur mise en oeuvre ne devant pas poser de problèmes.

2.2.3 Listes

Les listes sont en quelque sorte des ensembles ordonnés finis. Nous en donnons la spécification classique à la figure 3 dans laquelle T désigne une sorte quelconque. Dans cette spécification apparaissent des *préconditions* indiquant le domaine de validité des fonctions concernées.

```
sorte
     T*                                   % liste d'éléments de sorte T

fonctions
     [] : ∅ T*                            % liste vide
     [_ | _] : T T* ∅ T*                  % ajout en tête
     tl : T* ∅ T                          % tête de liste
     cl : T* ∅ T*                         % corps de liste
     est_vide : T* ∅ bool                 % test de vacuité
     long : T* ∅ entier                   % nombre d'éléments
     _ # _ : T* entier ∅ T          % nème élément

préconditions
     pré(tl(L)) = non(est_vide(L))
     pré(cl(L)) = non(est_vide(L))
     pré(L # N) = (N > 0) & (N ≤ long(L))

axiomes
     tl([E|C]) = E

     cl([E|C]) = C

     est_vide([]) = vrai
     est_vide([E|C]) = faux

     long([]) = 0
     long([E|C]) = 1 + long(C)

     L # N = si N=1 alors tl(L) sinon cl(L) # (N-1)
```

figure 3 : spécification de listes.

3. Spécification d'un univers géométrique simple

Nous présentons dans cette section un exemple simple de spécification d'univers géométrique. Cette exemple servira de base aux notions plus générales décrites à la section 4. Nous ne sommes évidemment pas les premiers à proposer une spécification d'univers géométrique ([Goguen 89], [Kin & al 89]). Dans [Goguen 89], J. A. Goguen se sert même d'un tel exemple pour montrer l'utilité d'un langage de spécification. Cependant son approche s'apparente à la géométrie analytique et n'entre donc pas dans le cadre que nous nous sommes fixés.

Dans la suite, nous utiliserons des fonctions booléennes plutôt que des prédicats. Ceci engendre quelques lourdeurs, dont nous prions le lecteur de nous excuser, mais permet de rester dans le cadre d'une logique équationnelle.

3.1. Spécification de base

Traduisant la définition du chapitre 1, nous devons avoir les sortes, que nous appelerons les sortes géométriques, et les fonctions de la figure 4. Nous avons ajouté la notion de longueur par rapport à la section 2.1. pour des raisons de commodité.

```
sortes
     longueur
     point
     droite
     cercle

fonctions
     dist : point point ∅ longueur          % distance entre deux points

     O, I : ∅ point                         % points de base
     interdd : droite droite ∅ point  % intersection de deux droites
     interdc : droite cercle ∅ point  % intersection d'une droite et d'un cercle
     intercd : cercle droite ∅ point  % intersection d'un cercle et d'une droite
     intercc : cercle cercle ∅ point  % intersection de deux cercles

     dro : point point ∅ droite       % droite définie par deux points

     ccr : point longueur ∅ cercle          % cercle défini par centre et rayon
```

figure 4 : signature de base pour les constructions à la règle et au compas

Sans même parler d'axiomes, cette formalisation s'éloigne bien sûr de la géométrie décrite à la section 2 : les opérations d'intersection donnent un seul objet de la sorte point (et non pas un ensemble) et l'opération `dro` donne un seul objet de la sorte droite. Il nous faudra de plus tenter de détecter les cas dégénérés. Nous verrons plus loin comment nous avons pu rattraper ces lacunes, tant du point de vue des cas dégénérés particuliers que des intersections multiples, au moyen des notions de symble fonctionnel *ambigu*, comme `intercd`, et de programme de construction. Notons dès à présent, que le fait de ne considérer qu'un point par intersection (avec choix remis au moment de l'interprétation d'un programme de construction) nous interdit de considérer des constructions élémentaires comme le milieu de `[O, I]` par exemple. On peut remédier à cet inconvénient en choisissant de faire retourner une liste de points par `interdc` et `intercc` (ces points étant classés par ordre lexicographique) : le milieu de `[O, I]` pourrait alors s'exprimer par le terme :

```
interdd(dro(intercc(ccr(O,dist(O,I),ccr(I,dist(O,I))#1,
            intercc(ccr(O,dist(O,I),ccr(I,dist(O,I))#2,dro(O,I)).
```

Nous n'avons pas retenu cette manière de faire à cause des difficultés de distinction entre les points d'intersections multiples lors de manipulations formelles, mais c'est une voie à explorer.

3.2. Enrichissement

Dans notre approche, nous avons rendu l'ensemble des symboles fonctionnels facilement enrichissable de manière à pouvoir remédier ponctuellement à cet état de fait. Les symboles fonctionnels décrits précédemment sont les symboles fonctionnels de base pour les constructions à la règle et au compas, tandis que les nouveaux symboles fonctionnels introduits représenteront des constructions élémentaires. Cette manière de faire permet également, d'une part, d'avoir un vocabulaire proche de celui qui est traditionellement employé en géométrie élémentaire et, d'autre part, d'éviter les termes trop compliqués comme celui désignant le milieu d'un bipoint à la section précédente. On peut, par exemple, introduire les symboles fonctionnels de la figure 5.

```
fonctions

      did : point droite ∅ longueur          % distance d'un point à une droite
      rayon : cercle ∅ longueur              % rayon d'un cercle donné
      _ / 2 : longueur ∅ longueur            % division par 2

      mil : point point ∅ point              % milieu d'un bipoint
      prj : point droite ∅ point             % projeté orthogonal
      centre : cercle ∅ point                % centre d'un cercle donné

      dpp : droite point ∅ droite            % parallèle à une droite passant par un point donné
      dortho : droite point ∅ droite         % droite orthogonale à une autre par un point donné
      med : point point ∅ droite             % médiatrice d'un bipoint

      ccp : point point ∅ cercle             % cercle défini par son centre et un point incident
      cdiam : point point ∅ cercle           % cercle défini par un diamètre
      ccir : point point point ∅ cercle      % cercle circonscrit à un triangle
```

figure 5 : enrichissement fonctionnel.

Cet enrichissement ajoute de nouveaux cas de dégénérescence qui sont introduits par les fonctions `med`, `ccp`, `cdiam` et `ccir`. Ces cas dégénérés sont détectables si l'on sait déceler l'égalité entre deux termes. Il nous semble difficile de capter complétement l'égalité de la géométrie élémentaire, néanmoins nous essayons de retrouver les principaux cas au moyen d'axiomes bien choisis. Un premier pas dans ce sens réside dans l'introduction d'axiomes indiquant les permutations possibles des arguments dans les termes utilisés. Dans notre exemple, nous avons les axiomes de la figure 6.

```
dist(X, Y) = dist(Y, X)
interdd(X, Y) = interdd(Y, X)
intercd(X, Y) = interdc(Y, X)
```

```
intercc(X, Y) = intercc(Y, X)
mil(X, Y) = mil(Y, X)
dro(X, Y) = dro(Y, X)
med(X, Y) = med(Y, X)
cdiam(X, Y) = cdiam(Y, X)
ccir(X, Y, Z) = ccir(X, Z, Y)
ccir(X, Y, Z) = ccir(Y, X, Z)
ccir(X, Y, Z) = ccir(Y, Z, X)
ccir(X, Y, Z) = ccir(Z, X, Y)
ccir(X, Y, Z) = ccir(Z, Y, X)
```

figure 6 : axiomes de permutation

Cet ensemble d'axiomes n'est évidemment pas suffisant pour détecter des cas d'égalité même assez simples, en particulier en ce qui concerne les symboles fonctionnels `interdd`, `intercd`, `interdc`, `intercc`, `dro`, `ccp` et `ccir`. Par exemple, avec le symbole fonctionnel `dro`, si les points x_1, x_2 ... x_n sont alignés il serait souhaitable d'avoir les égalités $dro(x_i, x_j) = dro(x_k, x_l)$.

Dans [Buthion 79], M. Buthion a pour ce faire doté le symbole fonctionnel `dro` d'une arité variable, i.e. le seul argument de `dro` est une liste de points. Il nous semble quant à nous que ceci est étroitement lié avec la méthode des lieux et nous proposons de relier le symbole `dro` et les autres symboles fonctionnels qui présentent cette caractéristique comme `interdd`, `ccir` ou `ccp`, avec les relations d'incidence "appartenir à une droite", "appartenir à un cercle" et "passer par un point". Nous qualifions ces symboles fonctionnels de *décomposables*, terme que nous préciserons à la section 4. Nous introduisons donc des symboles de prédicat d'incidence (vu comme des symboles fonctionnels de coarité booléenne). Dans notre exemple, nous aurons les profils de la figure 7.

```
fonctions                    % relations d'incidence
     _ incidentdp _ : droite point ∅ bool
     _ incidentpd _ : point droite ∅ bool
     _ incidentpc _ : point cercle ∅ bool
     _ incidentcp _ : cercle point ∅ bool
     _ centrede _ : point cercle ∅ bool
     _ rayonde _ : longueur cercle ∅ bool

     _ eg _ : longueur longueur ∅ bool
     _ eg _ : point point ∅ bool
     _ eg _ : droite droite ∅ bool
     _ eg _ : cercle cercle ∅ bool
```

figure 7 : Prédicats d'incidence et égalité

Le symbole `eg` est utilisé pour figurer l'égalité. Son intérêt est de conserver des clauses de Horn lorsqu'on a des négations. Les axiomes exprimant le comportement de la relation d'égalité `eg` sont classiquement ceux de la figure 8.

```
X eg X = vrai
X eg Y = Y eg X
```

```
(X eg Y = vrai)  (Y eg z = vrai) ⊃ (X eg z = vrai)
% pour tout symbole fonctionnel f n-aire figurant dans l'univers géométrique :
(X_i eg Y_i = vrai) pour i=1, ... n ⊃ (f(X_1, ..., X_n) eg f(Y_1, ..., Y_n) = vrai)
```

figure 8 : des axiomes pour l'égalité

On lie les symboles fonctionnels décomposables aux relations d'incidence au moyen d'axiomes. Dans notre cas, nous aurons les axiomes de la figure 9.

```
(X incidentpd Z = vrai) ⌉ (Y incidentpd Z = vrai) ⌉ (X eg Y = faux) ⊃ Z = dro(X, Y)
(X incidentdp Z = vrai) ⌉ (Y incidentdp Z = vrai) ⊃ Z = interdd(X, Y)
(X incidentcp Z = vrai) ⌉ (Y incidentdp Z = vrai) ⊃ Z = intercd(X, Y)
(X incidentcp Z= vrai)  (Y incidentcp Z= vrai)  (X eg Y= faux) ⊃ Z = intercc(X, Y)

(X centrede Y = vrai) ⊃ X = centre(Y)
(X rayonde Y = vrai) ⊃ X = rayon(Y)
(X incidentpc Z = vrai) ⌉ (Y centrede Z = vrai) ⌉ (X eg Y = faux) ⊃ Z = ccp(Y, X)
(X1 incidentpc Y = vrai) ⌉ (X2 incidentpc Y = vrai) ⌉ (X3 incidentpc Y)= vrai) ⌉
(X1 eg X2 = faux) ⌉ (X1 eg X3 = faux) ⌉ (X2 eg X3 = faux) ⊃ Y = ccir(X1, X2, X3)
```

figure 9 (début) : relations entre incidence et symbole décomposable

```
                                % réciproques
Z = dro(X, Y) ⊃ (X incidentpd Z = vrai) ⌉ (Y incidentpd Z = vrai) ⌉ (X eg Y = faux)

Z = interdd(X,Y) ⊃ (X incidentdp Z = vrai) ⌉ (Y incidentdp Z = vrai) ⌉
                                                                (X eg Y = faux)
Z = intercd(X, Y) ⊃ (X incidentcp Z = vrai) ⌉ (Y incidentdp Z = vrai)
Z = intercc(X, Y) ⊃ (X incidentcp Z = vrai) ⌉ (Y incidentcp Z = vrai) ⌉
                                                                (X eg Y = faux)

Y = centre(X) ⊃ (Y centrede X = vrai)
Y = rayon(X) ⊃ (Y rayonde X = vrai)
Z = ccp(Y, X) ⊃ (X incidentpc Z = vrai) ⌉ (Y centrede Z = vrai) ⌉ (X eg Y = faux)
Y = ccir(X1, X2, X3) ⊃ (X1 incidentpc Y = vrai) ⌉ (X2 incidentpc Y = vrai) ⌉
          (X3 incidentpc Y) = vrai) ⌉ (X1 eg X2 = faux) ⌉ (X1 eg X3 = faux) ⌉
                                                                (X2 eg X3 = faux)
```

figure 9 (fin) : relations entre incidence et symbole décomposable

Notons que puisque l'opérateur ⌉ est commutatif, les axiomes indiquant les permutations possibles d'arguments pour `dro`, `interdd`, `intercc` et `ccir` sont redondants avec ceux de la figure 9 et peuvent être omis.

On peut enfin compléter notre exemple de signature d'univers géométriques en donnant des symboles de prédicats traduisant des relations géométriques usuelles comme par exemple ceux de la figure 10.

```
fonctions                       % relations géométriques
      tangentcd : cercle droite point ∅ bool % tangence cercle/droite en un point donné
      tangentdc : droite cercle point ∅ bool % tangence droite/cercle en un point donné
      tangentcc : cercle cercle point ∅ bool % tangence cercle/cercle en un point donné
      pll : point point point point ∅ bool        % les points forment un
parallélogramme
```

figure 10 : symboles de relations usuelles

Comme précedemment, on indique les permutations possibles d'arguments pour ces symboles relationnels au moyen des axiomes de la figure 11.

```
(X incidentdp Y) = (Y incidentpd X)
(X incidentcp Y) = (Y incidentpc X)
tangentcd(X, Y, z) = tangentdc(Y, X, z)
tangentcc(X, Y, z) = tangentcc(Y, X, z)
pll(X1, X2, X3, X4) = pll(X2, X3, X4, X1)
pll(X1, X2, X3, X4) = pll(X3, X4, X1, X2)
pll(X1, X2, X3, X4) = pll(X4, X1, X2, X3)
pll(X1, X2, X3, X4) = pll(X4, X3, X2, X1)
```

figure 11 : axiomes de permutation pour les symboles de relation

Il nous faut finalement exprimer les rapports existant entre les différents symboles fonctionnels et relationnels mentionnés. Nous tentons de traduire ceux-ci par des axiomes de nature géométrique comme ceux de la figure 12.

```
dist(X, Y) = Z ⊃ (X incidentpc ccr(Y, Z) = vrai)
(X incidentpc ccr(Y, Z) = vrai) ⊃ dist(X, Y) = Z
(dist(X, Y) = dist(X, Z)) ⅂ (Y eg Z = faux) ⊃ (X incidentpd med(Y, Z) = vrai)
(X incidentpd med(Y, Z) = vrai) ⊃ dist(X, Y) = dist(X, Z) ⅂ (Y eg Z = faux)
X = mil(Y, Z) ⅂ dist(Y, Z) = T ⊃ (Y incidentpc ccr(X, T/2) = vrai) ⅂
                                 (Z incidentpc ccr(X, T/2) = vrai)
pll(X1, X2, X3, X4) ⊃ mil(X1, X3) = mil(X2, X4) ⅂
                     (X1 eg X2 = faux) ⅂ (X1 eg X4 = faux)
mil(X1, X3) = mil(X2, X4) ⅂ (X1 eg X2)= faux) ⅂ (X1 eg X4 = faux) ⊃
                                                  pll(X1, X2, X3, X4)
X = mil(Y, Z) ⊃ (X incidentpd dro(Y, Z) = vrai)
X = prj(Y, Z) ⊃ (X incidentpd Z = vrai)
```

figure 12 : axiomes géométriques

Cette axiomatisation n'est évidemment pas complète : il nous semble très difficile de formaliser exactement et de manière raisonnablement efficace les constructions à la règle et au compas. Dans Progé, nous avons facilité l'introduction par un expert de nouveaux axiomes géométriques.

3.3. Utilisation

3.3.1. Contrôle

Quoi qu'il en soit, nous avons exposé une formalisation générale de la géométrie élémentaire : cette description peut tout autant servir à faire de la démonstration que des constructions. En fait, pour contrôler un tant soit peu un processus de construction dans un univers géométrique

tel que celui décrit précédemment, nous devons ajouter des notions propres aux constructions comme "construit" et "donné".

```
fonction
     donné _ : point ∅ bool
     donné _ : longueur ∅ bool
     donné _ : droite ∅ bool
     donné _ : cercle ∅ bool
     construit _ : point ∅ bool
     construit _ : longueur ∅ bool
     construit _ : droite ∅ bool
     construit _ : cercle ∅ bool

axiomes
(donné X = vrai) ⊃ (construit X = vrai)
construit O = vrai
construit I = vrai
(construit X = vrai) ⌉ (construit Y = vrai) ⊃ (construit dist(X, Y)= vrai)
(construit X = vrai) ⌉ (construit Y = vrai) ⊃ (construit interdd(X, Y) = vrai)
% de manière générale, pour tout symbole fonctionnel f
(construit X_i = vrai) pour i=1, ... n ⊃ (construit f(X_1, ... X_n) = vrai)
```

figure 13 : contrôle

3.3.2. Particularisation de l'univers géométrique

Par ailleurs, pour chaque problème de construction particulier, on ajoute la spécification décrite par l'énoncé.

Exemple.
L'énoncé suivant se traduit par la spécification de la figure 14.

Enoncé . *Etant donnés deux points distincts* a *et* b, *construire un point* c *à distance donnée de* a *et de* b.

```
fonctions
     a, b, c : ∅ point
     k, l : ∅ longueur

axiomes
     donné a = vrai
     donné b = vrai
     donné k = vrai
     donné l = vrai

     dist(a, c) = l
     dist(b, c) = k
     (a eg b) = faux
```

figure 14 : spécification correspondant à l'énoncé donné en exemple.

En ajoutant cette spécification à celle de l'univers géométrique donnée en exemple, on peut prouver que (`construit c = vrai`) est un théorème de cette théorie logique en utilisant le modus ponens (m.p.) comme règle d'inférence comme on peut le voir sur la figure 15.

```
dist(a, c) = l ⊃ ((c incidentpc ccr(a, l)) = vrai)          (axiome)
dist(a, c) = l                                              (axiome de l'énoncé)
((c incidentpc ccr(a, l)) = vrai)                           (m.p.)
dist(b, c) = k ⊃ ((c incidentpc ccr(b, k)) = vrai)          (axiome)
dist(b, c) = k                                              (axiome de l'énoncé)
((c incidentpc ccr(b, k)) = vrai)                           (m.p.)
(donné a = vrai) ⊃ (construit a = vrai)                          (axiome)
donné a = vrai                                              (axiome de l'énoncé)
construit a = vrai                                          (m.p.)
% ... de la même manière on a les trois propriétés suivantes
construit b = vrai
construit l = vrai
construit k = vrai

(construit a = vrai) ⅂(construit l = vrai) ⊃(construit ccr(a, l) = vrai) (axiome)
construit ccr(a, l) = vrai                                  (m.p.)
% ... de la même manière on a les deux propriétés suivantes
construit ccr(b, k) = vrai
construit intercc(ccr(a, l), ccr(b, k)) = vrai

(c incidentpc ccr(a, l) = vrai) ⅂(c incidentpc ccr(b, k) = vrai) ⅂(a eg b = faux)
⊃
                                          c = intercc(ccr(a, l), ccr(b, k))
c = intercc(ccr(a, l), ccr(b, k))                           (m.p.)
construit c = vrai
```

figure 15 : démonstration de la constructibilité de c

En réalité, ce que nous avons nommé axiome à la figure 15 sont des occurrences des schémas d'axiomes exposés plus haut. L'instanciation de ces schémas d'axiomes fait appel aux mécanismes d'inférence de la logique équationnelle définie par la spécification de cet univers géométrique.

Moyennant ceci, nous avons prouvé la constructibilité du point `c` dans l'univers géométrique particularisé par l'énoncé donné. On peut même extraire de cette démonstration une construction du point `c` qui réside dans l'égalité : `c = intercc(ccr(a, l), ccr(b, k))`. Nous verrons plus loin comment systématiser de telles démonstration dans un univers géométrique où les mécanismes d'inférences doivent être effectivement décrits. Ceci ne peut se faire qu'en passant au niveau "méta", c'est-à-dire en spécifiant la spécification d'un univers géométrique avec ses mécanismes de contrôles.

4. Méta-spécification d'un univers géométrique

Notre objectif est de décrire la mise en oeuvre d'une telle spécification d'univers géométrique lors de la résolution de problèmes de construction. Par ailleurs, nous avons noté plus haut qu'il est souhaitable de pouvoir enrichir un univers géométrique tant en ajoutant des axiomes géométriques que des sortes géométriques, des symboles fonctionnels et relationnels. C'est pourquoi, nous considérons dans notre approche un univers géométrique comme une donnée. Nous utilisons donc une sorte univers géométrique contenant des sortes représentant des concepts tels que "sorte géométrique", "symbole fonctionnel géométrique", "symbole

relationnel géométrique" et "axiomes". Nous décrivons ces différentes sortes en utilisant à nouveau le formalisme des spécifications algébriques : nous obtenons ainsi une spécification décrivant et manipulant une spécification. Outre son pouvoir descriptif, l'intérêt de cette nouvelle spécification consistera en l'existence d'une sémantique opérationnelle utilisant la réécriture. Cette méta-spécification pourra ensuite être mise en oeuvre, moyennant quelques arrangements, dans un langage de programmation, en l'occurrence le langage Prolog.

4.1. Sortes et opérations de base

Nous supposons l'existence d'un univers géométrique de base dans lequel les sortes représentées à la figure 16 possèdent les attributs géométriques que nous détaillons plus bas. Les mécanismes de modification d'un univers géométrique s'expriment facilement dans le formalisme de spécification choisi, mais pour simplifier la tâche du lecteur, nous remettons leur exposé à la section suivante. Par ailleurs, nous simplifions cette description en omettant dans les profils la sorte, notée `ug`, des univers géométriques. Par exemple, au lieu de noter :

`deg_max : sgeo ug ∅ entier`

nous aurons plus simplement :

`deg_max : sgeo ∅ entier`

Nous considérons en quelque sorte cet entier `deg_max(s)` comme un attribut de toute sorte géométrique `s` (dans l'univers géométrique de base). Voici le détail de ces sortes et opérations de base.

4.1.1. Sortes géométriques

La sorte `sgeo` figure la sorte des sortes géométriques. Une sorte géométrique a trois attributs dont l'interêt est plutôt d'ordre pratique : son degré de liberté, une fonction traçant à l'écran une figure correspondant à un objet de la sorte géométrique, une fonction permettant la saisie d'un objet de cette sorte (cf. figure 16).

```
sorte
     sgeo                  % sorte des sortes géométriques

fonctions
     deg_max : sgeo ∅ entier
     dessin : sgeo ∅ procédure_de_dessin
     saisie : sgeo ∅ procédure_de_saisie
```

figure 16 : sorte géométrique

Le *degré de liberté* d'une sorte géométrique représente le nombre de paramètres (ou coordonnées) requis pour une représentation algébrique de la sorte géométrique dans l'univers considéré.

Exemple.
Le degré de liberté de la sorte `point` dans un univers géométrique modélisant le plan vaut 2 tandis que pour la sorte `point` dans l'espace il vaut 3.

Nous avons vu au chapitre 1 qu'il était difficile de modéliser en toute généralité la notion de degré de liberté : dans Progé, c'est l'expert qui lors de la définition de l'univers géométrique de base décide du degré de liberté de chaque sorte géométrique. Un autre intérêt de la notion de degré de liberté, en liaison avec celle de degré de restriction d'une relation d'incidence (voir plus bas), est de mesurer l'état de construction dans objet à un instant donné de la phase de résolution d'un problème : par exemple, un point sans relation avec les autres objets de l'univers aura un degré de liberté égal à 2 (dans le cas du plan) tandis qu'un point assujetti à appartenir à une droite sans autres contraintes aura un degré de liberté égal à 1.

Les deux attributs `dessin` et `saisie` d'une sorte géométrique ont un rôle pratique dans l'interface utilisateur de Progé et sont utilisés lors de l'interprétation d'un programme de construction géométrique. Nous ne décrirons pas ici les sortes `procédure_de_dessin` et `procédure_de_saisie` qui sont, dans notre cadre, d'un intérêt théorique limité. En toute rigueur, un langage graphique devrait être défini pour qu'il soit possible spécifier correctement une interface expert de Progé. Pour l'heure, ces procédures sont écrites en Delphia Prolog à l'aide de la boîte à outil Pixia de Delphia.

4.1.2. Symboles fonctionnels géométriques

La sorte `cog` (constructeur d'objet géométrique) modélise les symboles fonctionnels utilisés dans un univers géométrique. Comme pour la sorte `sgeo`, nous avons assigné certains attrbuts à la sorte `cog` : ces attributs concernent l'arité, la coarité, l'*ambiguïté* et l'évaluation d'un symbole fonctionnel (cf. figure 17).

Les fonctions `arité` et `coarité` recouvrent les attributs classiques des symboles fonctionnels des spécifications algébriques. Notons ici que les fonctions constantes, dont l'arité est une liste vide, sont en général introduites par un énoncé comme à la section 3. Nous verrons par la suite que Progé réserve un traitement spécial à ces constantes.

```
sorte
     cog                          % symbole fonctionnel géométrique

fonctions
     nilf : ∅ cog          % symbole de fonction nul
     arité : cog ∅ sgeo*
     coarité : cog ∅ sgeo
     ambigu : cog ∅ bool
     févaluation : cog ∅ fonction_de_calcul
```

figure 17 : symboles fonctionnels géométriques

Un symbole fonctionnel géométrique est *ambigu* si les termes qu'il sert à construire désignent plusieurs objets.

Exemple.
Si dans l'univers considéré nous incluons le symbole fonctionnel `intercc` avec le sens de la section précédente, nous aurons `ambigu(intercc) = vrai`, signifiant par là qu'une intersection entre deux cercles peut contenir 0, 1 ou 2 points. Nous nous rapprochons ainsi de la signification habituelle de fonctions comme l'intersection entre cercle et droite ou

l'intersection entre deux cercles : lors d'un traitement numérique, nous saurons qu'il faut considérer une liste de valeurs plutôt qu'une seule en résultat de leur interprétation.

La fonction d'évaluation d'un symbole fonctionnel géométrique, donné par la fonction `févaluation`, est du même ordre que les fonctions de dessin et de saisie pour les sortes géométriques. Un langage de programmation pour définir la sorte `fonction_de_calcul` serait à définir lors de la spécification d'une interface expert destinée à construire des univers géométriques de base. Le résultat rendu lors de l'interprétation d'une fonction d'évaluation est soit une valeur du type de la représentation algébrique de la sorte attendue (i.e. fondamentalement une liste de réels associé à une sorte géométrique), soit une liste de valeurs de ce type losque le symbole fonctionnel géométrique est ambigu.

Un autre attribut important correspond à la *décomposabilité* des symboles fonctionnels en relations constructives. Mais pour le décrire correctement, nous devons tout d'abord parler des symboles relationnels géométriques.

4.1.3. Symboles relationnels géométriques

La sorte `crg` (constructeur de relations géométriques) représente les symboles relationnels géométriques. Parmi les symboles relationnels, nous distinguons les symboles de relation binaire "constructive" que nous appelons des symboles de *relation participative* et qui correspondent à la sorte `tp` (titre de participation), des autres symboles de prédicat. A cet effet, nous avons utilisé la notation de J. A. Goguen pour les *sortes ordonneés* [Goguen 89]. Les relations participatives sont celles qui interviennent dans la décomposition des symboles fonctionnels tandis que les prédicats non constructifs ou relations non participatives ne sont, en général, pas exploitables directement dans les constructions géométriques.

Exemple.
Avec les notations de la section précédente, le symbole de prédicat `incidentpd` est participatif car il intervient dans la décomposition du symbole fonctionnel `dro`. En revanche, la relation de tangence entre deux cercles -désignée par le symbole `tangentcc` dans la spécification d'univers géométrique de la section précédente- n'est pas participative parce qu'il n'intervient dans aucune décomposition de symbole fonctionnel.

```
sortes
     crg                              % symbole relationnel géométrique
     tp < crg                         % symbole de relation participative

fonctions

     profil : crg ∅ sgeo*
     ptitre : crg ∅ {egalité, incidence, cond}
     pévaluation : crg ∅ fonction_booléenne

     nilp : ∅ tp                      % symbole de prédicat nul
     deg_rest : tp ∅ entier           % degré de restriction d'une relation participative
     recipp : tp ∅ tp                 % relation réciproque d'une relation participative
     recipf : tp ∅ cog                % fonction réciproque d'une relation participative

     décomposition : cog ∅ tp* % attribut "retardataire" de la sorte cog
```

figure 18 : symboles relationnels géométriques

L'attribut `ptitre` indique si le symbole relationnel représente une relation d'égalité, une relation participative ou un autre type de relation. Dans Progé, les relations d'égalité sont typés pour des raisons d'efficacité. Pour les symboles relationnels généraux, les deux autres attributs présentés sont explicables de la même manière que pour les symboles fonctionnels.

Pour les relations participatives, nous avons ajouté un attribut entier correspondant au *degré de restriction* de la relation, c'est-à-dire le nombre de degrés de liberté restreint par la relation lorsque le deuxième argument est construit : ainsi le degré de restriction de la relation d'appartenance à une droite vaut 1 :

$$\texttt{deg\_rest(incidentpd)} = 1$$

De manière intuitive ce degré correspond algébriquement au nombre d'équations polynomiales introduites par la relation, aucune des équations ne devant être de la forme :

$$(P_1)^2 + (P_2)^2 + \ldots + (P_n)^2 = 0\,.$$

puisqu'une telle équation est équivalente aux n équations[6]

$$P_i = 0$$

Dans Progé, l'attribution d'un degré de restriction à une relation participative doit être le fait de l'expert.

Les deux derniers attributs d'un symbole de relation participative, sélectionnés par les fonctions `recipp` et `recipf`, sont les relations réciproques.

Exemple.
Dans l'univers de base utilisé par Progé, nous avons :

$$\texttt{recipp(incidentpd)} = \texttt{incidentdp}$$
$$\texttt{recipf(centrede)} = \texttt{centre}$$

pour traduire les axiomes de la section 3 :

$$\texttt{(x inicidentpd y)} = \texttt{(y incidentdp x)}$$
$$\texttt{(y centrede x = vrai)}\ \ \texttt{y = centre(x)}$$

Venons en à présent à l'attribut de décomposition d'un symbole fonctionnel laissé en suspens à la section 4.1.2. La fonction `décomposition` résulte d'une tentative de modélisation de la méthode des lieux : un symbole fonctionnel géométrique `f` à n arguments est *décomposable* en n relations binaires constructives ψ_i (i=1, ...n) si :

$$\texttt{o} = \texttt{f}(o_1, \ldots o_n)\ \ o_i\ \psi_i\ \texttt{o}\ \text{pour i=1, ...n}$$

Exemple.
Le symbole fonctionnel dro est décomposable dans Progé car

6Ceci n'est pas un cas d'école : qu'on considère le cas d'un cercle dégénéré réduit à un point.

`d = dro(a, b)` `a` est sur la droite `d` et `b` est sur la droite `d`

ce qu'on traduit dans Progé par :

`décomposition(dro) = [incidentpd, incidentpd]`

Notons que les axiomes d'incidence de la figure 9 deviennent ainsi attributs des symboles fonctionnels décomposables.

4.2. Termes et axiomes

Aprés avoir spécifié les notions de sorte, de symbole fonctionnel et de symbole relationnel, nous précisons comment sont construits les termes et les axiomes dans un univers géométrique.

La sorte `term0` désigne les termes de bas niveau (au niveau de l'univers géométrique) tandis que la sorte `var0` désigne les variables de l'univers géométrique. Les objets de sorte `term0` sont construits de manière usuelle (Cf. figure 19).

```
sorte
     term0
     var0 < term0

fonctions
     nouvar : ∅ var0                          % générateur de nouvelles variables
     est_var : term0 ∅ bool                   % teste la nature d'un terme

     nil : ∅ term0                       % terme nul
     _(_) : cog term0* ∅ term0           % constructeur usuel de termes
     fonctf : term0 ∅ cog                     % symbole fonctionnel d'un terme fonctionnel
     argsf : term0 ∅ term0*                   % arguments d'un terme fonctionnel
     est_fonc : term0 ∅ bool                  % teste la nature d'un terme
     cste : term0 ∅ bool                 % teste la nature d'un terme
```

figure 19 (début) : sortes de base (terme)

```
     _(_) : crg term0* ∅ term0           % constructeur de termes prédicatifs (surcharge de())
     fonctp : term0 ∅ crg                     % symbole relationnel d'un terme prédicatif
     argsp : term0 ∅ term0*                   % arguments d'un terme prédicatif
     est_pred : term0 ∅ bool                  % teste la nature d'un terme

préconditions
pré(F(L)) = L et arité(F) ont même longueur et la sorte de chaque élément de
      L coïncide avec la sorte de même place de arité(F).
pré(fonctf(T)) = pré(argsf(T)) = est_fonc(T)
pré(fonctp(T)) = pré(argsp(T)) = est_pred(T)

axiomes
     fonctf(F(L)) = F
     argsf(F(L)) = L
     fonctp(F(L)) = F
     argsp(F(L)) = L
```

figure 19 (fin) : sortes de base (terme)

La fonction `nouvar` fournit une variable qui n'est pas encore présente dans l'univers géométrique. En fait son profil exact devrait être :

`nouvar : ug` ∅ `var0 ug`

L'univers géométrique produit contient la nouvelle variable.

Les deux fonctions notées de manière infixe _(_) sont classiquement pour la formation de termes à l'aide de symboles fonctionnels ou de symboles de prédicat. La précondition exprimée de manière informelle à la figure 19 indique que la liste des arguments `term0*` doit correspondre en nombre et en sortes à l'arité donnée par la fonction de même nom. Rappelons que nous n'avons pas imposé de typage pour les variables.

Les trois fonctions `est_var`, `est_fonc` et `est_pred` indiquent la nature d'un terme, à savoir si c'est une variable, un terme construit avec un symbole fonctionnel ou relationnel. La fonction `cste` indique si un terme est une constante, c'est-à-dire un terme consruit avec un symbole fonctionnel d'arité nulle et une liste vide d'arguments.

A l'aide des termes, nous fabriquons les axiomes de notre théorie. Ces axiomes sont de différentes formes et sont distingués pour des raisons de contrôle. Notons que certains axiomes de la section 3 ont déjà été introduit dans la spécification d'univers géométrique sous forme d'attributs comme celui donné par la fonction décomposition pour la sorte `cog`.

```
sorte
     fegal                                  % forme égalitaire
     impl                                   % implication
     règle                                  % règle de production

fonctions
     _ = _ : term0 term0 ∅ fegal
     _ => _ : fegal term0* ∅ impl
     si _ et _ alors _ : term0* vérification* term0* ∅ règle
```

figure 20 : sortes de base (axiomes)

Les formes égalitaires correspondant à la sorte `fegal` sont des schémas d'axiomes indiquant l'égalité entre deux termes. A l'aide de ce type d'axiomes, nous pourrons indiquer de quelle manière on peut permuter les arguments d'un symbole fonctionnel ou relationnel.

Exemple.
Dans l'univers de base de Progé, nous avons :

`mil(X, Y) = mil(Y, X)`

ou

`pll(X1, X2, X3, X4) = pll(X2, X3, X4, X1)`

La sorte `impl` représente des implications qui seront appliquées automatiquement lors de certains traitements.

Exemple.
L'axiome

```
X = mil(Y, Z) => X incidentpd dro(Y, Z)
```

sera appliqué par Progé chaque fois qu'une égalité correspondant au membre gauche de l'implication aura été trouvée.

La troisième sorte d'axiome se présente sous la forme bien connue d'une règle de production. Ces règles représentent des axiomes géométriques comme ceux de la figure 12 de la section 3.

Exemple.
Nous avons la règle :

```
si [dist(X, Y) '=l=' L] et [connu L, connu X, pas_connu Y]
alors [Y incidentpc ccr(X, L)]
```

pour traduire l'assertion suivante.

Pour tout points X et Y si la distance de X à Y est construite ainsi que le point X alors le cercle de centre X et de rayon dist(X, Y) participe à la construction de Y.

Les éléments de vérification font références à une figure formelle et ne seront décrits qu'au chapitre 6.

L'introduction des axiomes dans un univers géométrique se fait de plusieurs manières dans notre approche : soit sous forme d'attributs des sortes `cog` et `crg`, soit sous forme de règles de production. En ce qui concerne les attributs, nous pouvons compléter notre description des entités d'un univers géométrique par les fonctions de la figure 21 qui permettent d'indiquer :
- pour `equiv`, quelles sont les permutations d'arguments possibles pour un terme fonctionnel ;
- pour `maj`, quelles sont les incidences immédiates de la prise en compte d'une forme égalitaire ;
- pour `pequiv`, quelles sont les permutations d'arguments possibles pour un terme relationnel.

```
fonctions
     equiv : cog ∅ fegal*
     maj : cog ∅ impl
     pequiv : crg ∅ fegal*
```

figure 21 : attributs complémentaires

Exemple.
Nous avons par exemple :

```
equiv(mil) = [mil(X, Y) = mil(Y, X)]
maj(prj) = (P = prj(M, D) => [P incidentpd D])
pequiv(tangentdc)= [tangentdc(D, C, X) = tangentcd(C, D, X)]
```

5. Utilisation des termes et axiomes : substitutions et filtrage

La conduite de raisonnements dans un univers géométrique nécessite des mécanismes d'unification et/ou de filtrage. Ces notions sont bien définies en mathématique mais pas de

manière constructive. Nous nous proposons ici de préciser leur mécanisme en utilisant le formalisme et la terminologie des spécifications algébriques.

5.1. Substitutions

Nous définissons tout d'abord, l'unification standard de manière classique en définissant les composants de substitution et les substitutions.

Définition (composant de substitution). Les *composants de substitution* [Delahaye 87] sont des couples, notés <V, T>, où V est une variable et T un terme. Nous appelons V la *référence* du composant de substitution <V, T> et T son *contenu*. Appliquer un composant de substitution <V, T> à un terme T', c'est remplacer toutes les occurrences de V dans T' par T (Cf. Figure 22).

```
sorte
     sue                                        % composant de substitution

fonctions
<_, _> : var0 term0 ∅ sue                  % constructeur d'un composant de substitution
ref  : sue ∅ var0                          % référence
cont  : sue ∅ term0                        % contenu
apse  : sue term0 ∅ term0                  % appliquer à un terme
apsel  : sue term0* ∅ term0*               % ... à une liste de termes

axiomes
ref(<X,T>) = X
cont(<X,T>) = T

apse(<X, T>, Y) = si est_var(Y)
                  alors
                        si X = Y
                        alors T
                        sinon Y
                  sinon
                        si est_fonc(Y)
                        alors fonctf(Y)(apsel(argsf(Y))
                        sinon fonctp(Y)(apsel(argsp(Y))

apsel(<X, T>, []) = []
apsel(<X, T>, L) = [apse(<X, T>, tête(L)) | apsel(<X, T>, corps(L))]
```

figure 22 : Composant de substitution

Exemple.

```
apse(<X, ccr(point0, long00)>, intercd(X, Y)) = intercd(ccr(point0, long00), Y)
```

Définition (substitution). Une *substitution* est une application s de l'ensemble des variables vers l'ensemble des termes telle que s(x) = x sauf pour un ensemble fini de variables appelé le *support* de la substitution. Une substitution est caractérisée par l'ensemble de ses composants de substitutions <x, s(x)>. Ceci peut se traduire en disant qu'une substitution est une liste de composants de substitution telle que deux composants distincts n'aient pas la même référence.

```
sorte
      su renomme sue*                    % substitution

fonctions
sv  renomme []                           % substitution vide
nils : ∅ su                              % substitution impossible
cs  : su su ∅ su                         % union de deux substitutions
aps : su term0 ∅ term0                   % appliquer une substitution
apsl : su term0* ∅ term0*

préconditions
pré(cs(s1, s2)) = les supports de s1 et de s2 sont disjoints
pré(aps(s, t)) = (s  nils)
pré(apsl(s, lt)) = (s  nils)

axiomes
cs(s, sv) = si s = nils alors nils sinon s
cs(s, nils) = nils
cs(s1, [se | s2]) = si s1 = nils alors nils sinon [se | cs(s1,s2)]

aps(sv, t) = t
aps([se | s], t) = aps(s, apse(se, t))

apsl(s, []) = []
apsl(s, [t | lt]) = [aps(s, t) | apsl(s, lt)]
```

figure 23 : substitutions

Exemple.

```
aps([<X, ccr(o, r)>, <Y, cdiam(A, B)>], intercc(X, Y))
                                          = intercc(ccr(o, r), cdiam(A, B))
```

La substitution impossible `nils` sera utilisée dans des opérations comme le filtrage ou l'unification pour indiquer l'inexistence d'une substitution normale. Dans la spécification de la figure 23, l'application d'une substitution est décomposée en l'application de chacun des constituants de substitution dans l'ordre où ils sont rangés dans la liste. Ceci s'écarte quelque peu de la définition d'une substitution où toutes les composants de substitution sont appliqués d'un bloc. Ainsi lorsque des variables du support se trouvent dans les termes images de la substitution, on peut avoir des résultats différents.

Exemple.
Soit s = [<X, Y>, <Y, X>], on a s(f(X, Y)) = f(Y, X) tandis qu'avec les axiomes de la figure 23, nous avons : aps(s, f(X, Y)) = f(X, X).

Dans l'usage que nous faisons des substitutions, cette situation n'arrive pas et nous pouvons nous contenter de cette description simplifiée.

5.2. Filtrage.

Définition (filtre). Nous utilisons cette spécification pour définir le *filtre* d'un terme `t1` par un terme `t2` qui est une substitution (si elle existe) `s` telle que `s(t1) = t2`. Nous avons considéré le filtrage dans un cas qui simplifie l'exposé et qui sera suffisant pour la suite : nous supposons que `t2` est un terme clos, c'est-à-dire un terme qui ne contient pas de variables. Ceci évite, en particiliers, les désagréments évoqués à la section précédente.

```
fonctions
fi : term0 term0 ∅ su                    % filtre du premier terme par le second
fili :term0* term0* ∅ su                 % filtre de la première liste de terme par la deuxième liste

préconditions
pré(fi(t1, t2)) = t2 est clos
pré(fili(L1, L2)) = L2 ne contient que des termes clos

axiomes
fi(t1, t2) =   si est_var(t1)
               alors [<t1, t2>]
               sinon si fonctf(t1) = fonctf(t2)
                     alors fili(argsf(t1), argsf(t2))
                     sinon nils

fili([], L) = si L = [] alors sv sinon nils
fili([t1|L1], []) = nils
fili([t1|L1], [t2|L2]) = cs(fili(apsl(S, L1), L2), S)
                 où S = fi(t1, t2)
```

figure 24 : filtrage

Dans les conditions simplificatrices où nous nous plaçons, nous avons le résultat attendu :

$$aps(fi(t1, t2), t1) = t2 \text{ si } fi(t1, t2) \neq nils$$

Exemple.

```
fi(ccr(X, dist(X, Y)), ccr(point00, dist(point00, b)))
= fili([X, dist(X, Y)], [point00, dist(point00, b)])
=cs(fili(apsl(fi(X, point00),[dist(X, Y)]),[dist(point00, b)]),fi(X,point00))
=cs(fili(apsl([<X,point00>],[dist(X, Y)]), dist(point00, b)]), [<X, point00>])
= cs(fili([dist(point00, Y)], [dist(point00, b)]), [<X, point00>])
= cs([<Y, b>], [<X, point00>])
= [<X, point00>, <Y, b>]
```

5.3. Application : recherche des termes équivalents par permutations d'arguments

A la section 5.2., il s'agit de filtrage simple où le contexte géométrique n'intervient pas : il pourrait en être ainsi sur n'importe quelle algèbre de termes. On peut cependant utiliser cette opération de filtrage pour appliquer des axiomes comme, par exemple, pour appliquer une forme égalitaire et pour déterminer tous les termes fonctionnels équivalents à un terme clos par permutation des arguments.

```
fonctions
      apfeg : term0 fegal ∅ term0              % applique une subst. à une forme égalitaire
      apfegl : term0 fegal* ∅ term0*           % idem, pour une liste de formes égalitaires
      term_equiv : term0 ∅ term0*              % termes equiv. modulo les permutations d'args.
                                  % fonctions secondaires
      term_equivl : term0* ∅ term0**   % idem, pour une liste de termes
      ajt : term0 term0** ∅ term0**            % ajt(t, L) ajoute t en tête de chaque liste de L
      ajtl : term0* term0** ∅ term0**  % comme ajt mais pour une liste en premier arg.
      perm_arg : term0 ∅ term0*
```

```
      perm_argl : term0* ∅ term0*          % comme perm_arg mais pour une liste
      univl : cog term0** ∅ term0*          % comme l'opérateur _(_) mais pour une liste d'arg.

préconditions
pré(apfeg(t, tv1 = tv2)) = t est un terme clos et fonctf(t) = fonctf(tv1)
pré(term_equiv(t)) = t est un terme fonctionnel clos

axiomes
apfeg(t, tv1 = tv2) = aps(fi(tv1, t), tv2)
apfegl(t, []) = []
apfegl(t, [t1| l]) = [apfeg(t, t1) | apfegl(t, l)]
term_equiv(t) = perm_argl([ t | apfegl(t, equiv(fonctf(t)))])

term_equivl([]) = [[]]
term_equivl([T|L]) = ajtl(term_equiv(T), term_equivl(L))

ajt(T, []) = []
ajt(T, [L1 | LL]) = [[T|L1] | ajt(T, LL)]
% ajtl(L1, L2) construit la liste de toutes les listes de L2 auxquelles on ajoute en tête chaque élément de L1.
ajtl([], L) = L
ajtl([T|L1], L) = ajt(T, L)) + ajtl(L1,L)          % + désigne la concaténation

% perm_arg(t) retourne les termes correspondants aux équivalent des arguments par permutation
perm_arg(F(L)) = univl(F, term_equivl(L))
perm_argl([]) = []
perm_argl([t | l]) = perm_arg(t) + perm_argl(l)     /* + désigne la concaténation

univl(F, []) = []
univl(F, [L1 | LL]) = [F(L1) | univl(F, LL)]
```

figure 25 : permutations

La fonction `apfeg` applique une forme égalitaire en intanciant celle-ci à l'aide de son premier argument et retourne le second membre de cette occurrence de la forme égalitaire.

Exemple.

`apfeg(med(a, b), med(X, Y) = med(Y, X)) = med(b, a).`

La fonction `apfegl` fait la même chose pour une liste de formes égalitaires. La fonction `term_equiv` fournit la liste des équivalents à un terme modulo les permutations d'arguments autorisées. La fonction `term_equivl` fait de même avec une liste de termes et fournit la liste des listes de termes équivalents à ceux de la liste en argument.

Exemple.

Nous ne démontrons pas le bien fondé de cette axiomatisation, mais nous en montrons le fonctionnement sur un exemple où nous cherchons les termes équivalents à `mil(mil(a, b), c)` :

Nous avons vu que `equiv(mil) = [mil(X, Y) = mil(Y, X)].` On a donc

```
term_equiv(mil(mil(a, b), c)) =
perm_argl([mil(mil(a,b),c) | apfegl(mil(mil(a,b),c), [mil(X,Y) = mil(Y,X)])])
```

Or

```
apfegl(mil(mil(a, b), c), [mil(X, Y) = mil(Y, X)])
```

```
      = [apfeg(mil(mil(a, b), c), mil(X, Y) = mil(Y, X))]
      = [aps(fi(mil(X, Y), mil(mil(a, b), c)), mil(Y, X))]
      = [aps([<X, mil(a, b)>, <Y, c>], mil(Y, X))] = [mil(c, mil(a, b))]
```

autrement dit :

```
term_equiv(mil(mil(a, b), c))
      = perm_argl([ mil(mil(a, b), c), mil(c, mil(a, b))]
      = perm_arg(mil(mil(a, b), c)) + perm_arg(mil(c, mil(a,b))
```

Examinons le premier terme de la concaténation :

```
perm_arg(mil(mil(a, b), c)) = univl(mil, term_equivl([mil(a, b), c]))
```

or (en raccourcissant un peu)

```
term_equivl([mil(a, b), c])
            = ajtl([mil(a, b), mil(b,a)], term_equivl([c]))
            = ajtl([mil(a, b), mil(b,a)], [[c]])
            = [[mil(a, b), c]] + [[mil(b, a), c]]
            = [[mil(a, b), c], [mil(b, a), c]]
```

et

```
univl(mil,[[mil(a,b),c],[mil(b,a),c]]) = [mil(mil(a,b),c),mil(mil(b,a),c)]
```

De la même manière :

```
perm_arg(mil(c, mil(a,b)) = [mil(c, mil(a, b)), mil(c, mil(b, a))]
```

d'où :

```
term_equiv(mil(mil(a, b), c)) =
[mil(mil(a,b),c), mil(mil(b,a),c), mil(c,mil(a,b)), mil(c,mil(b,a))]
```

5.4. Filtrage modulo les permutations d'arguments

On peut définir alors facilement le *filtrage modulo les permutations* comme à la figure 26 (nous nous plaçons toujours dans le cas où le deuxième terme est clos). Muni de cette opération de filtrage, il est possible d'envisager l'application des règles contenues dans l'univers : appliquer une règle c'est l'intancier à l'aide d'axiomes sans variables c'est-à-dire de faits qui sont dérivés de l'énoncé et enrichir l'univers géométriques des nouveaux faits obtenus, qui sont des théorèmes de l'univers géométrique. Nous verrons dans les chapitres ultérieurs comment est organisé cet emmagasinement de connaissances dans de nouvelles entités contenues dans l'univers géométrique à savoir une figure et un raisonnement formels.

```
fonctions
fip : term0 term0 ∅ su*     % filtrage du premier terme par le second mod. les permutions
fil : term0 term0* ∅ su*    % filtrage du premier terme par chacun des termes de la liste
axiomes
fip(t1, t2) = fil(t1, term_equiv(t2))

fil(t, []) = []
fil(t, [t1|l]) = [fi(t, t1) | fil(t, l)]
```

figure 26 : filtrage modulo les permutations d'arguments

6. En toute rigueur, extensions

Dans la section précédente, nous avons examiné les notions principales intervenant dans un univers géométrique, lui-même présenté comme un réceptacle de la spécification géométrique (de bas niveau) présentée à la section 3. Dans la présentation de la section 4, les entités de base ont des "attributs" géométriques qui n'existent habituellement pas dans les spécifications classiques. Tout ceci peut être représenté rigoureusement par une spécification algébrique de la sorte "univers géométrique". Cette spécification est d'ailleurs indispensable si l'on veut décrire la manière d'étendre un univers géométrique de base (par le biais d'une interface expert par exemple).

Une description complète de la sorte univers géométrique est simple mais fastidieuse, c'est pourquoi nous n'en donnons à la figure 23 qu'une partie correspondant à la gestion des sortes géométriques. On y trouve les fonctions d'adjonctions d'une sorte géométrique et de tous ses attributs ainsi que des fonctions de retraits de sortes géométriques pour appauvrir l'univers géométrique. Cette dernière possibilité peut être intéressante dans le cadre de l'enseignement des constructions géométriques. Ces fonctions sont celles utilisées par l'expert pour définir l'univers géométrique dans lequel va évoluer Progé. Il est, bien entendu, de la responsabilité de l'expert de s'assurer que l'univers qu'il décrit est réaliste et qu'il est cohérent : une sorte géométrique n'a qu'un attribut `deg_max`, qu'une procédure de dessin, etc. Si cela semble très facile lorsqu'il s'agit des sortes géométriques, il en va autrement en ce qui concerne les symboles fonctionnels, les symboles relationnels et les règles géométriques. Une interface expert/Progé "intelligente" nous semblerait être d'un grand interêt tant du point de vue pratique que théorique.

```
sortes
     ug                                          % sorte univers géométrique
     sgeo                                        % sorte sorte géométrique
fonctions
     uv : ∅ ug
                          % adjonctions d'une sorte et de ses attributs
     asg : ug sgeo ∅ ug                  % ajout d'une sorte à l'univers géométrique
     adm : ug sgeo entier ∅ ug           % ajout d'un degré de liberté pour une sorte donnée
     apd : ug sgeo procédure_de_dessin ∅ ug
     aps : ug sgeo procédure_de_saisie ∅ ug
     asga : ug sgeo entier procédure_de_dessin procédure_de_saisie ∅ ug
```

figure 27 (début) : univers et sorte géométrique

```
fonctions (suite)
                          % retrait d'une sorte et de ses attributs
     rsg : ug sgeo ∅ ug                  % retrait d'une sorte
     rdm : ug sgeo ∅ ug                  % d'un degré de liberté
     rpd : ug sgeo ∅ ug                  % d'une procédure de dessin
     rps : ug sgeo ∅ ug                  % d'une procédure de saisie
     rsga : ug sgeo ∅ ug          % de tout
                          % sélecteurs d'une sorte géométrique section 4
     deg_max : ug sgeo ∅ entier
     dessin : ug sgeo ∅ procédure_de_dessin
     saisie : ug sgeo ∅ procédure_de_saisie

     appartient : sgeo ug ∅ bool
```

```
     c2sg : chaîne ∅ sgeo
     sg2c : sgeo ∅ chaîne

préconditions
pré(asga(u, s, n, p, ps)) = (appartient(s, u) = faux)
pré(rsg(u, s)) = appartient(s, u)
pré(deg_max(u, s)) = appartient(s, u)

axiomes
asga(u, s, n, p, ps) = aps(apd(adm(asg(u, s), s, n), s, p), s, ps)

rsg(asg(u, t), s) = si t=s alors u sinon asg(rsg(u, s), t)
rsg(Ψ(u, a1, ... an), s) = Ψ(rsg(u, s), a1, ... an)
                  % où Ψ désigne un constructeur (n+1)aire comme adm, apd ou aps
                  % même genre d'axiomes pour rdm, rpd et rps
rsga(u, s) = rsg(rdm(rpd(rps(u, saisie(u,s)), dessin(u,s), deg_max(u,s), s)

deg_max(adm(u, t, n), s) = si s=t alors n sinon deg_max(u, s)
deg_max(Ψ(u, a1, ... an) , s) = deg_max(u, s)
                  % où Ψ désigne un constructeur (n+1)aire
                  % même genre d'axiomes pour les fonctions dessin, saisie et appartient

sg2c(c2sg(c)) = c
```

figure 27 (fin) : univers et sorte géométrique

Chapitre 5

Enoncés et programmes de construction

Ecarquillant les yeux tandis, tandis que le boucher
Disait : « Ce fut toujours un sacré plaisantin ! »
Ils le virent, joyeux, leur héros innomé,
Leur boulanger, au faîte d'un rocher voisin.

De son point de vue, l'utilisateur perçoit l'univers géométrique de Progé sous forme du vocabulaire et de la syntaxe utilisés pour écrire des énoncés et lire, éventuellement, des programmes de construction.

1. Introduction : des langages extérieurs

Les manières de communiquer avec les logiciels de construction géométrique sont trés variées. L'utilisateur peut se servir d'un langage de programmation et d'outils de désignation et sélection comme dans Euclide [Allard & Pascal 86]. Dans Géophile [Braun 88], il utilise une interface permettant d'entrer les différents éléments d'une construction dans l'ordre où il le désire, comme dans la méthode Médée [Ducrin 85], le système se chargeant de maintenir la cohérence en demandant les éléments manquants. Dans Cabri Géométre [Baulac 90], l'utilisateur se sert uniquement d'une interface graphique pour sa construction : ceci permet à celui-ci de reconnaître l'univers géométrique de Cabri sans passer par un apprentissage exagérément long. L'univers géométrique peut être enrichi en utilisant des macro-constructions qui peuvent prendre place dans les menus. La plupart des logiciels de dessin assisté par ordinateur combinent une interface graphique avec des possibilités de programmation comme dans AutoCad [Autodesk 85].

En ce qui concerne les logiciels chargés de résoudre des exercices de géométrie, le problème de la communication est plus aigu. Il s'agit pour le programme de comprendre un énoncé avec tous les sous-entendus que cela comporte suivant les domaines. En DAO, où une interface graphique est souvent utilisée, un segment qui semble être vertical *est*, habituellement, vertical, deux droites, ou plus généralement deux courbes, qui semblent parallèles *sont* parallèles [Aldefeld 91] : il s'agit de concevoir des interfaces graphiques capables de comprendre les intentions de l'utilisateur, domaine de recherche encore largement ouvert. En EIAO, où en principe un énoncé contient tout ce qui est nécessaire de manière non ambiguë à la résolution, il existe aussi des problèmes de sous-entendus indépendants de l'utilisation du langage naturel [Pintado 92]. Ainsi, dans l'énoncé

Soit deux points `A` *et* `B`, *et un point* `I` *tel que* `AI = BI`. *Que peut-on dire du point* `I` ?

il est implicite que les points `A` et `B` sont différents et qu'ils sont dans le plan (ou dans l'espace, ceci dépend du chapitre du livre). Et la réponse

`I` *est sur la médiatrice de* [`A, B`]

satisfera tout professeur de mathématiques normalement pointilleux et traitant à ce moment de la géométrie dans le plan. Un autre problème concernant les énoncés de construction est la considération de constructions implicites comme dans l'énoncé suivant :

Etant donnés trois cercles tangents, construire un quatrième cercle tangent aux trois premiers.

Ce type de problème et ses incidences sur l'interprétation d'un programme de construction sera discuté plus en détail au cours de ce chapitre.

En plus de la résolution d'un problème de construction, l'utilisateur s'attend à ce que celle-ci lui soit communiquée soit sous forme graphique, soit, en EIAO, sous forme d'une solution commentée qu'il peut demander au système d'expliquer sa solution. Progé fournit cette solution sous forme de programme de construction non commenté. L'étude d'un module d'explication mériterait d'être faite mais n'entre pas dans le sujet de notre thèse.

2. La sorte énoncé

Les énoncés habituels (en géométrie comme ailleurs) sont habituellement formulés en utilisant le langage naturel. Il ne s'agit pas pour nous de décrire un système comprenant le langage naturel dans le cadre spécialisé des constructions géométriques comme cela est fait dans d'autres domaines. Notre approche consiste à définir un langage et des structures de données rigides permettant de poser des problèmes de construction avec le minimum d'ambiguïté, à charge pour l'utilisateur de traduire un énoncé en langage naturel dans ce cadre. Ceci n'implique d'ailleurs pas qu'il y ait une unique traduction : nous verrons comment certains problèmes peuvent résulter d'énoncés "mal" posés.

Un énoncé précise les noms et la nature des objets décrits ainsi que les relations existants entre ceux-ci. Nous avons donc distingué dans un énoncé formel des déclarations et des contraintes.

2.1. Déclarations

Une déclaration introduit le nom d'un nouvel objet géométrique ainsi que sa nature. En outre, à l'occasion de la déclaration d'un objet, on indique le motif de son introduction c'est-à-dire si cet objet est donné, cherché ou sert à nommer une construction en vue d'une description plus facile.

Nous avons donc les sortes et les fonctions, qui en sont les constructeurs de base, de la figure 1.

```
sortes
     déclaration

fonctions
     donné : sgeo chaîne → déclaration
     cherché : sgeo chaîne → déclaration
     mentionné : sgeo chaîne → déclaration
     nomme : sgeo chaîne term0 → déclaration
```

figure 1 : Déclarations

Exemple.
On peut avoir, par exemple, les déclarations suivantes :

```
donné(point, a)
cherche(droite, d)
mentionné(point, a1)
nomme(point, o, centre(c))
```

La signification intuitive de ces déclarations est évidente sauf, peut être, la troisième. En fait, un objet mentionné est un objet qui sera utilisé dans la suite de l'énoncé mais qui n'est ni donné, ni cherché. Ce sera par exemple le nom d'un point de contact entre deux cercles.

Les déclarations utilisent donc l'univers géométrique de base à travers l'usage des sortes géométriques et des termes construits à l'aide de symboles fonctionnels géométriques. En retour, l'interprétation de ces déclarations enrichit l'univers géométrique de base -pour fournir un univers géométrique spécialisé- avec de nouveaux symboles fonctionnels, correspondant à des constantes, et de nouveaux axiomes.

Exemple.
Ainsi, traduire la déclaration :

```
donné(point, a)
```

revient à ajouter à l'univers géométrique de base le symbole fonctionnel géométrique `a` tel que

```
arité(a) = []
coarité(a) = point
ambigu(a) = faux
févaluation(a) = nilf
décomposition(a) = []
```

La fonction d'évaluation `nilf` indique que la fonction d'évaluation sera choisie au moment de l'interprétation du programme de construction. En fait, le traitement des constantes est particulier : les attributs sélectionnés par les fonctions `arité`, `ambigu`, `févaluation` et `décomposition` ont toujours la même valeur et ne sont pas explicitement représentés pour les constantes.

Les déclarations utilisant le mot réservé `nomme` introduisent une nouvelle constante en relation avec d'autres existant déjà dans la figure : la traduction de `nomme(s, x, t)` permet de prendre en compte l'objet géométrique de nom `x` et de sorte `s` et la forme égalitaire `x = t`.

Ce genre de déclaration a été ajouté pour pouvoir nommer des objets particuliers qui auraient sans doute été créés par Progé mais avec des noms peu évocateurs.

Exemple.
Pour une déclaration comme :

```
nomme(point, o, centre(c))
```

le symbole de constante de nom `o` et de coarité `point` est ajouté à l'univers géométrique avec la forme égalitaire :

```
o = centre(c)
```

Ces ajouts à l'univers géométrique de base ne concernent que des constantes et des formes égalitaires. Ils se feront à travers la structure particulière que nous appelons figure formelle (cf. chapitre 3) et qui sera décrite plus précisément au chapitre suivant. A cette occasion, nous spécifierons formellement les opérations de traduction des déclarations d'un énoncé.

2.2. Contraintes

Les contraintes traduisent les relations entre les objets nommés par les déclarations. Ces contraintes sont principalement exprimées à l'aide des symboles de prédicat de l'univers géométrique. La spécification de la sorte `contrainte` se réduit donc à un constructeur de base comme on peut le voir à la figure 2. Et nous pouvons alors définir la sorte `énoncé` qui est simplement l'association d'une liste de déclarations et d'une liste de contraintes.

```
sortes
     contrainte
     énoncé

fonctions
     cont : term0 → contrainte
     en : déclaration* contrainte* → énoncé

précondition
pré(cont(t)) = t est un term0 construit avec un symbole de prédicat géométrique
```

figure 2 : Les sortes `contrainte` et `énoncé`

Exemple.
Nous pourrons ainsi avoir :

```
cont( iso(a, b, c))
cont( i incidentdp med(a, b))
cont( a '=p=' mil(b, c))
```

pour traduire que les trois points `a`, `b` et `c` forment un triangle isocèle, que `i` est sur la médiatrice de `[a, b]` et que `a` est le milieu de `[b, c]` .

Les contraintes de l'énoncé sont traduites de manière différente suivant qu'elles représentent des relations participatives, des relations d'égalité ou d'autres relations. Toutes les contraintes de l'énoncé sont retranscrites dans la structure particulière de l'univers géométrique que nous

appelons le raisonnement (cf. chapitre 3). Cette notion de raisonnement formel sera développée au chapitre suivant.

Une contrainte représentant une relation participative est détectée grace à l'attribut `ptitre` attaché à son foncteur principal (Cf. chapitre 4). Une telle contrainte est ajoutée sous forme de fait à la fois à la figure et au raisonnement et les relations réciproques sont également ajoutées à la figure.

Exemple.
La contrainte

```
cont(p incidentpd med(a, b))
```

est ajoutée au raisonnement sous une forme que nous détaillerons plus loin, et à la figure avec le fait supplémentaire

```
med(a, b) incidentdp p .
```

Les contraintes qui concernent des égalités sont reconnues comme telles à l'aide de leur attribut `ptitre`. Pour ce type de contraintes, non seulement le raisonnement est mis à jour, mais en plus, on complète la figure formelle en y ajoutant la forme égalitaire correspondante en tenant compte des axiomes automatiques d'implication.

Exemple.
La contrainte

```
cont(i '=p=' mil(a, b))
```

est ajoutée au raisonnement et permet de compléter la figure en cours avec les faits suivants :

```
i = mil(a, b)
i incidendpt dro(a, b)
dro(a, b) incidentdp i
```

Le fait d'avoir deux structures différentes (une figure formelle et un raisonnement formel) pour mémoriser les nouveaux axiomes semble imposer une certaine redondance comme dans les exemples précédents où l'on traduit de deux manières que `i` est le milieu de `[a, b]`. Nous verrons plus loin que,même s'ils ont la même signification, les deux symboles `=` et `'=p='` sont, en fait, exploités de manière différente par Progé.

Exemple.
L'axiome sous forme égalitaire `i = mil(a, b)` indique que dans cet univers particulier, les deux termes sont synonymes et interchangeable sous certaines conditions tandis que l'axiome `i '=p=' mil(a, b)` est une propriété géométrique qui est destinée à être exploitée par le moteur d'inférence de Progé et dont la signification est concentrée dans le symbole fonctionnel `mil` plutôt que dans la relation d'égalité. Nous reviendrons plus loin sur cette question.

2.3. Enoncés sous-contraints et énoncés sur-contraints

Un énoncé de construction *sous-contraint* est un énoncé qui possède -indépendamment des moyens de construction- une infinité de solutions. Un énoncé *surcontraint* ne donne lieu à aucune solution [Petersen 90].

Il est difficile de dire si un problème est bien posé avant d'en avoir la solution (ou de montrer qu'il n'existe pas de solution). Il existe des critères utilisant les notions de degré de liberté et degré de restriction qui reviennent à compter le nombre d'équations et d'inconnues [Roller & al 88]. Mais ceux-ci ne sont pas très fiables : une partie d'énoncé sur-contraint peut compenser le déficit en équations d'une autre partie sous-contrainte. Dans Progé, il n'y a aucun contrôle de ce type : l'utilisateur doit s'assurer que son problème est bien posé.

Un type de surcontrainte est cependant intéressant à noter : il s'agit des surcontraintes résultant d'une relation entre objets donnés.

Exemple.
Dans certains énoncés on voit :

"*Etant donnés deux droites parallèles ...*"
"*Etant donnés trois cercles de même rayon ...*"
"*Etant donné un triangle équilatéral ...*"

Ces débuts d'énoncés sont surcontraints car, dans le premier cas par exemple, si on se donne deux droites, elles ne sont en général pas parallèles. Pour avoir la configuration indiquée par l'énoncé avec des valeurs numériques, il faut savoir déjà résoudre le problème de la construction de cette configuration. Avec notre exemple, il faut savoir construire une droite parallèle à une autre droite donnée et vérifiant une autre contrainte pour que le problème soit bien posé. Formellement, Progé résout un problème contenant de telles surcontraintes (cf. [Schreck 92]), mais l'interprétation numérique de la solution formelle pose des problèmes que nous verrons à la section suivante.

3. La sorte programme de construction

3.1. Pourquoi un programme de construction ?

Dans l'idéal, résoudre formellement un programme de construction, ce serait trouver un terme[7], ou un système de termes "triangulaire" s'il y a plusieurs objets à construire, dont on peut démontrer qu'il satisfait aux conditions de l'énoncé. Cependant, l'existence de *fonctions partielles* en géométrie fait qu'on ne peut se contenter d'une telle forme : si la considération d'un univers géométrique permet d'éviter des termes comme `interdd(dro(a,b),dro(b,a))`, il reste des termes qui ne reçoivent pas d'interprétation dans des cas particuliers.

Exemple. Si `a` et `b` sont deux points donnés, le terme `dro(a, b)` a un sens sauf quand on attribue la même valeur numérique à `a` et `b`.

Par ailleurs, il arrive que lors d'une construction, on ait des choix à faire.

Exemple.

7La plupart des logiciels de construction géométrique explicites, comme Géophile [Braun 88] ou Cabri [Baulac 90] représentent ce terme sous forme d'un graphe ou d'un hypergraphe.

Dans l'exercice 'buthpll' donné en annexe, on utilise la règle (écrite ici en langue naturelle) :

si dist(A, B) = dist(C, D)
    et les droites AB et CD sont parallèles
    et les points A, B, C et D sont distincts
alors
    soit (A, B, C, D) est un parallèlogramme
    soit (A, B, D, C) est un parallèlogramme

Or, à moins d'une discussion soutenue, on ne sait pas déterminer a priori dans quel cas l'on se trouve, et l'on doit faire des hypothèses qui ne sont pas dans l'énoncé pour continuer la construction dans un cas ou dans l'autre.

Ainsi, de notre point de vue, une solution formelle à un problème de construction est un programme de construction contenant des structures de contrôle, comme des conditionnelles et de l'itération.

3.2. Description de la sorte programme de construction

Les programmes de construction géométrique décrits ici utilisent un langage de programmation a priori non destiné à un programmeur, d'où l'emploi d'une syntaxe simple et l'absence de facilités habituellement présentes dans les langages usuels. La génération des programmes de construction par Progé sera décrite dans les chapitres suivants.

Un programme est constitué d'une liste de déclarations suivi d'une liste d'instructions. Une déclaration consiste essentiellement en l'association d'un identificateur et d'une sorte géométrique. Un identificateur est dans la terminologie du chapitre précédent un symbole fonctionnel (cog) d'arité vide et de coarité la sorte géométrique indiquée dans la déclaration de cet identificateur. Il y a six types différents d'instructions : les définitions, les conditionnelles, les itérations, les combinaisons itération/conditions, les vérifications et instructuction particulière d'échec.

Une *définition* est de la forme

```
Id := T
```

où Id est un identificateur, dans notre cas un nom d'objet géométrique soit donné par l'énoncé, soit produit par le système, et T est soit un terme construit avec un symbole fonctionnel géométrique, soit le symbole donné.

Exemple.
On peut ainsi avoir :

```
r := donné
droite02 := med(o2, o3)
```

Le mode d'élaboration d'un programme par Progé fait que les termes sont de profondeur 1. Pour chacun des termes intervenant, il faut être assuré que lors d'une interprétation celui-ci aura un sens. C'est le rôle des instructions conditionnelles de garantir ceci.

Une *instruction conditionnelle* est de la forme

`si` <`Cond`> `alors` <`Bloc1`> `sinon` <`Bloc2`>

où `Cond` est une liste de termes formés à l'aide de symboles de prédicat géométrique ou à l'aide du symbole de relation binaire `diff` représentant l'inégalité, et `Bloc1` et `Bloc2` sont des listes d'instructions. Les instructions conditionnelles sont utilisées pour prendre en compte l'existence possible de cas dégénérés, c'est pourquoi nous avons introduit le symbole de prédicat `diff` désignant l'inégalité numérique.

Exemple.
Dans le programme de l'exemple du chapitre 3, nous avons ainsi les instructions :

```
si [a diff b] alors
      droite01 := dro(a, b)
            ...
sinon
      échec
...
```

Une *instruction itérative* est de la forme

`pour` <`Id1`> `dans` <`Id2`> `faire` <`Bloc`>

où `Id1` et `Id2` sont des identificateurs et `Bloc` est une liste d'instructions. Ce type d'instruction a été introduit pour tenir compte de l'existence de plusieurs solutions à l'interprétation numérique d'un terme. Nous avons en effet indiqué au chapitre 4 que nous ne manipulions pas formellement les intersections en tant qu'ensemble mais comme un seul objet. Nous avions alors indiqué les problèmes que cela pose lorsqu'on a besoin de distinguer les points d'une intersection et comment nous les avons partiellement contournés en enrichissant l'univers géométrique (Cf chapitre 4 section 3). Il reste qu'au moment de l'interprétation d'une solution formelle, il faut choisir dans l'ensemble des points possibles celui ou ceux qui conviennent. La solution que nous avons adoptée est d'interpréter un terme construit avec un symbole fonctionnel ambigu comme une liste de valeurs numériques et d'utiliser une instruction itérative pour poser un point de choix.

Exemple.
C'est ainsi que dans l'exemple de programme de construction donné au chapitre 3 nous avons les expressions suivantes :

```
list00 := intercd(cercle00, droite00)
pour point01 dans list00 faire <suite du programme>
```

La partie du programme, appelée ici <suite du programme>, sera interprétée deux fois. Chacune des interprétations correspondant à l'un des points d'intersection de `cercle00` avec `droite00`. L'interprétation de cette branche du programme échoue s'il n'y a aucun point dans cette intersection.

Une combinaison *itération/conditions* est de la forme

`pour` <`Idcas`> `dans` <`ListeCas`> `faire`
`si` <`Idcas`> `eg` <`Cas1`> `alors` <`Bloc1`>
`si` <`Idcas`> `eg` <`Cas2`> `alors` <`Bloc2`>
...
`si` <`Idcas`> `eg` <`Casn`> `alors` <`Blocn`>

où `Idcas` est un identificateur de sorte `cas`, `ListeCas` est une liste de termes relationnels (en fait des noms de cas suffiraient) contenant les termes `Cas1`, `Cas2`, ..., `Casn`, et `Bloc1`, `Bloc2`, ..., `Blocn` sont des listes d'instructions. Une telle instruction sert à traduire l'existence de cas particuliers (qui ne sont pas des cas dégénérés)

Exemple.
Il en est ainsi dans l'exercice `Buthpll` du chapitre 10 où le programme produit contient les instructions :

```
list01 := [[pll(point00, m, b, n)], [pll(point00, m, n, b)]]
pour cas00 dans list01 faire
      si cas00 eg [pll(point00, m, b, n)] alors
            ...
      si cas00 eg [pll(point00, m, n, b)] alors
            ...
```

L'utilisation des itérations et des combinaisons itération/conditions permet d'avoir potentiellement toutes les soltuions numériques dans une seule solution formelle.

Les *instructions de vérification* sont de la forme

`vérifier(<Contrainte>)`

où `Contrainte` est un terme relationnel (en général tiré de l'énoncé). L'utilité de telles instructions est de contrôler numériquement que la solution particulière trouvée vérifie bien les contraintes de l'énoncé. En effet, le fait de *raisonner par conditions nécessaires* peut faire qu'un programme de construction produire des fausses solutions c'est-à-dire des valeurs numériques ne vérifiant pas l'énoncé. Ceci se produit en particulier lors de la pose de points de choix où toutes les valeurs numériques ne font pas forcément partie d'une solution.

L'instruction `échec` signifie que Progé n'a pas trouvé de construction formelle au problème posé avec les hypothèses ajoutées par les conditionnelles. Ceci peut signifier qu'il n'y a pas de solution ou que Progé n'a pas su trouver une solution : les deux cas ne sont pas distingués à l'heure actuelle.

Nous récapitulons les différentes sortes introduites dans cette section en donnant une spécification de celle-ci à la figure 3. Cette spécification ne contient que les constructeurs de base et sera complétée à la section suivante.

```
sortes
      id < term0                                  % identificateur
      déclaration
      instr
      corps renomme instr*
      pcg                                         % programme de construction géométrique

fonctions
      cas : → sgeo                          % sorte géométrique particulière sans attribut

      dec : id sgeo → déclaration

      _ := _ : id term0 → instr
      si _ alors _ sinon _ : term0 corps corps → instr
```

```
si _ alors _ : term0 corps → instr
pour _ dans _ faire _ : id id corps → instr
vérifier : term0 → instr
echec : → instr

prog : déclaration* corps → pcg  % un pcg contient les déclarations de sorte
```

figure 3 : spécification de la sorte `pcg`.

3.3. Programmes bien définis

Les instructions de vérification mises à part, le graphe des calculs possibles d'un programme synthétisé par Progé a une structure arborescente : les noeuds simples (un seul descendant) sont les définitions et les itérations, les noeuds à embranchement sont les conditionnelles `si_alors_sinon_` et les combinaisons itération/conditions, et les arcs matérialisent la relation d'antériorité. Pour une instruction donnée dans un programme, on a donc un seul chemin partant de la racine et allant à cette instruction. A un endroit donné du plan, un identificateur `I` est défini s'il existe dans la partie précédente du plan une unique instruction qui est soit une instruction de définition où `I` apparaît à gauche du symbole := , soit une instruction d'itération où `I` apparaît à gauche du mot réservé `dans`. Un programme de construction est bien défini si tout identificateur apparaisant dans un terme ou à droite du mot réservé `dans` d'une itération est défini à cet endroit du plan.

Dans notre approche, la manière d'engendrer des programmes de construction assure que ceux-ci ont une forme arborescente, qu'ils sont bien définis et que tous les termes apparaissant dans des définitions sont au plus de profondeur 1.

4. Interprétation

Un programme de construction est destiné à être *interprété* pour fournir des solutions numériques à un énoncé instancié avec des valeurs fournies par l'utilisateur lors de l'interprétation. Actuellement, une *figure graphique* est construite au fur et à mesure de l'interprétation et les données numériques sont saisies directement sur cette figure graphique au moyen de la souris.

Un programme de construction calcule toutes les solutions possibles par retour arrière sur les points de choix. On est assuré que les solutions sont correctes par vérification numérique de l'énoncé avec les valeurs trouvées.

4.1. Contexte d'évaluation

L'interprétation d'un programme se fait -comme d'habitude-, à l'aide d'un *contexte d'évaluation* qui à chaque identificateur associe une valeur numérique. Pour avoir un traitement homogène, la notion de valeur numérique décrite ici est assez élastique. Dans le cas d'un identificateur `i` d'objet géométrique, la valeur numérique de `i` est la liste de coordonnées réelles correspondant à la sorte géométrique de l'identificateur. Dans le cas où `i` correspond à

une liste, c'est le cas où l'on évalue un terme ambigu, la valeur numérique de `i` est la liste des listes des coordonnées des interprétations possibles du terme en question. Dans le cas où `i` est un identificateur de cas alors la valeur "numérique" de celui-ci est une liste de termes prédicatifs définissant le cas en question. Et, finalement, si `i` est un identificateur correspondant à une liste de cas, sa valeur numérique est une liste de listes de termes prédicatifs. Notre contexte d'évaluation est simplement constitué d'une liste de couples, notés `ov`(identificateur, valeur numérique) que nous appelons des *objets valués*. Les figures numériques indiquées à la figure 1 du chapitre 3 et produites par un programme de construction ne sont que les contextes d'évaluation obtenus à la fin de la phase d'interprétation.

```
sortes
      val_num                                  % valeur numérique d'un identificateur
      ov                                       % objet valué
      contexte renomme ov*                     % contexte d'évaluation

fonctions
      vn : sgeo réel* → val_num          % valeur numérique d'un objet géométrique
      vnl : sgeo réel** → val_num              % valeur numérique d'une liste d'objets geom.
      vnc : term0* → val_num                   % valeur numérique d'un cas
      vncl : term0** → val_num           % valeur numérique d'une liste de cas
                                % projections élémentaires
      coord : val_num → réel*                  % coordonnées de la valeur numérique d'un
                                               % objet géométrique
      coordl : val_num → réel**          % idem pour une liste d'objets géométriques
      vcas : val_num → term0*                  % idem pour un cas
      vlcas : val_num → term0**          % idem pour une liste de cas
      typv : val_num → sgeo                    % sorte associée à une valeur numérique

      val : id val_num → ov                    % constructeur d'objets valués
      valeur : ov → val_num                    % valeur d'un objet valué
      nom : ov → id                      % nom d'un objet valué
```

figure 4 : contexte d'évaluation.

A la figure 4, nous donnons la signature de la spécification de l'interpréte qui précise les notions de *valeur numérique*, *objet valué* et *contexte d'évaluation* représentées respectivement par les sortes `val_num`, `ov` et `contexte`. Les fonctions `vn`, `vnl`, `vnc`, `vncl` et `val` sont les constructeurs de base des sortes `val_num` et `ov` tandis que les fonctions `coord`, `coordl`, `vcas`, `vlcas`, `typvn`, `typvnl`, `valeur` et `nom` sont les projections initiales de ces agrégats. Nous ne détaillons pas davantage ces fonctions élémentaires.

4.2. Evaluation d'un contexte

Le contexte d'évaluation sert à mémoriser les valeurs numériques déjà calculées. Les fonctions d'évaluation de termes d'un programme de construction utilisent ce contexte pour fournir la ou les valeurs d'un identificateur intervenant dans le terme à déterminer. Ces fonctions sont présentées à la figure 5.

```
fonctions                    % évaluation de termes dans un contexte d'évaluation
type : id déclaration* → sgeo                % sorte d'un identificateur de l'énoncé
eval : id contexte → val_num                 % recherche de la valeur d'un id dans un contexte
evall : id* contexte → val_num*              % idem pour une liste d'identificateurs
evalue : term0 contexte → val_num            % évaluation d'un terme dans un contexte
evalue_cond : term0 contexte → bool          % idem pour une condition
evalue_condl : term0* contexte → bool        % idem pour une liste de conditions

préconditions
pré(vn(s, lr)) = (longueur(lr) = deg_max(s))
pré(eval(i, c)) = c contient un objet valué de nom i
pré(evalue(t, c)) = t est un terme fonctionnel de profondeur 1 ou 'donné'
pré(evalue_cond(t, c)) = t est un terme relationnel
```

figure 5 : évaluation d'un contexte.

L'évaluation d'un terme se fait de manière différente si on a un identificateur, un terme composé avec un symbole fonctionnel géométrique ou un symbole de prédicat. Dans le premier cas, qui correspond à la fonction `evall`, il ne s'agit que de la recherche dans une liste d'une valeur numérique dont le nom correspond à l'identificateur cherché. Dans les deux dernier cas, on fait appel à l'univers géométrique de base qui n'est pas précisé dans les arguments des profils de la figure 5. Plus précisément, ces fonctions d'évaluation suivent les axiomes de la figure 6.

```
axiomes
type(I1, [dec(S, I2) | D]) = si I1 = I2 alors S sinon type(I1, D)
eval(i1, [val(i2, vnum) | c]) = si i1 = i2 alors vnum sinon eval(i1, c)
evall([], c) = []
evall([i | is], c) = [eval(i, c) | evall(is, c)]

evalue(f(larg), c) =
           si ambigu(f) = vrai
           alors vnl(coarité(f), applique(févaluation(f), evall(larg)))
           sinon vn(coarité(f), applique(févaluation(f), evall(larg))))

evalue_cond(f(larg), c) = applique(pévaluation(f), evall(larg)))
evalue_condl([], cx) = vrai
evalue_condl([c | lc], cx) = evalue_cond(c, cx) & evalue_condl(lc, cx)
```

figure 6 : des axiomes pour l'interprète

Remarques :

- sur l'opération `applique` : il faudrait définir correctement la sorte `fonction d'évaluation` pour pouvoir expliquer l'opération `applique`. Nous ne le ferons pas ici.

- si `f` est un symbole fonctionnel ambigu, la valeur rendu par `applique(févaluation(f), evall(larg))` est une liste de listes de coordonnées. Un telle liste servira à l'évaluation avec choix d'un identificateur dans une instruction d'itération.

- pour le symbole de prédicat `diff,` une fonction d'évaluation standard est proposée sur la base de la norme sup de $\mathbb{R}^n$.

4.3. Fonctionnement de l'interprète.

On peut enfin (!) expliquer le fonctionnement de notre interpréteur à travers les axiomes de la figure 7. La fonction `interprète` ne fait qu'appeler la fonction `inter` avec les arguments nécessaires qui sont la liste des déclarations, le corps du programme, le contexte d'évaluation courant (vide initialement) et la liste des figures numériques déjà évaluées (vide au départ également).

```
fonctions
interprète : pcg → contexte*                    % interpréte : la liste rendue est celle de
                                                % toutes les figures numériques
possibles
inter : déclaration* corps contexte contexte* → contexte*
                                                % fonction auxilliaire d'interprétation
inter_iter : déclaration* id val_num* corps contexte contexte* → contexte*
                                                % interprétation d'une itération avec
                                                % toutes les valeurs possibles pour l'id

préconditions
pré(interprète(p)) = p est un programme de construction arborescent, bien
défini et où tous les termes sont de profondeur au plus 1.

axiomes
interprète(prog(Decls, Cp)) = inter(Decls, Cp, [], [])
```

figure 7 : fonctionnement de l'interprète.

La fonction `inter`, interprète le corps du programme instruction par instruction : pour une instruction d'itération, la fonction `inter_iter` est appelée et toutes les valeurs de la liste sont examinées pour donner lieu au contexte correspondant qui, s'il réussit les instructions de vérification de la fin, est ajouté à la liste des figures numériques solutions. La figure 8 fournit les axiomes décrivant la fonction `inter`.

```
axiomes
inter(Decls, [], Cx, Lf) = [Cx | Lf]          % la sol. numérique trouvée est correcte et ajoutée
                                              % à celles déjà trouvées
inter(Decls, [O = donné | Cps], Cx, Lf) =
     inter(Decls, Cps, [val(O, applique(saisie(type(O, Decls)))) | Cx], Lf)

inter(Decls, [O = T | Cps], Cx, Lf) =
           si type(O, Decls) = cas
           alors inter(Decls, Cps, [val(O, T) | Cx], Lf)
           sinon inter(Decls, Cps, [val(O, evalue(T, Cx)) | Cx], Lf)

inter(Decls, [ si Cond alors Bloc1 sinon Bloc2 | Vérifs], Cx, Lf) =
           si evalue_condl(Cond, Cx) = vrai
           alors inter(Decls, Bloc1 + Vérifs, Cx, Lf)
           sinon inter(Decls, Bloc2 + Vérifs, Cx, Lf)

inter(Decl, [si Cas eg Cas1 alors Bloc | Instrs], Cx, Lf) =
           si eval(Cas) = Cas1
           alors inter(Decl, Bloc + Instrs, Cx, Lf)
           sinon inter(Decl, Instrs, Cx, Lf)

inter(Decls, [pour O dans L faire Bloc | Vérifs], Cx, Lf) =
           inter_iter(Decls, O, eval(L), Bloc + Vérifs, Cx, Lf)

%      avec pour inter_iter

inter_iter(Decls, O, vnl(S,[]), Bloc, Cx, Lf) = Lf
inter_iter(Decls, O, vnl(S, [Valn | Lvn]), Bloc, Cx, Lf) =
           inter_iter(Decls, O, vnl(S, Lvn), Bloc, Cx, Lfb)
           où Lfb = inter(Decls, Bloc, [val(O, Valn) | Cx], Lf)

inter(Decls, [vérifier(Cond) | Vérifs], Cx, Lf) =
           si evalue_cond(Cond, Cx) = vrai
           alors inter(Decls, Vérifs, Cx, Lf
           sinon Lf
inter(Decl, [échec | Vérifs], Cx, Lf) = Lf
```

figure 8 : des axiomes pour `inter`.

Rappelons que l'opération + associée à deux listes désigne la concaténation.

Exemple.
L'énoncé suivant

Enoncé. *Construire un triangle* `abc` *dont les sommets* `a` *et* `b` *sont donnés et qui soit rectangle et isocèle en* `a`.

se traduit en Progé par l'énoncé :

```
1 'dec:' point :: c cherche.
2 'dec:' point :: b donne.
3 'dec:' point :: a donne.
0 'cont:' dro(a, b) ortho dro(a, c).
1 'cont:' iso(a, b, c).
```

La résolution formelle de cet énoncé par Progé fournit le programme :

```
b := donné
a := donné
long00 := dist(a, b)
si [b diff a] alors
     droite00 := dro(b, a)
     dir01 := diro(droite00)
     droite01 := dpdir(a, dir01)
     cercle00 := ccr(a, long00)
     list00 := intercd(cercle00, droite01)
     pour c dans list00 faire
          fin
     fin
sinon
     echec
     fin
vérifier(dro(a, b) ortho dro(a, c))
vérifier(iso(a, b, c))
```

L'interprétation de ce programme se fait par l'appel, via la fonction `interprète`, de la fonction `inter` avec comme arguments les déclarations de l'énoncé, le programme ci-dessus et deux listes vides : l'une destinée à mémoriser le contexte courant et l'autre à mémoriser toutes les figures numériques solutions déjà trouvées.

La première instruction

```
b := donné
```

est examinée : il s'agit d'une définition avec `donné`. L'interprète cherche dans la liste des déclarations la sorte de `b`, c'est-à-dire `point`, et appelle la fonction de saisie associée à la sorte `point`. Celle-ci renvoie une valeur, par exemple (5, 0), obtenue par pointage à l'écran. Le contexte est alors `[val(b, vn(point, [5, 0]))]`. La deuxième instruction

```
a := donné
```

est ensuite interprétée de la même manière, et on a alors le contexte `[val(a, vn(point, [0, 0])), val(b, vn(point, [5, 0]))]`, par exemple.

La longueur `long00` est ensuite évaluée avec l'instruction

```
long00 := dist(a, b)
```

A cette occasion, l'interprète cherche dans le contexte les valeurs numériques de `a` et de `b` et trouve respectivement `vn(point, [0, 0])` et `vn(point, [5, 0])`. Ces valeurs sont fournies à la fonction d'évaluation attribut du symbole fonctionnel `dist`. Celle-ci retourne la valeur `vn(longueur, [5])` qui est ajoutée au contexte. On obtient `[val(long00, vn(longueur, [5])), val(a, vn(point,[0, 0])), val(b, vn(point, [5, 0]))]`.

L'instruction suivante est la conditionnelle :

```
si [b diff a] alors ...
```

La condition `b diff a` est évaluée en comparant la somme des valeurs absolues des différences des coordonnées avec une valeur fixe (actuellement dans Progé : 0.01). Ici |5-0| + |0-0| > 0.01, donc seule la branche `alors` est examinée. Le contexte n'est pas modifié.

Les instructions suivantes sont successivement évaluées jusqu'à l'instruction

```
list00 := intercd(cercle00, droite01)
```

L'évaluation de `intercd`, avec ces arguments, fournit une liste de valeurs, soit ici `vnl(point, [[0, 5], [0, -5]])` qui est associée avec l'identificateur `list00` : `val(list00, vnl(point, [[0, 5], [0, -5]]))`.

L'interprète arrive alors à l'instruction d'itération

```
pour c dans list00 faire
fin
```

et la fonction `inter_iter` est appelée avec les arguments suivants

```
inter_iter(<declarations>, c, vnl(point, [[0, 5], [0, -5]]),
[vérifier(dro(a, b) ortho dro(a, c)),vérifier(iso(a, b, c))], <contexte>, [])
```

où <`déclarations`> sont les déclarations de type du programme et <`contexte`> et le contexte courant déjà calculé. Cette fonction fait d'abord l'appel

```
inter(<declarations>,[vérifier(dro(a, b) ortho dro(a, c)),
vérifier(iso(a, b, c))], [val(c, vn(point,[0, 5]) | <contexte>], [])
```

qui consiste à exécuter les deux vérifications, ce qui se fait sans problème et produit une première solution que nous appellerons <`contexte1`> dans laquelle `c` a pour coordonnées (0, 5). Ensuite `inter_iter` fait un appel récursif à elle-même avec les arguments :

```
inter_iter(<declarations>, c, vnl(point, [[0, -5]]),
[vérifier(dro(a, b) ortho dro(a, c)), vérifier(iso(a, b, c))], <contexte>,
[<contexte1>])
```

qui réappelle `inter` avec pour valeur de `c`, cette fois, `vn(point(0, -5)`. Les vérifications sont effectuées et une solution, <`contexte2`>, est trouvée est ajoutée à [<`contexte1`>] pour rendre la liste [<`contexte2`>, <`contexte1`>]. La fonction `inter_iter` est appelée une dernière fois, avec comme liste d'itération (troisième argument), une liste vide, et elle se contente de retourner son dernier argument qui est la liste des contextes trouvés et vérifiés. Il n'y a plus d'autres appels et, finalement, la fonction `interprète` retourne bien ce qu'on lui demande.

Chapitre 6

Figure formelle

Sublimement dressé, l'espace d'un instant.
L'instant d'aprés, hélas ! cette folle figure
(Comme en convulsions) s'abîmait dans un gouffre,
Tandis qu'ils écoutaient, anxieux, haletants.

La notion de figure formelle a été introduite de manière intuitive au chapitre 3. Nous l'y avons décrite comme l'association d'un hypergraphe des dépendances fonctionnelles et d'un graphe d'incidence consignant les relations constructives -comme les relations d'incidence- entre des objets géométriques.

Par ailleurs, au chapitre 4, nous avons présenté la figure avec le raisonnement comme une partie d'un univers géométrique particularisé par un énoncé. Nous y avons d'ailleurs mis en place les outils pour décrire formellement la notion de figure : l'hypergraphe fonctionnel associé à une figure est une structure de données qui correspond à un ensemble de formes égalitaires, et le graphe d'incidence représente un ensemble de termes clos construits avec des symboles de relations participatives. C'est ce point de vue que nous allons développer dans ce chapitre.

1. Objets géométriques

Nous avons indiqué au chapitre précédent que les symboles fonctionnels désignant des constantes étaient ajoutés à partir d'un énoncé, et, représentés et traités de manière particulière dans l'univers géométrique. L'univers géométrique de base reste inchangé et les constantes sont introduites dans une *figure formelle* sous forme d'*objets géométriques* . La raison d'être de ces constantes est en effet différente de celle des symboles fonctionnels : il s'agit en particulier d'objets que l'on cherche à déterminer. Il est par conséquent utile de mémoriser les renseignements concernant l'état de construction d'un objet géométrique.

Un *objet géométrique* est donc un quadruplet noté `fe(nom, type, deglib, def)` où `nom` est un symbole de constante (le nom de l'objet géométrique), `type` est la sorte géométrique de la constante, `deglib` est un entier représentant le degré de liberté de l'objet dans la figure et `def`, *définition contextuelle*, est soit le symbole réservé `donné`, soit un terme clos de profondeur 1, soit une liste de couples `nom/titre` dans lesquels `nom` est un symbole de constante et `titre` est un symbole de relation participative.

Exemple.
En traduisant l'énoncé donné au chapitre 3, nous avons, avant résolution, les objets géométriques suivants :

```
fe(a, point, 0, donné)
fe(b, point, 0, donné)
fe(c, cercle, 0, donné)
fe(point00, point, 0, centre(c))
fe(long00, longueur, 0, rayon(c))
fe(d, droite, 1, [a/incidentpd])
fe(e, point, 1, [c/incidentcp])
fe(f, point, 1, [c/incidentcp])
fe(long01, longueur, 1, [])
```

La définition `donné`, correspond à un objet donné de l'énoncé, et donc construit et de degré de liberté nul.

La définition contextuelle d'un objet géométrique dans une figure est *complète* si c'est un terme clos de profondeur 1 tel que tous ses arguments soient des noms d'objets géométriques construits, c'est-à-dire de degré de liberté nul. Une définition contextuelle partielle est constituée d'une liste sans doublon de couples `nom/titre` où `nom` est le nom d'un objet géométrique contenu dans la figure et construit.

Exemple.
Avec l'exemple précédent : `donné` est la définition de `a`, `rayon(c)` est la définition (complète) de `long00`, et [`a/incidentpd`] est la définition partielle, à ce moment de la construction, de la droite `d`.

Le *degré de liberté* d'un objet géométrique dans une figure est calculé en relation avec la définition contextuelle de l'objet. Au lieu de considérer toutes les relations auquelles l'objet prend part, nous ne considérons que les relations participatives qui seules permettent de faire progresser la construction de l'objet. Ainsi, dans Progé, le degré de liberté d'un objet géométrique non encore construit est-il calculé par la formule

$$\texttt{deg\_lib} = \texttt{deg\_max(sorte)} - \Sigma\, \texttt{deg\_restr}(p_i)$$

où p_i est une relation participative intervenant dans la définition contextuelle de l'objet. Lorsque ce degré de liberté devient nul, une définition complète de l'objet est calculée.

2. Opérations de base

La figure reflète les dépendances fonctionnelles entre les différents objets géométriques introduits par l'énoncé ou générés par Progé. Ceci se fait à travers des axiomes (sans variables) qui sont soit des formes égalitaires, soit des termes construits avec un symbole de relation participative. Les opérations de base sur une figure sont données aux figures 1, 2 et 3. Dans les profils proposés ici, il faudrait ajouter systématiquement l'univers géométrique de base aux arguments. Nous ne l'avons pas fait pour les mêmes raisons qu'au chapitre 4.

La figure 1 expose la signature correspondant aux sortes `symog`, `représentant` et `participant`. La sorte `symog` (symbole d'objet géométrique) représente les symboles de constantes. Les sortes `représentant`, `participation` et `participant` ont été présentées au chapitre 3. Rappelons que si on a une forme égalitaire de la forme `nom =`

$f(o_1, ..., o_n)$ où $f(o_1, ..., o_n)$ est un terme clos de profondeur 1, on dit que $f(o_1, ..., o_n)$ est un *représentant* de nom. Une *participation* est un terme clos de profondeur 1 construit avec un symbole de relation participative, et, si on a une participation $p(o_1, o_2)$, on dit que o_1 est un *participant* de o_2 *au titre* p, on note o_1/p ce participant.

Exemple.
A la figure 4 du chapitre 3, nous avons, par exemple, les participations

```
e incidentpd d
a incidenpd d
long00 rayonde c
```

C'est-à-dire que e/incidentpd et a/incidentpd sont des participants de d, et long00/rayonde est un participant de c.

Les fonctions données à la figure 1 pour ces sortes ne sont que les constructeurs de base et les projections élémentaires. Signalons cependant que la fonction auxiliaire associe_titre est l'équivalent de l'opération _/_ pour des listes.

```
sortes
      symog < term0                          % constante géométrique
      représentant < term0                   % terme fonctionnel clos de profondeur 1
      participation < term0                  % terme clos de prof. 1 construit avec une rel. part.
      participant                    % "demi-participation"

fonctions
                                             % fonctions concernant la sorte participant
      _ / _ : symog tp → participant                 % constructeur de base
      nom : participant → symog              % projection
      titre : participant → tp                       % projection
      associe_tire : symog* tp* → participant*       % comme / pour des listes
```

figure 1 : constantes, représentants et participant

La figure 2 précise les notions de définition contextuelle et d'objet géométrique exposées à la section 1. Les fonctions présentées dans cette signature ne sont que les opérations de base. Seule la fonction p2c mérite d'être détaillée.

```
sortes
      définition                                  % définition contextuelle partielle ou complète
      og                                          % objet géométrique

fonctions
                                                  % fonctions concernant la sorte définition
      dv renomme []                                     % définition vide
      dt : représentant → définition              % "casting"
      comp : définition → bool                          % définition complète ?
      p2c : sgeo définition → définition                % déf. partielle → déf. complète
                                                  %fonctions concernant les objets géométriques
(og)
      fe : symog sgeo entier définition → og      % constructeur de base
      nom : og → symog                                  % projection élémentaire
      type : og → sgeo                                  % idem
      deg : og → entier                                 % idem
      def : og → définition                             % idem
      construit : og → bool                             % degré de liberté nul ?
```

figure 2 : sortes `définition` et `og`

L'opération `p2c` est déclenchée lorsque l'objet géométrique concerné vient d'avoir un degré de liberté nul. Il s'agit alors de fabriquer la définiton complète d'un point à partir de la définition partielle. Nous n'allons pas décrire formellement cette opération, mais nous illustrons son utilisation sur un exemple.

Exemple
Lorsqu'on doit transformer pour un point `o`, une droite `d` et un cercle `c`, la définition partielle `[d/incidentdp, c/incidentcp]`, il faut d'abord chercher dans l'univers de base tous les symboles fonctionnels décomposanbles de coarité `point`, c'est-à-dire dans l'univers standard de Progé : `interdd, intercd, interdc, intercc`, puis leur décomposition :

```
décomposition(interdd) = [incidentdp, incidentdp],
décomposition(intercd) = [incidentcp, incidentdp],
décomposition(interdc) = [incidentdp, incidentcp],
décomposition(intercc) = [incidentcp, incidentcp]
```

Progé cherche enfin à faire coïncider les titres de participations contenus dans la définition partielle avec l'une des décompositions candidates : deux solutions sont trouvées, l'une correspond à `intercd` et l'autre à `interdc`, i.e. `intercd(c, d)` et `interdc(d, c)`. La première solution trouvée par Progé est conservée et l'autre n'est pas essayée (une définition suffit). On a finalement, pour cet exemple :

```
p2c(point, [d/incidentdp, c/incidentcp]) = intercd(c, d)
```

Le succès de l'opération dépend bien entendu de la complétude de l'univers géométrique en ce qui concerne les symboles fonctionnels décomposables et les relations participatives : pour toute définition partielle induisant un degré de liberté nul, il doit exister un symbole fonctionnel dont la décomposition correspond à la définition partielle. Si ça n'est pas le cas, il peut y avoir discordance entre le degré de liberté mémorisé et un degré de liberté calculé avec une définition partielle. Seul le degré de liberté mémorisé est alors pris en compte.

Les fonctions de base concernant les objets géométriques se résument au constructeur `fe` et au projections `nom`, `type`, `deg`, `def`. La fonction `construit` indique si un objet possède une définition complète ou non. La mention de degré de liberté semble redondante et pourrait être calculée à l'aide d'une définition partielle : nous avons vu plus haut qu'outre éviter les calculs, on peut ainsi esquiver certains problèmes si l'opération `p2c` sur les définitions ne parvient pas à constituer une définition complète.

A la figure 3, nous exposons les fonctions élémentaires concernant les notions de relation de synonymie et de figure formelle. L'utilisation d'une table de synonymes dans une figure formelle a été justifiée au chapitre 3.

```
sortes
      syn                                          % relation de synonymie
      figure                                       % figure formelle

fonctions
                                                   % fonctions concernant la relation de synonymie
      _ alias _ : symog symog → syn                % constructeur de base
                                                   % fonctions de base concernant la figure formelle
      fv : → figure                                % figure vide
      aog : og figure → figure                     % ajout d'un objet géométrique
      rog : og figure → figure                     % retrait d'un objet géométrique
      aeg : fegal figure → figure                  % ajout d'une forme égalitaire
      reg : fegal figure → figure                  % retrait d'une forme égalitaire
      apt : participation figure → figure          % ajout d'une participation
      rpt : participation figure → figure          % retrait d'une participation
      asy : syn figure → figure                    % ajout d'une occurrence de la rel. de syn.
      rsy : syn figure → figure                    % retrait d'une occurrence de la rel. de syn.

      aogp : og figure → figure                    % ajout d'un objet avec propagation des
                                                   % degré de liberté nuls
péconditions
pré(aog(O, Fig)) = il n'existe pas d'objet géométrique de même nom que O dans Fig
pré(rog(O, Fig)) = O est contenu dans Fig
pré(aeg(Feg, Fig)) = Feg n'est pas contenue dans Fig
pré(reg(Feg, Fig)) = Feg est contenue dans Fig
pré(apt(Part, Fig)) = Part n'est pas contenue dans Fig
pré(rpt(Part, Fig)) = Part est contenue dans Fig
pré(asy(Syn, Fig)) = Syn n'est pas contenue dans Fig
pré(rsy(Syn, Fig)) = Syn est contenu dans Fig
```

figure 3 : opérations de base sur la sorte `figure`

Nous n'allons pas ajouter à cette énumération fastidieuse les axiomes classiques qui correspondent à ces fonctions. Rappelons simplement qu'il s'agit des opérations de base : lors des adjonctions, il n'y a aucun contrôle de redondance. Ces contrôles s'appuient sur des opérations de recherche dans la figure, opérations que nous allons détailler à la section 3. Cependant, une opération un peu plus sophistiquée se distingue des autres : il s'agit de la fonction `aogp` qui ajoute un objet géométrique dans une figure en propageant les degrés de liberté nuls pour un objet construit : lorsqu'un objet construit est ajouté à la figure, tous les objets qui dépendent de lui, soit par le biais d'une forme égalitaire, soit par le biais de

participations, sont mis à jour. C'est cette fonction qui devra être utilisée systématiquement pour ajouter un objet à une figure formelle.

Exemple.
La traduction de l'énoncé proposé au chapitre 3 donne une figure `F` que nous présentons ici. Pour cause de lisibilité, la présentation de cette figure ne suit pas la syntaxe imposée par les fonctions décrites par la figure précédente.

Objets ajoutés à la figure avec la fonction `aog` *:*

```
fe(a, point, 0, donné)
fe(b, point, 0, donné)
fe(c, cercle, 0, donné)
fe(point00, point, 0, centre(c))
fe(long00, longueur, 0, rayon(c))
fe(d, droite, 1, [a/incidentpd])
fe(e, point, 1, [c/incidentcp])
fe(f, point, 1, [c/incidentcp])
fe(long1, longueur, 1, [])
```

Formes égalitaires ajoutées avec la fonction `aeg` :

```
point00 = centre(c)
long00 = rayon(c)

long01 = dist(f, b)
long01 = dist(e, b)
```

Participations ajoutées avec la fonction apt :

```
a incidentpd d

f incidentpd d
f incidentpc c

e incidentpd d
e incidentpd c

c incidentcp e
c incidentcp f
c apourrayon long00
c apourcentre point00

 d incidentdp e
d incidentdp f
d incidentdp a
```

Couples de synonymes ajoutés avec la fonction asy :

```
a alias a
b alias b
c alias c
point00 alias point00
long00 alias long00
d alias d
e alias e
f alias f
longO1 alias long01
```

Remarquons que dans ce cas précis, la table des synonymes manque d'interêt car aucune égalité entre objets géométriques n'a été découverte.

3. Opérations de recherche

Lors d'opérations de recherche dans une figure, il faut tenir compte de l'existence de synonymes et d'équivalences entre termes. La figure 4 propose une vue d'ensemble des fonctions de comparaison de termes et de recherche dans une figure. Les fonctions les plus importantes sont reprises et détaillées dans les sous-sections suivantes.

```
fonctions
                                                    % fonctions concernant la table des synonymes
      sypri : symog figure → bool                   % retourne le synonyme privilégié du nom donné
      synonymes : symog symog figure → bool         % les deux noms sont synonymes ?
      synonyml : symog figure → symog*              % retourne la liste des synonymes du nom donné

      équivalents : représentant représentant figure → bool     % voir texte

      rep : symog figure → représentant*            % renvoie la liste des représentants d'un nom
d'og.
      part : symog figure → participant*            % renvoie la liste des participants
      nom_og : symog figure → og                    % retourne l'og correspondant au nom donné

      rep_og : représentant figure → symog ∪ {ϖ}    % recherche d'un objet géom.
                                                    % à l'aide d'un représentant

      contenup : participation figure → bool        % la participation est dans la figure ?
      contenue : fegal figure → bool                % la forme égalitaire est dans la figure ?
      contenunom : symog figure → bool              % le nom donné correspond à un og de
la                                                  % figure ?
      construit : symog figure → bool               % le nom donné est-il celui d'un objet
                                                    % construit
      construit : représentant figure → bool        % idem pour un représentant
```

figure 4 : opérations de recherche

3.1. Synonymes

Les trois premières opérations concernent l'interrogation de la table des synonymes pour des symboles d'objets géométriques. La gestion de cette table est telle que pour chaque classe, il existe un élément distingué rendu par la fonction `sypri` : c'est-à-dire que deux noms `n1` et `n2` sont synonymes dans la figure courante si et seulement si `sypri`(n1) = `sypri`(n2). Les fonctions `synonymes` et `synonyml` sont alors immédiates.

3.2. Termes équivalents

La fonction `equivalents` indique si deux termes sont égaux, aux permutations d'arguments et à la relation de synonymie près. Par exemple, si les points `a` et `b` sont synonymes dans la figure `F`, on aura `équivalents(mil(a, c), mil(c, b), F) = vrai.` Cette opération ne s'applique qu'aux termes non décomposables : des mécanismes particuliers utilisant les participations sont mis en oeuvre pour les termes décomposables. Les

figure 5 et 6 donnent une description complète de cette opération. Les fonctions auxiliaires utlisées sont :

- `dls` qui indique si un terme figure dans une liste, aux synonymes près,
- `equiv_syn` qui indique si deux termes sont équivalents modulo les synonymes
- `equiv_synl` qui compare deux listes de termes suivant le principe précédent.

```
fonctions
      équivalents : représentant représentant figure → bool          % rappel ...
                                  %  fonctions auxiliaires servant à définir l'opération équivalents
      dls : term0 term0* figure → bool
      equiv_syn : term0 term0 figure → bool
      equiv_synl : term0* term0* figure → bool
```

figure 5 (début) : profil des fonctions définissant `équivalents`

```
préconditions
      pré(équivalents(R1, R2, F) = R1 et R2 sont des termes non décomposables et
            les constantes présentes dans R1 et R2 sont des noms d'objets
            géométriques contenus dans la figure F
      pré(dls(T1, LT, F)) = le terme T1 et ceux contenus dans LT sont clos et
      les constantes sont des noms d'objets géométriques de la figure F
      pré(equiv_syn(T1, T2, F)) = les termes T1 et T2 sont clos et les constantes
            sont des noms d'objets géométriques de la figure F
      pré(equiv_synl(LT1, LT2, F)) = les termes contenus dans les listes LT1 et
            LT2 sont clos et les constantes sont des noms d'objets géométriques
            de la figure F
```

figure 5 (fin) : préconditions des fonctions définissant `équivalents`

La figure 6 expose les axiomes définissant les mécanismes de ces différentes fonctions.

```
axiomes
équivalents(T1, T2, F) = dls(T1, term_equiv(T2), F)

dls(T1, [], F) = faux
dls(T1, [T2 | LT], F) = si equiv_syn(T1, T2, F) = vrai
                        alors vrai
                        sinon equiv_synl(T1, LT, F)

equiv_syn(T1, T2, F) = si cste(T1) = vrai
                        alors si cste(T2) = vrai
                              alors synonymes(T1, T2, F)
                              sinon faux
                        sinon si fonctf(T1) = fonctf(T2)
                              alors equiv_synl(args(T1), args(T2), F)
                              sinon faux

equiv_synl([], [], F) = vrai
equiv_synl([T1 | LT1], [T2 | LT2], F) = equiv_syn(T1, T2, F) & equiv_synl((LT1,
LT2, F)
```

figure 6 : spécification de la fonction `équivalents`

On pourra noter que les fonctions auxiliaires ont un champ d'action plus général que nécessaire : il aurait suffit ici que les arguments soient des représentants, i.e. des termes clos de profondeur 1. Cette généralisation ne demandant que peu de travail intellectuel nous nous sommes permis de présenter cette version.

3.3. Recherches, représentants et participants

Les opérations `rep`, `part`, et `nom_og`, respectivement de recherche de représentants, de participants et d'objets géométriques, exposées à la figure 4 ne posent pas de problèmes particuliers : pour la première, il suffit de parcourir l'ensemble des formes égalitaires contenues dans la figure et de comparer, modulo la relation de synonymie, leur membre de gauche avec le nom fourni en argument pour récupérer la liste des représentants cherchés. La même démarche est appliquée pour les deux opérations suivantes.

Exemple.
Avec la figure `F` donnée en exemple à la section 2, nous avons :

```
rep(d, F) = []
rep(point00, F) = [centre(c)]
rep(long01, F) = [dist(f, b), dist(e, b)]
```

et

```
part(d, F) = [a/incidentpd, e/incidentpd, f/incidentpd]
part(c, F)= [point00/apourcentre, long00/apourrayon, e/incidentpc,
                                                    f/incidentpc]
```

La fonction `rep_og` est un peu plus délicate : la recherche d'un symbole d'objet géométrique par un représentant dans la figure fait intervenir la fonction `équivalents` présentée plus haut si le représentant n'est pas décomposable. Dans le cas contraire, les participations sont examinées. La figure 7 donne une description complète de cette fonction primordiale.

```
fonctions
rep_og : représentant figure → symog ∪ {ϖ}                    % rappel ...
                                          % fonctions auxiliaires servant à définir rep_og
repd_og : représentant figure → symog ∪ {ϖ}        % repr. décomposable
repi_og : représentant figure figure → symog ∪ {ϖ}        % repr. non décomp.
                                          % fonctions auxiliaires pour repd_og
nomdob : figure → symog*                                      % retourne les noms d'og de la fig
chdlc : représentant symog* figure → symog ∪ {ϖ}          % auxiliaire de repd_og
décomposition : représentant → participant*                   % déc. d'un repr. décomposable
inclu_syn_titre : participant* participant* → bool        % Cf. texte
efface_syn_titre : participant participant* → participant* ∪ {ϖ}     %Cf. texte

axiomes
rep_og(Rep, Fig) = si décomposable(fonct(Rep)) = vrai
                   alors repd_og(Rep, Fig)
                   sinon repi_og(Rep, Fig, Fig)
```

```
repi_og(Rep, fv, Fig) = ϖ
repi_og(Rep, aeg(Nom = Repb, F), Fig) = si équivalents(Rep, Repb, Fig) = vrai
                                        alors Nom
                                        sinon repi_og(Rep, F, Fig)

% pour tout autre constructeur de base Φ de la sorte figure :
repi_og(Rep, Φ(a1, ..., an, F), Fig) = repi_og(Rep, F, Fig)

repd_og(Rep, Fig) = chdlc(Rep, nomdob(Fig), Fig)
% où nomdob retourne la liste des noms d'objets géométriques compris dans la figure

chdlc(Rep, lv, Fig) = ϖ
chdlc(Rep, [Nom1 | Lnoms], Fig) =
           si inclu_syn_titre(décomposition(Rep), part(Nom1, Fig), Fig)
           alors Nom1
           sinon chdlc(Rep, Lnoms, Fig)
```

figure 7 (début) : recherche d'un nom d'`og` à partir d'un représentant

```
décomposition(Rep) = associe_titre(args(Rep), décomposition(fonctf(Rep))

inclu_syn_titre(lv, Lpart, Fig) = vrai
inclu_syn_titre([Nom1/Titre1 | Lat], Lpart, Fig) =
                        si L = ϖ
                        alors faux
                        sinon inclu_syn_titre(Lat, L, Fig)
           où L = efface_syn_titre(Nom1/Titre1, Lpart, Fig)

efface_syn_titre(Nom1/Titre1, lv, Fig) = ϖ
efface_syn_titre(Nom1/Titre1, [Nom2/Titre2 | Lpart], Fig) =
           si Titre1 = Titre2 & synonymes(Nom1, Nom2, Fig)
           alors Lpart
           sinon [Nom2 / Part2 | efface_syn_titre(Nom1/Titre1, Lpart, Fig)]
```

figure 7 (fin) : recherche d'un nom d'`og` à partir d'un représentant

La recherche par un représentant indécomposable est le résultat de la fonction `repi_og`. Pour un représentant décomposable `Rd`, cette recherche se fait à l'aide de la fonction `repd_og`. La fonction `chdlc` cherche parmi la liste des noms d'objets géométriques contenus dans la figure si il existe un nom `N` tel que la décomposition de `Rd` est incluse dans la liste des participants de `N` comme ceci est expliqué à la figure 7.

Exemple.
On veut chercher dans la figure `Fig`, donnée à la section 2, le cercle de centre `point00` et passant par `f`, c'est-à-dire un nom d'objet géométrique correspondant à `ccp(point00, f)`. On a

`rep_og(ccp(point00, f), Fig) = repd_og(ccp(point00, f), Fig)`

puisque `ccp` est un symbole fonctionnel décomposable. On cherche donc parmi tous les noms d'objets géométriques celui qui convient :

`chdlc(ccp(point00, f), [a, b, c, point00, ...], Fig)`

En sautant quelques étapes, on s'aperçoit que les points `a` et `b` ne conviennent pas et qu'il faut essayer avec le cercle `c`. On doit donc vérifier si

```
inclu_syn_titre([point00/apourcentre, f/incidentpc], [point00/apourcentre,
            long00/apourrayon, e/incidentpc, f/incidentpc], Fig) = vrai
```

le premier argument résultant de `décomposition(ccp(point00, f))` et le second de `part(c, Fig)`. Cette vérification se fait sans problème en utilisant la fonction `efface_syn_titre`. Et le nom `c` est finalement retourné par la fonction `rep_og`.

Remarque : on améliore beaucoup l'efficacité de cette recherche en ne considérant dans `chdlc` que les noms d'objets géométriques qui sont de la même sorte que le représentant donné.

Les fonctions `contenue`, `contenup`, `contenunom` vérifient respectivement qu'une forme égalitaire, qu'une participation et qu'un symbole d'objet géométrique sont contenus dans la figure. Les deux fonctions `construit` vérifient qu'un objet géométrique ou un terme fonctionnel est construit dans la figure. Pour la définition de ces fonctions, il n'y a pas de problèmes particuliers, il faut cependant noter que ces recherches se font modulo les synonymes et les permutations possibles des arguments dans le cas de `contenue`.

4. Adjonctions contrôlées

Les opérations de recherche décrite à la section précédente permettent la définition de fonctions gérant de manière plus fine la sorte `figure`. La figure 8 en donne une vue d'ensemble en présentant les profils de ces opérations d'adjonction.

```
fonctions
      ajobj : og figure → figure                    % ajout d'un objet géométrique
      ajpart : participant og figure → og           % ajout d'un participant à un objet géom.
      ajpt : participation figure → figure          % ajout d'une participation à une figure
      ajptl : participation* figure → figure        % idem pour une liste
      ajeg : fegal figure → figure                  % ajout d'une forme égalitaire à une figure

préconditions
pré(ajpart(Nom/Titre, O, Fig)) = construit(Nom, Fig)
pré(ajeg(N = T, Fig)) = N est un nom d'objet géométrique et T est un représentant
```

figure 8 : opérations d'adjonctions fines à une figure

La fonction `ajobj` ajoute simplement un objet à la figure en vérifiant qu'il n'existe pas déjà un objet géométrique ayant pour nom un synonyme de celui de l'objet géométrique donné en argument.

4.1. Adjonction d'un participant à une définition

La fonction `ajpart` ajoute un nouveau participant à la définition partielle d'un objet géométrique donné en argument (Cf. figure 9). La définition résultant de cette adjonction est éventuellement transformé en définition complète.

```
fonctions
ajpart : participant og figure → og          % rappel ...
                                  % fonctions auxiliaires servant à définir ajpart
ajpartbis : participant définition figure → définition
deg_lib : définition sgeo → entier           % degré de liberté d'une définition
deg_restl : définition → entier              % degré de restriction d'une définition partielle

axiomes
ajpart(Nom1/Titre, fe(Nom2, Type, Deg, Def), Fig) =
          si Deg = 0 alors fe(Nom2, Type, Deg, Def)
                    sinon si deg_lib(NDef, Type) = 0
                          alors fe(Nom2, Type, 0, p2c(Type, NDef))
                         sinon fe(Nom2, Type, deg_lib(NDef, Type), NDef)
          où NDef = ajpartbis(Nom1/Titre, Def, Fig)
ajpartbis(Nom/Titre, dv, Fig) = ajp(Nom/Titre, dv)
ajpartbis(Nom/Titre, ajp(Nom1/Titre1, Def), Fig) =
          si synonymes(Nom, Nom1, Fig)= vrai ∧ Titre = Titre1
                alors Def
                sinon ajp(Nom1/Titre1, ajpartbis(Nom/Titre, Def)

deg_lib(Def, S) = deg_max(S) - deg_restl(Def)
deg_restl(dv) = 0
deg_restl(ajp(Nom/Titre, Def) = deg_rest(Titre) + deg_restl(Def)
```

figure 9 : description de la fonction `ajpart`.

La fonction `deg_lib` calcule un nouveau degré de liberté pour l'objet géométrique dont la définition est modifiée. Si celui-ci est nul, la définition partielle est transformée en définition complète (Cf. figure 9). Si cette transformation échoue, l'ancien degré de liberté est conservé ainsi que la définition partielle obtenue aprés adjonction du participant. Pour la transformation d'une définition partielle en définition complète, le lecteur pourra se reporter à l'exemple de la section 2.

4.2. Adjonction d'une participation à une figure.

La fonction `ajpt` ajoute une participation à la figure en s'assurant qu'elle n'y existe pas encore. La participation réciproque calculée est également ajoutée à la figure grâce à la fonction `ajrecip`. Enfin, si le premier argument de la participation est construit dans la figure, l'objet géométrique correspondant au deuxième argument est mis à jour à l'aide de la fonction `ajpart`. La figure 10 explique précisément le fonctionnement de cette fonction à l'aide des fonctions auxiliaires `ajrecip`, `ajrecipp` et `ajrecipf` ajoutant les relations réciproques. La fonction `ajptl` ajoute successivement les participations d'une liste à la figure : de facture classique nous ne la détaillerons pas ici.

```
fonctions
ajpt : participation figure → figure          % rappel
                                  % fonctions auxiliaires servant à décrire la fonction ajpt
ajrecip : participation figure → figure       % ajout de la participation ...
ajrecipp : participation figure → figure      % ... dans le cas où c'est une relation
ajrecipf : participation figure → figure      % ... dans le cas où c'est une fonction
```

```
préconditions
pré(ajrecipp(Part, Figure)) = recipp(fonctp(Part)) \= nilp
pré(ajrecipf(Part, figure)) = recipp(fonctp(Part)) = nilp

axiomes
ajpt(Part, Fig) =
            si contenup(Part, Fig)
            alors Fig
            sinon si construit(arg1(Part), Fig)
                  alors ajrecip(Part, apt(Part, aogp(Nog, rog(Ao, Fig))))
                  sinon ajrecip(Part, apt(Part, F))
            où Ao = nom_og(arg2(Part), Fig)
              Nog = ajpart(arg1(Part)/fonctp(Part), Ao, Fig)
ajrecip(Part, Fig) = si recipp(fonct(Part)) = nilp
                        alors ajrecipf(Part, Fig)
                        sinon ajrecipp(Part, Fig)
ajrecipp(Part, Fig) = si contenup(Trap, Fig)
                  alors Fig
                  sinon si construit(arg1(Trap), Fig)
                        alors apt(Trap, aogp(Nog, rog(Ao, Fig)))
                        sinon apt(Trap, Fig)
                  où Trap = recipp(fonctp(Part))(arg2(Part), arg1(Part))
                     Ao = nom_og(arg2(Trap, Fig))
                     Nog = ajpart(Trap, Ao, Fig)
```

figure 10 (début) : ajout de participations à une figure.

```
ajrecipf(Part, Fig) = si contenue(Eg, Fig)
                  alors Fig
                  sinon si construit(arg2(Part), Fig)
                        alors si construit(arg1(Part), Fig)
                              alors aeg(Eg, Fig)
                              sinon aeg(Eg,  aogp(Nog, rog(Ao, Fig)))
      où Eg = (arg1(Part) =recipf(fonctp(Part))(arg2(Part)))
        Ao = nom_og(arg1(Part), fig)
        Nog = fe(nom(Ao), type(Ao), 0, dt(recipf(fonctp(Part))(arg2(Part))))
```

figure 10 (fin) : ajout de participations à une figure.

4.3. Adjonction d'une forme égalitaire à une figure

La fonction `ajeg` tente d'ajouter une forme égalitaire `N = T` à une figure comme suit :
- si le représentant `T` désigne un objet géométrique de la figure dont le nom `Np` n'est pas un synonyme de `N`, alors les deux objets géométriques de nom `N` et `Np` sont fusionnés et la table des synonymes est mise à jour,
- sinon si `T` n'est pas décomposable, la forme égalitaire est ajoutée avec éventuellement mise à jour de la définition de l'objet géométrique si celui-ci n'est pas encore construit et si tous les arguments de `T` sont construits. Par ailleurs, les opérations de mise à jour automatiques données en attribut du symbole fonctionnel sont mises en oeuvre.
- sinon `T` est décomposé à l'aide de la fonction `décomposition` donnée plus haut et les participations correspondantes sont ajoutées à la figure par applications successives de la fonction `ajpt`.

Exemples.
Dans la figure `Fig` donnée à la section 2, on veut ajouter la forme égalitaire

`long00` = `dist(point00, e)`

Le représentant `dist(point00, e)` n'est pas décomposable. On commence par chercher si le représentant `dist(point00, e)` désigne déjà un objet dans la figure. Ça n'est pas le cas. Par ailleurs, `long00` représente un objet déjà construit qu'il ne faut donc pas mettre à jour. Finalement, la forme égalitaire est ajoutée telle quelle sans autres incidences sur la figure.

Supposons maintenant que cette même figure contienne la droite `droite01` médiatrice de `[e, f]`. On veut ajouter la forme égalitaire

`droite01` = `dro(b, point00)`

Le représentant `dro(b, point00)` est décomposable et ne correspond encore à aucun objet géométrique de la figure. Il est donc décomposé et chacune des participations et sa réciproque sont ajoutées à la figure qui va ainsi contenir :

`b incidentpd droite01`
`point00 incidentpd droite01`
`droite01 incidentdp b`
`droite01 incidentdp point00`

Ceci se traduit par la spécification de la figure 8 :

```
fonctions
ajeg : fegal figure → figure                          % rappel ...
                                % fonctions auxiliaires servant à définir la fonction ajeg
complete_fig : term0 figure → figure                  % complète la fig. avec les maj. autom.
complete_figl : term0* figure → figure                % idem, pour une liste de relations
fabpart : symog participant* → participation*         % fabrique des participations

préconditions
pré(complete_fig(T, F)) = T est un terme clos construit avec un symbole
                  relationnel d'égalité ou d'incidence
% même genre de chose pour complete_figl

axiomes
ajeg(Nom = Rep, Fig) =
           si Nomp = ϖ
           alors si L = []
                  alors si construit(Nom, Fig)
                         alors complete_figl(Lmaj, aeg(Eg, Fig))
                         sinon si construit(Rep, Fig)
                                alors complete_figl(Lmaj, aeg(Eg, Figbis))
                                sinon complete_figl(Lmaj, aeg(Eg, Fig))
                  sinon ajptl(fabpart(Nom, L), Fig)
           sinon si synonymes(Nom, Nomp, Fig)
                  alors Fig
                  sinon fusion(Nom, Nomp, Fig)
     où  L = décomposition(Rep)
         Nomp = rep_og(Rep, Fig)
         Figbis =  aogp(Nog, rog(Ao, Fig)))
         Ao = nom_og(Nom, Fig)
         Nog = fe(nom(Ao), type(Ao), 0, dt(Rep))
```

```
        Lmaj = apsl(fi(prem(Lmajv), Nom = Rep), concl(Lmajv))
        Lmajv = maj(fonctf(Rep))

complete_fig(Prop, Fig) = si ptitre(Prop) = incid
                          alors ajpt(Prop, Fig)
                          sinon ajeg(Prop, Fig)
complete_figl([], Fig) = Fig
complete_figl([Prop | L], Fig) = complete_figl(L, complete_fig(Prop, Fig))

fabpart(Nom, []) = []
fabpart(Nom, [Nom1/Titre1 | Lp]) = [ Titre1([Nom1, Nom]) | fabpart(Nom, Lp)]
```

figure 8 : Adjonction d'une forme égalitaire

5. Création de nouvelles constantes, atomisation.

La plupart des opérations que nous venons de décrire, notamment les opérations de recherche, prennent en compte des représentants, c'est-à-dire des termes clos de profondeur 1. Le fait de ne considérer que de tels termes permet beaucoup de simplifications quant à la gestion de la figure, mais il faut s'assurer que tous les termes introduits par des sources "extérieures" -comme lors de la traduction d'un énoncé ou de l'application d'une règle- sont sous cette forme. Ceci est assuré par une opération dite d'*atomisation* qui transforme tout terme `T` en un nom d'objet géométrique possédant comme représentant `Rep` construit sur `T` en atomisant tous ses arguments (Cf. figure 12). Ceci permet de "normaliser" les termes contenus dans la figure et de traiter d'une certaine manière le problème de l'égalité entre objets géométriques.

```
fonctions
     gen_nom : figure sgeo → figure symog              % génération d'un nouveau nom
     atomvar : figure term0 → figure symog   % atomisation avec création d'un og
     atominst : figure term0 symog → figure % atomisation sans création
```

figure 12 : opérations d'atomisation

La première fonction `gen_nom` engendre un nom d'objet géométrique dépendant de la sorte géométrique donné et n'existant pas encore dans la figure.

Exemple.
Dans une certaine figure `fig1`,

```
gen_nom(fig1, point) = fig2, point04
```

produit une nouvelle figure `fig2` qui contient un objet géométrique de nom `point04`. Plus précisément, `fig2` se déduit de `fig1` en lui adjoignant d'une part l'objet géométrique `fe(point04, point, 2, dv)` et d'autre part une entrée dans la table des synonymes à l'aide du terme `point04 alias point04`. Nous ne décrirons pas davantage cette fonction.

5.1. Atomisation avec création d'un nouvel objet géométrique

La fonction `atomvar` utilise cette fonction de la manière exposée à la figure 13.

```
fonctions
```

```
atomvar : figure term0 → figure symog          % rappel ...
atomvarl : figure term0* → figure symog*       % comme atomvar pour une liste de termes

axiomes
atomvar(Fig, Term) = si cste(Term)
            alors Fig, Term
            sinon si rep_og(Terma, Fi) =
                  alors ajeg(Nom = Terma, aogp(fe(Nom, Typ, Deg, dv), Fi)), Nom
                              où Typ = coarité(fonctf(Term))
                                 Nom = gen_nom(Fig, Typ)
                                 Deg = deg_max(Typ)
                                 Fi, Larga = atomvarl(Fig, args(Term))
                                 Terma = fonctf(Term)(Larga)
                  sinon Fig, rep_og(Terma, Fi)
atomvarl(Fig, []) = Fig, []
atomvarl(Fig, [Term | Lt]) = Fi, [Nom | La]
     où Fib, La = atomvarl(Fig, Lt)
        Fi, Nom = atomvar(Fib, Term)
```

figure 13 : spécification de `atomvar`

Un exemple d'utilisation de cette fonction a été donné au chapitre 3.
Nous dirons qu'un terme clos `tc` est *représenté* dans la figure `fig` si l'atomisation de `tc` par la fonction `atomvar` laisse la figure inchangée, i.e. si

`atomvar(fig, tc) = fig, nom`

où `nom` est un symbole d'objet géométrique correspondant à `tc`.

5.2. Atomisation sans création

La fonction `atominst` tente de résoudre une égalité entre un terme fonctionnel `T` clos quelconque et un nom d'objet géométrique `N` en atomisant les arguments du terme `T` pour donner le terme `Ta`. Si `Ta` est effectivement un représentant de N dans la figure initiale, alors elle reste inchangée, sinon la figure est modifiée par les atomisations successives.

```
axiome
atominst(Fig, Term, Nom) = ajeg(Nom = Terma, Fi)
                  où Fi, Larga = atomvarl(Fig, args(Term))
                     Terma = fonctf(Term)(Larga)
```

figure 14 : spécification de `atominst`

La spécification donnée à la figure 14 est particulièrement simple : tout le travail est fait par les fonctions `ajeg` et `atomvar`.

6. Filtrage modulo une figure.

Les opérations de filtrage sont essentielles pour pouvoir appliquer des règles géométriques. Comme nous l'avons indiqué auparavant, la notion de filtrage classique n'est pas très adaptée au problème des constructions géométriques [Buthion 79] : à titre d'exemple, nous avons

fourni au chapitre 4 une version améliorée de la notion de filtrage en tenant compte des permutations possibles des arguments d'un terme non décomposable. Nous reprenons à présent formellement la description informelle du filtrage modulo une figure, que nous avons introduit lors de la présentation de Progé au chapitre 3.

```
fonction
     unif : figure term0 term0 → su*

préconditions
pré(unif(Fig, Term1, Term2)) = Term2 est un terme clos représenté dans la figure
```

figure 15 : filtrage modulo une figure

La fonction `unif` trouve toutes les substitutions filtrant *modulo la figure* le premier terme par le deuxième qui doit être clos. Autrement dit,

si σ appartient à `unif(fig, t1, t2)`, on a : $\sigma(t1) \sim_{fig} t2$.

La relation d'équivalence $\sim_{fig}$ étant elle-même définie sur l'ensemble des termes représentés dans la figure par :

`t1` $\sim_{fig}$ `t2` si et seulement si `atomvar(fig, t1) = fig, nom1`,
`atomvar(fig, t2) = fig, nom2`
et `synonymes(fig, nom1, nom2)`

Chercher toutes les substitutions possibles est source de complication mais ceci est nécessaire pour simuler la remontée profonde en Prolog. La figure 16 donne une description formelle de cette opération.

```
fonctions
unif : figure term0 term0 → su*                    % rappel ...
                                % fonctions auxiliaires servant à définir la fonction unif
unifbis : figure term0 term0 → su*                 % voir texte plus bas ...
unifl : figure term0* term0* → su*                 % voir texte plus bas ...
unif_wl : figure term0* term0* → su*               % voir texte plus bas ...
unif_wl_equiv : figure term0 term0* → su*          % voir texte plus bas ...
unifiell : figure su* term0* term0* → su*          % voir texte plus bas ...
                                % fonction utilitaire sur les listes d'objets de sorte T
concll : T* T** → T**                              % concatène la première liste avec chacune
                                                   % des listes du deuxième argument
préconditions
pré(unifibis(F, T1, T2)) = composé(T1) et composé(T2)
pré(unifl(F,L1,L2)) = (long(L1) = long(L2)) et (L1 \= lv)

axiomes
unif(Fig, Term1, Term2) =
     si variable(Term1)
     alors [[<Term1, Term2>]]
     sinon si cste(Term2)
           alors si cste(Term1)
                 alors si synonymes(Term1, Term2, Fig)
```

```
                        alors [[]]
                        sinon []
                  sinon si décomposition(Term1) = []
                        alors unif_wl(Fig, Term1, rep(Term2, Fig))
                        sinon  unif_wl_equiv(Fig,Term1,drep(fonctf(Term1),  Term2,
Fig))
            sinon si fonctf(Term1) = fonctf(Term2)
                  alors unif_wl_equiv(Fig, Term1, équivalents(Term2, F))
                  sinon []

unifbis(F, T1, T2) =  si fonctf(T1) = fonctf(T2)
                  alors unifl(F, args(T1), args(T2))
                  sinon []

unifl(F, [T1 | L1], [T2 | L2]) = si L1 \= []
                        alors unifiell(F, unif(F,T1,T2), L1, L2)
                        sinon unif(F, T1, T2)

unifiell(F, [], L1, L2) = []
unifiell(F, [S | Ls], L1, L2) =
                  concll(S, unifl(F, apsl(S, L1), L2)) + unifiell(F, Ls, L1, L2)

unif_wl(F, T1, []) = []
unif_wl(F, T1, [T2 | L]) = unif(F, T1, T2) + unif_wl(F, T1, L)

unif_wl_equiv(F, T1, []) = []
unif_wl_equiv(F, T1, [T2 | L]) = unifbis(F, T1, T2) + unif_wl_equiv(F, T1, L)
```

figure 16 : description de l'opération `unif`

Si le principe de la fonction `unif` est assez simple, cette description formelle est fastidieuse. Expliquons en quelques mots le rôle des fonctions auxiliaires utilisées :
- `unifbis` réalise l'unification de deux termes fonctionnels composés sans considérer les permutations possibles des arguments,
- `unifl` rend les substitutions correspondant au filtrage des deux listes (la deuxième ne devant contenir que des termes clos), après chaque terme, les substitutions trouvées sont appliquées au reste de la première liste,
- `unif_wl` rend la concaténation de toutes les listes de substitutions trouvées en filtrant le premier terme avec chacun des termes de la liste,
- `unif_wl`_equiv fait la même chose, mais sans passer par les équivalents,
- `unifiell` réalise la réunion des substitutions trouvées lors du filtrage en utilisant toutes les substitutions trouvées précédemment.

Exemple.
Dans la figure donnée en exemple, voici le filtrage de `dist(A, B)` par `long01`, avec quelques raccourcis, notamment le passage des conditionnelles, qui ne devraient cependant pas en gêner la compréhension.

```
unif(Fig, dist(A, B), long01)
= unif_wl(Fig, dist(A, B), [dist(f, b), dist(e, b)])
= unif(Fig, dist(A, B), dist(f, b)) + unif(Fig, dist(A, B), dist(e, b))
= unif_wl_equiv(Fig, dist(A, B), [dist(f, b), dist(b, f)])
      + unif_wl_equiv(Fig, dist(A, B), [dist(e, b), dist(b, e)])
```

prenons la première liste de la concaténation :

```
unif_wl_equiv(Fig, dist(A, B), [dist(f, b), dist(b, f)])
= unifbis(Fig, dist(A, B), dist(f, b)) + unifbis(Fig, dist(A, B), dist(b, f))
= unifl(Fig, [A, B], [f, b]) + unifl(Fig, [A, B], [b, f])
```

prenons, à nouveau, la première liste de la concaténation :

```
unifl(Fig, [A, B], [f, b])
= unifiell(Fig, unif(Fig, A, f), [B], [b])
= unifiell(Fig, [[<A, f>]], [B], [b])
= concll([<A, f>], unifl(Fig, apsl([<A, f>]], [B]), [b]))
                                                    + unifiell(Fig, [], [B], [b])
= concll([<A, f>], unifl(Fig, apsl([<A, f>]], [B]), [b]) + []
= concll([<A, f>], unifl(Fig, [B], [b])
= concll([<A, f>], unif(Fig, B, b)
= concll([<A, f>], [[<B, b]])
= [[<A, f>, <B, b>]]
```

De manière semblable, on obtient les autres substitutions, et on obtient finalement :

```
unif(Fig, dist(A, B), long01)
= [[<A, f>, <B, b>], [<A, b>, <B, f>], [<A, e>, <B, b>], [<A, b>, <B, e>]]
```

Ce qui est bien ce que l'on désirait.

7. Exemple : initialisation d'une figure à l'aide d'un énoncé

Les opérations que nous venons de décrire ont une utilité évidente dans la définition d'une figure formelle. A titre d'exemple, nous allons décrire une opération laissée en suspens au chapitre 5, à savoir l'interprétation d'un énoncé en une figure initiale.

7.1. Interprétation des déclarations

La traduction des déclarations qui ne sont pas construite avec `nomme` se fait en ajoutant l'objet géométrique avec la définition indiquée : les objets donnés ont pour définition `donné` et un degré de liberté nul, les objets cherchés ou mentionnés recoivent une définition vide et un degré de liberté égal au degré de liberté de la sorte concernée.

Exemple.
Les déclarations suivantes

```
donné(point, a)
cherche(droite, d)
```

sont respectivement traduites en ajoutant à la figure les objets géométriques

```
fe(a, point, 0, donné)
fe(d, droite, 2, [])
```

et les relations de synonymie

```
a alias a
d alias d
```

La traduction d'une déclaration de la forme `nomme(s, n, t)` doit ajouter non seulement un nouvel objet géométrique de nom `n` et de sorte `s`, mais également une forme égalitaire qui correspond à `n = t`. L'objet `fe(n, s, deg_max(s), [])` est ajouté à la figure. Puis, dans cette figure, la fonction `atominst` est appelée pour :
- mettre le terme `t` sous forme d'un représentant, avec éventuellement des répercussions sur la figure ;
- ajouter la forme égalitaire voulue ;`
- mettre à jour l'objet géométrique initialement ajouté à la figure.

Exemple.
La déclaration

```
nomme(point, o, centre(c))
```

ajoute dans un premier temps, l'objet géométrique

```
fe(o, point, 2, [])
```

à la figure courante pour donner la figure `Fig`. Puis la fonction `atominst` est appelée sous la forme

```
atominst(Fig, centre(c), o)
```

Ceci provoque l'ajout à `Fig` de la forme égalitaire

```
o = centre(c)
```

Et, dans l'hypothèse où le cercle `c` est donné, l'objet géométrique précédent est retiré de la figure pour être remplacé par l'objet géométrique

```
fe(o, point, 0, centre(c))
```

7.2. Interprétations des contraintes

Du point de vue de la figure, seules les contraintes ayant trait à l'égalité ou à une relation participative sont prises en compte.

Pour une contrainte d'égalité, de la forme `égalité(t1, t2)`, le traitement est le suivant : le terme `t1` est atomisé dans la figure initiale `Fig1`, à l'aide de `atomvar(Fig1, t1)` pour obtenir un nom `n1`, la figure `Fig1` a éventuellement été modifiée en `Fig2` si `t1` ne correspondait pas à un objet celle-ci. Puis `t2` est atomisé dans `Fig2` par `atominst(Fig2, t2, n1)`. Ceci a pour effet de placer dans la figure tous les termes mentionnés dans l'égalité sous forme d'objets géométriques et d'ajouter toutes les formes égalitaires souhaitables. S'il se trouve que `t1` et `t2` correspondent tous deux à des objets géométriques de `Fig1`, ces deux objets sont fusionnés et la table des synonymes est mise à jour.

Pour une relation participative de la forme `particip(t1, t2)`, les deux termes `t1` et `t2` sont atomisés à l'aide de `atomvar`, donc avec modification éventuelle de la figure initiale, pour donner les noms d'objet géométrique `o1` et `o2`. Ensuite la participation `particip(o1, o2)` est ajoutée à la figure ainsi que la participation réciproque. Lors de cette opération, les objets géométriques de nom `o1` et `o2` sont éventuellement mis à jour si l'un d'eux est construit. Ceci est assuré par la fonction `ajpt`.

Chapitre 7

Raisonnement, application des règles

« C'est un Snark ! » tel fut le son qui d'abord frappa
Leurs oreilles, semblant, pour être vrai, trop beau.
Puis ce fut un torrent de rires, de hourras,
Puis, de mauvais augure, les mots : « C'est un Boo...»

La figure géométrique est une manière de mémoriser les renseignements constructifs trouvés par Progé. Les autres propriétés démontrées dans l'univers géométriques sont également mémorisées sous forme d'un *raisonnement formel*. Ce chapitre approfondit cette notion introduite au chapitre 3. Par ailleurs, nous montrons comment le moteur d'inférence utilise les règles sur le raisonnement et la figure pour faire progresser la construction.

1. Introduction

Le raisonnement formel est dans Progé l'équivalent de la base de faits dans les systèmes experts classiques : les faits déduits à l'aide des règles y sont mémorisés et servent à leur tour à déclencher d'autres inférences. Comme nous considérons des règles d'ordre 1, les mécanismes de contrôle sont évidemment un peu plus compliqués que dans le cas d'ordre 0 ou 0+ : on ne peut ainsi pas "brûler" des règles qui pourraient encore s'appliquer à bon escient. Il est également dangereux de ne plus considérer dans la suite les faits ayant déjà servi à déclencher une règle. Diverses stratégies classiques ont été proposées dans le cadre général de la démonstration automatique (Cf. [Delahaye 87], [Chang & Lee 73] ou [Hayes-Roth & al 83] par exemple).

Dans le cadre des constructions géométriques, M. Buthion a décrit dans Geom3 [Buthion 79] des mécanismes de contrôle basés sur les notions intuitives de *richesse* d'une relation géométrique et de degré de liberté. L'historique des déductions prend la forme d'une forêt explorée en largeur. Chaque arbre correspond aux déductions possibles à partir d'une seule relation donnée par l'énoncé. L'exploration d'un arbre est arrêtée lorsque le nombre de degrés de liberté supprimés grâce à lui est égal à la richesse de la relation dont il est issu. Cette structuration impose aux règles utilisées de n'avoir qu'une seule prémisse ce qui est limitatif. Par ailleurs, dans Geom3, il semble qu'on puise réutiliser certains arbres, déjà brûlés, mais M. Buthion n'indique pas sur quels critéres le logiciel procède à cette réactivation.

Dans notre approche, la base de faits se présente sous forme d'un hypergraphe que nous appelons raisonnement formel (Cf. chapitre 3). Dans la version actuelle de Progé, l'exploration du raisonnement se fait en considérant le degré d'utilisation des propriétés déjà déduites : les propriétés les moins employées sont utilisées en priorité. Cette exploration est modulée par la prise en compte du nombre de degrés de liberté que les propositions ont contribué à

supprimer. Les phénomènes de bouclage sont évités en vérifiant l'existence d'une propriété dans le raisonnement modulo la figure.

Les règles font partie de l'univers géométrique de base et à ce titre une description statique en a été donnée au chapitre 4. La section 3 de ce chapitre décrit l'application des règles par le moteur d'inférence en précisant leur action sur la figure et le raisonnement. Les règles considérées dans ce chapitre sont des *règles simples* dont nous rappelons la syntaxe :

```
     si <liste de termes relationnels> et <liste de vérifications>
     alors <liste de termes>
```

En réalité, dans Progé, on considère deux autres types de règles. Leur syntaxe et leur mise en oeuvre ne seront précisés qu'au chapitre suivant pour éviter de compliquer inutilement notre propos.

2. Description statique de la sorte raisonnement

2.1. Sommet

Intuitivement, un sommet du raisonnement représente une propriété de la configuration donnée par l'énoncé. Certains renseignements supplémentaires sont mémorisés à des fins de contrôle. Nous notons en particulier le nombre de degrés de liberté que la propriété a contribué à supprimer et que nous appelons le nombre d'*utilisations actives*. En effet, des propriétés ayant déjà fortement servi à construire la figure doivent être utilisées en dernier. Pour les mêmes raisons, nous associons à un sommet le nombre d'utilisations d'une propriété.

Formellement, c'est un quadruplet noté `som(num, prop, nba, nbp)` où `num` est un entier servant à identifier le sommet dans un raisonnement, `prop` est un terme relationnel clos de profondeur 1 que nous appelons une *propriété atomisée*, `nba` est le *nombre d'utilisations actives* du sommet, `nbp` est un entier appelé *nombre d'utilisations passives* et indiquant le nombre de fois où le sommet a été utilisé dans le raisonnement, i.e. le nombre de successeurs dans l'hypergraphe. Ceci est résumé à la figure 1 par la donnée des fonctions élémentaires de la sorte `sommet`.

```
sorte
     sommet                                          % sommet du raisonnement formel

fonctions
som : entier term0 entier entier → sommet            % constructeur de base
sommet_num : sommet → entier                         % projection élémentaire
sommet_prop : sommet → term0                         % idem
sommet_nba : sommet → entier                         % idem
sommet_nbp : sommet → entier                         % idem
```

figure 1 : sorte `sommet` (du raisonnement formel)

Exemple.

Dans l'exemple donné au chapitre 3, le raisonnement formel contient, à un instant de la construction, les sommets :

```
som(5, long01 =l= long01, 0, 1)
som(6, iso(b, f, e), 0, 2)
```

Le premier sommet correspond à une égalité entre les représentants de `long01`. Cette notation surprenante de la propriété est due à son atomisation avant de la placer dans le raisonnement. L'utilisation du filtrage modulo une figure permet de retrouver toutes les égalités possibles entre les représentants de `long01` lors de l'application d'une règle (Cf. plus bas dans ce chapitre). A cet instant de la construction, cette propriété a été utilisé une fois par le moteur d'inférence et n'a servi à supprimer aucun degré de liberté.

Le second sommet indique que "l'isocélicité" du triangle (`b, f, e`) est une propriété de la configuration donnée par l'énoncé, et que cette propriété a été utilisée deux fois par Progé mais pas pour participer à la construction d'un objet de la figure.

2.2. Dérivation

Intuitivement, une dérivation est l'occurrence d'une règle appliquée lors de la construction. Formellement, c'est un quadruplet noté `deriv(num, prem, conc, nregle)` où `num` est un entier servant à identifier la dérivation, `prem` est la liste des identificateurs entiers des propriétes ayant satisfait les prémisses de la règle identifiée par l'entier `nregle` et `conc` l'identificateur entier d'une proposition conclusion de ladite règle. La figure 2 précise les opérations élémentaires concernant la sorte `dérivation`.

```
sorte
     dérivation                                          % dérivation du raisonnement

fonctions
deriv : entier entier* entier  entier → dérivation       % constructeur de base
deriv_num : dérivation → entier                          % projection élémentaire
deriv_prem : dérivation → entier                         % idem
deriv_conc : dérivation → entier                         % idem
deriv_regle : dérivation → entier                        % idem
```

figure 2 : sorte `dérivation`

Exemple.
Voici une dérivation tirée de l'exemple du chapitre 3 :

```
deriv(1, 6, 5, 102).
```

Celle-ci signifie que de la propriété numérotée 6 dans le raisonnement, Progé a déduit la propriété numéro 5 à l'aide de la règle 102.

2.3. Raisonnement.

Un raisonnement est un hypergraphe constitué d'un ensemble de sommets et d'un ensemble de dérivations. Les fonctions proposées à la figure 3 décrivent les fonctions élémenatires régissant les incorporations de sommets et de dérivations dans un raisonnement. En plus des fonctions classiques d'adjonction, de retrait et de test d'appartenance, nous avons deux

fonctions, `gnum_som` et `gnum_rais`, servant à engendrer des numéros de sommet et des numéros de dérivation qui n'existent pas encore dans le raisonnement.

```
sortes
     raisonnement                                          % raisonnement formel

fonctions
rv : → raisonnement                                        % raisonnement vide
asom : sommet raisonnement → raisonnement                  % ajout d'un sommet
rsom : entier raisonnement → raisonnement                  % retrait d'un sommet
ader : dérivation raisonnement → raisonnement              % ajout d'une dérivation
rder : entier raisonnement → raisonnement                  % retrait d'une dérivation
approp : term0 raisonnement figure → bool                  % appartenance d'une prop. à un raison.
appder : entier* entier raisonnement → bool                % idem pour une dérivation
                                        % Opérations auxiliaires
gnum_som : raisonnement → entier raisonnement              % nouveau num. de sommet
gnum_deriv : raisonnnement → entier raisonnement           % nouveau num. de dérivation
app1_syn : term0 term0* figure → bool                      % utilisée par approp
term_pequiv : term0 figure → term0*                        % termes relat. équivalents

préconditions
pré(som(N, Prop, Na, Np)) = Prop est un terme relationnel clos de profondeur 1
pré(asom(Som, Rais)) = Som n'est pas encore présent dans Rais
pré(rsom(Num, Rais)) = il existe un sommet identifié par Num dans Rais
pré(ader(Der, Rais)) = Der n'est pas contenue dans Rais
pré(rder(Num, Rais)) = il existe une dérivation identifiée par Num
pré(approp(Prop, Rais, F)) = Prop est un terme relationnel clos de profondeur 1

axiomes
% nous ne décrivons ici que la fonction approp qui teste l'appartenance d'un sommet à un raisonnement.
approp(Prop, rv, Fig) = faux
approp(Prop, ader(Ded, Rais), Fig) = approp(Prop, Rais, Fig)
approp(Prop2, asom(som(Num, Prop1, Na, Np), Rais), Fig) =
     si app1_syn(Prop1, term_pequiv(Prop2), Fig) = vrai
     alors  vrai
     sinon approp(Prop2,  Rais), Fig)
app1_syn(Term, [], Fig) = faux
app1_syn(Term1, [Term2 | Lt], Fig) =
     si fonctp(Term1) = fonct(Term2) et
        synonyml(args(Term1), args(Term2), Fig) = vrai
     alors vrai
     sinon app1_syn(Term1, Lt, Fig)
```

figure 3 : signature de la sorte `raisonnement`

La plupart des fonctions de la figure 3 sont de facture classique, c'est pourquoi nous n'y précisons que le fonctionnement de la fonction `approp` uniquement. En particulier, nous ne détaillons pas le fonctionnement de la fonction `term_pequiv` lequel se déduit directement de la fonction `term_equiv` décrite au chapitre 4.

Ces fonctions de base sont utilisées pour définir les fonctions `insom`, pour insérer un sommet dans le raisonnement à partir d'une propriété et `insar`, pour insérer une dérivation à partir d'une clause de Horn et d'un numéro de règle. Ces fonctions sont précisées à la figure 4.

```
fonctions
insom : term0 raisonnement figure → raisonnement
insar : entier* entier entier raisonnement → raisonnement

préconditions
pré(isom(Prop, R, F)) = Prop est un terme relationnel clos de profondeur
1
pré(app1_syn(Prop, Lp, Fig)) = Prop et les termes contenus dans Lp sont
                               des termes relationnels clos de profondeur 1

axiomes
insom(Prop, Rais, Fig) =
           si approp(Prop, Rais, Fig)
           alors Fig
           sinon asom(som(gnum_som(Rais), Prop, 0, 0), Rais)

insar(Lnat, Nat, Numreg, Rais) =
           si appder(Lnat, Nat, Rais)
           alors Rais
           sinon ader(deriv(gnum_der(Rais), Lnat, Nat, Numreg), Rais)
```

figure 5 : axiomes pour la spécification de la sorte `raisonnement`

2.4. Recherches élémentaires dans un raisonnement

Les fonctions de recherche décrites à la figure 6 peuvent être utilisées par le moteur d'inférence. Il s'agit de faire des recherches dans le raisonnement en suivant différents critères comme le nombre d'utilisations passives ou actives d'une propriété. D'autres fonctions de recherche sont envisageables pour permettre d'implanter d'autres stratégies que celles décrites à la section suivante.

```
fonctions
rech_crg_abs : crg raisonnement → sommet*
rech_rais_crg : crg entier entier raisonnement → sommet*
rech_rais_act : entier raisonnement → sommet*
rech_rais_pas : entier raisonnement → sommet*
rech_rais_act_pas : entier  entier raisonnement → sommet*
```

figure 6 : recherches dans un raisonnement

Nous passons en revue, brièvement ces fonctions de recherche :

- `rech_crg_abs` rend la liste des sommets dont la propriété est formée à l'aide du symbole relationnel donné en argument ;
- `rech_rais_crg` rend la liste des sommets dont la propriété est formée à l'aide du symbole relationnel donné en argument, et dont les nombres d'utilisations actives et passives sont inférieurs aux valeurs données ;
- `rech_rais_act` rend la liste des sommets dont le nombre d'utilisations actives est inférieur à la valeur donnée ;
- `rech_rais_pas` fait la même chose que rech_rais_act en considérant le nombre d'utilisations passives ;

- `rech_rais_act_pas` fait la même chose que les deux opérations précédentes en considérant les deux sortes d'utilisation des sommets.

Le fonctionnement de ces opérations de recherche est extrèmement classique et nous ne les approfondirons pas.

2.5. Sorte configuration

Une configuration dans un univers géométrique de base est défini par l'association d'une figure et d'un raisonnement formels. La figure 7 en précise les constructeurs de base et les projections élémentaires. Cette notion sera utilisée à la section suivante.

```
sorte
     configuration

fonctions
config : figure raisonnement → configuration
fig : configuration → figure
rais : configuration → raisonnement
```

figure 7 : description de la sorte `configuration`

3. Description du moteur d'inférence

Si la base de faits est un peu particulière (figure, raisonnement), le fonctionnement de notre moteur d'inférence est, dans son principe, traditionnel : une règle est choisie, puis instanciée, si c'est possible, à l'aide du raisonnement et de la figure. Lorsque cette phase réussit, le raisonnement et la figure sont mis à jour à l'aide des propriétés déduites. Cette section en précise le fonctionnement.

3.1. Vue d'ensemble du moteur d'inférence

La figure 8 définit l'ossature du moteur d'inférence avec une stratégie particulière correspondant à la fonction `essai_règle`. Elle consiste à sélectionner les règles dans l'ordre où elles sont écrites et à tenter de les appliquer en utilisant dans le raisonnement les propriétés les moins utilisées : plus exactement toutes les règles sont essayées en ne considérant dans un premier temps que les propriétés qui n'ont jamais été utilisées, puis, en cas d'échec, la liste des règles est à nouveau parcourue en examinant les propriétés qui ont été utilisées au plus une fois, etc. Nous ne savons pas répondre à la question de l'efficacité de cette stratégie. Elle peut d'ailleurs être facilement remise en cause, et la programmation d'une autre stratégie n'a que peu d'incidence sur les autres modules de Progé.

La fonction `moteur` est la fonction de plus haut niveau organisant les appels à la fonction `essai_regle` appelant elle-même les fonctions auxiliaires `essai_regle_niv` et `essai_regles_niv`. Diverses fonctions auxiliaires sont également mentionnées à la figure 8 comme les fonctions `tab_act` et `tab_pas` qui indiquent jusqu'à quel niveau d'utilisations actives et passives les propriétés servant à instancier les règles doivent être cherchées dans le raisonnement. La fonction `applique` qui est le coeur du moteur d'inférence tente d'appliquer

une règle donnée à l'aide de la configuration courante et en fonction des nombres d'utilisations actives et passives donnés. Cette fonction va être décrite plus précisément à la section 3.3.

```
fonctions
moteur : ug configuration pcg → configuration pcg
essai_regle : ug configuration pcg → configuration pcg bool
applique : règle entier entier configuration pcg → configuration pcg bool
                                        % fonctions auxiliaires ...
essai_regle_niv : ug configuration pcg entier → configuration pcg bool
essai_regles_niv : ug regles* configuration pcg entier → configuration pcg bool
fini : configuration → bool             % la construction est achevée ?
niv_max : ug → entier                   % niveau de recherche des propriétés
tab_act : ug entier → entier            % nbr. d'utlisations actives et passives max
tab_pas : ug entier → entier            % à considérer lors la recherche des prop.

axiomes
moteur(U, Conf, Prog) =
           si fini(Conf) = vrai
           alors Conf, Prog                         % terminaison avec réussite
           sinon si Ok = faux
                 alors Conf, Prog                   % terminaison avec échec
                 sinon moteur(U, Confbis, Progbis)
           où Confbis, Progbis, Ok = essai_regle(U, Config, Prog)

essai_regle(U, Config, Prog) = essai_regle_niv(U, Config, Prog, 0)
essai_regle_niv(U, Config, Prog, N) =
                       essai_regles_niv(U, regles(U), Config, Prog, N)

essai_regles_niv(U, [], Config, Prog, N) =
                 si N < niv_max(U)
                 alors essai_regle_niv(U, Config, Prog, N+1)
                 sinon Config, Prog, faux           % échec final !
essai_regles_niv(U, [Reg | Lr], Config, Prog, N) =
           si Appliquée = vrai
           alors essai_regle(U, Configbis, Progbis)
           sinon essai_regles_niv(U, Lr, Config, Prog, N)
où Configbis, Progbis, Appliquée =
           applique(Reg, tab_act(U, N), tab_pas(U, N), Config, Prog)
```

figure 8 : une partie de la spécification du moteur d'inférence.

Notons qu'au cours de l'évolution de la configuration, un programme de construction géométrique est engendré. Pour simplifier l'exposé, nous ne détaillons la génération de ce programme qu'à la section 4 pour nous consacrer ici à la manipulation d'une configuration par le moteur d'inférence.

3.2. L'application d'une règle

L'application d'une règle se fait trois phases : la recherche de propriétés du raisonnement unifiables modulo la figure avec les prémisses de la règle dans le raisonnement, les vérifications sur la figure demandées dans la règle et les conséquences sur l'évolution de la figure et du raisonnement. Pour pouvoir décrire la fonction `applique`, nous aurons donc besoin des fonctions auxiliaires dont les profils sont donnés à la figure 9. Parmi celles-ci,

-les fonctions `prouve` et `prouvl` sont chargées de trouver dans le raisonnement les propriétés auxquelles la règle peut s'appliquer ;
- les fonctions `vérifie` et `vérifil` s'occupent de traiter les vérifications demandées par la règle. Ces fonctions ne sont pas décrites formellement ici, le lecteur pourra se reporter à l'exemple donné plus bas ;
- la fonction `complete_configl` s'attache à mettre en oeuvre la conclusion d'une règle instanciée à l'aide des fonctions précédentes. Ceci est fait en complétant la figure et le raisonnement à l'aide des fonctions `complete_fig`, `complete_raisa` et `complete_raisp`.

```
fonctions
prouve : term0 entier entier configuration → su* entier*
prouvl : term0* entier entier configuration → su* entier**
vérifie : vérification figure → bool
vérifil : vérification* figure → bool

complete_configl : term0* entier* configuration pcg → configuration pcg bool
complete_fig : term0 figure pcg  → figure pcg              % rappel du chapitre précédent
complete_raisa : term0 entier* raisonnement → raisonnement bool
complete_raisp : term0 entier* raisonnement → raisonnement bool

préconditions
pré(prouve(Rel, Na, Np, Conf)) = Rel est un terme construit à l'aide d'un
symbole                                relationnel.
pré(prouvl(Lrel, Na, Np, Conf)) = Lrel est une liste de termes construits à
                        l'aide d'un symbole relationnel.
pré(complete_configl(Lrel, Llp, Conf, Prog)) = Lrel est une liste de termes clos
                                construits à l'aide d'un symbole relationnel.
pré(complete_fig(Rel, Fig, Prog)) = Rel est un terme clos construit à l'aide
                                d'un symbole relationnel.
pré(complete_raisa(Rel, Lp, Rais)) = Rel est un terme clos de profondeur 1
                        construit à l'aide d'un symbole relationnel
                  constructif.
pré(complete_raisp(Rel, Lp, Rais)) = Rel est un terme clos de profondeur 1
                        construit à l'aide d'un symbole relationnel.
```

figure 9 : fonctions auxiliaires pour la fonction `applique`

On peut décrire informellement les fonctions présentées de la manière qui suit :
- `prouve(T, Na, Np, Conf)` rend d'une part toutes les substitutions filtrant `T` modulo la figure incluse dans `Conf` avec les propriétés du raisonnement de `Conf`, dont les nombres d'utilisations actives et passives sont respectivement inférieurs à `Na` et `Np`, et d'autre part la liste des numéros de sommets ayant servi à calculer ces substitutions ;
- `prouvl(Lt, Na, Np, Conf)` fait la même chose avec la liste `Lt` : chaque substitution trouvée appliquée à `Lt` fournit des propriétés de la configuration `Conf` ; la liste rendue est la liste des listes des numéros de sommets s'unifiant avec `Lt` dans la substitution correspondante ;
- `vérifie(Verif, Fig)` contrôle la validité de la vérification `Verif` dans la figure ;
- `vérifil`, idem mais pour une liste de vérifications ;
- `complete_configl(Lrel, LLp, Conf, Pcg)` complète la configuration `Conf` et le programme de construction `Pcg` à l'aide des termes relationnels clos de `Lrel` et des listes de

prémisses de `LLp` ; le booléen rendu est `faux` si la mise à jour du raisonnement tente d'ajouter une dérivation déjà présente ;
- `complete_fig` est l'opération de mise à jour de la figure ;
- `complete_raisa` met à jour le raisonnement à l'aide d'une propriété constructive dont le degré de restriction sert à mettre à jour le nombre d'utilisations actives des sommets concernés ; le booléen rendu est `vrai` si la dérivation ajoutée n'était pas dans le raisonnement ;
- `complete_raisp` met à jour le raisonnement en recalculant en particulier le nombre d'utilisations passives des sommets concernés ; le booléen rendu est `vrai` si la dérivation ajoutée n'était pas dans le raisonnement.

Nous sommes maintenant en mesure de préciser la fonction `applique` et les fonctions de preuve. Les quelques fonctions auxiliaires ajoutées n'ont d'autre utilité que de démonter les différentes listes en argument.

```
fonction                    % fonctions auxiliaires de démontage de listes
essai_l : su* entier** vérification* term0* pcg configuration
                                                  → configuration pcg
bool
prouvl_bis : su* entier* term0* entier entier configuration → su* entier**
unif_som : sommet* term0 figure → su* entier*

axiomes
applique(si Lrel et Lverif alors Lconc, Nba, Nbp, Pgm, Conf) =
           essai_l(Ls, LLp, Lverif, Lconc, Pgm, Conf)
           où Ls, LLp = prouvl(Lrel, Nba, Nbp, Conf)

essai_l([], [], Lverif, Lconc, Pgm, Conf) = Conf, Pgm, faux
essai_l([Sig | Ls], [Nprem | LLp], Lverif, Lconc, Pgm, Conf) =
     si verifil(aplsl(Sig, Lverif), fig(Conf)) = vrai et Ok = vrai
     alors Configbis, Pgmbis, vrai
     sinon essai_l(Ls, LLp, Lverif, Lconc, Pgm, Conf)
où Configbis, Pgmbis, Ok =
                complete_configl(aplsl(Sig, Lconc), Nprem, Pgm, Conf)

prouvl([], Nba, Nbp, Conf) = [sv], [[]]
prouvl([Rel | Lrel], Nba, Nbp, Conf) =
     si Lsub = []
     alors [], []                    % échec
     sinon prouvl_bis(Lsub, Lsom, Lrel, Nba, Nbp, Conf)
     où Lsub, Lsom = prouve(Rel, Nba, Nbp, Conf)
prouvl_bis([], [], Lrel, Nba, Nbp, Conf) = [], []
prouvl_bis([Sub1 | Lsub], [Som1 | Ls], Lrel, Nba, Nbp, Conf)=
     si Lsubf = []
     alors [], []
     sinon [Lsubf | Lsubp], [Lsomf | Lsomp]
où   Lsubf = ajt(Sub1j, Lsubb)       % Cf Chapitre 4 pour ceux qui ne suivent pas !
     Lsomf = ajt(Som1,Lsomb)              % aspirine pour les autres ...
     Lsubb, Lsomb = prouvl(aplsl(Sub1, Lrel), Nba, Nbp, Conf)
     Lsubp, Lsomp = prouvl_bis(Lsub, Ls, Lrel, Nba, Nbp, Conf)

prouve(Rel, Nba, Nbp, Conf) =
     unifie_som(Lsom, Rel, fig(Conf))
     où Lsom = rech_rais_crg(fonctp(Rel), Nba, Nbp, rais(Conf))
unifie_som([], Rel, Fig) = [], []
unifie_som([som(N, Prop, Na, Np) | Ls], Rel, Fig) =
     si Lsub1 = []
     alors unifie_som(Ls, Rel, Fig)
```

```
      sinon Lsub1 + Lsub2, copy_id(Long, N) + Lsom2
      où    Lsub1 = unifp(Fig, Rel, Prop)
            Lsub2, Lsom2 = unifie_som(Ls, Rel, Fig)
            Long = long(Lsub1)
```

figure 10 : description formelle de la fonction `applique`

Nous ne décrirons pas davantage la fonction `unifp` qui tente l'unification de deux termes relationnels et qui utilise la fonction `term_pequiv`, vue plus haut et la fonction primordiale `unif` décrite au chapitre 6.

3.3 Mise à jour de la configuration

Passons maintenant à la mise à jour la configuration. L'application d'une conclusion a pour résultat immédiat la mémorisation des propriétés découvertes dans le raisonnement courant. A cette occasion, la figure peut être mise à jour lors de l'atomisation des arguments intervenant dans les propriétés énoncées par la conclusion. Par ailleurs, les relations d'égalité ou participatives découvertes servent directement à mettre à jour la figure. La figure 11 donne le principe de l'opération `complete_configl` dont le profil a été donné à la figure 9.

```
fonctions
complete_config : term0 entier* configuration pcg → configuration pcg
bool
                                          % fonctions auxiliaires
atomise_prop : term0 figure → term0 figure    % atomisation d'une propriété
particip : term0 → bool                       % vrai si term0 est une rel. particip.
                                              % ou d'égalité

axiomes
complete_configl([],Llp, Conf, Pgm) = Conf, Pgm, vrai
complete_configl([P|Lp], Llp, Cf, Pgm) =
           si ok = vrai
           alors complete_configl(Lp, Llp, Cf2, Pgm2)
           sinon Cf, Pgm, faux
           où Cf2, Pgm2, ok = complete_config(P, Llp, Cf, Pgm)

complete_config(P, Llp, Cf, Pgm) =
           si particip(P) = vrai
           alors Cfa, Pgm2, Boola        % la figure doit être complétée explicit.
           sinon Cfp, Pgm2, Boolp        % l'atomisation suffit à la compléter
           où Cfa, Boola = complete_raisa(Pa, Llp, Cf2)
              Cf2 = conf(rais(Cf), Fig2)
              Fig2, Pgm2 = complete_fig(Pa, Fig, Pgm)
              Cfp, Boolp = complete_raisp(Pa, Llp, Cf3)
              Cf3 = conf(rais(Cf), Fig)
              Pa, Fig = atomise_prop(P, fig(Cf))
```

figure 11 : mise à jour d'une configuration

La fonction `complete_fig` a été décrite au chapitre précédent. En ce qui concerne les fonctions `complete_raisa` et `complete_raisp`, nous en avons donné toutes les opérations de base et leur compréhension ne pose pas de problème. Aussi pour ne pas alourdir le texte, nous n'allons pas les décrire formellement.

3.4. Exemple.

Nous donnons en exemple l'évolution de la configuration lors du début de la construction donnée en exemple au chapitre 3. Nous préciserons à la section suivante comment évolue conjointement le programme de construction.

Soit la figure formelle du chapitre 3 (figure 4) décrite par les objets géométriques et les faits :

```
fe(a, point, 0, donné)
fe(b, point, 0, donné)
fe(c, cercle, 0, donné)
fe(point00, point, 0, centre(c))
fe(long00, longueur, 0, rayon(c))
fe(d, droite, 1, [a/incidentpd])
fe(e, point, 1, [c/incidentcp])
fe(f, point, 1, [c/incidentcp])
fe(long1, longueur, 1, [])

point00 = centre(c)
long00 = rayon(c)

long01 = dist(f, b)
long01 = dist(e, b)

a incidentpd d

f incidentpd d
f incidentpc c

e incidentpd d
e incidentpd c

c incidentcp e
c incidentcp f
c apourrayon long00
c apourcentre point00

 d incidentdp e
d incidentdp f
d incidentdp a
```

Il faut noter que ceci représente en fait un grand terme plein d'`aog`, `aeg` et `apt` partout et un `fv`. Et le raisonnement résultant de la traduction de l'énoncé correspondant :

```
som(0,e est_sur c,0,0)
som(1,f est_sur c,0,0)
som(2,e est_sur d,0,0)
som(3,f est_sur d,0,0)
som(4,a est_sur d,0,0)
som(5,long01=l=long01,0,0)
```

(aucune dérivation)

La première règle appliquée par Progé est la règle 100 (Cf. chapitre 3, figure 5), c'est-à-dire la règle :

`si [dist(A, B) =l= dist(A, C)] et [differents [A, B, C]] alors [iso(A, B, C)]`

Seul le sommet 5 du raisonnement, recherché à l'aide de la fonction `rech_rais_crg`, peut s'unifier avec l'unique prémisse de la règle. Nous ne détaillons pas toutes les substitutions trouvées : elles résultent des permutations possibles autour des égalités

`long01` = `dist(e, b)` et `long01` = `dist(f, b)`.

Voici celles tentées par Progé et leur résultat sur l'instanciation de la règle :

`si[dist(f,b)=l=dist(f,b)]et[differents[f,b,b]]alors[iso(f,b,b)]`
(échec à cause de la vérification `differents[f, b, b]`)

`si[dist(b,f)=l=dist(b,f)]et[differents[b,f,f]]alors[iso(b,f,f)]`
(échec à cause de la vérification `differents[b, f, f]`)

`si[dist(b,f)=l=dist(b,e)]et[differents[b,f,e]]alors[iso(b,f,e)]`

La dernière substitution réussit et le terme `iso(b, f, e)` sert à mettre à jour la configuration précédente. La figure ne change pas car d'une part cette propriété est déjà atomisée, et d'autre part le symbole de prédicat `iso` ne représente ni une égalité, ni une relation constructive. Le nouveau raisonnement à l'issue de cette règle est le suivant :

```
sommet(0,e est_sur c,0,0)
sommet(1,f est_sur c,0,0)
sommet(2,e est_sur d,0,0)
sommet(3,f est_sur d,0,0)
sommet(4,a est_sur d,0,0)
sommet(6,iso(b,f,e),0,0)
sommet(5,long01=l=long01,0,1)

deriv(0,5,6,100)
```

Une dérivation est ajoutée, et le nombre d'utilisations passives du sommet 5, qui a servi de prémisse à cette dérivation, passe à 1. La règle 102 est ensuite appliquée par Progé (Cf figure 5 du chapitre 3). Elle ne modifie pas la figure et le raisonnement s'enrichit d'une nouvelle dérivation. Plus intéressante est l'application de la règle 105 qui est la troisième règle appliquée (Cf. figure 5 du chapitre 3) :

`si [iso(A, B, C)] et [pas_connu B]`
`alors [I nomme mil(B, C), dro(A, I) =d= med(B, C), dro(A, I) ortho dro(B, C)]`

Le symbole `nomme` est une facilité ajoutée pour éviter de compliquer l'écriture de la règle : `nomme` provoque l'atomisation dans la figure du terme à sa droite pour instancier la variable à sa gauche. `nomme` ne provoque aucune mise à jour du raisonnement. Cette règle ne peut s'instancier qu'avec le sommet 6, et deux substitutions seulement sont possibles correspondant aux permutations permises sur les termes construits avec `iso`. Voici l'application de la substitution utilisée à la règle :

```
si[iso(b,f,e)]et[pas_connu f]
alors
[_23180 nomme mil(f,e),dro(b,_23180)=d=med(f,e),dro(b,_23180)ortho dro(f,e)]
```

Le terme `_23180` est une variable non encore instanciée au moment de l'application de la règle : ceci est une particularité du symbole relationnel `nomme`. Une valeur est attribuée à cette variable par atomisation de `mil(f, e)` dans la figure : comme ce terme n'y figure pas encore, une nouvelle constante de la sorte point est crée et l'objet géométrique et les formes égalitaires correspondantes sont ajoutées à la figure qui contient alors

```
fe(point01,point,2,[])
```

```
point01 = mil(f,e)
```

Lors de cette création, les déclarations de mise à jour automatiques liées au symbole `mil` sont appliquées pour ajouter les deux relations constructives :

```
point01 incidentpd d
d incidentdp point01
```

La deuxième propriété découverte devient alors `dro(b, point01) =d= med(f, e)`. Cette relation est atomisée : comme aucun des deux termes `dro(b, point01)` et `med(f, e)` n' est déjà présent dans la figure, un seul objet nouveau est créé et ajouté à la figure :

```
fe(droite00,droite,1,[b/point/incid])
```

avec les faits suivants :

```
droite00 = med(f, e)

droite00 incidentdp b
droite00 incidentdp point01
point01 incidentpd droite00
b incidentpd droite00
```

La troisième propriété, `dro(b, point01) ortho dro(f, e)` est également atomisée. Mais comme chacun des termes est représenté dans la figure, et comme le symbole relationnel `ortho` n'est pas constructif, celle-ci ne change pas lors du traitement de cette propriété. Le raisonnement est mis à jour. En voici la nouvelle version :

```
sommet(0,e est_sur c,0,0)
sommet(1,f est_sur c,0,0)
sommet(2,e est_sur d,0,0)
sommet(3,f est_sur d,0,0)
sommet(4,a est_sur d,0,0)
sommet(5,long01=l=long01,0,1)
sommet(7,droite00=d=droite00,0,0)
sommet(8,droite00 ortho d,0,0)
sommet(6,iso(b,f,e),0,2)

deriv(0,5, 6,100)
deriv(1,6, 5,102)
deriv(2,6, 7,105)
deriv(3,6, 8,105)
```

4. Elaboration du programme de construction

Nous avons passé sous silence l'élaboration du programme de construction dans les explications de la section précédente. En fait, la formation du programme de construction est du ressort des fonctions de gestion de la figure et plus exactement de l'adjonction d'objets géométriques à une figure. Nous ne l'avons pas fait dans un soucis de clarification de l'exposé, mais toutes les fonctions du chapitre 6 qui font appel à la fonction `aogp`, fonction de haut niveau ajoutant un objet géométrique à une figure, auraient du posséder la sorte `pgc` dans leur arité et dans leur coarité. Nous aurions ainsi dû écrire :

`aogp : og figure pgm` → `figure pgm`

Lorsqu'un objet de la forme `fe(Nom, Sort, 0, Def)`, c'est-à-dire construit et donc avec une définition complète, est ajouté à la figure, l'instruction de définition

```
Nom := Def
```

est ajoutée à la fin du corps d'instruction du programme donné en argument. Et la déclaration

```
dec(Nom, Sort)
```

est ajoutée à la liste des déclaration du programme. Ceci dans le cas où la définition `Def` n'est pas ambiguë. Lorsque c'est le cas, une instruction de définition et une itération sont ajoutées à la fin du corps du programme :

```
List := Def
pour Nom dans List faire <bloc>
```

A cet effet, un nom d'identificateur de la sorte `liste` est engendré par Progé. Cette manière de faire permet d'avoir la liste des définitions des objets construits dans l'ordre où elles sont obtenues. Comme Progé s'assure que dans une définition d'un objet géométrique les arguments sont construit, on est sûr d'avoir un programme bien défini. La simplicité de ce mécanisme et l'absence de contrôle a cependant cet effet gênant : tout objet dont Progé a trouvé la construction est consigné dans le programme de construction même s'il n'a aucun rapport avec les objets que l'énoncé demande de chercher. Nous verrons dans les exemples donnés en annexe que ceci se produit effectivement. Une solution consisterait à éliminer ces définitions parasites en utilisant un tri topologique par exemple, mais nous n'avons pas testé cette méthode.

L'exemple de programme donné au chapitre 3 et la description que nous avons donnée de la sorte `pcg` au chapitre 5 sous-entendent une forme plus générale des programmes de construction que celle que nous venons d'exposer. En fait l'introduction de conditionnelles et de structures itération/conditions résulte de la prise en compte des exceptions liées à un symbole fonctionnel et de la possibilité de cas de figure. Toutes choses que nous allons détailler au chapitre suivant. Cependant, pour chaque corps d'instructions dans un programme, que ce soit dans une conditionnelle ou une itération, le processus d'élaboration est celui dépeint ici.

Chapitre 8

Cas de figure, exceptions

Et puis, plus rien. Certains crurent, dans l'air, ouïr
Un soupir divagueur, équivoque et lassé,
Qui pouvait être : « ... jum ! » Les autres prétendirent
Que c'était seulement la brise qui passait.

Les descriptions formelles des chapitres précédents modélisent *grosso modo* les phases d'analyse et de synthèse d'un exercice de construction : l'analyse correspond à l'application des règles, la synthèse correspond à l'extraction d'un programme de construction de la figure formelle au fur et à mesure de son élaboration. Nous avons montré au chapitre 5 comment les structures conditionnelles d'un programme de construction répondent à la question de la discussion d'une construction. Nous exposons dans ce chapitre les mécanismes à l'aide desquels Progé discute une construction en cours et leur incidence sur l'élaboration du programme de construction.

1. Introduction

Nous avons retenu dans Progé deux raisons de discuter une construction : l'existence de cas dégénérés d'une part, et l'existence de cas de figure d'autre part.

Les *cas dégénérés* correspondent à la non validité d'un terme employé dans un programme de construction.

Exemple.
L'exemple standard est la tentative de définition d'une droite par deux points qui se trouvent confondus lors d'une interprétation numérique.

L'utilisation d'une figure formelle telle que nous l'avons décrite dans les chapitres précédent permet d'éliminer la plupart des cas de dégénérescence détectables formellement, c'est-à-dire les cas où il y a dégénérescence quelle que soit l'interprétation des constantes. Comme nous voulons produire un programme de construction le plus général possible, il nous faut envisager les cas particuliers. La manière dont sont traités les dégénérescences dans Progé est expliquée à la section 2.

Le problème des *cas de figure* est plus général : il arrive parfois qu'avec les mêmes hypothèses on ait à choisir entre deux figures présentant chacune des propriétés contradictoires avec l'autre.

Exemple.
Il en est par exemple ainsi, si l'on considère quatre points A, B, C et D tels que les droites (AB) et (CD) soient parallèles et les distances AB et CD soient égales : deux types de figure

répondent à cette spécification : dans un cas (ABCD) est un parallélogramme, dans l'autre cas (ABDC) est un parallélogramme. Suivant qu'un cas de figure ou l'autre est choisi, la suite de la construction se déroule de manière différente. Ceci peut se traduire par une règle comme celle-ci :

si dist(A, B) = dist(C, D)
    et les droites AB et CD sont parallèles
    et les points A, B, C et D sont distincts
alors
    soit (A, B, C, D) est un parallèlogramme
    soit (A, B, D, C) est un parallèlogramme

Remarquons que les cas de figure sous-tendus par cette règle ne sont pas exclusifs *a priori*, c'est-à-dire qu'à partir de la même configuration tous les cas de figure exprimés par la règle sont possibles. Nous appelons une telle règle une *règle disjonctive à cas de figures non exclusifs* et leur mise en oeuvre est détaillée à la section 3.

Il n'est pas toujours facile de décider si l'on doit traduire des cas particuliers par des cas dégénérés ou par des règles disjonctives.

Exemple.
Prenons le cas des bissectrices de deux droites. Il y a trois cas à distinguer a priori, soit les droites sont concourantes, soit les droites sont strictement parallèles, soit les droites sont confondues. Il est difficile de traiter ces cas comme étant des cas dégénérés si l'on ignore quelle est la définition des bissectrices parmi celles-ci :
- est-ce le lieu des points équidistants à deux droites ?
- est-ce le lieu des points coupant un angle de droite en deux angles égaux ?
- est-ce un axe de symétrie ?

Si dans le cas général (droites concourantes), ces trois définitions coïncident, il n'en vas pas de même pour les cas particuliers évoqués plus haut. C'est pourquoi, il nous semble préférable d'expliciter les conditions de validité dans des règles d'obtention des bissectrices comme la suivante, qui reprend la première définition :

si la distance de A à la droite D1 égale la distance de A à la droite D2
  et D1 et D2 sont *a priori* différentes et construites, et A n'est pas encore connu
alors
  soit D1 et D2 ne sont pas parallèles, on a alors que A est sur une bissectrice de (D1, D2)
  soit D1 et D2 sont strictement parallèles, on a alors que A est sur la droite "milieu" de (D1, D2)
  soit D1 et D2 sont confondues, on ne peut alors rien dire de spécial sur la localisation de A

Dans cette règle, on a, en quelque sorte, des *prémisses secondaires* qui servent à départager les cas de figure. Ces prémisses sont testables avant l'application de la règle et les cas de figure sont mutuellement exclusifs : nous disons qu'une telle règle est *disjonctive à cas de figure exclusifs*. Le formalisme de telles règles est expliqué à la section 3.

2. Exceptions

Les cas de dégénérescence de base reconnus par Progé sont ceux liés à la tentative d'utiliser une définition invalide. Les conditions d'utilisation des symboles fonctionnels sont incluses dans l'univers géométrique de base sous forme d'attribut de symbole fonctionnel géométrique. La description d'univers géométrique donnée au chapitre 4 doit donc être complétée par le sélecteur `except` de profil :

```
except : cog -> feg impll*
```

Les sortes `cog`, désignant les symboles fonctionnels, et `feg`, désignant les formes égalitaires, ont été introduites et expliquées au chapitre 4. La sorte `impll` utilisée à cette occasion représente des *implications* notées avec le symbole >> dans lesquelles les prémisses et la conclusion peuvent être soit des termes prédicatifs, soit des négations de termes prédicatifs. Dans la liste des implications rendue par `except`, la première indique les conditions de validité du *cas général* dans lequel le terme incriminé a un sens. Les implications suivantes indiquent quels sont les *cas dégénérés* à considérer et leur incidence sur la figure formelle.

Exemples.
Dans l'univers géométrique de base de Progé, nous avons considéré les exceptions suivantes pour les symboles fonctionnels `dro` et `mil`.

```
except(dro) = (D = dro(A, B), [[A diff B] >> [], [A =p= B] >> []])
except(mil) =
      (M = mil(A, B), [[A diff B] >> [A diff M, B diff M], [A =p= B] >> [A =p= M]]
except(dist) = (L = dist(A, B), [])
```

Dans le premier exemple, on indique qu'une droite `D` est définie à l'aide de `dro(A, B)` si les deux points sont différents (`A diff B`) et qu'il n'y a pas d'autres conséquences à considérer dans le cas général. Il n'y a qu'un cas dégénéré qui correspond à l'hypothèse "`A` égale `B`". Dans cette éventualité, la droite `D` n'est pas définie et la figure formelle est mise à jour en utilisant l'égalité `A =p= B`.
Dans le deuxième exemple, il est précisé que dans le cas général pour `M = mil(A, B)`, on doit avoir `A` différent de `B`, et alors `A`, `B` et `M` sont différents. Le cas où `A` égale `B`, même s'il n'est pas à proprement parler dégénéré, est suffisamment particulier pour être traité à part. Dans ce cas, on a que `A` égale `M`.
Le troisième exemple montre un symbole fonctionnel, `dist`, pour lequel il n'y a pas de cas dégénérés.

Ce nouvel attribut `except` est mis en oeuvre de la manière suivante. Au moment où Progé découvre une définition complète pour un objet géométrique de la figure, il examine l'existence de cas dégénérés liés au symbole fonctionnel de la définition en cause. S'il n'y en a pas, comme dans la troisième cas de l'exemple ci-dessus, la construction se poursuit comme cela a été décrit aux chapitres précédents. S'il y a des exceptions, la première implication de la liste concerne le cas général. Les conditions en prémisses sont vérifiées pour voir si ce cas est toujours réalisé ou non : si Progé peut prouver les prémisses, la définition est acceptée et Progé continue comme précédemment. Si, au contraire, Progé prouve que les prémisses ne sont jamais réalisées dans la configuration actuelle, alors la définition est refusée et les cas de dégénérescence ne sont pas examinés. Si Progé ne peut démontrer ni les prémisses du cas général, ni leur contraire, alors la configuration actuelle est empilée avec la liste des cas particuliers encore à traiter. Le cas général est traité en prenant en compte la définition et en ajoutant à la configuration les propriétés caractérisant le cas général sous forme de *faits*

d'exceptions. Le programme de construction contient en ce point une structure conditionnelle de la forme

`si` <prémisses du cas général> `alors` <bloc correspondant au cas général>
`sinon` <bloc correspondant aux cas dégénérés>

Lorsque le cas en cours a été examiné, c'est-à-dire qu'une construction a été trouvée ou qu'au contraire Progé a échoué, et si la pile des configurations n'est pas vide, alors la configuration au sommet est dépilée, les prémisses du cas particulier sont examinées et si Progé ne peut pas les infirmer, une nouvelle construction est lancée à partir de cette configuration à laquelle sont ajoutés, sous forme de faits d'exception, les propriétés particularisant ce cas de figure. Cette construction complète le programme de construction en élaborant le bloc correspondant à l'alternative `sinon` laissée en suspens.

Nous ne donnerons pas de description formelle de ces manipulations : pour faire les choses proprement, il faudrait en effet revoir les spécifications de la figure formelle et du moteur d'inférence. On peut cependant résumer la procédure précédente par le pseudo-code de la figure 1 dans lequel nous appelons `C` la configuration actuelle, `P` le programme de construction engendré jusqu'alors et `D` la définition à discuter.

Si la définition `D` présente des cas de dégénérescence possibles
alors
si on prouve qu'on est dans le cas général dans `C`
alors ajouter la définition `D` à `C` pour obtenir `C'`
ajouter la définition `D` à `P` pour obtenir `P'`
continuer la construction à partir de `C'` et de `P'`
(*le programme de construction ne présentera pas de conditionnelle en ce point*)
sinon si on prouve que le cas général n'est jamais réalisé
alors refuser la définition
continuer la construction à partir de `C` et de `P`
sinon (*on est soit dans le cas général, soit dans le cas dégénéré*)
placer une conditionnelle dans le programme `P` avec :
pour condition, celle du cas général
pour la branche `alors` la partie de programme pour le cas général
obtenue à partir de `C'` et `D'` définis en ajoutant `D` respectivement à `C` et à `D`
pour la branche `sinon` la partie du programme obtenue à partir de `C` en
examinant les cas dégénérés
sinon (*il n'y a pas de cas particuliers attachés à cette définition*)
ajouter la définition `D` à `C` pour obtenir `C'`
ajouter la définition `D` à `P` pour obtenir `P'`
continuer la construction à partir de `C'` et de `P'`

figure 1 : pseudo-code de la gestion des cas dégénérés

Exemple
Dans le programme donné en exemple au chapitre 3, nous trouvons le fragment de code (qui conclut la construction dans le cas où l'on n'a pu prouver ni `point01 diff a`, ni son contraire) :

```
...
si [point01 diff a] alors              % cas général
      d := dro(point01,a)
      e := intercd(c,d)
      f := intercd(c,d)
      fin
sinon                                  % cas dégénéré où
      dir04 := diro(droite00) % point01 égale a
      d := dpd(a, dir04)
      e := intercd(c,d)
      f := intercd(c,d)
      fin
...
```

Dans la branche `alors` le point `point01` est différent de `a`, dans la seconde `point01` = `a`. Avant la conditionnelle, la figure formelle comporte, entre autres, les objets et les relations suivants :

```
fe(point01, point, 0, intercd(cercle00, droite00))
fe(a, point, 0, donné)
fe(d, droite, 1, [a/incidentpd])

point01 incidentpd d
```

Le point `point01` a été introduit comme étant le milieu de [`e, f`] (Cf. chapitre 3, le programme de construction donné en exemple). Par les mécanismes de mise à jour automatique, Progé en a immédiatement déduit que `point01` appartenait à la droite (`ef`) = `d`. Le point `point01` vient d'être construit et, par propagation de contrainte, Progé tente de construire la droite `d` à l'aide de la définition incomplète [`point01/incidentpd, a/incidentpd`] qui devient `dro(point01, a)` à l'aide de la fonction de la fonction de recomposition `p2c` (Cf. chapitre 6). Avant de considérer cette définition les exceptions concernant le symbole fonctionnel `dro` sont examinées : le cas général correspond à `point01` différent de `a`, ce que Progé ne sait ni prouver, ni infirmer dans la configuration actuelle. Celle-ci est donc empilée avec le cas dégénéré non traité correspondant à `a = point01`. Puis la construction continue avec la configuration dans laquelle Progé a noté que les points `a` et `point01` sont différents. La droite `d` est construite, ce qui assure le succès dans ce cas, et le programme de construction est complété pour définir les points `e` et `f` utilisés dans les vérifications. Après examen de ce cas, la pile des configurations est dépilée et la configuration correspondant au cas où `a=point01` est examinée : Progé trouve qu'elle n'est pas contradictoire et le processus de construction peut être lancé. Progé commence par fusionner les `point01` et `a` qui deviennent synonymes, et on obtient le fragement de figure suivant :

```
fe(a, point, 0, donne)
fe(d, droite, 1, [a/incidentpd])

a alias a
point01 alias a
(a et point01 sont synonymes)
```

Le fait que `d` soit orthogonale à la `droite00`, qui est construite, est exploité pour produire une nouvelle définition de `d` = droite passant par `a` et de direction orthogonale à `droite00`, ce qui conclut favorablement la construction dans ce cas dégénéré.

3. Règles disjonctives

Si la mise en oeuvre des mécanismes de gestion des cas dégénérés relève de la gestion de la figure formelle, l'application des règles disjonctives est l'affaire du moteur d'inférence et plus particulièrement de la fonction `applique` vue au chapitre précédent.

3.1 Cas de figure exclusifs

Une règle *disjonctive à cas de figure exclusifs* est de la forme :

> `si` <prémisses> et <vérifications>
> `alors`
> `soit` <liste non vide de prémisses secondaires> `et` <propriétés corresp. à ce cas>
> ...
> `soit` <liste non vide de prémisses secondaires> `et` <propriétés corresp. à ce cas>

Les prémisses secondaires annoncées par le mot clé `soit` doivent être mutuellement exclusives, mais ceci n'est pas vérifié par Progé.

Exemple.
La règle concernant les bissectrices donnée à la section 1 se traduit, dans la syntaxe de Progé, de la manière suivante :

```
si [did(A,D1) '=l=' did(A,D2)]
et
[differents [D1,D2], connu D1, connu D2, pas_connu A]
alors
soit [dird(D1) diff dird(D2)] et [ A est_sur bis(D1,D2) : 1]
   ou
soit [dird(D1) eg dird(D2), D1 diff D2] et [A est_sur dmd(D1,D2) : 1]
   ou
soit [D1 eg D2] et [].
```

Lorsque Progé à réussi à prouver les prémisses et à justifier la liste des vérifications d'une règle disjonctive à cas de figure exclusifs, il tente de prouver qu'on est dans un des cas particuliers énumérés par la règle en examinant les prémisses secondaires. Si il est prouvé qu'on est dans un cas particulier, ceux-ci s'excluant mutuellement, seule la conclusion correspondant à ce cas particulier est examiné. Si Progé prouve qu'aucun cas particulier n'est applicable, la règle échoue. Pour une règle valide qui recouvre bien tous les cas, ceci signifie qu'il y a une contradiction et que la construction n'est pas possible. Si l'on n'est dans aucune de ces deux éventualités, c'est-à-dire, si après avoir éliminé les cas impossibles, il reste plusieurs cas qu'on ne peut ni prouver, ni infirmer, alors ces cas restants sont examinés successivement de la manière décrite à la section précédente : la construction continue avec le premier cas particulier, la configuration actuelle avec les cas subsidiaires est empilée sur la pile des configurations à traiter et une instruction conditionnelle est placée dans le programme de construction.

Exemple.
Considérons la résolution de l'exercice suivant (Cf. figure 2) :

Enoncé. *Construire un cercle* `s` *tangent à un cercle* `c` *donné en un point* `a` *donné et à une droite* `d` *donnée également.*

Progé découvre d'abord que `s` et c possédent une tangente commune, nommé `droite03` (`d03` sur la figure 2). Il trouve ensuite que les distances du centre de `s`, nommé `point00` (`p0` sur la figure 2), aux droites `d` et `droite03` sont égales. La règle vue plus haut peut alors être appliquée avec la substitution :

`A` ← `point00, D1` ← `droite03, D2` ← `d`

C'est ainsi que Progé doit appliquer la conclusion :

```
soit [dird(droite03) diff dird(d)] et [ point00 est_sur bis(droite03,d)]
   ou
soit [dird(droite03) eg dird(d), droite03 diff d]
                                    et [point00 est_sur dmd(droite03,d)]
   ou
soit [droite03 eg d] et [].
```

figure 2 : figure correspondant à l'exercice exo8b

Progé ne peut ni prouver, ni infirmer aucune des prémisses secondaires dans la configuration courante et les trois cas de figure sont envisagés : dans le premier cas `d` et `droite03` sont concourantes, dans le deuxièmes les droites `d` et `droite03` sont strictement parallèle et dans le troisième les droites `d` et `droite03` sont confondues.

Progé poursuit la construction du premier cas en ayant noté qu'alors `droite03` et `d` n'étaient pas parallèles et qu'elles possédaient donc des bissectrices. Le programme de construction possède en ce point la condition suivante :

`si [dird(droite03) diff dird(d)]`
`alors` < construction correspondant à l'existence de bissectrices>
`sinon` <constructions correspondant aux deux cas restants>

La construction s'achève avec succès dans ce cas, puis la configuration correspondant aux deux cas restants est dépilée. Le premier des cas restant correspond à la condition `[dird(D1) eg dird(D2), D1 diff D2]` de la règle, c'est-à-dire que dans notre cas les droites `droite03` et `d` sont strictement parallèles. Avant d'examiner la construction dans ce cas, Progé réempile la configuration avec le dernier cas restant. Une conditionnelle est placée dans le programme de construction :

`si [dird(droite03) eg dird(d), droite03 diff d]`
`alors` < construction correspondant à `d` et `droite03` strictement parallèles>
`sinon` < construction correspondant à `d` et `droite03` confondues>

La construction dans ce deuxième se termine avec succès, et la configuration correspondant au troisième cas est dépilée. Dans ce cas, les droites `d` et `droite03` sont égales et sont donc fusionnées dans la figure formelle. Progé relance le processus de construction à partir de cette configuration pour trouver une "fausse solution". Cet exemple est développé complétement à l'annexe 1.

3.2. Cas de figure non exclusifs

Une règle *disjonctive à cas de figure non exclusifs* est de la forme :

```
si <prémisses> et <vérification>
alors
  soit [] et <propriétés corresp. à ce cas>
     ...
  soit [] et <propriétés corresp. à ce cas>
```

A la différence des règles disjonctives précédentes, les règles disjonctives à cas de figure non exclusifs ont pour prémisses secondaires des listes vides, i.e. on ne peut pas vérifier à l'avance si on est dans un cas ou l'autre.

Exemple.
La règle du parallélogramme de la section 1 s'écrit ainsi dans la syntaxe indiquée :

```
si [dist(A,B) '=l=' dist(C,D), dird(dro(A,B)) '=di=' dird(dro(C,D))]
et
 [differents [A,B,C,D]]
alors
soit [] et [pll(A, B, C, D)]
 ou
soit [] et [pll(A, B, D, C)].
```

Lors de l'application d'une telle règle, Progé vérifie qu'on n'est pas déjà dans un des cas de figure envisagés en examinant si l'une des conclusions ou une propriété contradictoire avec l'une de celles-ci n'a pas déjà été découverte par Progé. Si c'est le cas, la règle échoue et ceci évite de boucler intempestivement. Sinon, le premier cas est examiné par Progé tandis que la configuration est empilée avec les cas restants. En ce qui concerne le programme de construction, un identificateur de liste de cas est créé auquel on affecte la liste des conclusions à considérer. Puis on ajoute une instruction d'itération/conditions où chaque condition correspond à l'un des cas examinés.

Exemple.
Soit l'exercice donné par l'énoncé :

Enoncé. *Etant donnés deux droites* `d1` *et* `d2` *parallèles, un point* `a` *sur* `d1`*, un point* `b` *sur* `d2` *et un point* `o`*, construire une droite* `d` *coupant* `d1` *en* `m` *et* `d2` *en* `n` *de sorte que la somme des distances* `am + bn` *soit égale à une constante* `l` *donnée*.

Lors de la construction, l'application de la règle du parallélogramme donnée plus haut se traduit dans le programme de construction par :

```
list01 := [[pll(point00, m, b, n)], [pll(point00, m, n, b)]]
pour cas00 dans list01 faire
      si cas00 eg [pll(point00, m, b, n)] alors
                  <programme correspondant à ce cas>
      si cas00 eg [pll(point00, m, n, b)] alors
                  <programme correspondant à ce cas>
```

L'exemple complet est développé à l'annexe 1.

4. Prouver des conditions de dégénérescence ou des prémisses secondaires

La manière de prouver une condition de dégénérescence ou une prémisse secondaire est sensiblement différente de celle utilisée lors du déclenchement d'une règle. Les mécanismes de preuve d'une règle s'appuient essentiellement sur le raisonnement et le filtrage modulo la figure tandis que pour démontrer un cas de dégénérescence, Progé peut :

- soit vérifier qu'un tel cas n'a pas déjà été mémorisé dans les faits d'exception,

- soit chercher directement les informations dans la figure pour les conditions d'égalités et d'incidence notamment,

- soit exploiter des règles spéciales notamment pour montrer que deux objets sont différents,

- soit en dernier recours explorer le raisonnement.

Parmi ces quatre points, seul le troisième est nouveau par rapport à ce que nous avons déjà décrit. Malheureusement, et nous arrivons ici aux confins des possibilités actuelles de Progé, la syntaxe des règles spéciales et leurs mécanismes d'application ne sont encore ni stabilisés, ni décrits formellement. L'idée est d'avoir des règles heuristiques pour éviter la multiplication de conditionnelles sans intérêt.

Exemple.
Dans la version actuelle, pour montrer que deux points `A` et `B` sont différents, Progé examine les faits d'exception déjà mémorisées dans la configuration parmi lesquels on peut avoir explicitement `A diff B`. Puis, il tente successivement de prouver qu'il existe une longueur `L` non nulle telle que `dist(A, B) = L`, puis que `A` est le centre d'un cercle contenant `B`, puis que `A` et `B` sont sur deux cercles concentriques de rayons différents. Si toutes ces tentatives échouent, il est inutile d'explorer le raisonnement qui ne contient aucune propriété construite avec le symbole relationnel spécial `diff`, et la démonstration de la non-coïncidence de `A` et `B` échoue.

Chapitre 9

Implantation

Jusqu'à la nuit, chassant, ils cherchèrent en vain
Une plume, un bouton, un indice quelconque
Qui permît d'affirmer qu'ils foulaient le terrain
Où le boulanger avait rencontré le Snark.

En tant que langage de manipulation de termes, Prolog est un langage privilégié pour implanter des spécifications algébriques de types abstraits [Bouma & Walters 89]. Il a donc été choisi naturellement pour décrire et manipuler l'algèbre de termes que constitue un univers géométrique. Les mécanismes d'unification standard et de retour arrière ont permis de simplifier l'implantation du filtrage modulo une figure et du moteur d'inférence. Avec la restriction habituelle pour le moteur d'inférence : Prolog favorise la recherche en profondeur d'abord et rend difficile, au moins psychologiquement, l'implantation d'un autre genre de stratégie. Par ailleurs, le passage d'une formulation fonctionnelle à une formulation fonctionnelle ne pose pas de problème (l'inverse est plus difficile). Les seuls écarts regrettables avec la spécification donnée dans ce rapport, mais nécessaires pour avoir un prototype efficace, résident dans un implantation dynamique des notions de figure, raisonnement et programme de construction et dans l'utilisation d'effets de bord pour manipuler ces structures de données.

Les explications qui suivent sont un peu techniques. Les lecteurs non intéressés par la question pourront omettre ce chapitre. D'autres pourront l'écarter en première lecture.

1. Univers géométrique

L'*univers géométrique de base* est, lors d'une construction, une constante connue de la plupart des fonctions de Progé. Nous avons donc choisi de le coder sous forme d'un ensemble de clauses Prolog. Plus exactement, chaque attribut d'une entité géométrique de l'univers est codé à l'aide d'un prédicat. L'association d'une entité géométrique avec ses attributs se fait grâce au mécanisme habituel de "pattern matching". Nous avons ainsi un programme Prolog donnant une description de l'univers géométrique conforme aux spécifications du chapitre 4.

1.1. Sortes géométriques

De cette façon, les attributs `deg_max`, `dessin` et `saisie` de la sorte `sgeo` des sortes géométriques (cf. chapitre 4) sont implantés par les prédicats `dmax`, `dessine` et `saisie` (Cf. figure 1). Pour des raisons pratiques, nous avons ajouté trois prédicats pour décrire les sortes géométriques : il s'agit des prédicats de nom `autom`, `princip` et `dessinable`. Le prédicat `autom` est utilisé lors de l'introduction d'un nouvel objet de la sorte géométrique

mentionnée pour ajouter automatiquement de nouveaux objets dépendant quasi canoniquement du premier. Ainsi, si un cercle est considéré dans une construction, il paraît naturel de considérer également son centre et son rayon. A l'aide du prédicat `autom`, Progé se charge d'introduire automatiquement ces deux objets dans la figure. Le prédicat `princip` donne la liste des symboles fonctionnels décomposables liés au type donné. Ceci évite de faire des recherches dans l'univers géométrique au moment de la reconstruction d'un terme à partir de participants. Le prédicat `dessinable` indique quant à lui si les objets de la sorte mentionnée peuvent être représentés graphiquement.

Exemple.
Pour la sorte géométrique `droite` nous avons les clauses de la figure 1.

```
dmax(droite,2):- !.

autom(droite,D,[dir :: Di nomme dird(D)], [dpdir(nul, Di)]) :- !.

princip(droite,[dro,dpdir]):- !.

dessinable(droite).

dessine(Nom, d(A,B,C)) :-
     point_bas(A,B,C,Xb,Yb),
     point_haut(A,B,C,Xh,Yh),
     tbw_draw_line(0,Xb,Yb,Xh,Yh,1,0).

saisie(droite, Nom, d(A,B,C)) :-
     nl, write('**** saisie de : '), write(Nom), write(' (droite)'),
     nl, write('----> premier point '), nl,
     tbo_get_event(E1,[tb_click_down_wdw, tb_ms_move_wdw]),
     s_traite_event(E1, p(Xe1,Ye1)),
     write(' ok'),
     coord_ec(X1,Y1,Xe1,Ye1), dessine(' ',p(X1,Y1)),
     nl, write('----> deuxieme point...'), nl,
     tbo_get_event(E2,[tb_click_down_wdw, tb_ms_move_wdw]),
     s_traite_event(E2, p(Xe2,Ye2)),
     coord_ec(X2,Y2,Xe2,Ye2), dessine(' ',p(X2,Y2)),
     write(' ok'), nl,
     A is Y1 - Y2,
     B is X2 - X1,
     C is -A*X1 - B*Y1.
```

figure 1 : clauses pour la sorte géométrique `droite`

Une droite a un degré de liberté maximal égal à 2. Progé considère, pour toute droite, sa direction : ceci permet de gérer la relation de parallélisme à l'aide de l'égalité entre directions. Les symboles fonctionnels décomposables pour une droite sont `dro`, définissant une droite par deux points, et `dpdir`, définissant une droite par un point et une direction. Une droite est dessinable, et la procédure de dessin correspondante est le corps de la clause du prédicat `dessine` et indiquée à la figure 1. La saisie d'une droite se fait à l'aide de la procédure indiquée dans le corps de la clause `saisie(droite, Nom, d(A, B, C))` mentionnée plus haut. Une droite est saisie par pointage à l'écran de deux de ses points. Une droite est ici représentée algébriquement par un terme de la forme `d(A, B, C)` où `A`, `B` et `C` sont des réels tels que l'équation de la droite, dans le repère fixé au chapitre 2, soit $Ax + By + C = 0$.

1.2. Symboles fonctionnels

La sorte `cog` des symboles fonctionnel sgéométriques dont les attributs sont `arité`, `coarité`, `ambigu`, `décomposition`, `equiv`, `maj`, `févaluation` (Cf. chapitre 4) et `except` (Cf. chapitre 8), est décrite par les prédicats de nom : `profil`, `rmult`, `decomp`, `equiv`, '`maj:`', `evaluation` et '`except:`'. Les attributs `arité` et `coarité` sont regroupés dans le seul prédicat `profil`. L'attribut `ambigu` est implanté par le prédicat `rmult` (résultats multiples) avec la petite différence suivante qui résulte de la gestion implicite des booléens : `rmult(f)` réussit lorsque le symbole fonctionnel `f` est ambigu et échoue sinon. L'attribut `décomposition` est renommé `decomp`. En ce qui concerne la permutation des arguments, nous avons préféré traduire chaque permutation autorisée par une seule clause du prédicat `equiv` :

`equiv(T) = [T1, T2, ... Tn]` est implanté par l'ensemble de clauses

```
T equiv T1.
T equiv T2.
  ...
T equiv Tn.
```

Le parcours de cet ensemble de clause est assuré par le mécanisme de retour arrière. La clause

```
Term equiv Term.
```

assure que tout terme est équivalent à lui-même et permet d'éviter l'écriture d'une telle clause pour chaque symbole fonctionnel.

Exemples.
Ainsi pour le symbole fonctionnel `mil`, nous avons les clauses de la figure 2.

```
profil(mil, point x point >> point):- !.

mil(A,B) equiv mil(B,A).

'maj:' M eg mil(A,B) ==> [M est_sur dro(A,B)] :- !.

evaluation(mil(p(Xa,Ya), p(Xb, Yb)), p(X,Y)) :-
     X is (Xa+Xb)/2,
     Y is (Ya+Yb)/2.

M eg mil(A,B) 'except:' [[A diff B] >> [], [A eg B] >> [ M eg A]] :- !.
```

figure 2 : clauses pour le symbole fonctionnel `mil`.

Ces clauses précisent :
- le profil de `mil` ;
- les permutations permises des arguments sans changer la signification ;
- le fait que si `m` est le milieu de [`a`, `b`], alors `m` est sur la droite `ab` ;
- l'interprétation analytique de `mil` ;
- et, finalement, les conditions de validité de `mil` et les cas dégénérés.

Il n'y a pas de clauses pour les prédicats `rmult`, `decomp` en ce qui concerne le symbole fonctionnel géométrique `mil`.

Pour le symbole fonctionnel `intercd`, nous avons les clauses de la figure 3 qui précisent :

- le profil de `intercd` ;
- le fait que `intercd` est un symbole ambigu ;
- la propriété de `intercd` d'être un symbole fonctionnel décomposable, et sa décomposition en relations participatives ;
- enfin, l'interprétation analytique de `interdd`.

```
profil(intercd, cercle x droite >> point):- !.

rmult(intercd) :- !.

decomp(intercd,[cercle/incid, droite/incid]):- !.

evaluation(intercd(c(p(Xo, Yo), R), d(A, B, C), Lres) :-
     resout_cd(Xo, Yo, R, A, B, C, Lres), !.
```

figure 3 : clauses pour le symbole fonctionnel `intercd`.

Le symbole `intercd` étant décomposable, il n'a pas besoin de clause pour le prédicat `equiv` : toutes les permutations possibles sont calculées grâce aux mécanismes de décomposition/recomposition décrits au chapitre 6. Par ailleurs, il n'y a ni mises à jour automatiques, ni considération de cas dégénérés pour `intercd`.

1.3. Symboles relationnels

Les attributs de la sorte `crg` des symboles relationnels géométriques `profil`, `pequiv`, `ptitre` et `pevaluation` (Cf. chapitre 4) sont respectivement implantés par les prédicats `pprofil`, `pequiv`, `ptitre` et `effectue` comme on peut le voir à la figure 4 avec les symboles relationnels `iso` (triangle isocèle en le premier argument) et '=p=' (égalité entre deux points).

```
% pour le symbole relationnel iso
pprofil(iso, point x point x point) :- !.

ptitre(iso, cond) :- !.

iso(A, B, C) pequiv iso(A, C, B).

effectue(iso(p(Xa, Ya), p(Xb, Yb), p(Xc, Yc))) :- !,
     E is abs((Xa-Xb)*(Xa-Xb)+(Ya-Yb)*(Ya-Yb) - (Xa-Xc)*(Xa-Xc)-(Ya-Yc)*(Ya-Yc)),
     E < 0.01.

% pour le symbole relationnel '=p='
pprofil('=p=', point x point).

ptitre('=p=', egalite).

effectue(p(X, Y) '=p=' p(XX, YY)) :- !,
     E is abs(X - XX) + abs(Y - YY),
     E < 0.01.
```

figure 4 : exemple de description de symboles relationnels

Il n'y a pas de clause concernant les prédicats d'égalité pour le prédicat `pequiv` : l'atomisation systématique des propriétés du raisonnement formel et la méthode d'unification se chargent de permuter éventuellement les arguments d'une telle relation. Par ailleurs, comme pour le prédicat equiv, la clause

```
Prop pequiv Prop.
```

assure que toute propriété est équivalente à elle-même.

La vérification des termes relationnels se fait à une *erreur numérique* près fixée ici arbitrairement à 0.01 : ceci peut faire échouer une interprétation théoriquement valide d'un programme de construction mais ça ne remet pas en cause la construction formelle.

2. Figure formelle, unification modulo la figure

La sorte *figure formelle* a été décrite au chapitre 6 comme l'association d'objets géométriques, de formes égalitaires, d'occurrences de relations géométriques et d'une table de synonymes. Cette association est codée sous forme de clauses dans la base de faits Prolog. Plus précisément, à l'aide du prédicat `fe`, nous avons regroupé dans une même clause un objet géométrique, ses représentants et ses participants. La table de synonymes est implantée également sous forme de faits de base à l'aide du prédicat `alias`.

Exemple.
La figure décrite au chapitre 3 (figure 4) est codée dans Progé sous forme de l'ensemble de faits de base, i.e. de faits clos, de la figure 5.

```
fe(a,point,0,donne,[],[d/droite/incid]).
fe(b,point,0,donne,[],[]).
fe(c,cercle,0,donne,[],[f/point/incid,e/point/incid,long00/long/rayon,
                                                  point00/point/centre]).
fe(point00,point,0,centre(c),[centre(c)],[]).
fe(long00,long,0,rayon(c),[rayon(c)],[]).
fe(dir00,dir,1,[],[],[d/droite/incid]).
fe(e,point,1,[c/cercle/incid],[],[d/droite/incid,c/cercle/incid]).
fe(f,point,1,[c/cercle/incid],[],[d/droite/incid,c/cercle/incid]).
fe(d,droite,1,[a/point/incid],[],[a/point/incid,f/point/incid,
                                        e/point/incid,dir00/dir/incid]).
fe(long01,long,1,[],[dist(f,b),dist(e,b)],[]).

a alias a.
b alias b.
c alias c.
point00 alias point00.
long00 alias long00.
d alias d.
dir00 alias dir00.
e alias e.
f alias f.
long01 alias long01.
```

figure 5 : figure formelle sous forme de faits de base.

L'*unification modulo* la figure se fait à travers les prédicats `pequiv`, pour les symboles relationnels, `equiv`, pour les symboles fonctionnels, à travers également la table des synonymes, les représentants d'un objet ou ses participants. A la différence de la description faite aux chapitres 4 et 6, nous avons utilisé les facilités d'unification et de retour arrière pour implanter le plus simplement possible les opérations de recherche et d'unification.

La *manipulation des entités d'une figure*, c'est-à-dire les retraits, les ajouts et les modifications se font à l'aide des prédicats prédéfinis de manipulation de la base de faits Prolog, à savoir `assert` et `retract`. Le fait d'agir ainsi par effets de bord pose les problèmes habituels : il est parfois difficile de savoir sur quelle figure exactement un prédicat est en train de travailler. Par ailleurs, nous perdons ainsi les possibilités de recherches exaustives en nous privant des facultés de retour-arrière pour les opérations de modification de la figure.

3. Raisonnement, moteur d'inférence

D'une manière similaire, la sorte `raisonnement` des raisonnements formels est décrite par une collection de clauses des prédicats `sommet` et `deriv`. L'implantation du raisonnement est très proche de la description faite au chapitre 7, et les exemples fournis à cette occasion conviennent ici aussi. La manipulation des sommets et des dérivations dans la base de fait se fait à l'aide des prédicats Prolog prédéfinis `assert`, `asserta` et `retract` avec les mêmes inconvénients que ceux décrits à la section précédente. Remarquons, par exemple, qu'en agissant de cette manière, il devient trés difficile de chercher tous les programmes de constructions possibles.

Les *règles* présentées dans les chapitres précédents sont écrites telles quelles en Prolog. Nous avons profité des facilités offertes par Prolog quant à la définition de nouveaux opérateurs pour donner une syntaxe acceptable à nos règles. Pour ajouter ou retrancher des règles, il faut donc éditer le fichier concerné et, finalement, programmer en Prolog.

Le *choix des règles* et des *propriétés* à exploiter est directement issu du fonctionnement en profondeur du moteur d'inférence de Prolog. Nous avons tenté de tempérer ceci à peu de frais en introduisant le nombre d'utilisations actives et passives comme ceci est décrit au chapitre 7.

4. Elaboration d'un programme de construction géométrique

Pour éviter d'utiliser un argument suplémentaire dans un bon nombre de prédicats, nous avons choisi de stocker temporairement les *instructions du programme de construction* à l'aide de clauses dans la base de fait. Le prédicat à deux arguments de nom `pcg` est utilisé à cet effet. Le premier argument est un numéro de programme et le second une instruction (Cf. figure 6). Chaque fois qu'un nouvel objet est construit, il est placé dans le programme de construction avec sa définition. S'il se trouve que la définition est ambiguë, i.e. accepte plusieurs valeurs numériques, une affectation de liste et une instruction d'itération sont insérées dans le programme. Lors de la pose d'une conditionnelle, le numéro de programme correspondant à la branche `sinon` est empilé avec la configuration à traiter. Au moment du dépilement, c'est ce numéro qui servira de numéro de programme courant.

A la fin de la construction formelle, tous les faits `pcg` sont rassemblés dans une structure arborescente conservant l'ordre dans lequel les clauses du prédicats `pcg` ont été mémorisée. Cette manière de faire assure qu'on a des programmes bien définis.

Exemple.
Après résolution du cas général de l'exercice présenté au chapitre 3, la base de faits contient les clauses de la figure 6. Dans cet exemple, les branches correspondant aux alternatives

`sinon` n'ont pas encore été élaborées. Lors de l'étude des cas dégénérés, Progé étudiera les programmes correspondant aux numéros 7, puis 4 et enfin 2.

```
pcg(0,a:=donne) .
pcg(0,b:=donne) .
pcg(0,c:=donne) .
pcg(0,point00:=centre(c)) .
pcg(0,long00:=rayon(c)) .
pcg(0,cercle00:=cdiam(b,a)) .
pcg(0,point02:=centre(cercle00)) .
pcg(0,long02:=rayon(cercle00)) .
pcg(0,long03:=dist(a,b)) .
pcg(0,si[a diff b]alors 1 sinon 2) .
pcg(1,droite01:=dro(a,b)) .
pcg(1,dir02:=dird(droite01)) .
pcg(1,long04:=dist(point00,b)) .
pcg(1,si[point00 diff b]alors 3 sinon 4) .
pcg(3,droite00:=dro(point00,b)) .
pcg(3,dir01:=dird(droite00)) .
pcg(3,list00:=intercd(cercle00,droite00)) .
pcg(3,pour point01 dans list00 faire 5) .
pcg(5,long05:=dist(point01,a)) .
pcg(5,si[point01 diff a]alors 6 sinon 7) .
pcg(6,d:=dro(point01,a)) .
```

figure 6 : fragment d'un programme de construction sous forme de clauses.

5. Interfaçage, environnement.

Le *lancement* de Progé et son *fonctionnement* se font à l'intérieur de l'interpréteur Prolog comme on peut le voir dans le mode d'emploi sommaire qui suit. Les interfaces utilisateur/Progé et expert/Progé sont assez sommaires et se résument à quelques outils d'explorations de la figure et du raisonnement formels, d'exploration de la figure numériques et, devons-nous l'avouer ? d'outils de débogage de Progé.

Aprés avoir lancé, à partir du répertoire contenant les fichiers nécessaires à Progé, une version de Delphia Prolog (on un Prolog standard sur PC), contenant les primitives de la librairie graphique Pixia, on charge le fichier `proge` par :

```
?- consult(proge).
```

Puis, on lance le chargement de Progé par :

```
?- run.
```

L'écran affiche les fichiers chargés.
On peut ensuite charger un énoncé, ou saisir celui-ci dans l'éditeur ligne appelé par l'instruction `shell`. Par exemple, on peut taper :

```
?- consult(exo5).
```

ou, pour l'édition à l'aide de Progé :

```
?- shell.
```

```
      commande : h

 C : edition des contraintes
 D : edition des declarations
 L : listing d un enonce
---------------------------------
 G : lance la resolution
---------------------------------
 F : interrogations sur la figure
 R : interrogations sur le raisonnement
---------------------------------
 P : requete Prolog
---------------------------------
 A : quitter le shell
 Q : quitter Prolog
 H : aide
commande : d

editions des declarations : h

--- editions de declarations :
 L : lister les declarations
 A : ajouter une declaration
    (fin = arret de l ajout)
 I : inserer une declaration
 M : mouvoir une declaration
 N : renumeroter les declarations
 D : effacer une declaration
 Z : effacer toutes les declarations
 H : aide
 Q : quitter l edition

editions des declarations : a

ed> point :: a donne.
```

... les déclarations sont introduites ici ...

```
ed> fin.

editions des declarations : q

commande : c

editions contraintes : h

--- editions de contraintes :
 L : lister les contraintes
 A : ajouter une contraintes
    (fin = arret de l ajout)
 I : inserer une contrainte
 M : mouvoir une contrainte
 N : renumeroter les contraintes
 D : effacer une contrainte
 Z : effacer toutes les contraintes
 H : aide
 Q : quitter l edition

editions contraintes : a

ec> dist(a,c) '=l=' k.

ec> dist(b,c) '=l=' l.

ec> fin.
editions contraintes : q

commande : l

----- declarations :
```

```
0 dec:point::a donne
1 dec:point::b donne
2 dec:point::c cherche
3 dec:long :: k donne
4 dec:long :: l donne

----- contraintes :
0 cont:dist(a,c)=l=k
1 cont:dist(b,c)=l=l


commande : q

confirmation retour a MS-DOS (o/n) : n

*** Commande non reconnue
commande : a

true

?-
```

Le lancement de la résolution est obtenu par

```
?- roule.
```

Après quelques informations, plus ou moins fournies suivant le niveau de débogage en cours, sur le déroulement de la résolution, l'utilisateur est sollicité pour choisir les éléments de base (les éléments `donnés`) à la souris dans la fenêtre définie par Progé.

Un petit module permet d'interpréter des commandes pour explorer les figures numériques obtenues. On peut, par exemple, visualiser toutes les solutions particulières, modifier la fenêtre d'affichage, demander une réinterprétation complète ou partielle (en changeant la valeur d'un seul élément donné) du programme de construction trouvé, sauver ou lire un programme de construction précédemment écrit sur disque. La liste des commandes disponibles est obtenue en tapant `h` au prompt, on obtient :

```
r: h

 Aide :
d      : dessine
s      : solution suivante
p      : solution precedente
c      : changer un objet de base
i      : nouvelle interpretation (complete)
x      : coordonnees
e      : shell
p      : sauver (temporairement) un programme
w      : ecrire un programme sur disque
r      : recuperer un programme
a      : afficher le programme
k      : kill programmes sauves
f      : modifier la fenetre
q      : quitter
```

Par ailleurs, le prédicat `shell` lance un module d'exploration de la figure et du raisonnement formels courants. Cette exploration sert essentiellement au débogage et à la confection d'exemples.

Conclusion

Au milieu de ce mot qu'il essayait de dire,
Au milieu de sa joie et de son rire fous,
Soudain, tout doucement, il avait disparu -
Car ce Snark, c'était un Boojum, figurez-vous.

Nous avons présenté dans ce rapport un cadre permettant l'automatisation de constructions géométriques à la règle et au compas. Ce cadre a été testé avec succès à l'aide d'un prototype nommé Progé dans lequel l'univers géométrique est une donnée du moteur d'inférence au même titre que l'énoncé de l'exercice à résoudre. Contrairement aux systèmes experts usuels, cet univers géométrique n'est pas uniquement constitué de règles de production : les particularités des sortes géométriques, symboles fonctionnels et relationnels sont prises en compte dans une certaine mesure à l'aide d'attributs. Par ailleurs, notre démarche s'apparente plutôt à l'EIAO et Progé se distingue notablement des logiciels d'automatisation de constructions géométriques de la CAO.

1. Rapports avec l'existant

1.1. Construction Assistée par Ordinateur

Divers systèmes de constructions géométriques sont actuellement expérimentés et même commercialisés dans le domaine de la CAO. La plupart de ces systèmes suivent une démarche trés éloignée de la nôtre.

Dans certains logiciels de CAO comme UnigraphicsII, sont mises en oeuvre des méthodes variationnelles comme la méthode de Newton-Raphson [Roller & al 88]. Celles-ci sont instables par nature et souffrent de nombreux défauts comme la non convergence ou la convergence vers une seule solution qui n'est peut être pas celle voulue. Ces méthodes sont inutilisables dans le cadre d'un logiciel d'EIAO sur les constructions géométriques.

Dans [Owen 91], J. C. Owen met en oeuvre une méthode algébrique numérique en essayant de se restreindre à la résolution d'équations de degré au plus 2, en ordonnant la prise en compte des contraintes. Ceci est généralement possible dans le cadre de la CAO où on a plutôt des *systèmes quasi triangulaires* [Roller & al 88] mais cette démarche n'est pas généralisable à tous les problèmes de construction à la règle et au compas.

L'intéressante approche de Sunde [Sunde 87], [Schoneck & Verroust 88] systématise la triangulation par une méthode de rigidification progressive. Les seules contraintes prises en compte sont des contraintes de distance et d'angle ce qui est souvent suffisant en CAO. Cette approche est numérique, mais peut être adaptée à une résolution formelle : la spécification et l'implantation d'une telle méthode est actuellement en cours à l'Université Louis Pasteur.

Finalement, l'approche qui s'apparente le plus à la nôtre est celle d'Aldefeld [Aldefeld 88], [Aldefeld 91] qui utilise des moyens purement géométriques pour obtenir une construction formelle qui est ensuite interprétée. Les différences d'objectifs induisent des différences de démarche vis à vis de Progé :
- l'énoncé de construction est constitué de l'ensemble des cotations sur une esquisse. Cet énoncé est éventuellement complété par des contraintes implicites ;
- une seule solution est souhaitée qui doit se rapprocher le plus possible de l'esquisse proposée par le dessinateur ;
- l'univers géométrique est bien déterminé et est constitué de règles modélisant les constructions élémentaires usuelles en dessin technique.
Les notions d'univers géométriques, de cas dégénérés ou de cas particuliers, et de programme de construction avec structures de contrôle ne sont donc pas développées.

1.2. En EIAO

La géométrie élémentaire est un sujet exploré depuis longtemps en démonstration automatique [Gelernter 63], [Tarski 59], [Coelho & Pereira 79], [Chou 88], [Wu 84] et en didactique [Anderson 83], [Chouraqui & Inghiltera 90], [Py 90]. Les constructions géométriques ont également été étudiées à travers des logiciels comme Euclide [Allard & Pascal 86], Géométrie plane [PIE 92], Geophile [Braun 88], Cabri-géomètre [Baulac 90]. Mais, bizarrement, le problème de la résolution automatique de constructions géométriques a été, à notre connaissance, trés peu étudié. On peut citer l'article de Scandura et al. [Scandura & al 74], le logiciel Tricon [Barz & Holland 89], le prototype Minip [Koch & al 89], et le programme Geom3 [Buthion 75].

Le seul programme général ayant donné effectivement des résultats semble être Geom3 développé par M. Buthion [Buthion 75]. Notre travail s'inspire à la fois d'idées de Minip et de Geom3. Il se distingue de Geom3 notamment du point de vue de l'utilisateur par la considération d'énoncés géométriques et la production d'un programme de construction interprétable traitant des exceptions et des cas particuliers; et du point de vue de l'expert par la possibilité d'étendre l'univers géométrique.

2. Bilan

Progé résout actuellement les exercices donnés à l'annexe 2. Notre approche a toujours été empreinte de pragmatisme : l'univers géométrique, en particulier la base de règles, utilisée par Progé a été mis au point pour résoudre les exercices que nous lui proposions. Des exercices classiques de construction à la règle et au compas notamment en rapport avec la géométrie projective ne sont pas résolus par Progé actuellement. Ceci n'est pas rédhibitoire, un expert géomètre pourrait facilement compléter l'univers géométrique de sorte que ce genre d'exercices soit résolu.

Il se pose naturellement la question de la caractérisation des exercices résolubles par Progé ou plus exactement :

Etant donné un univers géométrique, quelle est la classe des exercices résolus par Progé dans cet univers géométrique ?

En fait, nous ne savons même pas répondre à la question de savoir quels sont les exercices *actuellement* résolus par Progé. D'ailleurs, pour ces deux questions, il faudrait préciser le sens de résoudre. Nous proposons de dire qu'un exercice est *résolu au sens fort* si dans tous les cas de figures où il existe un nombre fini de solutions, on est capable de fournir une construction donnant toutes ces solutions. Un exercice, pour lequel il existe des solutions formelles, est *résolu au sens faible* si dans au moins un cas de figure des solutions sont trouvées. Par exemple, l'exercice `buth32b` dont une solution est proposée à l'annexe 1, est résolu au sens faible par Progé, mais pas au sens fort .

Pour l'interprétation usuelle de l'univers géométrique de Progé, notre implantation n'est pas complète. C'est ainsi que deux droites ayant deux points communs ne sont parfois pas reconnues comme égales par Progé.

Exemple.
Soient les trois formes égalitaires
(1) `d = dro(a, b)`
(2) `d1 = dro(a, c)`
(3) `d = dro(e, c)`
La mise à jour de la figure courante par adjonction de ces formes égalitaires provoquera l'ajout des faits suivants :
pour (1) (en supposant qu'il n'y ait pas déjà `dro(a, b)` dans la figure)

```
a incidentpd d
b incidentpd d
```

pour (2)

```
a incidentpd d1
c incidentpd d1
```

à ce moment, il n'y a aucune raison pour que `d1=d2`. Lors de l'adjonction de (3), Progé cherche si `dro(e, c)` représente un objet géométrique dans la figure, ça n'est pas le cas dans cet exemple. Ainsi les faits :

```
e incidentpd d
c incidentpd d
```

sont ajoutés à la figure. Autrement dit, les points `a` et `c` appartiennent aux droites `d` et `d1` et Progé n'a pas détecté l'égalité entre ces deux droites.
Il faut remarquer que si ces formes égalitaires avaient été rencontrées dans l'ordre (1), (3) et (2) alors Progé aurait trouvé que les droites `d` et `d1` étaient égales.

Ce point précis aurait pu être évité moyennant des procédures de contrôle supplémentaires lors de l'ajout de participants. Outre le problème du temps de résolution, la question est : est-ce que cela suffit à bien modéliser l'égalité ou sinon jusqu'où faut-il aller ? N'ayant pas approfondi cette question pour l'instant, l'implantation de ces procédures de contrôle n'a pas été faite.

Remarquons que ces quelques points, ainsi que d'autres évoqués dans ce rapport, soulignent l'importance du travail qu'il reste à faire.

3. Perspectives

3.1. Perspectives pratiques

Ce travail peut être enrichi pour servir de base à un logiciel d'EIAO résolvant des problèmes de construction géométrique. Dans un tel logiciel, il serait évidemment nécessaire d'approfondir la question de l'interface utilisateur/logiciel tant du point de vue de l'introduction d'un énoncé et de l'exploration des solutions proposées que des explications sur la construction trouvée.

Dans la même optique, il paraît important de pouvoir sélectionner un univers géométrique correspondant à une situation étudiée en cours pour éviter les solutions incompréhensibles par un élève ou, au contraire, trop triviale. Ceci ne peut se faire qu' à travers une interface expert/logiciel qui doit être très conviviale et le plus "intelligente" possible.

Dans le même ordre d'idée, pour fournir une palette d'outils cohérents à un utilisateur en géométrie, il nous semble intéressant de réfléchir à l'interfaçage d'un tel logiciel avec des logiciels de constructions impératifs classiques et/ou des logiciels de démonstration en géométrie élémentaire.

Il paraît en outre souhaitable de revoir l'implantation dans un autre langage de programmation que Prolog, en repensant éventuellement certains algorithmes de manière à améliorer l'efficacité de Progé. Il serait alors envisageable, en définisant un univers géométrique adéquat, d'interfacer Progé avec un logiciel de DAO/CAO. Un autre sujet de recherche seraît alors l'étude des constructions géométriques en 3D en définissant les outils adéquats [Brüderlin 86], [Martin 88].

3.2. Perspectives théoriques

Comme nous l'avons souligné, la grande question laissée en suspens est celle de la caractérisation de la classe des exercices résolus en fonction d'un univers géométrique donné. Cependant d'autres problèmes comme l'étude de la complexité en temps et en espace de cette approche nous paraissent digne d'intérêt.

Un de nos grands soucis est également de revoir - de repenser !- la spécification algébrique que nous avons fourni afin que celle-ci soit tout à fait formelle et complète.

Bibliographie

[Aldefeld 88], B. Aldefeld, *Variation of geometries based on a geometric-reasoning method*, CAD, vol.20, n°3, 1988.

[Aldefeld & al 91], B. Aldefeld, H. Malberg, H. Richter and K. Voss, Rule-Based Variational Geometry in Computer-Aided Design, in Artificial Intelligence in Design, D. T. Pham editor, Springer-Verlag, 1991.

[Allard & Pascal 86], J. C. Allard et C. Pascal, *Euclide, un langage pour la géométrie plane*, Cédic-Nathan, Paris, 1986.

[Allen & al 90], R. Allen, P. Nicolas et L. Trilling, *Figure correctness in an expert system for teaching geometry*, pub. interne, 1990.

[Anderson 83], Anderson, *Acquisition of proof skills in geometry, in Machine Learning : an IA approach*, TIOGA, 1983, pp 191-218.

[Autodesk 85], Autodesk A. G., *Autocad*TM, v. 2.1, User ref. pub. 106-006F, 1985.

[Baulac 90], Y. Baulac, *Un micromonde de géométrie, Cabri-géomètre,* Thèse de l'Université J. Fourier, Grenoble, 1990.

[Barz & Holland 89], *Intelligent tutoring systems for training in geometrical proof and construction problems*, GIT Report, 1989.

[Bergstra & al 89], J. A. Bergstra, J. Heering and R. Klint (eds.), *Algebraic Specifications*, Addison-Wesley, 1989.

[Bouma & Walter 89], Bouma and Walter, , in Algebraic Specifications, J. A. Bergstra & al editors, Addison-Wesley, 1989, pp - .

[Braun 88], G. Braun, *Sur la programmation de constructions géomètriques*, Thèse de l'Université L. Pasteur, Strasbourg, 1988.

[Brüderlin 88], B. Brüderlin, *Automatizing geometry proofs and constructions, Computational Geometry and its Application*, LNCS, 1988.

[Brüderlin 86], B. Brüderlin, *Constructing three-dimensional geometric objects defined by contraints*, in Interactive 3D Graphics, pp 111-129, october 1986.

[Buthion 75], M. Buthion, *Un programme qui résout formellement des problèmes de constructions géométriques*, Thèse de l'université Pierre et Marie Curie, Paris 6, 1975.

[Buthion 79], M. Buthion, *Un programme qui résout formellement des problèmes de constructions géométriques*, RAIRO Informatique/Computer Science, vol 13 n°1, 1979.

[Carrega 89], J.C. Carrega, *Théorie des corps, la règle et le compas*, Herman, 1989.

[Carroll 89], L. Carrol, *La Chasse au Snark*, in Oeuvres, coll. Bouquins, Robert Laffont éd., novembre 1989.

[Chang & Lee 73], C. L. Chang and R. C. Lee, *Symbolic logic & mechanical theorem proving*, Academic Press, New York, 1973.

[Chen 92], G. Chen, *Les constructions à la règle et au compas par une méthode algébrique*, rapport de DEA d'informatique, Centre de Recherche en Informatique de l'Université Louis Pasteur, Strasbourg 1992.

[Chou 84], S.C. Chou, *Proving elementary geometry theorems using Wu's algorithm*, Contemporary Mathematic, Vol. 29, American Mathematical Society, 1984.

[Chou 88], S.C. Chou, *Mathematical geometry theorem proving*, Univ. of Texas, col. Math. & its applications, Austin, 1988.

[Chouraqui & Inghiltera 90], E. Chouraqui et C. Inghiltera, *ARCHIMEDE : an intelligent tutoring system for plane geometry*, pub. int. CNRS Marseille GRTC/416/Juin 90.

[Coelho & Pereira 79], H. Coelho et L. M. Pereira, *A Prolog geometry theorem prover*, Report 525, Laboratorio Nacional de Engenharia Civil, Lisbon, 1979.

[Colette 73], J. P. Colette, *Histoire des mathématiques*, tome 1, éditions du renouveau pédagogique inc, Montréal, 1973, pp 61-82.

[Delahaye 87], J. P. Delahaye, *Outils logique pour l'intelligence artificielle*, Eyrolles, Paris, 1987 .

[Ducrin 85], A. Ducrin (nom collectif), *Programmation*, vol. 1 & 2, Dunod, 1985.

[Dufourd 90], J.F. Dufourd, *Programmation et résolution de problèmes de constructions géométriques*, journées du GRECO-Programmation Bigre-Globule n° 70, 1990, pp 136-147.

[Dufourd 90, 2], G. Dufourd et J. F. Dufourd, *Varied universes and tools to begin programming*, IFIP 5th World Conf. on Computers and Education, Sydney (Australia), North Holland, 1990, pp 265-270.

[Ehrig & Mahr], H. Ehrig and B. Mahr, *Fundamentals of algebraic specification 1: equations and initial semantic*, EATC Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[Fuller & Prusinkiewicz 89], N. Fuller and P. Prusinkiewicz, *Application of Euclidean Constructions to Computer Graphics*, The Visual Comp. 5, 1989, pp 53-67.

[Gelernter 63], H. Gelernter, *Realization of a geometry-theorem proving machine,* Computer and Thought, Mac Graw-Hill, 1963.

[Gelernter & al 63], H. Gelernter, J.R. Hansen, D.W. Loveland, *Empirical explorations of the geometry theorem proving machine*, Computer and Thought, Mac Graw-Hill, 1963, pp 134-152.

[Goguen & Meseger 89], J. A. Goguen and J. Meseger, *Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations*, Technical Report SRI-CSL-89-10, SRI International Computer Science Lab, July 1989.

[Goguen 87], J. A. Goguen, *Modular algebraic specification of some basic geometrical constructions*, in Artificial Intelligence, Special Issue on Computational Geometry, D. Kapur and J. Mundy editors, 1988, pp 123-153.

[Goguen & al 92], J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi et J. P. Jouannaud, *Introducting OBJ,* in Applications of Algebraic Specification Using OBJ, J.A. Guogen, D. Coleman and R. Gallimore editors, Cambridge University Press, 1992.

[Hayes-Roth & al 83], F. Hayes-Roth, D. A. Waterman and D. B. Lenat, *Building expert systems*, Addison Wesley, London, 1983.

[Hilbert 71], D. Hilbert, *Les fondements de la géométrie*, P. Rossier (éd.), Dunod, Paris, 1971.

[Kalkbrenner 91], M. Kalkbrener, *Elimination theory*, preprint, Research Institute for Symbolic Computation, Johannes Kepler Universität Linz, Austria 1991.

[Kaltofen 83], E. Kaltofen, *Factorization of polynomials*, Computer Algebra Symbolic and Algebraic Computation, B. Buchberger & al editors, Springer-Verlag, 1983, pp 95-113.

[Kin & al 89], N. Kin, T. Noma et T. L. Kunii, PictureEditor : *A 2D Picture Editing System Based on Geometric Constructions and Constraints*, Proc. Comp. Graphics Int.'89, Leeds, Springer-Verlag, 1989, pp 193-208.

[Koch & al 92], B. Koch, I. Riegel, P. Schreck, *Minip, un programme Prolog qui résout des problèmes de construction de triangles*, bulletin de l'EPI, n°66, juin 1992.

[Kramer 90] G. A. Kramer, *Geometric reasoning in the kinematic analysis of mechanisms, technical report,* TR. 91.02, Schlumberger Laboratory for Computer Science, 1990.

[Lebesgue 50], H. Lebesgue, *Leçons sur les constructions géométriques*, Gauthier-Villars, Paris, 1950.

[Lehmann & Bkouche 88], D. Lehmann et R. Bkouche, *Initiation à la géométrie*, Presse Universitaire de France, Paris, juin 1988.

[Light & al 81], Light, Lin and Gossard, *Variational geometry in CAD*, in Computer Graphic, vol. 15 n°3, August 1981.

[Martin 88], D. Martin et P. Martin, *An expert system for polyhedra modelling*, Proc. Eurographics'88, Nice, North Holland, 1988.

[Monfroy & al 90], E. Monfroy, M. Rusinowitch et R Schott, *Spécification Opérationnelle des Contraintes Géométrique*, Actes des Journées de géométrie algorithmique, INRIA sophia-Antipolis, Juin 1990, pp 41-50.

[Owen 91], J. C. Owen, *Algebraic Solution for Geometry from Dimensional Constraints*, ACM 1991, pp 397-407.

[Petersen 90], J. Petersen, *Problèmes de constructions géométriques*, rééd., Jacques Gabay, 1990.

[Pintado 92], M. Pintado, *Apprentissage et boites noires*, JFAEC, Avril 92.

[Py 90], D. Py, *Reconnaissance de plan pour l'aide à la démonstration dans un tuteur intelligent de la géométrie*, Thèse de l'université de Rennes 1, Rennes, 1990.

[Requicha 80], A. A. G. Requicha, *Representation for rigid solids*, ACM Comp. Survey, vol 12 n° 4, pp 437-464, 1980.

[Roller & al 88], D. Roller, F. Schonek et A. Verroust, *Dimension driven geometry in CAD*, pub. int, LIENS, Ecole Normale Supérieure, Paris, 1988.

[Scandura & al 74], J.M. Scandura, J.H. Durnin, W.H. Wulfeck II, *Higher Order Rule Characterization of heuristics for compass and straight edge constructions in geometry*, in Artificial Intelligence 5, pp 149-183, 1974.

[Schonek & Verroust 88], F. Schonek et A. Verroust, *A rule oriented method to parametrize CAD design*, pub. int., LIENS, Ecole Normale Supérieure, Paris, 1988.

[Schreck 90], P. Schreck,*Constructions à la règle et au compas, quelques remarques*, rapport de recherche, Centre de Recherche en Informatique, Université Louis Pasteur, novembre 90.

[Schreck 91], P. Schreck, *Automatisation des constructions géométriqes sous contraintes*, Actes des deuxièmes Journées EIAO de Cachan, Editions de l'Ecole Normale Supérieure de Cachan, pp. 61-75, janvier 91.

[Schreck 91], P. Schreck, *Modélisation d'une figure géométrique adaptée aux problèmes de constructions*, Actes des Journées GROS PLAN de Lille, pp. 131-139, décembre 91.

[Schreck 92]], P. Schreck, *Automatisation des constructions géométriques, état de Progé*, rapport de recherche, Centre de Recherche en Informatique, Université Louis Pasteur, juin 92.

[Serrano 91], D. Serrano, *Automatic Dimensioning in Design for Manufacturing*, ACM 1991, pp 379-386.

[Sortais 87], Sortais et Sortais, *La géométrie du triangle*, Hermann, 1987.

[Sunde 87], G. Sunde, *A CAD system with declarative specification of shape*, Eurographics Workshop on Intelligent CAD System, April 1987.

[Tarski 59], A Tarsky, *That is elementary geometry*, Axiomatic method with a special reference to geometry and physics, edited by L. Henkin, P. Suppes and A. Tarski, Amsterdam, 1959.

[Toussaint 90], G. T. Toussaint, *A new look at Euclid's second proposition*, Technical report n° SOCS 90-21, School of Computer Science, Mc Gill, November 90.

[Viry 78], G. Viry, *Factorisation des polynômes à plusieurs variables à coefficients entiers*, RAIRO Informatique Théorique, 12 (1978), pp 305-318.

[Wang 76], P. S. Wang, *Factoring multivariate polynomials over algebraic number fields*, Math. Comp., vol. 30, 1976, pp 324-336.

[Weinberger & Rothschild 76], P. J. Weinberger and L. P. Rothschild, *Factoring multivariate polynomials over algebraic number fields*, ACM Trans. on Mathematical Software vol. 2, n° 4, 1976, pp 335-350.

[Winston 88], Winston, *L'intelligence artificielle*, col. IIA, InterEdition, 1988.

[Wirsing 90], Wirsing, *Algebraic Specifications*, in Handbook of theoretical Comp. Science, tome 2, J. van Leeuwen editor, Elsevier Science Publishers BV, 1990.

[Wu 84], W. T. Wu, *Basic principles of mechanical theorem proving in elementary geometries*, J. Sys. Sci. and Math. Scis., vol. 4 (1984), pp 207-235.

[Yokoyama & al], K. Yokoyama, M. Noro and T. Takeshima, *Computing primitive elements of extension fields*, J. Symbolic Comp., vol. 8 (1989), pp 553-580.

Annexe 1
Exemples de résolutions

Dans cette annexe, nous donnons dans leur integralité la résolution de trois exemples évoqués dans les chapitres précédents. Ces exemples illustrent la classe des exercices actuellement résolus par Progé. Pour chaque exercice, nous donnons l'énoncé en clair, suivi de la traduction dans le langage d'énoncé de Progé, puis le programme de construction trouvé et une copie d'écran correspondant à une interprétation.

1. Exo12b

Ce premier exemple est un exercice classique de classe de Première Scientifique. Plusieurs constructions sont possibles en utilisant notamment une homothétie. Voici la solution proposée par Progé.

Enoncé (exo12b). Construire un cercle **s** tangent à un cercle **c** donné et à une droite **d** donnée en un point **a** donné sur **d** (Cf. figure 7).

```
0 'dec:' cercle :: c donne.
1 'dec:' point :: a1 donne.
2 'dec:' point :: a2 donne.
5 'dec:' droite :: d nomme dro(a1,a2).
3 'dec:' point :: b mentionne.
4 'dec:' cercle :: s cherche.

0 'cont:' tangentcc(s,c,b).
1 'cont:' tangentcd(s,d,a1).
```

Programme de construction.

```
c:=(donne)
point00:=centre(c)
long00:=rayon(c)
a1:=(donne)
a2:=(donne)
d:=dro(a2,a1)
dir00:=dird(d)
droite00:=dorth(d,a1)
dir01:=dird(droite00)
droite05:=dpp(droite00,point00)
dir06:=dird(droite05)
list00:=intercd(c,droite05)
pour point03 dans list00 faire
   long03:=dist(point03,a1)
   si [point03 diff a1] alors
      droite06:=dro(point03,a1)
      dir07:=dird(droite06)
      list01:=intercd(c,droite06)
      pour b dans list01 faire
         long04:=dist(b,a1)
         si [b diff a1] alors
```

```
            droite01:=dro(b,a1)
            dir02:=dird(droite01)
            droite02:=med(b,a1)
            dir03:=dird(droite02)
            point01:=interdd(droite02,droite00)
            long02:=dist(point01,point00)
            droite03:=dro(point01,point00)
            dir04:=dird(droite03)
            droite04:=dorth(droite03,b)
            dir05:=dird(droite04)
            long01:=did(point01,droite04)
            list02:=cr2p(long01,b,a1)
            pour s dans list02 faire
               fin
            fin
          sinon
            echec
            fin
         fin
      fin
    sinon
      echec
      fin
   fin
verifier(tangentcc(s,c,b))
verifier(tangentcd(s,d,a1))
verifier(a1 diff a2)
fin
```

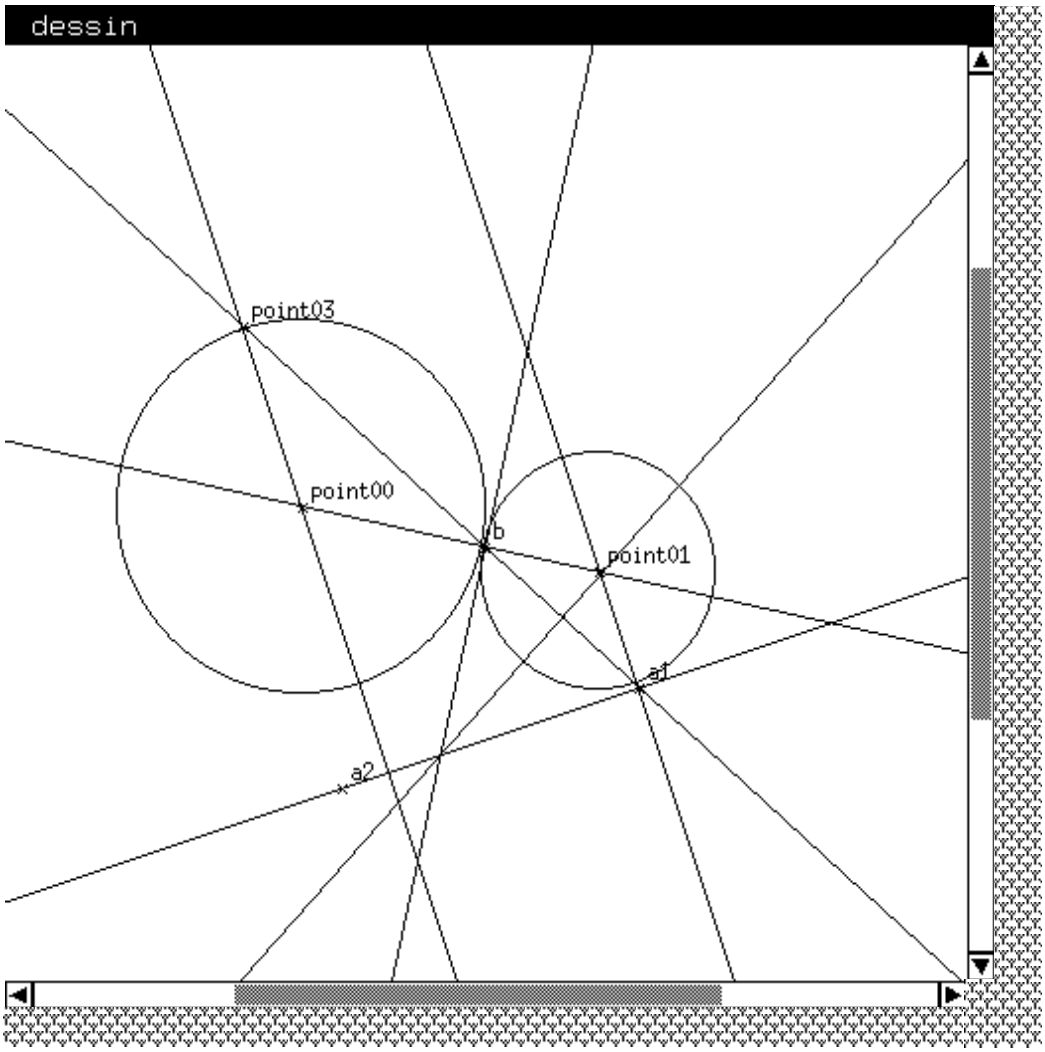

Figure 1 : copie d'écran 1.

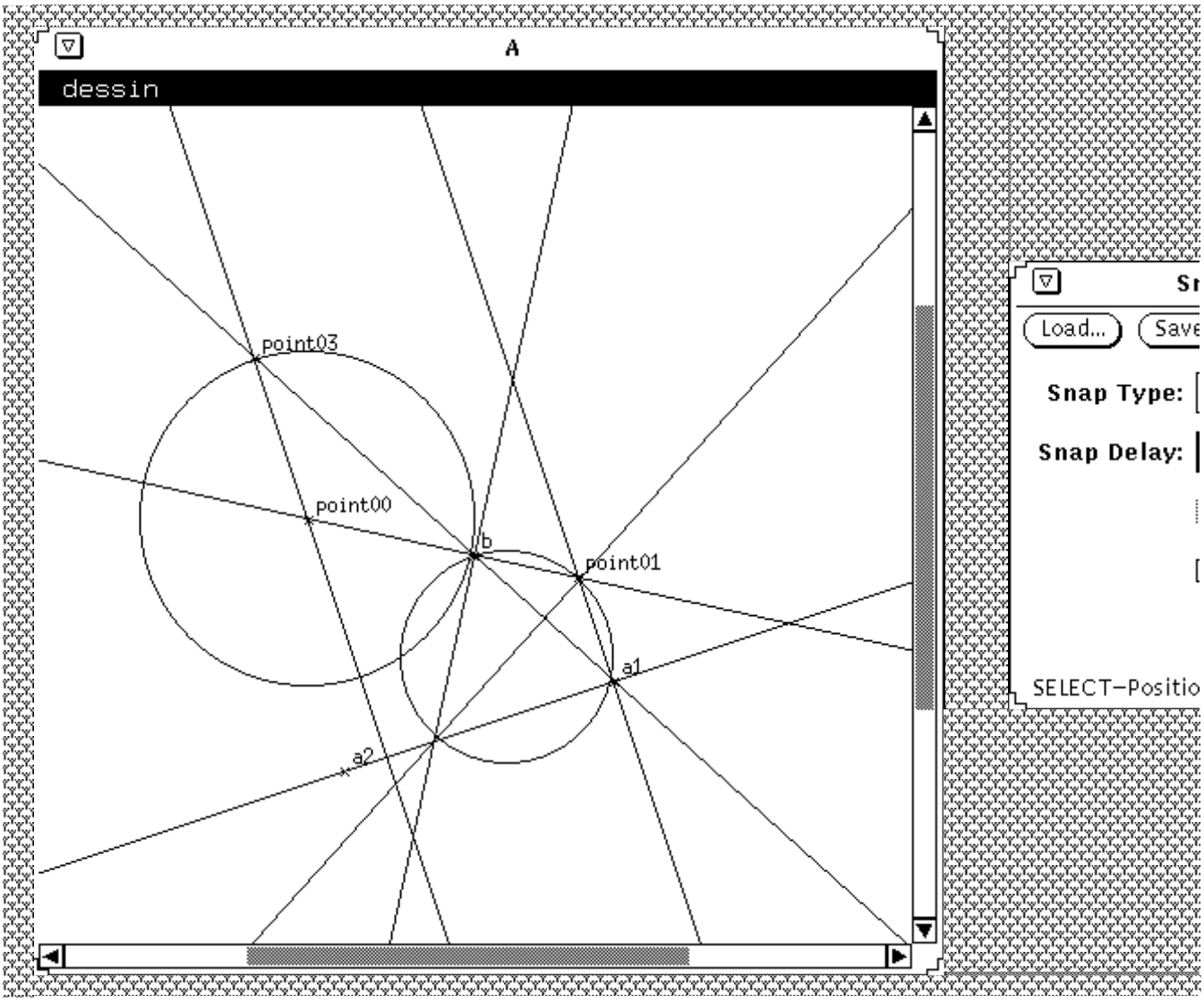

Figure 2 : copie d'écran 2 (fausse solution) .

Les figures 1, 2, 3 et 4 sont des copies d'écran obtenues à l'interprétation du programme de construction précédent sans tenir compte des instructions de vérification. Ainsi, les figures 2 et 4 représentent des constructions qui ne sont pas des soltuions de l'énoncé `exo12b`. Ces deux figures sont obtenues en réinterprétant le terme ambigu `cr2p(long01, b, a1)` qui représente un des deux cercles définis par le rayon `long01` et passant par les deux points `b` et `a1`. Sans tenir compte des vérifications, Progé trouve au total 8 constructions différentes pour ce programme de construction. Seules deux d'entre elles sont solutions de l'énoncé `exo12b`.

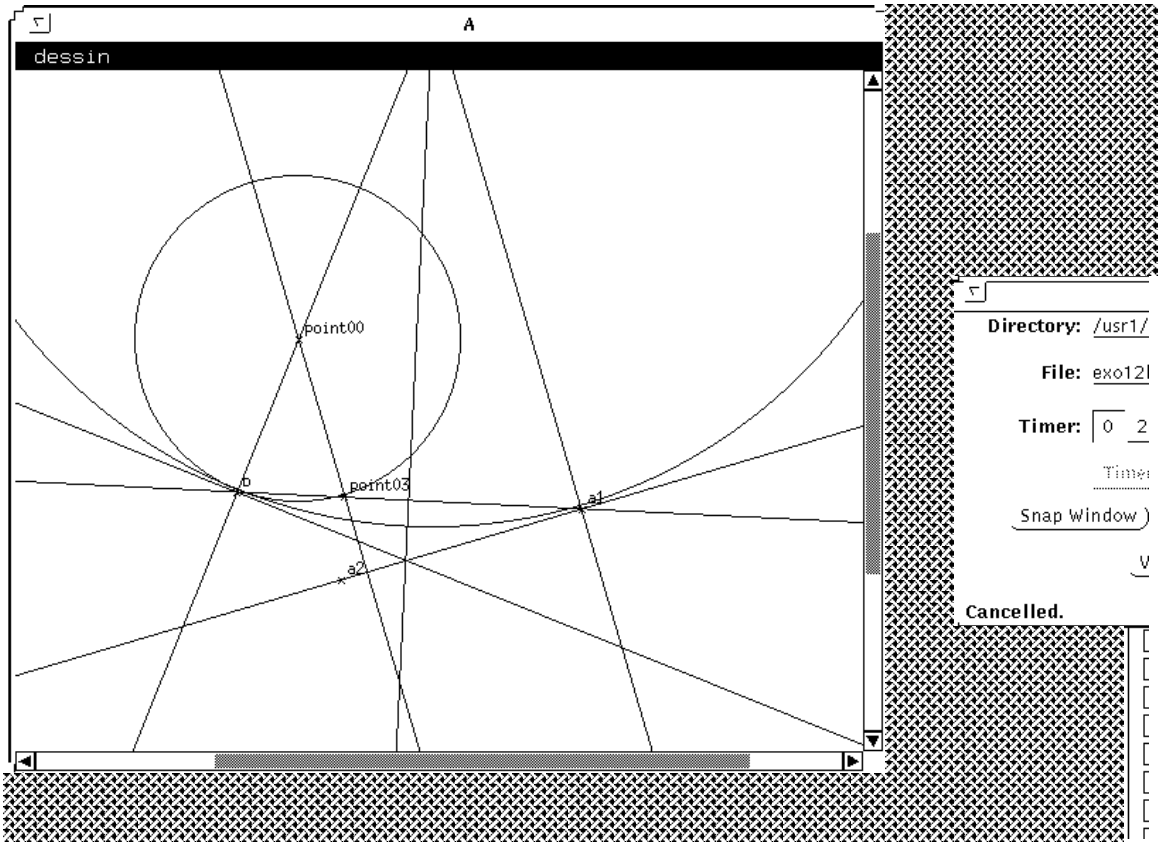

figure 3 : deuxième solution correcte.

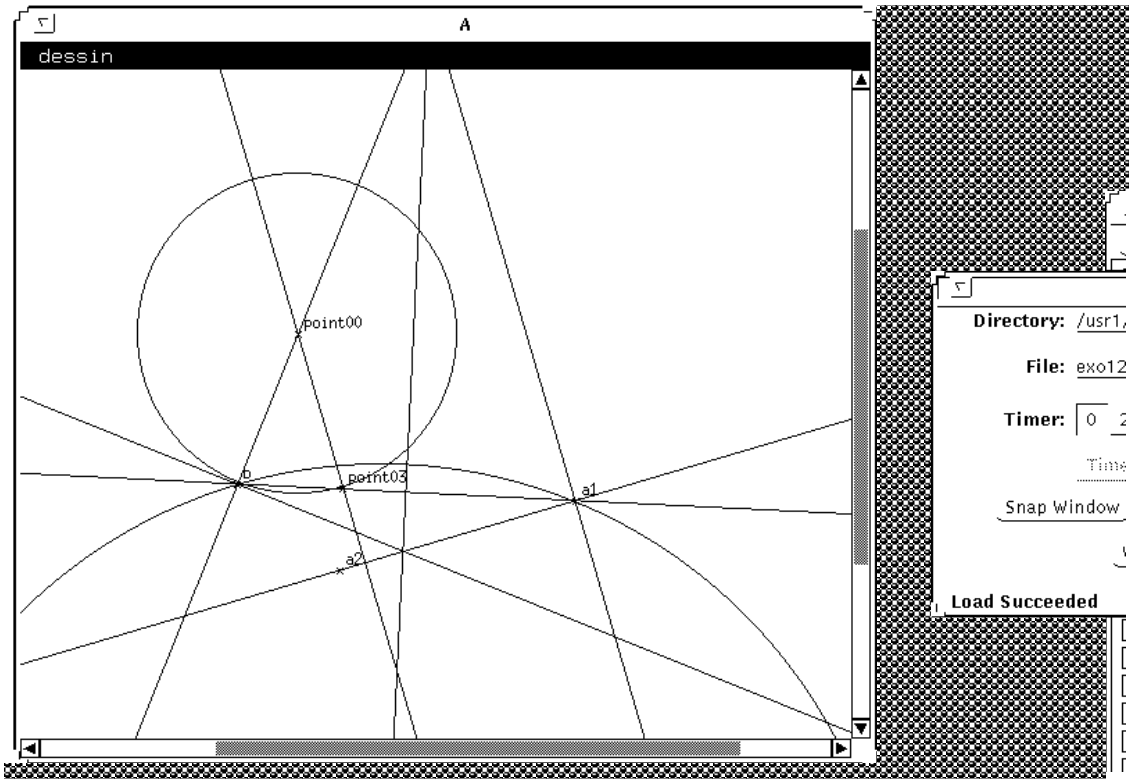

figure 4 : une fausse solution.

2. Exo8b

Cet exercice utilise la règle des bissectrices (règle numéro 7). Plusieurs méthodes de résolution sont là aussi possibles. Dans [Buthion 76], M.Buthion expose une solution trouvée par son programme Geom3. La solution proposée par Progé est sensiblement différente.

Enoncé (exo8b). On cherche à construire un cercle **s** tangent au cercle **c** donné en un point **a**, donné, et à une droite **d** donnée (Cf. figure 2 du chapitre 8).

```
0 'dec:' point :: a donne.
0 'dec:' point :: o donne.
1 'dec:' droite :: d donne.
2 'dec:' cercle :: c nomme ccp(o,a).
3 'dec:' cercle :: s cherche.
4 'dec:' point :: b mentionne.

1 'cont:' tangentcc(s,c,a).
2 'cont:' tangentcd(s,d,b).
```

Programme de construction.

```
a:=(donne)
o:=(donne)
d:=(donne)
dir00:=dird(d)
c:=ccp(o,a)
long00:=rayon(c)
long03:=dist(o,a)
droite02:=dro(a,o)
dir03:=dird(droite02)
droite03:=dorth(droite02,a)
dir04:=dird(droite03)
si [dird(droite03)diff dird(d)] alors
   list00:=bis(droite03,d)
   pour droite04 dans list00 faire
      dir05:=dird(droite04)
      point02:=interdd(d,droite03)
      point00:=interdd(droite04,droite02)
      long02:=dist(point00,o)
      s:=ccp(point00,a)
      b:=intercd(s,d)
      fin
   fin
 sinon
   si [dird(droite03)eg dird(d),droite03 diff d] alors
      droite05:=dmd(droite03,d)
      dir06:=dird(droite05)
      point00:=interdd(droite05,droite02)
      long02:=dist(point00,o)
      s:=ccp(point00,a)
      b:=intercd(s,d)
      fin
    sinon
      cercle00:=cdiam(o,a)
      point04:=centre(cercle00)
      long04:=rayon(cercle00)
      list01:=intercc(c,cercle00)    % remarque : ceci redéfinit le point a !
```

```
      pour point03 dans list01 faire
         long05:=dist(point03,a)
         si [point03 diff a] alors  % ceci ne se passe jamais ...
            droite00:=dro(point03,a)  % la construction suivante est donc sans objet ...
            b:=interdd(droite00,droite03)
            long06:=dist(b,a)
            si [b diff a] alors
               point01:=mil(b,a)
               droite01:=med(b,a)
               dir02:=dird(droite01)
               point00:=interdd(droite01,droite02)
               long02:=dist(point00,o)
               long01:=did(point00,droite03)
               list02:=cr2p(long01,b,a)
               pour s dans list02 faire
                  fin
               fin
             sinon
               s:=[]
               fin
            fin
          sinon                              % cas où a = point03 (toujours vérifié)
            long07:=dist(point04,point03)
            echec
            fin
         fin
      fin
   fin
verifier(tangentcc(s,c,a))
verifier(tangentcd(s,d,b))
fin
```

On obtient à l'interprétation les sorties graphiques des figures 5 et 6 suivantes :

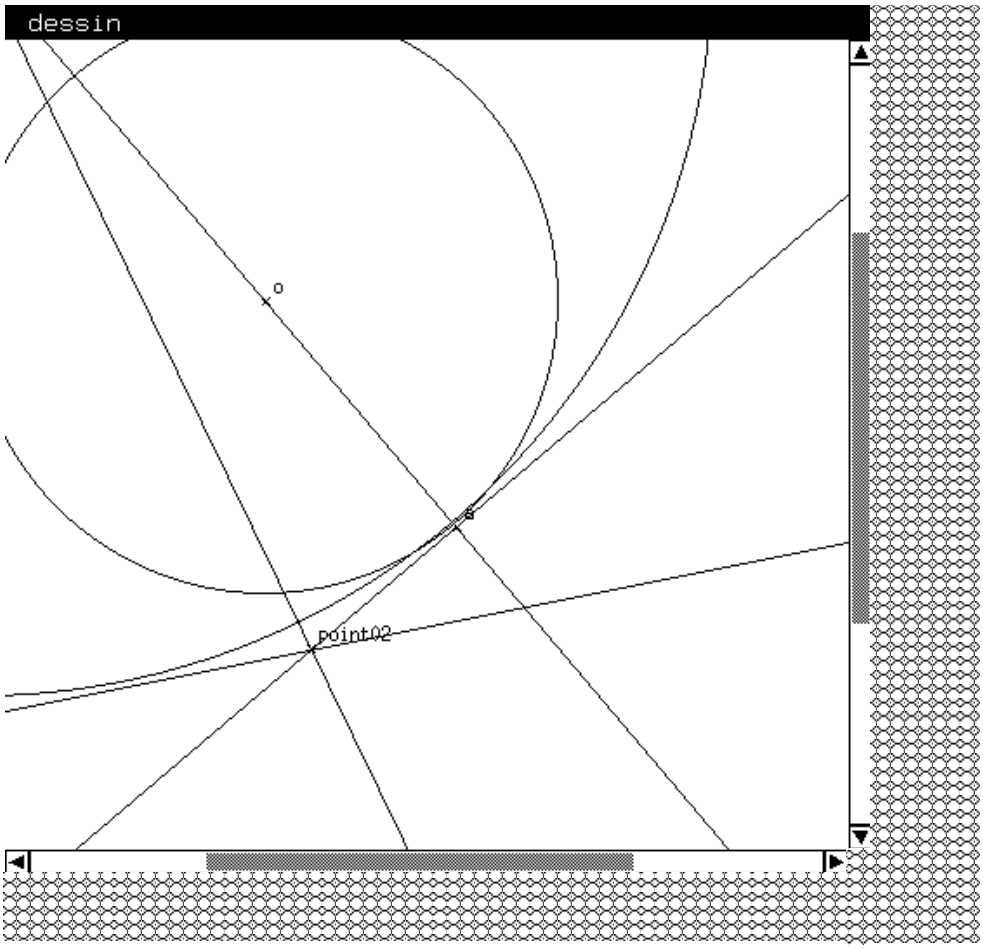

Figure 5 : copie d'écran 1.

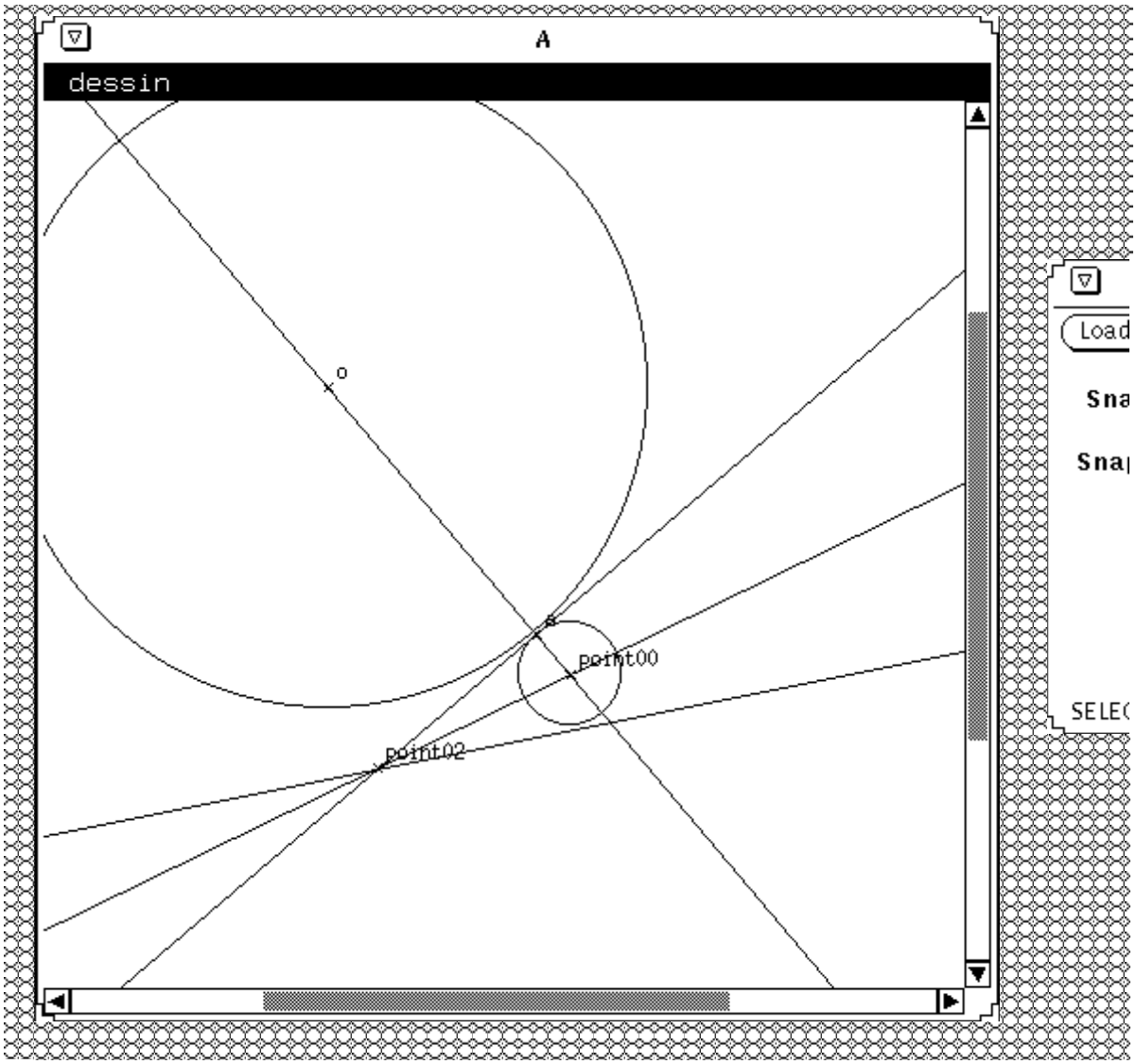

Figure 6 : copie d'écran 2.

Remarque : on voit dans ce programme se développer une branche conditionnelle sans objet, car la condition ne se produit jamais. Il nous paraît difficile d'empêcher ce genre de phénomène dans le cas général. En tout état de cause, les solutions numériques calculées sont néanmoins correctes puisque le bout de programme incriminé n'est jamais interprété.

3. Buthpll

Cet exercice est tiré de [Buthion 76]. Dans sa résolution par Geom3 seul un cas de figure est envisagé. Progé envisage les deux cas de figure mais le programme trouvé souffre, comme pour l'exercice précédent, de digressions fâcheuses.

Enoncé (Buthpll). Etant donnés deux droites **d1** et **d2** parallèles, un point **a** sur **d1**, un point **b** sur **d2** et un point **o**, construire une droite **d** coupant **d1** en **m** et **d2** en **n** de sorte que la somme des distances **am** + **bn** soit égale à la longueur **l** donnée (Cf. figure 10).

```
0 'dec:' point :: o donne.
1 'dec:' dir :: di donne.
2 'dec:' point :: a donne.
3 'dec:' point :: b donne.
9 'dec:' long :: l donne.
4 'dec:' droite :: d1 nomme dpdir(a,di).
5 'dec:' droite :: d2 nomme dpdir(b,di).
6 'dec:' droite :: d cherche.
7 'dec:' point :: m mentionne.
8 'dec:' point :: n mentionne.

0 'cont:' m est_sur d1.
1 'cont:' n est_sur d2.
2 'cont:' m est_sur d.
3 'cont:' n est_sur d.
4 'cont:' o est_sur d.
4 'cont:' di '=di=' di.
5 'cont:' dist(a,m) + dist(b,n) '=l=' l.
6 'cont:' d1 diff d2.
```

Programme de construction.

```
o:=(donne)
di:=(donne)
a:=(donne)
b:=(donne)
l:=(donne)
d1:=dpdir(a,di)
d2:=dpdir(b,di)
cercle00:=ccr(a,l)
list00:=intercd(cercle00,d1)
pour point00 dans list00 faire
   list01:=[[pll(point00,m,b,n)],[pll(point00,m,n,b)]]
   pour cas00 dans list01 faire
      si cas00 eg[pll(point00,m,b,n)] alors
         long03:=dist(point00,b)
         si [point00 diff b] alors
            point01:=mil(point00,b)
            droite00:=dro(b,point00)
            dir01:=dird(droite00)
            long04:=dist(point01,o)
            si [point01 diff o] alors
               d:=dro(point01,o)
               m:=interdd(d,d1)
               n:=interdd(d,d2)
               fin
             sinon
               echec
               fin
            fin
          sinon
            echec
            fin
         fin
      si cas00 eg[pll(point00,m,n,b)] alors
         long05:=dist(b,point00)
         si [b diff point00] alors
            droite05:=dro(b,point00)
            dir06:=dird(droite05)
```

```
            d:=dpdir(o,dir06)
            m:=interdd(d,d1)
            n:=interdd(d,d2)
            fin
          sinon
            cercle01:=cdiam(b,o)
            point04:=centre(cercle01)
            long06:=rayon(cercle01)
            long07:=dist(o,b)
            si [o diff b] alors
               droite07:=dro(o,b)
               dir08:=dird(droite07)
               echec
               fin
             sinon
               echec
               fin
            fin
         fin
      fin
   fin
verifier(m est_sur d1)
verifier(n est_sur d2)
verifier(m est_sur d)
verifier(n est_sur d)
verifier(o est_sur d)
verifier(di=di=di)
verifier(dist(a,m)+dist(b,n)=l=l)
verifier(d1 diff d2)
fin
```

Voici les copies d'écran correspondant à une exécution de ce programme :

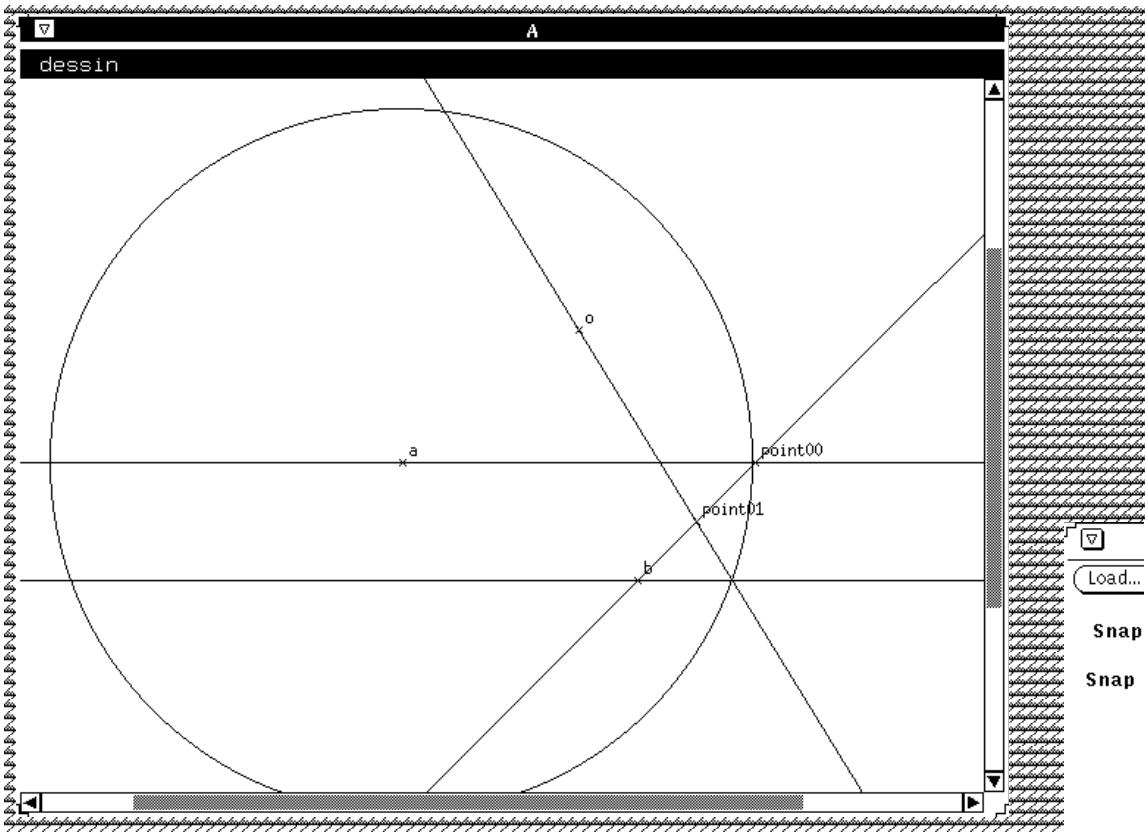

figure 7 : copie d'écran correspondant au premier cas de figure.

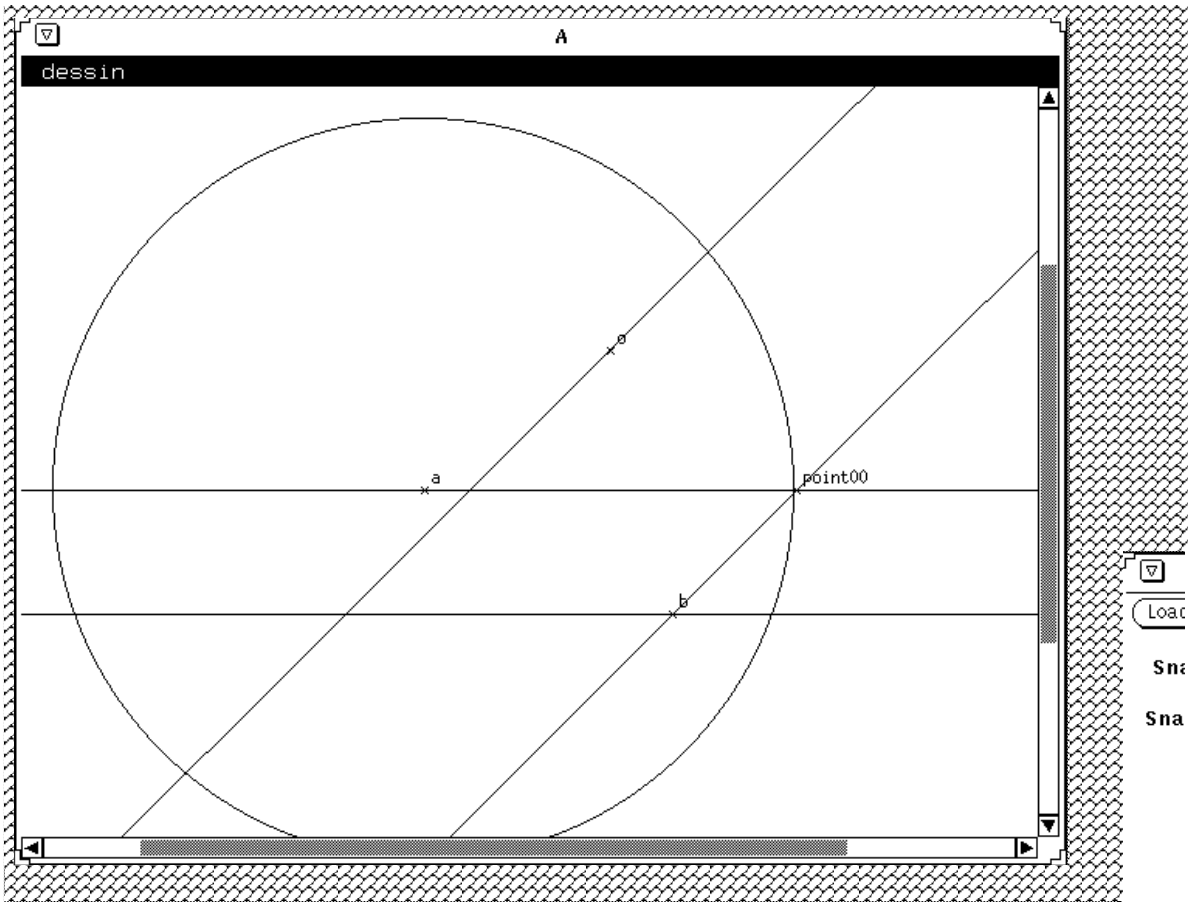

figure 8 : copie d'écran correspondant au deuxième cas de figure.

Annexe 2
Exemples de problèmes résolus par Progé

Après le titre (en gras souligné), l'énoncé de l'exercice est donné tel qu'il s'écrit en Progé (Prolog), suivi de l'énoncé en langage courant.
(After the name of the exercice, the statement is given at the left as it is written in Prolog within Progé, and in french natural language in the right.
The numbers used are here just for convenience when editing the statement.

- 'dec' means declaration of a geometric variable with three possible status :
  - donne means « given » (donné in french) by the user. It is like a free object in dynamic geometry)
  - cherche means « to be found » (cherché in french)
  - mentionne means auxilliary object just mentionned in order to simplify the statement
- the geometric types are point, droite (line), cercle (circle), longueur (length))
- 'cont' means constraint (contrainte in french). As a contrainsts we can have
  - est_sur which means « is_on » and noting incidence constraints
  - iso(a, b, c) means that triangle (a,b,c) is isocele
  - ortho is put for orthogonality constraint
  - '=p=', '=d=' ... are equalities constraints (the letter between the = signs note the type of the equality point, line, ...)
  - tangentxx is for tangency between circle(s) and/or line
  - centre is for center, rayon is for radius
  - nomme is not really a constraint, this is just a directive for naming a functional term
  - pr is for orthogonal projection, mil is for midpoint (milieu in french)

buth19

0 'dec:' point :: a donne.
1 'dec:' point :: b donne.
2 'dec:' point :: c donne.
3 'dec:' cercle :: s cherche.

Construire un cercle s passant par trois points a, b et c donnés.

0 'cont:' a est_sur s.
1 'cont:' b est_sur s.
2 'cont:' c est_sur s.

buth2

0 'dec:' point :: b donne.
1 'dec:' point :: c donne.
2 'dec:' point :: a cherche.

Construire un triangle abc, b et c étant donnés, isocèle et rectangle en b.

0 'cont:' iso(b,c,a).
1 'cont:' dro(b,c) ortho dro(b,a).

buth20

0 'dec:' point :: o donne.
1 'dec:' point :: a mentionne.
2 'dec:' droite :: d donne.
3 'dec:' cercle :: c cherche.

Construire un cercle c de centre donné et tangent à une droite d donnée.

0 'cont:' o '=p=' centre(c).
1 'cont:' tangentcd(c,d,a).

buth21

0 'dec:' point :: o donne.
1 'dec:' point :: a mentionne.
2 'dec:' cercle :: d donne.
3 'dec:' cercle :: c cherche.

Construire un cercle c de centre o donné et tangent à un cercle d donné.

0 'cont:' o '=p=' centre(c).
1 'cont:' tangentcc(c,d,a).

buth22

0 'dec:' point :: a donne.
1 'dec:' point :: b donne.
2 'dec:' cercle :: c donne.
3 'dec:' cercle :: s cherche.

Construire un cercle s, passant par un donné a et tangent à un cercle donné c en un point b donné.

0 'cont:' a est_sur s.
1 'cont:' tangentcc(s,c,b).

buth23

0 'dec:' point :: a donne.
1 'dec:' point :: b donne.
2 'dec:' droite :: d donne.
3 'dec:' cercle :: s cherche.

Construire un cercle s passant par un point a donné et tangent à une droite d en un point b donné.

0 'cont:' a est_sur s.
1 'cont:' tangentcd(s,d,b).

buth24

0 'dec:' point :: a donne.
1 'dec:' point :: b donne.
2 'dec:' cercle :: s cherche.
3 'dec:' point :: o nomme centre(s).
4 'dec:' droite :: d donne.

Construire un cercle s passant par deux points donnés, a et b, et dont le centre o est sur une droite donnée.

0 'cont:' a est_sur s.
1 'cont:' b est_sur s.
2 'cont:' o est_sur d.

buth25

0 'dec:' droite :: d1 donne.
1 'dec:' droite :: d2 donne.
2 'dec:' point :: a nomme interdd(d1,d2).
3 'dec:' cercle :: s cherche.
4 'dec:' point :: o nomme centre(s).
5 'dec:' long :: l donne.
6 'dec:' point :: b1 mentionne.
7 'dec:' point :: b2 mentionne.

Construire un cercle dont le centre o est à une distance donnée l d'un point a, intersection de deux droites d1 et d2 données et tangent à celles-ci.

0 'cont:' dist(o,a) '=l=' l.
1 'cont:' tangentcd(s,d1,b1).
2 'cont:' tangentcd(s,d2,b2).

buth26

0 'dec:' droite :: d1 donne.
1 'dec:' droite :: d2 donne.
2 'dec:' long :: r donne.

Construire un cercle s de rayon r donné et tangent à deux droites d1 et d2 données.

3 'dec:' cercle :: s cherche.
4 'dec:' point :: b1 mentionne.
5 'dec:' point :: b2 mentionne.

0 'cont:' r '=l=' rayon(s).
1 'cont:' tangentcd(s,d1,b1).
2 'cont:' tangentcd(s,d2,b2).

buth27

0 'dec:' cercle :: c1 donne. Construire un cercle s de rayon r donné et tangent à deux
1 'dec:' cercle :: c2 donne. cercles c1 et c2 donnés.
2 'dec:' long :: r donne.
3 'dec:' cercle :: s cherche.
4 'dec:' point :: b1 mentionne.
5 'dec:' point :: b2 mentionne.

0 'cont:' r '=l=' rayon(s).
1 'cont:' tangentcc(s,c1,b1).
2 'cont:' tangentcc(s,c2,b2).

buth29

0 'dec:' droite :: d donne. Construire un cercle s de centre o tel que o est sur
1 'dec:' droite :: t donne. une droite donnée d et s est tangent à une droite t
2 'dec:' point :: a donne. donnée en un point a donné.
3 'dec:' cercle :: s cherche.
4 'dec:' point :: o nomme centre(s).

0 'cont:' o est_sur d.
1 'cont:' tangentcd(s,t,a).

buth32

0 'dec:' long :: r mentionne. Construire un cercle s tangent à trois cercles c1, c2
1 'dec:' cercle :: c1 donne. et c3 donnés et de même rayon r.
2 'dec:' cercle :: c2 donne.
3 'dec:' cercle :: c3 donne.
4 'dec:' cercle :: s cherche.
5 'dec:' point :: a1 mentionne.
6 'dec:' point :: a2 mentionne.
7 'dec:' point :: a3 mentionne.

0 'cont:' r '=l=' rayon(c1).
1 'cont:' r '=l=' rayon(c2).
2 'cont:' r '=l=' rayon(c3).
3 'cont:' tangentcc(s,c1,a1).
4 'cont:' tangentcc(s,c2,a2).
5 'cont:' tangentcc(s,c3,a3).

buth32b

0 'dec:' long :: r donne. idem, mais formulé différemment.
8 'dec:' point :: o1 donne.
9 'dec:' point :: o2 donne.
10 'dec:' point :: o3 donne.
1 'dec:' cercle :: c1 nomme ccr(o1,r).
2 'dec:' cercle :: c2 nomme ccr(o2,r).
3 'dec:' cercle :: c3 nomme ccr(o3,r).
4 'dec:' cercle :: s cherche.

5 'dec:' point :: a1 mentionne.
6 'dec:' point :: a2 mentionne.
7 'dec:' point :: a3 mentionne.

3 'cont:' tangentcc(s,c1,a1).
4 'cont:' tangentcc(s,c2,a2).
5 'cont:' tangentcc(s,c3,a3).

buthpll

0 'dec:' point :: o donne.
1 'dec:' dir :: di donne.
2 'dec:' point :: a donne.
3 'dec:' point :: b donne.
9 'dec:' long :: l donne.
4 'dec:' droite :: d1 nomme dpdir(a,di).
5 'dec:' droite :: d2 nomme dpdir(b,di).
6 'dec:' droite :: d cherche.
7 'dec:' point :: m mentionne.
8 'dec:' point :: n mentionne.

Etant donnés un point o, deux droites d1 et d2 parallèles et deux points a, sur d1, et b, sur d2, construire une droite d passant par o et coupant d1 en m et d2 en n tels que la somme des distance am + bn soit égale à une longueur donnée l.

0 'cont:' m est_sur d1.
1 'cont:' n est_sur d2.
2 'cont:' m est_sur d.
3 'cont:' n est_sur d.
4 'cont:' o est_sur d.
4 'cont:' di '=di=' di.
5 'cont:' dist(a,m) + dist(b,n) '=l=' l.
6 'cont:' d1 diff d2.

exo0

1 'dec:' point :: a donne.
2 'dec:' point :: b donne.
3 'dec:' point :: c cherche.
4 'dec:' long :: k donne.
5 'dec:' long :: l donne.
0 'cont:' dist(a,c) '=l=' k.
1 'cont:' dist(b,c) '=l=' l.

Construire un triangle abc dont les sommets a et b sont donnés et les longueurs des côtés aussi.

exo1

1 'dec:' point :: c cherche.
2 'dec:' point :: b donne.
3 'dec:' point :: a donne.
0 'cont:' dro(a,b) ortho dro(a,c).
1 'cont:' iso(a,b,c).

Construire un triangle abc dont les sommets a et b sont donnés et rectangle et isocèle en a.

exo10

0 'dec:' point :: a donne.
1 'dec:' point :: aa donne.
2 'dec:' point :: b cherche.
3 'dec:' point :: c cherche.
4 'dec:' long :: k donne.
5 'dec:' long :: l donne.

Construire un triangle abc dont le sommet a et le milieu aa de (b, c) sont donnés et dont on connaît les longueur des côtés ab et bc.

0 'cont:' aa '=p=' mil(b,c).
1 'cont:' dist(a,b) '=l=' k.
2 'cont:' dist(b,c) '=l=' l.

exo11

0 'dec:' point :: a donne.
1 'dec:' point :: ha donne.
2 'dec:' point :: ap mentionne.
3 'dec:' point :: c cherche.
4 'dec:' point :: b cherche.
5 'dec:' long :: l donne.
6 'dec:' long :: k donne.

Construire un triangle abc dont sont donnés le sommet a, le pied ha de la hauteur issue de a, la longueur aap de la médiane issue de a t la distance de b à c.

0 'cont:' ha '=p=' prj(a,dro(b,c)).
1 'cont:' ap '=p=' mil(b,c).
2 'cont:'dist(b,c) '=l=' l.
3 'cont:' dist(a,ap) '=l=' k.

exo12

0 'dec:' cercle :: c donne.
1 'dec:' droite :: d donne.
2 'dec:' point :: a donne.
3 'dec:' point :: b mentionne.
4 'dec:' cercle :: s cherche.

Construire un cercle s tangent à un cercle c donné et à une droite d donnée en un point a de d donné.

0 'cont:' tangentcc(s,c,b).
1 'cont:' tangentcd(s,d,a).

exo12b

0 'dec:' cercle :: c donne.
1 'dec:' point :: a1 donne.
2 'dec:' point :: a2 donne.
5 'dec:' droite :: d nomme dro(a1,a2).
3 'dec:' point :: b mentionne.
4 'dec:' cercle :: s cherche.

idem, mais énoncé de manière constructive.

0 'cont:' tangentcc(s,c,b).
1 'cont:' tangentcd(s,d,a1).

exo13

0 'dec:' cercle :: c donne.
1 'dec:' point :: a donne.
2 'dec:' droite :: d cherche.
3 'dec:' point :: b mentionne.

Construire une droite d passant par un point a donné et tangent à un cercle c donné.

0 'cont:' a est_sur d.
1 'cont:' tangentcd(c,d,b).

exo2

1 'dec:' point :: a donne.
2 'dec:' point :: b donne.
3 'dec:' point :: c cherche.
5 'dec:' long :: l donne.
0 'cont:' iso(a,b,c).
1 'cont:' dist(b,c) '=l=' l.

Construire un triangle isocèle en a et dont on donne les sommets a et b et la distance de b à c.

exo3

1 'dec:' point :: a donne.
2 'dec:' point :: b donne.
3 'dec:' point :: c cherche.
4 'dec:' long :: k donne.

Construire un triangle abc dont on donne les sommets a et b, le côté ac et la longueur de la hauteur issue de c.

5 'dec:' long :: l donne.
0 'cont:' dist(a,c) '=l=' k.
1 'cont:' did(c,dro(a,b)) '=l=' l.

exo4

1 'dec:' point :: c donne. Construire un triangle abc isocèle et rectangle en a et dont
2 'dec:' point :: b donne. on connaît les sommets b et c.
3 'dec:' point :: a cherche.
0 'cont:' dro(a,b) ortho dro(a,c).
1 'cont:' iso(a,b,c).

exo5

0 'dec:' point :: a donne. Cf. l'énoncé du chapitre 3 !
1 'dec:' point :: b donne.
2 'dec:' cercle :: c donne.
3 'dec:' droite :: d cherche.
4 'dec:' point :: e mentionne.
5 'dec:' point :: f mentionne.

0 'cont:' e est_sur c.
1 'cont:' f est_sur c.
2 'cont:' e est_sur d.
3 'cont:' f est_sur d.
4 'cont:'a est_sur d.
5 'cont:' dist(e,b) '=l=' dist(f,b).

exo6

1 'dec:' point :: a donne. Construire un triangle abc de sommets a et b donnés et dont
2 'dec:' point :: b donne. on connaît la longueur des hauteurs issues de a et de b.
3 'dec:' point :: c cherche.
4 'dec:' long :: k donne.
5 'dec:' long :: l donne.
0 'cont:' did(a, dro(b,c)) '=l=' k.
1 'cont:' did(b, dro(a,c)) '=l=' l.

exo7

0 'dec:' droite :: d donne. Construire un cercle cc de rayon r donné tangent à une droite
1 'dec:' cercle :: c donne. d donnée et à un cercle c donné également.
2 'dec:' long :: r donne.
3 'dec:' point :: a mentionne.
4 'dec:' point :: b mentionne.
5 'dec:' cercle :: cc cherche.
0 'cont:' r '=l=' rayon(cc).
1 'cont:' tangentcc(cc,c,a).
2 'cont:' tangentcd(cc,d,b).

exo8b

0 'dec:' point :: a donne. Construire un cercle s tangent à un cercle c donné
0 'dec:' point :: o donne. en un point a donné et à une droite d donnée.
1 'dec:' droite :: d donne.
2 'dec:' cercle :: c nomme ccp(o,a).
3 'dec:' cercle :: s cherche.
4 'dec:' point :: b mentionne.

1 'cont:' tangentcc(s,c,a).
2 'cont:' tangentcd(s,d,b).

exo9

0 'dec:' point :: a donne. Construire un cercle s inscrit ou exinscrit à abc.
1 'dec:' point :: b donne.
2 'dec:' point :: c donne.
3 'dec:' cercle :: s cherche.
4 'dec:' point :: m mentionne.
4 'dec:' point :: n mentionne.
4 'dec:' point :: l mentionne.

0 'cont:' tangentcd(s,dro(a,b),m).
1 'cont:' tangentcd(s,dro(a,c),n).
2 'cont:' tangentcd(s,dro(c,b),l).

dwell

0 'dec:' long :: l1 donne. Exercice d'articulations genre CAO ...
1 'dec:' long :: l2 donne.
2 'dec:' long :: l3 donne.
3 'dec:' long :: l4 donne.
4 'dec:' long :: l5 donne.
5 'dec:' long :: l6 donne.
6 'dec:' long :: l7 donne.
7 'dec:' point :: a donne.
8 'dec:' point :: b donne.
9 'dec:' point :: c donne.
10 'dec:' point :: z donne.
11 'dec:' point :: m cherche.
12 'dec:' point :: n cherche.
13 'dec:' point :: p cherche.
14 'dec:' point :: q cherche.

0 'cont:' dist(a,m) '=l=' l1.
1 'cont:' dist(b,n) '=l=' l2.
2 'cont:' m est_sur dro(a,z).
3 'cont:' dist(c,q) '=l=' l3.
4 'cont:' dist(m,n) '=l=' l7.
5 'cont:' dist(m,p) '=l=' l5.
6 'cont:' dist(n,p) '=l=' l6.
7 'cont:' dist(p,q) '=l=' l4.

Annexe 3
Règles actuellement utilisées par Progé

Le listing suivant est le programme Prolog codant les règles utilisées par Progé : ces règles sont données telles quelles. Certaines formulations sont, sans doute, maladroites, le choix des règles pourra parfois paraître surprenant à certains géométres...

La syntaxe d'une règle normale est la suivante :

```
<nom> # si <liste de faits à chercher dans le raisonnement>
          et <liste de vérifications à faire sur la figure>
        alors <liste de nouveaux faits>
```

Les nouveaux fais obtenus servent à mettre à jour la figure et le raisonnement.

Pour une règle disjonctive, on a la syntaxe :

```
<nom> # si <liste de faits à chercher dans le raisonnement>
          et <liste de vérifications à faire sur la figure>
        alors
         soit <liste de cond. supplémentaires à vérifier> et <nouveaux faits>
        ou
         soit <liste de cond. supplémentaires à vérifier> et <nouveaux faits>
        ou
                  ...
```

les listes de conditions supplémentaires sont éventuellement vides (règles disjonctives non exclusives). Dans ce cas, elles doivent être toutes vides dans cette règle (ceci n'est pas vérifié par Progé).

```
/*---------------------------------*
*       REGLES.PRO                *
*  Regles pour Proge              *
*---------------------------------*/


1 # si  [ dist(A,B) '=l=' K ]
      et
       [connu A, connu K, pas_connu B]
   alors
      [ B est_sur ccr(A,K) : 1].

2 # si [ did(A,D) '=l=' H]
     et
      [connu D, connu H, pas_connu A]
   alors
     [ A est_sur dpd(D,H) : 1].
```

```
3 # si [ did(A, dro(B,C)) '=l=' L]
      et
      [connu A, connu B, connu L, differents [B, prj(A, dro(B,C))],                    pas_connu C]
   alors
      [
       H nomme prj(A, dro(B,C)),
       H est_sur cdiam(A,B),
       H est_sur ccr(A,L) : 1
      ].

33 # si [ H '=p=' prj(A, dro(B,C))]
     et
           [connu A, connu H,  pas_connu B, pas_connu C]
    alors
           [
             D nomme dorth(dro(A,H), H),
             D '=d=' dro(B,C)
            ].

 5 # si [ D ortho Dp ]
     et
      [ connu D,  pas_connu Dp]
   alors
      [
       Di nomme diro(D),
       Di '=di=' dird(Dp): 1
      ].

4 # si [ dro(A,B) ortho dro(A,C)]
     et
      [connu B, connu C, pas_connu A]
   alors
      [ A est_sur cdiam(B,C) : 1].



6 # si [ dist(A,M) '=l=' dist(B,M) ]
     et
      [ differents [A,B], connu med(A,B), pas_connu M]
   alors
      [ M est_sur med(A,B) : 1
      ].

7 # si  [did(A,D1) '=l=' did(A,D2)]
     et
           [differents [D1,D2], connu D1, connu D2, pas_connu A]
    alors
           soit [dird(D1) diff dird(D2)] et [ A est_sur bis(D1,D2) : 1]
     ou
     soit [dird(D1) eg dird(D2), D1 diff D2] et [A est_sur dmd(D1,D2):1]
     ou
     soit [D1 eg D2] et [].
```

```
8 # si  [ I '=p=' mil(A,B)]
     et
          [ connu I, connu A, pas_connu B]
   alors
          [ B '=p='symp(I,A) : 2 ].

9 # si     [ I '=p=' mil(A,B), dist(A,B) '=l=' K]
     et
          [ connu I, connu K, pas_connu A, pas_connu B]
    alors
          [ A est_sur ccr(I,K/2) : 1, B est_sur ccr(I,K/2) : 1].

20 # si [ tangentcd(ccr(O,R),D,A)]
    et
          [connu R, connu D, connu A, pas_connu O]
   alors
          [
           O est_sur dorth(D,A) : 1,
           O est_sur dpd(D,R)
          ].

21 # si   [ tangentcd(ccr(O,R),D,A)]
      et
          [connu R, connu D, pas_connu O, pas_connu A]
    alors
          [ O est_sur dpd(D,R) ].

22 # si [ tangentcd(ccr(O,R),D,A)]
    et
          [connu D, connu A, pas_connu O, pas_connu R]
   alors
          [
          O est_sur dorth(D,A)
          ].


23 # si [  tangentcc(ccr(O,R), ccr(Op,Rp),A)]
     et
          [ connu Op, connu Rp, connu R, pas_connu O]
    alors
          [ O est_sur ccr(Op, R+Rp) : 1 ].


24 # si [  tangentcc(ccr(O,R), ccr(Op,Rp),A)]
          et
          []
    alors
          [dist(O,Op) '=l=' R + Rp].


100 # si [dist(A,B) '=l=' dist(A,C)]
       et
        [differents [A,B,C] ]
     alors
        [iso(A,B,C)].
```

```
102 # si [ iso(A,B,C) ]
      et
      []
   alors
     [dist(A,B) '=l=' dist(A,C)].

105 # si [ iso(A,B,C) ]
      et
       [pas_connu B]
     alors
      [
       I nomme mil(B,C),
             dro(A,I) '=d=' med(B,C),
        dro(A,I) ortho dro(B,C)
      ].

110 # si [M est_sur ccr(O,R)]
      et []
     alors
      [ dist(O,M) '=l=' R].

120 # si [tangentcc(C,D,A)]
      et
           []
     alors
           [ A est_sur C, A est_sur D].

121 # si [tangentcd(C,D,A)]
      et
           []
     alors
           [ A est_sur C, A est_sur D].

122 # si [tangentdc(C,D,A)]
      et
           []
     alors
           [ A est_sur C, A est_sur D].

123 # si [tangentcd(ccr(O,R),D,A) ]
           et
            []
     alors
            [ R '=l=' did(O,D)].

 124 # si [tangentcc(ccr(O1,R1),ccr(O2,R2),A)]
      et
           []
     alors
           [
            D1 nomme dro(O1,O2),
            A est_sur D1,
            D nomme dorth(D1,A),
            tangentcd(ccr(O1,R1),D,A),
            tangentcd(ccr(O2,R2),D,A),
            tinhibe(22)
           ].

125 # si [tangentcc(ccr(O1,R1),ccr(O2,R2),A), B est_sur ccr(O1,R1)]
           et
```

```
          [ connu B, connu O2, connu R2, pas_connu A, connu dro(B,O1) ]
    alors
          [ BB nomme intercd(ccr(O2,R2),dpp(dro(B,O1),O2)),
            A est_sur dro(B,BB),
              A diff BB
          ].

200 # si [dist(A,B) '=l=' dist(C,D), dird(dro(A,B)) '=di=' dird(dro(C,D))]
          et
           [differents [A,B,C,D]]
    alors
          soit [] et [pll(A, B, C, D)]
     ou
          soit [] et [pll(A, B, D, C)].

202 # si [pll(A,B,C,D)]
          et
           []
    alors
          [mil(A,C) '=p=' mil(B,D)].

203 # si [pll(A,B,C,D)]
          et
           []
    alors
           [dird(dro(A,B)) '=di=' dird(dro(C,D)),
            dird(dro(A,D)) '=di=' dird(dro(B,C))].



300 # si [dist(A, M) + dist(B, N) '=l=' L]
          et
            [connu dro(A, M)]
          alors
          [
                    X nomme interdc(dro(A, M), ccr(A, L)),
                    dist(X,M) '=l=' dist(B,N),
                    thinibe(300)
          ].
```