Animation de programmes fonctionnels

Yannis Chicha
DEA informatique
Université de Nice - Sophia Antipolis

Encadreur: Stephen Watt - UNSA/INRIA/I3S

Rapporteur: Erick Gallesio
1995-1996

Résumé

L'*animation d'algorithmes* est une discipline encore peu connue en informatique. Un système d'animation propose des outils permettant de visualiser le déroulement d'un programme et d'agir sur certains aspects durant son exécution. Les systèmes d'animation existant se basent sur des langages procéduraux ou orienté-objet. Aucun, jusqu'à présent, ne permet d'animer des *programmes fonctionnels* car ces différents systèmes ne répondent pas aux besoins spécifiques des programmeurs utilisant des langages fonctionnels. Ainsi, un système d'animation adapté devrait pouvoir montrer l'évolution des structures de données manipulées par le programme. L'utilisateur doit avoir la possibilité de visualiser la fréquence et le temps d'utilisation des données et la façon dont elles sont touchées, manipulées, consultées. Un système d'animation de programmes fonctionnels doit faire comprendre à son utilisateur de quelle manière les données et les fonctions sont en corrélation dans un programme sans affectation et utilisant des évaluations strictes ou paresseuses.

Dans ce rapport, nous tenterons de définir la nature de l'animation d'algorithmes. Nous expliquerons l'intérêt de cet outil et nous montrerons quelques systèmes d'animation. Puis nous présenterons le travail que nous avons réalisé autour de la programmation fonctionnelle, les défis qui nous sont lancés ainsi que les aspects importants que nous avons traités (objets composites, constructeurs de modules, évaluation paresseuse). Enfin, nous exposerons notre système d'animation au travers de quelques exemples caractéristiques, notamment nous présenterons une animation sur les bases de Gröbner.

Table des matières

Introduction

L'animation de programmes est un sujet sur lequel peu de personnes ont travaillé jusqu'à présent. Il est pourtant intéressant de constater que de gros besoins apparaissent dans le domaine de l'aide à la compréhension de programmes. En effet, il est tout aussi important de visualiser l'évolution d'un programme que d'obtenir son résultat. Nous pensons que l'animation d'algorithmes pourrait être la solution ou tout au moins une solution.

Il est important dès maintenant de faire la distinction entre animation de programmes et animation d'algorithmes. Dans le premier cas (animation de programmes), le but est de traiter un programme déjà écrit. Dans le deuxième cas (animation d'algorithmes), une implémentation de l'algorithme n'est pas déjà écrite, ce qui nous autorise à écrire ou ré-écrire une partie importante de l'implémentation.

Certains travaux, sur lesquels nous reviendrons, ont déjà été menés dans ce sens. Cependant, il existe encore de gros manques, notamment en programmation fonctionnelle. C'est pourquoi nous avons choisi de traiter ce sujet. Ce rapport présente les résultats que nous avons obtenus.

Dans un premier temps, nous allons tenter de définir l'animation d'algorithmes, nous présenterons certaines applications possibles ainsi que des systèmes existants (Zeus, Aladdin, Agat) que nous avons étudiés afin de découvrir les avantages et les inconvénients de chacun. Une deuxième partie nous permettra de présenter notre travail. Une petite récapitulation des manques des systèmes actuels introduira notre étude, puis nous définirons les aspects que nous avons choisis de traiter, notamment nous verrons sur quels points de la programmation fonctionnelle nous avons choisi de travailler.

Afin de montrer que nos résultats sont applicables, nous avons décidé de travailler sur quelques exemples. Nous avons sélectionné le langage $A^\sharp$ pour les programmer. Un chapitre présentera ce langage et ses particularités, nous y expliquerons également pourquoi $A^\sharp$ s'avère plutôt intéressant pour la réalisation de ce travail. Puis, nous exposerons en détail les exemples sur lesquels nous avons travaillés, à savoir l'arithmétique sur des polynômes (objets composites), sur les développements infinis (évaluation paresseuse) ainsi qu'une animation d'un algorithme de résolution des bases de Gröbner (opérations de haut niveau et modification minimale de code).

Nous terminerons par une projection sur l'avenir, ce qu'il reste à faire et dans quel cadre nos résultats seront utilisés.

Chapitre 1

L'animation d'algorithmes

1.1 Définition

Pour définir l'animation d'algorithmes de façon succinte, nous pouvons dire qu'il s'agit d'un moyen d'illustrer un programme afin de suivre son déroulement et de mieux en comprendre les mécanismes. En fait, il est aussi important d'obtenir le résultat d'un programme que de voir comment ce programme évolue pour calculer ce résultat.

L'animation d'algorithmes représente donc un outil puissant qui permettrait de comprendre plus rapidement et plus simplement le comportement dynamique d'un programme. Il existe, comme nous le verrons par la suite, de nombreuses applications à cet outil. Ainsi, l'animation de programmes se destine à être un support de cours ou un moyen d'analyse menant à l'amélioration d'algorithmes existants.

Basée sur des éléments visuels, graphiques, l'animation d'algorithmes permet également une forte interaction avec l'utilisateur pour que celui-ci puisse contrôler le déroulement du programme et l'évolution des données. Ces évolutions sont illustrées par des images et des animations voire des sonorisations, ceci dans le but de présenter des statistiques, des résultats et des variations de manière attrayante et compréhensible rapidement. De plus, un avantage des systèmes d'animation de programmes est la possibilité de visualiser un même aspect du problème selon plusieurs points de vue différents, la compréhension n'en est que meilleure.

Un vieil adage dit qu' "un dessin vaut mieux que mille mots", l'animation d'algorithmes nous fournit l'application concrète de ce dicton. Cet outil irait même plus loin, car il fournirait non seulement des dessins, mais également des animations et des bruitages : nous entrons dans l'ère du multimédia.

1.2 Applications d'un système d'animation

Nous venons de voir ce qu'un outil d'animation d'algorithmes permet de faire. Nous allons maintenant voir un peu plus en détail quelles en sont les applications.

1.2.1 Un intérêt scientifique

L'animation d'algorithmes ou de programmes s'avère particulièrement utile dans le domaine scientifique. Ainsi, la recherche bénéficie, avec les systèmes d'animation, de puissants outils de présentation de résultats et de recherche de meilleures solutions.

En effet, lors d'un exposé, il est possible de montrer les réalisations effectuées de façon plus explicite que d'exécuter le programme et de voir le résultat final. Il est intéressant de pouvoir visualiser l'évolution complète d'un algorithme avec plus ou moins de détails suivant le contenu de l'exposé par exemple. Dans ce cadre, cela peut représenter un support utile pour l'échange d'idées.

D'autre part, dans le cadre de la recherche d'un nouvel algorithme plus rapide, l'animation permet d'étudier l'existant afin d'y déceler "à l'oeil nu" les endroits où les complexités en temps ou en espace peuvent être améliorée. De même, l'animation d'algorithmes peut permettre également de trouver de nouvelles heuristiques pour résoudre un problème. L'intuition est très importante dans le domaine de la recherche. On se souvient notamment que Kepler a trouvé sa troisième loi sur le parcours des planètes autour du soleil en observant des données. Un des intérêts de l'animation d'algorithmes, c'est qu'elle offre un support permettant d'exacerber les intuitions car cet outil donne la possibilité de visualiser, de comprendre et de tester.

1.2.2 Un intérêt pédagogique

L'apprentissage est le deuxième grand domaine d'application de l'animation de programmes. L'enseignement et l'apprentissage des algorithmes revêtissent souvent deux formes : soit l'étude se base sur une présentation et une explication formelle dans un ouvrage, soit elle se fait grâce à une implémentation.

Ces deux solutions présentent des avantages et des inconvénients. L'étude formelle est exhaustive et précise. Elle permet plusieurs niveaux d'analyse (de la vulgarisation à la compréhension détaillée). Cependant, elle est bien souvent peu attrayante et même si l'article ou le livre exposant l'algorithme comporte des schémas, la frustration de ne pas assister à une expérimentation de cet algorithme est grande.

Face à cela, la compréhension par l'étude d'une implémentation pourrait nous être secourable. En effet, il est possible de tester et de modifier le programme afin de mettre en valeur certains aspects de l'algorithme. Cependant, plusieurs inconvénients se présentent immédiatement. Les implémentations ne sont pas toujours bien écrites et donc bien peu compréhensibles pour quelqu'un n'ayant pas écrit lui-même ces implémentations. Même si le code est clair, le temps d'étude d'un algorithme complexe peut se révéler assez long. Enfin, un problème subsiste : il est possible de s'attarder sur des détails d'implémentation et ne pas voir les points importants de l'algorithme.

L'animation de programmes tente d'apporter une solution à ces problèmes. En effet, les systèmes d'animation peuvent être utilisés comme support de cours pour l'enseignement d'un algorithme. Grâce aux différents niveaux de détail, l'enseignant peut contrôler la discrimination des éléments de l'algorithme à apprendre. De plus, un système d'animation offre un moyen attrayant de découvrir et d'appréhender un algorithme. C'est en outre une manière efficace de l'étudier, car ces systèmes présentent de manière graphique et simple le comportement dynamique du programme animé.

Bien entendu, rien n'empêche la consultation complémentaire d'un ouvrage sur le sujet, mais l'animation d'un programme doit pouvoir suffire à l'étude complète de l'algorithme.

Un exemple significatif du but pédagogique de l'animation d'algorithmes est la présentation du déroulement de trois tris effectués en parallèle (exemple de programmation en Java [13]). On peut visualiser rapidement quel tri est le plus rapide et la manière dont évoluent les éléments à trier.

1.3 Relations avec le débogage

Pourquoi parler de débogage alors que nous sommes dans un exposé sur l'animation d'algorithmes ?

La raison est simple : les buts de l'animation d'algorithmes se révèlent être très proches de ceux du débogage de programmes. Ces deux outils montrent, en effet, la façon dont fonctionne un programme. Grâce à ces deux systèmes, il est possible de suivre, de comprendre et d'analyser l'évolution dynamique d'un programme.

Bien souvent, on fait l'erreur de dire que l'animation n'est rien de plus que du débogage et du profiling. Cependant, l'animation de programmes présente des particularités qui la différencient de ces deux domaines. Le but principal de l'animation d'algorithmes est de faire comprendre et de permettre l'analyse d'un algorithme. Le débogage a principalement pour but de trouver les erreurs d'une implémentation. Grâce au débogage, il est possible d'aller beaucoup plus loin dans le détail d'une exécution, mais l'animation permet de plus nombreux niveaux d'abstraction.

L'analyse d'un programme passe bien évidemment par la visualisation et la consultation de statistiques ce que réalise parfaitement un outil de profiling. Cependant, l'animation permet une analyse plus fine et plus visuelle : différentes informations peuvent être consultées de plusieurs manières et suivant différents points de vue. Un utilisateur désirant améliorer un algorithme n'aura peut être pas envie de voir des statistiques lui donnant le temps passé dans une fonction, il aura envie de comprendre à partir de quel moment un algorithme n'est plus optimal et quelles sont les structures de données mises en cause, ainsi que le contexte d'exécution.

Le débogage et l'animation d'algorithmes sont des outils complémentaires. Chacun travaille à un niveau différent, mais rien n'empêche une forte collaboration entre ces deux systèmes.

1.4 Quelques systèmes existants

TANGO et BALSA sont les systèmes "pionniers" de l'animation d'algorithmes. La plupart des articles dans ce domaine font référence à ces systèmes comme les premiers essais "aboutis". Pour plus de renseignements, vous pouvez consulter [18] et [3].

Nous avons souhaité étudier des systèmes plus récents et pour deux d'entre eux moins connus. Nous allons maintenant présenter ces trois systèmes d'animation d'algorithmes. Nous verrons quels sont leurs avantages et leurs inconvénients et également en quoi certains aspects de ces systèmes pourraient nous être utiles.

1.4.1 Zeus

Zeus est un système d'animation d'algorithmes écrit en Modula-3. La version que nous avons étudiée date de 1992. C'est un produit développé par M. Brown pour Digital Equipment Corporation en 1988-1989. Pour une présentation complète vous pouvez consulter [4].

C'est un des systèmes d'animation d'algorithmes les plus connus. Il consitue une référence en la matière. Par certains aspects, cette réputation est fondée, mais nous verrons que certaines facettes sont moins intéressantes. Zeus représente le passage obligé lorsque l'on commence à étudier les systèmes d'animation d'algorithmes. En effet, grâce à Zeus, on peut comprendre assez rapidement et assez intuitivement ce que constitue un système d'animation. La présentation graphique de Zeus est exemplaire voire même excessive.

Zeus est, comme nous l'avons dit plus haut, un système basé sur le langage Modula-3. C'est pourquoi il bénéficie de la programmation orientée objets, du typage fort et des processus "légers" pour le parallélisme. Zeus permet non seulement de créer des animations d'algorithmes mais également des éditeurs multi-vues (voir [4]).

Le système se base sur l'approche de BALSA. Il s'agit, brièvement, de créer des *annotations* dans un algorithme afin de choisir les opérations fondamentales à visualiser. Ces annotations sont appelées *interesting events* (évènements intéressants). Chaque vue de l'algorithme est en liaison avec ces annotations. Lorsqu'un évènement se produit, les différentes vues sont mises à jour. L'intérêt d'un système d'animation d'algorithmes est qu'il permet à un utilisateur non seulement de visualiser des opérations mais également d'*interagir* avec le programme. C'est ce que permet de faire Zeus par un panneau de contrôle.

Zeus utilise une technique classique basée sur des "threads" afin d'obtenir un parallélisme des opérations. Un serveur permet d'établir des communications entre l'algorithme et les vues. Ainsi, les évènements se produisant durant l'exécution de l'algorithme sont transmis aux différentes vues et les évènements se produisant dans une vue sont transmis à l'algorithme. De cette manière, l'utilisateur pourra non seulement visualiser une animation d'un programme mais également agir avec lui pendant son déroulement afin de le contrôler son exécution ainsi que de réaliser certaines petites expérimentations comme modifier le contenu d'une variable en cours d'exécution.

L'inconvénient manifeste de Zeus devient évident lorsque l'on souhaite réaliser une animation d'un programme. La puissance de ce système contribue à sa "lourdeur". En effet, les multiples possibilités offertes par Zeus se doivent d'être accédées et pour se faire, Zeus propose un mini-langage de spécification et utilise un préprocesseur (appelé *Zume*) permettant de générer les différentes définitions nécessaires à l'animation. La présence d'un langage supplémentaire constitue un poids supplémentaire pour un système d'animation, de plus, plus le système est évolué plus le langage associé doit l'être lui aussi. C'est pourquoi, Zeus est très intéressant, mais également très lourd en mise en oeuvre.

Malgré cet inconvénient majeur, Zeus reste un système intéressant et nous nous sommes inspirés de son idée de client-serveur pour réaliser notre système.

Il est à noter que Zeus est un des seuls systèmes utilisant également le son comme moyen d'animation (voir l'article [5] de M. Brown à ce sujet).

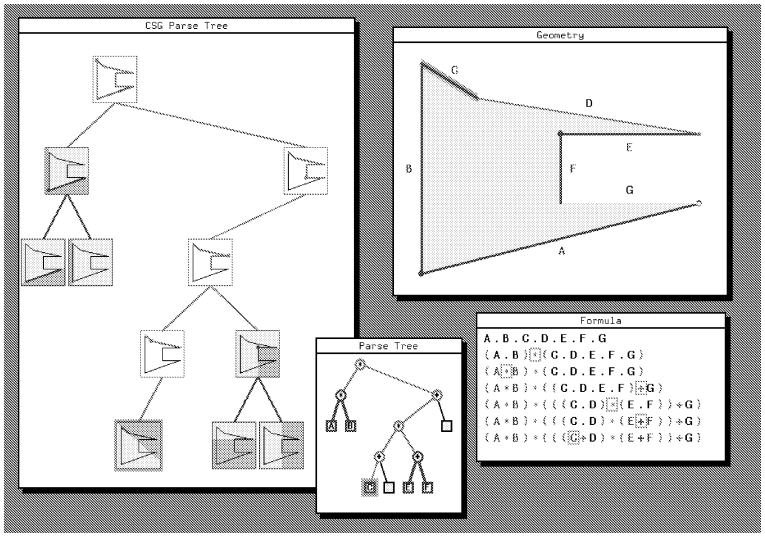


Figure 1.4.1: Exemple d'animation avec Zeus

1.4.2 Aladdin

Aladdin (ALgorithm Animation Design and Description using INteraction) est un système d'animation d'algorithmes quelque peu différent de Zeus. La motivation de ce système était de permettre d'animer un algorithme sans avoir à programmer quoi que ce soit (ou presque). Les principaux atouts d'Aladdin sont deux outils très intéressants : l'ESA (Editor for Specifying Animations) et l'AULIKKI (Animation User's Lovely Interactive Knack KIt). Vous pouvez trouver une présentation complète du système Aladdin dans l'article [12].

La création du système Aladdin part d'une constatation assez logique : l'animation est un outil très intéressant et très utile. Cependant, son utilisation n'est que très peu répandue à cause de son manque de facilité de création. En effet, tous les systèmes offrent des possibilités évoluées, mais aucune ne permet de créer des animations avec un effort minime. Autrement dit, créer une animation peut se révéler très prohibitif.

C'est pourquoi Aladdin se voulait être un système simple d'emploi et de création. Ce système est essentiellement basé sur l'interaction avec l'utilisateur. Son but est de permettre l'animation d'un algorithme :

- sans avoir à programmer les animations
- pour n'importe quel algorithme écrit en Modula-2 (certains systèmes offrent des animations sur une liste d'algorithmes)
- en ayant un contrôle total sur le déroulement de l'animation

Le principal slogan d'Aladdin est : "Pas de programmation d'animations, mais spécification graphique d'animations". Il s'agit effectivement de l'utilisation de ce système. Il se base, comme nous l'avons dit plus haut, sur deux outils : ESA et AULIKKI.

ESA est un éditeur de textes qui va permettre la spécification totale de l'animation en utilisant trois entités : *graphical type* , qui décrit l'apparence graphique d'un objet d'animation, *graphical variable* , qui va permettre la liaison avec le code animé et *animation statements* , qui décrit en détail l'animation.

Aladdin permet de créer très simplement de nouveaux objets graphiques en utilisant un simple logiciel de dessin du style MacDraw (Aladdin tourne principalement sur Macintosh). Aladdin s'est, en fait, concentré sur l'utilisation et la création simple des éléments graphiques d'une animation.

AULIKKI fournit l'environnement nécessaire à l'exécution d'une animation. Cet outil permet de contrôler le déroulement complet de l'animation et par là même l'exécution de l'algorithme. AULIKKI offre la véritable interaction avec l'utilisateur, il est possible de modifier les différentes vues d'un algorithme afin d'agir sur l'exécution du code et non plus seulement sur la simple apparence de l'animation. De nombreuses possibilités sont offertes comme rejouer une animation, modifier dynamiquement l'ensemble de l'application (i.e. animation et/ou code), changement de taille, zoom, etc.

Aladdin est un système d'animation qui a beaucoup de qualités. C'est un tentative intéressante de simplification pour la création d'animation. ESA et AULIKKI sont des outils bien aboutis et très évolués. L'accent est particulièrement mis sur l'interaction avec l'utilisateur (AULIKKI est un environnement d'animation complet). Cependant, il s'agit d'un système très lourd, car pour exécuter une animation l'environnement est nécessaire. De plus, l'instrumentation du code est tout de même présente et non automatique. De même, la spécification n'est pas réellement une programmation d'animation mais elle s'en rapproche beaucoup (même si cela est caché par l'editeur ESA).

En bref, Aladdin représente une bonne réalisation en ce qui concerne l'interaction avec l'utilisateur. Il y a tout de même des manques au niveau de l'animation elle-même.

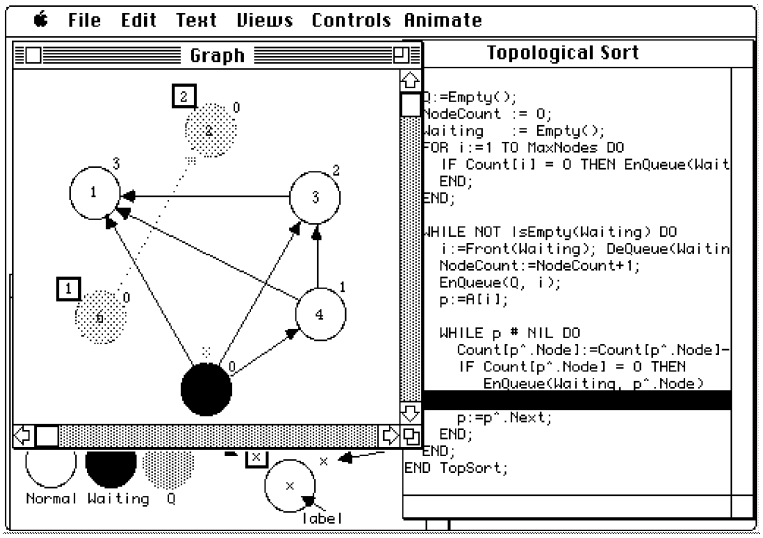


Figure 1.4.2: Exemple d'animation avec Aladdin

1.4.3 Agat

Agat (A*nother* G*raphic* A*nimation* T*ool*) a été créé à l'INRIA en 1993 par Olivier Arsac. La création de ce nouveau système d'animation fut motivée par l'étude d'autres systèmes (comme Zeus ou TANGO). En effet, comme nous l'avons vu précédemment, ces systèmes sont très complets mais souffrent d'un défaut important : la lourdeur de mise en oeuvre. Le but d'Agat|2| était la réalisation d'un système d'animation d'algorithmes simple à utiliser.

Ce système utilise un modèle client-serveur. Durant l'exécution de l'algorithme, les valeurs suivies sont transmises par des flux de données. Ces flux sont exploités par un processeur de flux qui permet ensuite de produire l'animation. Le système Agat a été écrit en C et Xlib.

Les étapes nécessaires à la production d'une animation sont :

- La préparation du code pour l'animation
- L'écriture d'un script d'animation

La préparation du code s'effectue par l'instrumentation de ce code. En utilisant une librairie C (*libagat.a*), le programmeur peut utiliser des fonctions simples de transmission de données sur un flux. Il est ainsi possible de suivre l'évolution des variables au cours de l'exécution d'un programme.

Mais, ces opérations ne permettent pas encore de produire l'animation d'un programme. En effet, il faut passer par un langage appelé *agat* (à différencier du système d'animation *Agat*). Ce petit langage permet de spécifier les différentes animations souhaitées pour les valeurs transmises. Il est possible dans ce langage d'utiliser des "librairies" de fonctions prédéfinies ainsi que de créer de nouvelles fonctions d'animation. Il existe deux types de fonctions dans le langage *agat*. Elles ne sont différenciables que par leurs sémantiques. Une première classe de fonctions permet de définir les valeurs suivies ainsi que les réglages sur les communications par flux (comme par exemple des délais). La deuxième classe de fonctions permet de créer des objets graphiques (e.g. *plot* , *thinbar*).

La simplicité et la puissance d'Agat reposent sur deux faits principaux : pour animer un programme existant, il suffit d'instrumenter le code de façon simple en incluant une librairie et des appels à des fonctions de transmissions de données. L'animation n'est pas spécifiée dans le code source. Le modèle client-serveur constitue le deuxième fait intéressant car cela permet de changer très facilement l'animation d'un algorithme; en effet, le programme source transmet ses données intéressantes et ne se préoccupe pas de la manière dont elles sont animées à l'autre bout du flux. Le programme *agat* interprète les données reçues et met à jour l'animation créée par le script d'animation.

Agat se révèle être également un outil intéressant par sa possibilité de "rejouer" des animations existantes. C'est à dire qu'il est possible de sauvegarder l'animation dans un fichier et de lancer le programme de visualisation d'animation afin de voir à nouveau cette même animation. Il est alors possible d'effectuer différents réglages comme la modification de la vitesse d'exécution pour voir une animation en 10 s. au lieu de 20 mn par exemple.

Des extensions graphiques ont été réalisées par la suite par Stéphane Lavirotte. Ainsi, une librairie 3D et de nouveaux opérateurs 2D agrémente à présent ce système d'animation.

Agat pourra nous être utile par son modèle client-serveur et par sa volonté de simplicité. Cependant, il existe dans Agat deux défauts. Tout d'abord, l'utilisation d'un nouveau

langage alourdit la création d'animation (même si cela permet l'indépendance du code animé et de l'animation) et les éléments graphiques sont écrits avec la Xlib, ce qui limite la simplicité de création de nouveaux opérateurs graphiques. Par ailleurs, Agat répond bien aux besoins de programmes impératifs, mais ce système ne permet pas l'animation des aspects intéressants de la programmation fonctionnelle. Par exemple, on peut tracer des variables, mais Agat ne permet pas de montrer l'évolution de listes infinies ou de fonctions de construction de haut-niveau sur des structures de données. Nous expliquerons dans la suite de ce rapport la manière dont nous avons traités ces notions.

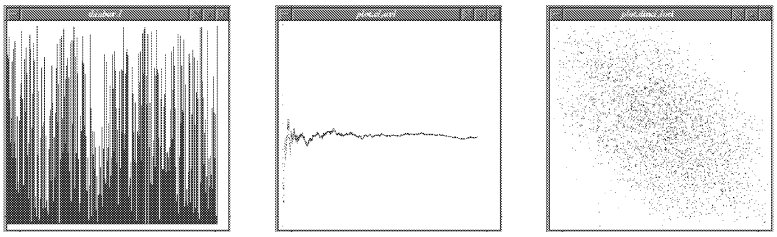


Figure 1.4.3: Trois vues d'un générateur de nombres aléatoires

Chapitre 2

Présentation de notre étude

2.1 Etat des manques

Comme nous l'avons vu avec les systèmes d'animation que nous venons d'étudier, certains aspects demandent à être améliorés, notamment la transparence et la simplicité du système d'animation. *Agat* constitue un des premiers efforts dans le sens de la simplicité et de la vitesse de mise en oeuvre. Mais, il faut aller plus loin, car il est intéressant de choisir l'implémentation d'un algorithme et de l'animer en touchant au code de façon minimale.

De plus, aucun système d'animation ne répond actuellement aux besoins des programmes fonctionnels, or certaines applications de la programmation fonctionnelle peuvent nécessiter un système d'animation permettant de mettre en valeur ses spécificités. Ainsi, un système de ce type doit permettre la visualisation de la fréquence et du temps d'utilisation des données et la manière dont elles évoluent et sont utilisées. Un système en programmation fonctionnelle doit faire comprendre à son utilisateur quelles relations unissent les données et les fonctions dans un programme ne contenant pas d'affectation et utilisant des évaluations strictes ou paresseuses.

2.2 Les aspects que nous avons traités

L'animation de programmes fonctionnels constitue un sujet d'étude assez vaste. Nous avons choisi de nous attacher à certains points importants caractéristiques de l'animation de programmes et de la programmation fonctionnelle. Nous nous sommes principalement préoccupés de la visualisation de l'évolution des structures de données.

Nous avons traités le problème des *objets composites*. Brièvement, le problème du suivi de l'évolution d'un objet composite se pose lors de l'instrumentation globale d'un code. Il est simple de "voir" un objet, mais il est plus difficile d'entrer dans les détails afin de voir ses "sous-objets".

L'*évaluation paresseuse* est une technique intéressante et très utilisée en programmation fonctionnelle. L'évaluation paresseuse permet de différer des calculs jusqu'au moment où l'on a besoin de leurs résultats. Le suivi ne doit pas bloquer l'animation en attente d'un résultat ou forcer le calcul de ce résultat.

La préparation d'une animation constitue un des gros problèmes de ce domaine. Il est nécessaire, pour donner la possibilité de réaliser des applications concrètes, de faciliter

au maximum la construction de l'animation. C'est pourquoi, nous avons travaillé dans l'optique de modifier le moins possible le code à animer. Pour ce faire, nous avons étudié un aspect intéressant de la programmation fonctionnelle : les constructeurs de type ou *opérations de haut-niveau.*

2.3 Les défis de la programmation fonctionnelle

Tous les systèmes d'animation existants utilisent des langages impératifs ou orienté-objets. Ceci implique que de tels systèmes d'animation d'algorithmes doivent tracer des affectations de variables. Or, nous travaillons ici sur les langages fonctionnels où les affectations de variables sont inexistantes, il n'existe donc pas de traçage d'affectations de variables.

Une autre spécificité de la programmation fonctionnelle correspond au fait que les effets de bord n'y existent pas. En effet, il est théoriquement impossible de réaliser une affectation de variable globale ou une entrée-sortie. Ceci vient du fait que la programmation fonctionnelle permet à ses utilisateurs de connaitre la valeur d'une fonction par la valeur de ses composants et rien d'autre. C'est pourquoi, un langage fonctionnel n'autorise pas, en théorie, l'affectation de variables globales ou l'affichage de données, puisque la valeur de la fonction serait alors son résultat *plus* l'affectation de la variable globale *plus* l'affichage. L'avantage de cette règle assez contraignante est la cohérence des programmes et son rapprochement par rapport aux mathématiques.

Cepedant, dans notre cas, nous devons effectuer des sorties à l'écran puisqu'il nous est nécessaire d'afficher les données que nous suivons. Bien sûr, la plupart des langages dits fonctionnels autorise les effets de bord. Le point que nous pouvons préciser à ce sujet, c'est qu'il nous est possible de rester fonctionnel en effectuant des sorties écran s'il nous est permis d'utiliser un langage non fonctionnel qui prend en entrée des valeurs (émises par le programme fonctionnel) et qui affiche lui-même les informations nécessaires à l'ecran. C'est ce que nous ferons pour ce projet.

Il est important et intéressant de noter que lorsque l'on travaille en programmation fonctionnelle, l'abstraction n'est pas forcément l'élément primordial. En effet, la plupart du temps, les programmes fonctionnels utilisent uniquement des fonctions (dont les résultats peuvent servir d'entrées à d'autres fonctions etc.) mais ne nécessitent pas obligatoirement l'utilisation de types abstraits de données.

Cependant, pour pouvoir animer et même simplement utiliser des structures de données intéressantes, on doit pouvoir se placer dans un contexte de programmation modulaire. Rien n'empêche d'utiliser des structures de données évoluées tout en restant fonctionnel. En effet, il suffit de traiter les types abstraits comme des valeurs au même titre que des entiers ou des fonctions. L'abstraction en programmation fonctionnelle permet de travailler sur des objets complexes. Si l'on peut donner un type et une valeur à une type abstrait, il est alors possible de traiter ces structures complexes comme de simples objets fonctionnels. Les langages ML utilisent déjà des types abstraits (appelés modules).

En animation de programmes fonctionnels, on doit pouvoir animer n'importe quelle valeur. Il est donc important de donner la possibilité d'animer des types abstraits au même titre que des fonctions. La question reste : comment ? Si l'on considère que l'abstraction appliquée à la programmation fonctionnelle s'avère équivalente aux autres concepts, on

peut déjà prévoir que très peu d'efforts seront à fournir pour réaliser cela. Comme nous le verrons plus loin, nous allons tenter d'instrumenter les fonctions afin de voir la manière dont elles s'exécutent. Nous utiliserons la même technique pour les types abstraits de données.

2.4 Les aspects traités

2.4.1 Objets composites

Présentation

Lorsque l'on veut suivre l'évolution des structures de données durant l'exécution d'un programme, plusieurs problèmes se posent. Nous avons tout d'abord décidé de traiter le cas des objets composites.

Que sont les objets composites?

Il est important de savoir exactement de quoi il s'agit. Les objets composites sont des données incorporant des sous-données. C'est à dire que ces objets contiennent des objets sémantiquement plus petits et dont l'usage est souvent interne à l'objet principal. Notre but est de pouvoir animer ce type de données. Des opérations sont effectuées sur ces objets, des fonctions qui travaillent sur ces objets sont appelées, des consultations son faites sur ces objets. Tout cela nous devons pouvoir le suivre et le montrer dans le détail. C'est à dire que les sous-objets (les composants de l'objet composite) doivent être également pris en compte. Affichés puis animés, ces composants permettent à l'utilisateur (i.e. la personne qui regarde l'animation) de comprendre quels sont les sous-objets mis en jeu et à quel moment.

Pour ce faire, nous avons défini un modèle d'utilisation, une sorte de schéma général. L'avantage de ce modèle est la possibilité de l'étendre aux autres aspects que nous devons traiter et même de pouvoir par la suite s'en servir pour des extensions du système. Nous avons choisi un modèle pseudo-"client/serveur". En effet, il s'agit en fait de réaliser un découpage de notre système. Trois parties se distinguent (le code à animer, un serveur d'animation et les animations). Chacune de ces parties communiquent par des appels de fonctions.

Grâce à ce modèle, les objets principaux (que nous appelons objets composites) se connectent au serveur. Ce sont eux qui communiqueront les informations nécessaires à l'animation. Leurs sous-objets ne sont pas en communication directe avec le serveur, pour transmettre leurs informations propres, ils doivent passer par leur objet englobant.

Application : Arithmétique sur les polynômes

Le problème de l'arithmétique sur les polynômes permet d'illustrer de manière intéressante le suivi de l'évolution des structures de données. Cela représente également une application concrète de l'animation des objets composites.

Brièvement, il s'agit ici de visualiser l'addition, la multiplication, toutes les opérations associées aux anneaux polynomiaux. Chaque polynôme est composé d'un certain nombre de monômes. Lors d'une addition par exemple, l'utilisateur peut voir à quel moment sont utilisés chacun des monômes. On peut voir également "en temps réel" (i.e. monôme par monôme) la construction du polynôme calculé par l'addition.

Dans cet exemple, l'objet composite est représenté par le polynome. Chaque sous-objet correspond à un monôme. Le polynôme communiquera au serveur les informations de coefficient et de puissance de chaque monôme.

2.4.2 Evaluation paresseuse

Présentation

L'évaluation paresseuse constitue un des problèmes les plus intéressants et une des techniques les plus utiles en programmation fonctionnelle. Il s'agit d'être capable de retarder l'évaluation d'une opération jusqu'à ce qu'on ait besoin du résultat. Cette technique présente deux avantages. D'une part, cela permet de ne pas exécuter des opérations inutiles et donc d'améliorer les temps de calcul. D'autre part, l'évaluation paresseuse ouvre de nouveaux horizons car elle permet de construire de nouvelles structures de données comme les listes infinies.

L'animation d'algorithmes utilisant l'évaluation paresseuse pose un problème. Comment tracer une évaluation de ce type sans forcer les calculs ni bloquer l'exécution d'autres instructions ?

Souvent, l'évaluation paresseuse est difficile à suivre (voir [15]). Il serait donc intéressant de trouver un moyen de montrer l'évolution de objets "paresseux" en ne modifiant ni la structure ni la façon dont ils évoluent. En effet, il ne serait pas très avisé de forcer ce type de calcul, sous peine de perdre tout l'intérêt de cette technique. Il faut donc attendre que les calculs soient effectués et que de nouvelles valeurs soient présentes pour pouvoir les présenter et mettre à jour l'animation. Cependant, le programme ne doit pas scruter l'objet paresseux sinon plus aucun calcul n'est possible (dans ce cas, on peut comprendre rapidement le peu d'intérêt que présente cette solution !).

La solution que nous avons retenue permet non seulement de régler ce problème mais aussi de toucher très peu au code de départ. Il s'agit en effet d'englober (ou d'emballer) la fonction d'évaluation. Ainsi, le système d'animation ne force pas son exécution et ne se met pas en attente d'une nouvelle valeur. La fonction englobante permet simplement d'exécuter des instructions supplémentaires (instructions d'animation par exemple) avant ou après la "vraie" fonction d'évaluation.

Application : Arithmétique sur les développements infinis

Ce choix d'application a été motivé par le fait que les développements infinis sur des séries de puissance constituent un problème bien connu. Notre but ici est de réaliser une expérience sur un exemple intéressant et utilisé afin de montrer le potentiel de notre animation des évaluations paresseuses.

De plus, cette application présente deux avantages pédagogiques. Tout d'abord, une animation montrant l'évolution des opérations arithmétiques sur les développements infinis peut représenter un support de cours intéressant pour l'enseignement de ce sujet. D'autre part, dans un domaine plus "informatique", ce genre d'animation permet en même temps de comprendre ce qu'est l'evaluation paresseuse et pourquoi elle est nécessaire.

Cet exemple va permettre d'illustrer notre propos sur l'animation d'une évaluation paresseuse. Le code de départ décrit une "stream", c'est à dire un flux de données infini.

Cette objet constitue la brique de base des objets que nous désirons construire. Les développements infinis sont des objets composites avec comme sous-objets des monômes. Nous souhaitons animer les différentes opérations arithmétiques d'un anneau (addition, multiplication ...) définies pour ces développements infinis. L'utilisateur pourra visualiser à quel moment les calculs sur le flux sont nécessaires (i.e. quand la fonction d'évaluation est appelée).

2.4.3 Modification minimale du code et opérations de haut-niveau

Présentation

Les opérations de haut-niveau permettent d'exploiter un aspect important de la programmation fonctionnelle. Il s'agit de sa relation avec l'abstraction. Nous avons identifié trois cas sur lesquels nous pouvons tirer des bénéfices de ces opérations dites de "haut-niveau" :

Tout d'abord, les fonctions que l'on peut emballer constituent l'application la plus simple. En fait, il s'agit uniquement de créer une fonction animée à partir d'une fonction à animer.

```
f(g) == { ..... g(h,k) ..... }   -- on a abstrait la fonction g
f(+)                             -- on utilise f sur +
f(anim(+))                       -- ou sur anim(+) qui execute egalement
                                 -- des instructions d'animation
```

Ensuite, l'abstraction de types permet d'emballer les fonctions travaillant sur un type. Ainsi, l'utilisation d'un programme à des fins d'animation permettra de ne pas modifier le programme. La modification intervient au niveau des types utilisés.

```
T est un type avec l'operation + : (T,T) -> T
h(T) est aussi un type avec    + : (h(T),h(T)) -> h(T)

La fonction + de h(T) execute une animation et le + de T.
```

Enfin, il existe une troisième possibilité concernant les opérations de haut-niveau, ce sont les constructeurs de type. Il s'agit d'une combinaison des deux idées précédentes : on peut emballer la fonction qui construit un type pour obtenir une fonction similaire qui rend le même type mais dont les fonctions sont instrumentées pour l'animation.

On discerne deux avantages principaux de l'utilisation de ces constructeurs de types. D'une part, cela permet d'obtenir une bonne abstraction des données que l'on manipule tout en restant dans un cadre fonctionnel. D'autre part, cette technique va nous aider dans notre objectif de modification minimale de code à animer. En effet, nous nous sommes donné comme but de pouvoir animer un programme avec le moins d'instrumentation possible du code. Les constructeurs de type génèrent des types èquivalents aux types de base que les applications manipulent. On peut donc substituer le type "animé" au type de base en une ligne de code. Il faudra vraisemblablement rajouter au programme une instruction d'initialisation du système. Bien entendu, des animations par défaut seront prévues. Si, par la suite, l'utilisateur souhaite des animations plus élaborées, il lui faudra rajouter des

instructions dans le code afin de spécifier les animations désirées. Cependant, cette dernière possibilité ne constitue pas une obligation et dans la plupart des cas, l'animation par défaut pourrait être suffisante.

Application : Résolution des bases de Gröbner

Sans développer ici toute la théorie des bases de Gröbner [8], nous avons souhaité appliquer notre système d'animation à un exemple concret et de taille importante (ceci dans le but de sortir des "cas d'école"). Les bases de Gröbner sont des bases particulières pour des ensembles de polynômes appelés idéaux. Lorsque l'on considère un polynôme, on souhaite le représenter de façon canonique. L'intérêt de la théorie des bases de Gröbner est qu'il s'agit d'une théorie mathématique qui très simplement transposable en méthode algorithmique. Cela permet de résoudre de nombreux problèmes basés sur la théorie des idéaux polynomiaux et la géométrie algébrique. Des applications réelles se modélisent souvent par des ensembles de polynômes. Après la transformation de ces sytèmes par un algorithme de résolution des bases de Gröbner, on peut résoudre beaucoup plus facilement les problèmes posés. Il existe de nombreuses applications comme la preuve automatique de théorêmes géométriques, décomposition primaire d'objets géométriques ou encore cinématique inverse dans la programmation d'un robot).

En ce qui concerne notre sujet, la taille importante de cet exemple et sa nécessité d'utilisation d'un type abstrait de données constituent son intérêt principal. La taille "réelle" de cette application de notre système d'animation nous permettra de connaître sa robustesse et son comportement face à un programme utile. Par une présentation appropriée de l'évolution du programme et des structures de données, une animation d'algorithmes de résolution des bases de Gröbner pourrait être utile. Elle permettrait aux scientifiques travaillant sur le sujet de concevoir de nouveaux algorithmes plus performants en les aidant à trouver de nouvelles heuristiques par exemple.

D'autre part, un aspect important est traité par cet exemple. Il s'agit de démontrer les intéractions entre l'abstraction des données et la programmation fonctionnelle. Nous avons vu dans la partie présentation que les types de données vont être emballés afin d'animer leur évolution. Il est important de se rappeler qu'en programmation fonctionnelle, la valeur d'un objet peut être déduite de la valeur de ses composantes. Or, la base de notre animation correspond tout à fait à cela. En effet, pour emballer un type, il suffit d'emballer les composantes qui l'utilisent autrement dit les fonctions travaillant sur ce type.

Chapitre 3

$A\sharp$ et Tcl/Tk

3.1 Le langage $A\sharp$

3.1.1 Présentation

$A\sharp$ se situe à mi-chemin entre la programmation orientée objet et la programmation fonctionnelle. En effet, il est par exemple possible de construire un type et les fonctions associées dans une seule et même structure de données. De plus, les fonctions et les types manipulés par $A\sharp$ sont de première classe, c'est à dire qu'on peut les utiliser comme n'importe quel autre objet (entier ou flottant par exemple).

$A\sharp$ est un langage qui se destine aux utilisateurs de calculs algébriques informatisés. Plusieurs constatations sont à l'origine de sa création. Tout d'abord, la plupart des systèmes de calcul algébrique sont fermés, c'est à dire que leurs relations avec le monde extérieur sont pratiquement inexistantes. Les échanges avec des langages comme C ou Fortran s'avèrent très coûteux. $A\sharp$ se veut un langage permettant de réaliser des extensions pour des systèmes comme AXIOM et offre des liaisons facilitées avec des langages comme C. La notion principale utilisée par $A\sharp$ est le partage de code et la compilation séparée. De cette manière, on peut appeler dans un programme des fonctions venant de différentes librairies. Il est également possible d'exporter et d'importer n'importe quelle donnée. Il existe par exemple un mapping entre les fonctions C et les fonctions $A\sharp$. Un interfaçage fort avec AXIOM a été également réalisé. Enfin, $A\sharp$ permet de générer du code C ou Lisp afin de le porter vers d'autres systèmes.

L'interactivité prend une place grandissante chez les utilisateurs de systèmes algébriques, c'est pourquoi $A\sharp$ offre également un interpréteur et donne la possibilité de construire des prototypes d'applications en utilisant l'environnement interactif d'AXIOM.

Le langage possède un autre avantage : l'optimisation. La plupart des systèmes algébriques fermés offrent des fonctionnalités généralement coûteuse en temps de calcul. Ceci est dû à leur structure qui est généralement constituée par un noyau écrit dans un autre langage. Les fonctions nécessaires aux applications sont écrites dans des couches placées au-dessus de ce noyau. Le problème est que ces fonctions peuvent être de 10 à 10 000 fois moins rapides que des fonctions analogues qui auraient pu être écrites dans le noyau. En $A\sharp$, il est possible de décider à la compilation quel type d'optimisation on souhaite appliquer afin d'obtenir des exécutions efficaces.

Le langage $A^\sharp$ est décrit en détail dans [19]. L'article [20] présente le langage et donne les motivations de sa création.

3.1.2 La compilation et le monde extérieur en $A^\sharp$

La machine abstraite / Foam

La compilation en $A^\sharp$ utilise un langage intermédiaire appelé *Foam* (pour **F**irst **O**rder **A**bstract **M**achine). *Foam* a été construit de manière à contenir les concepts réalisables efficacement à la fois en Lisp, en C et en codes machines. Par exemple, il est impossible de demander l'adresse d'une variable car cela serait inefficace en Lisp (il serait nécessaire de créer deux fermetures).

Cependant, *Foam* n'est pas restreint à l'exact intersection entre Lisp et C. Certains concepts peuvent être traités par des librairies (par exemple, il supporte l'idée du ramasse-miettes, cela étant géré en C par une librairie).

La machine abstraite n'utilise pas les types de $A^\sharp$. Le générateur de code doit lui fournir les appels pour la création et la gestion des types. Cela présente l'avantage de pouvoir utiliser ces appels pour ajouter au système de nouvelles représentations de types.

Un générateur de code *i386* a été écrit en 1996 par un étudiant du département informatique d'ETH à Zurich.

Optimisation

Comme nous l'avons dit plus haut, un des avantages de la compilation de $A^\sharp$ est la possibilité de choisir le degré d'optimisation. Plusieurs optimisations sont proposées.

Ainsi, le compilateur $A^\sharp$ propose la *program specialization* , qui permet d'exploiter les instances particulières des types génériques (ceci évite les résolutions au moment de l'exécution).

La *procedural integration* (appelée également *inlining*) élimine les appels de certaines fonctions pour les remplacer par l'exécution directe du code de la fonction (on évite ainsi le coût d'un appel de fonction).

On peut également citer *data structure elimination, dataflow analysis of condition variables* ou encore *copy propagation* .

Ouverture vers d'autres langages

Un des principaux buts de $A^\sharp$ est d'offrir une interface entre les systèmes de calcul algébrique et les autres langages plus classiques comme *C* ou *Fortran* . C'est pourquoi, le compilateur $A^\sharp$ permet de générer du code C ou du code Lisp directement à partir du code Foam.

Le code C peut être généré au format K&R ou ANSI. Il utilise des macros pour les différentes opérations arithmétiques.

En ce qui concerne le Lisp, le code généré est actuellement au format d'un langage de macros qui peut être implémenté soit en Common Lisp, soit en Scheme. Le code Lisp généré utilise des macros pour donner accès aux informations dont aurait besoin un compilateur Lisp (opérations de bas niveau).

En $A^\sharp$, il est possible de générer du code objet classique (i.e. au format des fichiers objets généré par C). $A^\sharp$ permet également d'importer du code objet et d'avoir des liaisons avec des librairies de fonctions. Par exemple, le langage donne un accès à la librairie des fonctions XWindow. Afin de rester en liaison avec les systèmes algébriques, $A^\sharp$ offre aussi un fort interfaçage avec *AXIOM* .

3.2 Pourquoi $A^\sharp$?

3.2.1 Notions fondamentales

Un premier élément important et très intéressant du langage est la possibilité d'abstraire les données manipulées. Les *domaines* sont les entités donnant accès à cette abstraction. La section suivante décrit les types abstraits de données utilisés dans $A^\sharp$. C'est cet aspect du langage qui nous permet de dire que $A^\sharp$ est en partie un langage orienté objet.

Un autre aspect de $A^\sharp$ correspond à sa partie fonctionnelle. En effet, le langage permet au programmeur de travailler sur des fermetures afin de créer toutes sortes d'objets. De plus, comme les types et les fonctions sont considérés comme des valeurs comme les autres, il est possible de construire des modules de façon très simple. Ces deux notions importantes (abstraction et programmation fonctionnelle) furent les deux critères prédominants pour notre choix du langage d'application.

3.2.2 Un langage bien adapté à notre problème

Pour réaliser notre système d'animation, nous avions besoin d'un langage fonctionnel de haut niveau nous permettant de réaliser une application de notre étude. Pour cela, $A^\sharp$ constitue un langage intéressant. En effet, il permet de réaliser tout ce dont nous avons besoin en programmation fonctionnelle. C'est un langage fortement typé qui offre des opérations de haut-niveau sur des structures de données. Il est possible par exemple de construire dynamiquement des types par des appels de fonctions. Cette paramétrisation offre un potentiel intéressant de modularité. Etant un véritable langage fonctionnel, $A^\sharp$ permet de manipuler des types, des fonctions et des valeurs numériques de la même manière. Enfin, $A^\sharp$ permet de réaliser des programmes utilisant l'évaluation paresseuse, mais surtout il est possible d'élaborer des programmes sophistiqués avec des fermetures. C'est pourquoi il s'agit bel et bien d'un langage permettant de réaliser de "vrais" programmes fonctionnels.

Par ailleurs, $A^\sharp$ est assez répandu ce qui nous offre un potentiel intéressant d'utilisateurs et surtout d'applications possibles à animer. Notre système d'animation conçu pour ce langage pourra être utilisé en situation réelle, ce qui constitue l'aboutissement le plus important de tout travail.

Enfin, son ouverture vers l'extérieur nous permettra d'utiliser simplement des outils graphiques pour réaliser nos animations. Une intégration de ces outils dans le langage sera possible assez rapidement et simplement. De plus, comme $A^\sharp$ offre une forte interface avec le langage C, nous avons à notre disposition des possibilités importantes et intéressantes en matière d'extension et d'interfaçage de boites à outils graphiques.

3.3 Les types de $A^\sharp$

Le typage fort constitue un autre intérêt du langage $A^\sharp$. Les types de $A^\sharp$ sont, comme nous l'avons dit plus haut, des objets de première classe. Ils sont donc manipulables comme toute autre valeur (entier par exemple).

Nous allons voir, dans cette section, comment construire des types en $A^\sharp$. Nous introduirons également la notion de *types dépendants*. Enfin, un petit paragraphe expliquera brièvement l'inférence de types du langage.

3.3.1 Domaines

Les domaines peuvent être utilisés pour deux buts :

- Créer un agrégat ou *package*
- Créer un *type abstrait de données*

Le *package* est la forme la plus simple de domaine. Il s'agit d'un simple regroupement de valeurs (types, fonctions, listes, entiers, ...). L'intérêt d'une telle structure réside dans le fait qu'on peut l'importer globalement et ainsi, en une instruction, avoir accès à tous les objets qu'elle contient.

Les domaines sont exploitables sous une deuxième forme. Elle est un peu plus complexe et par là même beaucoup plus intéressante, car elle offre un plus vaste champ d'applications. Il s'agit du *type abstrait de données*. Le principe de ce type de domaine est de structurer des valeurs et les opérations sur ces valeurs.

L'instruction clé pour créer les domaines s'appelle **add**. Cette instruction permet de créer un domaine à partir de rien ou basé sur un autre domaine. Il peut donc y avoir un lien d'héritage entre deux domaines. Le format du add est le suivant : *[Parent] add AddBody* . Le corps du add est une suite de définitions.

Un concept intéressant des domaines est la *représentation de types* (*Rep*). Grâce à cela, un domaine précise la structure interne des données manipulées par le domaine. Deux expressions spéciales (*rep* et *per*) associées à la représentation de données permettent de passer du domaine défini à sa représentation et *vice-versa* .

3.3.2 Catégories

Les catégories comme les domaines sont des types particuliers. Elles déterminent un ensemble de types et sont elles-même des types. Elles constituent, en fait, l'interface publique des domaines. Une catégorie contient la liste des opérations minimales qu'un domaine souhaitant appartenir à cette catégorie doit proposer. Il est donc possible d'avoir un contrôle sur les opérations minimales que doit fournir une "famille" de domaines.

Les catégories peuvent se baser sur d'autres catégories (un domaine doit alors fournir l'*union* des opérations demandées par les deux catégories).

L'instruction clé de création de catégorie se nomme **with**.

Voici un exemple de création d'un type par la définition d'une catégorie et d'un domaine :

```
categorie == with {
        scale: (Integer,Integer) -> Integer;
        n: Integer;
}
```

Cette catégorie ne se base sur aucune autre catégorie (pas d'héritage) et demande aux domaines qui lui appartiendront de définir une fonction *scale* et un entier *n*. Voici un domaine répondant à ces critères :

```
domaine: categorie == add {
        scale(i: Integer, j: Integer): Integer == 2*(i+j);
        n: Integer == 0;
}
```

Le domaine *domaine* est de type *categorie* et definit la fonction *scale* et un entier *n*. Rien n'empêche le domaine de définir des fonctions supplémentaires (soit locales, soit publiques).

3.3.3 Types dépendants

Un *type dépendant* est un type T dans lequel le type d'une sous-expression de T dépend de la valeur d'une autre sous-expression. Il est ainsi possible de déterminer le type d'un paramètre d'une fonction en fonction des valeurs passées à d'autres paramètres. Les types dépendants sont utilisés par les travaux en logique combinatoire [10].

Voici un exemple caractéristique :

```
sumlist(R: Ring, l: List R): R == {
        s: R := 0;
        for x in l repeat s:= s + x;
        s
}

import form List Integer, List SingleFloat;
print « sumlist(Integer, [2,3,4,5]) « newline;
print « sumlist(SingleFloat, [2.0,2.1,2.2,2.4]) « newline;
```

Le *type* de l et le *type* de la fonction (i.e le type de retour de la fonction) dépendent tous les deux du paramètre R. Le type *Ring* est une catégorie.

La principale raison de l'existence des types dépendants réside dans la possibilité de factoriser du code. Cela permet de gagner un temps important de développement. On y gagne également en puissance d'expression, car, de cette façon, il est possible de créer des types génériques. Cette généricité de code vient principalement du fait que les types dépendants permettent une liaison tardive (*late binding*), ce qui permet de retarder la résolution de types et offre donc une plus grande souplesse d'utilisation ; cependant, la sécurité de programmation est conservée grâce à la vérification statique de ce typage.

Les types dépendants de $A^\sharp$ existent par la manière dont sont créés les types dans ce langage. En effet, comme nous sommes dans un contexte de programmation fonctionnelle, la création de types est réalisée par des fonctions. C'est pourquoi, il est possible de passer des paramètres à ces fonctions de création.

3.4 Quelques notions sur le langage

Pour terminer cette présentation du langage $A^\sharp$, voici un petit exemple commenté. Bien sûr, nous ne donnerons ici qu'un aperçu du langage. Pour plus de renseignements, il est possible de se référer au guide du langage ainsi qu'à l'article de présentation de $A^\sharp$.

Voici le source d'une définition simple d'un type *Stack* .

Construisons tout d'abord la catégorie *Stack_ Cat* .

```
define Stack_Cat(S: BasicType): Category == BasicType with {

                    pop    : % -> %;
                    push   : (%,S) -> %;
                    peek   : % -> S;
                    empty? : % -> Boolean;
                    empty  : () -> %;

                    export from S;
}
```

- *Stack_ Cat* est une fonction qui prend en paramètre un domaine S de catégorie *BasicType* et qui rend une catégorie. Il s'agit ici de pure programmation fonctionnelle puisque les constructions de types sont de simples fonctions.

- *BasicType* est une catégorie qui contient des types comme *Integer* ou *String* .

- *Stack_ Cat* est une catégorie basée sur la catégorie BasicType et est définie par le *with* .

- Pour définir un type *Stack* (c'est à dire pour créer un domaine correspondant), il faudra définir les opérations classiques sur les piles (push, pop, ...) ainsi que les opérations demandées par BasicType (égalité, sample, «).

- *export from S* permet, lors de l'importation du type Stack, d'importer automatiquement les opérations du type S.

```
Stack(S: BasicType): Stack_Cat(S) == add {

        -- Representation interne

        Rep == Record(contents : List S);
        import from Rep;

        -- Operations sur une pile

        pop(s: %) : % == per [rest rep(s).contents];

        push(s:%,elem:S) : % == per [cons(elem, rep(s).contents)]

        peek?(s: %) : S == first rep(s).contents;
        empty?(s: %) : Boolean == empty? rep(s).contents;
        empty() : % == per [empty()];

        -- necessaire pour satisfaire BasicType

        (p: TextWriter) « (s: %): TextWriter == p « "<stack>";
        (a: %) = (b: %) : Boolean == error "no equality on stacks";
        sample: % == empty();
}
```

- *Stack* est une fonction qui prend un domaine S en paramètre (de catégorie *BasicType*) et qui rend un domaine de catégorie Stack_Cat(S).
- La définition du domaine se découpe en trois parties : *Représentation interne, Opérations sur les piles, Opérations demandées par BasicType* .
- La représentation interne est indiquée par le type **Rep**. Ici le type Stack est représenté par une liste d'objets de type S (*List S*). *Import from Rep* permet d'importer toutes les opérations sur le type List.
- Ensuite, viennent les définitions des opérations sur le type Stack. Le % désigne le type qui est en train d'être construit (à savoir Stack(S)). Prenons par exemple la fonction *pop* : l'instruction *per* transforme une liste en *per*cent (%), c'est à dire convertit la liste dans le type *Stack*.
- Les opérations nécessaires pour satisfaire BasicType sont la sortie écran et le test d'égalité. Il faut en outre définir un objet *sample* qui doit contenir une valeur du type que l'on construit (ici la pile vide).

Le code pour la définition du type *Stack* est maintenant écrit. Voici une petite fonction de test qui va nous permettre d'essayer ce nouveau type.

```
test():() == {
        import from Stack SingleInteger;
        s : Stack SingleInteger := empty();

        s:=push(push(push(push(s,3),4),5),6);

        print « "Peek : " « peek s « newline;

        while not empty?(s) repeat {
                print « peek s « newline;
s := pop s;
        }
}

test();
```

- la fonction ne prend pas de paramètres et ne retourne rien. Elle ne vaut donc que par ses effets de bord.
- Nous allons utiliser une pile d'entiers (*import from Stack SingleInteger*).
- Déclaration de la variable s comme une pile initialisée à la pile vide.
- Plusieurs appels à la fonction *push* pour empiler des valeurs.
- Affichage du sommet de pile. Ici, la syntaxe est proche du C++ : "<<" envoie une chaine de caractères sur un flux (ou *TextWriter* dans $A^\sharp$).
- Boucle **while** pour afficher successivement les valeurs de la pile.
- Enfin, appel de la fonction *test* .

3.5 Intégration de Tk dans $A^\sharp$

Les animations d'algorithmes ne présente un intérêt potentiel que si ces animations sont un tant soit peu conviviales et agréables à utiliser et à regarder. Il est donc nécessaire de prévoir l'utilisation d'une interface graphique. Cependant, $A^\sharp$ ne dispose pas en standard d'un accès rapide à utiliser à des fonctions graphiques (à part la Xlib mais qui n'est pas un modèle de simplicité).

Nous avons donc choisi d'utiliser Tk. Il y a plusieurs raisons à cela. Tout d'abord Tk offre un interfaçage fort avec C. Il sera donc possible si besoin est de programmer de nouveaux éléments assez facilement. De plus, Tk permet de programmer une interface graphique plus simplement qu'avec la Xlib ou même Motif. Une intégration complète à $A^\sharp$ pourra offrir aux programmeurs d'animation l'accès à une librairie graphique simple à programmer. Enfin, Tk offre un nombre assez important de fonctionnalités ce qui permet de construire des objets graphiques suffisamment élaborés pour travailler avec des animations intéressantes.

3.5.1 Les librairies Tcl/Tk en C

Le premier travail d'interfaçage a été l'étude (rapide) de la manière d'appeler des fonctions Tk 4.0 à partir de C [21, 16]. Il existe deux moyens d'arriver à ce résultat. La possibilité la plus simple (celle que nous avons retenue pour l'instant) mais aussi la moins rapide consiste à créer un interprète Tcl (`Tcl_CreateInterp`) et à lui passer des commandes TclTk par une fonction appropriée (`Tcl_Eval`). Les fonctions `Tk_MainLoop` ou `Tk_DoOneEvent` permettent de gérer les évènements de l'application. La fonction `Tk_CreateMainWindow` crée la fenêtre Tk principale, celle qui va correspondre au '.' lorsque l'on programme en tcl/tk.

Pour compiler l'application, il suffit d'utiliser la `libtcl.a` et la `libtk.a`. En ce qui concerne les boutons et les champs de saisies, il existe des fichiers de bindings permettant d'associer des évènements à ces objets.

3.5.2 Le passage à $A\sharp$

Le langage $A\sharp$ offre une possibilité d'interfaçage avec le C. Il est possible d'importer directement des fonctions C dans $A\sharp$, par l'instruction suivante :

```
import { [fonctions] } from Foreign C;
```

Par la suite, la fonction pourra être appelée dans le programme de la même manière que n'importe quelle autre fonction définie dans $A\sharp$. Ainsi, une fonction du type :

```
int foo(char *);
```

sera traduite en $A\sharp$ par

```
foo: String -> SingleInteger;
```

Actuellement, le problème est que l'on ne peut importer que des fonctions. Il existe un 'mapping' pour plusieurs types, mais dans sa version actuelle $A\sharp$ n'offre pas de liaison avec le C pour les structures. Il a donc fallu définir en $A\sharp$, des structures équivalentes à celles définies en C pour le TclInterpreter. Ceci étant fait, il ne reste plus qu'à importer les fonctions TclTk et l'interfaçage est réalisé.

3.5.3 Etat actuel de l'interfaçage

Nous avons réalisé actuellement une intégration de base de Tk dans $A\sharp$. Pour le moment, seuls les objets absolumment nécessaires à nos animations ont été intégrés, à savoir les `canvas`, les `frames`, les `labels` et les `toplevel`. Il est évident que ce n'est pas suffisant, mais nous comptons réaliser, dans la suite du stage, un interfaçage complet avec Tk, afin de donner accès à toutes les fonctionnalités de cette boite à outils.

3.5.4 L'utilisation de Tk dans notre système

L'animation d'un programme ne doit pas changer la manière dont il fonctionne, c'est pourquoi nous avons souhaité ne pas dominer le programme par le graphique. Habituellement, les programmes utilisant des fonctions graphiques comme Motif ou Tk nécessitent

une boucle pour gérer tous les évènements. Ici, nous souhaiterions exécuter le programme et, uniquement quand il y a une modification, nous voulons appeler le graphique. En Tk, il existe une fonction `Tk_MainLoop` (comme nous l'avons vu plus haut), cependant elle ne correspond pas à ce que nous souhaitons puisqu'il s'agit d'une boucle infinie qui récupère les évènements dès leur arrivée et qui empêche toute instruction ultérieure.

Dans l'optique de pouvoir afficher une fenêtre dès le début et de pouvoir ensuite la mettre à jour si le programme le souhaite, nous avons utilisé la fonction `Tk_DoOneEvent` qui permet une mise à jour ponctuelle. Le problème reste la gestion des évènements (comme un clic souris sur un bouton par exemple). Pour cela, nous avons retenu deux solutions : soit utiliser deux processus qui communiquent par pipes (un processus pour le code et un autre pour l'interface graphique), soit utiliser un timer qui appellerait la fonction `Tk_DoOneEvent` à intervalles réguliers. Nous pensons utiliser la deuxième solution qui offrirait l'avantage de ne pas alourdir la structure de notre système.

Chapitre 4

Objets composites

4.1 Traiter des objets composites

Ce premier exemple nous a permis de travailler sur un élément important de l'animation d'algorithme pour le suivi des structures de données. Il s'agit du suivi (ou traçage) des objets composites. Le problème du traçage des objets composites est le suivant : lorsque l'on veut montrer quelles opérations ont été réalisées ou quelles actions ont été faites sur une structure, on peut le faire de deux façons.

La plus simple et la plus naturelle tend à vouloir montrer de manière globale ce qui est arrivé à l'objet, c'est à dire que la structure de données est vue de manière unitaire comme un objet unique et compact. Dans ce cas, il n'y a pas de réelle difficulté, il suffit d'englober les fonctions agissant sur cette structure par des fonctions équivalentes qui appellent l'ancienne fonction et qui font également appel à des fonctions d'animation. Par exemple, si l'on veut suivre les opérations sur les entiers, on ne cherche pas à voir les actions sur chacun des bits qui les composent. On veut uniquement voir quel entier est utilisé ou voir un graphe représentant le nombre d'appels de l'addition sur telle ou telle valeur. Dans ce cas, il suffit de redéfinir la fonction + en lui demandant d'appeler une fonction `trace`. Celle-ci pourrait mettre à jour une ou des variables de comptage et afficherait un graphique correspondant aux données enregistrées puis appellerait la véritable fonction + pour donner le résultat. C'est une sorte de détournement d'appel de fonction.

Voici un exemple de ce que l'on peut faire pour "emballer" une fonction en $A^\sharp$:

```
f(a: Integer): Integer == a + 1;
   ++ Il s'agit ici de la fonction a animer

anim(g: Integer -> Integer): Integer -> Integer == {

     fonc(b: Integer) : Integer == {
           print « ``La fonction '' « g « `` a ete appelee'';
           g(b)
     }

     return fonc
}
-- La fonction anim cree une fonction d'emballage (fonc)
-- qui notifie l'appel de la fonction de base et puis realise
-- effectivement l'appel a cette fonction
```

En $A^\sharp$, la façon la plus naturelle d'écrire cela aurait été d'utiliser un fonction currifiée :

```
anim(g: Integer -> Integer)(b: Integer): Integer == {
       print « ``"La fonction '' « g « `` a ete appelee'';
       g(b)
}
```

Cependant, un problème se pose ici, car une animation sur des objets composites se doit de montrer l'évolution de ces objets *et* de leurs composants. C'est pourquoi la solution précédente ne peut plus raisonnablement s'appliquer.

Tout d'abord, expliquons quel est l'intérêt de montrer les sous-éléments d'un objet composite. La plupart des structures de données ne sont pas à proprement parler des modèles de simplicité, bien souvent, les programmes travaillent sur des structures permettant soit de représenter un nombre important d'informations hétérogènes (en ce qui concerne le type des éléments de cette structure), soit de construire un nouvel objet à partir de briques de base que nous appellerons composants. Dans ce dernier cas, la plupart des opérations portant sur ces objets englobants se réduiront à des opérations sur leurs composants. Autrement dit, un calcul sur un objet sera en fait des calculs sur des sous-objets. C'est pour cette raison qu'il est nécessaire de se demander comment suivre ce type d'opération. En effet, l'animation du simple objet englobant peut se révéler insuffisante au niveau des informations que souhaiterait voir l'utilisateur. Il faut également montrer comment sont exploités les composants de cet objet.

Pour ce faire, nous allons définir un modèle de fonctionnement pour l'animation. Ce modèle sera la base de notre système et définira un schéma général sur lequel les autres types d'animation devront calquer leur fonctionnement.

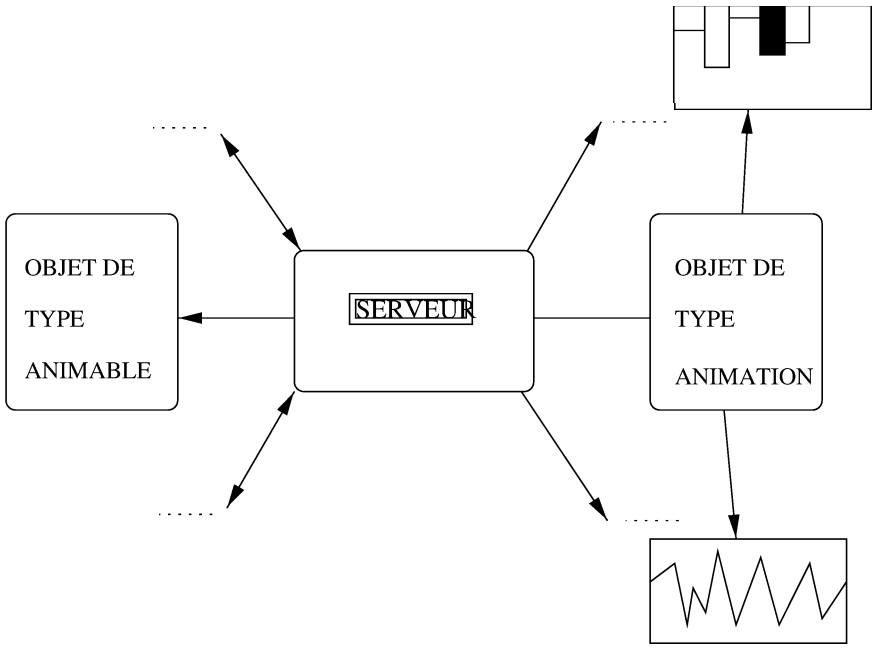


Figure 4.1: Schéma général de fonctionnement

La catégorie `Animable` permettra de représenter l'objet (composite ou non). Le serveur centralisera les requêtes (demande d'animation, notification d'évolution) et les transmettra à l'objet d'animation. Cependant, toutes les opérations ne passeront pas par le serveur. En fait, son rôle est principalement de mettre en communication l'objet animé et son animation. De cette manière l'objet animé pourra transmettre toutes les données que lui demande l'animation pour mettre à jour ses affichages. Par ce moyen, on peut alors animer des objets composites. Ce sont, en effet, les objets englobants qui vont transmettre à l'animation les informations que recèlent les composants de ces objets. Les composants mettent à disposition de leur objet englobant toutes les données nécessaires à leur animation. Les mises à jour nécessaires sont spécifiées par un appel de fonction. Le code est instrumenté ce qui permet une gestion fine des informations que peut voir l'utilisateur.

Actuellement, ce modèle pseudo-"client/serveur" est réalisé dans un seul et même processus. Les requêtes et transmissions d'informations sont en fait représentées par des appels de fonctions. Ce modèle présente l'avantage de bien structurer toutes les communications entre les différentes entités (objet animé, serveur et animation). De plus, un autre avantage est que, lorsque nous souhaiterons étendre notre application, nous pourrons la transférer sur un véritable modèle client-serveur de manière peu complexe et donc assez rapidement. Nous gardons pour l'instant notre structure de fonctionnement car cela représente un schéma beaucoup moins lourd et donc plus propice à des recherches sur l'essence même d'une animation.

Expliquons maintenant comment fonctionnent les animations. En fait, nous créons des types d'animation, c'est à dire que ce sont des objets spécialisés dans une animation. Chaque animation est en relation étroite (bien qu'indépendante avec l'objet qu'elle anime). Les animations savent de quelles informations elles ont besoin et sont donc à même de

demander à l'objet animé les données nécessaires. Ce ne sont pas des animations génériques, ce qui présenterait peu d'intérêt, car elles ne seraient pas assez précises.

Voici les fonctions que mettent à disposition les différentes entités :

`Animable`	`Server`	`Animation`
createGr	askForAnim	new
modify	stopAnim	createItems
srv	touched	change

- Les fonctions du serveur.

  La fonction `askForAnim` permet à un objet de demander à devenir un objet animé. La fonction `stopAnim` a l'effet inverse, i.e. ne plus animer un objet. Enfin, `touched` avertit le serveur de l'évolution d'une donnée dans l'objet animè.

- Les fonctions du module Animable.

  La fonction `createGr` permet de transmettre les informations utiles pour la création des objets à animer. La fonction `modify` transmet les données nécessaires à la mise à jour des affichages. Enfin, `srv` donne le serveur auquel l'objet animé est attaché.

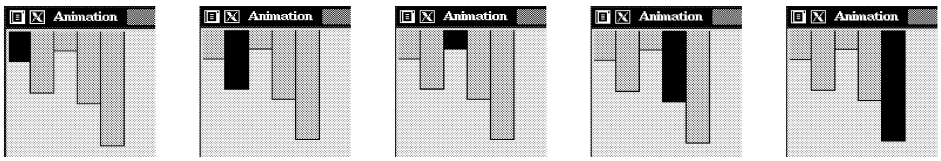
- Les fonctions du module Animation.

  La fonction `new` crée une animation. La fonction `createItems` est typiquement appelée par la fonction `createGr` du module *Animable* dans le cas d'un suivi d'objets composites. Enfin, `change` répercute les évolutions d'une structure de données sur l'animation (c'est une fonction de mise à jour).

4.2 Exemple : Les polynômes et les monômes

Afin de savoir si ce que nous venons d'exposer relève de la fantaisie ou si cela se révèle tout à fait exploitable, nous avons décidé de réaliser une application implémentant et utilisant ce système. L'exemple des polynômes nous permet de façon relativement simple et rapide de traiter cet aspect des objets composites. En effet, les polynômes sont des objets englobant des monômes, ce qui correspond tout à fait au schéma que nous avons décrit précédemment. Les objets composites seront donc les polynômes et les composants seront les monômes.

Tout d'abord, nous allons expliquer ce que nous désirons réellement faire avec cet exemple. L'animation des polynômes se basera en fait sur les opérations que l'on peut effectuer sur des anneaux polynômiaux, c'est à dire addition, multiplication, soustraction. Le but est de visualiser la manière dont ces opérations sont réalisées. Un algorithme de calcul de ces opérations sera implémenté, l'animation permettra de suivre le déroulement de ces opérations. On pourra voir ainsi quel monôme est utilisé à quel moment et comment se construit le polynôme résultat. Voici un exemple de ce que l'on peut voir lors de l'animation :



Pour ce faire, nous allons définir le polynome comme un objet “animable”, c'est à dire qui fournira les fonctions que nous avons vues plus haut. Cet objet étant un objet composite, il utilisera le composant monôme et fournira lui-même les informations relatives au monôme. Donc seul le polynôme est un objet animable. Le serveur quant à lui ne change pas quelque soit l'objet animable auquel il se réfère. La seule condition est actuellement que le serveur soit paramétré par UN objet animable. Nous avons donc autant de serveurs que de types animables. C'est une solution un peu limitative que nous comptons modifier par la suite pour utiliser un véritable serveur. En ce qui concerne la troisième entité de notre modèle, deux types d'animations sur les polynômes sont prévus. Le premier est le suivi de l'utilisation des monômes d'un polynôme existant (voir écran précédent). Lors de l'exécution d'une addition, nous verrons ainsi se mettre en valeur les monômes utilisés. La seconde animation permet de voir se construire un polynôme, c'est à dire que l'on voit les monômes “naître” les uns après les autres dans le polynôme résultat. Par exemple, voici deux images successives de l'animation montrant un polynôme en construction :

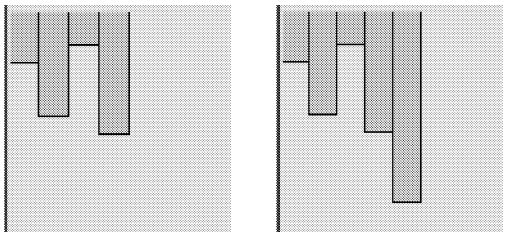


Figure 4.2: Polynôme en construction

Le schéma de fonctionnement ainsi décrit ressemble à cela :

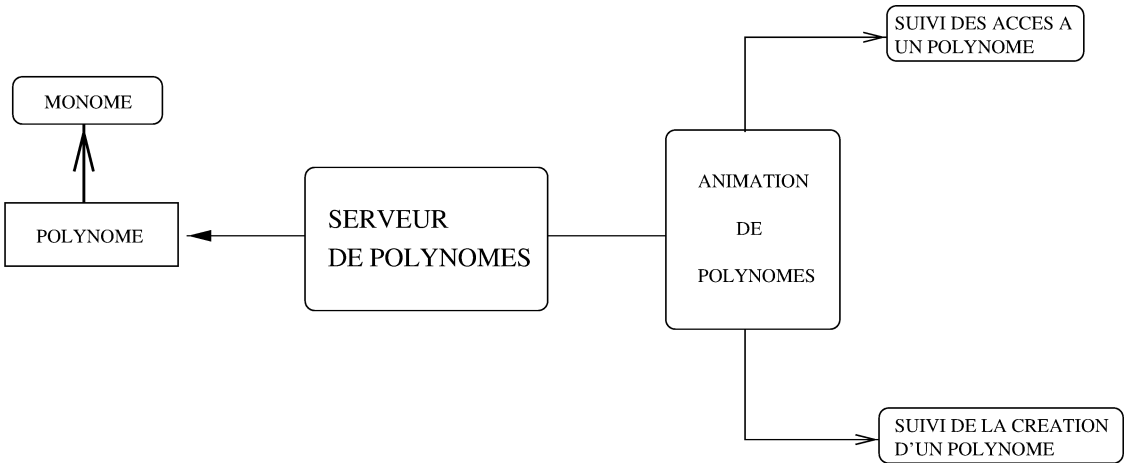
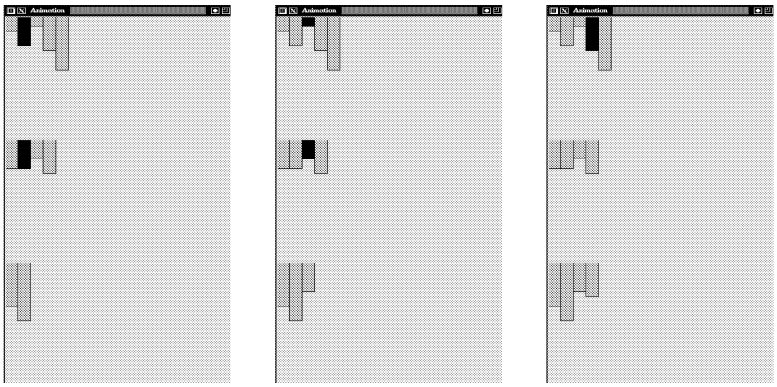


Figure 4.2: Schéma modifié

Pour réaliser cette animation, nous avons instrumenté le code, c'est une solution semblable à celle qu'ont choisie la plupart des systèmes d'animation comme Zeus. La simplicité

n'est pas au rendez-vous, car les modifications sont relativement importantes. Nous souhaitons éviter au maximum de telles modifications de code. C'est l'objet du troisième exemple (portant sur les bases de Gröbner). Voici le code instrumenté de l'addition de deux polynômes :

(Nous ne présentons que l'algorithme instrumenté car il est en fait inutile de présenter les détails de langage de $A^\sharp$, ce n'est pas ici notre propos)



```
(p1: polynome) + (p2: polynome): polynome ==
Debut

   Creation et initialisation de l'objet qui va contenir le resultat
   Demande d'animation

   i1 <- 1
   i2 <- 1
   TQ (i1 <= nombre de monomes de p1) et (i2 <= nombre de monomes de p2)

     si exposant(p1(i1)) = exposant(p2(i2)) alors

          * Mise a jour de l'animation pour les deux polynomes.
          * Addition des deux monomes p1(i1) et p2(i2).
          * Ajout du monome resultant dans le polynome resultat.
          * Mise a jour de l'animation du polynome resultat.
          * i1:=i1+1.
          * i2:=i2+1.

     sinon si puissance(p1(i1)) > puissance(p2(i2)) alors

          * Mise a jour de l'animation du polynome 1.
          * Ajout de p1(i1) dans le polynome resultat.
          * Mise a jour de l'animation du polynome resultat.
          * i1:=i1+1.
```

```
      sinon   -- puissance(p1(i1)) < puissance(p2(i2))

           * Mise a jour de l'animation du polynome 2.
           * Ajout de p2(i2) dans le polynome resultat.
           * Mise a jour de l'animation du polynome resultat.
           * i2:=i2+1.

      }

   Fin du TQ

   Ajout des monomes restants du polynome 1 ou 2 dans
          le polynome resultat.
   Mise a jour de l'animation des polynomes concernes.

   Retour du resultat
Fin
```

Chapitre 5

Evaluation paresseuse

5.1 Evaluation paresseuse

En programmation fonctionnelle, il existe une technique très puissante de programmation. Cette technique permet de créer des programmes qu'il était impossible de réaliser auparavant. Cette technique s'appelle l'évaluation paresseuse. Les paragraphes qui suivent donnent un petit aperçu de la nature de lévaluation paresseuse. Pour plus d'informations sur l'évaluation paresseuse, il est possible de consulter [1].

L'évaluation paresseuse permet de retarder l'évaluation d'une expression jusqu'à ce qu'on ait besoin du résultat. Plus clairement, cette technique evite d'effectuer des calculs inutiles, car comme ils ne sont faits que dans le cas où un résultat est nécessaire, ils ne sont pas réalisés si aucun élément du programme n'en a besoin ! Le gain de temps et de puissance d'expression peut alors être très important. Le gain de temps est évident, puisqu'on diminue le nombre d'évaluations effectuées. En ce qui concerne la puissance d'expression, l'évaluation paresseuse ouvre de nouvelles voies.

En effet, elle offre la possibilité de créer de nouvelles structures de données comme les listes infinies. Lorsque l'on demande le 20^e élément de la liste, si le programme avait déjà calculé cette valeur, il suffit de donner cette valeur stockée, sinon on effectue le calcul de toutes les valeurs non déjà calculées jusqu'au 20^e élément. Voici pour illustrer ces propos, le code en SML du crible d'Erathostene :

```
(* LISTES INFINIES, EVALUATION PARESSEUSE *)

(* Ici definition de la structure de donnees permettant
   de gerer les listes infinies.
   ``unit -> 'a Stream'' est la signature d'une fonction
   ne prenant rien en argument et renvoyant un flux (Stream)
   d'objets de type 'a.
   'a est donc le type des valeurs du flux
 *)
datatype 'a Stream = Nil | Cons of 'a * (unit -> 'a Stream);
```

```
(* Fonctions exploitant la structure precedente pour donner les informations *)
fun head (Cons(x,_)) = x
and tail (Cons(_,xf)) = xf();

(* Exemple simple : liste infinie d'entiers *)
fun LIST_INT n = Cons(n, fn () => LIST_INT(n+1));

(* Le crible d'Erathostene *)
fun sauf_multiple (s : int Stream) =
tail(Cons (head(s),
   fn () => list_filtre(tail(s),fn x => x mod head(s) <> 0)
         )
     );

fun NB_PREM (s: int Stream) =
Cons(head(s),fn () => NB_PREM(sauf_multiple(s)));
```

Ici le "stream" ou flux de données ne conserve pas les calculs qu'il a effectué. On voit très nettement qu'il s'agit d'une liste composée de deux éléments : une donnée et une fonction permettant de calculer la suite de la liste (donc une nouvelle liste qui a comme éléments une donnée et la fonction permettant de calculer la suite de la nouvelle liste etc.).

Nous allons maintenant exposer les problèmes que pose l'animation de programmes exploitant l'évaluation paresseuse. En fait, il faut se rappeler que le but de l'animation de programmes est de "montrer" ce que fait un calcul et pas seulement de donner le résultat. Lorsque l'on travaille en utilisant l'évaluation paresseuse, non seulement il n'existe aucune garantie sur la date à laquelle une expression va être évaluée, mais il n'y a également aucune garantie sur le fait qu'elle va être évaluée. C'est un problème que se sont posé G. Lapalme et M. Latendresse dans [15] à propos du débogage de programmes fonctionnels utilisant cette technique.

Si l'on souhaite animer un programme fonctionnel de ce type, il est nécessaire de garantir que l'on ne forcera pas l'exécution d'une instruction ou l'évaluation d'une expression uniquement dans le but de "montrer" quelque chose. Si l'on ne respecte pas cette règle, l'utilisateur risque fort de voir une animation infinie sur une liste infinie !

Un autre problème réside dans le fait qu'un programme d'animation ne doit pas "scruter" les évaluations paresseuses. C'est à dire qu'un système désirant animer un programme utilisant cette technique n'a pas à se mettre en attente d'une quelconque évaluation, ceci est basé sur deux éléments fondamentaux. D'une part, rien ne nous assure que nous ne ferons pas évoluer notre système d'animation vers un système animant des algorithmes parallèles. Dans ce cas, la mise en attente du système peut se révéler très fâcheuse, voire totalement inacceptable. D'autre part, comme nous ne savons pas si une évaluation va réellement être réalisée, nous ne devons pas générer des attentes infinies. La solution que nous avons retenue consiste à "emballer" ou englober la fonction d'évaluation (celle qui permet dans notre précédent exemple de calculer la suite de la liste). Cette fonction emballée deviendrait bien sûr une nouvelle fonction équivalente à la précédente. La différence est que cette nouvelle

fonction va contenir des instructions supplémentaires pour animer cette exécution. Cette solution nous garantit de ne pas bloquer le système en attente d'un résultat puisqu'aucune évaluation particulière ne sera exécutée si la fonction d'évaluation n'est pas elle-même exécutée. De plus, par ce moyen, il n'y a aucune chance de forcer une évaluation pour montrer une animation, car la fonction englobante va se substituer à la fonction d'évaluation et donc ne sera exécutée que lorsque le programme aura besoin d'exécuter la fonction emballée.

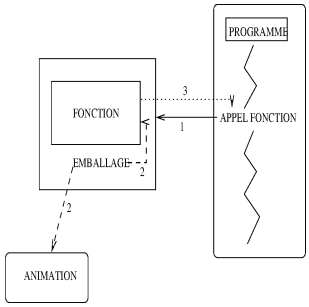


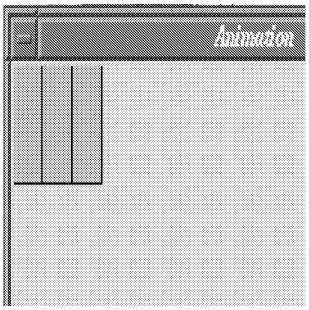
Figure 5.1: Emballage de la fonction d'évaluation

5.2 Le développement en séries

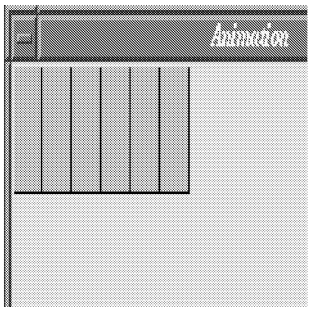
Cet exemple d'application va nous permettre d'illustrer pleinement à la fois le fonctionnement de l'évaluation paresseuse et le moyen utilisé pour emballer la fonction d'évaluation. Le développement en séries de puissance va se baser sur le type monôme utilisé dans l'exemple précédent. En ce qui concerne l'évaluation paresseuse, nous allons utiliser un type `Stream` que nous avons développé précédemment.

L'intérêt de l'utilisation de cet exemple réside dans le fait que le développement en série est une application mathématique simple à comprendre et rapide à développer mais surtout son implémentation pour être réelle se doit de faire un large usage de l'évaluation paresseuse. Son animation aura donc deux buts pédagogiques comme nous le précisions dans la présentation du sujet : Animer pour faire comprendre l'évaluation paresseuse et Animer pour faire comprendre le développement en séries de puissance.

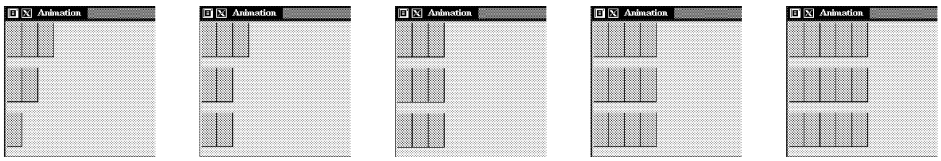
Le but que nous nous sommes donnés pour réaliser cette animation est de donner à l'utilisateur un bon aperçu de la manière dont évolue un programme utilisant l'évaluation paresseuse. Ainsi, nous avons souhaité montrer l'évolution de la structure de données `Stream` au fur et à mesure des demandes du programme. A chaque fois que le programme lance une consultation du coefficient d'un élément d'un développement, on désire voir la progression du calcul de la suite de la série si l'élément considéré n'a pas encore été calculé. Par exemple, après avoir consulté le coefficient du 3^e monôme du développement, on obtient un écran comme celui-ci :



Si ensuite on demande le coefficient du 2^e monôme, rien ne se passe car cet élément a déjà été stocké. Mais, si l'on demande le coefficient du 6^e monôme, on voit une progression de la série jusqu'à :



Ceci illustre bien ce que nous désirons réaliser, cependant, il ne s'agit que d'un exemple simple; en fait, nous souhaitons animer des opérations arithmétiques évaluées sur ces flux. En nous servant des "briques" de base que nous venons de décrire, nous voudrions effectuer l'ajout de deux développement en série. Lorsque nous demandons le cinquième coefficient du monôme du développement résultat de l'addition, nous voudrions voir évoluer les calculs des deux développement nécessaires au calcul du troisième. Voici le résultat de cette réalisation :



La première image montre le moment de l'algorithme où le programme a calculé trois éléments pour le premier développement, deux pour le deuxième et un pour le troisième qui est le résultat de l'addition des deux premiers. Les images suivantes montrent l'évolution des développements après une demande de consultation du 5^e monôme du troisième développement.

On peut voir nettement que pour donner le cinquième coefficient du dernier développement, le programme a dû également calculer les éléments des deux premiers développements.

Cependant, cette réalisation n'avait d'intérêt que s'il était possible de l'intégrer à notre système en respectant le schéma de fonctionnement que nous avons défini pour notre étude sur les objets composites. C'est ce que nous avons fait en définissant le type "englobant" le type `Stream` comme une structure de données '`animable`' et en créant un objet graphique d'animation adapté à l'évaluation paresseuse. Bien sûr, aucune modification sur le code du serveur n'a été nécessaire. Voici le schéma de notre système d'animation ainsi complété :

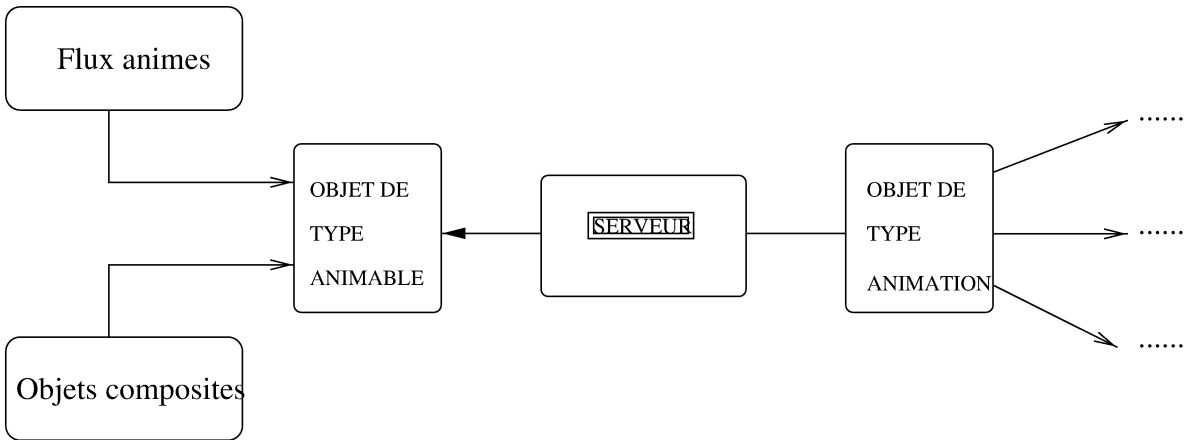


Figure 5.2: Schéma de fonctionnement complété

Le code suivant est celui de la fonction de trace, c'est celle qui englobe ou qui emballe la fonction d'évaluation :

Note : `(a: B): B +->E(a)` *en* $A^\sharp$ *représente* $a \mapsto E(a)$ *et correspond à* **lambda(a: A): B.E(a) dans un λ-calcul typé**

```
follow(ws: WrappedStream, f: () -> Stream(R))(): Stream(R) == {

    -- La fonction follow prend un flux emballe et une
    -- fonction d'evaluation et rend une fonction d'evaluation
    -- de signature equivalente : ceci permet d'ajouter du code
    -- sans que l'evaluation du flux ne soit perturbee.

    -- Note : La fonction est currifiee

    touched(ws,nil);
          -- touched indique un changement a l'animation
          -- ceci aura pour effet la mise a jour graphique

    f(); -- appel de la fonction d'evaluation
}
```

Chapitre 6

Opérations de haut-niveau

L'exemple des bases de Gröbner va nous permettre d'appliquer notre système d'animation sur un cas réel et de taille intéressante. D'autre part, il nous permettra d'illustrer notre travail sur les modifications minimales de code et sur les opérations de haut niveau. Ce travail est actuellement en cours de réalisation.

6.1 Opérations de haut-niveau

Comme nous l'avons vu, $A^\sharp$ manipule des fonctions et des types de la même manière que des entiers ou des flottants, tous ces éléments du langage sont considérés comme des valeurs. Les opérations que nous appelons "opérations de haut-niveau" sont des fonctions travaillant sur des types, sur des fonctions travaillant sur des types, etc. Le but est d'utiliser un constructeur prenant une fonction de création de type en paramètre et rendant une fonction de création du même type, mais animé. Ce sont des fonctions qui vont nous permettre de créer de nouvelles structures de données. Ceci est un aspect très intéressant de la programmation fonctionnelle puisque les opérations de haut-niveau nous offrent d'intéressantes possibilités de modularité et de puissance d'expression.

Dans cette étude, notre but est de pouvoir écrire des constructeurs prenant des structures de données en paramètre et créant des types animés. Il s'agit, dans une optique fonctionnelle, d'écrire des fonctions prenant un type à animer en paramètre et rendant un nouveau type basé sur le type paramètre mais dont les fonctions le manipulant sont instrumentées pour l'animation. En emballant les fonctions manipulant le type à animer, il sera possible de créer un nouveau type équivalent au type de base (dans le sens où les fonctions offertes de manipulation de ce type sont les mêmes) mais permettant en plus de visualiser son évolution.

Un des avantages de cette technique est, comme nous allons le voir, la possibilité d'animer des fonctions en effectuant des modifications minimes de code.

6.2 Modifications minimales du code à animer

Même si l'étude des opérations de haut-niveau a son propre intérêt, nous l'utilisons ici plutôt comme un moyen que comme un but. En fait, notre finalité est d'animer un

programme par une modification minimale de code. Comme nous l'avons vu précédemment, ces opérations un peu particulières (mais, en fait, tout à fait classiques en programmation fonctionnelle) permettent de construire des types animés à partir des types utilisés dans les programmes. Par ce moyen, il sera possible de substituer à un type utilisé dans une implémentation un nouveau type, basé sur l'ancien et totalement équivalent à ce dernier. La différence de ce nouveau type se comprend lorsque l'on utilise les opérations qu'il offre. Ces opérations rendent les mêmes résultats que les opérations non emballées mais elles font appel à des opérateurs graphiques permettant l'animation du programme.

Afin de donner une plus grande liberté, nous souhaitons mettre en place deux solutions d'instrumentation du code. La première consiste à changer uniquement le nom du ou des type(s) utilisé(s) dans le code avec une animation par défaut ce qui permettra une instrumentation minimale sans se préoccuper réellement de ce que l'on va voir. Le but ici est uniquement de voir une explication du programme. La deuxième solution ne coûte pas beaucoup plus cher. Il s'agit de rajouter une ou deux instructions dans le code afin de choisir l'animation à associer à un type emballé.

D'autre part, si l'utilisateur souhaite aller plus loin dans l'animation de son programme ou de son algorithme, il lui est possible de programmer ses propres animations Tk depuis $A^\sharp$. Nous allons voir maintenant quelles modifications sont à réaliser pour animer un code.

Les modifications à effectuer pour animer un code s'avèrent assez simples. Il s'agit, en fait, d'utiliser à la place du nom du type de base le nom du type animé (par exemple, au lieu d'utiliser **Type** on utilisera **AnimType**). Lors de l'exécution des opérations du type, ceci permettra de communiquer avec le serveur. Ce dernier doit être initialisé dès le début de l'exécution du code. Autrement dit, il est nécessaire de rajouter une instruction de lancement du serveur.

Plus particulièrement, l'animation de programmes (c'est à dire de code existant) se révèle un peu plus complexe. En effet, il n'est pas raisonnable de demander à l'utilisateur de modifier le nom des types à *tous* les endroits où il les a utilisés. Cependant, il existe plusieurs solutions simples et peu coûteuses qui permettront à l'utilisateur de ne pas modifier l'ensemble de son code. Par exemple, si l'on passe par un renommage de types au début du programme, il serait possible d'utiliser l'ancien nom (celui qui est utilisé dans le code) pour le type animé. La modification serait ici minime et permettrait d'animer le programme. Dans la suite, nous allons donner un exemple sur lequel nous avons travaillé pour utiliser des opérations de haut-niveau et pour tenter d'animer un programme de taille importante en instrumentant le moins possible.

6.3 Les bases de Gröbner

6.3.1 Présentation

Pour donner une définition simple des bases de Gröbner, on pourrait dire que ce sont des ensembles particuliers de polynômes à plusieurs variables. Informellement, un ensemble fini F de polynômes est une base de Gröbner si et seulement si un certain processus de réduction des polynômes mène toujours à un résultat unique.

Afin d'être plus précis dans cette présentation, nous avons besoin de présenter quelques notions de base. Celles-ci se retrouvent dans la plupart des algorithmes de résolution. Bien

entendu, nous retrouverons ces notions dans l'algorithme que nous allons présenter. Vous l'aurez compris, un des intérêts principaux des méthodes relatives aux bases de Gröbner réside dans leur potentiel à être transposées d'un modèle mathématique à un modèle algorithmique.

Ordre sur des monômes et Termes de tête

L'ordre sur les monômes constitue la première notion fondamentale pour ces algorithmes. Il s'agit, en fait, de déterminer un ordre total par lequel on puisse classer les monômes dans un polynôme en fonction de leurs variables et de leurs puissances. Par exemple, voici un ordre possible pour trois variables x,y,z : (ordre lexicographique)

$1 < x < y < z < x^2 < xy < y^2 < yz < z^2 < x^3 < x^2y...$

Les termes de tête (ou *headterms*) sont également des éléments importants. En fait, ils n'existent que par la donnée d'un ordre sur des monômes. Dans un polynôme, le *headterm* correspond au plus grand monôme dans le sens de l'ordre. Par exemple, prenons le polynôme :

$-x^2 + 5x^2y + 1$

Le *headterm* est $5x^2y$.

Réduction d'un polynôme

Dans la définition, nous parlions d'un processus de réduction de polynômes. Soient f et g deux polynômes, On réduit f *modulo le polynôme* g de la manière suivante. Si un terme t dans f est un multiple du headterm de g, alors on ajoute à f un multiple de g construit de manière à supprimer t. Formellement,

$\forall t \in termes(f),\ si\ m.t = hd(g)\ alors\ f \longrightarrow_g (f - f.m)$

On itère ces opérations sur le nouveau polynôme. La réduction se poursuit jusqu'à stabilité, c'est à dire que l'on ne peut plus trouver de multiple pour les termes du polynôme résultat.

La réduction *modulo un ensemble* G de polynômes correspond à réduire un polynôme par tous les polynômes G_i de l'ensemble. Il s'agit, en fait, d'itérer sur cet ensemble afin d'obtenir une stabilité.

$G_1 = x^2yz - xy$ (x^2yz est le headterm) $G_2 = xyz^2 - 2z^2$ (xyz^2 est le headterm)

$$f = x^3yz + 5xy^2z^2 - 3xyz$$
$$\longrightarrow_{G_1} x^2y + 5xy^2z^2 - 3xyz$$
$$\longrightarrow_{G_2} x^2y + 10yz^2 - 3xyz$$

Ensuite, il n'y a plus de réductions possibles.

S-Polynôme et Base de Gröbner

Dernière notion importante, le S-polynôme de deux polynômes f et g (noté $SP(f, g)$) est calculé de la manière suivante :

- Calcul du **ppcm** ou **lcm** (*least common multiple*) des headterms de f et g
- Multiplication de f et g par les monômes $\overline{f}$ et $\overline{g}$ afin que les termes de têtes de f et g deviennent identiques
- Calculer $SP(f,g) = \overline{f}.f - \overline{g}.g$

Exemple :

$$f = 5xy - 3x \qquad g = 7y^2 + 2x$$
$$lcm(xy, y^2) = xy^2$$
$$SP(f,g) = (7y).f - (5x).g = -21xy - 10x^2$$

En fait, le S-polynôme représente une forme de produit avec une réduction de deuxième ordre.

La base de Gröbner d'un ensemble de polynômes se calcule alors en prenant un idéal polynômial, en calculant un ensemble de paires de polynômes et, pour chacune de ces paires, en réduisant le S-polynôme des deux polyômes considérés.

```
Base_Grobner(PolySet) =
      BG <- PolySet
      PolyPairs <- Calcul des paires (f,g). f et g appartenant a BG.
      TQ PolyPairs non vide
           (pi,pj) <- premiere paire de PolyPairs
           Polypairs <- reste de PolyPairs
           p <- Reduire le S-Poly(pi,pj)
           si p <> 0 et p n'est pas dans BG
                PolyPairs <- pour tout q de BG, ajouter a Polypairs
                                                    les paires (q,p)
                ajouter p a BG
           is
      QT
```

6.3.2 Les applications

Pourquoi s'intéresser aux bases de Gröbner ? En fait, il existe de nombreux problèmes de la théorie des idéaux polynomiaux qui peuvent être plus facilement résolus grâce aux bases de Gröbner. De plus, un nombre important de problèmes peuvent être modélisés par des systèmes de polynômes et se traitent alors par des méthodes de géométrie algébrique. Voici un exemple d'application dans lequel l'utilisation des bases de Gröbner permettent une réelle amélioration des calculs. Il s'agit d'un problème appelé "inverse kinematics in robot programming" dans la littérature. Il est à noter que le champ d'utilisation des bases de Gröbner est assez vaste. En effet, grâce à ces bases, des problèmes comme la preuve automatique de théorêmes géométriques ou des problèmes de représentation de surfaces peuvent être résolus plus rapidement.

Inverse Kinematics in Robot Programming

Cette application pose le problème de la détermination pour un robot donné des distances aux jointures prismatiques et des angles pour les jointures de révolution qui permettent d'atteindre une position et une orientation donnée de la tête du robot. Ce problème se modélise en mathématiques par un systéme d'équations à plusieurs variables. Ces équations sont déterminés par les relations entre plusieurs caractéristiques du robot qui vont correspondre aux variables (longueur, angles, position etc.). On obtient alors un système d'une vingtaine d'équations ([9] donne le détail de ce système) pour un robot simplifié.

Il existe trois manières de voir le problème. Une version simple (Real Time) permet d'obtenir des valeurs numériques par la donnée de plusieurs valeurs au départ puis le calcul des autres par le système. Les deux autres versions tentent d'abstraire les calculs en éliminant les valeurs numériques et en essayant d'obtenir comme solutions des expressions symboliques. Sur ces deux dernières versions, il est alors possible d'utiliser les bases de Gröbner pour résoudre le système.

Après avoir défini une relation d'ordre, il suffit de prendre en entrée de l'algorithme le système d'équations modélisant le problème. La base de Gröbner calculée a une structure intéressante puisqu'elle se trouve être triangularisée, ceci permet une résolution du système beaucoup plus simple.

6.4 Notre étude actuelle

Comme nous l'avons dit en introduction de ce chapitre, l'étude sur les opérations de haut-niveau et son application sur les bases de Gröbner sont en cours. Nous souhaitons réaliser une animation sur la résolution de ces bases en utilisant une instrumentation minimale du code. La finalité de ce travail est de pouvoir présenter différentes informations sur l'exécution de l'algorithme.

Nous pensons pouvoir animer, par des opérations de haut-niveau, l'algorithme présenté plus haut. Il peut être, en effet, intéressant de visualiser l'évolution de la base de polynômes et de voir quels polynômes sont utilisés. Une animation sur un algorithme résolvant les bases de Gröbner présente deux intérêts. Le premier est clairement pédagogique. Visualiser l'évolution dans le temps de l'ensemble de polynômes et voir comment sont traités les couples de polynômes permet d'apprendre et de comprendre beaucoup plus rapidement l'essence même de l'algorithme. Le deuxième intérêt de cette animation s'avère purement scientifique. En visualisant de quelle manière sont utilisés les polynômes et en consultant la "température" d'un polynôme (i.e. depuis combien de temps un polynôme n'a pas été touché), on pourrait concevoir de nouveaux algorithmes de résolution par de nouvelles heuristiques. On pourrait alors animer ces algorithmes. Ces animations permettraient de trouver de nouvelles heuristiques etc.

De plus, nous avons commencé à travailler sur l'implémentation des constructeurs de types appliqués aux monômes et aux polynômes. A partir d'un algorithme existant de résolution des bases de Gröbner écrit en $A^\sharp$, nous avons pour but d'utiliser indifféremment (modulo les petites modifications que nous avons exposées plus haut) un type `Polynome` ou un type `AnimPolynome`. En fait, nous avons prévu, pour l'instant, quatre configurations possibles à partir des types polynômes et monômes. Le type `Polynome` est paramétré par

le type `Monome`. Ces quatre configurations sont les suivantes :

Polynome(Monome)	Pas d'animation
Polynome(Anim(Monome))	Seuls les monômes sont animés
(Anim(Polynome))(Monome)	Seuls les polynômes sont animés
(Anim(Polynome))(Anim(Monome))	Les polynômes *et* les monômes sont animés.

Notre but serait de réaliser des fonctions sur des types. On appellerait ces fonctions des constructeurs. Ici, `Polynome(Monome)` est un type mais `Polynome` seul dénote la fonction qui prend en paramètre un `Monome` et rend le type `Polynome(Monome)`. Rien ne nous empêche alors d'instrumenter la fonction `Polynome` afin de réaliser des animations sur le type qu'elle donne comme résultat.

Nous comptons présenter les différentes animations que nous aurons conçues sur les bases de Gröbner au SNAP'96 (Symbolic-Numeric Algebra for Polynomials) qui aura lieu à Sophia Antipolis aux 15-17 juillet 1996.

Conclusion

Animation d'algorithmes

Comme nous l'avons vu, la plupart des systèmes d'animation d'algorithmes existants ne se manipulent pas simplement. Des systèmes comme Zeus ou Tango sont coûteux en mise en oeuvre. Aladdin se base sur la profusion d'outils évolués pour aider à la conception de tels systèmes. Enfin, Agat apporte une solution intéressante à l'animation simple de programmes.

Cependant, tous ces systèmes s'appuient sur des langages impératifs ou orienté-objets. Aucun de ces systèmes ne s'est penché sur l'animation de programmes fonctionnels. Pourtant, animer des programmes fonctionnels peut apporter des informations dans de nombreux domaines scientifiques.

Le système que nous avons conçu utilise un langage appelé $A^\sharp$ et traite actuellement trois des questions fondamentales posées par l'animation d'algorithmes et la programmation fonctionnelle. L'animation des objets composites, l'évaluation paresseuse et les opérations de haut-niveau sur des objets.

La partie graphique, réalisée en Tk, nécessite d'être améliorée. En effet, elle est actuellement minimale et ne gère pas encore l'interaction avec les utilisateurs (ce qui peut être un aspect important de l'animation de programmes). Les exemples que nous avons choisis sont caractéristiques des applications possibles de l'animation de programmes. Ainsi, l'animation sur les développements infinis permet de comprendre deux concepts simultanément : la façon dont évolue un développement infini dans le temps et l'idée principale de l'évaluation paresseuse. L'animation sur les objets composites peut aider à la conception de programmes utilisant des structures de données complexes. Enfin, l'exemple de la résolution des bases de Gröbner constitue sans aucun doute l'application la plus utile. En effet, les bases de Gröbner sont l'objet de recherches poussées dans un nombre important de laboratoires et une animation d'algorithmes existants leur permettrait d'en visualiser les aspects importants afin de trouver des raffinements sur des méthodes et de développer de nouveaux algorithmes plus optimaux.

En conclusion, il est évident que les systèmes d'animation d'algorithmes constitueront, dans les années à venir, un outil incontournable en informatique au même titre que les debuggers. Un nouveau secteur pourrait même se développer, on pourrait le nommer Aide à la Compréhension de Programmes Assistée par Ordinateur.

Plan sur l'avenir

Pour finir, nous avons souhaité exposer les travaux que nous comptons réaliser dans les trois prochains mois. Le système d'animation que nous avons réalisé n'est pas complet. C'est pourquoi nous souhaitons l'enrichir. Tout d'abord, nous projetons de rajouter une interaction forte avec l'utilisateur afin que celui-ci puisse contrôler complètement le déroulement de l'animation. Ceci implique de poursuivre et de terminer l'intégration de Tk dans $A^\sharp$. De plus, nous voudrions construire d'autres animations à mettre à disposition des utilisateurs afin d'offrir un nombre suffisant d'éléments prédéfinis. Enfin, nous étudierons les possibilités d'application du système à d'autres langages fonctionnels tels que SML.

Bibliographie

[1] H. Abelson and G. Sussman (with J. Sussman), *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge Mass, 1985.

[2] O. Arsac, S. Dalmas, M. Gaetano, *Algorithm Animation with AGAT*, preprend, 1993, http://zenon.inria.fr/safir/SAM/Agat/agat.html

[3] M. Brown, *Exploring algorithms using BALSA 2*, IEEE Computer, 18(8):29-35, 1988

[4] M. Brown, *Zeus: A System for Algorithm Animation and Multi-View Editing*, Digital SRC Research Report 75, 1992

[5] M.Brown, J. Hershberger, *Color and Sound in Algorithm Animation*, Digital SRC Research Report 76a, 1991

[6] M. Brown, M. Najork, *Algorithm Animation Using 3D Interactive Graphics*, Digital SRC Research Report 110a, 1993

[7] M. Brown, R. Sedgewick, *Techniques for Algorithm Animation*, IEEE Software 2(1):28-39, 1985

[8] B. Buchberger, *A criterion for detecting unnecessary reductions in the construction of Gröbner-bases*, Proc. Eurosam 79, Edward W. Ng (rédacteur), LNCS no 72, Springer-Verlag, Berlin, 1979.

[9] B. Buchberger, *Applications of Gröbner bases in non-linear computational geometry*, Mathematical Aspects of Scientific Software p. 59-87, Springer-Verlag, 1987

[10] H.B. Curry and R. Feys, *Combinatory Logic*, North Holland, Amsterdam, 1958

[11] M. Delest, N. Rouillon, JM Fédou, *An Overview of CalICo*, preprend, 1995, http://www.labri.u-bordeaux.fr/ maylis/publications.html

[12] E. Helttula, A. Hyrskykari, KJ Räihä, *Graphical Specification of Algorithm Animations with ALADDIN*. In B. Shriver, editor, Proceedings of the Twenty-Second Annual Hawaii International Conference on System Science, volume 2, p. 892-901, IEEE compuct. Soc., Press., 1989

[13] D. Huang, *Sort Algorithm Demonstrations*, http://www.cc.gatech.edu/people/home/dhuang/myjava.htn

[14] M. Krishnamoorthy, R. Swaminathan, *Program Tools for Algorithm Animation*, Software - Practice and Experience, 19(6):505-513, 1989

[15] G. Lapalme, M. Latendresse, *A Debugging Environment for Lazy Functional Languages*, Lisp and Symbolic Computation, 1992

[16] J. Ousterhout, *Draft - Tcl and the Tk Toolkit*, 1993, Addison-Wesley, ISBN 0-201-63337-X

[17] J. O'Donnell, C. Hall, *Debugging in Applicative Languages*, Lisp and Symbolic Computation, 1988

[18] J. Stasko, *Tango: a framework and system for algorithm animation*, IEEE Computer, 23(9):27-39, 1990.

[19] S. Watt, P.Broadbery, S. Dooley, P. Iglio, S. Morrison, J. Steinbach, R. Sutor, *A# User's Guide*, 1994, The Numerical Algorithms Group.

[20] S. Watt, P.Broadbery, S. Dooley, P. Iglio, S. Morrison, J. Steinbach, R. Sutor, *A First Report on A# compiler*, ISSAC 94 Proceedings on Symbolic and Algebraic Computation, 1994, ACM.

[21] B. Welch, *Draft - Practical Programming in Tcl and Tk*, 1995, Prentice Hall, ISBN 0-13-182007-9