Université Henri Poincaré – Nancy I
Faculté des Sciences

CRIN-CNRS
INRIA-Lorraine

[UFR STMIA]
École Doctorale IAE + M
Département de Formation Doctorale en Informatique

ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes

THÈSE

présentée et soutenue publiquement le 31 octobre 1994

pour l'obtention du

Doctorat de l'Université Henri Poincaré – Nancy I

(Spécialité Informatique)

par

Marian Vittek

Composition du jury

Président :	M. Sintzoff
Rapporteurs :	H. Comon
	J. Meseguer
Examinateurs :	C. Kirchner
	H. Kirchner
	D. Méry

Je remercie Claude Kirchner d'avoir accepté de diriger mes recherches pendant ces trois dernières années. Sans ses qualités humaines, sa disponibilité et son soutien permanent autant moral que professionnel, cette thèse n'aurait probablement jamais été achevée.

Je remercie également les membres du jury : Michel Sintzoff, professeur à l'Université Catholique de Louvain, qui a accepté d'être le président du jury; les rapporteurs : Hubert Comon, directeur de recherche au LRI d'Orsay, pour ses rapports et commentaires pertinents provoquant ensuite des améliorations importantes du document; José Meseguer, senior computer scientist à SRI International à Menlo Park, qui m'a aidé en s'intéressant à mon travail et qui a accepté de faire un rapport sans avoir la satisfaction de participer au jury; Dominique Méry, professeur à l'Université de Nancy I, pour son attitude sympathique en acceptant d'être le rapporteur interne au dernier moment. Je remercie Hélène Kirchner, chargé de recherche au CNRS, pour son soutien humain ainsi que pour sa contribution pratique à ce travail.

Je remercie les membres de l'équipe PROTHEO et du laboratoire INRIA-Lorraine & CRIN de m'avoir accueilli et de m'avoir offert d'excellentes conditions de travail. En particulier je remercie Eric Domenjoud pour son aide concernant les aspects pratiques d'implantation et de rédaction de la thèse.

Je remercie Philippe de Groote pour les discussions fructueuses sur des points fixes que nous avons mené pendant la rédaction de la thèse. Je suis reconnaissant à tous ceux qui ont accepté de lire et de corriger le manuscrit, merci en particulier à Daniel Briaud, Jean-Michel Hufflen, Hélène Touzet, Francois Leclerc et Laurent Vigneron de m'avoir expliqué avec patience les nuances de la langue française.

Je dois beaucoup aussi à Miki Herman qui m'a aidé dès le début à me repérer dans l'équipe et dans le laboratoire.

Je remercie également les étudiants de DEA qui ont eu le courage d'utiliser ELAN *dans des conditions pratiques et de m'avoir fourni leurs très importants commentaires.*

Table des matières

1
Préface

Les langages de programmation font partie de la vie quotidienne de l'informatique d'aujourd'hui. Ils offrent un moyen de formaliser et d'implanter des algorithmes. En conséquence, une partie de l'informatique théorique s'occupe de preuves de propriétés de programmes écrits dans différents langages de programmation, ce qui exige une bonne connaissance des propriétés des langages de programmation eux-mêmes.

Dans ce contexte, parmi tous les langages de programmation, un sous-ensemble intéressant consiste dans les *langages de programmation logique* avec ou sans *contraintes*. Au sens général du terme, ils couvrent un large spectre de langages dont la sémantique est basée sur une certaine logique. L'utilisation de contraintes permet d'augmenter l'expressivité et l'efficacité d'exécution de ces langages. Mais d'un autre côté, de nouveaux problèmes naissent de cette combinaison de différentes logiques et contraintes. Pour pouvoir expérimenter ces interactions il est donc intéressant de disposer d'un logiciel permettant de spécifier et exécuter, d'une façon formelle et uniforme, différents langages de programmation avec contraintes.

Mettre au point un tel logiciel est l'objectif de cette thèse.

Dans ce contexte, nous devons mentionner des systèmes existants appelés *cadres logiques* et permettant de spécifier des raisonnements dans différentes logiques. Néanmoins, ces systèmes en général ne supportent pas les logiques avec contraintes et de plus ils se concentrent plutôt sur la spécification de la relation de prouvabilité d'une logique que sur la spécification d'un algorithme précis de résolution représentant la sémantique opérationnelle d'un langage de programmation. Les cadres logiques existants reflètent donc peu l'utilisation de logiques pour la sémantique de langages de programmation.

L'idée de développer un environnement permettant d'expérimenter des combinaisons de différentes procedures de déduction et différents résolveurs de contraintes s'est concrétisée dans le projet Elios du GDR programmation du CNRS. Dans le cadre de la construction des langages de spécification exécutables intégrés à leur environnement de preuve afin de permettre la vérification des spécifications décrites dans ces langages, le projet Elios a eu pour objectif le développement d'un langage de spécification pour décrire des résolveurs des contraintes ainsi que plusieurs logiques intéressantes.

Le projet Elios a été développé en parallèle au CRIN & INRIA-Lorraine à Nancy et au LRI à Orsay. Les définitions du langage de spécification ont abouti à la définition de deux formalismes différents: ECOLOG [Hab93, Habar] à Orsay et ELAN [KKV93b, KKV95] à Nancy. Ce dernier fait l'objet de cette thèse. Nous présentons une comparaison plus détaillée d'ELAN et d'ECOLOG dans le chapitre 7 consacré aux travaux reliés.

Dans nos travaux nous sommes partis du fait que la sémantique des langages de programmation logique dans le sens général du terme, la construction de preuves de théorèmes et la résolution de contraintes peuvent être décrites d'une façon uniforme en donnant la syntaxe des formules, l'ensemble des axiomes et l'ensemble des règles de déduction. Les trois processus peuvent être vus comme l'instance du même schéma d'application des règles de réécriture (de déduction) sur des formules, suivant une stratégie permettant qu'une forme spéciale soit obtenue. Pour réaliser ce point de vue, nous utilisons la logique de réécriture comme une formalisation du processus général de réécriture. La logique de réécriture a été introduite par José Meseguer dans [Mes89, Mes92], où il est montré qu'elle offre les caractéristiques générales des cadres logiques. En effet, il est possible de représenter différentes logiques comme des théories de la logique de réécriture. Cette représentation est effectuée par la transformation d'une théorie d'une logique objet dans une théorie de la logique de réécriture, telle que les preuves des formules dans les deux formalismes se correspondent.

D'un autre côté, il est aussi naturel de décrire des résolveurs de contraintes en termes de règles de réécriture, c'est-à-dire comme des théories de la logique de réécriture. L'application des règles de réécriture calcule dans ce cas une forme résolue des contraintes. Plusieurs résolveurs de contraintes étaient déjà décrits dans ce cadre, par exemple la résolution d'équations [Kir85, JK91], de diséquations [Com91], d'inéquations [Com90b, NR92] sur les termes comparés avec un ordre de simplification, de contraintes d'appartenance [Com90a].

La logique de réécriture est une logique du premier ordre et son calcul de preuves est basé sur la réécriture, ce qui donne la possibilité d'une implantation efficace et directement opérationnelle de notre langage; de plus nous pouvons profiter des travaux faits sur l'implantation des systèmes de réécriture.

Pour toutes ces raisons, nous avons choisi cette logique comme base du mécanisme d'évaluation de notre langage. Dans nos travaux, nous l'avons enrichie d'une notion de *stratégie* et obtenu ainsi un *système de calcul*. Les stratégies d'un système de calcul permettent de raffiner les descriptions d'algorithmes et en même temps, elles restreignent l'ensemble des formules prises en compte, ce qui réduit considérablement la complexité de la déduction de la logique de réécriture.

Pour pouvoir mieux spécifier différents langages de programmation logique, ELAN permet, en plus de l'implantation des systèmes de calcul, de spécifier la transformation d'une théorie d'une logique donné dans un système de calcul.

Le sujet de cette thèse est donc la définition formelle des systèmes de calcul, la définition du langage ELAN et la réalisation de son implantation. Le document est organisé de la façon suivante: Nous commençons par un survol du cadre général de la thèse, où nous présentons d'une manière informelle les notions principales de la logique de réécriture, des systèmes de calcul et des codages de logiques et de résolveurs de contraintes. Nous y présentons brièvement aussi le langage ELAN.

Dans le chapitre suivant, nous définissons formellement les systèmes de calcul en commençant par la définition de la logique de réécriture et en précisant la définition des stratégies. Nous donnons également une sémantique exacte au langage des stratégies.

La thèse continue par le manuel de référence d'ELAN. Dans ce manuel, nous mentionnons la liaison entre un programme et un système de calcul et nous définissons la syntaxe et la sémantique opérationnelle du langage. Les constructions définies sont illustrées sur les nombreux exemples.

Nous présentons ensuite la bibliothèque des modules standards. Il s'agit d'un ensemble de modules écrits en ELAN et qui sont d'utilité générale. Parmi tous ces modules, une place spéciale est occupée par les modules facilitant le codage de logiques du premier ordre. Ils codent les notions de base comme les termes ou les atomes et décrivent quelques algorithmes de résolution de contraintes, tels que l'unification syntaxique ou commutative.

Quelques exemples de langages de programmation logique implantés en ELAN sont présentés dans le 6-ème chapitre de la thèse.

Le chapitre 7 est consacré à une comparaison de notre formalisme à quelques systèmes similaires. En particulier, nous nous comparons avec d'autres cadres logiques, avec le langage Maude basé lui-aussi sur la logique de réécriture et avec ECOLOG dont les objectifs originels étaient identiques à ceux d'ELAN.

Nous concluons enfin après avoir donné une brève description de l'implantation du langage ELAN.

2
Introduction

Dans ce chapitre, nous introduisons les concepts généraux de notre travail et nous faisons un survol des résultats de la thèse. Toutes les notions introduites ici au niveau informel (souvent simplifié) seront ensuite développées en détail dans la suite de la thèse.

Le chapitre est organisé de la façon suivante: au début nous présentons des motivations qui nous ont mené au développement d'ELAN, puis nous présentons brièvement la logique de réécriture et nous introduisons au niveau informel la notion de système de calcul. Le chapitre continue ensuite par deux exemples illustrant comment nous pouvons exprimer la résolution de contraintes et la sémantique opérationnelle d'un langage de programmation logique par des systèmes de calcul. Nous finissons ce chapitre par une brève description de l'environnement ELAN permettant de définir et exécuter des systèmes de calcul.

2.1 Motivations

Comme nous l'avons déjà mentionné dans la préface et à la suite du projet Elios, nous nous intéressons à l'implantation de langages de programmation logique et de résolveurs de contraintes. Nous prenons l'expression *langage de programmation logique* dans le sens général du terme, comme un langage dont la sémantique est basée sur une certaine logique: par exemple, PROLOG basé sur la logique du premier ordre des clauses de Horn; un langage comme OBJ basé sur la logique équationnelle; un langage fonctionnel basé sur la logique d'ordre supérieur.

Nous voulons donc développer un logiciel (nous l'appelons ELAN, de *Elios LANguage*) qui nous permet de spécifier d'une façon commode des mécanismes d'évaluation de tous ces langages et quelques résolveurs de contraintes afin de pouvoir les combiner. De plus, nous voulons que notre logiciel permette de spécifier facilement la syntaxe de formules et requêtes de ces langages.

L'aspect combinaison d'une syntaxe et d'un mécanisme d'évaluation est important dans le prototype que nous développons. Cet aspect nous paraît fondamental aussi pour un grand nombre de logiciels déjà existants. Pour mieux illustrer ce que nous entendons par cette combinaison, nous mentionnons quelques logiciels connus qui peuvent être placés dans cette perspective. Nous pouvons commencer par YACC (*Yet Another Compiler Compiler*) qui utilise les grammaires LALR(1) pour spécifier la syntaxe et le langage C pour l'évaluation. Nous pouvons considérer aussi par exemple le langage fonctionnel CAML permettant d'utiliser les grammaires LALR(1) afin de spécifier la syntaxe des entrées. Le logiciel Centaur basé sur la *sémantique naturelle*. Les langages OBJ et ASF+SDF (*Abstract Specification Formalism + Syntaxe Definition Formalism*) qui combinent définitions de syntaxe avec spécifications algébriques. D'ailleurs, les programmes de ces derniers sont proches de programmes d'ELAN. Ces deux langages nous ont en effet beaucoup influencé pendant le développement d'ELAN.

Tous ces logiciels sont, ou peuvent être, utilisés pour l'implantation de résolveurs de contraintes et de langages de programmation. Malgré cela, nous avons décidé de développer un nouveau logiciel basé sur un mécanisme d'évaluation différent. La raison majeure pour cette décision était l'observation qu'un grand nombre d'algorithmes et de procédures de déduction sont exprimés d'une façon très naturelle à l'aide de règles de transformations. Ces règles sont souvent appliquées d'une façon non-déterministe et nécessitent une exploration des dérivations possibles. Par exemple les procédures d'unification commutative, d'unification d'ordre supérieur, ainsi que les règles de déduction dans différentes logiques et de procédures de résolutions, telles que la SLD-résolution ou la surréduction tombent dans cette catégorie. Nous avons donc voulu mettre au point un logiciel permettant d'exécuter directement des règles de réécritures appliquées d'une manière non-déterministe. La facilité avec laquelle nous pouvons ainsi spécifier différents algorithmes non triviaux nous paraît être une motivation très forte.

En ELAN nous voulons donc présenter un mécanisme d'évaluation où nous utilisons les règles de réécriture avec une sémantique donnée par la logique de réécriture [Mes92]. Cela nous permet de spécifier d'une façons très naturelle des procédures non-déterministes, telles que par exemple l'unification modulo différentes théories, la SLD-résolution ou la surréduction. En principe ces règles de réécriture décrivent un pas de transition d'un terme sur un autre et l'interpréteur d'un programme explore ensuite toutes les transitions possibles. Pour raffiner cette recherche tout en gardant l'utilisation non-déterministe des règles de réécriture, nous avons introduit dans les programmes d'ELAN la notion de *stratégie*. Un tel programme définit alors une syntaxe, un ensemble de règles de réécriture et un ensemble de *stratégies*. Les stratégies restreignent l'espace de recherche en décrivant la forme des dérivations "intéressantes". Elles permettent aussi de combiner les non-déterminismes *don't know* et *don't care*, c'est-à-dire les non-déterminismes où, à un certain moment de l'évaluation, soit plusieures sous-branches doivent être alternativement explorées, soit une seule sous-branche est choisie pour continuer l'évaluation tandis que les autres sont oubliées.

Les stratégies jouent une rôle crucial pour l'utilisation pratique de la logique de réécriture. Pour cela nous leur donnons une attention particulière dans cette thèse. Nous utilisons même un nouveau terme, *système de calcul*, pour désigner un ensemble de règles de réécriture avec des stratégies. De plus nous montrons que la notion de *stratégie*, comme nous l'avons introduit, est compatible avec les idées de la logique de réécriture. Nous pouvons même exprimer l'utilisation de ces stratégies dans le cadre de la logique de réécriture elle-même.

Pour résumer, nous allons nous intéresser à la spécification de langages de programmation logiques et à la résolution de contraintes. Les procédures de déduction et résolution étant souvent non-déterministes, nous avons choisi d'implanter un nouveau mécanisme d'évaluation basé sur la logique de réécriture avec stratégies. C'est un choix que nous avons fait et que nous essayons de justifier dans la suite de la thèse. En particulier nous montrons qu'en utilisant ce mécanisme pour implantation de procédures non-déterministes, nous sommes très proche de leur description dans les articles de recherche et nous montrons la facilité avec laquelle ces procédures peuvent être implantées.

2.2 La logique de réécriture

La logique de réécriture, introduite par José Meseguer, est une logique dont les théories sont représentées par un système de réécriture $\mathcal{R}$, écrit dans une signature de la logique de réécriture qui contient entre autres un ensemble d'équations E. Les règles de réécriture $\mathcal{R}$ agissent sur des classes d'équivalence de termes modulo cet ensemble d'équations E. En pratique cela implique que les règles de réécriture sont appliquées modulo une théorie équationnelle, telle que commutativité, associativité-commutativité, etc. Les règles de réécriture sont étiquetées, elles ont donc la forme $\ell : [l]_E \to [r]_E$ où ℓ est une étiquette et $[l]_E, [r]_E$ sont des classes d'équivalence modulo E contenant respectivement les termes l et r. Les formules dans la logique de réécriture sont de la forme $[s]_E \Rightarrow [t]_E$ dont la sémantique intuitive est que $[s]_E$ peut être réécrit en $[t]_E$. Les règles formelles de déduction de la logique de réécriture sont données dans le chapitre 3 de la thèse.

Intuitivement, chaque preuve d'une formule $[s]_E \Rightarrow [t]_E$ peut être présentée sous la forme d'une chaîne de classes d'équivalence $[s]_E = [s_0]_E \to [s_1]_E \to \cdots \to [s_n]_E = [t]_E$, où chaque $[s_{i+1}]_E$ est obtenu à partir de $[s_i]_E$ par un pas de réécriture utilisant une règle de réécriture de $\mathcal{R}$.

Imaginons par exemple un système de réécriture $\mathcal{R}$ contenant les deux règles:

$$\begin{array}{lcrcl} \ell_1 & : & element(cons(x,y)) & \to & x \\ \ell_2 & : & element(cons(x,y)) & \to & element(y) \end{array}$$

où x et y sont des variables. Ici et dans le reste du texte, nous utilisons simplement la notation t pour la classe d'équivalence $[t]_E$ si la théorie équationnelle E est vide, c'est-à-dire si elle est présentée par un ensemble vide d'axiomes. La formule

$$element(cons(a, cons(b, nil))) \Rightarrow b$$

est un théorème de théorie $\mathcal{R}$, et sa preuve est représentée par la chaîne

$$element(cons(a, cons(b, nil))) \to element(cons(b, nil)) \to b$$

où la première réduction est effectuée par l'application de la règle ℓ_2 et la deuxième par la règle ℓ_1.

La logique de réécriture comme langage de programmation

La logique de réécriture munie de son calcul de preuve offre les possibilités d'un langage de programmation. Étant donnée une théorie dans la logique de réécriture représentée par un système de réécriture $\mathcal{R}$, nous pouvons chercher toutes les solutions d'une formule

$$[t]_E \Rightarrow [x]_E$$

où t est un terme clos et x une variable. Nous sommes donc intéressés par tous les $[x]_E$ tels que $[t]_E \Rightarrow [x]_E$ est un théorème de $\mathcal{R}$.

Une procédure de recherche de l'ensemble de toutes les solutions pour cette formule doit explorer toutes les dérivations obtenues à partir de la classe $[t]_E$ en appliquant les règles de $\mathcal{R}$. L'exploration de l'arbre de toutes les dérivations peut être effectuée par une procédure d'exploration d'arbres connue, telle que en profondeur d'abord. Nous pouvons voir cette recherche des solutions comme une procédure algorithmique, ou bien comme la sémantique opérationnelle d'un langage de programmation basé sur la logique de réécriture. Un programme d'un tel langage est représenté par un ensemble des règles de réécriture et un terme de départ.

2.3 Les systèmes de calcul

Les systèmes de calcul représentent le mécanisme d'évaluation du langage ELAN, ils enrichissent le formalisme de la logique de réécriture par une notion de *stratégie*. Un *système de calcul* est une théorie de la logique de réécriture qui est enrichie par un ensemble de stratégies décrivant des calculs. Les stratégies contrôlent les applications des règles de réécriture en spécifiant des parcours dans l'arbre de toutes les dérivations et en même temps, elles décrivent quels sont les nœuds considérés comme les résultats d'un calcul. Elles sont utilisées pour décrire le déroulement des preuves qui nous intéressent d'une part, et pour restreindre l'espace de recherche de ces preuves, d'autre part.

Les systèmes de calcul ont été introduits par Claude Kirchner, Hélène Kirchner et l'auteur. Les capacités des systèmes de calculs pour exprimer la sémantique opérationnelle des langages de programmation logique et des résolveurs des contraintes sont présentées et discutées dans [KKV92, KKV93b, KKV93a, KKV95]. Nous reviendrons sur cette argumentation dans la suite de cette section.

Pour justifier la définition des systèmes de calcul, reprenons la vue de la logique de réécriture comme un langage de programmation et concentrons nous sur le problème de la recherche dans l'arbre de toutes les dérivations possibles. Il est évident qu'une augmentation du nombre de règles de réécriture et de la taille des termes provoque une explosion de l'espace des dérivations possibles. De plus, dans le cadre de la logique de réécriture, tous les termes qui se trouvent dans cet arbre de recherche sont considérés comme des résultats. Il est donc souhaitable de définir quelques conventions pour restreindre l'espace de recherche et l'ensemble des résultats. Il est pourtant très difficile de choisir de telles conventions en général pour tous les systèmes de réécriture à la fois.

Par exemple, supposons données deux règles de réécriture définissant l'opérateur element qui énumère tous les éléments d'une liste:

$$\begin{array}{lcl} \ell_1 & : & element(cons(x,y)) \rightarrow x \\ \ell_2 & : & element(cons(x,y)) \rightarrow element(y) \end{array}$$

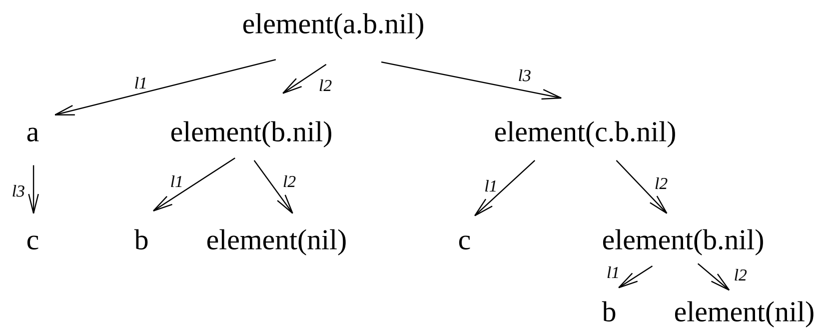
et supposons que ces deux règles sont contenues dans un système de réécriture qui contient d'autres règles de réécriture, par exemple la règle:

$$\ell_3 \ : \ a \ \rightarrow \ c$$

Pour énumérer tous les élément de la liste $cons(a, cons(b, nil))$, nous cherchons alors toutes les réponses de la requête:

$$element(cons(a, cons(b, nil))) \Rightarrow x$$

et nous obtenons l'espace de recherche dessiné dans la figure suivante.



Dans cette arbre, nous utilisons la notation infixe pour les listes, et nous écrivons donc a.b.nil à la place $cons(a, cons(b, nil))$. En logique de réécriture, chaque terme de l'arbre représente un résultat pour notre requête. Pourtant, l'ensemble des résultats intéressants (l'ensemble des éléments de la liste) est l'ensemble $\{a, b\}$.

Il est difficile de définir une stratégie générale qui sélectionne exactement cet ensemble parmi tous les résultats d'une requête. Si nous définissions, par exemple, les résultats comme les termes obtenus par les plus longues dérivations (les formes normales des termes en un certain sens), nous obtiendrions l'ensemble $\{c, b, element(nil)\}$. Nous ne pouvons pas dire non plus que l'ensemble des résultats est obtenu comme des formes normales, mais en appliquant les règles de réécriture seulement sur la tête des termes, puisque nous obtiendrions le même ensemble de résultats.

L'autre problème de la logique de réécriture est représenté par la taille de l'espace de recherche. En principe nous pouvons réduire cet espace de deux façons: premièrement en coupant les branches qui ne mènent pas à un résultat et deuxièmement en coupant les branches qui mènent à des résultats redondants, c'est-à-dire vers les résultats qui sont déjà obtenus par un autre calcul. La possibilité d'élagage de certaines branches est souvent liée aux propriétés particulières des systèmes de réécriture.

Pour ces raisons, nous avons introduit la notion de *stratégie* dans la logique de réécriture. La possibilité de définir des stratégies d'évaluation particulières pour chaque système de réécriture particulier permet de traiter les problèmes de sélection des résultats et d'élagage des branches redondantes discutés ci-dessus.

Tout au long de la thèse, nous utilisons ensuite le mot *stratégie* pour parler d'un mot du langage particulier permettant d'exprimer différentes suites d'application des règles de réécriture définissant ainsi des *calculs*. Ce langage est présenté informellement dans la section suivante et formellement dans le chapitre 3.4.

Stratégies

Nous présentons dans cette section le formalisme de stratégies utilisé dans le langage ELAN. Ces stratégies représentent un choix que nous avons fait pour exprimer des restrictions sur l'espace de recherche de la logique de réécriture en utilisant un langage d'expressions construites à partir de noms de règles par des opérateurs de concaténation, itération et choix. Les stratégies d'ELAN peuvent être récursives et elles contiennent différents opérateurs de choix (*don't care* et *don't know*) qui permettent une gestion relativement fine de l'exploration de l'arbre de recherche.

La base de départ pour la définition de ces stratégies est l'ensemble des expressions régulières. Dans la suite de la section, nous décrivons donc d'abord un langage d'expressions régulières et nous définissons l'ensemble des *calculs* correspondants. Seulement ensuite, nous introduisons le langage complet des stratégies d'ELAN avec leurs calculs appropriés.

Une expression régulière, construite sur les noms des règles de réécriture avec des opérateurs d'itération, de choix et de concaténation, exprime un ensemble de chaînes de noms de règles. Étant donnée une telle expression régulière $\mathcal{E}$, soit $Lan(\mathcal{E})$ l'ensemble des chaînes engendré par $\mathcal{E}$. Une dérivation $[t]_E = [t_0]_E \to \cdots \to [t_n]_E = [t']_E$ obtenue à partir d'une classe d'équivalence $[t]_E$ par une suite d'applications de règles de réécriture $\ell_1 \cdots \ell_n$ sur la position de tête est un *calcul*, si la chaîne des noms des règles $\ell_1 \cdots \ell_n$ appartient à l'ensemble $Lan(\mathcal{E})$. Nous disons dans ce cas que la formule $[t]_E \Rightarrow [t']_E$ *est prouvée* par ce calcul. Nous disons aussi que la classe $[t']_E$ est un *résultat du calcul* obtenu à partir de la classe $[t]_E$. Pour désigner l'ensemble des calculs à partir d'une classe $[t]_E$ correspondant à une expression $\mathcal{E}$, nous parlons de l'*application de $\mathcal{E}$ sur $[t]_E$*.

Une description formelle des calculs est donnée dans le chapitre 3 de la thèse.

> *En reprenant les règles ℓ_1, ℓ_2 et ℓ_3 de l'exemple de la partie précédente, nous pouvons définir par exemple la stratégie:*
>
> $$\ell_2^* \, \ell_1$$
>
> *disant que les* calculs *sont uniquement les dérivations contenant quelques (ou aucune) applications de la règle ℓ_2 suivies par exactement une application de la règle ℓ_1. Ainsi, nous obtenons à partir du terme:*
>
> $$element(cons(a, cons(b, nil)))$$
>
> *l'ensemble des résultats $\{a, b\}$, où le résultat a est obtenu par le calcul:*
>
> $$element(cons(a, cons(b, nil))) \to_{\ell_1} a$$

et le résultat b est obtenu par le calcul:

$$element(cons(a, cons(b, nil))) \rightarrow_{\ell_2} element(cons(b, nil)) \rightarrow_{\ell_1} b$$

car les deux calculs représentés par les chaînes "ℓ_1" et "$\ell_2\ \ell_1$" font bien partie de l'ensemble des calculs définis par l'expression $\ell_2^*\ \ell_1$.

Nous introduisons maintenant le langage de stratégies dont les expressions régulières représentent un sous-langage avec le constructeur d'itération décrit par la stratégie **iterate - enditerate**, l'opérateur de choix décrit par la stratégie **dont know choose(-)** et la concaténation exprimée simplement par une suite de stratégies.

A part les stratégies correspondant directement aux opérateurs dans les expression régulières, ELAN permet de définir des opérateurs réduisant encore plus l'espace de recherche dans l'arbre de toutes les dérivations.

L'opérateur de choix *don't care* permet d'élaguer des branches redondantes (des branches menant au même résultat) car, dans ce cas, seulement une des branches possibles (la première donnant un résultat) est choisie pour continuer l'évaluation. Nous distinguons donc deux opérateurs de choix dans les stratégies d'ELAN. L'opérateur **dont know choose** qui force l'interpréteur à explorer toutes les possibilités, et l'opérateur **dont care choose** qui arrête l'exploration des sous-arbres dès qu'il obtient un résultat. Dans ce cas, seulement ce sous-arbre est exploré.

Un autre opérateur pour construction des stratégies d'ELAN et l'itérateur **while - endwhile**. La stratégie **while** $\mathcal{E}$ **endwhile** avec une sous-stratégie $\mathcal{E}$ signifie l'itération de $\mathcal{E}$ le plus de fois possible, c'est-à-dire seulement les termes sur lesquels la sous-stratégie $\mathcal{E}$ ne peut plus être appliquée sont considérés comme des résultats. On peut voir cet opérateur d'itération comme une sorte d'opérateur de normalisation. Remarquons que l'itérateur **while - endwhile** peut donner plusieurs résultats, puisque nous pouvons avoir plusieurs dérivations finissant par une forme normale par rapport à la sous-stratégie $\mathcal{E}$.

Ceci donne finalement deux opérateurs d'itération: **iterate - enditerate** correspondant à l'étoile des expressions régulières et l'iterateur **while - endwhile** correspondant à un itérateur de normalisation.

Prenons par exemple les règles ℓ_1 et ℓ_2 de l'exemple précédent. La stratégie:

$$\textbf{dont know choose}(l_1, l_2)$$

appliquée sur le terme $element(cons(a, cons(b, nil)))$ donne l'ensemble des résultats:

$$\{a, element(cons(b, nil))\}$$

Quelques exemples de l'application de plusieurs stratégies sur le terme $element(cons(a, cons(b, nil)))$ sont montrés dans le tableau suivant.

Stratégie:		Résultat:
dont know choose(l_1, l_2)	$\rightsquigarrow$	$\{a, element(cons(b, nil))\}$
dont care choose(l_1, l_2)	$\rightsquigarrow$	$\{a\}$
dont care choose$(l_1\ l_1\ ,\ l_2\ l_2)$	$\rightsquigarrow$	$\{element(nil)\}$
iterate dont know choose(l_1, l_2) **enditerate**	$\rightsquigarrow$	$\{element(cons(a, cons(b, nil))), a, element(cons(b, nil)), b, element(nil)\}$
while dont know choose(l_1, l_2) **endwhile**	$\rightsquigarrow$	$\{a, b, element(nil)\}$

Dans les premières versions du langage ELAN les stratégies étaient construites par les expressions décrites jusque là. Ensuite, le langage des stratégies a été enrichi par des *stratégies récursives* et par la possibilité de spécifier une *application de stratégie à l'intérieur des termes*. Les deux possibilités se réalisent par l'intermédiaire d'affectations locales dans le corps des règles de réécriture. L'affectation locale dans une règle est notée par une construction *where* après la partie droite de la règle. Elle spécifie l'affectation d'une variable par les résultats d'application d'une stratégie sur un terme. Puisque la stratégie appelée dans la partie *where* peut donner plusieurs résultats, la règle de réécriture devient ainsi générique et peut être alors appliquée pour chaque valeur de l'affectation. De plus, puisqu'une stratégie peut utiliser une règle de réécriture avec une affectation locale faisant l'appel à la même stratégie, les définitions des stratégies deviennent ainsi récursives.

Prenons par exemple les règles

$$\begin{array}{lcllll} \ell_4 & : & element(cons(x,y)) & \to & x & \\ \ell_5 & : & element(cons(x,y)) & \to & z & where\ z := (extract)\ element(y) \end{array}$$

et la stratégie extract *qui est récursive puisqu'elle utilise la règle ℓ_5 faisant ainsi appel à elle-même:*

$$extract = \textbf{dont know choose}(\ell_4, \ell_5)$$

Ainsi ce système de calcul énumère tous les éléments d'une liste. Si nous appliquons la stratégie extract *sur le terme $element(cons(a, cons(b, cons(c, nil))))$, nous obtiendrons d'abord le a par une application de la règle ℓ_4 et après, puisqu'il y a deux affectations possibles (b et c) pour la variable z dans la règle ℓ_5, nous obtenons deux autres résultats b et c par l'application de la règle ℓ_5.*

La présentation intuitive des stratégies d'ELAN est étayée dans la section 3.4 de la thèse.

Nous illustrons maintenant sur deux exemples comment utiliser les systèmes de calcul d'une part pour spécifier des résolveurs de contraintes et d'autre part pour spécifier la sémantique opérationnelle de langages de programmation logique.

2.4 Un résolveur de contraintes comme système de calcul

Une description des résolveurs de contraintes basée sur des règles de réécriture rend plus faciles les preuves de complétude et de correction. Plusieurs résolveurs de contraintes sont déjà décrits sous cette forme [MM76, Com91, JK91]. Du point de vue de nos travaux, nous pouvons voir ces descriptions comme des systèmes de calculs.

Par exemple l'algorithme d'unification dans l'algèbre libre peut être décrit par des transformations d'un système d'équations entre termes. A chaque pas d'évaluation, le système est remplacé par un autre système "plus simple" jusqu'à ce que la *forme résolue* soit obtenue. Comme règles de transformation d'un système d'équations, nous pouvons prendre par exemple les règles pour l'unification données en [JK91]. Ces règles de réécriture sont présentées d'une manière schématique dans la figure 2.1. Elles sont prouvées correctes et complètes pour une stratégie où elles sont appliquées dans n'importe quel ordre sur la position de tête des termes jusqu'à ce qu'aucune règle ne soit plus applicable. C'est ce qui correspond à la stratégie ELAN (nommé *treeunify*):

$$\begin{array}{lcl} treeunify & = & \textbf{while} \\ & & \quad \textbf{dont care choose}(\text{Delete,Decompose, Conflict,Coalesce,Check,Eliminate}) \\ & & \textbf{endwhile} \end{array}$$

Delete		$P \wedge s = s$	
	$\rightarrow$	P	
Decompose		$P \wedge f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	
	$\rightarrow$	$P \wedge s_1 = t_1 \wedge \ldots \wedge s_n = t_n$	
Conflict		$P \wedge f(s_1, \ldots, s_n) = g(t_1, \ldots, t_p)$	
	$\rightarrow$	$\mathbf{F}$	if $f \neq g$
Coalesce		$P \wedge x = y$	
	$\rightarrow$	$\{x \mapsto y\}P \wedge x = y$	if $x, y \in \mathcal{V}ar(P)$ *and* $x \neq y$
Check		$P \wedge x = s$	
	$\rightarrow$	$\mathbf{F}$	if $x \in \mathcal{V}ar(s)$ *and* $s \notin \mathcal{X}$
Eliminate		$P \wedge x = s$	
	$\rightarrow$	$\{x \mapsto s\}P \wedge x = s$	if $x \notin \mathcal{V}ar(s), s \notin \mathcal{X}, x \in \mathcal{V}ar(P)$

Figure 2.1. treeunify: Les règles pour l'unification syntaxique

2.5 Coder la procédure de SLD-résolution dans un système de calcul

La SLD-résolution est une procédure de résolution particulièrement bien étudié dans le contexte de la logique des clauses de Horn et langages de programmation logique. Un programme d'un tel langage est donnée par une signature contenant un ensemble de variables X, deux ensembles $\mathcal{P}, \mathcal{F}$ de symboles de prédicats et de symboles de fonctions, munis d'arités, et d'un ensemble $\Gamma = \{\alpha : -\beta_1, \cdots, \beta_n \mid \alpha, \beta_i \in atom(\mathcal{P}, \mathcal{F}, \mathcal{X})\}$ de clauses de Horn.

Les formules de la logique des clause de Horn ont une forme:

$$\Gamma \vdash \psi$$

dont la sémantique informelle est que l'atome ψ est une conséquence des axiomes Γ.

Nous allons maintenant coder la procédure de la *SLD-résolution* de cette logique dans un système de calcul. La description de cette procédure sous forme de règles de transformation appartient au folklore des programmeurs. Nous pouvons trouver cette forme de description par exemple dans [Sak90] ou [JM94]. La description présentée dans cette partie est tirée de la thèse de Kô Sakai [Sak90], où elle est présentée dans le contexte des langages avec contraintes.

Etant donné l'ensemble des axiomes Γ et un atome α cette procédure calcule toutes les substitutions θ telle que $\Gamma \vdash \theta\alpha$. Cette procédure est formulée en termes de règles de transformation d'un *but* de la forme

$$(\alpha_1, \cdots, \alpha_n \mid \Delta)$$

où $\alpha_1, \cdots, \alpha_n$ sont des atomes et Δ est une contrainte d'unification satisfiable, c'est-à-dire un système d'équations entre des termes. Si $n = 0$ nous appelons le but *succès*. Le but μ est appelé un *SLD-résolvant* d'un autre but $\nu = (\alpha_1, \cdots, \alpha_n | \Delta)$, noté $\nu \rightarrow \mu$, si:

- $\alpha_1 = p(s_1, \cdots, s_m)$ pour un symbole de prédicat p,
- il y a une variante $p(t_1, \cdots, t_m) : -\beta_1, \cdots, \beta_k$ d'une clause (ne contenant pas de variables communes avec ν) dans Γ,

- la contrainte $\Delta' = \Delta \wedge \{s_1 = t_1 \wedge \cdots \wedge s_m = t_m\}$ est satisfiable,
- et le but $\mu = (\beta_1, \cdots, \beta_k, \alpha_2, \cdots, \alpha_n \mid \Delta')$.

L'ensemble des succès SS est défini comme l'ensemble de tous les $(\alpha_1, \cdots, \alpha_n \mid \Delta)$ tels que

$$(\alpha_1, \cdots, \alpha_n \mid \emptyset) \to^* (\mid \Delta)$$

L'ensemble de toutes les substitutions θ telles que θ est une solution de Δ pour un $(\alpha \mid \Delta) \in SS$ est l'ensemble complet des solutions pour la requête α.

Nous codons une théorie de la logique des clauses de Horn représentée par la signature $X, \mathcal{P}, \mathcal{F}$ et par l'ensemble de clauses Γ dans un système de calcul avec une signature contenant les symboles de fonction $\mathcal{F}' = X \cup \mathcal{P} \cup \mathcal{F} \cup \{\ (\_|\_)\ \}$, où $(\_|\_)$ est un symbole spécial de construction d'un but. L'ensemble de clauses de Horn Γ est traduit en règles de réécriture de la façon suivante:

Pour chaque clause de Horn $p(t_1, \cdots, t_m) : -\beta_1, \cdots, \beta_k \ = C \in \Gamma$, nous définissons une règle de réécriture

$$\begin{array}{rcl} \ell_{step} : (p(s_1, \cdots, s_m), \alpha_2, \cdots, \alpha_n \mid \Delta) & \to & (\theta\beta_1, \cdots, \theta\beta_k, \alpha_2, \cdots, \alpha_n \mid \Delta') \\ & & where \quad \Delta' := (unifiable)\ \Delta \wedge s_1 = \theta t_1 \wedge \cdots \wedge s_m = \theta t_m \end{array}$$

où θ est une substitution de renommage des variables de C sur de nouvelles variables et la stratégie *unifiable* contrôle la satisfiabilité de la contrainte. C'est-à-dire elle transforme la contrainte en une forme où il est évident que la contrainte est satisfiable. Si une telle forme n'existe pas (la contrainte est insatisfiable) la stratégie ne donne pas de résultat, la variable Δ' n'est pas affectée et la règle ne peut pas être appliquée.

Nous définissons aussi des règles additionnelles; premièrement les règles introduisant l'état initial de la procédure de SLD-résolution:

$$\ell_{begin} \ : \ \alpha \ \to \ (\alpha \mid \emptyset)$$

La dernière règle définit l'étape finale de la résolution:

$$\ell_{fin} \ : \ (\mid \Delta) \ \to \ \theta \quad where \quad \theta := (unify)\ \Delta$$

où la stratégie *unify* calcule un unificateur minimal de Δ. Afin d'obtenir un système de calcul, nous complétons ces règles par une stratégie disant qu'un calcul doit commencer par une application de la règle ℓ_{begin}, continuer par plusieurs applications de la règle ℓ_{step} et finir par une application de la règle ℓ_{fin}, ce qui donne la stratégie:

dont know choose(ℓ_{begin})
iterate
    dont know choose(ℓ_{step})
enditerate
dont know choose(ℓ_{fin})

Remarquons ici, pour éviter la confusion que nous utilisons le mot stratégie dans le sens de systèmes de calcul, c'est-à-dire comme une description comment appliquer les règles de réécritures. Ce sens n'a rien de commun avec le sens utilisé dans le contexte de la SLD-résolution où on appelle une stratégie une fonction de sélection d'atome à développer dans la règle ℓ_step. Cette sélection atome, de notre point de vue, concerne uniquement la règle ℓ_step elle même mais pas le niveau des stratégies. Par exemple, dans la description de la règle, que nous utilisons, l'atome sélectionné est toujours le premier atome de la liste.

Le système de calcul que nous avons présenté spécifie la procédure de la SLD-résolution. En particulier, pour un ensemble d'axiomes Γ et un atome β, il existe un calcul:

$$\beta \to_{\ell_{begin}} (\beta \mid \emptyset) \to_{\ell_{step}} \cdots \to_{\ell_{step}} (\alpha \mid \Delta_{n-1}) \to_{\ell_{step}} (\ \mid \Delta_n) \to_{\ell_{fin}} \theta$$

si et seulement si il existe à partir d'atome β une SLD-dérivation résultant à la substitution θ. De plus il existe une correspondance évidente entre l'ensemble des SLD-dérivations et l'ensemble des calculs définie ci-dessus.

Remarquons ici que le même codage peut être utilisé pour spécifier la sémantique opérationnelle de n'importe quel langage dans le cadre des langages de programmation logique avec contraintes CLP. Le seul changement à effectuer est le remplacement des stratégies *unifiable* et *unify* par des stratégies agissant sur des contraintes appropriées.

2.6 L'environnement ELAN

Les idées sur l'implantation de différentes procédures de résolution et résolveurs de contraintes dans des systèmes de calculs sont à l'origine du développement du langage ELAN, conçu pour la spécification des langages de programmation logique avec contraintes.

Le langage ELAN permet de définir et exécuter des systèmes de calculs. Pour une plus grande commodité de ces définitions ELAN offre à l'utilisateur plusieurs outils intéressants. De même que dans le langage OBJ un programme en ELAN est composé de modules, un module est l'unité de base de construction d'un système de calcul. Il définit sa signature, ses règles de réécriture et ses stratégies. Les modules peuvent être paramétrés et peuvent importer d'autres modules, construisant ainsi des systèmes de calcul plus complexes à partir de systèmes de calcul plus simples. Nous avons basé le langage ELAN sur la version multi-sortée de systèmes de calculs. De même que dans le langage ASF+SDF les symboles de fonction sont utilisés dans une notation mix-fixée, ce qui permet à l'utilisateur de définir sa propre forme syntaxique des termes.

Comme nous l'avons pu remarquer dans les deux grandes exemples de ce chapitre, pour définir un système de calcul nous avons souvent besoin de décrire des schémas de règles de réécriture plutôt que directement les règles elles-mêmes. Pour pouvoir exprimer en ELAN ce gendre de schématisation le texte des modules est pré-traité par une procédure de pré-processeur syntaxique.

Le pré-processeur d'ELAN permet de faire de simples répétitions d'un morceau de texte, des remplacements de macro-instructions par un texte et des répétitions d'un morceau du texte pour plusieurs valeurs d'arguments. Cela nous permet d'exprimer des constructions comme des pointillés (par exemple dans $p(t_1, \cdots, t_n)$) souvent utilisées dans les descriptions informelles d'algorithmes. La construction la plus puissante du pré-processeur est la construction FOR EACH qui permet d'engendrer un morceau du texte pour des affectations de variables correspondant aux résultats d'un calcul. Grâce à cette construction, nous pouvons par exemple engendrer une règle de réécriture pour chaque élément d'une liste de termes; les termes peuvent représenter des clauses de Horn et nous pouvons donc engendrer les règles de réécriture pour un ensemble des clauses de Horn donné.

Par exemple en supposant qu'un ensemble Γ de clauses de Horn est représenté par une liste, nous pouvons formaliser la quantification pour la règle ℓ_{step} de la partie précédente en écrivant:

FOR EACH C **SUCH THAT** $C := (extract)element(\Gamma)$ {
< *corps de la règle* >
}
qui impose que le corps de la règle sera généré pour chaque résultat C d'un calcul à partir du terme de départ $element(\Gamma)$ avec la stratégie $extract$, c'est-à-dire pour chaque clause de Horn de Γ.

La puissance de pré-traitement syntaxique d'ELAN est le mieux visible en combinaison avec la possibilité de parametrisation des modules. En continuant l'exemple précèdent nous pouvons par exemple définir un module spécifiant la SLD-résolution qui peut être paramétré par un ensemble de clause de Horn. Nous pouvons aussi bien écrire un module spécifiant la résolution de contraintes qui peut être paramétré par la signature sur laquelle les contraintes sont écrites.

2.6.1 Bibliothèque

La modularité du langage nous a permit la création d'une bibliothèque de modules standards, c'est-à-dire des modules souvent utilisés par des programmes différents. La bibliothèque d'ELAN contient des modules faisant l'interface pour les fonctions predéfinies d'interpréteur, des modules définissant des structures de données d'un usage général et enfin un ensemble de modules pour le codage de notions liées aux logiques du premier ordre. A la dernière catégorie appartiennent des modules spécifiant par exemple les différentes signatures (non-sortées ou multi-sortées), puis des modules définissant l'ensemble de termes, des atomes etc.

La description de la bibliothèque des modules standards fait l'objet du chapitre 5 de la thèse.

3
Systèmes de calcul

Le but de ce chapitre est la présentation de fondements théoriques du langage ELAN. Un programme en ELAN est, en général, représenté par un ensemble de modules, dont quelques uns sont des instances de modules paramétrés. Un programme définit une signature, un ensemble de règles de réécriture et un ensemble de stratégies. L'exécution d'un programme est lancée par application d'une stratégie sur un terme de départ. Une telle exécution applique les règles de réécriture sur le terme de départ en suivant cette stratégie et donne un ou plusieurs résultats. En principe, l'exécution peut aussi boucler et afficher un ensemble infini de résultats.

Le problème que nous voulons aborder dans ce chapitre est une formalisation adéquate d'un programme ELAN. Comme nous l'avons déjà dit dans l'introduction, ELAN est un langage basé sur la logique de réécriture. La formalisation que nous présentons doit donc démontrer la connexion entre la logique de réécriture et un programme ELAN. Grâce à cette connexion, nous pouvons ensuite profiter de travaux théoriques, fait en majorité par J.Meseguer et N.Marti-Oliet, sur les modèles de la logique de réécriture d'un côté et sur l'utilisation de la logique de réécriture comme cadre logique d'un autre côté. Ceci doit nous permettre de mieux comprendre quels sont les modèles des programmes ELAN et comment fonctionne l'implantation de différentes logiques en ELAN.

Un programme ELAN définit une signature, des règles de réécriture et des stratégies décrivant comment appliquer ces règles. Par opposition, une théorie de la logique de réécriture contient uniquement une signature et un ensemble de règles de réécriture. Évidemment, la différence entre ELAN et la logique de réécriture est créée par l'utilisation des stratégies. Pour connecter les deux formalismes nous voulons donc exprimer quel est l'effet de l'utilisation des stratégies dans le contexte de la logique de réécriture. En réalité, les stratégies restreignent tout simplement l'espace de toutes les preuves (toutes les dérivations) d'une théorie à un ensemble des preuves "*intéressantes*". Pour formaliser cette restriction, nous utilisons la notion déjà existant de *terme de preuve* et nous présentons une notion de *système de calcul* qui est une simple généralisation des théories de la logique de réécriture.

Ainsi, dans ce chapitre nous présentons la logique de réécriture et nous généralisons la notion de théorie de cette logique à la notion de *système de calcul*. Un système de calcul restreint les preuves de la logique de réécriture à un sous-ensemble présenté d'une façon explicite. Nous discutons ensuite la liaison entre les systèmes de calcul et les programmes d'ELAN où l'ensemble des preuves admises est exprimé à l'aide de *stratégies*.

3.1 Définitions de base

Dans cette section, nous définissons les notions nécessaires pour pouvoir présenter la logique de réécriture. En particulier, nous commençons par la définition de la notion de signature. Nous construisons ensuite l'ensemble des termes bâtis sur une signature et nous continuons en définissant une équation entre deux termes, une relation de congruence engendrée par un ensemble d'équations et nous finissons par la définition d'un ensemble quotient des termes. Les objets ainsi construits (les classes d'équivalence des termes modulo une congruence engendrée par un ensemble d'équations) représentent les objets de base de la logique de réécriture.

Nous définissons ici une version multi-sortée de toutes les notions qui correspondent le mieux au cas utilisé dans l'implantation d'ELAN.

La première notion que nous définissons est donc la notion de signature qui sert à la désignation des objets. Une signature multi-sortée définit un ensemble de sortes et un ensemble d'opérateurs. Les opérateurs sont munis d'une fonction qui affecte à chaque opérateur son arité et de deux fonctions qui déterminent *le profil* de l'opérateur, c'est-à-dire les sortes de ses domaine et codomaine.

Définition 1 (signature) *Une* signature Σ *est un couple* $(\mathcal{S}, \mathcal{F})$ *où:*
- $\mathcal{S}$ *est un ensemble de* sortes
- $\mathcal{F}$ *est un ensemble d'*opérateurs *(appelé aussi des* symboles de fonction*) munis des fonctions* domaine: $\mathcal{F} \to \mathcal{S}^*$ *et* codomaine: $\mathcal{F} \to \mathcal{S}$*.*

La fonction arity $: \mathcal{F} \to N$ *qui associe à un symbole de fonction* f *la taille de* $domaine(f)$*. Nous appelons une* constante *un opérateur d'arité zéro.*

Nous allons souvent écrire $f : s_1 \times \cdots \times s_n \to s$ *pour un symbole* f *tel que* $domaine(f) = s_1, \cdots, s_n$ *et* $codomaine(f) = s$*.*

Ayant une signature Σ, nous pouvons définir une autre notion syntaxique, les termes. Les termes sont utilisés pour désigner des expressions construites sur des opérateurs définis par une signature. Ils peuvent être enrichis par un ensemble de variables, qui donne une plus grande généralité aux expressions utilisées.

Définition 2 (termes) *Soit* $\Sigma = (\mathcal{S}, \mathcal{F})$ *une signature et* $\mathcal{X}$ *une famille d'ensembles de variables* X_s *pour chaque sorte* $s \in \mathcal{S}$*.* Les termes *sur* Σ *et* $\mathcal{X}$ *sont définis récursivement comme une famille d'ensembles* $\mathcal{T}_s(\mathcal{F}, \mathcal{X})$*, pour chaque* $s \in \mathcal{S}$ *par:*
- $X_s \subseteq \mathcal{T}_s(\mathcal{F}, \mathcal{X})$*,*
- si $t_1, \cdots, t_n \in \mathcal{T}_{s_1}(\mathcal{F}, \mathcal{X}), \cdots, \mathcal{T}_{s_n}(\mathcal{F}, \mathcal{X})$ *et si* $f : s_1, \cdots, s_n \to s \in \mathcal{F}$*, alors* $f(t_1, \cdots, t_n)$ *est un terme de* $\mathcal{T}_s(\mathcal{F}, \mathcal{X})$*.*

Nous utilisons souvent l'abréviation X *pour l'union disjointe des ensembles* X_s*. Si l'ensemble des variables* X *est vide nous appelons l'ensemble des termes* $\mathcal{T}_s(\mathcal{F}, \emptyset)$ *l'ensemble des termes clos, et nous le décrivons simplement par* $\mathcal{T}_s(\mathcal{F})$*. Nous utilisons la notation* $t_{[x_1, \cdots, x_n]}$ *pour exprimer le fait, que terme* t *contient les variables* $x_1, \cdots, x_n \in X$*. Parfois nous le notons par* $t_{[\overline{x}]}$ *si le contexte est évident.*

Nous allons maintenant poursuivre notre construction en définissant un ensemble quotient des termes modulo un ensemble d'équations. Les éléments de cet ensemble sont des classes d'équivalence des termes et ils sont comme les objets de base pour la logique de réécriture.

D'abord nous définissons une identité entre deux termes:

Définition 3 (identité) *Etant donnés une signature* $\Sigma = (\mathcal{S}, \mathcal{F})$ *et un ensemble des variables* X*,* identité *est un couple des termes* $(t_1, t_2) \in \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \times \mathcal{T}_s(\mathcal{F}, \mathcal{X})$*, notée* $t_1 = t_2$*.*

Un ensemble d'identités vu comme un ensemble d'axiomes définit une relation d'égalité. Pour pouvoir définir formellement la relation d'égalité (nous l'appelons la congruence) engendrée par un ensemble d'identités nous définissons d'abord les substitutions:

Définition 4 (substitution) *Soit* $\Sigma = (\mathcal{S}, \mathcal{F})$ *une signature et* X *un ensemble de variables. Une fonction* $\theta : X \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ *est une* substitution *si pour chaque* $x \in X_s$*,* $\theta(x) \in \mathcal{T}_s(\mathcal{F}, \mathcal{X})$*. C'est-à-dire si elle est compatible avec les sortes.*

Une substitution est étendue en un morphisme des termes, par la règle: - $\theta(f(t_1, \cdots t_n)) = f(\theta(t_1), \cdots, \theta(t_n))$ *pour chaque symbole* $f \in \mathcal{F}$ *avec* $arity(f) = n$.

Nous utilisons la notation $\{\omega_1/x_1\}$ *pour exprimer une substitution qui assigne le terme* ω_1 *à une variable* x_1. *Nous utilisons également une version abrégée* $\{\overline{\omega}/\overline{x}\}$ *pour exprimer une substitution d'un ensemble de variables* $x_1, \cdots, x_n$ *sur les termes respectifs* $\omega_1, \cdots, \omega_n$, *si le contexte est évident.*

Maintenant nous pouvons définir la relation de congruence engendrée par un ensemble d'identités.

Définition 5 (congruence engendrée) *Soit* Σ *une signature,* X *un ensemble de variables et* E *un ensemble d'identités. Alors la* relation de congruence $\equiv_E \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$ engendrée par E *est définie inductivement par:*
- $\theta'(u) \equiv_E \theta'(v)$ *pour chaque substitution* $\theta : X \to \mathcal{T}(\mathcal{F})$ *et chaque identité* $u = v \in E$
- $t \equiv_E t$ *pour chaque terme* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$
- *si* $u \equiv_E v$ *alors* $v \equiv_E u$
- *si* $u \equiv_E v$ *et* $v \equiv_E t$ *alors* $u \equiv_E t$
- *si* $u_i \equiv_E v_i$, $f \in \mathcal{F}$ *et* $arity(f) = n$ *alors* $f(u_1, \cdots, u_n) \equiv_E f(v_1, \cdots, v_n)$

Enfin nous abordons la construction principale de cette section, il s'agit de l'ensemble des termes modulo une relation de congruence. En prenant un ensemble des termes et une relation de congruence, les termes égaux modulo cette congruence représentent un seul objet, appelé classe d'équivalence. Nous avons donc:

Définition 6 (ensemble quotient des termes) *Soit* $\Sigma = (\mathcal{S}, \mathcal{F})$ *une signature,* X *un ensemble de variables et* $\equiv_E$ *une relation de congruence. Alors l'ensemble quotient des termes* $\mathcal{T}_E(\mathcal{F}, \mathcal{X})$ *est une famille d'ensembles* $\mathcal{T}_{Es}(\mathcal{F}, \mathcal{X}) = \{[u]_E | u \in \mathcal{T}_s(\mathcal{F}, \mathcal{X})\}$ *où* $[u]_E$ *est la classe d'équivalence définie par* $[u]_E = \{v | v \in \mathcal{T}_s(\mathcal{F}, \mathcal{X}), v \equiv_E u\}$ *pour chaque sorte* $s \in \mathcal{S}$.

3.2 La logique de réécriture

Dans cette section, nous définissons la logique de réécriture qui a été introduite par J. Meseguer [Mes92] comme une conséquence de travaux sur des logiques générales. La logique de réécriture formalise le processus de la réécriture, par opposition à la logique équationnelle qui considère les règles de réécriture comme des égalités et utilise la réécriture pour calculer un représentant unique d'une classe d'équivalence. La logique de réécriture utilise les règles de réécriture pour calculer une relation de "réécrivabilité" entre des termes. Elle permet d'exprimer le fait qu'un terme peut être réécrit en un autre.

Nous définissons donc dans cette section formellement la logique de réécriture, en particulier nous définissons la forme de ses signatures, formules, théories et ses règles de déduction.

Les objets principaux de la logique de réécriture sont les classes d'équivalence des termes. Une signature de la logique de réécriture est alors formée par un ensemble de sortes $\mathcal{S}$, un ensemble d'opérateurs $\mathcal{F}$ et un ensemble d'équations E. Cette signature détermine une relation de congruence $\equiv_E$ engendrée par E et un ensemble quotient des termes modulo $\equiv_E$. Les formules de la logique de réécriture sont construites en utilisant des classes d'équivalence de cet ensemble. En particulier, les formules de la logique de réécriture ont la forme $[t]_E \Rightarrow [t']_E$ représentant intuitivement la proposition: " la classe d'équivalence contenant le terme t peut être réécrite en la classe d'équivalence contenant le terme t' ". Plus formellement:

Définition 7 (signature de la logique de réécriture) *Une* signature de la logique de réécriture $\Theta = (\mathcal{S}, \mathcal{F}, E)$ *est donné par un ensemble de sortes, un ensemble de symboles de fonction et un ensemble d'identités* $E \subseteq \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \times \mathcal{T}_s(\mathcal{F}, \mathcal{X})$.

Définition 8 (formules de la logique de réécriture) *Les* formules de la logique de réécriture *pour une signature* $(\mathcal{S}, \mathcal{F}, E)$ *sont de la forme*

$$[t]_E \Rightarrow [t']_E$$

où $[t]_E, [t']_E \in \mathcal{T}_E(\mathcal{F})$ *sont des classes d'équivalence de termes* t *et* t' *dans l'ensemble quotient des termes modulo* E.

Nous définissons maintenant une théorie de la logique de réécriture. Une théorie de la logique de réécriture est donnée par une signature et par un ensemble de règles de réécriture représentant des axiomes. Les règles de réécriture agissent sur les classes d'équivalence de termes; de plus elle sont étiquetées, nous ajoutons donc à une théorie un ensemble d'étiquettes.

Définition 9 (théorie de la logique de réécriture) *Une* théorie de la logique de réécriture *est un triplet* $\mathcal{R} = (\Theta, L, R)$ *où* $\Theta = (\mathcal{S}, \mathcal{F}, E)$ *est une signature de la logique de réécriture,* L *est un ensemble d'*étiquettes *et* $R \subseteq L \times \mathcal{T}_{Es}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}_{Es}(\mathcal{F}, \mathcal{X})$ *est un ensemble de règles de réécriture,* s *étant une sorte de* $\mathcal{S}$. *Nous écrivons* $\ell : [l]_E \Rightarrow [r]_E$ *un élément* $(\ell, [l]_E, [r]_E)$ *de* R.

Étant donnée une théorie, l'ensemble des formules *impliquées par* (conséquences logiques de) cette théorie est défini par les règles de déduction de la logique de réécriture. Ces règles formalisent la notion de la réécriture des termes (plus précisément des classes d'équivalence des termes) en définissant comment les preuves des théorèmes sont formées.

Définition 10 (règles de déduction de la logique de réécriture) *Pour une théorie* $\mathcal{R} = ((\mathcal{S}, \mathcal{F}, E), L, R)$, *nous disons que* $\mathcal{R}$ *implique une formule* $[t]_E \Rightarrow [t']_E$ *si cette formule peut être obtenue par une suite finie d'applications des règles de déduction suivantes (nous utilisons ici la notation fractionnaire où les prémisses sont notées au-dessus de la barre et la conclusion en-dessous):*

Réflexivité *Pour chaque* $t \in \mathcal{T}(\mathcal{F})$*:*

$$\frac{}{[t]_E \Rightarrow [t]_E}$$

Congruence *Pour chaque symbole de fonction* $f \in \mathcal{F}$ *tel que* $arity(f) = n$*:*

$$\frac{[t_1]_E \Rightarrow [t'_1]_E \quad \cdots \quad [t_n]_E \Rightarrow [t'_n]_E}{[f(t_1, \cdots, t_n)]_E \Rightarrow [f(t'_1, \cdots, t'_n)]_E}$$

Remplacement *Pour chaque règle de réécriture* $\ell : [t]_E \Rightarrow [t']_E$ *de* R *contenant les variables* $x_1, \cdots, x_n$.

$$\frac{[w_1]_E \Rightarrow [w'_1]_E \quad \cdots \quad [w_n]_E \Rightarrow [w'_n]_E}{[\{\overline{w}/\overline{x}\}t]_E \Rightarrow [\{\overline{w'}/\overline{x}\}t']_E}$$

Transitivité

$$\frac{[t_1]_E \Rightarrow [t_2]_E \qquad [t_2]_E \Rightarrow [t_3]_E}{[t_1]_E \Rightarrow [t_3]_E}$$

Intuitivement, une théorie de la logique de réécriture est un système de réécriture. Nous illustrons ces notions sur un exemple:

Prenons par exemple la théorie de la logique de réécriture avec la signature $(\mathcal{S}, \mathcal{F}, E)$ *où:*

$$\begin{array}{rl} \mathcal{S} = & \{\ nat\ \} \\ \mathcal{F} = & \left\{ \begin{array}{ll} zero & : \rightarrow nat, \\ s & : nat \rightarrow nat, \\ plus & : nat \times nat \rightarrow nat \end{array} \right\} \\ E = & \{\ plus(X,Y) = plus(Y,X)\ \} \end{array}$$

et avec les ensembles d'étiquettes et règles de réécriture:

$$\begin{array}{l} L = \{\ \ell_0,\ \ \ell_1\ \} \\ R = \left\{ \begin{array}{lllll} \ell_0 & : & [plus(zero,zero)]_E & \Rightarrow & [zero]_E, \\ \ell_1 & : & [plus(s(X),Y)]_E & \Rightarrow & [s(plus(X,Y))]_E \end{array} \right\} \end{array}$$

En prenant cette théorie nous pouvons prouver par exemple la formule:

$$[plus(zero, s(zero))]_E \Rightarrow [s(zero)]_E$$

en utilisant la règle de déduction Réflexivité *pour obtenir la formule* $[zero]_E \Rightarrow [zero]_E$*, cette formule est ensuite utilisée comme prémisse pour la règle de déduction* Remplacement *avec la règle* ℓ_1 *(la règle est appliquable grâce à la commutativité de plus) afin d'obtenir la formule* $[plus(zero, s(zero))]_E \Rightarrow [s(plus(zero, zero))]_E$*. Puis nous utilisons encore une fois la déduction* Remplacement*, mais avec la règle* ℓ_0 *afin d'obtenir* $[plus(zero, zero)]_E \Rightarrow [zero]_E$ *et nous finissons la preuve en utilisant sur cette formule la règle de déduction* Congruence *pour le symbole* s *et la règle de* Transitivité *pour concaténer les deux applications des règles* ℓ_1 *et* ℓ_0*.*

3.3 Termes de preuve

Dans cette section, pour poursuivre la construction des systèmes de calculs, nous introduisons la notion de *terme de preuve*. Cette notion formalise les preuves de la logique de réécriture en affectant à chaque preuve un terme d'une structure algébrique des preuves. Un terme de preuve est une formalisation d'une preuve en logique de réécriture. Il contient les informations sur la construction d'une formule valide, i.e. les informations sur les règles de déduction appliquées et sur leurs paramètres.

Un terme de preuve π représente une preuve d'une formule de la logique de réécriture de la forme $[t]_E \Rightarrow [t']_E$. Nous formalisons ce fait en définissant deux fonctions qui, appliquées sur π retournent respectivement les classes $[t]_E$ et $[t']_E$. Pour suggérer le point de vue que les termes de preuves représentent les réécritures d'un terme sur un autre, nous appelons ces fonctions respectivement *dom* et *cod*.

L'ensemble des termes de preuve Π est donc muni de deux fonctions $dom : \Pi \rightarrow \mathcal{T}_E(\mathcal{F})$ et $cod : \Pi \rightarrow \mathcal{T}_E(\mathcal{F})$. Dans la définition suivante et dans la suite du chapitre nous écrivons:

$$\pi : [t_1]_E \rightarrow [t_2]_E$$

pour exprimer que π est un terme de preuve avec le $dom(\pi) = [t_1]_E$ et le $cod(\pi) = [t_2]_E$.

Définition 11 (termes de preuve [Mes92]) *Soit* $\mathcal{R} = ((\mathcal{S}, \mathcal{F}, E), L, R)$ *une théorie de la logique de réécriture,* L'ensemble $\Pi \subseteq \mathcal{T}(\mathcal{F} \cup L \cup \{;\})$ des termes de preuve *de* $\mathcal{R}$ *avec les fonctions dom et cod correspondant est obtenu par une suite finie d'applications des règles suivantes:*

Réflexivité *Pour chaque* $t \in \mathcal{T}(\mathcal{F})$*:*

$$\frac{}{t : [t]_E \to [t]_E}$$

Congruence *Pour chaque symbole de fonction* $f \in \mathcal{F}$*:*

$$\frac{\pi_1 : [t_1]_E \to [t'_1]_E \quad \cdots \quad \pi_n : [t_n]_E \to [t'_n]_E}{f(\pi_1, \cdots, \pi_n) : [f(t_1, \cdots, t_n)]_E \to [f(t'_1, \cdots, t'_n)]_E}$$

Remplacement *Pour chaque règle de réécriture* $\ell : [t]_E \Rightarrow [t']_E$ *de* R *contenant les variables* $x_1, \cdots, x_n$.

$$\frac{\pi_1 : [w_1]_E \to [w'_1]_E \quad \cdots \quad \pi_n : [w_n]_E \to [w'_n]_E}{\ell(\pi_1, \cdots, \pi_n) : [\{\overline{w}/\overline{x}\}t]_E \to [\{\overline{w'}/\overline{x}\}t']_E}$$

Transitivité (composition)

$$\frac{\pi_1 : [t_1]_E \to [t_2]_E \qquad \pi_2 : [t_2]_E \to [t_3]_E}{\pi_1; \pi_2 \ : \ [t_1]_E \to [t_3]_E}$$

Nous illustrons maintenant la notion de termes de preuve sur un exemple:

Reprenons l'exemple de la partie précédente, où nous avions deux règles de réécriture:

$$\begin{array}{lcll} \ell_0 & : & [plus(zero, zero)]_E & \Rightarrow & [zero]_E, \\ \ell_1 & : & [plus(s(X), Y)]_E & \Rightarrow & [s(plus(X, Y))]_E \end{array}$$

avec le symbole $plus$ *commutatif. Nous avons déjà prouvé la formule:*

$$[plus(zero, s(zero))]_E \Rightarrow [s(zero)]_E$$

Nous pouvons maintenant formaliser une preuve en utilisant les termes de preuve. En particulier un terme de preuve correspondant à une preuve de cette formule est:

$$\ell_1(zero, zero); s(\ell_0)$$

qui exprime une application de la règle ℓ_1 *suivie par une application de la règle* ℓ_0 *au sous-terme du symbole* s*. Nous pouvons vérifier que:*

$$\ell_1(zero, zero); s(\ell_0) \quad : \quad [plus(zero, s(zero))]_E \to [s(zero)]_E$$

peut être déduite par les règles données dans la définition précédente.

Prenons maintenant par exemple la formule:

$$[plus(s(plus(zero, zero)), zero)]_E \Rightarrow [s(zero)]_E$$

et deux preuves intéressantes de cette formule correspondant aux termes de preuve:

$$\ell_1(plus(zero, zero), zero); s(\ell_0)$$

et

$$plus(s(\ell_0), zero); \ell_1(zero, zero)$$

Le premier correspond à l'application de la règle ℓ_1 *à la tête du terme suivie par l'application de la règle* ℓ_0 *au sous-terme, tandis que le deuxième correspond à l'application de la règle* ℓ_0 *sur le sous-terme suivie par l'application de la règle* ℓ_1 *à la tête du résultat. Les deux termes prouvent la même formule. À part eux nous pouvons trouver encore plusieurs termes de preuve pour la même formule, par exemple:*

$$\begin{array}{l} plus(zero, s(\ell_0)); \ell_1(zero, zero) \\ plus(zero, s(\ell_0)); (\ell_1(zero, zero); s(zero)) \\ (plus(zero, s(\ell_0)); \ell_1(zero, zero)); s(zero) \end{array}$$

3.4 Systèmes de calcul

Dans cette section nous définissons les *systèmes de calcul* qui servent à la formalisation d'un sous-ensemble des programmes en ELAN. Un système de calcul est une théorie de la logique de réécriture avec un ensemble de termes de preuve définissant les *calculs*.

Définition 12 (système de calcul) *Un* système de calcul *est un quadruplet $\mathcal{CS} = (\Theta, L, R, P)$, où (Θ, L, R) est une théorie de la logique de réécriture et P est un ensemble de termes de preuve.*

Nous appelons $\mathcal{R} = (\Theta, L, R)$ la théorie du système de calcul $\mathcal{CS}$. Nous désignons parfois le système de calcul comme étant un couple $(\mathcal{R}, P)$ de la théorie $\mathcal{R}$ et l'ensemble de termes des preuves P.

Nous définissons maintenant l'ensemble des calculs d'un système de calcul. Les calculs sont déterminés par l'ensemble des termes de preuves P. De plus nous incorporons les calculs triviaux et la règle de transitivité des calculs. Nous obtenons donc la définition:

Définition 13 (calcul) *Soit $\mathcal{CS} = (\mathcal{R}, P)$ un système de calcul. Une formule $[t]_E \Rightarrow_P [t']_E$ est* un calcul *de $\mathcal{CS}$, s'il peut être obtenu par une suite finie d'applications des règles:*

identités: *Pour chaque terme t:*

$$[t]_E \Rightarrow_P [t]_E$$

calculs de base: *Si il existe un terme de preuve $\pi \in P$ tel que:*

$$\pi : [t]_E \to [t']_E$$

alors:

$$[t]_E \Rightarrow_P [t']_E$$

transitivité: *si $[t]_E \Rightarrow_P [s]_E$ et $[s]_E \Rightarrow_P [t']_E$ sont deux calculs, alors:*

$$[t]_E \Rightarrow_P [t']_E$$

3.5 Systèmes de calcul et ELAN

Dans cette section nous voulons présenter la correspondance entre le langage ELAN, qui fait l'objet de la thèse et les systèmes de calcul définies dans ce chapitre. Les systèmes de calcul représentent une formalisation pour le mécanisme d'exécution du langage. L'idée est qu'un programme en ELAN définit un système de calcul $\mathcal{CS} = (\mathcal{R}, P)$. L'ensemble des résultats d'une exécution d'un tel programme avec un terme de départ t est l'ensemble:

$$Res = \{ \; [t']_E \;\mid\; [t]_E \Rightarrow_P [t']_E \; \}$$

Nous avons présenté dans l'introduction de la thèse des exemples montrant que les systèmes de calcul constituent un formalisme très naturel pour implanter des résolveurs de contraintes et des procédures de déduction pour divers logiques. Mais ces systèmes sont souvent très grands, très compliqués et ils dépendent de la signature dans laquelle les contraintes sont écrites et de la théorie pour laquelle la procédure de résolution est utilisée, par exemple les règles de réécriture codant un pas de la SLD-résolution sont générées à partir de l'ensemble T des clauses de Horn. Ceci nous oblige à utiliser différents systèmes de calcul à chaque changement de la signature et de la théorie T.

L'idée d'ELAN est alors d'offrir un langage qui permette de définir des systèmes de calcul d'une manière schématique. Pour cela il contient des constructions très puissantes, notamment des *stratégies* pour schématiser un ensemble de termes de preuve; des modules pour composer différents systèmes de calculs; des modules paramétrés pour définir des systèmes de calcul comme des instances de systèmes génériques et, finalement, le pré-processeur syntaxique pour décrire, entre autre, comment générer automatiquement des systèmes de calcul pour une signature ou un ensemble de formules T.

Ce chapitre s'est donné pour but de définir les fondations théoriques du langage, néanmoins, la définition formelle de toutes ses constructions sort du cadre de la thèse. De plus, des constructions du langage comme la modularité, les compositions de modules ou la paramétrisation sont des constructions existant dans autres langages similaires (les spécifications algébriques, par exemple) et leur sémantique est bien étudiée. Il est donc inutile de la présenter ici pour le cas d'ELAN. Nous nous concentrons donc sur un seul module d'ELAN et nous montrons sa relation aux systèmes de calcul. Le point important pour établir cette liaison est de pouvoir donner une signification aux stratégies d'ELAN. Cette signification devrait définir directement un ensemble de termes de preuves. Dans la suite du chapitre, nous définissons donc une fonction $\mathcal{PT}(.)$ qui affecte un ensemble $\mathcal{PT}(\mathcal{S})$ de termes de preuve à une stratégie S. Ainsi nous saurons définir le système de calcul correspondant à un programme d'ELAN formé d'un seul module non paramétré qui de plus n'importe pas d'autres modules. Un tel module définissant une signature Θ, un ensemble de règles de réécriture R (étiquetées $\ell_1, \cdots, \ell_n$) et une stratégie S définit alors le système de calcul $(\Theta, \{\ell_1, \cdots, \ell_n, \imath\}, R, \mathcal{PT}(\mathcal{S}))$.

3.5.1 Stratégies d'ELAN

Dans cette section nous présentons donc formellement la notion de *stratégie* proposée pour le langage ELAN et nous donnons une caractérisation, en termes de plus petit point fixe, de l'ensemble de termes de preuve définie par une telle stratégie. Nous abordons brièvement aussi la sémantique opérationnelle d'application d'une stratégie sur une classe d'équivalence x, néanmoins la preuve de l'équivalence des deux notions n'est pas présentée.

Les stratégies d'ELAN sont définies par une identité $\imath = S$ entre un nom i et le corps $S \in \mathcal{ST}$, où l'ensemble $\mathcal{ST}$ des corps est définie récursivement par les règles:

- si $\imath$ est un nom d'une stratégie alors $\imath \in \mathcal{ST}$.
- si $\ell \in L$ est une étiquette d'une règle de réécriture alors **dont know choose**$(\ell) \in \mathcal{ST}$ et **dont care choose**$(\ell) \in \mathcal{ST}$.
- si $S_1, \cdots, S_n \in \mathcal{ST}$, alors la concaténation $S_1 \cdots S_n \in \mathcal{ST}$, **dont know choose**$(S_1, \cdots, S_n) \in \mathcal{ST}$ et **dont care choose**$(S_1, \cdots, S_n) \in \mathcal{ST}$.
- si $S \in \mathcal{ST}$, alors **iterate** S **enditerate** $\in \mathcal{ST}$ et **while** S **endwhile** $\in \mathcal{ST}$.
- si f est un symbole de fonction d'arité n et $S_1, \cdots, S_n \in \mathcal{ST}$, alors $f(S_1, \cdots, S_n) \in \mathcal{ST}$.

Nous avons ainsi défini le langage des stratégies[1]. Ce langage permet de spécifier des procédures qui combinent des évaluations non-déterministes et déterministes. Les stratégies spécifient, sur une occurence d'évaluation, les règles de réécriture qui doivent être appliquées pour continuer cette évaluation, et aussi le non-déterminisme (*don't care* ou *don't know*) avec lequel l'exploration des possibilités continue. Les stratégies permettent donc (à l'aide de la concaténation et la récursion) d'identifier un endroit dans l'arbre des dérivations, puis (à l'aide de la descente dans

1. Remarquons ici que les stratégies actuellement implantées en ELAN ne contiennent pas une possibilité directe d'application d'une règle sur un sous-terme et d'appel d'une autre stratégie. Mais ces deux cas peuvent être réalisés à l'aide d'affectations locales dans les règles de réécriture, comme nous l'avons illustré dans la section 2.3. C'est pour cette raison que nous devons les inclure dans cette formalisation, de plus dans les prochaines versions d'ELAN ces deux possibilités serons incorporées.

des sous-termes et la récursion) une position dans le terme à réduire et finalement (à l'aide des opérateurs de choix **dont care choose** et **dont know choose**) des règles de réécriture (ou des sous-stratégies) à appliquer ainsi que la gestion du non-déterminisme pour continuer l'évaluation.

Les stratégies d'ELAN sont donc utilisées généralement pour expliciter la gestion du non-déterminisme dans les algorithmes spécifiés. Les autres aspects, par exemple, que certaines règles peuvent être appliquées uniquement si certaines conditions sont valides, sont d'habitude délégués au niveau de règles de réécriture (qui en ELAN peuvent être conditionnelles). Ce langage de stratégies peut paraître à première vue très restreint et peu expressif, mais il nous a pour l'instant suffi pour définir toutes les nuances dont nous avions besoin. De plus, les stratégies écrites dans ce langage sont d'habitude très courtes, elle contiennent uniquement quelques lignes de texte, ce qui facilite la compréhension.

Nous illustrons l'utilisation des stratégies sur deux exemples vers la fin du chapitre. Maintenant, pour alléger le texte de la suite du chapitre nous faisons quelques simplifications:

> Nous imaginons que la concaténation et les opérateurs de choix sont des opérateurs binaires. Le cas général peut être traduit dans ce cadre plus restreint, par exemple une stratégie **dont know choose**$(S_1, \cdots, S_n)$ peut être représenté par la stratégie **dont know choose**$(S_1, \cdots$ **dont know choose**$(S_{n-1}, S_n) \cdots)$.
>
> Dans la suite de la section nous omettons les stratégies **while-endwhile** et **iterate-enditerate** qui peuvent être exprimés à l'aide d'autres primitives. Nous omettons également la stratégie **dont care choose**(ℓ) appliquant d'une façon déterministe la règle de réécriture ℓ. En général, une application d'une règle de réécriture ℓ sur une classe d'équivalence peut donner plusieurs résultats parce que le filtrage équationnel n'est pas forcément unitaire. Pour ces raisons l'utilisateur a la possibilité de choisir uniquement un des résultats en utilisant la stratégie **dont care choose**(ℓ). Le choix du résultat dépend de l'algorithme du filtrage utilisé et il est donc difficile à formaliser. Nous ne prenons donc pas en compte cette stratégie dans la suite du chapitre.
>
> Dans la suite de la section nous utilisons souvent tout simplement la notation ℓ pour exprimer la stratégie **dont know choose**(ℓ).
>
> Nous nous restreignons également au cas d'une seule stratégie récursive, le cas de plusieurs stratégies récursives est une simple généralisation de ce cas simple.

Nous commençons notre construction par une brève formalisation de la sémantique opérationnelle des stratégies d'ELAN. Pour cela, nous présentons d'abord des identités définissant une fonction $[\![.]\!]_\rho : \mathcal{T}_E(\mathcal{F}) \to 2^{\mathcal{T}_E(\mathcal{F})}$ représentant l'évaluation d'application d'une stratégie sur une classe d'équivalence x. Pour pouvoir réutiliser ensuite ces identités dans plusieurs circonstances nous indiçons cette fonction par un paramètre ρ qui représente en général une fonction du même profil que $[\![.]\!]_\rho$.

Les identités sont:

$$
\begin{array}{lcl}
[\![\textbf{dont know choose}(\ell)]\!]_\rho(x) & = & \ell\langle x\rangle \\
[\![\textbf{dont know choose}(S_1, S_2)]\!]_\rho(x) & = & [\![S_1]\!]_\rho(x) \,\cup\, [\![S_2]\!]_\rho(x) \\
[\![\textbf{dont care choose}(S_1, S_2)]\!]_\rho(x) & = & if\ ([\![S_1]\!]_\rho(x) = \emptyset)\ then\ \ [\![S_2]\!]_\rho(x)\ \ else\ \ [\![S_1]\!]_\rho(x) \\
[\![f(S_1, \cdots, S_n)]\!]_\rho(x) & = & \{\ f(y_1, \cdots, y_n)\ \mid\ x = [f(x_1, \cdots, x_n)]_E\ \wedge\ y_i \in [\![S_i]\!]_\rho(x_i)\ \} \\
[\![S_1 S_2]\!]_\rho(x) & = & [\![S_2]\!]_\rho([\![S_1]\!]_\rho(x)) \\
[\![\imath]\!]_\rho(x) & = & \rho(x)
\end{array}
$$

Dans ces identités nous avons utilisé la notation $\ell\langle x\rangle$ pour exprimer l'ensemble de tous les résultats d'application de la règle ℓ sur la classe d'équivalence x. Nous utilisons également la notation $[\![S_2]\!]_\rho([\![S_1]\!]_\rho(x))$ pour exprimer l'ensemble:

$$\{\ z \mid y \in [\![S_1]\!]_\rho(x)\ \wedge\ z \in [\![S_2]\!]_\rho(y)\ \}$$

Nous pouvons maintenant dire que la sémantique opérationnelle d'une stratégie $\imath = S$ correspond à l'évaluation paresseuse d'une fonction $h : \mathcal{T}_E(\mathcal{F}) \to 2^{\mathcal{T}_E(\mathcal{F})}$ définie récursivement par l'identité:

$$h(x) = [\![S]\!]_h(x)$$

Par exemple pour une stratégie:

$$\imath = \textbf{dont know choose}(\ell_1, \ell_2\imath)$$

nous obtenons la fonction définie récursivement par

$$h(x) = [\![\textbf{dont know choose}(\ell_1, \ell_2\imath)]\!]_h(x)$$

ce qui est équivalent à la définition:

$$h(x) = \ell_1(x) \cup h(\ell_2\langle x\rangle)$$

L'évaluation de l'application de la stratégie $\imath$ à une classe d'équivalence $[t]_E$ correspond à l'évaluation de l'expression $h([t]_E)$. Ayant, par exemple, les règles de réécriture $\ell_1 : b \to a$ et $\ell_2 : a \to b$ (a, b sont des constantes) et en évaluant l'expression $h(a)$ nous obtenons le résultat $\{a\}$.

Pour spécifier plus précisément le comportement des stratégies, nous devons préciser la définition de la fonction h. Ainsi, suivant la tradition, nous disons que la fonction h, définie par l'identité $h(x) = [\![S]\!]_h(x)$, est le plus petit point fixe de $[\![S]\!]$, c'est-à-dire:

$$h = \mathit{lfp}([\![S]\!]) = \lim_{n\to\infty} [\![S]\!]^n$$

où "les puissances" de la fonction $[\![S]\!]$ sont définie par:

$$\begin{array}{lcll} [\![S]\!]^0 & = & \perp & (\perp \text{ représente la fonction toujours indéfinie}) \\ [\![S]\!]^{n+1} & = & [\![S]\!]_{[\![S]\!]^n} & \end{array}$$

La fonction h, ainsi définie, caractérise le comportement de la stratégie S, néanmoins cette caractérisation a deux désavantages. Premièrement, elle ne définit pas l'ensemble de termes de preuves associés à S (comme nous l'avons voulu pour les systèmes de calcul), mais une fonction $\mathit{lfp}([\![S]\!]) : \mathcal{T}_E(\mathcal{F}) \to 2^{\mathcal{T}_E(\mathcal{F})}$ associant à un terme de départ un ensemble des résultats. Deuxièmement, cette caractérisation ne fonctionne pas pour les stratégies qui bouclent, mais qui peuvent pourtant donner au fur et à mesure des résultats raisonnables.

Par exemple, pour ℓ une étiquette d'une règle de réécriture, la fonction:

$$h(x) = \ell\langle x\rangle \cup h(x)$$

correspondant à la stratégie:

$$\imath = \textbf{dont know choose}(\ell, \imath)$$

n'est pas définie, parce que la deuxième sous-stratégie de **dont know choose** *boucle. Mais il est tout à fait raisonnable d'exiger une caractérisation de cette stratégie qui contienne une application de la règle ℓ.*

La caractérisation des stratégies présentée ci-dessus est basée sur l'idée que les itérations faites pour obtenir le plus petit point fixe donnent une fonction de plus en plus définie, c'est-à-dire une fonction h donnant le résultat $h(x)$ pour de plus en plus de valeurs de x. Mais une fois que la valeur $h(t)$ est définie pour un certain t, elle doit être définitive. On ne peut plus ajouter d'autres résultats à la valeur $h(t)$ dans les itérations suivantes. C'est pourquoi nous ne pouvons pas spécifier les stratégies qui bouclent. En effet, nous ne connaissons à aucune itération l'ensemble complet des résultats. Cette caractérisation a été dérivée de la présentation des stratégies par des fonctions, c'est-à-dire par des programmes dans un langage fonctionnel. Mais, en principe, nous pouvons imaginer les stratégies aussi comme des programmes dans un langage du type PROLOG.

Par exemple, si $\ell : a \to b$ est une règle de réécriture, alors la stratégie:

$$\imath = \textbf{dont know choose}(\ell, \imath)$$

peut être représentée par un ensemble de clause de Horn définissant un prédicat $eval_\imath$ associant une classe d'équivalence x aux résultats d'application de la stratégie S à x. Le prédicat $eval_\imath$ est alors définie par les clauses:

$$\begin{array}{l} eval_\ell(a,b). \\ eval_\imath(X,Y) \;:-\; eval_\ell(X,Y). \\ eval_\imath(X,Y) \;:-\; eval_\imath(X,Y). \end{array}$$

A la différence d'un programme fonctionnel, par exemple, l'ensemble des Y tels que $eval_\imath(a,Y)$ est bien défini et il est égal à l'ensemble $\{b\}$. Le calcul de toutes les conséquences logiques d'un programme de ce style est basé sur l'idée de partir de l'ensemble des faits du programme et d'ajouter dans les itérations successives les conséquences directes (obtenues par un pas de déduction) qui peuvent être déduites à partir de cet ensemble. Nous utilisons donc cette idée pour définir l'ensemble de termes de preuve associé à une stratégie.

Pour une stratégie $\imath = S$ nous définissons donc la fonction $[\![S]\!] : 2^\Pi \to 2^\Pi$ enrichissant l'ensemble des termes de preuves déjà calculé par les termes de preuves obtenus par une itération de S. Pour obtenir l'ensemble de tous les termes de preuves nous partirons ensuite de l'ensemble vide et nous appliquons successivement la fonction $[\![S]\!]$ jusque l'obtention du plus petit point fixe.

Dans cette approche une complication est posée par l'opérateur **dont care choose**. Rappelons que nous avons défini les résultats de **dont care choose**$(S_1, S_2)(x)$ comme $S_1(x)$ si $S_1(x) \neq \emptyset$ et comme $S_2(x)$ sinon. Ce qui entraîne une négation dans notre traduction des stratégies en programmes PROLOG et complique ainsi la caractérisation des conséquences logiques de ces programmes. Ce problème est, dans le contexte de la programmation logique, traité par la définition d'un ensemble d'*échecs finis*. Dans notre cas, nous pouvons le traiter de la même façon en exprimant l'ensemble des échecs finis à l'aide de la fonction $[\![.]\!]$ définie auparavant.

Ainsi, finalement nous obtenons la définition de la fonction $[\![S]\!] : 2^\Pi \to 2^\Pi$:

$$\begin{array}{lcl}
[\![\textbf{dont know choose}(\ell)]\!](\rho) & = & \{\ \ell(x_1,\ldots,x_n) \ \mid\ x_i \in \mathcal{T}_E(\mathcal{F})\ \} \\
[\![\textbf{dont know choose}(S_1,S_2)]\!](\rho) & = & [\![S_1]\!](\rho)\ \cup\ [\![S_2]\!](\rho) \\
[\![\textbf{dont care choose}(S_1,S_2)]\!](\rho) & = & [\![S_1]\!](\rho) \cup \{\ \pi \ \mid\ \pi \in [\![S_2]\!](\rho) \wedge \exists n : [\![S_1]\!]^n(dom(\pi)) = \emptyset\ \} \\
[\![f(S_1,\cdots,S_n)]\!](\rho) & = & \{\ f(\pi_1,\cdots,\pi_n) \ \mid\ \pi_i \in [\![S_i]\!](\rho)\ \} \\
[\![S_1 S_2]\!](\rho) & = & \{\ \pi_1;\pi_2 \ \mid\ \pi_i \in [\![S_i]\!](\rho)\ \wedge\ dom(\pi_2) = cod(\pi_1)\ \} \\
[\![\imath]\!](\rho) & = & \rho
\end{array}$$

(Remarquons que dans ces identités le symbole ρ est utilisé pour désigner un ensemble de termes de preuve et non une fonction comme c'était le cas pour la fonction $[\![.]\!]_\rho$).

La fonction $[\![S]\!]$ est monotone et continue par rapport à l'inclusion sur les ensembles, il existe donc le plus petit point fixe:

$$lfp([\![S]\!]) = \bigcup_{i=0}^{\infty} [\![S]\!]^i(\emptyset)$$

Ainsi nous avons achevé la construction de l'ensemble des termes de preuve associé à une stratégie $\imath = S$, c'est-à-dire l'ensemble:

$$\mathcal{PT}(\imath = \mathcal{S}) = lfp([\![S]\!])$$

En connaissant l'ensemble des termes de preuve associé à une stratégie nous pouvons maintenant facilement définir par exemple des ensembles de dérivations associés à des stratégies "classiques" de la réécriture, telles que *parralel innermost* ou *leftmost outermost.*

> *Par souci de simplicité nous considérons dans cet exemple uniquement un seul symbole de fonction f d'arité 2 et une seule constante a et une seule règle de réécriture ℓ. Considérons également que nous avons défini une stratégie void qui effectue une identité. Nous pouvons maintenant définir la stratégie PI décrivant les dérivations* parralel innermost*:*
>
> $$PI = \textbf{dont care choose}(\ f(PI, PI)\ ,\ void\ )\ \textbf{dont care choose}(\ \ell\ PI\ ,\ void\ )$$
>
> *Cette stratégie dit que les dérivations* parralel innermost *sont obtenues par applications simultanées de dérivations* parralel innermost *sur les sous-termes d'un terme suivies par une application (si possible) de la règle ℓ sur la position de tête du terme suivie par une dérivation* parralel innermost.
>
> *Similairement, nous pouvons définir la stratégie définissant les dérivations* leftmost outermost *en définissant deux stratégies: LOS pour définir un pas des dérivations et LO pour définir les dérivations entières. Les deux stratégies sont définie par:*
>
> $$\begin{array}{lcl} LOS & = & \textbf{dont care choose}(\ \ell\ ,\ f(LOS, void)\ ,\ f(void, LOS)\ ) \\ LO & = & \textbf{dont care choose}(\ LOS\ LO\ ,\ void\ ) \end{array}$$

Ces deux exemples montrent bien que les stratégies d'ELAN peuvent être utilisées pour définir des dérivations non-triviales. D'autant plus que la vraie puissance des stratégies est dans l'utilisation de non-déterminisme *don't know*, que nous n'avons même pas utilisé dans ces exemples. Un exemple utilisant aussi ce non-déterminisme peut être fourni par la stratégie pour l'unification dans les théories commutatives (c.f. 5.3.7).

4
Le langage ELAN

ELAN est un logiciel destiné à spécifier des langages de programmation logique. La spécification d'un langage de programmation logique est effectuée par le codage de sa syntaxe et sa sémantique opérationnelle dans un système de calcul. Le système de calcul ainsi obtenu est ensuite exécuté par l'interprète d'ELAN.

La description de systèmes de calcul en ELAN est modulaire, un module représentant un système de calcul. Il définit donc une signature, des règles de réécriture et des stratégies. Contrairement à la définition des systèmes de calcul donnée dans le chapitre théorique, les règles de réécriture dans l'implantation pratique peuvent être *conditionnelles* et peuvent contenir des *affectations locales*. De même, le langage des stratégies implanté ne permet pas directement l'application de stratégies sur des sous-termes d'un terme, cette option doit être réalisée un utilisant des affectation locales dans les règles de réécriture. Les modules en ELAN peuvent être paramétrés et peuvent importer d'autres modules construisant ainsi des systèmes de calcul plus compliqués à partir de systèmes de calcul plus simples. Du point de vue de l'exécution, le système de calcul résultant d'importations de modules, est obtenu comme l'union de tous les symboles de fonction de toutes les règles de réécriture et de toutes les stratégies définies dans tous les modules importés soit directement, soit par la clôture transitive d'importations. Néanmoins du point de vue syntaxique la signature obtenue par l'importation d'un module est limitée à la signature *globale* du module (c.f. 4.4).

Pour pouvoir spécifier la sémantique opérationnelle d'un langage de programmation logique ELAN permet de définir des objet spéciaux appelés *description d'un langage de programmation logique* et *spécification*, cette dernière représentant un programme dans le langage spécifié. Pour exprimer le codage d'une procédure de résolution dans un système de calcul, ELAN contient des outils spéciaux réalisés par un pré-processeur.

En général, un programme exécutable en ELAN est une *description d'un langage de programmation logique*. Cette description *importe* des *modules* dont le texte est traité par le pré-processeur. Le système de calcul résultant de l'importation des modules pré-traités est utilisé pour spécifier la syntaxe et le mécanisme d'évaluation des requêtes du langage logique spécifié.

Ce chapitre contient le manuel de référence et la description de la sémantique opérationnelle du langage ELAN. Notons que dans le chapitre nous ne discutons pas les choix que nous avons fait pour définir des parties du langage, ni les possibilités que nous avons considérées. Nous présentons dans ce chapitre le langage ELAN "en bloc" et dans les deux chapitres suivants nous montrons que le langages est bien avéré pour implanter des résolveurs de contraintes et des langages de programmation logique. Nous procédons de cette manière parce que le nombre des possibilités pour différentes constructions d'un langage est grand, et les implications entre les possibilités sont parfois compliquées. Parmi d'autres, nous avons fait des choix concernant les visibilités des sortes et symboles de fonction par rapport à la modularité, les possibilités de renommage et d'*aliasing*, les différents modes d'importation des modules et de paramétrisation. Dans tout ces choix, nous avons essayé d'augmenter l'expressivité du langage tout en gardant sa simplicité et l'uniformité ainsi que la possibilité d'une implantation facile. Ceci est aussi la raison majeure pour laquelle nous avons choisi pour ELAN la version multi-sortée des systèmes de calcul. Une signature multi-sortée correspond directement à une grammaire hors contexte. Les termes correspondent alors directement aux arbres syntaxiques. Nous n'avons donc pas besoin de contrôler le bon typage d'un terme, il nous suffit d'implanter un analyseur syntaxique puissant.

Le chapitre est organisé de la façon suivante: il commence par une description des conventions syntaxiques utilisées dans le manuel, il continue par une section décrivant la lexicographie du langage suivie par une section décrivant définitions de signatures, règles de réécriture et stratégies. Cette section décrit aussi la sémantique opérationnelle des règles et des stratégies. La troisième section décrit les modules d'ELAN. Les modules sont liés par des relations d'importation et de visibilité qui sont aussi précisées dans cette section. L'extension de la notion de module aux modules paramétrés fait sujet de la partie suivante. La cinquième section décrit comment spécifier en ELAN des langages de programmation logique en utilisant des modules. Ceci est lié au mécanisme générique réalisé par le pré-processeur qui est l'outil principal pour la réalisation du codage de la logique d'un langage particulier dans un système de calcul. A la fin de ce chapitre, nous mentionnons quelques informations indispensables pour l'utilisation pratique d'ELAN. Il s'agit notamment des objets câblés et de la description de l'utilisation d'ELAN sous UNIX.

Le chapitre contient par ailleurs beaucoup d'exemples illustrant la plupart des constructions du langage.

Version BNF utilisée

Pour définir la syntaxe nous utilisons une version de la forme BNF où:

• <*non-terminal*>	les symboles non terminaux sont écrits en italique et encadrés par des chevrons.
• **mot clés**	les mots clés sont écrits en gras.
• **. , ; : ( ) ...**	les caractères spéciaux sont plus grands et en gras.
• { itération 0 et plus }*	l'itération zéro et plus est entre parenthèses et marquée par une étoile.
• { itération 1 et plus }$^+$	l'itération une et plus est entre parenthèses et marquée par un plus.
• [omission]	les parties facultatives sont entre crochet.
• (choix 1 | choix 2)	le choix entre deux possibilités est enfermé dans des parenthèses et séparé par le symbole |.
• choix1 | choix2	si les deux items dans l'opérateur de choix sont évidents on peut omettre les parenthèses.

Les règles de grammaire ont une forme:

<*non-terminal de partie gauche*> ::= expression de la partie droite

S'il y a plusieurs règles (à la suite) avec la même partie gauche, on peut éviter la répétition en utilisant l'opérateur de choix |.

<*non-terminal de partie gauche*> ::= expression de la partie droite
| une autre expression pour la même partie gauche

4.1 Lexicographie

Séparateurs blancs

Tous les caractères dont le code ASCII est plus petit ou égal à 32 (code ASCII de l'espace) sont considérés par l'interpréteur comme des séparateurs blancs, donc y compris les caractères de nouvelle ligne, de tabulation etc.

Les unités lexicales

Une unité lexique en ELAN est soit un identificateur, soit un nombre entier, soit un caractère spécial. Les identificateurs sont formés par une chaîne des lettres, chiffres et du caractere "_" commençant par une lettre.

<*lettre*> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<*chiffre*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<*identifier*> ::= <*lettre*> { <*lettre*> | <*chiffre*> | _ }*

Un nombre est une chaîne des chiffres.

<number> ::= { *<chiffre>* }$^+$

Tous les caractères, à part des lettres, chiffres, séparateurs blancs et les caractères

{ } ˜

sont considérés comme des caractères spéciaux et nous les référençons par le non-terminal *<char>* . Nous avons exclu ces caractères parce que les caractères { } ˜ sont utilisés par le pré-processeur.

Dans les noms de symboles de fonction nous utilisons aussi l'unité lexicale *<nchar>* qui correspond à n'importe quel caractère de *<char>* à part les caractères:

"@" et ":"

Commentaires

Les textes enfermés entre les symboles /* (sans espace entre les deux) et */ et les textes après le double symbole // jusqu'à la fin de la ligne sont considérés comme des commentaires.

4.2 Définition de signature, règles de réécriture et stratégies

Les signatures ELAN sont multi-sortées, elles sont donc formées par définition de sortes et de symboles de fonction. La signature définit l'ensemble des termes, qui sont généralement utilisés dans une écriture mix-fixée. Cela veut dire que la syntaxe des termes doit être entièrement spécifiée par l'utilisateur en même temps que la signature. Nous utilisons en ELAN une forme de definition de la signature similaire à celle de langages OBJ [FGJM84, GW88]. Le nom de chaque symbole de fonction exprime formation syntaxique des termes. La définition d'un ensemble de symboles de fonction aboutit donc à une grammaire hors contexte des termes. L'interprète d'ELAN dispose de l'algorithme d'Earley pour effectuer l'analyse syntaxique d'un terme.

Nous décrivons dans cette section une signature en ELAN, où les définitions des symboles de fonction peuvent contenir de plus des options pour la résolution des ambiguïtés syntaxiques et la définition des propriétés sémantiques des symboles.

4.2.1 Sortes

Une sorte en ELAN est donnée par son nom. Un nom de sorte est soit un identificateur, soit un identificateur suivi par une chaîne de noms de sortes séparés par des virgules et mis entre crochets. C'est-à-dire:

<sort name> ::= *<identifier>*
| *<identifier>* [*<identifier>* { , *<identifier>* }*]

Par exemple les noms: boolean, int, list[boolean], pair[int,list[boolean]] *sont des noms de sortes en* ELAN.

4.2.2 Symboles de fonction

Une déclaration de symbole de fonction en ELAN est soit une déclaration d'un nouveau symbole de fonction, soit une définition d'alias. Une déclaration de nouveau symbole contient le nom du symbole, son profil et des options.

<symbol declaration> ::= *<new symbol declaration>*
| *<alias>*

<new symbol declaration> ::= *<name>* : *<profil>* [*<options>*]

Noms des symboles de fonction

Un nom de symbole de fonction en ELAN spécifie la syntaxe des termes formés par ce symbole. Les noms de symboles sont représentés en général par une chaîne d'unités lexicales quelconques. Le nom peut contenir un caractère spécial '@' qui réserve *la place pour l'argument.* Si n est l'arité d'un symbole, le nom du symbole doit contenir exactement n occurrences du caractère '@'. Les unités lexicales dans la définition d'un nom de symbole peuvent être enfermées dans des apostrophes et une construction spéciale est utilisée pour pouvoir introduire des caractères particuliers par leurs codes ASCII. Un nom de symbole est:

<name> ::= { *<symbol>* }$^+$

<symbol> ::= *<single lexem>*
| **@**
| ' *<single lexem>* '
| '\ *<number>* '

<single lexem> ::= *<identifier>*
| *<number>*
| *<nchar>*

où *<identifier>* est n'importe quel identificateur sauf les mots clés **local** et **endop**. Le *<nchar>* est n'importe quel caractère à part des lettres ou chiffres et les caractères suivants

{ } ˜ : @

Les caractères { } ˜ sont réservés pour le pré-processeur, le caractère ':' marque la fin du nom et le caractère '@' note une place pour un argument. On peut utiliser les caractères ':' et '@' sans leur sens spécial en les mettant entre des apostrophes. Le nombre après le caractère '\' mis entre apostrophes représente un caractère défini par son code ASCII. Cette possibilité sert à ajouter dans la forme syntaxique des termes certains caractères non visibles ou non imprimable. Le caractère définie par le code ASCII 32 est compris par l'interpréteur d'ELAN comme un caractère vide, et peut être utilisé pour définir des grammaires avec des règles epsilon.

Termes

La syntaxe des termes en ELAN est précisée dans la section 4.2.3 du chapitre, nous anticipons ici une partie de cette définition pour pouvoir mieux expliquer et exemplifier les options syntaxiques utilisées dans le langage.

En principe, pour écrire un terme en ELAN, on écrit le nom du symbole de tête où les occurrences des caractères '@' sont remplacées par des sous-termes appropriés.

> *Par exemple si nous avons défini un symbole de fonction* if @ then @ else @ fi *et si* x,y,z *sont des variables, alors le texte* if x then y else z fi *correspond au terme ayant le symbole de tête* if @ then @ else @ fi *et les sous-termes* x,y,z.

Profil

La définition du profil d'un symbole de fonction est formée par les noms des sortes des sous-termes entre parenthèses suivis par le nom de la sorte du résultat.

<profil> ::= *<sort name>*
| ({ *<sort name>* }$^+$) *<sort name>*

Aliasing

Aliasing est la possibilité de définir deux noms pour le même symbole de fonction. La forme syntaxique de la définition d'un alias est:

<*alias*> ::= <*symbol declaration*> **alias** <*old name*> **:**

Dans cette définition le symbole défini dans le <*symbole déclaration*> ne représente qu'un autre nom (une autre forme syntaxique) pour le symbole <*old name*> .

Par exemple ayant défini:

@ + @	**:** *(int int) int*	
+(@,@)	**:** *(int int) int*	**alias** *@+@:*

les deux chaînes "$x + y$" et "$+(x, y)$" représentent le même terme.

Entre autre, l'aliasing peut être utilisé pour résoudre des ambiguïtés syntaxiques.

Par exemple le texte:

$$x + y + z$$

est ambiguë. Il y a deux termes qui correspondent à ce texte (ce sont les termes $+(+(x, y), z)$ et $+(x, +(y, z))$). Nous pouvons éviter cette ambiguïté en utilisant l'écriture du terme définie par l'alias, c'est-à-dire par exemple $+(+(x, y), z)$, mais nous pouvons aussi définir encore un alias:

(@+@)	**:** *(int int) int*	**alias** *@+@:*

qui nous permet de parentheser les expressions d'une façon habituelle. Ainsi nous pouvons écrire par exemple:

$$(x + y) + z$$

pour exprimer le terme $+(+(x, y), z)$.

Une autre façon de résoudre des ambiguïtés syntaxiques dans certains cas est l'utilisation d'options spéciales.

Options pour la résolution des ambiguïtés syntaxiques

En ELAN dans le cas de symboles infixés, il est possible de définir l'associativité syntaxique de symboles. Cela se fait en ajoutant l'option **assocLeft** ou **assocRight** à la définition du symbole.

Par exemple la définition de l'opérateur de multiplication:

*@ * @*	**:** *(int int) int*	**assocLeft**
**(@,@)*	**:** *(int int) int*	**alias** *@*@:*

résout l'ambiguïté dans le texte $x * y * z$ en préférant le terme $*(*(x, y), z)$.

Mais l'associativité syntaxique ne résout que des conflits entre les mêmes symboles. Pour le cas de symboles différents (par exemple dans le $x + y * z$) ELAN permet de définir la priorité syntaxique du symbole. La priorité syntaxique se définit comme une option où le mot clé **pri** est suivi par un entier définissant la priorité.

Par exemple on peut redéfinir les opérateurs d'addition et de multiplication par

@ + @	**:** *(int int) int*	**assocLeft pri** *10*
*@ * @*	**:** *(int int) int*	**assocLeft pri** *20*

pour obtenir un analyseur habituel des expressions numériques.

Coercion

Un autre trait des signatures en ELAN est l'utilisation de symboles de fonction "cachés", c'est-à-dire des symboles qui ne sont pas visibles dans l'écriture des termes. Ce genre de symbole est très souvent utilisé pour des conversions implicites de sortes.

Par exemple définissons un symbole de fonction qui convertit les nombres en entiers.

ntoi(@) **:** *(number) int*

(La sorte number *est une sorte prédéfinie en* ELAN *qui correspond à l'unité lexicale* <number> *d'un nombre décimal). Nous pouvons maintenant définir aussi la version "cachée" de ce symbole*

@ **:** *(number) int* **alias** *ntoi(@):*

avec laquelle nous pouvons écrire par exemple

$$1 + 2 * 3$$

pour exprimer le terme

$$+(ntoi(1), *(ntoi(2), ntoi(3)))$$

Options définissant des propriétés sémantiques

En plus des options permettant la résolution des conflits syntaxiques, on peut définir des options définissant des propriétés sémantiques équationelles (c.f.3.2). Dans l'implantation actuelle, la théorie équationelle utilisable est la théorie associative commutative. On peut spécifier qu'un symbole est associatif et commutatif en ajoutant l'option (AC) dans la définition du symbole. Dans ce cas, toutes les règles de réécriture seront appliquées modulo AC.

Par exemple le constructeur d'union d'ensembles peut être défini comme

@ ∪ @ **:** *(set set) set* *(***AC***)*

Remarque: Si un symbole est déclaré comme associatif-commutatif, ELAN suppose (si ce n'est pas explicitement défini autrement) qu'il est syntaxiquement associatif à droite.

L'option code

La dernière option est l'option **code** étroitement liée a l'interpréteur d'ELAN et qui ne peut être utilisée que dans les modules standard. L'option code fait la liaison entre la forme syntaxique des fonctions prédéfinies et leur représentation interne. Si on utilise l'option **code** dans la définition d'un symbole, on déclare après le mot clé **code** un entier, qui définit l'indice du symbole dans la table des symboles de l'interpréteur d'ELAN.

Par exemple le symbole true est défini dans le module boolean par la ligne:

true **:** *boolean* **code** *1*

Pour résumer, la syntaxe des options pour les déclarations de symboles de fonction en ELAN est:

$$
\begin{array}{lll}
<\!option\!> & ::= & \textbf{assocLeft} \\
 & | & \textbf{assocRight} \\
 & | & \textbf{pri}\ <\!number\!> \\
 & | & \textbf{(}\ \textbf{AC}\ \textbf{)} \\
 & | & \textbf{code}\ <\!number\!>
\end{array}
$$

4.2.3 Termes

La syntaxe des termes en ELAN est entièrement dépendante de la signature définie par l'utilisateur. Le principe pour l'écriture des termes est le même que dans l'environnement ASF+SDF [Kli93] ou dans les langages OBJ [GW88, GKK+87].

En principe, pour écrire un terme en ELAN on écrit le nom du symbole de tête où les occurrences des caractères '@' sont remplacées par des sous-termes appropriés.

Plus précisément, à partir d'une signature multi-sortée Θ l'interpréteur d'ELAN génére une grammaire $\mathcal{G}$ pour les termes telle que les sortes de Θ sont les non-terminaux de $\mathcal{G}$ et pour chaque déclaration d'un symbole de fonction dans Θ:

nom **:** (*domaine*) *codomaine*

la grammaire $\mathcal{G}$ contient une règle:

<*codomaine*> ::= *nom′*

ou *nom′* est le nom du symbole *nom* où les occurrences du symbole @ sont remplacées par des non-terminaux des *domaines* appropriés.

Ainsi, par exemple la déclaration du symbole de fonction:

@ + @ **:** (*int int*) *int*

est transformée sur la règle de grammaire:

<int> ::= <int> **+** <int>

Cette transformation aboutit donc à une grammaire hors contexte où chaque non-terminal <*sss*> définit la syntaxe des termes de la sorte sss. [1] Dans la suite du chapitre pour ne pas confondre les sortes avec les non-terminaux nous ajoutons le mot "term" dans les noms de non-terminaux correspondant aux sortes, ainsi nous utilisons par exemple le non-terminal <*boolean term*> pour les termes de la sorte boolean. Nous utilisons également le non-terminal <*term*> tout court pour les termes d'une sorte qui est précisée dans le texte accompagnant.

4.2.4 Règles de réécriture

ELAN permet d'utiliser des règles de réécriture conditionnelles et enrichies par des affectations locales. Une règle de réécriture est donc donnée par un couple de termes suivie par une chaîne facultative de conditions et d'affectations locales. Une condition est un terme de la sorte boolean introduit par le mot clé **if**. Une affectation locale est introduite par le mot clé **where** et elle affecte à une variable des résultats obtenus par application d'une stratégie sur un terme.

Dans la syntaxe concrète pour une règle de réécriture, l'étiquette de la règle est introduite par le mot clé **rule**, la sorte des termes de la partie gauche et droite de la règle est introduite par le mot clé **for** et les variables utilisées dans la règle sont définies après le mot clé **declare**.

Le terme dans l'affectation locale doit être de la même sorte que la variable affectée.

La syntaxe d'une règle de réécriture est:

1. Cette grammaire contient en réalité aussi des informations supplémentaires sur les priorités et associativités syntaxiques et sur les aliasings de symboles.

<rewriting rule> ::= **rule** [*<rule name>*] **for** *<sort name>*
declare { *<variable declar>* }$^+$
body
<rule body>
end of rule

<rule body> ::= *<term>* **=>** *<term>*

<rule body> ::= *<rule body>* **if** *<boolean term>*

<rule body> ::= *<rule body>* **where** *<var name>* **:=** **(** [*<strategy name>*] **)** *<term>*

<variable declar> ::= [{ *<var name>* **,** }* *<var name>*] **:** *<sort name>* **;**

<var name> ::= *<identifier>*

<rule name> ::= *<identifier>*

Le nom d'une règle de réécriture n'est pas forcément unique. Plusieurs règles de réécriture peuvent avoir le même nom. Les ambiguïtés ainsi provoquées sont gérées par les stratégies. D'autre part, puisque le nom d'une règle n'est pas obligatoire, on peut définir des règles sans noms. L'application de ces règles est dirigée par la stratégie de normalisation, qui est une stratégie "câblée" d'ELAN. La façon dont les règles de réécriture sont appliquées est décrite après la description de familles de règles et de stratégies, dans la section 4.3 de ce manuel.

Familles des règles

La syntaxe d'une règle de réécriture est assez verbeuse et la définition d'un programme mène a un ensemble de règles dont la description est longue et répétitive. Comme de nombreuse règles partagent des parties de déclarations telle que la sorte des termes, variables locales etc, nous avons introduit une syntaxe courte pour les groupes de règles appelés *familles de règles*.

L'utilisation de familles de règles représente aussi un avantage méthodologique puisqu'elle permet de regrouper des règles qui sont logiquement liées. Par exemple toutes les règles définissant une seule fonction peuvent être définies comme une famille de règles.

La syntaxe d'une famille de règles est:

<family of rules> ::= **rules for** *<sort name>*
[**declare** { *<variable declar>* }$^+$]
bodies
{ *<non complete rule>* }$^+$
end of rules

<non complete rule> ::= **rule** [*<rule name>*]
[**declare** { *<variable declar>* }$^+$]
body
<rule body>
end | **end of rule**

<non complete rule> ::= **[** *<rule name>* **]** *<rule body>*
end | **end of rule**
| **[ ]** *<rule body>*
end | **end of rule**

<non complete rule> **::=** *<family of rules>*
| *<rewriting rule>*

Les déclarations définies pour une famille sont valides pour toutes les règles incluses dans la famille.

Par exemple dans la famille des règles:

```
rules for boolean
bodies

  [not1]  not true        => false              end
  [not2]  not false       => true               end

  rules for boolean
  declare P : boolean;
  bodies
    [or1]  P or true      => true               end
    [or2]  P or false     => P                  end

    rule and
    declare Q : boolean;
    body
        P and Q           => not ( not P or not Q )
    end
  end of rules

  rule inteq
  declare n1, n2 : number;
  body
    n1 = n2               => numequal(n1,n2)
  end

end of rules
```

*Toutes les règles utilisent l'information sur la sorte de la partie gauche et droite, puis les règles or*1 *et or*2 *utilisent de plus la déclaration de la variable P et la règle and utilise aussi la déclaration de la variable Q, tandis que la règle inteq ne peut utiliser que ses propres variables n*1 *et n*2.

4.2.5 Stratégies

Les stratégies d'ELAN servent à guider les applications des règles de réécriture. Une particularité des stratégies d'ELAN vient de la gestion du non-détermisme. Dans le cas où l'exécution peut continuer de plusieurs façons différentes (il y a plusieurs règles applicables sur le terme courant ou il y a plusieurs possibilités d'appliquer une seule règle) il est possible de forcer l'interpréteur d'ELAN à explorer successivement toutes les possibilités en faisant un chaînage arrière. En conséquence une évaluation partant d'un terme avec une stratégie peut donner plusieurs résultats.

Les stratégies ELAN sont exprimées par des expressions construites à partir de noms de règles par l'opérateur de concaténation, les opérateurs d'itération et les opérateurs de choix. C'est-à-dire:

<strategy definition> **::=** **strategy** *<strategy name>*
<strategy>
end of strategy

<strategy> **::=** *<rule choose>*
| *<concatenation>*
| *<iterator>*
| *<choosing>*

<*strategie name*> **::=** <*identifier*>

Dans le reste de cette section, nous précisons la syntaxe et la sémantique opérationnelle pour chaque type de construction de stratégie.

Les stratégies en ELAN ne décrivent que des évaluations où les règles de réécriture sont appliquées uniquement sur la tête des termes (le cas d'application de règles à l'intérieur de termes peut être fait par l'utilisation des affectations locales dans des corps de règles).

Choix d'une règle

La stratégie la plus simple décrit une seule application d'une seule règle de réécriture. Mais puisque en général une application d'une règle de réécriture peut donner plusieurs résultats, il faut décrire aussi la gestion de ce non-déterminisme. Pour cela il faut enfermer le nom de la règle dans un des opérateurs de choix. Les opérateurs de choix ELAN sont **dont care choose** pour un choix déterministe et **dont know choose** pour un choix non-déterministe. La syntaxe d'une stratégie appliquant une règle est donc:

<*rule choose*> **::=** **dont care choose(** <*rule name*> **)**
| **dont know choose(** <*rule name*> **)**

Pour une règle de réécriture nommée ℓ l'application de la stratégie **dont care choose**(ℓ) sur un terme ne donne qu'au plus un seul résultat même dans le cas, où l'application de la règle ℓ peut en engendrer plus. Par contre, l'application de la stratégie **dont know choose**(ℓ) force le chaînage arrière de l'interpréteur ELAN et mène à l'ensemble de tous les résultats possibles obtenus par une seule application de la règle ℓ.

Prenons par exemple une signature contenant les déclarations :

empty	: set	
@	: (elem) set	
@ ∪ @	: (set set) set	**(AC)**
element(@)	: (set) elem	

et une règle de réécriture:

```
rule extractrule for elem
declare S : set;
        e : elem;
body
    element(S ∪ e)        => e
end of rule
```

Dans ce cas, en essayant d'appliquer la règle extractrule *sur la tête du terme*

$$empty \cup a \cup b$$

où a et b sont des constantes de la sorte elem*, le filtrage modulo AC de ce terme avec la partie gauche de la règle* extractrule *nous donne deux filtres possibles: le premier* $\{S \to empty \cup a\ ,\ e \to b\}$ *et le deuxième* $\{S \to empty \cup b\ ,\ e \to a\}$*. De là, la règle peut être appliquée de deux façons en donnant deux résultats différents. La gestion des résultats dépend alors de l'opérateur de choix avec lequel la règle est utilisée.*

En appliquant la stratégie **dont care choose***(extractrule) on obtient comme résultat soit la constante a, soit la constante b. Lequel des deux résultats nous obtenons dépend de l'implantation de l'algorithme du filtrage AC utilisé par* ELAN.

D'autre part, l'application de la stratégie **dont know choose***(extractrule) sur le même terme donne toujours deux résultats, c'est-à-dire les deux constantes a et b.*

Choix de sous-stratégies

De même que précédemment, on va distinguer deux possibilités pour l'opérateur de choix.

$$
\begin{array}{lll}
<\!\mathit{choosing}\!> & ::= & \textbf{dont care choose}\big(\ <\!\mathit{strategy}\!>\ \{\ \big|\big|\ <\!\mathit{strategy}\!>\ \}^*\ \big) \\
 & | & \textbf{dont know choose}\big(\ <\!\mathit{strategy}\!>\ \{\ \big|\big|\ <\!\mathit{strategy}\!>\ \}^*\ \big)
\end{array}
$$

La signification des deux opérateurs ressemble à celle de la partie précédente.

La stratégie "dont know choose" exécute toutes les sous-stratégies et retourne tous les résultats de toutes les sous-stratégies.

La stratégie "dont care choose" retourne tous les résultats de la première de ses sous-stratégies ayant au moins un résultat. L'interpréteur d'ELAN commence donc l'évaluation de la stratégie "dont care choose" par l'évaluation de la première sous-stratégie. Si celle-ci ne donne pas de résultat, il continue par l'évaluation de la sous-stratégie suivante, jusqu'au moment où une des sous-stratégies donne un résultat. Dans ce cas, il retourne tous les résultats de cette sous-stratégie et il termine l'évaluation de la stratégie "dont care choose' majeur. Si aucune des sous-stratégies ne donne pas de résultat, la stratégie majeure n'en donne pas non plus.

Prenons par exemple quatre règles de réécriture:

```
[r1]  a   => c        end
[r2]  a   => d        end
[r3]  b   => e        end
[r4]  b   => f        end
```

et la stratégie:

```
strategy strat
  dont care choose(
    dont know choose( r1 )
   ||
    dont know choose( r2 )
   ||
    dont know choose( r3 r4 )
  )
end of strategy
```

Dans ce cas, la stratégie strat *appliquée sur le terme a donne un seul résultat, le terme c. La même stratégie appliquée sur le terme b donne deux résultats, les termes e et f.*

En travaillant avec ELAN on rencontre souvent des stratégies où l'opérateur de choix d'une règle est inclus dans le même opérateur de choix de sous-stratégies. ELAN permet d'abréger cette construction et de mettre dans l'opérateur de choix une liste des règles. Dans ce cas, la stratégie:

operateur de choix ($rule_1 \cdots rule_n$)

est une abréviation pour la stratégie:

opérateur de choix (
  opérateur de choix ($rule_1$)
  ||
  ⋮
  ||
  opérateur de choix ($rule_n$)
)

où l'*opérateur de choix* est soit **dont care choose**, soit **dont know choose**.

Sur le même principe, s'il y a plusieurs règles ayant le même nom, l'opérateur de choix appliqué sur cette règle est compris comme une abréviation pour l'opérateur de choix de la liste de toutes les règles avec ce nom.

Concaténation

La stratégie qui permet d'enchaîner deux stratégies a la syntaxe:

<concatenation> ::= *<strategy>* *<strategy>*

L'application de la concaténation de deux stratégies sur un terme t a pour résultat tous les résultats de l'application de la deuxième stratégie sur l'ensemble de tous les résultats de l'application de la première stratégie sur t. Dans le cas particulier où une des deux stratégies ne donne aucun résultat la concaténation n'en donne pas non plus.

Itérations

Les opérateurs d'itération servent à définir des stratégies qui concatenent successivement la même stratégie. En ELAN on peut utiliser deux opérateurs d'itération, l'opérateur **iterate** et l'opérateur **while**. [2] La syntaxe d'une stratégie construite par l'opérateur d'itération est:

<iterator> ::= **iterate** *<strategy>* **enditerate**
| **while** *<strategy>* **endwhile**

L'opérateur **iterate** correspond à l'opérateur étoile des expressions régulières. Si on applique une stratégie **iterate** s **enditerate** sur un terme t, on obtiendra le terme t lui même, comme résultat correspondant à l'itération zéro fois, puis tous les résultats de l'application de la stratégie s sur le terme t, puis tous les résultats de l'application de la concaténation s s sur t etc. Bref on obtiendra tous les résultats de l'application de la stratégie s^n sur t, où n est un entier.

L'opérateur **while** est un opérateur d'itération qui calcule les formes normales de la tête d'un terme par rapport à sa sous-stratégie. Autrement dit, si t' est un résultat de l'application de n itérations d'une stratégie s sur un terme t, et si l'application de s sur t' ne donne pas de résultat, alors t' est un résultat de l'application de la stratégie **while** s **endwhile** sur t.

Prenons par exemple une signature contenant:

```
@            : ( elem ) nelist
@ . @        : ( elem nelist ) nelist
element(@)   : ( nelist ) elem
```

et les règles, qui extraient un élément à partir d'une liste d'éléments non vide:

```
rules for elem
declare e : elem;
        l : nelist;
bodies
  [extract]  element(e)      => e            end
  [extract]  element(e.l)    => e            end
  [extract]  element(e.l)    => element(l)   end
end of rules
```

2. dans les versions antérieurs du langage l'opérateur **while** était appelé **repeat**

Supposons de plus, que les constantes a, b, c sont de la sorte elem.

Dans ce cas, l'application de la stratégie " **while dont know choose***(extract)* **endwhile***" sur la liste "a.b.c" donne l'ensemble des résultats* $\{a, b, c\}$.

Tandis que l'application de la stratégie " **iterate dont know choose***(extract)* **enditerate***" sur la même liste donne l'ensemble des même résultats mais aussi des résultats "intermédiaires", c'est-à-dire l'ensemble* $\{element(a.b.c),\ a,\ element(b.c),\ b,\ element(c),\ c\}$.

*On peut essayer d'appliquer aussi la stratégie "***while dont care choose***(extract)* **endwhile***" sur la même liste. Dans ce cas, on obtient un seul élément de la liste comme résultat. Et ce sera le premier élément a.*

4.3 Mécanisme d'évaluation

Nous avons présenté dans cette section la syntaxe de définition d'un système de calcul. Une exécution d'ELAN est lancée par l'application d'une stratégie sur un terme de départ. Normalement, les règles sont appliquées en suivant la stratégie et en effectuant un retour arrière en cas d'échec. Si l'exécution réussit à appliquer la stratégie entière, un terme obtenu est annoncé comme un des résultats d'exécution et les autres résultats sont cherchés par retour arrière.

Cependant, après des observations pratiques, nous avons introduit un système d'évaluation qui diffère légèrement de ce modèle théorique.

Dans un système de calcul ELAN, les règles de réécriture sont divisées en deux ensembles majeurs: les règles étiquetées par un certain nom et dont les applications sont guidées par des stratégies définies par l'utilisateur et les règles sans nom qui ne peuvent donc pas être référencées dans les stratégies. Les application de règles sans noms sont guidées par une stratégie spéciale câblée dans l'interpréteur ELAN. Cette stratégie est appelée une stratégie de *normalisation* parce qu'elle force l'application des règles sans nom dans n'importe quel ordre sur n'importe quelle position du terme jusqu'à l'obtention d'une forme normale. [3]

L'évaluation d'ELAN lancé par l'application d'une stratégie sur un terme se déroule donc en appliquant les règles de réécritures nommées en suivant la stratégie. Après chaque application réussie, la stratégie de normalisation est utilisée pour évaluer les fonctions apparues dans le terme. Le mécanisme d'évaluation d'ELAN applique donc une règle de réécriture nommée et ensuite il calcule la forme normale du terme par rapport aux règles de réécriture sans nom. L'évaluation continue à partir de ce terme normalisé.

Ce mécanisme d'évaluation correspond au cas où les règles avec nom (souvent appliquées d'une façon non-déterministe) réalisent la partie du système de calcul correspondant a un niveau supérieur de l'évaluation, tandis que les règles sans nom définissent l'évaluation des fonctions ordinaires (et déterministes) apparaissant dans des parties droites de règles. Les règles sans nom forment d'habitude un système de réécriture confluent et terminant et leur application simplifie les termes plus compliqués en des termes plus simples (mais en restant dans la même classe d'équivalence au sens des spécifications algébriques).

Application d'une règle de réécriture générale

Maintenant, nous avons suffisamment de connaissance pour pouvoir décrire l'application d'une règle de réécriture. Notons que l'application d'une règle générale $g \Rightarrow d$ avec une étiquette ℓ et avec une chaîne de conditions et d'affectations locales $s_1, \cdots, s_n$ sur un terme peut donner plusieurs résultats puisque le filtrage entre le terme à réduire et g peut donner plusieurs filtres et puisque les affectations locales peuvent appeler des stratégies non-déterministes. Toutes

3. En réalité les règles de réécriture sans noms sont appliquées sur les positions leftmost-innermost en essayant d'appliquer sur chaque position toutes les règles dans l'ordre dans lequel elles sont présentées dans le programme.

les possibilités d'application doivent être prise en compte, ce qui est réalisé par une procédure de retour arrière.

L'application d'une règle de la forme:

$$\ell : g \Rightarrow d \qquad s_1, \cdots, s_n$$

sur un terme t commence par le filtrage de g par t. Le filtrage calcule une substitution σ telle que $\sigma(g) =_{AC} t$. S'il y a plusieurs filtres possibles, une des substitutions est choisi et les autres sont gardés pour les eventuels retours arrière. L'application continue ensuite par l'évaluation des conditions et des affectations locales $s_1, \cdots, s_n$ en commençant par l'évaluation de s_n et en allant vers s_1 par le processus suivant:

1. Si le s_i actuellement évalué est une affectation locale:

   $$\textbf{where } x_i := (strat_i) \ t_i$$

   alors un des résultats de l'application de la stratégie $strat_i$ sur le terme t_i est choisi (soit t'_i) pour enrichir la substitution σ par une liaison $x_i \rightarrow t'_i$. Les autres résultats sont gardés pour le cas de retour arrière et l'évaluation continue par l'évaluation de s_{i-1}.

   Si l'application de $strat_i$ sur le terme t_i ne donne aucun résultat, l'évaluation doit faire un retour arrière et choisir une autre valeur parmi les affectations locales du s_{i+1} (ou un autre filtre σ si s_i était la dernière affectation).

2. Si le s_i actuellement évalué est une affectation locale:

   $$\textbf{where } x_i := () \ t_i$$

   c'est-à-dire une affectation sans appel de stratégie, alors seulement la stratégie de normalisation est appliquée sur t_i pour obtenir un terme t'_i et la substitution σ est enrichie par l'affectation $x_i \rightarrow t'_i$.

3. Si le s_i actuellement évalué est une condition:

   $$\textbf{if } c_i$$

   alors le terme c_i est évalué avec la stratégie de normalisation. Si le résultat est la constante $true$ l'évaluation avance sur l'évaluation du s_{i-1} si non, l'évaluation fait un retour arrière vers s_{i+1}.

L'évaluation des affectations locales et conditions est donc réalisée par une procédure de *backtracking* où les conditions non satisfaites (et les affectations non applicables) sont les points d'échec et les affectations locales sont les points de branchement.

Si toutes les conditions et affectations locales sont évaluées avec succès, le terme t de départ peut être remplacé par $\sigma(d)$, c'est-à-dire par la partie droite de la règle instanciée par la substitution résultant du filtrage et de l'évaluation des affectations locales. Les autres résultats possibles d'application de cette règle sur t sont trouvés en continuant la procédure de retour arrière.

Prenons par exemple le problème des huit reines sur un échiquier. Définissons la règle

```
rule queensrule for list[int]
declare p1,p2,p3,p4,p5,p6,p7,p8 : int;
body

   queens        => p1.p2.p3.p4.p5.p6.p7.p8.nil
```

```
                if ok(1,p8,p7.p6.p5.p4.p3.p2.p1.nil)
                where p8:=(extract) elem(1.2.3.4.5.6.7.8.nil)
                if ok(1,p7,p6.p5.p4.p3.p2.p1.nil)
                where p7:=(extract) elem(1.2.3.4.5.6.7.8.nil)
                if ok(1,p6,p5.p4.p3.p2.p1.nil)
                where p6:=(extract) elem(1.2.3.4.5.6.7.8.nil)
                if ok(1,p5,p4.p3.p2.p1.nil)
                where p5:=(extract) elem(1.2.3.4.5.6.7.8.nil)
                if ok(1,p4,p3.p2.p1.nil)
                where p4:=(extract) elem(1.2.3.4.5.6.7.8.nil)
                if ok(1,p3,p2.p1.nil)
                where p3:=(extract) elem(1.2.3.4.5.6.7.8.nil)
                if ok(1,p2,p1.nil)
                where p2:=(extract) elem(1.2.3.4.5.6.7.8.nil)
                where p1:=(extract) elem(1.2.3.4.5.6.7.8.nil)

end of rule
```

où l'opérateur elem énumère tous les éléments d'une liste et l'appel de la fonction $ok(1, newQueen, oldQueens)$ contrôle si une nouvelle reine peut être posée sur la position $newQueen$, en sachant que nous avons déjà posé les reines $oldQueens$. Cette fonction est définie par les règles:

```
rules for boolean
declare p,d,diff : int;
        l : list[int];
bodies
  [] ok(diff,d,nil)        => true                                                end
  [] ok(diff,d,p.l)        => d != p and d-p != diff and p-d != diff and ok(diff+1,d,l)   end
end of rules
```

Une seule application de cette règle donne une position correcte des huit reines. Donc, si on applique la stratégie **dont care choose***(queensrule) sur la constante queens, on obtiendra une seule position des reines, tandis que avec la stratégie* **dont know choose***(queensrule) on obtiendra tous les résultats.*

4.4 Modules

Le module est l'unité principale du langage. Il correspond à la définition d'un système de calcul dans le sens de la section 3.4. Un module contient donc la définition d'une signature, des règles de réécriture et des stratégies. De plus, les modules sont liés par la relation d'importation. Le graphe des importations en ELAN doit être sans cycle. Quand un module en importe un autre, il ajoute le système de calcul défini dans celui-ci à son propre système. Le système de calcul d'un module correspond donc au système de calcul défini par lui-même et enrichi par l'ensemble de tous les systèmes de calcul définis dans les modules appartenant à la clôture transitive de ses importations. Un module en ELAN a la syntaxe:

<module> ::= **module** *<formal module name>*
[*<imports>*]
[*<signature definition>*]
{ *<rewriting rule>* | *<familly of rules>* }*
{ *<strategy definition>* }*
end of module

Visibilité

On dit qu'un objet est visible dans un module s'il peut être utilisé dans les constructions définies dans ce module. Des règles spéciales définissent la visibilité des objets en ELAN. La visibilité des règles de réécriture, stratégies et noms des sortes est globale. [4] C'est-à-dire que n'importe quelle sorte, nom de règle ou nom de stratégie peut être utilisé dans n'importe quel module, ne dépendant pas des importations de celui-ci. Cela signifie aussi, que les noms des sortes, règles et des stratégies sont uniques pour tout l'ensemble des modules.

Par contre, la visibilité des noms des symboles de fonction est locale. C'est-à-dire que dans un module on ne peut utiliser que des symboles de fonction définis dans le module lui même où dans des modules explicitement nommés dans la partie des importations.

Du point de vue des importations, on distingue deux types de symboles de fonction à l'intérieur d'un module: les symboles globaux et locaux. Les symboles globaux sont les symboles qui seront exportés, si un autre module importe ce module. Ce sont donc des symboles visibles de l'extérieur du module. Les symboles locaux sont des symboles qui ne peuvent être utilisés qu'à l'intérieur du module où ils sont définis. A l'intérieur d'un module, nous pouvons donc parler de signatures globale et locale.

Le fait, qu'un module importe un autre module signifie que sa signature locale est enrichie par tous les symboles globaux du module importé. Et comme la signature locale n'est plus exportée nulle part, cela nous garantit la non transitivité de la visibilité des symboles de fonction. Cependant, pour les cas où on veut définir la visibilité de façon transitive, on peut le faire en ajoutant le mot clé **global** devant la liste des importations. Cela provoque l'enrichissement de la signature globale du module par les symboles importés et donc ces signatures seront exportées vers les autres modules. La syntaxe de la partie définissant la signature est:

<*imports*> ::= **import**
    [**global** { <*module name*> }$^+$;]
    [**local**] { <*module name*> }$^+$;

<*signature definition*> ::= [**sort** { <*sort name*> }$^+$;]
    [**op**
        [**global** { <*symbol declaration*> ; }$^+$]
        [**local** { <*symbol declaration*> ; }$^+$]
    endop
    ]

<*module name*> ::= <*sort name*>

Visibilité d'un alias

Pour la visibilité des définitions des alias, on applique les mêmes règles que pour la visibilité des symboles de fonction. Le cas intéressant est celui où un module importe localement un symbole, et définit un alias pour ce symbole dans sa partie globale de signature. Dans ce cas, sa signature globale contient la définition d'alias mais pas la définition du symbole original. Par conséquent, seul le nom du symbole défini par alias est visible à l'extérieur du module. On ne voit pas le nom original du symbole, ce qui nous permet de faire des renommages de symboles de fonction.

4. Néanmoins, l'utilisation d'une sorte qui n'était pas déclarée dans le module actuel ou dans ses importations provoque un message de warning. Par contre, une double déclaration de la même sorte n'est pas considérée comme une erreur et ne donne pas de message de warning.

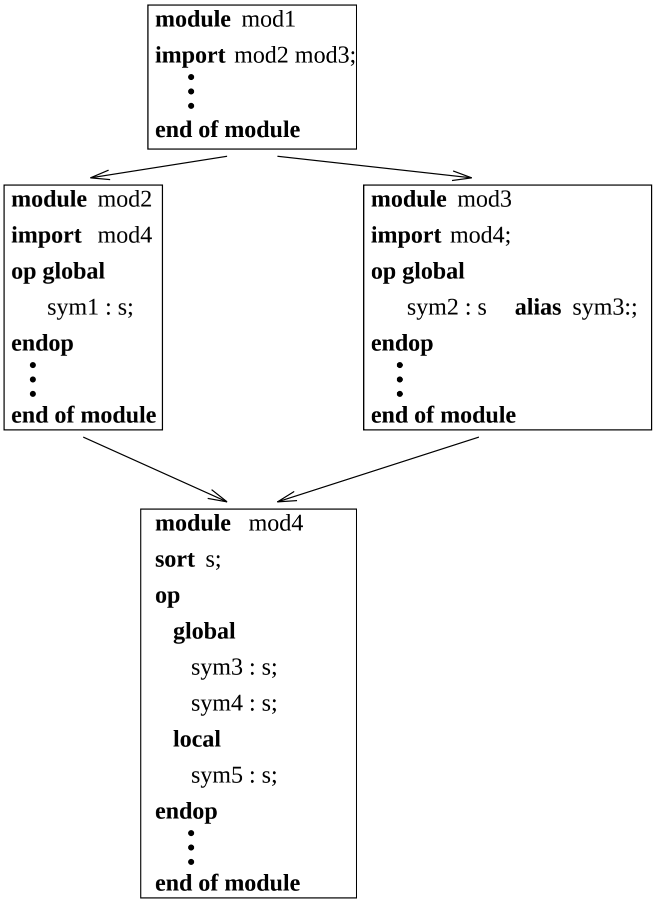


Figure 4.1. Un exemple de visibilité des symboles de fonction par rapport aux importations. Les visibilités des symboles sont les suivantes: Le symbole *sym5* n'est visible que dans le module *mod4*. Le symbole *sym4* est visible dans les modules *mod2*, *mod3* et *mod4*. Le symbole *sym3* est visible dans les modules *mod2*, *mod3*, *mod4*, mais il est visible aussi dans le module *mod1* sous le nom *sym2*. Le symbole *sym2* n'est qu'un autre nom pour le symbole sym3 visible dans les modules *mod1* et *mod3*. Le symbole *sym1* n'est visible que dans les modules *mod1* et *mod2*.

4.5 Modules paramétrés

L'interpréteur d'ELAN contient un mécanisme simple de passage des paramètres, ce qui permet de définir des modules paramétrés. Le nom formel d'un module est donc constitué par le nom du module suivi par des paramètres formels.

<formal module name> **::=** *<identifier>*
| *<identifier>* **[** *<identifier>* { **,** *<identifier>* }* **]**

L'instanciation d'un module par ses arguments effectifs consiste en un remplacement syntaxique des arguments effectifs par les arguments actuels dans le corps du module (c.f. aussi 4.6). Puisque l'instanciation est faite au niveau purement syntaxique il n'y a pratiquement pas de restriction sur la nature des arguments effectifs. Il y a cependant une restriction sur la syntaxe des arguments effectifs. L'interpréteur d'ELAN n'accepte que des arguments dont la syntaxe correspond aux noms des sortes ou aux noms des modules. Le bon sens de programmation devrait restreindre les paramètres effectifs sur des objets de base du langage, tels qu'un nom de module ou de sorte.

Un exemple d'un module paramétré est une définition du module `list` *paramétré par la sorte des éléments:*

```
module list[X]
sort X list[X];
op
  global
     nil           : list[X];
     @ . @         : ( X list[X] ) list[X];
     append(@,@) : ( list[X] list[X] ) list[X];
endop

rules for list[X]
declare e         : X;
        l,l1,l2    : list[X];
bodies
  [] append(nil,l)          => l                         end
  [] append(e.l1 , l2)      => e . append(l1,l2)         end
end of rules

end of module
```

Un autre exemple un peu tordu de l'utilisation de la parametrisation textuelle est fourni par le module `mapcar`. *Il définit une seule fonction mapcar_Fun qui applique une fonction Fun sur tous les éléments d'une liste. Le module est paramétré par la sorte des éléments de la liste, par le nom de module où est définie la fonction à appliquer et enfin par le nom de cette fonction (elle est obligatoirement utilisée dans l'écriture préfixée). Le texte de ce module est:*

```
module mapcar[X,Mod,Fun]
import list[X] Mod;
op global
     mapcar_Fun(@)        : ( list[X] ) list[X];
endop

rules for list[X]
declare e : X;
        l : list[X];
bodies
  [] mapcar_Fun(nil)        => nil                       end
  [] mapcar_Fun(e.l)        => Fun(e) . mapcar_Fun(l)    end
end of rules

end of module
```

4.6 Pré-processeur

Afin d'autoriser une paramétrisation forte et augmenter ainsi les possibilités de définition de systèmes de calcul ELAN contient une procédure de pré-évaluation textuelle. Le texte de chaque module avant de passer à l'interpréteur d'ELAN est traité par une procédure de pré-traitement qui reconnaît un certain nombre de constructions, les évalue, et remplace le morceau de texte original par le résultat. Le texte ainsi obtenu est ensuite compilé comme un module ordinaire.

Le pré-processeur d'ELAN permet de faire de simples répétitions d'un bloc de texte, des remplacements de macro-instructions par un texte et des répétitions d'un bloc du texte pour plusieurs valeurs des arguments.

Remarquons que la paramétrisation des modules en ELAN peut être comprise comme une construction du pré-processeur où les arguments formels sont des macro-instructions à remplacer par les arguments effectifs. C'est d'ailleurs ainsi que la paramétrisation est implantée.

Pour exprimer la syntaxe des constructions du pré-processeur, nous utiliserons la notion d'expression constante et de texte. Voici leurs définitions formelles avec la signification évidente:

<constant expression> ::= *<number>*
| (*<subexpression>*)

<subexpression> ::= *<number>*
| *<subexpression>* + *<subexpression>*
| *<subexpression>* - *<subexpression>*
| *<subexpression>* * *<subexpression>*
| *<subexpression>* / *<subexpression>*
| *<subexpression>* % *<subexpression>*
| (*<subexpression>*)

<lexem> ::= *<identifier>*
| *<number>*
| *<char>*

<text> ::= { *<lexem>* }$^+$

Les constructions du pré-processeur sont données ci après.

La répétition simple

La construction la plus simple du pré-processeur est une répétition n fois d'un morceau du texte, où n est une constante. La syntaxe de la répétition simple est:

<simple repetition> ::= *<lexem>* ~ *<constant expression>*
| { *<text>* }~ *<constant expression>*

Par exemple le texte “a{, b}˜3” sera remplacé par le texte “a, b, b, b”.

La répétition avec un argument

La répétition avec un argument récopie un morceau de texte en substituant a chaque fois un identificateur par une valeur numérique prise dans un intervalle.

<arg.repetition> ::= { *<text>* }_ *<macro_name>* = *<first value>* ... *<last value>*

<macro_name> ::= *<identifier>*

<first value> ::= *<constant expression>*

<last value> ::= *<constant expression>*

Par exemple le texte “P { & s_I=t_I }_I=1...3” sera remplacé par le texte “P & s_1=t_1 & s_2=t_2 & s_3=t_3”.

Une forme spéciale de la répétition avec un argument est une construction d'une liste d'identificateurs avec des indices. Cette construction a la syntaxe:

<arg.repetition> ::= *<identifier>* _ *<number>* *<char>* ... *<char>* *<identifier>* _ *<number>*

Par exemple le texte “t_1,...,t_5” sera remplacé par le texte “t_1 , t_2 , t_3 , t_4 , t_5”.

Construction FOR EACH

La construction FOR EACH est la construction la plus puissante du pré-processeur. Elle permet de lier des remplacements successifs de variables du texte à l'évaluation d'une chaîne d'affectations et de conditions où les conditions sont introduites par le séparateur **ANDIF** et les affectations par le séparateur **AND**. L'évaluation de ces affectations et conditions s'effectue comme l'évaluation des affectations locales et des conditions dans le corps d'une règle de réécriture. La différence réside dans le fait que ces affectations et conditions sont évaluées durant la phase de compilation et utilisent le système de calcul importé des paramètres du module. [5]

L'affectation d'une variable est comprise comme définition d'une macro-instruction pour remplacer le nom d'une variable par le texte correspondant à sa valeur. C'est-à-dire dans le morceau du texte répété, toutes les occurrences des noms des variables seront remplacés par leurs valeurs textuelles.

La construction FOR EACH a la syntaxe suivante:

<for each const.> `::=` **FOR EACH** *<vardecl>* **SUCH THAT** *<varaffect>* **:** {
    <text>
}

<vardecl> `::=` *<varname>* { **,** *<varname>* }* **:** *<sort name>*
    `|` *<vardecl>* **;** *<vardecl>*

<varaffect> `::=` *<varname>* **:=** (`[` *<strategy name>* `]`) *<term>*
    `|` *<varaffect>* **AND** *<varaffect>*
    `|` *<varaffect>* **ANDIF** *<condition>*

<condition> `::=` *<term>*

Avant d'illustrer l'utilisation de la construction **FOR EACH**, rappelons que les symboles

{ } ~

sont réservés pour des constructions du pré-processeur, il est donc impossible de les utiliser en dehors de leur signification spéciale. Le pré-processeur utilise également le symbole "_" pour "coller" des morceau des identificateurs. Par exemple le texte "`a f _1_ x b`" est pré-traité sur une suite des trois identificateurs "`a`", "`f_1_x`" et "`b`". Le caractère "_" fait donc partie d'un identificateur même s'il était séparé de celui là par un séparateur blanc. Il est donc déconseillé d'utiliser le symbole '_' en dehors des constructions du pré-processeur et des identificateurs. Le symbole '_' sert d'habitude pour indicer des identificateurs pendant le pré-traitement. Le pré-processeur lui même ne respecte pas les identificateurs avec le symbole "_". Donc s'il y a une partie d'identificateur séparé par "_" reconnue comme une macro-instruction il la remplace par sa valeur effective.

> *En utilisant les constructions du pré-processeur, nous pouvons décrire, par exemple, la règle de décomposition de l'algorithme d'unification, présentée dans la figure 2.1 du chapitre 2. Si nous supposons qu'un symbole de fonction est défini par un couple (nom du symbole, arité), nous pouvons représenter un ensemble de symboles par une liste de couples. Par exemple, pour une liste* $fss = cons([f,2], cons([g,1], nil))$*, nous pouvons donc écrire:*

5. En réalité puisque l'implantation actuelle ne fait pas de différence entre les arguments et les importations, on utilise tous les modules importés dans le module actuel pour l'évaluation des expressions de la construction FOR EACH.

```
FOR EACH FS:pair[identifier,number]; F:identifier; N:number
SUCH THAT FS:=(extract)elem(fss) AND F:=()first(FS) AND N:=()second(FS) :{
 rule decompose
 declare     s_1,...,s_N : term;
             t_1,...,t_N : term;
 body
             P & F(s_1,...,s_N)=F(t_1,...,t_N)          => P { & s_I=t_I }_I=1...N
 end
}
```

Le pré-processeur engendre dans ce cas les règles suivantes:

```
rule decompose
declare     s_1,s_2 : term;
            t_1,t_2 : term;
body
            P & f(s_1,s_2)=f(t_1,t_2)                  => P & s_1=t_1 & s_2=t_2
end
rule decompose
declare     s_1 : term;
            t_1 : term;
body
            P & g(s_1)=g(t_1)                          => P & s_1=t_1
end
```

4.7 Description d'un langage de programmation logique

ELAN est un langage conçu initialement pour la spécification de langages de programmation logique et il offre des moyens particulier pour spécifier cette classe de langages. Une entité exécutable d'un langage de programmation logique est formée par un texte de programme (qu'on appellera la spécification pour ne pas confondre avec un programme en ELAN) et par une requête. La spécification peut être composée de plusieurs parties; la partie signature et la partie des clauses de Horn, par exemple. Le corps de chaque partie est représenté comme un terme en ELAN. C'est-à-dire que pour pouvoir décrire un langage de programmation il faut d'abord écrire les modules d'ELAN définissant la syntaxe de chaque partie. Les langages de programmation ont souvent des exigences sémantiques supplémentaires pour des parties de leurs programmes. Ces propriétés ne peuvent pas être exprimées par une grammaire hors contexte, comme par exemple le bon typage des termes dans une signature compliquée. Dans ce cas, on peut définir une fonction de contrôle, qui retourne la constante *true* si le corps de la partie est accepté (s'il était bien typé, par exemple). Cette fonction est introduite dans la description d'un langage par les mots clés **check with** suivi par un terme du type boolean qui peut contenir une variable spéciale *spart* instanciée au moment de l'évaluation par le corps de la partie. Si la grammaire de cette partie est ambiguë et s'il y a plusieurs possibilités d'analyser le corps de la partie, l'interpréteur d'ELAN essaie toutes les analyses syntaxiques possibles jusqu'à ce que l'analyse fournisse un résultat accepté par la fonction de contrôle.

Pour ces raisons une description d'une partie doit contenir la sorte du corps de la partie et le module ELAN (ou les modules) où est définie la signature contenant cette sorte. Elle peut contenir aussi un appel à une fonction de contrôle de propriétés sémantiques.

Il est nécessaire de définir également la requête et le mécanisme d'exécution en dehors de la spécification. Pour spécifier la syntaxe et le contrôle des propriétés sémantiques d'une requête, on utilise le même mécanisme que pour une partie de la spécification. Pour spécifier le lancement de l'exécution, on donne la stratégie à appliquer et le terme de départ avec sa sorte. Le terme de départ peut contenir une variable spéciale *query* qui sera instanciée au moment du lancement de l'exécution par la requête introduite par l'utilisateur.

La syntaxe concrète de la description d'un langage de programmation logique et d'une specification est :

```
<lpl description> ::= LPL <language name> description
                      specification description
                          { <specification part description> }+
                      end
                      query
                              of sort <query sort name>
                              result of sort <result sort name>
                              module[ s ] { <module name> }+
                              [ check with <boolean term> ]
                              start with <startterm>
                      end of LPL description

<specification part description> ::= part <partname>
                                      of sort <part sort name>
                                      module[ s ] { <module name> }+
                                      [ check with <boolean term> ]


<specification> ::= specification <specification name>
                        { <partname>  <term> }+
                    end of specification

<language name> ::= <identifier>

<specification name> ::= <identifier>

<partname> ::= <identifier>

<part sort name> ::= <sort name>

<query sort name> ::= <sort name>

<result sort name> ::= <sort name>

<startterm> ::= < <result sort name> term>
```

Passage des informations

Pour permettre l'accès aux informations décrites dans une spécification, ELAN contient un mécanisme spécial. Pendant la lecture d'une spécification, l'interprète d'ELAN génère des nouveaux modules, que nous appelons *modules spéciaux*. Pour une partie de la spécification nommée *partname* avec un corps t, l'interpréteur engendre une nouvelle constante *partname* et une règle de réécriture

$$partname \Rightarrow t$$

La définition de ce symbole et la nouvelle règle sont placées dans un module qui est également nommé *partname*. Ainsi, les autres modules, s'ils ont besoin d'utiliser les informations définies dans cette partie *partname* de la spécification peuvent simplement importer ce module *partname*. L'interpréteur d'ELAN génère un tel module, dès qu'il a lu une partie de la spécification. Les informations contenues dans le corps d'une partie peuvent alors être utilisées dans les modules définissant les parties suivantes. Par exemple, l'ensemble des symboles de prédicats défini dans une partie de la spécification peut être utilisé dans un module définissant les clauses de Horn.

Dans la suite de la thèse, nous appelons *modules spéciaux* les modules contenant une constante du même nom que le module. Ces modules spéciaux sont d'habitude utilisés comme des arguments effectifs des autres modules, qui utilisent alors le nom d'argument de deux façons: comme nom d'un module à importer et comme nom d'une constante qui s'évalue sur un terme important. Ce qui permet entre autre d'utiliser les constructions de paramétrisation forte du pré-processeur pour définir, par exemple, les termes sur une certaine signature (c.f.5.3.2).

Les modules générés par l'interpréteur d'ELAN, sont dans notre terminologie des modules spéciaux mais, en principe, nous pouvons définir d'autres modules spéciaux à la main.

Reprenons l'exemple d'un langage fondé sur la logique des clauses de Horn (c.f. 6.1). La spécification consistera alors en déclarations des variables, de symboles de fonction, de symboles de prédicat et de l'ensemble des clauses de Horn bâties sur cette signature. Elle sera donc composée de quatre parties. La requête dans ce langage sera représentée par un atome et le terme de départ de l'exécution sera l'état initial de la procédure de SLD résolution. Un fichier de description d'un langage pour cette logique est par exemple le fichier:

```
LPL simpleprolog description

  specification description

    part Vars          of sort list[identifier]
                       modules anyidentifier signatures
    part Ops           of sort list[pair[identifier,number]]
                       modules anyidentifier signatures
    part Predicates    of sort list[pair[identifier,number]]
                       modules anyidentifier signatures
    part Clauses       of sort list[pair[atom,list[atom]]]
                       module HornClauses[Vars,Ops,Predicates]
  end

  query          of sort atom
                 result of sort SLD_res_state
                 module SLD_resolution[Vars,Ops,Predicates,Clauses]
                 start with (SLD_resolution) (query.nil | true)
end of LPL description
```

Et un fichier correspondant à un programme dans ce langage simpleprolog *est par exemple:*

```
specification member

  Vars
                 X   Y   Z ;
  Ops
                 cons:2  nil:0  a:0  b:0  c:0  ;
  Predicates
                 member:2 ;

  Clauses

                 member(X,cons(X,Y)).
                 member(X,cons(Y,Z)) :- member(X,Z),.
                 ;

end of specification
```

*L'interpréteur d'*ELAN*, pendant la compilation, génère au fur et à mesure les modules spéciaux* `Vars`, `Ops`, `Predicates`, `Clauses`. *Voici par exemple les corps des deux premières:*

```
module Vars
import anyidentifier signatures;
op
   Vars : list[identifier];
endop
rule for list[identifier]
body
      Vars        => X Y Z ;
end of rule
end of module
```

```
module Ops
import anyidentifier signatures;
op
   Ops  : list[pair[identifier,number]];
endop
rule for list[pair[identifier,number]]
body
      Ops         => cons:2  nil:0 a:0 b:0 c:0 ;
end of rule
end of module
```

Ces modules peuvent être ensuite utilisés dans les modules définissant d'autres parties du langage, par exemple pour instancier le module `HornClauses` *décrivant les clauses de Horn bâties sur une signature* Vars, Ops *et* Predicates.

Remarquons ici que nous pourrions créer les modules spéciaux `Vars` *ou* `Ops` *aussi à la main en les définissant comme des modules ordinaires. Dans ce cas aussi nous pourrions les utiliser pour instancier les modules* `HornClauses` *et* `SLD_resolution`. *La description d'un langage de programmation et le mécanisme de passage d'informations d'*ELAN *peuvent donc être considérés comme un "sucre syntaxique" pour rendre plus jolies et plus compactes les descriptions des différentes langages de programmation.*

4.8 Divers

4.8.1 Fonctions predéfinies

Etant un langage de programmation, ELAN contient des structures des donnés prédéfinies. Au moment de la conception et du développement du langage, nous avons essayé de diminuer au maximum le nombre de constructions câblés. Nous sommes partis du principe que le langage doit être suffisamment expressif pour ne pas avoir besoin de structures prédéfinies. Nous avons donc limité les objets prédéfinis aux nombres entiérs et aux identificateurs, qui sont des notions de base et qui correspondent aux unités lexicalles de <*number*> et <*identifier*> . Les deux sortes sont nécessaires pour pouvoir spécifier par exemple la syntaxe des définitions des signatures où on a besoin de définir de nouveau symboles de fonctions et leurs arités. Les deux sortes built-ins sont munies des opérations habituelles.

La sorte boolean est entièrement spécifiée en ELAN comme un module ordinaire, mais qui doit quand même avoir un lien avec l'interpréteur pour pouvoir utiliser les fonctions de la sorte boolean lors de l'application des règles conditionnelles.

Après avoir réalisé l'interpréteur d'ELAN et lorsque nous avons programmé la première bibliothèque, il nous a semblé intéressant d'incorporer quelques fonctions prédefinies pour des raison d'efficacité. Ainsi, nous avons implanté des fonctions pour tester l'égalité entre deux termes, la relation sous-terme et le remplacement d'un sous-terme par un autre.

Les objets câblés dans l'interpréteur ELAN sont donc:

les constantes *true* **et** *false* et la liaison entre la constante *true* et l'évaluation avec succès des conditions des règles de réécriture (cf. 5.1.1).

la sorte number représentant les entiers et correspondant à l'unité lexicale <*number*> . De plus, les fonctions PLUS, MINUS, TIMES, DIV, INTEQUAL, INTNONEQUAL, INTLESS, INTLESSOREQ, INTGREATER, INTGREATEROREQ sont aussi câblées dans l'interpréteur d'ELAN. Néanmoins il est recommandé de les utiliser uniquement par l'interface du module standard `int` qui définit des expressions numériques d'une façon plus agréable (cf.5.1.2).

la sorte t_identifier représentant un identificateur quelconque. Il est munie des relations "IDEQUAL,IDNONEQUAL" correspondant au test d'égalité. De même que précédemment il est recommandé d'utiliser la sorte t_identifier uniquement par l'interface produit par le module standard `identifier` (cf.5.1.3).

les fonctions *eq*, *occur* **et** *replace* réalisant respectivement le test d'égalité, le test de sous-terme et le remplacement de toutes les occurrences d'un sous-terme dans un terme par un autre terme (cf.5.1.4).

la *stratégie de normalisation* qui dirige l'application des règles sans noms et qui est appliquée après chaque application réussie d'une règle dirigée par une stratégie définie par l'utilisateur (cf.4.3).

4.8.2 Appel d'un résolveur de contraintes extérieur

Un programme ELAN peut appeler un programme extérieur. Le programme est a priori considéré comme un résolveur des contraintes qui prend un terme et retourne un ensemble de termes (les formes résolues). Il est exécuté comme un sous-processus de l'interpréteur d'ELAN et il est lié avec l'interpréteur par deux *pipes* UNIX. L'un sert à transmettre les contraintes de départ au processus et l'autre récupère les résultats. Ils sont respectivement attachés aux descripteurs d'entrée et de sortie standards du processus (stdin et stdout).

Il est souhaitable que le même processus soit utilisé plusieurs fois pour résoudre des contraintes différentes, toutefois certains programmes ne savent résoudre qu'un seul problème par appel. Ainsi, dans la description de l'appel d'un processus nous donnons le nombre de problèmes qui peuvent lui être envoyé avant de le tuer et de le remplacer par autre.

Pour le processus, aussi bien les termes qu'il reçoit que les résultats qu'il retourne sont sous forme de texte. La forme textuelle des termes est donnée par la signature du module où l'appel du processus est décrit.

L'appel d'un processus est décrit par l'utilisation des stratégies **dccall** et **dkcall** (qui sont des abréviations de *dont care call* et *dont know call*) ayant trois arguments. Le premier argument est le nom du processus, le deuxième est le nombre de problèmes qui peuvent être transmis au processus et le troisième est la sorte des résultats du processus. La syntaxe des stratégies définissant l'appel d'un processus extérieur est:

<*strategy*> ::= **dccall(** <*identifier*> **,** <*number*> **,** <*sort name*> **)**
| **dkcall(** <*identifier*> **,** <*number*> **,** <*sort name*> **)**

La différence entre les deux appels est que la stratégie dkcall retourne tous les résultats obtenus du processus, tandis que la stratégie dccall ne retourne que le premier, les autres résultats étant ignorés.

Pour pouvoir communiquer avec ELAN le processus de résolution des contraintes doit satisfaire certaines restrictions syntaxiques. Sa sortie pour un problème doit avoir la syntaxe suivante:

<processus output> ::= { # }$^+$ *<next solution>*

<next solution> ::= *<number>* { * }$^+$ *<term>* { # }$^+$ *<next solution>*
| **END** { # }$^+$

où *<term>* est un résultat. De la même façon le processus doit accepter la syntaxe de passage de contraintes initiales, qui est

<processus input> ::= *<term>* *<next solution demand>*

<next solution demand> ::= **.** *<next solution demand>*
| **;**

où *<term>* est la contrainte à résoudre. Le scénario d'une communication normale est qu'ELAN fournit une contrainte à résoudre au sous-processus et qu'à chaque fois qu'il a besoin d'une solution il envoie le symbole "**.**" et attend la réponse du processus sur sa sortie standard. La réponse doit être dans la syntaxe correspondant au non-terminal *<next solution>* . Si ELAN n'a plus besoin de solutions où si elles sont épuisées, il envoie le symbole "**;**", qui provoque le passage au problème suivant (ou "end of file" si la limite d'utilisation du processus est atteinte).
Remarque: la synchronisation entre les processus étant faite par les *pipes*, les deux processus doivent ré-initialiser leurs *buffers* après chaque sortie importante. Par conséquent le processus appelé doit exécuter la procédure fflush(stdout) après la sortie de chaque solution.

Imaginons par exemple, que nous voulions utiliser la commande de shell "sort" *pour ordonner une liste d'entiers. Le premier pas est la création d'un fichier avec le simple shell-script qui fait appel à la commande* sort *(avec une option -n, parce qu'il s'agit de valeurs numériques) et qui suit les conventions requises par* ELAN. *Nous appellerons ce fichier "elansort" et il contiendra les commandes:*

```
echo "## 0 **"
sort -n
echo "END#"
```

Puis il faut adapter également la syntaxe des entrées et des sorties des listes d'entiers. La commande sort *peut être utilisée uniquement pour ordonner des lignes d'un fichier. Pour cela il faut changer la syntaxe des listes d'entiers pour que chaque entier soit sur une ligne différente. Après quoi nous appelons tout simplement notre processus "elansort". Le module qui réalise cette interface est:*

```
module intListSort
import global int list[int];
op local
    @ '\10' @          : ( int list[int] ) list[int]          alias @.@:
endop
```

strategy *sortIntList*
  dccall*(elansort,1,list[int])*
end of strategy
end of module

Maintenant si nous appliquons la stratégie sortIntList *sur une liste d'entiers, par exemple la liste* 1.5.4.6.*nil, nous obtiendrons comme résultat la liste ordonnée, donc* 1.4.5.6.*nil.*

4.9 Utilisation d'ELAN sous UNIX

Pour pouvoir utiliser ELAN sous UNIX il faut d'abord ajouter le chemin vers l'installation d'ELAN dans la variable d'environnement PATH. Puis il faut définir la variable d'environnement ELANLIB sur le chemin vers la bibliothèque des modules standards.

Les suffixes standard des fichiers d'ELAN sont:

.eln pour un fichier contenant un module

.lgi pour un fichier contenant une description d'un langage de programmation logique

.spc pour un fichier contenant une spécification (programme dans le langage spécifié).

Un appel d'ELAN est fait par une ligne de commande de la forme:

```
elan [options] <LPL_description_file> <specification_file>
```

où le `<LPL_description_file>` est le nom du fichier avec la description d'un langage de programmation logique et le `<specification_file>` est le nom du fichier avec le programme écrit dans ce langage.

Les options sont:

-w	Suppression des messages d'avertissement (*warning*).
-q	Régime *quiet* d'exécution. Dans ce régime aucune trace d'exécution n'est produite. Seuls les résultats sont affichés.
-b	Régime *batch* d'exécution. Encore plus silencieux que *quiet*. Ce régime est destiné à permettre l'utilisation d'ELAN par d'autres logiciels.
-d	Option *dump*. Cette option provoque l'affichage de la signature, de toutes les règles de réécriture et toutes les stratégies lu par l'interpréteur. L'affichage est effectué juste après avoir lu tous les modules, c'est-à-dire avant l'entrée du terme de requête.
-t <*number*>	Régime *trace* d'exécution. Le nombre après l'option $-t$ signifie la profondeur des traces (2 par défaut). Si aucun nombre n'est pas donné les traces sont effectuées dans la profondeur maximale.
-l <*library*>	Définitions d'une bibliothèque de modules ELAN personnelle. Si un module n'est pas trouvé dans le répertoire actuel, il est cherché d'abord dans la bibliothèque personnelle (.. par défaut) et puis dans le répertoire ELANLIB.

Après avoir tapé la ligne de commande, l'interpréteur d'ELAN lit tous les modules concernés et demande le terme de la requête. L'utilisateur devrait alors entrer sa requête terminiée par le mot clé **end** pour que l'interpréteur lance l'exécution.

Après avoir affiché tous les résultats l'interpréteur revient de nouveau à la demande d'un autre terme de requête. Si l'utilisateur ne souhaite plus d'exécution, il peut entrer le symbole `^D` (Control D) et la session se termine.

5
Bibliothèque ELAN

La bibliothèque d'ELAN est un ensemble de modules écrits en ELAN et offrant des constructions souvent utilisées dans les programmes. Elle contient des modules faisant l'interface avec les fonctions câblés et elle offre l'implantation de quelques unes des structures de données génériques les plus utilisées. Mais le plus grand apport de cette bibliothèque est représenté par un ensemble de modules fréquemment utilises pour le développement de résolveurs de contraintes et pour le codage de différentes logiques objet. A cette catégorie appartiennent les modules permettant de spécifier plusieurs types de signatures et les modules définissant les termes ainsi que quelques modules de résolution de contraintes.

La bibliothèque actuelle a été développée puis améliorée pendant les deux années d'évolution de l'interpréteur d'ELAN. Elle représente ainsi une partie non négligeable des travaux présentés dans la thèse.

Le chapitre présente le texte intégral de la plus grande partie de modules de la bibliothèque. Elle présente ainsi des exemples de programmes ELAN qui illustrent des constructions du langage introduites dans le manuel de référence. Dans les exemples des modules présentés dans ce chapitre les mots clés sont en gras et lorsque le texte d'un module n'est pas présenté dans son intégralité les parties omises sont représentées par trois points verticaux.

5.1 Interfaces des objets predéfinis

Nous présentons dans cette section les modules de la bibliothèque qui réalisent l'interface pour les fonctions et les sortes câblés d'ELAN. Les sortes câblées dans l'interpréteur ELAN sont: la sorte t_identifier représentant les constantes construites par n'importe quelle identificateur et la sorte number représentant les constantes numériques. La sorte number est munie des opérateurs arithmétiques usuels. Nous présentons aussi dans cette section une description de la sorte boolean introduisant les valeurs booléennes. Les opérateurs de cette sorte ne sont pas câblées, mais ils doivent quand même être liés à l'interpréteur d'ELAN à cause des conditions de règles de réécriture conditionnelles. La description de quelques fonctions génériques facilitant en pratique la programmation en ELAN est aussi incluse dans ce chapitre.

5.1.1 Booléens

Le module `boolean` définit des constantes et des opérateurs de la sorte boolean. Il définit les constantes $true$ et $false$ et les opérateurs *and*, *or* et *not*, avec leurs formes parenthésées et des règles de réécriture pour l'évaluation des expressions booléennes bâties par ces opérateurs.

Le module illustre aussi la façon dont la liaison entre les structures câblés dans l'interpréteur d'ELAN et les opérateurs utilisés dans les programmes est effectuée. Le module `boolean` définit les constantes $true$ et $false$ comme des constantes ordinaires, mais il force la représentation interne des deux constantes en utilisant l'option **code**.

La définition complete du module `boolean` *est:*

```
module boolean
sort boolean;
op
global
  true          : boolean                       code 1;
  false         : boolean                       code 2;
  @ and @       : ( boolean boolean ) boolean   assocLeft pri 100;
  @ or @        : ( boolean boolean ) boolean   assocLeft pri 100;
  (@ and @)     : ( boolean boolean ) boolean   alias @ and @:;
  (@ or @)      : ( boolean boolean ) boolean   alias @ or  @:;
  not(@)        : ( boolean ) boolean           pri 200;
  not @         : ( boolean ) boolean           pri 200          alias not(@):;
endop

rules for boolean
declare x:boolean;
bodies
 []   true and x      => x        end
 []   x and true      => x        end
 []   false and x     => false    end
 []   x and false     => false    end
 []   true or x       => true     end
 []   x or true       => true     end
 []   false or x      => x        end
 []   x or false      => x        end
 []   not true        => false    end
 []   not false       => true     end
end of rules

end of module
```

5.1.2 Entiers Relatifs

Les deux modules `builtinint` et `int` représentent une interface d'utilisation pour la sorte câblée `number` exprimant les nombres entiérs. Chaque nombre (séquence de chiffres) correspondant à l'unité lexique <*number*> peut être considéré par l'interpréteur d'ELAN comme un terme de la sorte `number`.

Le module `builtinint` définit la forme syntaxique des fonctions câblées pour les nombres entiérs. Ces fonctions ne sont applicables que sur les nombres, c'est-à-dire sur les constantes de la sorte `number`. Pour introduire les expressions arithmétiques habituelles au niveau des termes, nous introduisons une nouvelle sort `int`. Cette sorte munie des opérations habituelles est définie dans le module `int` qui importe le module `builtinint` pour évaluer les expressions.

Le module `builtinint` définit donc la forme syntaxique des fonctions câblées applicables sur les nombres entièrs. Il contient uniquement la définition d'une signature où les liens avec les fonctions câblées sont effectués par l'option **code**.

La signature définie par le module `builtinint` *est:*

```
  PLUS(@,@)     : ( number number ) number      code 3;
  MINUS(@,@)    : ( number number ) number      code 4;
  TIMES(@,@)    : ( number number ) number      code 5;
  DIV(@,@)      : ( number number ) number      code 6;

  INTEQUAL(@,@)        : ( number number ) boolean    code 8;
  INTNONEQUAL(@,@)     : ( number number ) boolean    code 9;
  INTLESS(@,@)         : ( number number ) boolean    code 10;
  INTLESSOREQ(@,@)     : ( number number ) boolean    code 11;
  INTGREATER(@,@)      : ( number number ) boolean    code 12;
```

```
INTGREATEROREQ(@,@) : ( number number ) boolean code 13;
```

Le module `int` définit les expressions arithmétiques dans leurs syntaxe habituelle en utilisant les possibilités syntaxiques d'ELAN. Le module définit aussi pour chaque opérateur un alias exprimant sa version parenthésée. Ceci permet l'utilisation habituelle de parenthèses dans les expressions arithmétiques.

La signature définie par le module `int` *est:*

```
@           : ( number ) int;

@ + @       : ( int int ) int        pri 500 assocLeft;
@ − @       : ( int int ) int        pri 600 assocLeft;
@ ∗ @       : ( int int ) int        pri 700 assocLeft;
@ / @       : ( int int ) int        pri 800 assocLeft;
− @         : ( int ) int            pri 900;

(@ + @)     : ( int int ) int        alias @ + @    :;
(@ − @)     : ( int int ) int        alias @ − @    :;
(@ ∗ @)     : ( int int ) int        alias @ ∗ @    :;
(@ / @)     : ( int int ) int        alias @ / @    :;
(− @)       : ( int ) int            alias − @      :;


@ == @      : ( int int ) boolean;
@ != @      : ( int int ) boolean;
@ <  @      : ( int int ) boolean;
@ <= @      : ( int int ) boolean;
@ >  @      : ( int int ) boolean;
@ >= @      : ( int int ) boolean;

max(@,@)    : ( int int ) int;
@ ?max @    : ( int int ) int        pri 400 assocLeft alias max(@,@):;
```

En plus de cette signature le module `int` définit des règles de réécriture qui évaluent les expressions arithmétiques en utilisant les fonctions du module `builtint`. Pour chaque opérateur définie dans le module `int`, il y a donc une règle de réécriture faisant appel à une fonction d'évaluation correspondante.

Le module contient donc les règles de réécriture de la forme:

```
rules for int
declare n1,n2:number;
bodies
  [] n1 + n2     => PLUS(n1,n2)          end
  [] n1 − n2     => MINUS(n1,n2)         end
                 ⋮
end of rules
```

Remarquons que les fonctions du module `builtinint` câblées en ELAN sont évaluées immédiatement après leurs apparitions dans le terme. Il faut donc être sûr que les arguments d'une telle fonction ne soit pas des *expressions* mais effectivement des *constantes* de la sorte number. Ce qui est garanti, dès lors que les règles d'évaluation des expressions ne sont appliquées que dans le cas où leurs arguments sont de la sorte number (cette restriction est forcée par la sorte des variables utilisées dans les règles).

Par exemple dans un terme $2 + f(3)$ la règle de réécriture pour l'opérateur + sera applicable uniquement quand la fonction $f(3)$ sera évaluée sur un entier, ce qui empêche la fonction $PLUS$ d'être appliquée sur des valeurs non appropriées.

5.1.3 Identificateurs

Une autre sorte câblée dans l'interpréteur d'ELAN est la sorte t_identifier correspondant à l'unité lexicale *<identifier>* et représentant n'importe quel identificateur. Tous les identificateurs peuvent donc être considérés comme des constantes de la sorte t_identifier.

Par analogie avec l'implantation des nombres nous utilisons deux modules hiérarchiquement liés pour définir l'interface de cette sorte. Le module `builtinident` définissant les opérateurs câblés pour la sorte t_identifier et le module `identifier` définissant une nouvelle sorte identifier, celle des expressions avec des identificateurs. Mais la situation avec les identificateurs est un peu plus difficile que pour les entiers, puisque les noms des variables dans les règles de réécriture sont également des identificateurs. L'utilisation des variables de la sorte t_identifier provoque ainsi une ambiguïté. Pour chaque variable x de la sorte t_identifier, le terme x peut être considéré comme une variable, mais dès que son nom est un identificateur, il peut être considéré aussi comme une constante de la sorte t_identifier.

Si nous avions défini un opérateur caché de coercion de la sorte t_identifier vers la sorte identifier, le même problème serait apparu au niveau des expressions de la sorte identifier. Pour cette raison, à la différence de l'injection de la sorte sortes number en int, nous avons défini ici un constructeur visible, exprimé par des doubles guillemet. De cette façon, en utilisant uniquement la sorte identifier, aucun conflit syntaxique n'apparaît, puisque la seule possibilité de former un terme valide avec des identificateurs est de les entourer par des doubles guillemet.

> *Le module* `identifier` *définit donc la sorte* identificateur *et il contient la déclaration de l'opérateur de coercion et des opérateurs de comparaison des identificateurs: égalité et non égalité. Il définit donc la signature:*

```
" @ "        : ( t_identifier ) identifier;

@ == @       : ( identifier identifier ) boolean;
@ != @       : ( identifier identifier ) boolean;
```

Malgré tout, parfois nous avons besoin de définir une signature où les nouveaux identificateurs ne sont pas entre les doubles guillemets. Par exemple si nous définissons la syntaxe d'un langage de programmation où nous avons besoin d'utiliser des nouveaux identificateurs, nous ne voulons pas contraindre leurs apparitions entre les doubles guillemets. Pour ces cas, nous avons défini dans la bibliothèque un module appelé `anyidentifier` qui définit la forme cachée de la coercion.

> *Le module* `anyidentifier` *contient une seule déclaration:*

```
@    : ( t_identifier ) identifier                alias " @ ":;
```

L'utilisation de ce module à l'intérieur d'un programme provoque des conflits syntaxiques entre les variables et leurs noms dans les corps des règles de réécriture. Mais si nous n'utilisons ce module que dans le fichier de description d'un langage de programmation il n'y a pas de conflits à priori, et ce module permet d'utiliser les identificateurs sans problème. Le mécanisme d'aliasing nous permet alors d'utiliser la syntaxe "ouverte" (sans doubles guillemets) des identificateurs à l'extérieur du programme, et la syntaxe "fermée" (dans les doubles guillemets) à l'intérieur des modules ELAN en évitant ainsi les conflits syntaxiques.

5.1.4 Fonctions génériques câblés

Les trois modules `eq`, `occur` et `replace` réalisent une interface pour les fonctions génériques câblées:

eq - réalisant le test d'égalité entre deux termes.

occurs - teste si un terme contient un autre terme comme sous-terme

replace - remplaçant toutes les occurrences d'un sous-terme dans un terme par un autre terme.

Le module `eq` est paramétré par la sorte X des deux termes, qu'il faut comparer. Il déclare donc la fonction d'égalité entre deux termes de sorte X.

Le texte suivant présente le module `eq` *et donne également un exemple illustrant la syntaxe des modules paramétrés en* ELAN.

```
module eq[X]
import boolean;
sort X;
op
  global

    @ == @     : ( X X ) boolean                code 18;
    @ != @     : ( X X ) boolean;

endop

rule for boolean
declare e1,e2 : X;
body
  e1 != e2        => not (e1 == e2)
end of rule

end of module
```

Les modules `occur` et `replace` sont paramétrés par deux sortes X et Y dont la signification est évidente à partir de déclarations suivantes.

Le module `occur` *contient la définition:*

```
    occurs @ in @          : ( X Y ) boolean      code 17;
    occurs(@,@)            : ( X Y ) boolean      alias occurs @ in @ :;
```

Le module `replace` *contient la définition:*

```
    replace @ by @ in @    : ( X X Y ) Y          code 16;
    replace(@,@,@)         : ( X X Y ) Y          alias replace @ by @ in @ :;
```

Remarquons que les trois fonctions prennent en compte les classes d'équivalences définies par la signature. Par exemple, l'expression "$occurs\ a + b\ in\ a + c + b$" est évaluée en *true* pour un symbole + déclaré associatif-commutatif.

5.2 Structures des donnés standards

Outre les sortes câblées dans l'interpréteur, nous pouvons bien sûr définir de nouvelles structures de données. Le langage ELAN a toutes les capacités pour définir des structures de données comme des types abstraits de données, c'est-à-dire en spécifiant les opérateurs de construction et les opérateurs d'accès au type de données spécifié. La possibilité de définir des modules paramétrés et celle d'utiliser les noms de sortes composées permettent de définir facilement des types de données paramétrés. De plus avec le mécanisme d'évaluation fondé sur les systèmes de calcul nous pouvons définir des opérateurs non-déterministes, donnant plusieurs résultats, par exemples l'opérateur *element* pour énumérer l'ensemble de tous les éléments d'une structure de donnée.

Dans cette section, nous illustrons sur deux exemples de la bibliothèque comment on peut définir des structures de donnés paramétrés en ELAN. Il s'agit notamment des modules définissant des couples, et des listes.

Notons que les modules présentés à partir d'ici sont entièrement définis en langage ELAN lui même et ils n'ont aucun support spécial dans son interpréteur.

5.2.1 pair

Le module `pair` définit des couples sur des sortes X et Y, où X et Y sont des arguments effectifs du module. Un couple de termes x et y est représenté par un terme $[x, y]$ de la sorte pair[X,Y] muni de deux opérateurs de sélection des éléments, "*first*" et "*second*".

Le texte du module `pair` *illustrant une définition d'une simple structure de donnés générique est:*

```
module pair[X,Y]
sort X Y pair[X,Y];
op
  global

    [@,@]        : ( X Y ) pair[X,Y];

    first(@)     : ( pair[X,Y] ) X;
    second(@)    : ( pair[X,Y] ) Y;

endop

rule for X
declare x : X;
        y : Y;
body
  first([x,y])     => x
end of rule

rule for Y
declare x : X;
        y : Y;
body
  second([x,y])    => y
end of rule

end of module
```

5.2.2 list

D'une façon similaire nous définissons la sorte list[X] des listes dont les éléments sont de sorte X. Le module `list` définit dans sa signature deux constructeurs de listes; *nil* pour la liste vide et l'opérateur binaire *cons* pour la concaténation d'un élément à une liste. L'opérateur de concaténation peut être utilisé aussi en plusieurs formes infixées. Le module définit aussi les fonctions suivantes:

$append(l_1, l_2)$ pour attacher une liste l_2 à la fin de la liste l_1.

$ccat(n, l)$ qui retourne la liste l attachée n fois à elle même.

$elem(l)$ utilisé avec la stratégie *extract* pour énumérer tous les éléments d'une liste l (voir aussi 2.3).

$i - th\ elem(l)$	qui retourne i-ème élément de la liste l
$size(l)$	donnant le nombre d'éléments de la liste l

Le module `list` *définit donc la signature:*

```
nil           : list[X];
cons(@,@)     : ( X list[X] ) list[X];

nil_X         : list[X]                       alias nil:;
@ @           : ( X list[X] ) list[X]         alias cons(@,@):;
@ , @         : ( X list[X] ) list[X]         alias cons(@,@):;
@ . @         : ( X list[X] ) list[X]         alias cons(@,@):;

append(@,@)   : ( list[X] list[X] ) list[X];
@ '@' @       : ( list[X] list[X] ) list[X]   assocRight alias append(@,@):;
ccat(@*@)     : ( int list[X] ) list[X];

elem(@)       : ( list[X] ) X;
@-th elem(@)  : ( int list[X] ) X;

size(@)       : ( list[X] ) int;
```

Remarquons, pour toutes les définitions des structures de données génériques, que le simple mécanisme de paramétrisation d'ELAN (réalisé par un pré-processeur textuel) avec l'utilisation de noms de sortes composés permet d'utiliser les compositions d'importations de modules génériques. Nous pouvons ainsi importer par exemple le module `pair[int,list[int]]` pour définir la sorte **pair[int,list[int]]** des couples composés d'entiers et de listes d'entiers.

5.3 Bibliothèque de support pour le codage des logiques du premier ordre

La possibilité de coder différentes logiques (que nous appelons les *logiques objet*) était un des objectifs principaux du développement du langage ELAN. Les logiques du premier ordre sont bien sur une première source d'intérêt. Dans ce cas, plusieurs notions, comme par exemple les définitions de signatures, les notions de termes et d'atomes construits sur une signature donnée sont partagées par des logiques différentes. Nous offrons alors à l'utilisateur une bibliothèque de modules définissant la plupart de ces notions communes. Une bibliothèque de support pour des logiques d'ordre supérieur est aussi envisagée pour le futur, mais elle dépasse le cadre de cette thèse.

Pour ne pas confondre dans le texte suivant les notions de logique spécifiée avec des notions similaires d'ELAN nous ajoutons l'adjectif *objet* aux noms de ces notions. Nous appelons donc dans cette section les termes de la logique objet les *termes objet* et les termes du système de calcul défini les *termes* ELAN ou les termes tout court. De la même façon nous utilisons la notion de *variables objet* par opposition aux variables ELAN.

Dès lors que l'un des buts de notre projet était de pouvoir spécifier des logiques objets indépendamment de leurs versions syntaxiques (syntaxe préfixée, infixée, mixfixée) et puisque nous voulons garantir la plus grande réutilisation possible des modules déjà écrits, nous avons défini pour chaque notion spécifiée deux écritures, que nous appelons respectivement la *syntaxe abstraite* et la *syntaxe concrète*. La syntaxe abstraite est utilisée à l'intérieur des modules ELAN définissant par exemple un algorithme de résolution, tandis que la syntaxe concrète peut être utilisée à l'extérieur des modules (au niveau de la description d'un langage) pour définir la syntaxe concrète d'une logique (ou d'un langage de programmation). Un changement au niveau de la syntaxe concrète ne provoque alors pas de changement au niveau des modules décrivant l'évaluation d'une requête, puisque les modules sont fondés sur la syntaxe abstraite des termes. Néanmoins, les deux définitions doivent être très proches (elle doivent représenter les mêmes termes ELAN) et elles diffèrent seulement dans leurs écritures. La liaison entre les deux est effectuée en utilisant le mécanisme d'*aliasing* d'ELAN.

La structure de la bibliothèque de support pour le codage des logiques objets est la suivante:

- Les spécifications de la syntaxe et la sémantique des définitions de signatures objet sont présentées dans les modules `signatures` et `signaturesFunctions`. Ces modules spécifient pour l'instant les signatures sans sortes en version préfixée et les signatures multi-sortées dans leurs versions pré et mixfixées. La bibliothèque est conçue de telle sorte que l'enrichissement par d'autres possibilités ne doit pas poser de graves problèmes.
- L'ensemble des termes objet est construit à l'aide de cinq modules `term_v`, `term_f`, `extTerm_v`, `extTerm_f` et `termCommons`. Ces modules utilisent les systèmes de calcul définis dans le module `signaturesFunctions`, pour générer les termes objets à partir des définitions des variables et des symboles de fonction objet. De façon similaire sont définis les modules `atom`, `extAtom` et `atomCommons` pour générer les atomes objet à partir des définitions des symboles de prédicats. Tous ces modules sont conçus de telle sorte qu'il puissent être utilisés pour n'importe quel type de signature (du premier ordre) mentionné ci-dessus.
- La notion de sorte objet ainsi que des fonctions de typage de termes sont définies dans les modules `typeCommons`, `type`, `typecheckCommons`, `typecheck_v` et `typecheck_f`.
- La notion de substitution est spécifiée dans les modules `substitution` et `applysubston`.
- Les algorithmes d'unification syntaxique et commutative sont spécifiés à l'aide des modules `eqSystem`, `syntacticUnification` et `commUnification`.

La bibliothèque actuelle résulte de plusieurs approximations réalisées avec plusieurs versions du langage ELAN. Mais, même si elle est relativement sophistiquée, son plan est malgré tout le résultat d'un choix. Plusieurs notions pouvaient être définies différemment (probablement mieux) pour faciliter la description des algorithmes possibles. Nous voulons insister sur le fait que cette bibliothèque est entièrement programmée dans le langage ELAN lui-même, c'est-à-dire qu'elle n'utilise pas de notions prédéfinies spéciales. Par conséquent, si une autre représentation des signatures et des termes objet, plus appropriée aux besoins futurs est développée, aucun changement ne sera nécessaire ni au niveau du langage ELAN ni au niveau de son interpréteur. Le seul changement concernera la bibliothèque elle même.

5.3.1 Signatures

Les deux modules décrits dans cette section fournissent un support pour le codage d'une théorie objet de la forme (Σ, Γ) dans un système de calcul en spécifiant la forme des signatures objet Σ. L'idée de la bibliothèque est d'offrir à l'utilisateur une bibliothèque assez large de signatures objet, telle que signatures sans sortes, multi-sortées ou ordo-sortées, pour que l'utilisateur puisse choisir une signature correspondant à ses besoins.

Les modules `signatures` et `signaturesFunctions` spécifient respectivement la syntaxe concrète de signatures Σ et les fonctions d'accès aux informations contenues. Une signature objet est représentée par un terme ELAN, ainsi par exemple un ensemble de symboles de fonction avec leurs arités peut être représenté par un terme ELAN de la sorte list[pair[identifier,number]] où les noms de symboles sont représentés par des identificateurs et leurs arités sont données par des nombres.

Dans cette section nous présentons en détail les parties des deux modules consacrées aux signatures préfixées sans sortes. Les autres cas sont réalisés de façon similaire et nous les présentons seulement brièvement.

Le module `signatures` définit donc les formes syntaxiques concrètes pour les définitions des signatures Σ. Il ne définit pas de structures de donnés nouvelles, mais seulement une autre forme syntaxique pour des structures de données existantes. Par exemple, la définition d'un symbole de fonction f d'arité deux avec la syntaxe concrète "f : 2" est représentée par un terme de la sorte pair[identifier,number]. Seule la syntaxe concrète utilisée pour le constructeur de couples est différente. Pour les signatures sans sortes, le module `signatures` définit donc une nouvelle syntaxe pour la sorte list[identifier] utilisée pour définir un ensemble de variables et pour la sorte list[pair[identifier,number]] utilisée pour définir un ensemble de symboles de fonctions.

Le module `signatures` *contient donc les définitions:*

```
module signatures
import list[identifier] pair[identifier,number] list[pair[identifier,number]] ··· ;
op
global

  @ ':' @      : ( identifier number ) pair[identifier,number]            alias [@,@]:;

  ;            : list[identifier]                                         alias nil:;
  @   @        : ( identifier list[identifier] ) list[identifier]         alias cons(@,@):;

  ;            : list[pair[identifier,number]]                            alias nil:;
  @   @        : ( pair[identifier,number] list[pair[identifier,number]] )
               list[pair[identifier,number]]                              alias cons(@,@):;

                 ⋮

endop

end of module
```

Nous pouvons donc écrire par exemple le terme

```
f : 2   g : 1   a : 0  ;
```

de la sorte list[pair[identifier,number]] qui peut être considéré comme une définition d'une signature objet de trois symboles f, g et a d'arités respectives: deux, un et zéro.

Dans le cas des signatures multi-sortées, une signature est représentée par une liste de couples de noms de symboles et de leurs *profils*. Un nom de symbole est soit un identificateur dans le cas d'une signature préfixée, soit une chaîne de caractères pour une signature mixfixée. Un profil est un couple représentant le domaine et le codomaine du symbole, le domaine étant représenté par une liste de types (c.f. 5.3.4). Une définition d'une signature multi-sortée préfixée est donc un terme de la sorte list[pair[identifier,pair[list[type],type]]] et une définition d'une signature multi-sortée mixfixée est un terme de la sorte list[pair[list[sdchar],pair[list[type],type]]] où list[sdchar] est une sorte utilisée pour exprimer les noms des symboles mix-fixés et qui est définie dans le module `mixfixSymNames` de la bibliothèque. Le module `signatures` définit des syntaxes concrètes pour les deux sortes. Ainsi nous pouvons écrire par exemple le terme:

```
f : s1 s2 --> s2     g : s1 --> s1     a: --> s1  ;
```

de la sorte list[pair[identifier,pair[list[type],type]]] pour définir une signature objet multi-sorté préfixé définissant les symboles de fonctions objet symboles $f : s1 \times s2 \rightarrow s2$, $g : s1 \rightarrow s1$ et la constante a de la sorte $s1$. Dans le cas d'une signature mixfixée nous pouvons écrire par exemple le terme:

```
_ f _ : s1 s2 --> s2     g(_) : s1 --> s1     a : --> s1  ;
```

de la sorte list[pair[list[sdchar],pair[list[type],type]]] pour définir cette signature avec l'opérateur f utilisé dans la syntaxe infixée.

Après la récupération d'un terme de la définition Σ, il faut pouvoir accéder aux informations qu'il contient. Puisque du point de vue d'ELAN, Σ est un terme de sorte standard il est bien sûr possible d'utiliser les fonctions définies pour cette sorte. Par exemple on peut utiliser la fonction *second* définie sur les couples pour obtenir l'arité d'un symbole de fonction à partir de sa déclaration. Mais cette approche nous a paru insuffisamment transparente et trop rigide. Par exemple un changement de la représentation d'une signature nécessiterait alors le changement dans tous les modules ELAN utilisant cette signature. Nous avons donc décidé de définir des fonctions d'accès aux signatures qui garantiront une transparence de l'utilisation des signatures. Ces fonctions sont définie dans le module `signatureFunctions`, qui définit une nouvelle sorte symbol munie d'opérations d'accès (surchargé pour chaque sorte de Σ) aux informations contenues dans un symbole d'une signature. Par exemple la sorte symbol est munie d'un opérateur *arity* retournant l'arité de son argument.

La sorte symbol est munie aussi des opérations *domain* et *codomain* pour les signatures multi-sortées et ordo-sortées. De plus, comme nous l'avons déjà mentionné, les symboles peuvent avoir différentes formes syntaxiques pour construire les termes. Pour ces raisons, la sorte symbol est munie aussi d'un opérateur *concretes* qui donne comme résultat la syntaxe concrète pour l'écriture des termes.

Nous utilisons la sorte concretesyntax pour représenter la syntaxe concrète des opérateurs. Cette sorte et la sorte char sont définies dans le module `concretesyntax` dont nous ne présentons pas le texte pour des raisons de place.

Le module `signaturesFunctions` *contient donc entre autres les définitions:*

```
module signaturesFunctions
import global   identifier list[identifier] pair[identifier,number] list[pair[identifier,number]];
       local    boolean int signatures concretesyntax;
sort symbol;
op
  global

  @       : ( identifier ) symbol;
  @       : ( pair[identifier,number] ) symbol;

  arity(@)          : ( symbol ) int;

  concretes(@)      : ( symbol ) concretesyntax;

  domain(@)         : ( symbol ) type;
  codomain(@,@)     : ( int symbol ) type;
                    ⋮
endop


rules for int
declare id      : identifier;
        n       : number;
bodies
  [] arity(id)          => 0                end
  [] arity([id,n])      => n                end
end of rules

                    ⋮
end of module
```

5.3.2 Modules définissant les termes objet

La notion de terme est une notion fondamentale pour spécifier des logiques objets et des résolveurs de contraintes symboliques. En principe, il y a deux possibilités de coder les termes objet dans un système de calcul. Le codage *externe* (cf. 7.1) définit les termes à l'aide de structures des données standard. C'est-à-dire il représente un terme objet par un couple d'un symbole de fonction de tête et d'une liste de sous-termes.

L'autre possibilité est une injection des termes objet dans une sorte de la signature ELAN. Dans ce codage (appelé *interne*) les termes objet sont représentés par les même termes ELAN d'une sorte particulière **term**. Dans cette approche chaque symbole de fonction objet f d'arité n est représenté par un symbole de fonction ELAN f' : **term** $\times \cdots \times$ **term** $\rightarrow$ **term** d'arité n. En établissant une bijection entre les constructeurs de la sorte **term** et les symboles de fonction de Σ nous avons une correspondance bi-univoque entre les termes de la logique objet et les termes ELAN de la sorte **term**.

Actuellement la bibliothèque contient uniquement les outils de la deuxième approche, parce qu'elle est plus proche de l'intuition quant au *codage par syntaxe* (cf. 7.1.1) d'une logique objet dans un système de calcul. De plus le contrôle de formation des termes par rapport aux arités des symboles est délégué dans cette approche au niveau de l'analyse syntaxique d'ELAN, tandis que dans l'autre approche, ce contrôle doit être effectué par des fonctions spéciales.

De la même façon que les symboles de fonction objet sont codés dans la sorte **term**, les variables objet sont codés par des constantes ELAN de la sorte **variable**.

Comme nous l'avons déjà mentionné, nous spécifions deux formes syntaxiques des termes objet en ELAN. Pour chaque symbole de fonction objet, nous définissons une syntaxe concrète et une syntaxe abstraite. La syntaxe concrète des symboles est déterminée par la signature utilisée (plus précisément par la fonction *concretes* définie sur les symboles). La syntaxe abstraite est une forme habituelle préfixée des termes où le symbole de tête est suivi par une parenthèse gauche, la suite des sous-termes et par une parenthèse droite.

Les définitions des symboles pour la construction des termes sont distribués dans plusieurs modules de la bibliothèque comme suit: chaque symbole de fonction est défini dans sa forme de syntaxe abstraite dans le module `term_f`. En même temps un *alias* est défini pour ce symbole dans le module `extTerm_f`. Cet alias représente la syntaxe concrète du symbole. L'utilisation des deux formes dépend alors du contexte où nous écrivons un terme. Si nous écrivons un terme dans un contexte où seul le module `extTerm_f` est importé, nous devons l'écrire dans sa syntaxe concrète. Si par contre nous écrivons un terme dans un contexte où seul le module `term_f` est importé, nous devons l'écrire dans sa syntaxe abstraite. D'habitude les modules définissant des systèmes de calcul importent uniquement le module `term_f` et ils utilisent les termes dans leur écriture préfixée standard, tandis que le module de description d'un langage importe le module `extTerm_f` permettant ainsi à l'utilisateur d'utiliser la syntaxe concrète qui est plus commode et conventionnelle. Les variables sont définies d'une manière similaire dans les modules `term_v` et `extTerm_v`.

Pour pouvoir manipuler les termes objet dans les programmes, la sorte **term** est munie des opérations suivantes:

$head(t)$	- retournant le symbole de tête du terme t,
$i - th\ subterm(t)$	- retournant le i-ème sous-terme de t,
$isvar(t)$	- retournant $true$ si t est une variable,
$nvoc(t)$	- une fonction non-déterministe, qui énumère l'ensemble des positions du terme t qui ne sont pas celle d'une variable. Une position dans un terme est représentée par une liste d'entier. Cette fonction doit être appelée avec la stratégie $extract$,

t *at* p - retournant le sous-terme de t à la position p,

$t[t']$ *at* p - retournant le terme obtenu en remplaçant le sous-terme de t à la position p par le terme t',

$rename\_subst(l, k)$ - une fonction utilisée pour le renommage de variables. Pour une liste de variables $l = x_1, \cdots, x_n, nil$ et un entier k elle retourne une substitution (c.f. 5.3.3) $\sigma = \{x_1 \to var(k+1), \cdots, x_n \to var(k+n)\}$.

L'évaluation de toutes ces fonctions est définie par les règles de réécriture dans les modules définissant les termes objet.

Dans la suite nous présentons les textes des modules définissant les termes objet en ELAN. Nous ne présentons pas les textes entiers des modules pour des raisons de place, mais nous présentons toujours la signature entière et nous avons choisi la fonction $i - th \quad subterm$ et $head(t)$ comme des fonctions modèles, dont les définitions sont montrées. Les autres fonctions sont définies de manière similaire. Les modules définissant les termes sont:

- **Le module** `termCommons` définissant les parties de termes ne dépendant pas d'une signature Σ particulière.

  Le module contient le texte:

```
module termCommons
import global  eq[term] substitution;
       local   int boolean list[int] list[variable] signaturesFunctions;
sort variable term;
op
  global

  var(@)               : ( int ) variable;
  @                    : ( variable ) term              pri 1000;

  head(@)              : ( term ) symbol;
  @-th subterm(@)      : ( int term ) term;
  isvar(@)             : ( term ) boolean;
  nvocc(@)             : ( term ) list[int];

  @ at @               : ( term list[int] ) term;
  @[@] at @            : ( term term list[int] ) term;

  rename_subst(@,@)    : ( list[variable] int ) substitution[term];

endop
            ⋮
end of module
```

- **Le module** `term_f` définit la forme abstraite de termes construits sur un ensemble de symboles de fonction. Cet ensemble est le paramètre du module, plus précisément le module est paramétré par un module spécial (cf. 4.7) représentant la liste des symboles de fonctions. Il introduit un symbole de fonction ELAN pour chaque symbole de fonction objet de cette liste. Les symboles de fonction sont introduites dans la signature ELAN en utilisant la construction du pré-processeur FOR EACH. Le module `term_f` définit aussi des règles de réécriture pour compléter les définitions de fonctions sur la sorte `term`.

  Le module `term_f` *est définit par Rappelons que la stratégie extract en combinaison avec la fonction elem énumère tous les éléments d'une liste et que la fonction arity donne l'arité d'un symbole. Le module* `term_f` *est défini par:*

```
module term_f[fss]
import global termCommons;
       local fss int boolean signaturesFunctions list[int];
op
  global

    FOR EACH F:symbol SUCH THAT F:=(extract) elem(fss) ANDIF arity(F)==0 :{
      F          : term;
      F()        : term                                      alias F:;
    }

    FOR EACH F:symbol; N:int
    SUCH THAT F:=(extract) elem(fss) AND N:=() arity(F) ANDIF N > 0 :{

     F (@ {,@}~(N-1))  :({term}~N) term;
    }

endop


FOR EACH F:symbol;N:int SUCH THAT F:=(extract)elem(fss) AND N:=()arity(F):{
  rules for term
  declare t_1,...,t_N : term;
  bodies

   {
    [] I-th subterm(F(t_1,...,t_N))          => t_I
    end
   }_I=1...N

    rule for symbol
    body
      head(F(t_1,...,t_N))             => F
    end

  end of rules
}

          ⋮

end of module
```

- **Le module** `extTerm_f` définit la syntaxe concrète de termes. Le module doit être paramétré par le même module spécial que dans le cas du module `term_f`.
- **Le module** `term_v` définit l'ensemble des variables objet. Il est paramétré par un module spécial représentant une liste de variables. Le module définit de plus des opérateurs utilisés pour le renommage des variables dans les termes. Ce sont les opérateurs:

  varlist_vars - une constante évaluée sur la liste des variables définies,

  varnum_vars - une constante évaluée sur le nombre des variables définies.

  Le module `term_v` *contient le texte:*

```
module term_v[vars]
import global termCommons;
       local int vars signaturesFunctions list[variable];
op
  global

    FOR EACH X:symbol SUCH THAT X:=(extract)elem(vars) :{
      X      : variable;
    }

    varlist_vars               : list[variable];
```

```
    varnum_vars          : int;

endop

FOR EACH X:symbol SUCH THAT X:=(extract)elem(vars) :{
  rule for symbol
  body
    head(X)      => X
  end of rule
}
           ⋮
end of module
```

- **Le module** `extTerm_v` définit la syntaxe concrète pour les variables. Il doit être paramétré par le même module spécial que dans le cas du module `term_v`.

Nous illustrons l'effet d'utilisation de ces modules sur un exemple.

Par exemple pour une signature, non sortée préfixée, définie par les termes:

$$\mathcal{F} = cons([f,1], cons([a,0], nil))$$

$$\mathcal{V} = cons(X, cons(Y, nil))$$

ces modules construisent pour les sortes variable *et* term *la signature* ELAN *suivante:*

le module `termCommons`*:*

@	**:** *(variable) term*

le module `term_v`*[$\mathcal{V}$]:*

X	**:** *variable*
Y	**:** *variable*

le module `term_f`*[$\mathcal{F}$]:*

[f,1](@)	**:** *(term) term*
[a,0]	**:** *term*

et le module `extTerm_f`*[$\mathcal{F}$]:*

f(@)	**:** *(term) term*	**alias** *[f,1](@):*
a	**:** *term*	**alias** *[a,0]:*

5.3.3 Substitutions

La notion de substitution est étroitement liée à la notion de terme. Nous définissons dans cette section les modules qui définissent la notion de substitution sur des termes objet. Ils définissent aussi le résultat de l'application d'une substitution sur un terme. Les constructeurs de la sorte substitution sont définis dans le module `substitution`. Une substitution est construite à partir d'une constante *identity* représentant l'identité et d'un constructeur $v \to t$ pour exprimer une substitution remplaçant une variable v par le terme t. Les substitutions peuvent être composées par l'opérateur de composition $\circ$.

Le module `substitution` *est défini par:*

```
module substitution
sort substitution variable term;
op
 global

  identity       : substitution;
  @ -> @         : ( variable term ) substitution;
  @ o @          : ( substitution substitution ) substitution              assocLeft;

endop

end of module
```

Le module `applysubston` définit une opération d'application d'une substitution sur un terme de sorte `X`, qui est l'argument du module. Nous avons défini un opérateur générique à la place d'un opérateur défini uniquement sur la sorte `term` puisque nous avons souvent besoin d'appliquer une substitution sur des objets différents contenant des termes, par exemple sur les atomes, clauses de Horn, équations entre deux termes etc. Le module utilise la fonction câblé *replace* (voir sect. 5.1.4) pour définir le résultat de l'application d'une substitution sur un terme.

Le module `applysobston` *définit l'opération apply utilisable dans sa forme préfixée ou mixfixée. Il contient le texte:*

```
module applysubston[X]
import termCommons replace[term,X];
sort X;
op
  global

    apply(@,@)  : ( substitution X ) X;
    @(@)        : ( substitution X ) X            alias apply(@,@):;

endop

rules for X
declare tt      : X;
        s1,s2   : substitution;
        v       : variable;
        t       : term;
bodies
  [] apply(identity,tt)          => tt                        end
  [] apply(s1 o s2 ,tt)          => apply(s2,apply(s1,tt))    end
  [] apply(v->t,tt)              => replace(v,t,tt)           end
end of rules

end of module
```

5.3.4 Modules définissant les sortes objet

Pour coder des signatures multi-sortées la bibliothèque contient des modules définissant des sortes objet. Les sortes objet sont représentées par des constantes de la sorte ELAN `type`. Cette sorte est spécifiée à l'aide de deux modules:

- Module `typeCommons` définissant la sorte `type` et des notions indépendantes d'un ensemble de sortes objet particulier.
- Module `type` paramétré par un module spécial d'un ensemble de sortes objet. Cet ensemble est représenté par une liste d'identificateurs.

La bibliothèque contient aussi des modules réalisant des fonctions de typage pour les termes objet. Il s'agit notamment de fonctions:

$type(t)$	- qui retourne le type du terme t,
$typecheck(t)$	- condition qui retourne $true$ si le terme t est bien typé.

La définition de ces fonctions est divisée en trois modules. Le module `typecheckCommons` définit la partie ne dépendant pas d'une signature objet concrète et deux autres modules complètent ces définitions par rapport aux ensembles de symboles de fonction et symboles de variables particuliers. Il s'agit notamment des modules:

- Module `typecheck_f` paramétré par un module spécial représentant l'ensemble des symboles de fonction,
- Module `typecheck_v` paramétré par un module spécial représentant l'ensemble des variables.

5.3.5 Modules définissant les atomes objet

Les atomes sont des objets souvent utilisés dans les définitions des différentes logiques. Un atome est défini comme l'application d'un symbole de prédicat d'arité n sur n termes. L'implantation des atomes objet en ELAN est similaire à celle des termes. Les atomes objet sont injectés dans une sorte ELAN nommée atom. Pour définir les atomes, la bibliothèque ELAN contient trois modules:

- **Module** `atomCommons` définissant les notions communes pour les atomes. Il définit la sorte atom et les fonctions permettant l'accès à sa structure:

$head(a)$	retournant le symbole de prédicat de la tête de l'atome a
$i - th\ subterm(a)$	retournant le i-ème sous-terme d'un atome a.

- **Module** `atom` définissant la syntaxe abstraite des atomes. Le module est paramétré par un module spécial définissant les symboles de prédicats.
- **Module** `extAtom` définissant la syntaxe concrète des atomes. Le module est paramétré par le même module définissant des symboles de prédicats que le module `atom`.

5.3.6 L'unification syntaxique

L'unification syntaxique est un algorithme nécessaire pour le codage de mécanismes de preuves dans la plupart des logiques objet. Comme nous l'avons déjà mentionné dans l'introduction, l'algorithme d'unification est naturellement exprimable sous la forme des règles de réécriture. Il n'y a donc pas de problème principal pour l'implanter en ELAN. Nous avons par exemple implanté directement les règles d'unification syntaxique introduites dans la figure 2.1 de l'introduction.

L'implantation de l'unification syntaxique en ELAN est divisée en deux modules, le module `eqSystem` qui définit la sorte `eqSystem` de systèmes d'équations et le module `syntacticUnification` qui définit les règles de réécriture et les stratégies de l'algorithme d'unification. Un système d'équations entre des termes est construit par un constructeur commutatif d'égalité (en ELAN nous l'avons déclaré étant associatif-commutatif, parce que la théorie commutative n'est pas implantée) et un constructeur associatif-commutatif de conjonction &. En réalité les règles d'unification supposent que l'opérateur de conjonction est associatif-commutatif avec un élément neutre, mais nous pouvons contourner les problèmes provoqués par la restriction à la théorie AC en garantissant l'omni-présence de l'élément neutre dans le système.

De plus dans notre système d'équations, nous admettons les valeurs booléennes avec la signification habituelle de la constante $true$ comme un système trivialement valide et la constante $false$ comme système non satisfiable.

Le module `eqSystem` en plus des constructeurs pour la sorte `eqSystem`, définit aussi une fonction *system_to_subst* transformant un système d'équation résolu en une substitution.

Le module `eqSystem` *est:*

```
module eqSystem
import global applysubston[eqSystem];
         local boolean int termCommons;
sort eqSystem equation;
op
global

  @ = @          : ( term term ) equation                         (AC);

  @              : ( boolean ) eqSystem;
  @              : ( equation ) eqSystem;
  @ & @          : ( eqSystem eqSystem ) eqSystem                 (AC);

  system_to_subst(@)      : ( eqSystem ) substitution;

endop

rules  for substitution
declare P:eqSystem;
         t:term;
         x:variable;
bodies
  [] system_to_subst(true)          => identity                         end
  [] system_to_subst(x=t)           => x->t                             end
  [] system_to_subst(x=t & P)       => x->t o system_to_subst(P)        end
end of rules


end of module
```

Le module `syntacticUnification` définit les règles de réécriture pour la procédure d'unification. La règle *decompose* doit être engendrée pour chaque symbole de fonction défini dans un module spécial représentant l'ensemble de symboles de fonction.

Pour des raisons techniques liées à l'utilisation d'un symbole de conjonction associatif-commutatif sans élément neutre, le module contient aussi des règles supplémentaire qui gèrent les occurrences du symbole neutre (représenté par la constante $true$) dans le système.

Et finalement le module définit deux stratégies *unify* et *unifys* décrivant la transformation d'un système d'équations en sa forme résolue. La stratégie *unify* donne comme résultat soit la forme résolue d'un système soit la constante $false$ si le système n'a pas de solution. La stratégie *unifys* donne un résultat uniquement dans le cas où il y a une solution, sinon la stratégie ne donne aucun résultat. Les définitions des deux stratégies diffèrent dans la dernière ligne où la stratégie *unify* autorise l'application de la règle *falsepropag* pour la propagation du résultat $false$, tandis que la stratégie *unifys* n'autorise pas cette possibilité.

le module `syntacticUnification`*:*

```
module syntacticUnification[fss]
import  vars fss boolean int termCommons term_f[fss] eqSystem
        signaturesFunctions occur[variable,term] occur[variable,eqSystem];


rules for eqSystem
```

```
declare P     : eqSystem;
        s,t  : term;
        x,y  : variable;
bodies
  [delete]       P & s=s          => P                                    end

  FOR EACH F:symbol; N:int SUCH THAT F:=(extract)elem(fss) AND N:=()arity(F) :{
   rule decompose
   declare       s_1,...,s_N : term;
                 t_1,...,t_N : term;
   body
                 P & F(s_1,...,s_N)=F(t_1,...,t_N)         => P { & s_I=t_I }_I=1...N
   end
  }

  [conflict]     P & s=t          => false
                                     if head(s) != head(t) and not(isvar(s)) and not(isvar(t))
  end
  [coalesce]     P & x=y          => apply(x->y,P) & x=y
                                     if occurs x in P   and   occurs y in P   and   x!=y
  end
  [occ_check]    P & x=s          => false
                                     if occurs x in s and not isvar(s)
  end
  [eliminate]    P & x=s          => apply(x->s,P) & x=s
                                     if not(occurs x in s) and occurs x in P  and  not isvar(s)
  end


  [delete2]      P & true & true => P & true                              end
  [delete3]      P & false        => false                                end
  [trueelim]     P & true         => P                                    end
  [trueadd]      P                => P & true                             end
  [falsepropag]  false            => false                                end

end of rules


strategy unify
  dont care choose(trueadd)
  while
    dont care choose(delete delete2 delete3 decompose conflict coalesce occ_check eliminate)
  endwhile
  dont care choose(trueelim falsepropag)
end of strategy

strategy unifys
  dont care choose(trueadd)
  while
    dont care choose(delete delete2 delete3 decompose conflict coalesce occ_check eliminate)
  endwhile
  dont care choose(trueelim)
end of strategy

end of module
```

5.3.7 L'unification commutative

L'unification commutative décrit un résolveur de contraintes équationnelles en présence de symboles commutatifs. La théorie commutative est finitaire, c'est-à-dire l'ensemble complet des unificateurs plus généraux est fini, mais il n'est pas un singleton. Nous devons donc utiliser un algorithme non-déterministe pour évaluer les calculs possibles menant aux unificateurs appropriés. En utilisant une bonne stratégie, l'espace de recherche peut être considérablement restreint. C'est le cas de notre implantation qui montre en même temps les possibilités d'ELAN pour exprimer ce genre d'algorithme non-déterministe.

De façon similaire à l'unification syntaxique, l'algorithme d'unification commutative peut être décrit sous la forme de règles de réécriture, en ajoutant une règle appelée *ComMutate* aux règles de l'unification syntaxique. Pour un symbole f commutatif, la règle *ComMutate* est:

$$\begin{array}{rl} \textbf{ComMutate} & P \wedge f(s_1,s_2) = f(t_1,t_2) \\ \rightarrow & P \wedge s_1 = t_2 \wedge s_2 = t_1 \end{array}$$

Notons que si nous pouvons appliquer la règle *ComMutate* pour une équation, alors nous pouvons appliquer sur la même équation aussi la règle *Decompose* et nous obtenons ainsi deux systèmes d'équations avec des solutions différentes. Nous devons explorer deux arbres obtenus par les applications respectives de la règle *ComMutate* et *Decompose* sur une équation. Plus précisément l'ensemble de toutes les formes résolues d'une unification commutative est obtenue par la construction suivante:

> L'espace de recherche nécessaire à explorer pour trouver toutes les solutions est représenté par un arbre dont les noeuds sont labellés par des systèmes d'équations et les branches par des noms de règles de réécriture. La racine de cet arbre est labellée par le système d'équations de départ. Un noeud de cet arbre possède un fils unique si il était obtenu par l'application d'une des règles *Delete, Conflict, Coalesce, Check, Eliminate* ou par une application de la règle *Decompose* appliquée sur une équation avec des symboles non commutatif aux têtes des termes. Un noeud de cette arbre possède deux fils représentant les applications respectives des règles *Decompose* et *ComMutate* sur une équation avec un même symbole commutatif en tête des deux termes. Les feuilles de cet arbre sont des noeuds représentant les systèmes d'équations sur lesquels aucune règle ne peut être appliquée. Elles représentent l'ensemble correct et complet des formes résolues pour notre problème d'unification.

Pour exprimer la stratégie décrivant cet arbre, nous devons combiner deux méthodes de recherche: *dont care* pour tous les règles de l'unification syntaxique et *dont know* pour l'alternance de la règle *Decompose* et *ComMutate*. La premier idée est de définir alors la stratégie suivante:

```
repeat
  dont care choose(
    dont care choose(Delete Conflict Coalesce Check Eliminate)
  ||
    dont know choose(Decompose ComMutate)
  )
endrepeat
```

Cette stratégie nous donne l'ensemble des solutions. Mais les solutions seront calculées plusieurs fois, parce que cette stratégie ne spécifie pas que le branchement doit être effectué uniquement si les deux règles *Decompose* et *ComMutate* sont appliquées sur la *même* équation $f(s_1,s_2) = f(t_1,t_2)$ d'un système. La stratégie présentée explore toutes les possibilités c'est-à-dire aussi les cas d'applications de la même règle sur deux équations différentes, ce qui est inutile. Pour obtenir alors une stratégie plus efficace, nous avons divisé la règle *ComMutate* en deux règles différentes: la règle *ComChoose* qui sélectionne une équation (elle est utilisée de manière *dont care choose*) et la règle *ComMutate* avec la règle *ComDecompose* qui s'appliquent conjointement sur cette équation choisie. Pour marquer une équation choisie, nous avons introduit dans le système d'équations un constructeur spécial.

L'unification commutative est définie dans le module `commUnification` paramétré par deux modules spéciaux, représentant respectivement l'ensemble des symboles libres et l'ensemble des symboles commutatifs. Comme dans le cas de l'unification syntaxique, le module définit deux stratégies *communify* et *communifys* réalisant l'unification commutative. La stratégie *communify* retourne toutes les formes correspondant aux feuilles de l'arbre de recherche, y compris les noeuds d'échec, tandis que la stratégie *communifys* donne uniquement les résultats correspondant aux solutions.

le module `commUnification`*:*

```
module commUnification[fss,comfss]
import  fss comfss boolean int term_f[comfss] eqSystem
        signaturesFunctions list[eqSystem] syntacticUnification[fss];

op
  local
    @ + (@,@)=(@,@)  : (eqSystem term term term term ) eqSystem;
endop

rules for eqSystem
declare P    : eqSystem;
        s_1,s_2,t_1,t_2 : term;
bodies

  FOR EACH F:symbol SUCH THAT F:=(extract)elem(comfss) :{
    [commChoose]       P & F(s_1,s_2)=F(t_1,t_2)        => P + (s_1,s_2)=(t_1,t_2)
    end
  }
  [commDecompose]      P + (s_1,s_2)=(t_1,t_2)          => P & s_1=t_1 & s_2=t_2
  end
  [commMutate]         P + (s_1,s_2)=(t_1,t_2)          => P & s_1=t_2 & s_2=t_1
  end

end of rules


strategy communify
  dont care choose(trueadd)
  repeat
    dont care choose(
      dont care choose(delete delete2 delete3 conflict coalesce decompose occ_check eliminate)
    ||
      dont care choose(commChoose)
      dont know choose(commDecompose commMutate)
    )
  endrepeat
  dont care choose(trueelim falsepropag)
end of strategy


strategy communifys
  dont care choose(trueadd)
  repeat
    dont care choose(
      dont care choose(delete delete2 delete3 conflict coalesce decompose occ_check eliminate)
    ||
      dont care choose(commChoose)
      dont know choose(commDecompose commMutate)
    )
  endrepeat
  dont care choose(trueelim)
end of strategy

end of module
```

6
Exemples

Nous présentons dans ce chapitre quelques exemples de langages de programmation (avec leurs logiques sous-jacentes) implantés en ELAN. La sémantique opérationnelle de ces langages est réalisée par différents systèmes de déduction dans des fragments de la logique du premier ordre. Nous présentons en particulier différents mécanismes de déduction dans la logique équationelle, notamment les procédures de la surréduction basique et la réécriture. Nous présentons aussi l'implantation d'une procédure de surréduction paresseuse utilisée dans le langage BABEL [MNRA92]. Nous présentons également un langage de programmation basé sur la logique des clauses de Horn dont la sémantique est basée sur la procédure de SLD résolution.

Les exemples présentés ici utilisent la bibliothèque de modules standards présentée dans le chapitre précédent et nous illustrons sur l'exemple de la surréduction basique la possibilité d'utiliser différentes versions des signatures d'une logique.

Mis à part les exemples présentés dans ce chapitre, le système ELAN a été utilisé pour spécifier des résolveurs de contraintes équationelles d'un système d'équations et diséquations syntaxiques [Dek93] et de contraintes d'ordre *lpo* [Mor94]. Il a été également utilisé pour spécifier la procédure de complétion de Knuth Bendix avec contraintes [Mor94].

6.1 Simple PROLOG

Nous présentons tout d'abord la définition du langage appelé *simpleprolog*. Il s'agit d'un langage simple similaire à PROLOG dont les programmes sont formés par une signature et une liste de clauses de Horn. La signature est composée de trois parties: définition des variables, des symboles de fonction et des symboles de prédicat. Le langage est basée sur la logique des clauses de Horn et sa sémantique opérationnelle est réalisée par la procédure de la SLD résolution. Cette procédure et son codage dans un système de calcul ont été présenté dans l'introduction de la thèse. Nous ne répétons pas ici cette description.

Le module réalisant ce codage en ELAN est appelé `SLD_resolution` et il est paramétré par des modules représentant respectivement l'ensemble des variables, des symboles de fonction, des symboles de prédicat et l'ensemble de clauses de Horn qui sont représentées par des termes de la sorte pair[atom,list[atom]].

```
module SLD_resolution[vars,fss,preds,clauses]
import  clauses preds int boolean atom[preds] term_v[vars] term_f[fss]
        eqSystem hClauses syntacticUnification[fss] signaturesFunctions
        list[atom] pair[atom,list[atom]] list[pair[atom,list[atom]]]
        applysubston[list[atom]] applysubston[term];

sort SLD_res_state ;
op
  global

    ( @ |@| @ )  : ( list[atom] int eqSystem ) SLD_res_state;
```

```
endop

FOR EACH P:symbol;N:int SUCH THAT P:=(extract) elem(preds) AND N:=() arity(P) :{

   FOR EACH HC:pair[atom,list[atom]]; H:atom; B:list[atom]; t_1, ... ,t_N :term
   SUCH THAT HC:=(extract) elem(clauses) AND H:=()first(HC) AND B:=()second(HC)
       ANDIF head(H) == P      {AND t_I := () I-th subterm(H)}_I=1...N :{

  rule SLD_step for SLD_res_state
  declare cc,cc1          : eqSystem;
          rest            : list[atom];
          s_1,...,s_N     : term;
          sigma           : substitution;
          n               : int;
  body
   (P(s_1, ... ,s_N).rest |n| cc)       => ( sigma(B) @ rest |n+varnum_vars| cc1)
                          where cc1 := (unifys) cc {& s_I = sigma(t_I) }_I=1...N
                          where sigma := () rename_subst(varlist_vars,n)
  end of rule

}}

rule SLD_success for SLD_res_state
declare sigma1,sigma2    : eqSystem;
        n                : int;
body
  ( nil |n| sigma1)       => ( nil |0| sigma2 )
                          where sigma2 := (delete_new_variables) sigma1
end of rule

strategy SLD_resolution
  while
    dont know choose(SLD_step)
  endwhile
  dont care choose(SLD_success)
end of strategy

end of module
```

Dans notre implantation nous avons aussi ajouté une valeur numérique dans le but, qui exprime le numéro de la plus grande variable présente dans le but. Cette valeur est utilisée pour effectuer le renommage des variables de la règle de réécriture choisie sur des variables "fraîches".

Pour pouvoir spécifier un langage de programmation avec la sémantique opérationnelle basée sur la résolution SLD, nous avons besoin de définir la syntaxe concrète de la sorte pair[atom,list[atom]] représentant une clause de Horn. Cela est fait dans le module `hClause`. Ce module définit en même temps la syntaxe concrète pour la sorte list[pair[atom,list[atom]]] représentant un ensemble de clauses de Horn.

Le module `hClauses` *est défini comme suit:*

```
module hClauses
import  atomCommons list[atom] pair[atom,list[atom]]
        list[pair[atom,list[atom]]];
op
 global

   .    : list[atom]                              alias nil:;
   @,@  : ( atom list[atom] ) list[atom]          alias cons(@,@):;


   @ @          : ( atom list[atom] ) pair[atom,list[atom]]    alias [@,@]:;
   @ ':'- @     : ( atom list[atom] ) pair[atom,list[atom]]    alias [@,@]:;

   ;      : list[pair[atom,list[atom]]]           alias nil:;
```

```
   @ @   : ( pair[atom,list[atom]] list[pair[atom,list[atom]]])
             list[pair[atom,list[atom]]]           alias cons(@,@):;

endop
end of module
```

Pour compléter l'exemple du langage *simpleprolog*, il faut spécifier le fichier de description du langage. Ce fichier décrit donc la forme des programmes dans ce langage. Un programme est formé de quatre parties appelées *Vars, Ops, Predicates* et *Clauses* qui correspondent aux déclarations de variables, symboles de fonction, symboles de prédicat et clauses de Horn. Les symboles de fonction et prédicat doivent contenir l'information sur les arités de symboles. Nous utilisons une version pré-fixée de la signature, les corps des parties sont alors représentés par des termes de sortes: list[identifier] pour la déclaration des variables, list[pair[identifier,number]] pour la déclaration des symboles de fonction et de prédicat et list[pair[atom,list[atom]]] pour les clauses de Horn. Une requête est un terme de la sorte atom et elle est évaluée par une application de la stratégie *SLD_resolution* sur le terme de départ (requête$|0|true$).

Le fichier de description du langage *simpleprolog* contient alors le texte:

```
LPL simpleprolog description

  specification description

    part Vars          of sort list[identifier]
                       modules anyidentifier signatures
    part Ops           of sort list[pair[identifier,number]]
                       modules anyidentifier signatures
    part Predicates    of sort list[pair[identifier,number]]
                       module anyidentifier signatures
    part Clauses       of sort list[pair[atom,list[atom]]]
                       modules extTerm_v[Vars] extTerm_f[Ops] extAtom[Predicates] hClauses
  end

  query          of sort atom
                 result of sort SLD_res_state
                 modules SLD_resolution[Vars,Ops,Predicates,Clauses]
                 start with (SLD_resolution) (query.nil |0| true)
end of LPL description
```

Un programme dans ce langage doit avoir quatre parties correspondant à cette description. Par exemple le programme pour calculer les éléments d'une liste contient le texte:

```
specification member

  Vars
                 X   Y   Z ;
  Ops
                 cons:2  nil:0 a:0 b:0 c:0 ;
  Predicates
                 member:2 ;

  Clauses

                 member(X,cons(X,Y)).
                 member(X,cons(Y,Z)) :- member(X,Z),.
                 ;
end of specification
```

6.2 La surréduction basique

Dans cette section nous montrons comment la procédure de la surréduction basique peut être réalisée et utilisée en ELAN. Nous commençons d'abord par une brève description de cette procédure en termes d'un système de calcul et nous présentons ensuite le module qui implante ce système. Après cela nous présentons l'utilisation de ce module dans la définition d'un langage de programmation. Nous présentons deux versions de ce langage, la première version utilise la logique équationelle non-sortée avec une écriture préfixée des termes. Ensuite nous présentons l'implantation du même langage, mais en utilisant la version multi-sortée de la logique avec une écriture des termes mix-fixée.

6.2.1 La procédure de surréduction basique

La surréduction basique est un processus qui permet de trouver toutes les solutions d'une équation de la forme $g(\overline{x}) = d(\overline{x})$ où l'égalité est comprise comme l'égalité engendrée par une théorie equationelle présentée sous la forme d'un système de réécriture PgR confluent et terminant. La surréduction basique a été introduite dans [Hul80]. La présentation de cette procédure comme un système de calcul est donnée dans [KKV95]. La surréduction basique peut être formalisée en définissant deux règles de réécriture:

$$
\begin{array}{ll}
\textbf{Narrow} & (\exists X, g = d \parallel S) \\
\to & (\exists X \cup \mathcal{V}ar(l,r),\ g[r]_\omega = d \parallel S \wedge (g_{|\omega} \equiv_\emptyset l)) \\
 & \text{if } l \rightarrowtail r \in Variants(PgR) \text{ and } (S \wedge (g_{|\omega} \equiv_\emptyset l)) \text{ est satisfaisable}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{Block} & (\exists X, g = d \parallel S) \\
\to & (\exists X, \top \parallel S \wedge (g \equiv_\emptyset d)) \\
 & \text{if } (S \wedge (g \equiv_\emptyset d)) \text{ est satisfaisable}
\end{array}
$$

où nous utilisons la notation:

$g_{\mid\omega}$	pour le sous-terme du terme g sur la position ω
$g[r]_\omega$	pour le terme résultant de remplacement du sous-terme de g sur la position ω par le terme r
$\equiv_\emptyset$	pour une équation entre deux termes modulo la théorie vide
$\mathcal{V}ar(t)$	pour l'ensemble des variables du terme t
$Variants(PgR)$	pour l'ensemble des règles PgR ne contenant que de nouvelles variables
$\top$	pour une équation triviale toujours satisfaite.

L'exécution de la surréduction basique cherchant toutes les solutions d'une équation $g = d$, se déroule de la manière suivante:

L'exécution part du *but* de départ:

$$(g = d \parallel \emptyset)$$

en explorant toutes les possibilités d'application de la règle $Narrow$ sur ce but. Si la règle $Narrow$ peut être appliquée, le but actuel est remplacé par le résultat d'application de cette règle. Après chaque application réussie de la règle $Narrow$, la procédure essaye d'appliquer la règle $Block$ afin d'obtenir une solution du problème original. Si la règle $Block$ est applicable, le résultat est retourné, cette branche d'évaluation est coupée et l'évaluation continue en évaluant d'autre possibilités d'application de la règle $Narrow$.

Si nous imaginons donc les branches des calculs menant aux résultats, elles sont toutes composées de plusieurs applications de la règle $Narrow$, suivies par exactement une application de la règle *Block*. Ce qui correspond à une stratégie décrite par l'expression:

$$Narrow^{*}Block$$

Le système de calcul défini par les deux règles de réécriture $Narrow$ et *Block* et par la stratégie mentionnée au-dessus peut être facilement implanté comme un module d'ELAN paramétré par le système de réécriture PgR, c'est-à-dire par un module spécial (dans le sens de sect.4.7) définissant ce système. Le système de réécriture PgR est représenté en ELAN par une liste des couples de termes, chaque couple représentant une règle de réécriture. Pour pouvoir construire l'ensemble des termes objet, le module doit être paramétré aussi par deux modules spéciaux définissant les ensembles de symboles de fonctions et variables.

Le module décrivant la surréduction basique est:

```
module basic_narrowing[vars,fss,PgR]
import int PgR term_v[vars] term_f[fss] syntacticUnification[fss]
        pair[term,term] list[pair[term,term]] eqSystem applysubston[term];
sort narrow_state Eequation;
op
  global

    @ = @        : ( term term ) pair[term,term]           alias[@,@]:;

    ( @ |@| @ ) : ( pair[term,term] int eqSystem) narrow_state;

    top          : pair[list,list];

endop


FOR EACH R:pair[term,term]; l,r:term
SUCH THAT R := (extract) elem(PgR) AND l := () first(R) AND r := () second(R) :{
  rule narrow for narrow_state
  declare cc,c1  : eqSystem;
        g,d      : term;
        sigma    : substitution;
        occ      : list[int];
        n        : int;
  body
    (g = d |n| cc)        => ( g[sigma(r)] at occ = d |n+varnum_vars| c1)
                             where c1 := (unifys) g at occ = sigma(l) & cc
                             where occ := (extract) nvocc(g)
                             where sigma:=()rename_subst(varlist_vars,n)
  end of rule
}


rule block for narrow_state
declare SSF,SF,cc: eqSystem;
        g,d      : term;
        n        : int;
body
  (g = d |n| cc)          => (top |0| SSF )
        where SSF := (delete_new_variables) SF
        where SF := (unifys) g=d & cc
end of rule

strategy narrow
  iterate
    dont know choose(narrow)
  enditerate
  dont know choose(block)
```

```
end of strategy

end of module
```

Dans notre implantation des règles de la surréduction, nous avons omis la quantification des variables, puisque elle sert uniquement pour distinguer les variables ajoutées pendant l'évaluation des variables originales de la requête. De même que dans le cas du langage *simpleprolog* nous avons aussi ajouté une valeur numérique dans le but, qui exprime le numéro de la plus grande variable présente dans le but. Cette valeur est utilisée pour effectuer le renommage des variables de la règle de réécriture choisi sur de nouvelles variables.

6.2.2 Description d'un langage basé sur la surréduction basique

Nous définissons maintenant un langage de programmation (nous l'appelons *narrowing*) dont la sémantique opérationnelle est basée sur la surréduction basique. Les programmes dans ce langage sont formés par trois parties appelés *Vars*, *Ops*, et *Rules* contenant respectivement les déclarations de variables, symboles de fonction et système de réécriture.

Par exemple un programme définissant les règles pour la fonction append dans notre langage est le suivant:

```
specification append

Vars
        x   y   z ;
Ops
        a:0   b:0   c:0   nil:0   cons:2   append:2 ;

Rules

        append(nil,x)          -> x
        append(cons(x,y),z)  -> cons(x,append(y,z))
        ;
end of specification
```

Pour pouvoir spécifier la syntaxe d'un programme de ce langage, nous devons spécifier la syntaxe de chaque partie. Pour les définitions des variables et des symboles de fonction nous pouvons utiliser les constructions définies dans la bibliothèque notamment celles du module `signatures`. Nous représentons les règles de réécriture par des couples de termes, donc par la sorte **pair[term,term]** et pour définir la syntaxe concrète des règles de réécriture, nous définissons un module appelé `rwRules`, qui définit une syntaxe concrète représentant une règle de réécriture par la sorte **pair[term,term]** et qui définit aussi une syntaxe concrète pour la sorte **list[pair[term,term]]** qui représente un système de réécriture.

Le module `rwRules` *est le suivant:*

```
module rwRules
import termCommons pair[term,term] list[pair[term,term]];
op
 global

  @ -> @      : ( term term ) pair[term,term]            alias [@,@]:;

  ;           : list[pair[term,term]]              alias nil:;
  @ @         : (pair[term,term] list[pair[term,term]]) list[pair[term,term]]
                                                   alias cons(@,@):;

endop
end of module
```

Maintenant nous pouvons spécifier le fichier définissant la syntaxe et la sémantique opérationnelle de notre langage *narrowing*. Nous spécifions dans ce fichier qu'un programme dans ce langage est composé de la définition des variables, la définition des symboles de fonctions, et la définition des règles de réécriture. Nous spécifions de plus qu'une requête r de notre langage est représentée par une équation entre deux termes et qu'elle doit être exécutée en appliquant la stratégie *narrow* sur le terme de départ $(r|0|true)$. Pour évaluer la requête, nous importons le module `basic_narrowing` instancié par les ensembles de variables, symboles de fonction et règles de réécriture.

Le fichier décrivant la syntaxe et la sémantique opérationnelle du langage narrow *est donc:*

```
LPL narrowing description

  specification description

    part Vars    of sort list[identifier]
                 modules anyidentifier signatures
    part Ops     of sort list[pair[identifier,number]]
                 modules anyidentifier signatures
    part Rules   of sort list[pair[term,term]]
                 modules extTerm_v[Vars] extTerm_f[Ops] rwRules

  end

  query          of sort Eequation
                 result of sort narrow_state
                 modules basic_narrowing[Vars,Ops,Rules]
                         extTerm_v[Vars] extTerm_f[Ops]
                         boolean int eqSystem
                 start with (narrow) (query |0| true)

end of LPL description
```

Notre langage est ainsi entièrement spécifié et nous pouvons lancer ELAN pour l'interpréter.

Mentionnons maintenant les changements nécessaires pour définir un langage similaire à *narrow*, mais utilisant une version de la logique équationelle multi-sortée.

Imaginons donc que nous ayons décidé de définir un langage de programmation appelé *mixfixnarrow*, dont la sémantique opérationnelle est aussi basée sur la surréduction basique, mais qui utilise une signature multi-sortée et une écriture des termes mix-fixée. Dans ce cas, nous pouvons ré-utiliser le module `basic_narrowing` définissant le processus de la surréduction, sans avoir besoin de le changer, puisque il est basé sur la syntaxe abstraite des termes. Pour spécifier la nouvelle signature et la nouvelle syntaxe concrète des termes nous pouvons utiliser des modules de la bibliothèque. Ainsi, seul le fichier de la description du langage doit être changé. Dans ce fichier nous devons ajouter une partie de programme spécifiant l'ensemble de types utilisés (Pour ne pas confondre, nous utilisons ici le mot *types* pour designer les sortes objet et le mot *sortes* pour les sortes ELAN). Nous devons également changer la sorte du terme définissant une signature, en la sorte représentant la signature mix-fixée multi-sortée. De plus nous devons importer un module `type` instancié par des définitions des types effectifs. Et enfin nous ajoutons un appel à une fonction de test de bon typage des termes. Cette fonction est définie dans les modules de la bibliothèque `typecheckCommons`, `typecheck_f` et `typecheck_v`, les deux dernièrs modules sont paramétrés respectivement par l'ensemble des symboles de fonction et l'ensemble des variables.

Le fichier de description du langage mixfixnarrow *est:*

```
LPL mixfixnarrowing description

  specification description

    part Types  of sort list[identifier]
                modules anyidentifier signatures
    part Vars   of sort list[pair[nelist[sdchar],type]]
                modules anyidentifier signatures type[Types]
    part Ops    of sort list[pair[nelist[sdchar],pair[list[type],type]]]
                modules anyidentifier signatures type[Types]
    part Rules  of sort list[pair[term,term]]
                modules extTerm_v[Vars] extTerm_f[Ops] rwRules
                        typecheckCommons
                        typecheck_v[Vars,Types] typecheck_f[Ops,Types]
                check with typecheck(spart)
 end

  query         of sort pair[term,term]
                result of sort narrow_state
                modules basic_narrowing[Vars,Ops,Rules]
                        type[Types] extTerm_v[Vars] extTerm_f[Ops]
                        boolean int eqSystem
                        typecheckCommons
                        typecheck_v[Vars,Types] typecheck_f[Ops,Types]
                check with typecheck(query)
                start with (narrow) (query |0| true)

end of LPL description
```

Dans le langage *mixfixnarrow*, le programme définissant la fonction *append* a la forme:

```
specification mixfixappend

Types  elem  list ;

Vars
        x : elem    y : list    z : list ;
Ops
        a               : --> elem
        b               : --> elem
        nil             : --> list
        _ . _           : elem list --> list
        append(_,_)     : list list --> list
        ;
Rules

        append( nil , z)        -> z
        append( x.y , z )       -> x . append(y,z)
        ;
end of specification
```

6.3 Études sur la réécriture

Dans cette section nous présentons l'implantation du processus de réécriture en ELAN. C'est-à-dire étant donné un ensemble de règles de réécriture $\mathcal{R}$ (représentées par des couples de termes), nous spécifions comment une forme normale d'un terme t est obtenue en répétant des pas de réécriture.

Nous montrons de deux façons complètement différentes comment nous pouvons définir un pas de réécriture en ELAN. La première possibilité est formée par une description détaillée du processus de la réécriture. C'est-à-dire en décrivant le choix d'un sous-terme à réduire, le choix d'une règle de réécriture, le filtrage de la partie gauche de cette règle avec le sous-terme choisi et le remplacement du sous-terme par une instance de la partie droite de la règle.

Mais d'un autre coté, nous savons que le langage ELAN dispose lui même des notions des règles de réécriture et du processus de la réécriture. De plus il dispose d'un mécanisme d'encodage qui devrait permettre d'encoder les formules de la logique objet dans les règles de réécriture. Dans ce cas précis les formules de la logique objet sont des règles de réécriture. Il parait évident que la façon la plus simple d'encoder les règles de réécriture objet dans les règles de réécriture ELAN est d'encoder chaque règle par elle même. Cette deuxième possibilité correspond à une représentation interne (c.f. 7.1) des règles de réécriture. Nous montrons maintenant deux modules ELAN définissant la réécriture des deux manières.

Le module décrivant la réécriture externe (en décrivant tous les pas d'évaluation) est:

```
module rewriting[vars,fss,rwrules]
import  rwrules termCommons eqSystem syntacticUnification[fss]
        pair[term,term] list[pair[term,term]] applysubston[term];


rule rewrite for term
declare PD      : eqSystem;
        s,g,d : term;
        theta : substitution;
        occ    : list[int];
        rwr    : pair[term,term];
body
  s       => s[theta(d)] at occ
                where theta := () system_to_subst(PD)
                where PD := (unifys) s at occ = g
                where g := () first(rwr)
                where d := () second(rwr)
                where rwr := (extract) elem(rwrules)
                where occ := (extract) nvocc(s)
end of rule


strategy rewrite
  while
    dont care choose(rewrite)
  endwhile
end of strategy

end of module
```

Dans le corps de la règle rewrite *nous pouvons utiliser la stratégie* unifys *réalisant l'unification à la place du filtrage, puisque l'algorithme d'unification est implanté dans la bibliothèque et dans l'exemple un de deux termes à unifier est clos, les deux algorithmes ont donc un comportement identique.*

Les variables de la règle rewrite *sont successivement affectées (et ré-affectées dans le cas de retour arrière quand la stratégie $unifys$ échoue) aux valeurs suivantes:*

occ	*sur une position non-variable du terme s*
rwr	*sur une règle de réécriture*
d	*sur la partie droite de la règle rwr*
g	*sur la partie gauche de la règle rwr*
PD et theta	*sur le filtre entre la partie gauche g et le souterme $s_{\|occ}$*

Pour implanter l'autre approche internalisant les règles de réécriture objet, nous donnons un module définissant les règles de réécriture nommées *top_rewrite* en les important directement de la spécification. Mais, puisque les règles ELAN contrôlées par une stratégie ne sont appliquées que sur la tête des termes, ce module doit définir aussi une règle de réécriture permettant de descendre dans la structure des termes, ce qui est fait par définition de la règle *inside_rewrite*. Le module spécifie de plus une stratégie contrôlant les applications de ces règles.

Le module réalisant la réécriture en internalisant les règles de réécriture est:

```
module internRewriting[vars,fss,rwrules]
import  vars fss rwrules signaturesFunctions int term_f[fss] extTerm_v[vars]
        pair[term,term] list[pair[term,term]] list[variable] identity[term];

FOR EACH RWR:pair[term,term]; L,R:term
SUCH THAT RWR:=(extract) elem(rwrules) AND L:=()first(RWR) AND R:=()second(RWR):{

  rule top_rewrite for term
  declare FOR EACH X:variable SUCH THAT X:=(extract)elem(varlist_vars):{ X : term; }
  body
        L        =>       R
  end of rule

}


FOR EACH F:symbol;N:int
SUCH THAT F:=(extract)elem(fss) AND N:=()arity(F) :{

  rule inside_rewrite for term
  declare s_1,...,s_N : term;
          t_1,...,t_N : term;
  body
     F(s_1,...,s_N)          => F(t_1,...,t_N)
                                    if F(s_1,...,s_N) != F(t_1,...,t_N)
                             {      where t_I := (PO_step) s_I  }_I=1...N
  end of rule

}


strategy PO_step
  dont care choose(top_rewrite inside_rewrite identity)
end of strategy

strategy PO_rewriting
 while
   dont care choose(top_rewrite inside_rewrite)
 endwhile
end of strategy


end of module
```

Remarquons que la stratégie *PO_rewriting* ici définie décrit une stratégie de la réécriture *parrallel outermost*, nous pouvons implanter par exemple la stratégie *parallel innermost* en changeant l'ordre des appels des règles *inside_rewrite* et *top_rewrite*.

6.4 Langage fonctionnel basé sur la surréduction paresseuse

Nous mentionnons dans cette section une implantation du langage de programmation BABEL [MNRA92] dont la sémantique opérationnelle est basée sur la surréduction paresseuse. Le langage BABEL a été développé à MCU Madrid, sa spécification en ELAN fait l'objet d'une coopération au sein du groupe de travail CCL. Nous ne présentons pas une implantation complète de ce langage, elle dépasse le cadre de la thèse. Nous nous contentons d'une brève description du langage et nous présentons un exemple de programme dans ce langage. Cet exemple illustre néanmoins que ELAN peut être utilisé dans des spécifications de mécanismes d'évaluation non-standards et relativement compliqués.

Un programme dans le langage BABEL consiste en un ensemble d'équations conditionnelles. Au niveau des signatures BABEL on distingue deux sortes de symboles de fonction: les constructeurs qui servent à la construction de structures de données et les symboles représentant des fonctions définies par des équations. Dans une équation on distingue deux sortes d'égalités: une *égalité tout court* (noté par =) définissant des fonctions et une *égalité stricte* (noté par ==) qui est une égalité utilisée dans les conditions et qui signifie l'égalité des deux termes dans leurs formes évaluées jusqu'aux constructeurs.

Nous présentons ci dessous un exemple de programme BABEL. Ce programme définit la fonction *from* construisant une liste infinie des entiers à partir de la valeur de son argument. Une autre fonction *takeuntil* prend une liste des entiers et retourne la partie de cette liste jusque à une certaine valeur. Pour définir cette fonction nous avons besoin de spécifier aussi une fonction auxilière *less* qui compare deux entiers.

Dans la syntaxe concrète des définitions dans BABEL une condition est introduite par le symbole <=.

```
specification takefrom

Vars
        X   Y   A   B ;

Consts
        zero:0   s:1   nil:0   cons:2    true:0   false:0 ;

Funss
        from:1   takeuntil:2 less:2 ;

Rules

        from(X) = cons(X,from(s(X)))

        takeuntil(X,cons(A,B))    = nil                           <=   less(X,A) == true
        takeuntil(X,cons(A,B))    = cons(X,nil)                   <=   X == A
        takeuntil(X,cons(A,B))    = cons(A,takeuntil(X,B))        <=   less(X,A) == false

        less(zero,zero)     = false
        less(zero,s(X))     = true
        less(s(X),zero)     = false
        less(s(X),s(Y))     = less(X,Y)

end of specification
```

Grâce à la stratégie paresseuse du langage nous pouvons appliquer la fonction *takeuntil* sur une liste infinie générée par la fonction *from*. Cette stratégie paresseuse peut être alors utilisée pour évaluer des requêtes comme par exemple:

$$takeuntil(s(zero), from(X)) == cons(zero, A)$$

et nous obtenons l'unique réponse:

$$X \to zero, A \to cons(s(zero), nil)$$

Après quoi l'exécution s'arrête.

7

Comparaison d'ELAN avec des systèmes similaires

Dans ce chapitre nous présentons quelques travaux dans le domaine des cadres logiques et quelques environnements de spécification de logiques qui sont proches du langage ELAN. Nous présentons une description des idées principales sur lesquelles les différents systèmes sont fondés et nous les comparons ensuite avec les principes d'ELAN. Le chapitre commence par les cadres logiques, en particulier par les cadres logiques basés sur le λ-calcul avec différents systèmes de typage et il continue par des langages qui ont des similitudes avec ELAN au niveau de la logique de base ou au niveau des buts postulés. Il s'agit notamment de Maude, un langage basé sur la logique de réécriture, et d'ECOLOG un langage de prototypage de logiques avec contraintes.

7.1 Cadres logiques

Les cadres logiques sont des formalismes qui permettent de spécifier différentes logiques. Un cadre logique doit être capable de coder la syntaxe (au moins au niveau abstrait) et les preuves (ou encore mieux des démonstrateurs automatiques) d'une logique. De plus il doit permettre un raisonnement formel sur les preuves de la logique codée. Nous appelons *méta-raisonnement* cette possibilité de formuler des propositions sur une logique. Un cadre logique doit obligatoirement être complètement formel pour pouvoir être implanté sur un ordinateur.

L'idée générale des cadres logiques est le choix d'une logique particulière (que nous appelons la *logique cadre*) et le codage des notions d'une logique spécifiée (que nous appelons la *logique objet*) dans cette logique cadre. Cette approche est très similaire de celle que nous avons choisi en ELAN pour spécifier différents langages de programmation logiques.

Les divers systèmes diffèrent par le choix de la logique cadre. Pour pouvoir comparer ELAN avec les autres systèmes, nous essayons de faire d'abord une classification des logiques cadres du point de vue de leurs capacités à coder différentes logiques et nous classifions le langage ELAN dans ce contexte. Dans cette optique il nous semble essentiel de distinguer deux manières différentes de coder une logique objet dans une logique cadre. Une notion de la logique objet peut être codée dans la logique cadre en utilisant soit une représentation *externe* soit une représentation *interne*. La différence entre les deux représentations est difficile à définir formellement. Intuitivement, comme le montre l'exemple ci-dessus, dans la représentation externe, un objet est défini plutôt par sa syntaxe et toutes ses propriétés sont définies par des définitions entières et explicites sur cette syntaxe. Tandis que dans une représentation interne une notion de la logique objet est représentée par une notion de la logique cadre avec les propriétés ressemblant aux propriétés de cette notion dans la logique objet. L'avantage de la représentation interne est que la notion ne nécessite pas une définition entière de ses propriétés, elles sont "évidentes" en connaissant les propriétés de son codage dans la logique cadre. L'inconvénient de la représentation interne est la perte d'accès à la structure de la notion codée. Souvent on ne peut donc pas coder toutes les fonctions liées à la notion originale. Dans le texte suivant nous utilisons l'expression *internaliser* pour exprimer le fait d'utiliser la représentation interne d'une notion.

Exemple: un codage interne notoirement connu est le traitement de quantificateurs d'une formule du premier ordre en lambda calcul. Prenons par exemple la formule:

$$\forall x : P(x)$$

où P est un prédicat. Pour la coder dans le lambda calcul nous introduisons une constante spéciale d'ordre supérieur $forall$ du type:

$$forall : (term \to form) \to form$$

avec le type $term$ représentant les termes codés, et le type $form$ représentant les formules. La liaison de la variable liée x est codée en utilisant la liaison similaire dans l'abstraction du lambda calcul. Cette formule est alors codée par le terme du lambda calcul du type $form$:

$$forall(\lambda x.P(x))$$

L'avantage de ce codage est que dans les manipulations avec des formules ainsi codées nous ne sommes pas obligés de faire attention au renommage des variables liées. Ce renommage est internalisé automatiquement par le α-conversion du lambda calcul. De plus, sans avoir besoin d'entrer dans la définition détaillée, nous pouvons coder l'application d'une substitution en utilisant la règle de β-conversion du lambda calcul. Prenons par exemple une règle de déduction de la logique du premier ordre remplaçant dans une formule Q toutes les occurrences de la variables x liées au quantificateur $\forall$ extérieur par un terme t, c'est-à-dire la règle:

$$\frac{\forall x : Q}{Q[x/t]}$$

Cette règle peut être codée dans notre représentation interne tout simplement par une règle:

$$\frac{forall(\lambda x.Q)}{(\lambda x.Q)t}$$

où le remplacement est fait automatiquement par une β-conversion juste après l'application de cette règle.

L'inconvénient de cette représentation est l'existence de fonctions sur des formules qui ne peuvent pas être exprimées. Par exemple la fonction donnant l'ensemble de toutes les variables liées d'une formule ne peut pas être définie si ce codage est utilisé.

Par opposition à cette représentation interne, une représentation externe correspond à la syntaxe abstraite des termes. Dans ce cas, le codage de la formule de l'exemple:

$$\forall x : P(x)$$

dans le lambda calcul contient l'introduction des deux constantes typées:

$$\begin{array}{lcl} a & : & var \\ forall & : & var \to form \to form \end{array}$$

La formule initiale est représentée par le terme ci-dessous du type $form$:

$$forall \;\; a \;\; (P(a))$$

Mais maintenant pour définir la règle remplaçant les occurrences de la variables a par un terme t nous sommes obligés de définir entièrement le mécanisme de ce remplacement.

Chaque logique est plus ou moins appropriée aux deux types de représentation des logiques objet. Nous essayons d'exprimer maintenant quelques critères pour la comparaison de différentes logiques cadres. Dans tous les cas il est important d'avoir la possibilité d'un codage commode et modulaire pour pouvoir définir différentes logiques objet sans effort exagéré. Il est aussi important d'avoir un codage assez transparent et intuitif pour ne pas se perdre dans les constructions de preuves et dans le méta-raisonnement sur la logique objet. Pour pouvoir utiliser la représentation interne des logiques objet, la *puissance* du formalisme de la logique cadre est déterminante. Il est impossible d'internaliser une notion compliquée (telle que le mécanisme de liaison des variables) dans une logique cadre qui n'a pas de notion similaire dans son formalisme. Les différentes logiques diffèrent donc par leurs possibilités d'internaliser certaines notions des logiques objet. Nous disons qu' une logique cadre est *plus puissante* qu'une autre logique si elle permet d'internaliser plus de notions. On peut penser alors qu'une logique plus puissante est toujours un meilleur candidat pour la logique cadre. Mais une augmentation de la puissance de la logique cadre (par exemple en introduisant des systèmes de typage plus compliqués) mène à une perte de simplicité et peut conduire à des difficultés concernant des propriétés (telles que la non-contradiction) de la logique cadre elle même. De plus il reste la limitation des codages internes qui peuvent être trop faibles pour exprimer toutes les formules (et méta-formules) dont nous avons besoin et dont nous aurons besoin un jour.

Par opposition, dans le cas de la représentation externe des logiques objet il n'y a pas de problème au niveau de la possibilité d'une logique cadre de pouvoir ou non coder une logique objet. Pratiquement dans chaque logique nous pouvons coder d'une façon externe toutes les autres logiques. Il suffit pour cela que la logique cadre permette de coder les fonctions récursives. Mais les diverses logiques diffèrent par la forme du codage. La commodité et la transparence sont alors les critères de comparaison.

Nous pouvons donc constater que la question du choix d'une logique cadre qui soit universelle ainsi que la question de la meilleure représentation est ouverte. Les cadres logiques pratiquement utilisés aujourd'hui diffèrent dans ces choix et utilisent différentes logiques cadres et pour spécifier différentes logiques objet ils utilisent souvent une combinaison des représentations internes et externes. Nous présentons brièvement dans cette section quelques cadres logiques connus et nous les comparons avec ELAN.

7.1.1 Cadres logiques basés sur les logiques d'ordre supérieur

Les cadres logiques basés sur les logiques d'ordre supérieur sont très populaires parmi les systèmes de codage des logiques. Ce choix de la logique cadre est motivé essentiellement par la possibilité du codage interne des quantificateurs et par des extensions du principe de Curry-Howard permettant de coder les propositions d'une logique objet par les types de la logique cadre. Suivant le même principe les preuves sont codées par des termes. Ce qui est dans notre optique une représentation interne de la notion de logique objet: “P est une preuve de x” par la notion: “$\lceil P \rceil$ est un terme du type $\lceil x \rceil$” dans la logique cadre (ici, les symboles $\lceil$ et $\rceil$ expriment la fonction de codage). En enrichissant le système de typage du λ-calcul utilisé, nous pouvons enrichir l'ensemble des propositions codées permettant ainsi de coder plus de formules intéressantes.

Dans l'histoire de systèmes utilisant ce type de codage, mentionnons au moins AUTOMATH [dB80] de de Bruijn qui s'est donné pour but de formaliser le langage mathématique, et LCF [GMW79] initialement destiné à la vérification des programmes. Aujourd'hui il existe un grand nombre de logiciels assez sophistiqués et spécialisés pour différentes tâches. Nous pouvons mentionner par exemple LF [HHP86], ELF [Pfe91], Coq [CPMDH89], Nuprl [C^+86], Isabelle [Pau89], λ-Prolog [MN86], FS0 [Fef88] parmi autres. Ils différent essentiellement par leur système de typage.

Nous en présentons quelques uns:

LF [The Edinburg Logical Framework]

LF dispose d'un langage de types dépendants. Il utilise trois niveaux de typage des termes, on distingue donc trois niveaux d'entités: les termes qui sont appelés les *objets*; les types appelés *familles* et les types de types appelés *Kinds*. Les trois notions sont définies par la grammaire:

$$\begin{array}{lcl} Kinds & : & K ::= Type \mid \Pi x : A.K \\ Familles & : & A ::= a \mid \Pi x : A.B \mid \lambda x : A.B \;\; AM \\ Objects & : & M ::= c \mid x \mid \lambda x : A.M \mid MN \end{array}$$

où les λ et Π sont des opérateurs de liaison pour les variables. Le Π est une extension du type fonctionnel "$\rightarrow$" du lambda calcul aux fonctions avec de dépendances de types.

Ce système de typage permet un codage interne des *jugements comme types* et de plus il permet de formuler des méta-jugements sur les formules.

ELF

est une extension du langage LF par un constructeur de type des produits dépendants avec une sémantique opérationnelle. Cela permet de combiner le cadre de LF avec une sémantique opérationnelle (style PROLOG), pour pouvoir spécifier des algorithmes de démonstrateurs de théorèmes. Le langage de types de ELF est:

$$\begin{array}{lcl} Kinds & : & K ::= Type \mid \Pi x : A.K \\ Familles & : & A ::= a \mid \Pi x : A.B \mid \lambda x : A.B \mid AM \mid \Sigma x : A.B \\ Objects & : & M ::= c \mid x \mid \lambda x : A.M \mid MN \mid (M,N) \mid fst\ M \mid snd\ M \end{array}$$

Dans les deux formalismes (LF et ELF) le constructeur de types λ n'est pas présent dans les formes normales des types des objet du langage, il peut donc être éventuellement omis.

Coq

utilise le système de typage du *Calcul des Constructions* avec des types inductifs. Par rapport au système de types LF le système de types de Coq est étendu par une couche supplémentaire de typage introduisant des types inductifs, des types polymorphes et des constructeurs de types.

Le système Coq supporte le paradigme de *preuves comme programmes* et il permet d'obtenir directement un programme fonctionnel à partir de la preuve d'une formule. La formule exprime le type du programme et donc aussi ses propriétés.

Comparaison avec ELAN

Pour pouvoir comparer ces systèmes avec ELAN rappelons que l'introduction des systèmes de typage forts a été motivée par la possibilité d'internaliser la plupart des notions d'une logique objet en utilisant le codage interne des *propositions comme types*. Rappelons aussi que ce codage n'est pas toujours totalement adéquat et qu'une augmentation de la puissance de typage (par exemple dans Coq) peut mener à des difficultés concernant les propriétés de la logique cadre elle-même. Par opposition, ELAN utilise comme logique cadre la logique de réécriture, une logique du premier ordre très simple (on peut dire très faible) ne contenant ni la possibilité d'internaliser des liaisons de variables, ni de couche de typage permettant d'internaliser des notions d'une logique objet. Toutes les notions d'une logique objet doivent être codées par des termes du premier ordre. Nous appelons ce style de codage le *codage par syntaxe*, parce que les termes du premier ordre nous semblent être une notion très syntaxique. Ils sont, par exemple, souvent représentés par des arbres correspondant aux arbres syntaxiques construits à partir de leur écriture. Ce codage par syntaxe est dans notre classification classé comme une représentation externe puisqu'il nécessite des définitions explicites et complètes exprimées sur la syntaxe des notions. Mais aussi c'est une présentation très naturelle et elle est utilisée couramment dans la littérature scientifique. Même les systèmes d'ordre supérieur présentés ici ont été définis de cette façon.

En utilisant ce principe de codage, ELAN force donc la représentation externe des logiques objet, mais ici il offre des moyens, qui rendent ce codage très proche des définitions naturelles (c'est-à-dire des définitions du style “papier et stylo”).

Sur les objets codés par les termes du premier ordre, ELAN permet de définir des algorithmes formels en utilisant des règles de réécriture et des stratégies. En utilisant les règles de réécriture on peut en effet de façon limitée internaliser les règles de déduction d'une logique objet, mais dans ce cas on n'a plus la possibilité de raisonner sur ces preuves. Normalement, si on veut raisonner sur les preuves d'une logique objet, on est obligé de les représenter d'une façon externe par des termes.

L'autre point de comparaison entre ELAN et les systèmes présentés précédemment est l'utilisation de formalismes permettant d'automatiser des parties de preuves. Les *tactics* et *tacticals*, introduites déjà dans AUTOMATH sont normalement utilisées dans ce contexte. Du point de vue opérationnel, ces notions sont très proche des stratégies d'ELAN, néanmoins les *tactics* et *tacticals* sont utilisés dans un contexte différent que les stratégies d'ELAN. Les *tactics* et *tacticals* sont des fonctions de base permettant de combiner des déductions élémentaires et utilisées directement à partir du langage d'implantation. Les stratégies d'ELAN décrivent des dérivations obtenus par applications de règles de réécriture. Elles sont exprimées à l'aide d'un langage fixe et assez restreint, mais qui contient par exemple un opérateur d'application d'une règle de réécriture sur des sous-termes, qui n'a pas de sens dans le contexte de *tacticals*.

Du point de vue opérationnel les possibilités d'expression des algorithmes par les stratégies ELAN sont plus restreintes en comparaison des *tacticals* qui peuvent être en effet des programmes d'un langage de programmation complet, mais d'un autre côté, grâce à cette restriction, nous pouvons faire des raisonnements sur les évaluations elles-mêmes.

La dernière grande différence entre ELAN et les systèmes précédents est représentée par le mécanisme de pré-traitement d'ELAN, qui n'existe pas dans le contexte des autres systèmes. Ce pré-traitement est destiné à un codage automatique d'une logique objet dans la logique de réécriture. Dans le cas d'autres environnements, ce codage doit être fait manuellement.

Dans le reste du chapitre nous présentons maintenant des systèmes qui sont proches d'ELAN dans leur manière de coder les différents formalismes.

7.1.2 Maude et Maudelog

Maude et Maudelog sont deux langages définis par J.Meseguer [Mes91, Mes93] basés sur la logique de réécriture. Un programme dans les deux langages définit une théorie de la logique de réécriture, c'est-à-dire une signature et un ensemble de règles de réécriture. Les moyens à la disposition de l'utilisateur pour la définition d'une telle théorie sont similaires à ceux de OBJ3 [GW88, GKK+87]: la signature utilisée est une signature ordo-sortée, les programmes sont définis d'une façon modulaire, chaque module représentant une théorie particulière. Les modules peuvent être paramétrés, et des arguments formels specifiés par des modules spéciaux. L'exécution d'un programme dans les deux langages est lancée par l'introduction d'un terme de départ. Dans le cas de Maude, ce terme doit être clos, il ne peut pas contenir de variables. Par opposition, les termes de départ de Maudelog peuvent contenir des variables. De là vient aussi la différence dans la sémantique opérationnelle des deux langages qui est représentée par la réécriture dans le cas de Maude et par la surréduction dans le cas de Maudelog.

Pour pouvoir comparer ces deux langages avec ELAN il faut mentionner, que Maude et Maudelog ne sont pour l'instant que des propositions de langages, ils ne sont pas encore développés jusqu'à une implantation. Il est donc difficile de comparer les comportements de ces langages de façon détaillée. Plusieurs propriétés qui sont incluses dans la définition d'ELAN pour des raisons pratiques n'ont pas d'équivalent dans Maude et Maudelog, mais il nous semble que pour une implantation pratique, les deux langages devront suivre un schéma similaire à celui d'ELAN.

Nous présentons maintenant une comparaison d'ELAN avec le langage Maude qui utilise la même sémantique opérationnelle de logique de réécriture qu'ELAN. La même comparaison peut être faite pour Maudelog avec une différence supplémentaire dans la sémantique opérationnelle.

Le langage Maude est très similaire à ELAN. Voici quelques différences:

- Par opposition à ELAN le langage Maude ne contient pas de mécanisme de pré-traitement permettant le codage automatique d'une logique objet dans la logique de réécriture.
- Maude utilise une version ordo-sortée de la logique de réécriture, tandis que ELAN utilise la version multi-sortée.
- Maude permet de spécifier des exigences sur les argument effectifs des modules, tandis qu'ELAN ne contient pas un tel mécanisme.

Les autres différences viennent essentiellement du fait que Maude n'est pas encore implanté; en particulier,

- Maude ne contient pas pour l'instant une notion similaire à la notion de stratégie d'ELAN. Ainsi, en Maude, l'espace de recherche à traverser durant une exécution est l'espace de toutes les dérivations possibles. Ceci est pratiquement ingérable et nous pensons que, pour une implantation pratique, le langage Maude sera obligé d'introduire un mécanisme similaires à celui des stratégies d'ELAN.
- Maude permet théoriquement d'utiliser n'importe quelle théorie equationelle pour définir les classes d'équivalences sur lesquelles les règles de réécriture sont appliquées (cf. 2.2 et 3.2). En ELAN ces théories sont restreintes aux théories associatives-commutatives pour de simples raisons d'implantation. Mais d'un autre côté, dans le cas de Maude, la possibilité de définir une théorie quelconque posera de graves problèmes pour l'implantation. Le filtrage d'une partie gauche d'une règle avec le terme (la classe d'équivalence) à réduire est indécidable et rend donc l'implantation de la réécriture pratiquement impossible. Il est donc clair que, pour une implantation réelle, les théories équationelles devront être radicalement restreintes.

7.2 ECOLOG

ECOLOG: *un Environnement pour la programmation en LOGique COntrainte* [Hab92, Hab93, Habar] est un environnement dont le développement au LRI d'Orsay est parti des mêmes motivations que le développement d'ELAN à Nancy. De même qu' ELAN, ECOLOG permet de spécifier et prototyper différents langages de programmation logique avec contraintes. Comme en ELAN, le mécanisme de déduction du langage spécifié et du résolveur des contraintes sont spécifiés en termes de règles de réécriture. ECOLOG dispose d'un mécanisme similaire au pré-traitement d'ELAN qui permet de coder automatiquement différentes logiques. Néanmoins, ECOLOG représente dans l'état actuel plutôt un environnement de support pour le prototypage qu'un véritable langage de spécification. Une grande partie de la spécification en ECOLOG doit être effectuée directement dans le langage d'implantation CAML (un langage fonctionnel de la famille ML). Ainsi par exemple l'utilisateur peut utiliser un formalisme d'ECOLOG pour spécifier la syntaxe concrète d'un langage objet, mais les procédures construisant la syntaxe abstraite doivent être écrites en CAML. L'utilisateur peut écrire des règles de réécriture, dans une syntaxe concrète et les termes contenus dans ces règles sont traduites dans une forme interne en utilisant les procédures de construction de la syntaxe abstraite spécifiées auparavant. Pour pouvoir appliquer une telle règle de réécriture l'utilisateur doit programmer, lui même, l'algorithme de filtrage agissant sur la syntaxe abstraite des termes. Pour contrôler le déroulement de l'évaluation, ECOLOG dispose d'un ensemble de *tactics* et *tacticales* de base qui peuvent être aussi utilisées à partir d'une procédure écrite en CAML.

Cette division entre les aspects syntaxiques, qui sont gérés au niveau d'ECOLOG, et les aspects abstraits, qui sont entièrement gérés au niveau des procédures écrites en CAML, donne à l'utilisateur la liberté de spécifier la syntaxe abstraite des termes. Cette syntaxe abstraite peut être réalisée par les structures de données les plus appropriées aux algorithmes qui l'utilisent. Par exemple: en présence de symboles associatifs-commutatifs, les termes abstraits peuvent être aplatis; en présence de structures cycliques, les termes peuvent être représentés par des graphes et en présence des quantificateurs, les termes abstraits peuvent refléter le mécanisme de liaisons.

Nous finissons cette description par une brève comparaison avec ELAN.

- La différence majeure entre ELAN et ECOLOG est bien sûr dans l'interprétation interne des termes: en ELAN les termes sont toujours représentés par des arbres, tandis qu'en ECOLOG, ils sont représentés par une structure de données quelconque implantée par l'utilisateur.

- Il n'y a pas de sémantique pour les spécifications ECOLOG, du fait du mélange du code ECOLOG et CAML dans les spécifications. De plus la sémantique des termes d'ECOLOG est cachée par l'utilisation de la syntaxe abstraite dont la construction est entièrement spécifiée en CAML.

- Par opposition à ELAN, le langage ECOLOG est pour l'instant très peu modulaire, un programme est toujours composé uniquement de trois modules bien spécifiques. Son extension modulaire sera assez exigeante, parce qu'une combinaison de deux modules spécifiant deux syntaxes abstraites différentes (et deux algorithmes de filtrage) devrait aboutir à une représentation mixtes des termes, avec un algorithme effectuant le filtrage dans la combinaison des deux théories.

8
Implantation

Une implantation expérimentale de l'interpréteur d'ELAN nous a permis de valider nos idées sur la construction des systèmes de calcul et sur le codage de logiques et résolveurs de contraintes. Les expériences faites avec l'implantation ont beaucoup influencé le développement du langage et ont ainsi contribué à la définition de sa version actuelle.

Nous avons inclus le chapitre décrivant l'implantation dans la thèse parce que l'implantation d'ELAN représente une grande partie de travaux effectués et elle prouve la validité de nos idées sur le codage de différentes logiques dans les systèmes de calcul.

Dans ce chapitre nous décrivons les principes de l'implantation d'ELAN elle même. Nous ne décrivons pas l'implantation en détail, nous nous contentons d'une brève description du cadre général d'implantation et puis nous mentionnons quelques parties particulières, qui nous semblent être non-standard ou utiles pour des implantations futures de ce genre de langages de programmation. Les autres parties du langage sont implantées en utilisant des techniques standards.

L'implantation d'ELAN est réalisée en langage C++, elle fait à peu près 10 000 lignes de code. Elle marche actuellement sous le système d'exploitation UNIX sur les stations de travail Sun et HP.

Le langage C++ a été choisi comme langage d'implantation surtout pour des raisons d'efficacité par rapport aux langages de haut niveau comme CAML, PROLOG ou LISP. De plus, puisque le niveau syntaxique (c'est à dire l'implantation d'un analyseur syntaxique compliqué, l'implantation d'un pré-processeur puissant, la nécessité de travailler avec grand nombre de fichiers au même temps...) est fondamental dans le système actuel, il nous semble qu'un langage classique (impératif) est mieux approprié à ce genre d'application, même si nous avons perdu la possibilité d'utiliser la gestion de mémoire implicite des langages non-impératifs.

Parmi les langages impératifs nous avons réellement considéré deux candidats, les langages C et C++. Les deux langages sont disponibles pratiquement sur chaque installation du système UNIX et ils sont bien portables, on ne doit donc pas rencontrer de difficultés pour installer ELAN sur n'importe quelle architecture avec le système d'exploitation UNIX.

Par rapport à C, le langage C++ contient des possibilités du programmation orientée objet, d'encapsulation des données, des possibilités de définir des opérateurs surchargés avec des liaisons statiques et dynamiques. Et même si nous n'utilisons pas excessivement des caracteristiques orientés objet du langage C++ dans notre implantation, C++ en comparaison avec C possède des caractéristiques qui en font un langage agréable pour une programmation pratique et tout-à-fait approprié à nos besoins.

8.1 Description générale de l'implantation

L'implantation actuelle d'ELAN est réalisée par un interpréteur de programmes. L'exécution d'un programme ne produit donc aucune traduction des textes sources dans un langage intermédiaire, qui serait exécuté après, mais l'interpréteur lit toutes les sources et il stocke les informations de signatures, règles de réécriture et stratégies dans la mémoire. Puis, il demande interactivement l'entrée d'une requête et utilise ensuite les informations lues auparavant pour l'évaluer.

Cela divise le code de l'interpréteur de façon naturelle en deux parties: la partie syntaxique qui s'occupe de la lecture des modules et de l'analyse syntaxique, et la partie d'exécution qui s'occupe de l'évaluation d'une requête.

Dans la figure 8.1 nous pouvons voir le dessin général de l'implantation. L'interpréteur est représenté par deux parties majeures représentant la partie syntaxique et la partie évaluation qui contient l'interprétation de la réécriture avec stratégies, gestion de la mémoire etc. Les deux parties sont appelées par le module `ldmain` qui gère le déroulement d'une session avec l'utilisateur. Les entrées de l'interpréteur entrent soit dans le module `ldmain` (ce sont les fichiers de description d'un langage, la spécification et la requête) ou directement dans la partie syntaxique (les fichiers avec les corps de modules). Les deux parties utilisent les structures des données contenant un système de calcul pour communiquer entre elles.

La description de l'interpréteur d'ELAN présentée dans ce chapitre suit cette division naturelle en deux partie.

8.2 Partie syntaxique

La partie syntaxique représente actuellement la plus grande partie du code de l'interpréteur. Puisque l'implantation actuelle est toujours une implantation expérimentale, la partie d'exécution était élaboré avec moins de soin que la partie syntaxique.

Les taches principales effectuées dans la partie syntaxique sont:

- l'analyse syntaxique d'un module et la construction d'un système de calcul,
- l'implantation des visibilités des opérateurs par rapport à la modularité,
- l'implantation du passage des paramètres pour les modules paramétrés,
- l'implantation du pré-processeur.

La gestion d'un module est implantée par le schéma des implantations dirigées par la syntaxe. L'analyseur syntaxique d'un module représente donc la procédure du plus haut niveau qui fait appel aux fonctions fournissant l'entrée, effectue l'analyse syntaxique et appelle ensuite des actions sémantiques chargées de stocker le système de calcul analysé dans la mémoire.

En regardant la syntaxe des modules ELAN, nous pouvons constater qu'il y a deux niveaux de définitions de la syntaxe. La syntaxe que nous appelons *fixée* est la syntaxe définie dans le manuel du langage (chapitre 4 de la thèse). La syntaxe non fixée est la syntaxe des termes qui ne peut pas être définie en avance puisqu'elle dépend de la signature définie par l'utilisateur. Pour faire une analyse syntaxique d'un module, il n'est donc pas possible d'utiliser tout simplement un outil du type YACC engendrant un analyseur syntaxique à partir d'une grammaire LALR(1). D'autant plus, qu'une grammaire pour un terme n'est pas restreinte à cette classe de grammaires et peut être une grammaire générale hors contexte. L'analyse syntaxique d'un terme nécessite donc l'implantation d'un analyseur syntaxique puissant du type de l'algorithme d'Earley. D'autre part il n'est pas souhaitable d'utiliser un tel analyseur syntaxique trop puissant pour l'analyse syntaxique d'un module entier à cause de ses complexités excessives d'espace et de temps. Dans notre implantation nous avons donc combiné les deux possibilités et nous utilisons un analyseur syntaxique LALR(1) pour les parties de modules décrites par une grammaire fixée et l'algorithme d'Earley pour l'analyse syntaxique des termes. L'analyseur syntaxique LALR(1) est engendré automatiquement par un outil similaire à YACC.

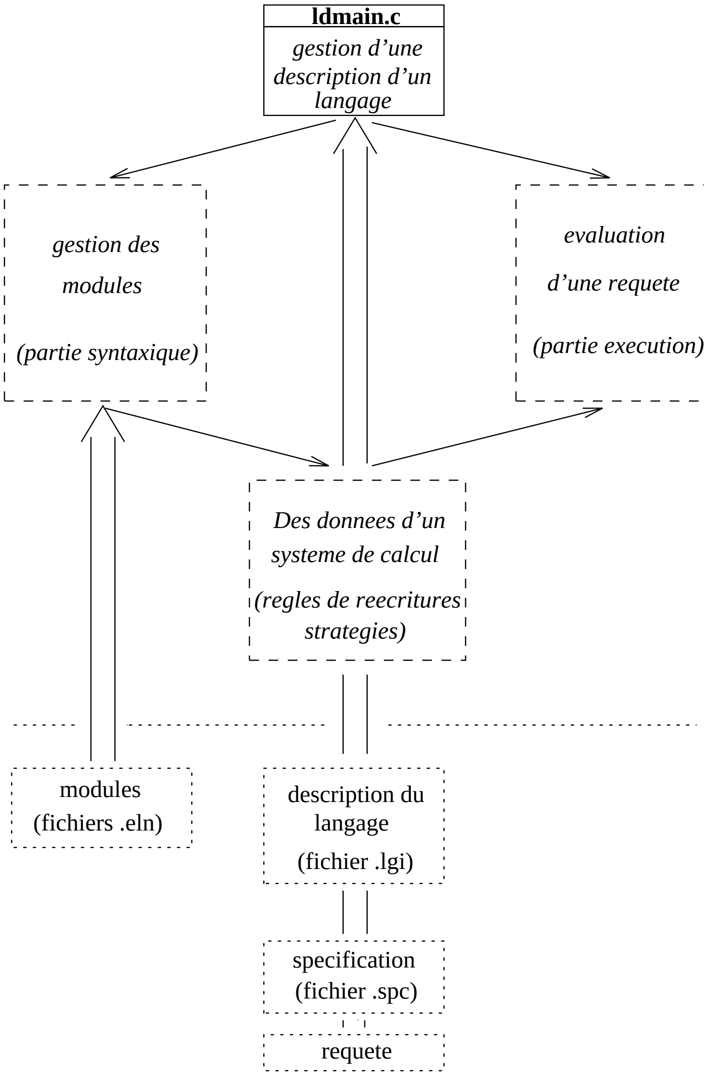


Figure 8.1. L'interpréteur d'`ELAN`**.**
Dans la figure, les cadres servent pour marquer l'ensemble des classes (dans le sens de la programmation objet) qui réalise une certaine partie. Les flèches entre les cadres expriment la relation "être utilisé par". Les fonctions définies dans les cadres placés dans la figure plus haut font appel aux fonctions définies dans les cadres placés plus bas. Les directions des flèches expriment les directions de transmissions des données. Une flèche menant de bas en haut représente un appel de fonction qui retourne des données importantes dans le résultat, tandis qu'une flèche menant de haut en bas représente un appel d'une procédure obtenant des données importantes dans l'argument.
Les flèches doubles représentent les entrées du programme.

Cela nous garantit une analyse syntaxique linéaire et donc efficace pour une grande partie des modules. L'algorithme d'Earley est utilisé seulement pour analyser de petits morceaux du texte et donc ses complexités de temps et d'espace n'influent pas tellement sur la complexité totale de l'analyse syntaxique du module.

Dans la figure 8.2, nous voyons la structure générale de la partie compilation d'un module dans des structures de données internes. Le texte d'un module passe par une cascade de modules, où l'analyseur lexical le transforme d'une chaîne des caractères en une chaîne d'unités lexicales. Cette chaîne passe après par le pré-processeur, qui cherche et évalue toutes les constructions du pré-traitement et enfin le texte pré-traité entre alternativement dans deux analyseurs syntaxiques en fonction du morceau de texte dont il s'agit. Si le morceau du texte actuellement lu est une partie du module décrit par la grammaire fixée, il est lu par l'analyseur syntaxique LALR(1). S'il s'agit d'un terme, il entre dans l'analyseur syntaxique réalisé par l'algorithme d'Earley.

Nous décrivons maintenant quelques parties intéressantes de notre implantation. Il s'agit en particulier de la représentation des unités lexicales pour l'analyse syntaxique des termes et de la version de l'algorithme d'Earley implantée.

8.2.1 Unités lexicales et l'analyseur syntaxique

Une unité lexicale (appelé de maintenant tout simplement un *lexème*) est une structure de donnée qui est retournée à chaque appel de l'analyseur lexical. Une chaîne de lexèmes est examinée par le pré-processeur et est ensuite passée à l'analyseur syntaxique. L'analyseur syntaxique transmet les lexèmes en arguments aux actions sémantiques appropriés. Une représentation compacte des lexèmes peut ainsi diminuer considérablement le temps de compilation. La représentation des lexèmes utilisée dans notre implantation est non standard à cause de la gestion spéciale des identificateurs et constantes numériques exigée par la définition du langage.

Les types des lexèmes utilisés en ELAN sont:
- caractères, pour tous les caractères différents des chiffres, lettres et caractères blancs.
- constantes numériques, pour toutes les chaînes de chiffres.
- identificateurs, pour tous les chaînes de lettres et de chiffres commençant par une lettre.

Nous pouvons voir qu'au niveau lexical, nous ne faisons pas de distinction entre les mots clés et les identificateurs. A priori chaque identificateur est vue comme un mot clé. S'il ne l'est pas il est introduit dans le tableau des mots clés par l'analyseur lexical et c'est l'analyseur syntaxique qui se charge de voir les mots clés aussi comme des identificateurs. Remarquons que la même situation arrive avec les constantes entières, puisqu'au niveau des signatures ELAN, l'utilisateur peut utiliser une sorte spéciale *<number>* représentant n'importe quel entier, mais en même temps, il peut utiliser des constantes numériques faisant partie de la grammaire. L'analyseur lexical retourne pour chaque nombre sa valeur effective comme valeur de lexème. De même que pour les identificateurs, c'est l'analyseur syntaxique qui est chargé de voir chaque entier particulier aussi comme un lexème de la sorte générale *<number>* .

Une représentation des lexèmes habituellement utilisée dans les compilateurs, représente un lexème par un couple contenant la sorte du lexème (caractère, mot clé, identificateur, nombre) et une partie avec la valeur effective du lexème. Si la sorte du lexème est par exemple "un identificateur", la partie de la valeur effective contient l'identificateur lui-même. En ELAN un lexème est codé par un seul nombre entier. L'idée du codage vient du fait, que le nombre des identificateurs est en effet restreint par la taille du tableau des identificateurs, égale à la constante MAXIDENT. Puisque dans l'interpréteur il n'y a qu'un seul tableau des identificateurs, nous pouvons représenter chaque identificateur par son indice et donc par un nombre entre zéro et MAXNIDENT. Le nombre des caractères spéciaux est aussi fini et nous pouvons donc coder les identificateurs et les caractères spéciaux dans différents intervalles positifs d'entiers. Les valeurs numériques étant non-restreintes, nous les codons par des valeurs négatives d'entiers.

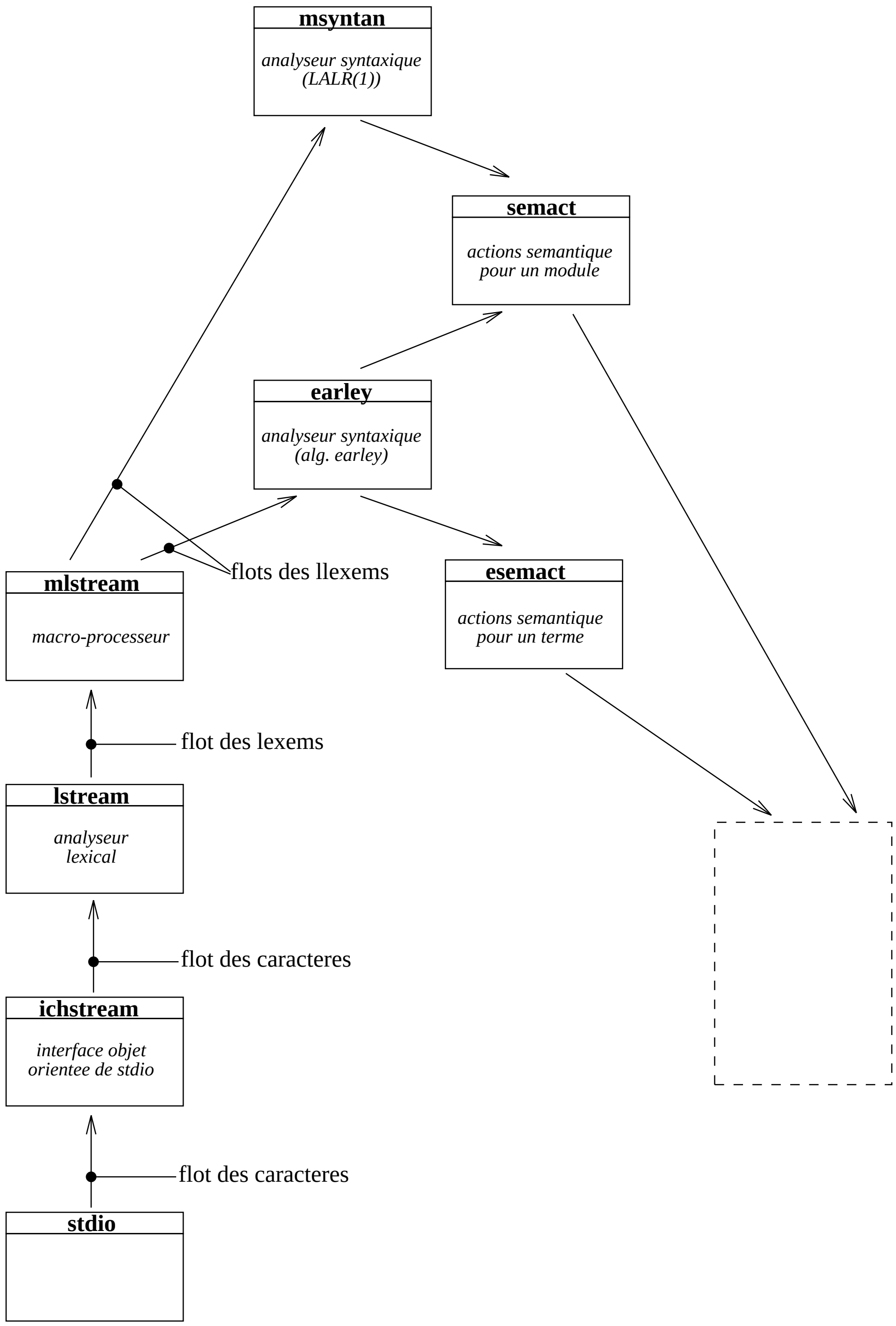


Figure 8.2. La partie syntaxique de l'interpréteur. Les conventions pour les positions relatives des cadres plus haut où plus bas et pour les directions des flèches sont les mêmes que dans la figure précédente. Les cadres représentent des classes des données (classes C++) et les noms marqués dans la partie supérieure des cadres référencent les noms des fichiers qui implantent ces classes. De plus, sur quelques flèches nous avons précisé quel genre de données sont transmies par ces voies.

En particulier les caractères spéciaux sont codés dans l'intervalle compris entre zéro et 255. Ils sont représentés directement par leurs codes ASCII, ce qui simplifie encore plus le texte de l'analyseur lexical. Les identificateurs sont codés dans un intervalle à partir de la constante BEGIDENTS, représentant le début d'intervalle pour les identificateurs, jusqu'au BEGIDENTS+MAXIDENT. L'intervalle entre 255 et BEGIDENTS est utilisé pour quelques lexèmes spéciaux comme la fin du fichier, un lexème spécial blanc, etc.

Cette représentation des lexèmes est plus compacte que la représentation habituelle. Nous précisons aussi que le tableau des identificateurs qui stocke les valeurs effectives des identificateurs n'est consulté en réalité que pendant l'analyse lexicale et pendant l'affichage des termes. Dans toutes les autres parties du programme (telle que l'analyseur syntaxique,...), il suffit d'utiliser les indices codés dans les lexèmes.

Algorithme d'Earley avec résolution des conflits et gestion des unités spéciales

Dans notre implantation d'ELAN, nous avons décidé d'utiliser l'algorithme d'Earley pour effectuer l'analyse syntaxique des termes. Cet algorithme d'analyse syntaxique pour la classe des grammaires hors contexte a été proposé par J. Earley dans [Ear70]. Les complexités de l'algorithme sont n^2 au niveau de l'espace et n^3 au niveau du temps pour une analyse syntaxique d'une chaîne de n lexèmes. Ces complexités ne dépendent pas de la taille de la grammaire. Donc, l'algorithme d'Earley est suffisamment puissant pour nos besoins, avec des complexité satisfaisantes. De plus il est conceptuellement assez simple, et plus facile à implanter que quelques autres algorithmes généraux, par exemple l'algorithme général dérivé des analyseurs syntaxiques LALR(k) [Tom85] implanté dans le système SDF [HHKR89]. Il est également possible de l'adapter à la gestion spéciale des identificateurs et des nombres requise par la définition du langage ELAN et il n'est également pas difficile d'ajouter une simple résolution des conflits syntaxiques par rapport aux priorités et associativités syntaxiques.

Dans l'implantation, une chaîne de lexèmes destinée à l'analyse syntaxique est stockée dans un tableau et l'ensemble des situations d'Earley est calculé pour chaque position de ce tableau. Si une situation "$\sigma \Rightarrow x.$" avec σ étant le non-terminal de départ est présente dans l'ensemble correspondant au dernière lexème, la chaîne des lexèmes est acceptée et la deuxième passe de l'algorithme démarre à partir de cet ensemble pour obtenir l'arbre de dérivation.

Par rapport à l'algorithme standard décrit dans [Ear70], nous avons effectué quelques modifications. Premièrement nous avons adapté l'algorithme à la gestion des identificateurs et des nombres quelconques exigée par les grammaires ELAN. Le changement concerne le première passe d'analyse syntaxique, c'est-à-dire la construction de l'ensemble de situations correspondants à un lexème l. Si l est un lexème décrivant un identificateur particulier, notre analyseur syntaxique engendre un ensemble de situations standard, mais il ajoute dans cet ensemble toutes les situations obtenues en voyant l comme un "identificateur quiconque", c'est-à-dire un lexème correspondant à la sorte *<t_identifier>* d'ELAN. Le même changement concerne les constantes numériques, où pour chaque constante numérique l'analyseur ajoute des situations correspondant à la sorte *<number>* d'ELAN.

La deuxième modification de l'analyseur syntaxique est impliquée par la nécessité de résoudre des ambiguïtés de la grammaire par rapport aux options syntaxiques des opérateurs ELAN. Ce changement concerne la construction de l'arbre syntaxique à partir d'ensembles de situations, c'est-à-dire la deuxième passe de l'algorithme. Si le calcul des positions acceptables pour un noeud de l'arbre retourne plus qu'une seule possibilité, une procédure de résolution des ambiguïtés est appelée. Seulement dans le cas où cette procédure retourne aussi plusieurs possibilités, un message de warning est affiché et une des possibilités est choisie au hasard.

8.3 Partie exécution

La partie exécution de l'interpréteur d'ELAN s'occupe essentiellement de l'évaluation d'une requête. Étant donné une stratégie et un terme de départ, la procédure d'évaluation interprète la stratégie en appliquant les règles de réécriture. En même temps, elle doit stocker la trace d'évaluation pour le cas de chaînage arrière. Même si un résultat est obtenu, c'est-à-dire si la stratégie est évaluée jusqu'au but, la trace d'exécution doit être conservée pour pouvoir continuer l'évaluation afin d'obtenir d'autres résultats. Cette trace d'exécution (que nous appelons aussi l'état d'exécution) représente donc la structure de donnée majeure de la partie d'exécution.

L'implantation de l'état d'exécution est réalisée par la définition d'une classe C++ nommée $stateOfExecution$ avec les méthodes suivantes:

new	: prenant un terme de départ et une stratégie, elle retourne un état d'exécution initial.
nextsolution	: pour un état d'exécution, elle retourne une valeur booleenne signalant s'il y a un autre résultat calculable à partir de cet état. Si la valeur booléenne est *true*, la méthode *nextsolution* retourne aussi le terme résultat et un nouveau état d'exécution obtenu après le calcul de ce résultat.
delete	: n'effectue aucun calcul, elle détruit l'état d'exécution et libère la mémoire occupée.

La représentation interne d'un état d'exécution suit la stratégie pour laquelle il était créé. Pour expliquer cette représentation, nous appelons dans le texte suivant une stratégie *élémentaire* une stratégie qui n'est pas obtenue par concaténation.

Pour une stratégie $s = s_1 \cdots s_n$ formée par une concaténation de n sous-stratégies élémentaires, l'état d'exécution est une liste de n éléments, chaque élément représentant un état particulier représenté par une classe particulière C++ pour chaque type de stratégie élémentaire. Chaque état pour une stratégie élémentaire est lui aussi muni des méthodes *new, nextsolution* et *delete* comme c'était le cas pour l'état majeur.

Par exemple l'état d'évaluation pour une stratégie élémentaire **dont know choose**(ℓ) de choix d'une règle de réécriture

$$\ell : g \to d \qquad c_1 \cdots c_k$$

où $c_1 \cdots c_n$ est une chaîne de conditions et d'affectations locales est représenté par l'état d'évaluation de l'algorithme de filtrage et par une liste de k états d'exécutions pour des affectations locales. De façon similaire sont implantés aussi des états d'évaluations pour les autres stratégies élémentaires.

L'implantation de la méthode *nextsolution* de la classe $stateOfExecution$ est alors basée sur le mécanisme de chaînage arrière qui énumère toutes les solutions d'une application d'une stratégie sur un terme. Toutes les structures de données sont adaptées à ce mécanisme en utilisant un schéma unique d'implantation avec les trois méthodes majeures pour créer une classe, obtenir la solution suivante et détruire la classe.

Remarquons que la mémorisation de tous les états d'exécution est énormément coûteuse en mémoire. La consommation de la mémoire est presque proportionnelle au temps d'exécution. Ce qui n'est pas acceptable, pour le cas des applications de stratégies déterministes. Si une évaluation est déterministe (elle n'était effectuée que par des stratégies déterministes), il est inutile de garder son état d'exécution dans la mémoire, puisque de toute façon nous savons qu'il n y aura pas d'autre solution. Nous avons donc implanté un mécanisme qui suit les évaluations déterministes et qui libère la mémoire allouée par ces évaluations dès que possible.

Dans les sections suivantes nous décrivons en particulier la méthode de gestion de la mémoire utilisée et la représentation d'un terme où nous mentionnons aussi l'implantation de la stratégie de normalisation.

8.3.1 Ramasse-miettes

Comme d'habitude dans les implantations de ce genre de langages, nous devons résoudre le problème de la gestion de mémoire. Plusieurs méthodes sont connues pour gérer l'espace mémoire ainsi que ses allocations et récupérations.

Un groupe d'algorithmes de gestion de la mémoire alloue la mémoire (sans la récupérer) jusqu'au le moment où il n y plus de place. A ce moment, ils appellent une procédure de *ramassage de miettes* qui repére toutes les données encore utilisées dans le programme, les garde et récupère le reste de la mémoire pour les prochaines allocations. Les procédures de *ramassage de miettes* peuvent être de différents groupes. Les procédures *sur place*, qui ne bougent pas les parties de la mémoire utilisées et qui fonctionnent bien en général si les morceaux de la mémoire utilisés sont tous de la même taille. L'autre groupe représente les procédures *compactantes*, qui déplacent tous les morceaux utilisés à un bout de la mémoire en obtenant ainsi une grande place libre à l'autre bout. Ces procédures marchent pour les morceaux de mémoire de tailles différentes, mais elles doivent changer tous les pointeurs dans le programme qui pointaient sur les morceaux déplacés.

L'autre approche (et c'est celle, que nous avons implanté en ELAN) est basée sur un *compteur de références*, c'est-à-dire chaque morceau de la mémoire contient un *compteur*, qui indique le nombre de pointeurs qui pointent sur lui. Si le compteur descend à zéro le morceau de mémoire est libéré. Cette approche est conceptuellement simple, avec l'avantage que l'allocation et la récupération de la mémoire est continue. Dans les méthodes précédentes, pendant le *ramassage de miettes* l'évaluation d'un programme doit attendre. Le désavantage général de cette méthode est qu'elle ne peut pas récupérer les structures avec des cycles (ce qui n'arrive heureusement pas dans le cas des termes ELAN).

Un autre avantage de cette méthode est qu'elle ne nous a pas obligé à implanter la gestion de la mémoire toute entière. En effet, pour l'allocation et pour la libération des morceaux de la mémoire avec le compteur zéro, nous utilisons les fonctions standards *malloc* et *free* du compilateur C++.

Cette gestion de la mémoire qui est à moitié gérée par la bibliothèque standard C++ et à moitié à la main est sûrement moins efficace qu'une gestion entièrement implantée à la main. Mais puisqu'il s'agit d'une implantation expérimentale du langage ELAN, la facilité d'implantation nous a semblé prioritaire sur la vitesse d'exécution.

8.3.2 La représentation des termes et la normalisation

Les termes représentent la structure de données fondamentale pour l'implantation du langage du type ELAN. Les termes dans notre implantation sont représentés d'une façon standard par un graphe acyclique orienté (nous l'appelons DAG) où les noeuds représentent des sous-termes. Chaque noeud est lui même composé de deux parties: la spécification de la tête du terme et un tableau de pointeurs sur les sous-termes. Chaque tête de terme contient un compteur de références pour la procédure de récupération de la mémoire.

Ceci nous donne donc la représentation des termes suivante. Un terme est représenté par une *structure* avec les *champs* suivants:

infos et *fsymi* - le champ *info* contient l'information sur la nature du terme, il peut contenir les valeurs suivantes: opérateur normal (dans ce cas, le *fsymi* contient l'indice de la description de ce symbole dans le tableau des symboles) , identificateur (*fsymi* est un indice dans le tableau des identificateurs), nombre (*fsymi* contient la valeur de ce nombre) et variable (*fsymi* est utilisé pour identifier la variable).

counter - un compteur de références, qui indique le nombre de pointeurs sur ce terme. Si le compteur descend jusqu'à zéro le terme sera détruit et la mémoire qu'il occupe sera libérée.

subterms - est un pointeur sur le tableau des sous-termes. La taille de ce tableau dépend de l'arité du symbole.

Dans notre représentation des termes, nous avons défini le champ *subterms* comme un pointeur sur un tableau de sous-termes et pas comme un tableau tout court. Ceci est motivé par l'implantation de la stratégie de normalisation où nous voulons optimiser les réductions pour les noeuds partagés à plusieurs positions d'un terme. Imaginons par exemple, que nous avons un terme s contenant un sous-terme t à plusieurs positions différentes mais représentées par un seul noeud du DAG. Si nous réduisons maintenant le sous-terme t en un terme t' par une règle de réécriture sans nom, nous voulons remplacer à la fois toutes les occurrences de t dans s par t'. Cela peut être réalisé simplement en plaçant le noeud représentant le terme t' exactement au même endroit dans la mémoire où était avant le noeud représentant le terme t. Mais cela ne peut être fait que dans le cas où les représentations des deux noeuds représentant les termes t et t' ont la même taille en mémoire. Ce qui est le cas dans notre représentation des termes et qui n'aurait pas été vrai dans une représentation avec un tableau des sous-termes faisant partie du noeud, puisque les noeuds avec des arités différentes auraient été représentés par des enregistrements de tailles différentes. Nous illustrons cette technique sur un exemple dans la figure 8.3.

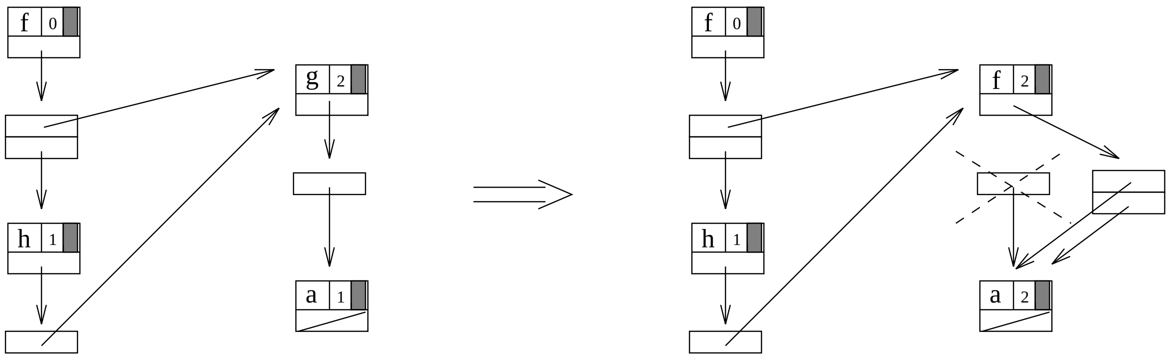


Figure 8.3. Réalisation d'un pas de réécriture réécrivant le terme $f(g(a), h(g(a)))$ sur le terme $f(f(a,a), h(f(a,a)))$ par une application de la règle $g(x) \to f(x,x)$

9
Conclusion

Dans cette thèse nous avons présenté un nouveau mécanisme d'évaluation basé sur les systèmes de calcul. Ce mécanisme dérivé de la réécriture effectue une recherche, raffinée à l'aide de stratégies, dans l'espace des dérivations de la logique de réécriture. Il s'est avéré bien approprié pour coder des logiques et résolveurs de contraintes, ce qui était l'objectif principal de son développement.

Sur la base de ce mécanisme d'évaluation, nous avons défini et implanté le langage ELAN, qui permet de décrire les systèmes de calcul en utilisant une syntaxe définie par l'utilisateur. La description des systèmes de calcul est divisée en modules qui peuvent être paramétrés. De plus, ce langage contient des outils spéciaux pour coder différentes logiques dans un système de calcul.

La modularité du langage nous a permis de développer une bibliothèque de modules standards, contenant des procédures d'utilité générale.

Nous concluons maintenant cette thèse par quelques perspectives de développement de ce langage.

Nous envisageons la formalisation du codage d'une logique objet dans un système de calcul ou dans une autre logique. Nous avons abordé ces aspects dans l'introduction de la thèse, le sujet est également abordé dans [MOM93], néanmoins les moyens techniques d'expression d'un tel codage ne sont pas encore explorés. En ELAN, nous utilisons pour l'instant un mécanisme de prétraitement syntaxique, ce mécanisme devrait être prochainement remplacée par un mécanisme dont la sémantique soit bien comprise.

L'autre point concerne la bibliothèque d'ELAN en particulier les modules pour coder différentes logiques objet. Il nous semble important d'élargir cette partie de la bibliothèque. Pour définir la bibliothèque actuelle nous avons procédé d'abord par nécessité. Ce n'est qu'ensuite que nous avons essayé de systématiser les notions définies. Il nous semble donc intéressant de poursuivre cette systématisation et d'essayer de faire la bibliothèque la plus générale possible. Cela nécessitera une réflexion sur la représentation des notions de logiques incluses et sur les fonctions nécessaires pour coder facilement différentes procédures de déduction. Par exemple, la bibliothèque actuelle contient peu de support pour les manipulations de variables, telles que le renommage dans les termes et les clauses.

Le langage ELAN lui même devrait encore évoluer. Les possibilités ne concernent pas uniquement le mécanisme du pré-traitement syntaxique, que nous avons déjà mentionné. D'autres points pratiques peuvent concerner l'élargissement du langage de stratégies actuel. Le premier changement serait d'introduire la possibilité d'appeler une stratégie dans le corps d'une autre.

Les règles de réécriture utilisées en ELAN agissent sur des classes d'équivalence des termes modulo une théorie équationnelle. Actuellement la seule théorie admise (à part la théorie vide) est la théorie associative-commutative. Les possibilités d'expression du langage peuvent être augmentées en ajoutons d'autres possibilités. Un élargissement aux théories uniquement associatives ou uniquement commutatives est envisagé.

Dans ce contexte, il nous semble intéressant de développer une théorie équationnelle que nous appelons *théorie des listes finies*. Cette théorie devrait formaliser la notion des pointillés ($\cdots$) souvent utilisés dans la littérature. Par exemple dans une telle théorie le terme $x_1, \cdots, x_n$ devrait filtrer le terme a, b, c, d avec une substitution $\sigma = \{n/4, x_1/a, x_2/b, x_3/c, x_4/d\}$. La possibilité d'introduction de cette théorie au niveau des signatures d'ELAN permettrait d'augmenter largement la puissance d'expression du langage. Néanmoins, l'introduction d'une telle théorie nécessiterait des changements majeurs au niveau du langage. Par exemple les déclarations des variables dans les règles de réécriture devront pouvoir spécifier un ensemble potentiellement infini de variables. Une possibilité est l'utilisation de définitions similaires au système ASF+SDF [Kli93] où un tel ensemble peut être défini en utilisant les expressions régulières dans les déclarations, par exemple écrire $x[0-9]^*$ pour pouvoir utiliser les variables $x_1, x_2, \cdots$.

Par ailleurs, il faudra envisager la division du fichier contenant la spécification (programme dans le langage spécifié) en différents fichiers. Une spécification en ELAN est formée de différentes parties, par exemple par une partie signature Σ et une partie axiomes Γ. Au moment de la compilation l'interpréteur d'ELAN génère, à l'aide du pré-traitement, un système de calcul correspondant à ce programme (Σ, Γ). En réalité, la plus grande partie du système de calcul (telle que l'ensemble de termes, l'algorithme d'unification) ainsi généré ne dépend pas de Γ mais uniquement de la signature Σ. Malgré cela après chaque changement de Γ, tout est recompilé. Si le système de calcul correspondant à (Σ, Γ) était généré en deux temps, d'abord le système de calcul dépendant uniquement de Σ, puis le reste, nous pourrions gagner du temps lors de la recompilation du programme.

Une autre économie de temps peut être réalisée grâce à l'utilisation de la technique de memorisation des résultats qui consiste à mémoriser des calculs effectués. Ainsi, si nous devons effectuer une deuxième fois le même calcul, nous n'avons qu'à regarder dans la mémoire pour obtenir les résultats. Cette technique peut être très efficace si nous avons besoin d'effectuer souvent les mêmes calculs; c'est le cas par exemple de la procédure de pré-traitement syntaxique où nous avons souvent besoin de l'ensemble des symboles de fonction pour générer des morceaux de texte. Cet ensemble pourrait être calculé une seule fois et mémorisé ensuite pour les utilisations prochaines.

Le développement d'autres implantations du langage peut être aussi intéressant. Il nous vient à l'esprit une implantation possible sur une architecture parallèle ou le développement d'un compilateur d'ELAN. Un compilateur serait intéressant surtout pour augmenter l'efficacité de l'exécution des programmes ELAN. Grâce à la modularité du langage, une compilation séparée de chaque module pourrait aussi augmenter l'efficacité de la phase de compilation en particulier lorsque les changements effectués entre deux compilations ne concernent que quelques modules; alors seuls les modules concernés devraient être recompilés. Ainsi, par exemple, la bibliothèque d'ELAN pourrait être entièrement pré-compilée.

- FIN -

Bibliographie

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1986.

[Com90a] H. Comon. Equational formulas in order-sorted algebras. In Paterson, editor, *Proceedings ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 674–688. Springer-Verlag, 1990.

[Com90b] H. Comon. Solving inequations in term algebras. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 62–69, June 1990.

[Com91] H. Comon. Disunification: a survey. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9, pages 322–359. The MIT press, Cambridge (MA, USA), 1991.

[CPMDH89] T. Coquand, C. Paulin-Mohring, G. Dowek, and G. Huet. The Calculus of Constructions. Documentation and user's guide 110, INRIA-ENS, Rocquencourt, 1989.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, New York (NY, USA), 1980.

[Dek93] A. Dekdouk. Etude de l'implantation de la disunification. Mémoire de DEA en informatique, Université de Nancy 1, September 1993.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[Fef88] Solomon Feferman. Finitary inductive systems. In R. Ferro, editor, *Proceedings of Logic Colloquium, Padova (Italy)*, pages 191–220. North-Holland, 1988.

[FGJM84] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. The language OBJ-2: Its syntax, semantics and implementation. Technical report, SRI International, Computer Science Laboratory, 1984.

[GKK+87] J. A. Goguen, Claude Kirchner, Hélène Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[GMW79] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York (NY, USA), 1979.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 333, Ravenswood Ave., Menlo Park, CA 94025, August 1988.

[Hab92] Marianne Haberstrau. ECOLOG: An Environment for COnstraint LOGics. In Eric Domenjoud and Claude Kirchner, editors, *Proceedings of CCL'92*, Le Val d'Ajol, 1992.

[Hab93] Marianne Haberstrau. *ECOLOG: un Environnement pour la programmation en LOGique COntrainte*. Thèse de Doctorat d'Université, Université de Paris-Sud, 1993.

[Habar] Marianne Haberstrau. ECOLOG: An Environment for COnstraint LOGics. In *Proceedings of the First International Conference CCL'94*, Munich, to appear.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalisme SDF - reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[HHP86] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. Technical report, Laboratory for Foundations of Computer Science University of Edinburgh, December 1986.

[Hul80] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, July 1980.

[JK91] J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[JM94] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, xx(yy), 1994. to appear.

[Kir85] Claude Kirchner. *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles*. Thèse de Doctorat d'Etat, Université de Nancy 1, 1985.

[KKV92] Claude Kirchner, Hélène Kirchner, and M. Vittek. Implementation of computational logics with constraints or ECOLOG-Nancy, a meta-language for specifing logic programming languages. In E. Domenjoud and Claude Kirchner, editors, *Proceedings of CCL'92*, Le Val d'Ajol, 1992.

[KKV93a] Claude Kirchner, Hélène Kirchner, and M. Vittek. Implementing computational systems with constraints. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming, Providence (R.I., USA)*, pages 166–175. Brown University, 1993.

[KKV93b] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In F. Orejas, editor, *Proceedings of the 2nd CCL Workshop, La Escala (Spain)*, September 1993.

[KKV95] Claude Kirchner, Hélène Kirchner, and M. Vittek. Designing CLP using Computational Systems. In P. Van Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming.* The MIT press, 1995.

[Kli93] P. Klint. The ASF+SDF Meta-environment User's Guide. Technical report, CWI, 1993.

[Mes89] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[Mes91] José Meseguer. Parallel programming in Maude. Technical report SRI-CSL-91-08, SRI International, 1991.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes93] J. Meseguer. A logical theory of concurrent objects and its realisation in the Maude language. In Agha, Wegner, and Yonezawa, editors, *Research Directions in Object-Based Concurrency.* The MIT press, 1993.

[MM76] A. Martelli and U. Montanari. Unification in linear time and space: A structured presentation. Technical Report B76-16, University of Pisa, July 1976.

[MN86] D. A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, volume 225 of *Lecture Notes in Computer Science*, pages 448–462. Springer-Verlag, 1986.

[MNRA92] J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: the language BABEL. *Journal of Logic Programming*, 12(3):191–223, February 1992.

[MOM93] N. Martì-Oliet and J. Meseguer. Rewriting logic as a logical and semantical framework. Technical report, SRI International, May 1993.

[Mor94] P.E. Moreau. Completion avec contraintes en ELAN. Mémoire de DEA en informatique, Université de Nancy 1, September 1994.

[NR92] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In D. Kapur, editor, *Proceedings of CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 477–491. Springer-Verlag, 1992.

[Pau89] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[Pfe91] F. Pfenning. Logic programming in the lf logical framework. In G. Huet and Plotkin G. D., editors, *Logical Frameworks.* Cambridge University Press, 1991.

[Sak90] Kô Sakai. *On Computer Aided Mathematical Reasoning.* PhD thesis, Institute for New Generation Computer Technology, 1990.

[Tom85] M. Tomita. Efficient parsing for natural languages. Kluwer Academic Publishers, 1985.