MASTER EN SCIENCES INFORMATIQUES

Génération de Prototypes PROLOG par Transformations de Spécifications Formelles

De Meulemeester, Rita; Laloy, José

Award date:
1989

Awarding institution:
Universite de Namur

Année académique 1988 - 1989.

Génération de Prototypes PROLOG

par Transformations de

Spécifications Formelles

Mémoire de fin d'études en vue de l'obtention du diplôme
de licence et maîtrise en informatique.

Rita DE MEULEMEESTER
José LALOY
Promoteur : A. VAN LAMSWEERDE

Facultés Universitaires Notre-Dame de la Paix
Faculté des Sciences
rue de Bruxelles 61, B-5000 **NAMUR**
Tél. 081-22.90.61

Télex 59222 facnam-b
Téléfax 081-23.03.91

Génération de prototypes Prolog par transformations de spécifications formelles.

DE MEULEMEESTER Rita
LALOY José

Résumé.

Le prototypage rapide de spécifications a pour but de vérifier l'adéquation d'une spécification aux besoins réels de l'usager.

Un outil de prototypage a été implémenté, transformant des spécifications formelles écrites en RSL (Requirement Structuring Language) en procédures PROLOG exécutables. Cette transformation, réalisée de façon semi-automatique, est effectuée par l'application de deux transformations. La première réécrit la spécification sous forme de relations définies en logique des prédicats du premier ordre. La deuxième transforme ces relations en procédures Prolog en tenant compte des caractéristiques particulières de ce langage et du mode souhaité d'utilisation du prototype. La structure du processus d'explicitation progressive, suivi pour l'élaboration de la spécification, est préservée dans les prototypes produits.

L'implémentation est réalisée à l'aide du Cornell Synthesizer Generator, un générateur d'environnements basés sur un éditeur syntaxique spécifié au moyen d'une grammaire attribuée.

Abstract.

Transformation Of Formal Specifications To Generate PROLOG Prototypes.

Rapid prototyping of specifications aims at checking that a specification meets the user's requirements.

A prototyping tool has been implemented to transform formal specifications written in RSL (Requirement Structuring Language) into PROLOG procedures. The transformation is carried out semi-automatically using two transformations. The first one rewrites a specification as a set of relations defined in the first order predicate logic. The second one transforms those relations into PROLOG procedures, taking into account the PROLOG execution mechanism and the prototype mode of use. The structure of the stepwise refinement process used to elaborate the specification is preserved in the produced prototypes.

Implementation was carried out using the Cornell Synthesizer Generator, which generates environments based upon a syntactic editor specified with an attribute grammar.

Mémoire de licence et maîtrise en Informatique
Année Académique 1988-1989
Promoteur : Prof. A. VAN LAMSWEERDE

Nous remercions Mr Van Lamsweerde d'avoir accepté d'être le promoteur de ce mémoire. Nous avons apprécié ses conseils et suggestions.

Nous exprimons notre plus sincère gratitude à Mr Habra pour sa constante disponibilité, son infinie patience, sa compréhension et son apport constructif à notre travail.

Nos remerciements vont encore à Mr Crismer pour l'intérêt qu'il a porté à ce que nous faisions, et pour le zèle qu'il a manifesté à résoudre nos problèmes "cornelliens".

Que tous ceux et celles qui nous ont apporté aide et soutien tout au long de ce travail trouvent ici l'expression de notre reconnaissance.

Table des matières

Introduction

Dans le cycle de vie d'un système informatique, il est possible de distinguer six étapes [VAN LAMSWEERDE 82]:

- l'analyse des besoins ou analyse d'opportunité,
- la spécification fonctionnelle,
- la conception,
- le codage,
- la conduite des tests, l'intégration et la mise au point,
- la maintenance.

Parmi ces étapes, l'étape de spécification occupe une place cruciale. En effet, elle a pour but de fixer avec précision les exigences des clients ayant commandé le système. Les spécifications qui en sont le résultat reflètent le compromis auquel arrivent demandeurs et concepteurs quant aux caractéristiques du système à développer. Elles sont la base du contrat liant les deux parties. C'est par conséquent par rapport aux spécifications qu'on vérifiera la correction du système réalisé. Disposer de spécifications de bonne qualité (complètes, non ambigües, non contradictoires, ...) peut faciliter la réalisation du système et finalement sa maintenance car les spécifications constituent la base de la documentation. Enfin, quand on sait qu'il est d'autant plus coûteux de revenir sur une décision - fût-ce pour corriger une erreur - qu'elle a été prise plus tôt dans le processus de développement, on conçoit aisément qu'il est important d'apporter un soin tout particulier à l'élaboration des spécifications puisqu'elles conditionnent ce qui vient en aval.

Elaborer des spécifications de bonne qualité n'est pas une tâche facile. Clients et informaticiens doivent parvenir, les uns à exprimer leurs exigences, les autres à comprendre et rencontrer celles-ci. Pour faciliter une approche rigoureuse et précise du problème de spécification, des langages formels ont été développés. Ceux-ci ont l'avantage de permettre l'automatisation de certains contrôles; ils ont souvent le désavantage d'être peu compréhensibles par les non-spécialistes. Pour pallier cet inconvénient, il est souvent utile de présenter aux clients une version partielle du système à réaliser, n'en couvrant que certains aspects tout en en donnant une idée réaliste, permettant également aux clients de clarifier leurs exigences et de valider

les spécifications. Le coût de développement de ce système expérimental (prototype) doit être faible par rapport au coût total. Le prototypage rapide vise à la production de tels systèmes dans des délais et à des coûts acceptables. Il fait l'objet du premier chapitre de ce mémoire.

Diverses tâches de spécification peuvent être identifiées dans le développement d'un système logiciel complexe [SOMMERVILLE 85].

La spécification du modèle conceptuel présente une vue générale des différents services que le système fournit et met en évidence la façon dont ces services s'agencent.

La spécification des fonctionnalités du système détaille chaque service que le système doit offrir, chaque fonction qu'il doit remplir.

On peut envisager là différentes sous-tâches de spécification et établir une distinction entre

- la spécification d'un noyau du système, formé de la description des objets/opérations dont la définition est stable, et
- la spécification de l'enveloppe du système, formée de la description des objets/opérations dont la définition risque de changer pendant la vie du logiciel [DUBOIS 85].

La spécification des contraintes non fonctionnelles fixe les contraintes sous lesquelles le système doit opérer (temps de réponse, place occupée, droits d'accès, etc.) et met ces contraintes en relation avec les exigences fonctionnelles.

La spécification de l'environnement décrit les objets et opérations manipulés par l'environnement avec lequel le système interagit. Par exemple, on est ainsi souvent amené à spécifier la structure logique d'une base de données. Une telle spécification décrit l'organisation logique des données utilisées par le système et les relations qui existent entre ces données [HAINAUT 86].

La spécification du comportement du système peut se présenter sous forme d'un ou plusieurs scénarios à soumettre à l'utilisateur.

La spécification de l'interface usager enfin peut faire l'objet d'une étude séparée, particulièrement importante lors du développement de systèmes interactifs [SHNEIDERMAN 87].

Pour répondre au problème que pose l'élaboration de spécifications de bonne qualité, différents langages de spécification ont été utilisés. Le but d'un langage de spécification est de permettre l'expression d'une définition précise et rigoureuse du système à développer.

La langue naturelle est utilisée comme langage de spécification. Son pouvoir d'expression est illimité. Cependant, elle a l'inconvénient de favoriser l'apparition d'ambiguïtés, voire de contradictions. Souvent, dans une spécification en langue naturelle, les exigences fonctionnelles, non fonctionnelles, et les buts poursuivis ne sont pas distingués. La vérification de la complétude et de la consistance d'une telle spécification est ardue, et ne peut être automatisée.

Des langages semi-formels ont été proposés, par exemple PSL/PSA [TEICHROW 77], ou SADT [SCHOMAN 77].

PSL/PSA est un langage dont la syntaxe est définie formellement, mais pas la sémantique. Il permet d'identifier des objets et de spécifier des relations entre ces objets, de collecter l'information recueillie dans une base de données manipulable par des outils logiciels permettant l'édition de rapports, la vérification syntaxique.

SADT est un langage basé sur une technique plus générale nommée l'analyse structurelle. L'analyse structurelle s'appuie sur des graphiques, des diagrammes de flux, des dictionnaires de données, des tables de décision pour exprimer des structures, des relations [DE MARCO 79]. Les représentations graphiques de SADT favorisent la communication des spécifications à l'utilisateur, mais ne peuvent toutefois pas se substituer à une définition de la sémantique de la spécification.

Enfin, des langages formels ont été développés, dont certains sont accompagnés d'une méthodologie de construction de spécifications.

Une spécification formelle est exprimée dans une notation dont le support est mathématique [WING 88]. La syntaxe et la sémantique du langage sont définies formellement, ce qui présente plusieurs avantages :

- la sémantique d'une spécification peut être établie par référence à celle du langage;
- la base mathématique permet de répondre à la question de l'équivalence de deux spécifications;
- des traitements automatisés poussés peuvent être envisagés;
- il est possible de prouver qu'un programme vérifie sa spécification.

Deux approches de l'élaboration de spécifications formelles peuvent être distinguées.

Dans l'approche opérationnelle, une spécification donne une définition constructive du système spécifié.

Dans l'approche définitionnelle, la spécification donne les propriétés du système, plutôt qu'une méthode de construction. Cette dernière approche couvre deux types de spécifications :

- les spécifications algébriques qui utilisent des axiomes exprimés sous forme d'équations pour spécifier les propriétés de programmes, ou de types abstraits [GOGUEN 78];
- les spécifications pré-post qui relèvent d'une approche dite axiomatique où des prédicats logiques exprimant des pré- et des postconditions sont utilisés pour la spécification.

Citons quelques langages formels : Z [SPIVEY 87] qui repose sur la théorie des ensembles, GIST [FEATHER 87] basé sur un modèle relationnel, et s'accompagnant d'une méthodologie de spécification; citons enfin le projet LARCH [LARCH] définissant une famille de langages formels.

L'élaboration de spécifications n'étant pas une chose facile, on a donc d'abord cherché à se donner des langages de spécification précis, favorisant la correction, expressifs, faciles à comprendre, ... Devant le constat de l'insuffisance de cette démarche, on a ensuite cherché à formaliser les processus sous-jacents à la construction d'une spécification et on s'est intéressé davantage à la méthode à suivre. Le chapitre 2 présente RSL (Requirement Structuring Language) [DUBOIS 84] [DUBOIS 85], un langage de spécification formel, doublé d'une description en langue naturelle pour la facilité de communication, accompagné d'une méthodologie de spécification. Une spécification RSL repose sur une bibliothèque de spécification contenant des types d'objets et des opérations pré-définis, et sur deux modèles complémentaires décrivant l'un les types d'objets du système, l'autre ses opérations. Lors de l'élaboration d'une spécification RSL, chaque type d'objet s'insère dans une hiérarchie; des constructeurs (produit cartésien, union disjointe, suite, ensemble, table) expriment les liens entre types composés et types composants. Chaque opération est vue comme une relation unissant son résultat à ses arguments et est spécifiée d'une manière constructive en termes d'opérations plus élémentaires, les liens entre celles-ci étant exprimés au moyen de connecteurs "and", "or", "forall", "with". Lorsque la construction maintient une structure des opérations parallèle à la structure des types, certaines relations entre objets peuvent être laissées implicites car elles peuvent être déduites à partir des types de ces objets.

L'objectif du présent travail est la réalisation d'un outil de prototypage permettant, à partir d'une spécification écrite en RSL, la production automatique ou semi-automatique d'un prototype exécutable en PROLOG. L'originalité de l'outil est qu'il supporte sous certaines conditions que des relations soient laissées implicites dans la spécification RSL.

Deux transformations préservant la sémantique sont appliquées à la spécification RSL d'une opération pour produire un programme PROLOG y équivalant. La première transforme la relation décrivant une opération et l'explicitation de cette relation en termes d'autres relations en une formule équivalente exprimée dans le langage des prédicats du premier ordre (FOPL). Cette transformation assure également que les relations qui ont été éventuellement laissées implicites dans la spécification initiale soient explicitées. Elle présente l'avantage d'être indépendante du langage de programmation cible: on peut imaginer la réutiliser dans la production de prototypes écrits non en PROLOG mais dans un autre langage, par exemple un langage fonctionnel tel ML [QUERE 88]. La deuxième transformation assure le passage à un programme PROLOG exécutable dont la terminaison est assurée et tel qu'il soit effectivement un prototype de l'opération initialement décrite. L'exposé théorique et la justification de ces deux transformations constituent la première partie des chapitres 4 et 5.

Pour implémenter les transformations, nous avons utilisé un outil manipulant des arbres syntaxiques attribués: le Cornell Synthesizer Generator (CSG) [REPS 87] [REPS 89], décrit au chapitre 3. Le CSG est en fait principalement conçu pour générer un éditeur syntaxique paramétré sur un langage qui lui est spécifié. La syntaxe du langage lui est fournie sous forme de règles de grammaire, sa sémantique y est ajoutée par des valeurs d'attributs. Il est intéressant de constater que les facilités qu'offre le CSG pour déclarer syntaxe et attributs, et pour calculer les valeurs de ceux-ci, peuvent être employées à d'autres fins que celles d'exprimer la sémantique d'un langage de programmation.

Nous avons utilisé le mécanisme des attributs pour implémenter d'une part la transformation de la spécification RSL d'une opération en formule FOPL équivalente, et d'autre part la transformation conduisant à un programme PROLOG équivalent. L'implémentation de chacune de ces transformations fait l'objet de la deuxième partie des chapitres 4 et 5.

Les limites de l'outil actuellement réalisé sont enfin exposées au chapitre 6, en même temps que quelques perspectives d'amélioration.

L'étude théorique des transformations présentées aux chapitres 4 et 5 est exposée dans [HABRA 88] et [HABRA 89] de même qu'une preuve de la correction de ces transformations. Le problème de l'explicitation des relations implicites dans la spécification RSL y est également résolu sous certaines hypothèses. La partie pratique de notre travail a consisté à implémenter cette description théorique.

Ce mémoire est le résultat d'un travail collectif. Nous avons réalisé ensemble la spécification du système de bibliothèque servant d'exemple au chapitre 2. Le choix des attributs, l'écriture des fonctions, etc. dont il est question dans les chapitres 3, 4 et 5 résultent d'un effort commun. De même, nous nous sommes partagés le travail de dactylographie du programme et la tâche d'amener le CSG à effectuer ce que nous avions imaginé. Seule la rédaction a été faite séparément. Les chapitres 1, 2, 3 et 6 ont été rédigés par R. De Meulemeester, tandis que les chapitres 4 et 5 sont de la plume de J. Laloy.

Chapitre 1

Le prototypage

1.1. Introduction.

Dans le développement d'un logiciel, l'étape d'élaboration des spécifications occupe une place fondamentale. En effet, les spécifications sont la base du contrat liant les clients et les concepteurs du système. Elles servent de définition par rapport à laquelle on testera la validité du produit. Par conséquent, il importe que ces spécifications correspondent au problème réel posé par les demandeurs, les utilisateurs du logiciel. Le prototypage est un moyen de vérifier l'adéquation des spécifications écrites aux exigences réelles des clients.

Bien qu'il soit possible d'établir des spécifications sans recourir à un formalisme particulier, disposer de spécifications formelles présente plusieurs avantages. L'utilisation d'un cadre formel impose une rédaction précise et rigoureuse qui force le spécifieur à analyser minutieusement tous les aspects du problème et l'aide à éliminer bruit, ambiguïtés et contradictions. Il permet aussi l'application de règles de vérification de propriétés et rend possible l'utilisation de traitements automatisés, par exemple pour tester complétude et cohérence. Des spécifications formelles présentent par contre le désavantage d'être peu compréhensibles pour les non-initiés. Cet inconvénient peut être pallié grâce au prototypage qui sert alors de moyen de communication entre les concepteurs du système et les clients.

De plus, il arrive que les demandeurs et les utilisateurs d'un logiciel n'aient d'une part pas une idée nette de ce qu'il faudrait attendre du système définitif. Parfois, les clients ne savent pas ce qu'ils veulent, ou ils éprouvent des difficultés à exprimer leurs exigences de manière précise, ou ils ignorent les possibilités qu'offre l'automation, ou ils ne peuvent prédire les modifications et extensions qui seront nécessaires par la suite. Parfois, les clients savent exactement ce qu'ils veulent et l'expriment clairement. D'autre part cependant, la solution technique à leur problème n'existe pas ou pas encore, et les informaticiens qui ne sont pas

nécessairement des spécialistes du domaine d'application des demandeurs peuvent ne pas rencontrer leurs exigences. De part et d'autre, il se peut donc qu'on ne réalise pas toute la portée des décisions de conception.

Le développement de prototypes est une technique fréquemment employée dans diverses disciplines. Construire une maquette d'un édifice à bâtir est une pratique courante en architecture. Dans l'industrie aéronautique, avant de produire un nouvel appareil, on teste sur un prototype s'il présente certaines caractéristiques indispensables. En génie logiciel, comme il est en général difficile et coûteux d'apporter des changements à des systèmes déjà implémentés, il est intéressant de donner aux clients, le plus vite possible dans le développement du logiciel, une idée réaliste de ce que sera le système fini. Le prototypage vise à rencontrer cette préoccupation.

1.2. Définition.

Un prototype est une version exécutable présentant une partie seulement des caractéristiques d'un système logiciel, destinée à en expérimenter les qualités avant la production du système lui-même. Le prototypage a pour objectif la production rapide et relativement bon marché d'un ou plusieurs prototypes.

1.3. Une classification des prototypages.

Il est possible de classer les prototypes selon différents critères.

1) les aspects mis en évidence par le prototype,

Un prototype logiciel sera généralement développé selon l'un des trois axes quasi orthogonaux suivants [WEISER 82]:

- performances en temps et en espace mémoire,
- interface usager
- fonctionnalités du système.

Dans la suite, nous n'envisagerons plus que cette dernière forme de prototypage.

2) l'usage qu'il sera fait ultérieurement du prototype et

3) les parties du système final couvertes par le prototype.

Ces deux derniers aspects sont souvent confondus et l'on distingue deux approches du prototypage.

Prototypage horizontal.

La première approche consiste à développer un modèle fonctionnel couvrant l'ensemble du système. Le but est de vérifier, pour un coût marginal faible que les spécifications satisfont les exigences du client. De ce prototype, on n'attend pas les performances de l'installation réelle. Une fois son objectif de validation atteint, le prototype sera abandonné. Une telle approche du prototypage s'insère facilement dans un cycle de vie traditionnel où chaque étape doit être terminée avant de passer à la suivante. Le coût consenti et le temps passé à élaborer le prototype une fois que l'on dispose de spécifications sont compensés tout au long du cycle de vie par la meilleure compréhension du problème qu'il a permis d'atteindre et par conséquent, les erreurs coûteuses évitées et la qualité accrue du produit final.

Prototypage vertical.

Une deuxième approche consiste à ne développer un prototype que de quelques fonctions, du moins dans un premier temps. Ce prototype initial est ensuite affiné et complété pour donner le système final. Les procédures peu efficaces seront optimisées ou remplacées et on étendra progressivement l'éventail des fonctions disponibles jusqu'à obtenir le système complet. Comme on évalue le prototype à chaque incrément, le produit final a davantage de chances de mieux correspondre aux attentes du client [BLUM 82].

Une classification alternative plus détaillée est proposée dans [HABRA 88] où les quatre points suivants permettent de qualifier les approches de prototypage.

"i. La (les) partie(s) du logiciel concernée(s) par le prototypage.

ii. Les aspects du logiciel représentés par le prototype
(adéquation des fonctionnalités, ergonomie, faisabilité, ...).

iii. La relation entre le prototype et la spécification
(" partie de " , " dérivé à partir de " , ...).

iv. La relation entre le prototype et le système final ("première version de", etc.)."

1.4. Caractéristiques d'un prototype intéressant.

Production rapide et bon marché.

Le prototypage n'est pas une fin en soi. S'il existait une méthode pour vérifier que des spécifications sont complètes, précises, respectent les contraintes fixées, et valider ces spécifications par rapport aux exigences réelles de l'utilisateur, le prototypage n'aurait pas de raison d'être [TAYLOR 82]. Le but ultime reste en l'occurrence la génération d'un système complet opérationnel le plus vite possible et au meilleur coût.

Modifiabilité.

Plusieurs cycles d'évaluation sont possibles avant d'arriver à une interprétation correcte des exigences du client. Si on veut réutiliser le prototype à chaque itération, il doit être aisément modifiable.

Aptitude à l'évaluation.

Il est souhaitable que le prototype soit une image fidèle de la spécification et/ou du système final. Il importe de s'assurer qu'il peut fournir ce qu'on attend de lui ; il faut donc lui consacrer un minimum de soin. Les questions auxquelles sa construction doit répondre doivent être clairement posées avant d'entreprendre cette construction [HEITMEYER 82].

Pour extraire d'un prototype des informations utiles, il faut l'exécuter sur une série de données tests et comparer les résultats obtenus avec les spécififcations. Une certaine forme de traçabilité est par conséquent souhaitable [TAYLOR 82].

Démontrabilité et aptitude à l'apprentissage.

Le prototype doit être suffisamment convivial pour faciliter la réaction des utilisateurs. S'il est représentatif du système final, les utilisateurs peuvent se familiariser avec le prototype en attendant ce système final [LANGELEZ 86].

1.5. Utilité du prototypage.

Expérimentation.

Le plus souvent, on développe un prototype pour se faire une expérience dans les domaines où règne le plus d'incertitude ou qui sont les plus importants pour la réalisation du projet [WEISER 82].

La place fondamentale qu'occupe l'élaboration des spécifications fait qu'il est intéressant d'utiliser le prototypage lors de cette étape [LANGELEZ 86].

Validation des spécifications.

Le prototype peut servir d'outil de communication entre l'analyste et l'utilisateur [DODD 82] . Il est souvent plus facile pour l'utilisateur de comprendre une exécution du prototype que de déchiffrer les spécifications. Le prototype permet d'évaluer le système en pratique et de suggérer des modifications [WASSERMAN 85].

C'est en général lors des premières utilisations que les inadéquations entre les fonctions proposées par un système et les exigences des demandeurs sont détectées. Lorsqu'un prototype est utilisé, la détection a lieu plus tôt et la mise au point est plus rapide et moins coûteuse.

Aide à la construction de la spécification.

Le spécifieur peut utiliser un prototype pour faire clarifier aux demandeurs leurs exigences fonctionnelles. L'expérience d'utilisation d'un système produit en général des idées quant aux améliorations possibles. Eventuellement, plusieurs variantes peuvent être proposées, et l'impact du choix de l'une ou l'autre peut être analysé.

S'il n'est pas dérivé des spécifications par des transformations préservant la sémantique, le prototype est une approximation de l'interprétation que l'implémenteur s'est construite de la spécification. En comparant le comportement du prototype à ce qu'il a voulu spécifier, le spécifieur peut vérifier sa spécification [COHEN 82].

Test de faisabilité.

Le prototypage permet également aux concepteurs de s'assurer de la faisabilité du système. Il fournit une preuve par construction qu'une implémentation est possible à partir des spécifications.

Préparation à l'intégration dans l'environnement.

Parmi les avantages du recours à un prototype, citons encore qu'on teste la spécification dans son environnement. En phase d'intégration, des parties du prototype peuvent parfois être employées comme modules souches ou conducteurs [CHOPPY 87].

Réduction du coût total

L'effort global de développement du système final est réduit car

1) on bénéficie de l'expérience acquise lors de la production du prototype [MAC EWEN 82] ; le système définitif peut parfois être raffiné à partir de celui-ci, et
2) le temps de test du logiciel final est diminué de manière significative car beaucoup d'erreurs de spécification et de conception sont découvertes et corrigées sur le prototype [KLAUSNER 82].

1.6. Inconvénients du prototypage.

Certains aspects importants du logiciel ne peuvent être mis en évidence par le prototypage. Le prototype reste un système incomplet. Il ne peut révéler ce que sera la maintenance du système définitif ni si celui-ci sera facilement modifiable; il ne permet pas de dire comment le système final se comportera si on le pousse à ses limites ou s'il interagira de manière harmonieuse avec les autres composants avec lesquels il devrait partager des données.

N'offrant qu'une vision partielle, le prototype peut induire les utilisateurs à mal percevoir le système final. En fait, un prototype met en évidence certaines caractéristiques mais il en escamote d'autres [TAYLOR 82].

Soulignons encore quelques pièges du prototypage.

On peut être tenté de substituer un prototype fonctionnellement correct au système définitif. Les coûts ultérieurs de maintenance de tels systèmes témoignent qu'il s'agit là d'une fausse économie [BLUM 82].

L'activité de prototypage ne remplace pas celle de spécification. Si cette dernière est négligée, il est vain d'attendre du prototype qu'il pallie cette lacune [LANGELEZ 86].

Certaines spécifications couvrent plusieurs comportements. Un prototype ne couvre qu'un seul de ces comportements possibles [FEATHER 82].

Des décisions de conception qui n'ont pas encore été prises dans les spécifications doivent être prises pour obtenir le prototype. Il convient de s'assurer de leur impact [HEITMEYER 82].

1.7. Méthodes de prototypage.

Bien que de taille réduite, les projets prototypes souffrent des mêmes problèmes que les projets eux-mêmes [MAC EWEN 82]. C'est pourquoi différentes méthodes et outils sont utilisés pour supporter la construction d'un prototype.

Les pratiques les plus couramment citées sont les suivantes :

- utilisation de langages de programmation de haut niveau à pouvoir d'expression élevé ou de langages spécialisés;

- interprétation ou exécution symbolique de spécifications formelles;

  Si le langage de spécification a une sémantique opérationnelle bien définie, il peut être interprêté [STAVELY 82] ou exécuté symboliquement. La spécification elle-même est le prototype, ce qui présente le risque de confondre le problème à résoudre avec une solution de ce problème. L'exécution symbolique a l'avantage de mettre en évidence différents comportements permis par une même spécification; sa mise en œuvre a l'inconvénient d'être complexe [COHEN 82].

- dérivation automatique ou semi-automatique du prototype à partir des spécifications formelles (approche transformationnelle).

Une implémentation correcte et rapide est obtenue par application à la spécification d'une suite de transformations préservant la sémantique. Chaque construction du langage de spécification est mise en correspondance avec un équivalent exécutable [FEATHER 82] [HABRA 88].

Notre travail s'inscrit dans le cadre de cette dernière approche.

Chapitre 2

Présentation d'un langage de spécification : RSL

2.1. Introduction.

L'approche transformationnelle du prototypage consiste à produire une maquette exécutable par application de transformations préservant la sémantique à une spécification écrite dans un langage formel. Ce chapitre a pour objectif de présenter le langage de spécification particulier qui a servi de base aux transformations qui seront décrites ultérieurement.

Une spécification a pour rôle de définir un problème indépendamment des éléments de solution qu'on pourra y apporter. L'élaboration d'une spécification correcte est une tâche complexe, et l'on convient d'y distinguer trois niveaux :

(i) le niveau de la spécification produite auquel sont définis les objets, relations, procédures, contraintes du système,

(ii) le niveau des processus décrivant les opérations du spécifieur lorsqu'il construit de manière incrémentale des fragments de spécification et

(iii) le niveau des stratégies où sont exposées les raisons sous-jacentes au choix et à l'application des opérations du spécifieur.

RSL (Requirements Structuring Language) dont le noyau de base est décrit dans [DUBOIS 85] est un langage de spécification supportant divers processus et assorti d'une méthodologie de spécification.

Nous commencerons par donner une description générale du langage (section 2.2.) puis nous présenterons quelques processus accompagnant l'élaboration d'une spécification RSL (section 2.3.) et quelques éléments de méthode de construction (section 2.4.). Nous présenterons ensuite un exemple de spécification où nous attirerons l'attention sur certains aspects méthodologiques (section 2.5.). Enfin, nous présenterons une définition de syntaxe concrète de RSL (section 2.6.).

2.2. Description du langage. Le niveau des spécifications produites.

La description qui suit s'inspire de [DUBOIS 84], [DUBOIS 85], [LANGELEZ 86], [DUBOIS 87] et [HABRA 88].

RSL est un langage où une double description est donnée :

- une description formelle fondée sur le calcul des prédicats du premier ordre typés et s'inspirant des types abstraits et
- une description en langue naturelle, agissant comme un commentaire destiné à faciliter la compréhension de la partie formelle.

Ces descriptions sont détaillées au point 2.2.1.

Une spécification RSL est structurée par explicitations successives de types et d'opérations. Elle repose sur une bibliothèque de spécification - expliquée au point 2.2.2. - et deux modèles complémentaires : le premier ou modèle des types d'objets décrit l'univers du problème et est exposé en 2.2.3., le second ou modèle des opérations décrit l'énoncé du problème et fait l'objet du point 2.2.4.

2.2.0. Notations.

Les types et les opérations sont notés en majuscules, les occurrences de types en minuscules.

Le symbole $\rightarrow$ représente l'opérateur d'explicitation.

2.2.1. Description des blocs de spécification.

Une spécification RSL se compose d'un ensemble de blocs où chaque bloc explicite un type d'objet ou une opération.

Un bloc se présente sous la forme de trois colonnes.

La première colonne intitulée "**LEXICON**" contient une description informelle explicative dont le rôle est de faciliter la communication de la spécification. On y trouve:

pour les types d'objets :
- une explication des opérations associées au type,
- une description des types d'objets figurant dans la description formelle et
- éventuellement un énoncé en langue naturelle de l'invariant.

pour les opérations :
- un énoncé de l'objectif de l'opération,
- une description de chaque objet utilisé dans la description formelle et
- éventuellement une explication de ce que représentent les préconditions.

La deuxième colonne intitulée "**OUTLINE**" contient la première partie de la spécification formelle c'est-à-dire :

i) une définition des profils des opérations et des préconditions.

Les opérations sont représentées par des relations mathématiques dont la définition en termes de domaine et de co-domaine constitue le profil. Une précondition restreint le domaine d'une opération et doit être vérifiée pour que l'opération soit définie. De manière générale, les préconditions peuvent être considérées comme des opérations ordinaires dont la seule particularité est de présenter un co-domaine booléen.

ii) une déclaration des types associés aux objets utilisés dans la troisième colonne.

La troisième colonne intitulée "**FORMAL DESCRIPTION**" contient le reste de la spécification formelle c'est-à-dire :

pour un type d'objets :
- la liste des opérations associées à ce type,
- une assertion explicitant la structure du type en fonction de types explicités ailleurs ou figurant dans la bibliothèque et
- les propriétés invariantes vérifiées par le type;

pour une opération OP :
- l'objet résultat r,
- les objets arguments a_i , i = 1, ..., n,
- une assertion explicitant la relation r = OP (a_1 , ..., a_n) représentant l'opération, en fonction d'autres opérations explicitées ailleurs ou figurant dans la bibliothèque,
- une assertion définissant si nécessaire la précondition de cette opération.

Ce qui précède peut être visualisé sur l'exemple présenté dans la section 2.5.

2.2.2. La bibliothèque de spécification.

Une spécification est construite par explicitations successives de types et d'opérations. La connaissance a priori de certains types et opérations assure que le processus itératif d'explicitation se termine.

La bibliothèque de spécification contient d'une part des types pré-définis algébriquement et les opérations qui y sont associées, et d'autre part des relations structurantes pré-définies associées à certains processus de construction de la spécification.

La bibliothèque comprend la spécification des types d'objets suivants:

- des types de base et leurs opérations associées,

exemples

INT : le type entier et les opérations arithmétiques sur les entiers,

CHARST : le type chaîne de caractères et des opérations telles la concaténation de deux chaînes de caractères; d'autres types de base sont

NAT : le type entier naturel,

CHAR : le type caractère,

BOOL : le type booléen,

DATE : le type date, sur lesquels on peut définir également des opérations de base;

- des types d'objets paramétrés et leurs opérations associées,

exemples

CP [T_1, ..., T_n] : le produit cartésien des types T_i , i = 1, ..., n,
et les opérations

- de construction d'une occurrence du type "produit cartésien" à partir d'objets x_i appartenant aux types T_i, i = 1, ..., n : **CONSTUPLE** (x_1, ..., x_n) également notée < x_1, ..., x_n >,et
- de sélection d'un élément composant de type T dans une occurrence x de produit cartésien : **SELECT** (T, x) également notée T (: x),

UNION [T_1, ..., T_n] : l'union disjointe des types T_i , i = 1, ..., n,
et les opérations

- de construction d'une occurrence du type "union disjointe" à partir des types composants (injection) : **INJ** (T_i , t_i) où i = 1, ..., n, et
- de sélection d'un élément de type composant T dans une occurrence x d'union disjointe (projection) : **AS** (T, x),

SEQ [T] : le type suite d'éléments de type T
et les opérations
- de construction d'une occurrence vide du type "suite" : **SEMPTY** et d'une occurrence non vide **CONS (e, s)**
  où e est un élément de type T et s une occurrence du type **SEQ [T]**,
- de sélection de l'élément en ième position dans la suite s : **ITH (i, s)** et
- de sélection de tous les éléments d'une suite s qui vérifient une propriété commune P : **FILTER (s , P)**,
- d'obtention de la longueur de la suite s : **LENGTH (s)** ou **# (s)**;
  des opérations booléennes enfin telles **IN (e, s)** dont le résultat est vrai si l'élément e figure dans la suite s, faux sinon.

SET [T] : le type ensemble dont les opérations correspondent à celles du type SEQ [T] à part celles portant sur l'ordre, et

TABLE [T → T'] : le type table dont chaque objet est vu comme une fonction mathématique mettant en relation deux types; des opérations de construction, et de sélection peuvent également être définies pour ce dernier type.

La bibliothèque comprend la spécification des relations structurantes suivantes:

- des relations d'explicitation de types, utilisées dans le modèle des types d'objets pour exprimer un type en fonction d'autres; à certaines de ces relations sont associés des connecteurs;
  exemples
  -un type peut être explicité par un type paramétré décrit ci-dessus,
  -l'utilisation du connecteur **is-a** définit une relation de sous-typage;

- des relations d'explicitation d'opérations, utilisées dans le modèle des opérations pour expliciter la définition d'une opération en fonction d'opérations plus fines; à ces relations correspondent différents connecteurs;

exemples

-le connecteur **and** permet d'expliciter une opération par une conjonction d'opérations plus élémentaires;

-le connecteur **or** permet d'expliciter une opération par une disjonction d'opérations plus élémentaires;

-le connecteur **with** introduit l'explicitation d'un résultat intermédiaire;

-le connecteur **forall** introduit l'explicitation d'une opération par une opération plus élémentaire qui s'applique à chacun des éléments d'une suite.

2.2.3. Le modèle des types d'objets.

L'univers du problème décrit par le modèle des types d'objets est composé d'un ensemble de blocs de spécification où chaque bloc explicite un type d'objet en fonction d'autres types d'objets plus élémentaires. La structure générale de l'univers du problème est donc une forêt où chaque arbre et sous-arbre est associé à une classe d'objets. Chaque nœud fait référence à un type d'objet qui peut figurer dans le domaine ou co-domaine d'une opération de l'énoncé du problème. Chaque arête correspond à une relation d'explicitation ou autrement dit, à l'application d'un opérateur de construction de spécification (voir section 2.3.). Une feuille correspond à un type pré-défini dans la bibliothèque ou défini ailleurs (réutilisation). L'univers du problème peut se représenter sous la forme graphique d'un ou plusieurs arbres (voir section 2.5. pour un exemple).

La structure d'un bloc objet a été décrite en 2.2.1. Rappelons ce qui y est essentiel :

1) l'assertion d'explicitation de type,
2) l'invariant, s'il en existe un,
3) la liste des opérations associées au type;
l'explicitation de ces opérations figure dans l'énoncé du problème.

2.2.4. Le modèle des opérations.

L'énoncé du problème décrit par le modèle des opérations contient la définition des opérations qui assurent les fonctions du système spécifié. Il est composé d'un ensemble de blocs de spécification d'opération; chaque opération, décrite dans un bloc, est explicitée en fonction d'autres opérations. La structure générale de l'énoncé du problème est une forêt où chaque arbre est associé à une fonction que le système doit réaliser. Chaque nœud fait référence à la spécification d'une opération. Chaque arête correspond à la relation existant entre l'opération explicitée et les opérations constituant l'explicitation. Une feuille correspond à une opération de la bibliothèque de spécification ou à une opération déjà explicitée ailleurs dans la spécification. L'énoncé du problème peut se représenter sous la forme graphique d'un ou plusieurs arbres (voir section 2.5. pour un exemple).

La structure d'un bloc de description d'opération figure en 2.2.1. On retiendra surtout qu'il y figure :

1) une assertion explicitant l'opération en fonction d'autres,
2) une assertion explicitant la précondition éventuelle de l'opération,
3) une définition du profil de l'opération;

les types qui y figurent sont décrits dans l'univers du problème.

Le modèle des types d'objets et celui des opérations sont complémentaires. L'union de l'univers et de l'énoncé du problème donne une spécification dont la redondance peut être utilisée à des fins de vérification de consistance et de complétude.

2.3. Le niveau des "processus".

La description qui suit s'inspire de [DUBOIS 84] [DUBOIS 87] [HABRA 88].

Une construction systématique tend à assurer que la spécification ait certaines qualités (consistance, complétude, vérifiabilité, ...). RSL permet une élaboration par niveaux d'abstraction. A chaque étape, on peut parler en termes des types et opérations pertinents à cette étape, sans devoir se référer aux types et opérations composant les premiers. En particulier, il est possible de reporter la spécification de détails et de choix d'explicitation jusqu'à la fin de l'élaboration de la spécification.

Parallèlement à ce mécanisme d'abstraction existent différents principes de structuration. Si on appelle opérateur une opération appliquée à la spécification pour obtenir une nouvelle spécification plus élaborée, on peut dire qu'une spécification RSL est construite et structurée par l'application répétée d'opérateurs. Les opérateurs correspondent à divers mécanismes. RSL supporte les mécanismes suivants :

- <u>décomposition / agrégation structurelle</u>

Un composant d'un certain niveau d'abstraction peut être vu comme un ensemble de sous-composants (relation "est partie de").

exemple

$T \leftrightarrow CP [T_1, ..., T_n]$: le type T est un produit cartésien des types T_1, ..., T_n.

La décomposition correspond à définir le type T comme étant composé des types T_i, i = 1, ..., n encore à définir ($\rightarrow$); l'agrégation introduit le type T comme la composition des types T_i, i = 1, ..., n déjà définis ($\leftarrow$).

- <u>décomposition / agrégation fonctionnelle</u>

Une opération peut être vue comme une composition d'opérations plus élémentaires à définir ultérieurement (décomposition). Inversément, plusieurs opérations plus élémentaires peuvent être agrégées en une seule opération (agrégation).

exemple

$r = OP (a) \rightarrow r_1 = OP_1 (a_1)$ and ... and $r_n = OP_n (a_n)$

L'opération OP peut se décomposer en la conjonction des opérations OP_i , i = 1, ..., n.

L'introduction de résultats intermédiaires est également permise.

exemple

r = OP (a) → r = OP' (inter) with inter = OP" (a)

L'opération OP est la composition de l'opération OP' après l'opération OP" (notée OP' o OP").

- spécialisation / généralisation

Un composant général peut contenir les propriétés qu'ont en commun ses sous-composants tout en ignorant leurs particularités propres (généralisation). Un composant spécifique peut être défini par les propriétés des composants généraux dont il est issu auxquelles s'ajoutent ses propriétés propres (héritage par spécialisation).

exemple

T' IS-A T

Le type T' est un sous-type du type T et hérite de T ses propriétés et opérations associées.

L'héritage peut être partiellement inhibé par la clause "without".

exemple

T' IS-A T without X

Le type T' hérite de T ses propriétés et opérations associées sauf l'opération X.

- classification

On peut associer un type à chaque objet.

exemple

objet : TYPE_OBJET

L'objet "objet" est de type "TYPE_OBJET". Cette relation est constamment utilisée dans la section "TYPES:" de la colonne "OUTLINE" de chaque bloc de spécification (voir 2.2.1.).

fig. 2.1.a. SPECIFICATION DE L'OPERATION : EMPRUNTS_CRITERE

LEXICON	OUTLINE	FORMAL DESCRIPTION
OBJECTIVE : Obtention d'une liste des emprunts répondant à un critère général. OBJECTS : emprunts_critère : suite des emprunts répondant à un critère emprunts : suite de tous les emprunts critère : un paramètre formel	TYPES : emprunts_critère : EMPRUNTS, emprunts : EMPRUNTS, critère : PARAMETRE OPERATIONS : EMPRUNTS_CRITERE : EMPRUNTS X PARAMETRE → EMPRUNTS FILTER : in toolbox PRED_CRIT : in toolbox	RESULT : emprunts_critère ARGUMENTS : emprunts, critère EXPLICITATION OF INPUT- OUTPUT ASSERTION : emprunts_critère = EMPRUNTS_CRITERE (emprunts, critère) →emprunts_critère = FILTER (emprunts, PRED_CRIT (emprunt, critère))

fig. 2.1.b. SPECIFICATION DE L'OPERATION : EMPRUNTS_EXEMPLAIRE

LEXICON	OUTLINE	FORMAL DESCRIPTION
OBJECTIVE : Obtention d'une liste des emprunts ayant porté ou portant sur un exemplaire donné. OBJECTS : emprunts_exemplaire : suite des emprunts ayant porté ou portant sur un exemplaire donné. emprunts : suite de tous les emprunts exemplaire : un exemplaire donné de livre emprunts_critère : suite des emprunts répondant à un critère critère : un paramètre formel emprunt : un emprunt de la suite des emprunts OPERATION : EMPRUNTS-CRITERE : obtention d'une liste des emprunts répondant à un critère général	TYPES : emprunts_ exemplaire : EMPRUNTS emprunts : EMPRUNTS exemplaire : EXEMPLAIRE emprunts_critère : EMPRUNTS critère : PARAMETRE emprunt : EMPRUNT OPERATIONS : EMPRUNTS_EXEMPLAIRE : EMPRUNT X EXEMPLAIRE →EMPRUNTS EMPRUNT_CRITERE : EMPRUNTS X PARAMETRE → EMPRUNTS PRED_CRIT : in toolbox (:) : in toolbox	RESULT : emprunts_exemplaire ARGUMENTS : emprunts, exemplaire EXPLICITATION OF INPUT-OUTPUT ASSERTION : emprunts_exemplaire = EMPRUNTS_EXEMPLAIRE (emprunts, exemplaire) →emprunts_critère = EMPRUNTS_CRITERE (emprunts, critère) where critère / exemplaire and PRED_CRIT (emprunt, critère) / EXEMPLAIRE_EMPRUNTE (: emprunt) = exemplaire

- paramétrage / instanciation

Une opération générique dans la définition de laquelle figurent des paramètres formels décrit une classe d'opérations. Une opération de cette classe, dite opération instanciée, s'obtient en fixant une valeur à chacun des paramètres formels.

exemple

La figure 2.1. présente un exemple, extrait de la spécification d'un système de bibliothèque, spécification dont un autre fragment est présenté dans la section 2.5. L'opération générique "EMPRUNTS_CRITERE" définit l'obtention d'une liste des emprunts ayant une propriété générale "PRED_CRIT" donnée. L'opération instanciée "EMPRUNTS_EXEMPLAIRE" définit l'obtention d'une liste des emprunts portant ou ayant porté sur un exemplaire donné. La construction "where ... / ... " dénote une substitution formelle de texte.

- réutilisation

La notion de réutilisation relève de l'économie de pensée : il est intéressant de ne pas spécifier plusieurs fois le même (sous-) problème. Lorsqu'un problème déjà spécifié se présente, il est possible d'en réutiliser la spécification.

exemples

Les types et opérations de la bibliothèque ne doivent plus être définis.

La relation d'héritage permet de ne pas redéfinir types et opérations hérités.

Une opération et un type ne doivent être spécifiés qu'une fois indépendamment du nombre de fois où ils apparaisent dans la spécification.

- extension / restriction / modification

Le mécanisme d'extension permet la construction d'une nouvelle spécification en réutilisant la spécification existante à laquelle on ajoute de nouveaux types et / ou opérations. Le mécanisme de restriction permet la construction d'une nouvelle spécification en réutilisant la spécification existante dans laquelle on supprime des types et / ou des opérations. On peut définir une modification comme la composition d'une restriction et d'une extension.

exemple

La structure d'un type peut être modifiée; par exemple, un type défini comme un produit cartésien binaire peut être redéfini comme un produit cartésien ternaire. La spécification contenant le type explicité comme produit cartésien binaire est restreinte à la spécification où cette explicitation n'est pas donnée. On étend alors cette spécification à une nouvelle où le type est cette fois explicité comme un produit cartésien ternaire. Enfin, les modifications doivent être répercutées à travers l'ensemble de la spécification.

2.4. Le niveau des stratégies.

Une méthode a pour but de déterminer la composition d'opérateurs la plus appropriée à appliquer pour obtenir une spécification "satisfaisante". Conserver la séquence des opérateurs appliqués permet de garder une trace formelle de l'activité de développement de la spécification, trace qui peut s'utiliser à des fins de vérification, de documentation, et de réutilisation partielle lors de la maintenance de la spécification. Dans un état particulier de la spécification, l'opérateur à appliquer pour atteindre un nouvel état de spécification dépend de la stratégie de contrôle choisie.

On distingue les stratégies locales qui déterminent la direction dans laquelle l'état suivant de spécification sera atteint des stratégies globales qui sont des combinaisons des précédentes et qui portent sur plusieurs états consécutifs.

2.4.1. Stratégies locales.

Stratégie descendante et stratégie ascendante.

Une stratégie descendante est associée aux mécanismes d'abstraction, de décomposition, de spécialisation. Dans la nouvelle spécification, on introduit des types d'objets et des opérations plus fins. L'alternative à une approche descendante est quant à elle le choix d'une stratégie ascendante dans laquelle on construit types et opérations plus généraux à partir de blocs plus élémentaires.

Stratégie progressive et stratégie régressive.

Dans une stratégie régressive, on examinera le type de l'objet résultat pour déterminer la façon d'atteindre un nouvel état de spécification. Par opposition , une stratégie progressive est guidée par la structure des arguments.

L'application d'une de ces deux stratégies à chaque niveau de l'analyse assure un parallélisme entre la structure des types d'objets et celle des explicitations d'opérations. Cette hypothèse sera exploitée au chapitre 4 (traitement des équations implicites).

Stratégie de réutilisation des connaissances.

On cherche à rapprocher le problème courant d'un problème déjà spécifié. Différents degrés de réutilisabilité sont à envisager : on peut imaginer une stratégie de réutilisation pure - recherche d'une équivalence sémantique entre le problème courant et un problème déjà spécifié - ou une statégie adaptative où des composants à buts généraux pourraient être taillés aux besoins des endroits où les insérer.

2.4.2. Stratégies globales.

Stratégie orientée-objet et stratégie orientée-opération.

Dans une stratégie orientée-objet, on sélectionne un type, puis une opération associée à ce type et on spécifie l'opération. Dans une stratégie orientée-opération, on choisit d'abord une opération, puis un type d'objet associé; on spécifie l'objet en termes d'autres objets puis de manière correspondante, on spécifie l'opération en termes d'autres opérations.

Stratégie en profondeur d'abord et stratégie en largeur d'abord.

Dans une stratégie en profondeur d'abord, on spécifie d'abord les derniers types et opérations introduits. Dans une stratégie en largeur d'abord, on spécifie tous les objets et opérations d'un niveau d'abstraction avant de passer à la spécification du niveau adjacent.

D'autres politiques de développement des fragments de spécification peuvent être appliquées : développement des fragments les plus critiques d'abord, développement des fragments les plus spécifiques d'abord, etc. Les stratégies globales dépendent fortement du type de problème à résoudre et de la façon dont le spécifieur voit le problème.

2.5. Exemple.

Pour nous familiariser avec RSL, nous nous sommes essayés à la spécification d'un système automatisé de gestion de bibliothèque. L'énoncé informel figure dans [DAVIS 87] et cet exemple est également traité en RSL dans [DUBOIS 87] et [HABRA 88].

Nous commencerons par exposer la méthode que nous avons suivie (2.5.1), ensuite nous présenterons pour la commenter une partie de la spécification élaborée (2.5.2.).

2.5.1. Méthode.

Nous nous sommes donnés a priori une structure arborescente de types généraux correspondant aux objets qui nous semblaient devoir apparaître dans le système à spécifier. Puis, nous avons essayé de spécifier l'une après l'autre les fonctions de l'énoncé. Lorsque nous constations qu'il nous était impossible de spécifier une opération avec la structure de types dont nous disposions alors, nous modifiions celle-ci et répercutions les modifications sur toutes les opérations déjà spécifiées. Le processus d'extension / modification de la spécification est peut-être celui qui a été invoqué le plus souvent. Finalement, l'arbre des types correspondant à notre spécification finale (voir fig. 2.2.) correspond très peu à notre idée initiale trop vague.

Cette approche est un mélange de stratégie orientée-objet et de stratégie orientée-opération. La première stratégie a été suivie lorsque la structure de types convenait, la seconde lorsqu'il était nécessaire de la modifier.

Nous avons également pratiqué une approche en profondeur d'abord, explicitant complètement les opérations nécessaires à une fonction avant de passer à la fonction suivante. Le problème posé étant relativement simple, chaque fonction ne nécessite l'introduction que de peu d'opérations non élémentaires. Néanmoins, le traitement de chaque nouvelle fonctionnalité peut remettre en question la structure de types avec les répercussions que cela entraîne dans ce qui a déjà été spécifié. Pour un problème plus important, il est vraisemblablement préférable de progresser parallèlement dans le raffinement de la spécification de chaque fonction.

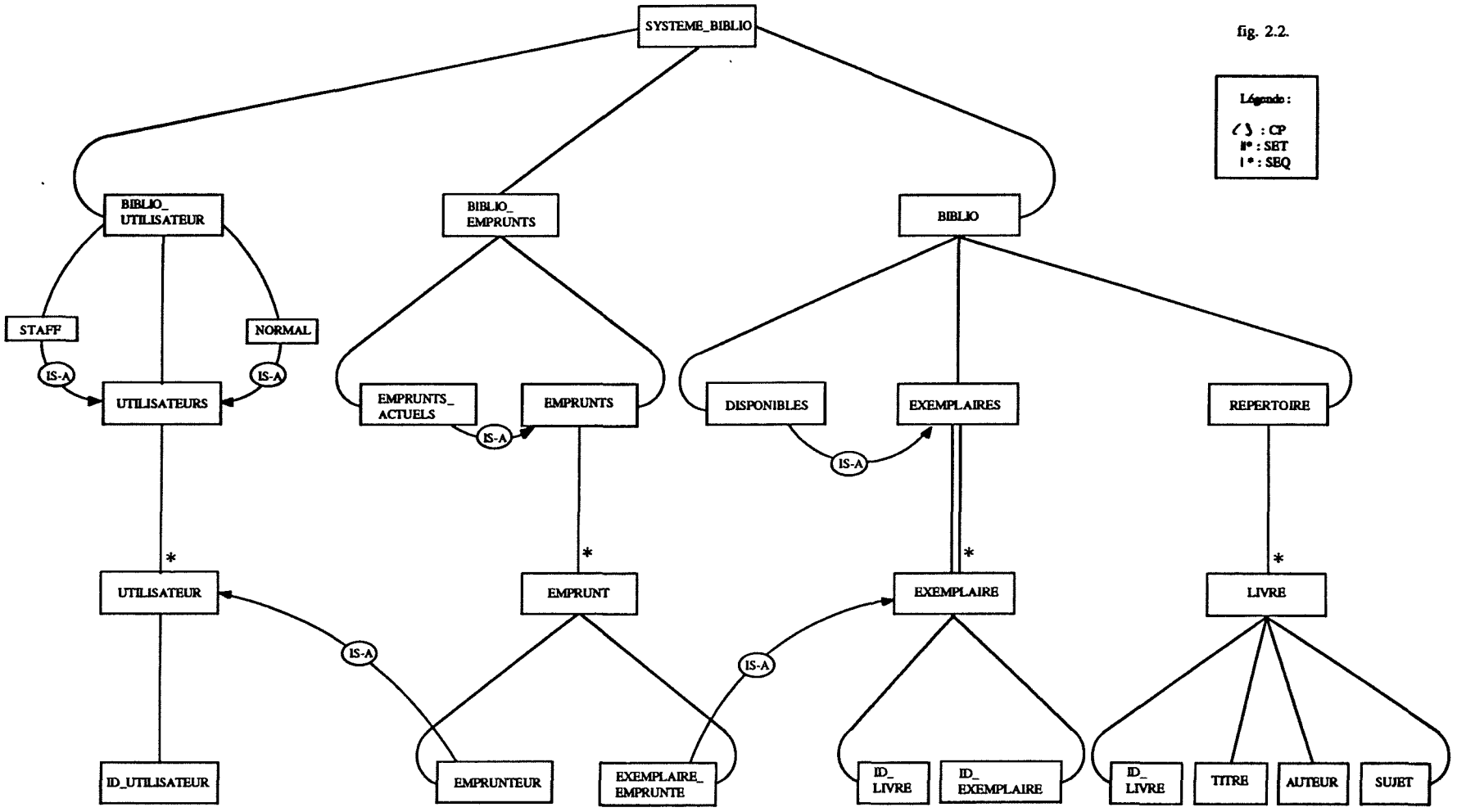

SYSTEME_BIBLIO
BIBLIO_ UTILISATEUR
BIBLIO_ EMPRUNTS
BIBLIO
STAFF
NORMAL
IS-A
IS-A
UTILISATEURS
EMPRUNTS_ ACTUELS
IS-A
EMPRUNTS
DISPONIBLES
IS-A
EXEMPLAIRES
REPERTOIRE
*
*
*
*
UTILISATEUR
EMPRUNT
EXEMPLAIRE
LIVRE
IS-A
IS-A
ID_UTILISATEUR
EMPRUNTEUR
EXEMPLAIRE_ EMPRUNTE
ID_ LIVRE
ID_ EXEMPLAIRE
ID_ LIVRE
TITRE
AUTEUR
SUJET
Légende :
( ) : CP
II* : SET
I * : SEQ

fig. 2.2.

fig. 2.3. SPECIFICATION OF OBJECT TYPE : BIBLIO

LEXICON	OUTLINE	FORMAL DESCRIPTION
OPERATIONS : TOUS_EXEMPLAIRES_DISPONIBLES : prédicat vrai si tous les exemplaires d'un livre donné sont disponibles, faux sinon AJOUT_DEF : définition de l'ajout d'un exemplaire donné et éventuellement du livre correspondant s'il ne fait pas partie du répertoire. SUPPRESSION_EXEMPLAIRES_DEF : définition de la suppression de tous les exemplaires d'un livre. SUPPRESSION_EXEMPLAIRE_DEF : définition de la suppression d'un exemplaire donné et éventuellement du livre correspondant. MAJ_BIBLIO : pour un emprunt donné, soustraction de l'exemplaire emprunté de l'ensemble des exemplaires disponibles. TYPES : SYSTEME_BIBLIO : le système automatisé de bibliothèque BIBLIO : la bibliothèque du système DISPONIBLES : les exemplaires disponibles de la bibliothèque EXEMPLAIRES : les exemplaires de la bibliothèque REPERTOIRE : le répertoire des livres de la bibliothèque EXEMPLAIRE : un exemplaire LIVRE : un livre EMPRUNT : un emprunt INVARIANTS : Un exemplaire ne correspond qu'à un et un seul livre. A tout livre correspond au moins un exemplaire. Tout exemplaire disponible est un exemplaire.	OPERATIONS : TOUS_EXEMPLAIRES_DISPONIBLES : LIVRE X BIBLIO $\rightarrow$BOOL AJOUT_DEF : EXEMPLAIRE X BIBLIO X LIVRE $\rightarrow$BIBLIO SUPPRESSION_EXEMPLAIRES_DEF : LIVRE X BIBLIO $\rightarrow$BIBLIO SUPPRESSION_EXEMPLAIRE_DEF : EXEMPLAIRE X BIBLIO $\rightarrow$BIBLIO MAJ_BIBLIO : EMPRUNT X SYSTEME_BIBLIO $\rightarrow$BIBLIO TYPES : disponibles : DISPONIBLES exemplaires : EXEMPLAIRES répertoire : REPERTOIRE exemplaire : EXEMPLAIRE livre : LIVRE	ASSOCIATED OPERATIONS : TOUS_EXEMPLAIRES_DISPONIBLES SUPPRESSION_EXEMPLAIRES_DEF SUPPRESSION_EXEMPLAIRE_ DEF AJOUT_DEF MAJ_BIBLIO EXPLICITATION OF TYPE STRUCTURE : BIBLIO$\rightarrow$ CP [DISPONIBLES, EXEMPLAIRES, REPERTOIRE] EXPLICITATION OF INVARIANTS : $\forall$ CONSTUPLE (disponibles, exemplaires, répertoire) : BIBLIO ($\forall$exemplaire) : [IN (exemplaire, exemplaires) $\Rightarrow$ ($\exists$! livre) (IN (livre, répertoire) $\wedge$ (ID_LIVRE (:exemplaire) = ID_LIVRE (: livre)))] $\wedge$ ($\forall$livre) : [IN (livre, répertoire) $\Rightarrow$ ($\exists$ exemplaire) (IN (exemplaire, exemplaires) $\wedge$ ID_LIVRE (: exemplaire) = ID_LIVRE (: livre))] $\wedge$ ($\forall$ exemplaire) : [IN (exemplaire, disponibles) $\Rightarrow$ IN (exemplaire, exemplaires)]

fig. 2.4. SPECIFICATION DE L'OPERATION : EMPRUNT_DEF

LEXICON	OUTLINE	FORMAL DESCRIPTION
OBJECTIVE : Définition de l'emprunt d'un exemplaire disponible OBJECTS : système_biblio_après:le système après définition de l'opération système_biblio_avant:le système avant définition de l'opération emprunt : l'emprunt à traiter biblio_utilisateur_après : suite des utilisateurs après définition de l'opération biblio_emprunts_après : la bibliothèque d'emprunts après définition de l'opération biblio_emprunts_avant : la bibliothèque d'emprunts avant définition de l'opération biblio_après : la bibliothèque après définition de l'opération biblio_avant : la bibliothèque avant définition de l'opération disponibles_avant : ensemble des exemplaires disponibles avant définition de l'opération exemplaire : un exemplaire de livre emprunteur : utilisateur empruntant emprunts_actuels_avant : suite des emprunts auxquel correspond, avant définition de l'opération, un exemplaire actuellement emprunté OPERATIONS : MAJ_BIBLIO_UTILISATEUR : définit l'action à effectuer sur la suite des utilisateurs MAJ_BIBLIO_EMPRUNTS : définit l'action à effectuer sur la bibliothèque d'emprunts MAJ_BIBLIO : définit l'action à effectuer sur la bibliothèque PRECONDITIONS : EMPRUNT_OK : prédicat vrai si les conditions nécessaires à la définition d'un emprunt sont vérifiées, faux sinon PEUT_ENCORE_EMPRUNTER : prédicat vrai si le nombre d'exemplaires actuellement détenus par l'emprunteur est strictement inférieur au maximum permis, faux sinon DETIENT_EXEMPLAIRE : prédicat vrai si pour un emprunt donné l'emprunteur détient déjà un exemplaire correspondant au même livre, faux sinon.	TYPES : système_biblio_après, système_biblio_avant : SYSTEME_BIBLIO, emprunt : EMPRUNT, biblio_utilisateur_après : BIBLIO_UTILISATEUR, biblio_emprunts_après, biblio_emprunts_avant : BIBLIO_EMPRUNTS, biblio_après, biblio_avant : BIBLIO, disponibles_avant : DISPONIBLES, exemplaire : EXEMPLAIRE_EMPRUNTE, emprunteur : EMPRUNTEUR, emprunts_actuels_avant : EMPRUNTS_ACTUELS OPERATIONS : EMPRUNTS_DEF : EMPRUNT X SYSTEME_BIBLIO → SYSTEME_BIBLIO MAJ_BIBLIO_UTILISATEUR : SYSTEME_BIBLIO → BIBLIO_UTILISATEUR MAJ_BIBLIO_EMPRUNTS : EMPRUNT X SYSTEME_BIBLIO → BIBLIO_EMPRUNTS MAJ_BIBLIO : EMPRUNT X SYSTEME_BIBLIO → BIBLIO EMPRUNT_OK : EMPRUNT X SYSTEME_BIBLIO → BOOL PEUT_ENCORE_EMPRUNTER : EMPRUNTEUR X EMPRUNTS_ACTUELS → BOOL DETIENT_EXEMPLAIRE : EMPRUNT X EMPRUNTS_ACTUELS → BOOL IN : in toolbox (:) : in toolbox	RESULT : système_biblio_après ARGUMENTS : emprunt, système_biblio_avant EXPLICITATION OF INPUT-OUTPUT ASSERTION : système_biblio_après = EMPRUNT_DEF (emprunt, système_biblio_avant) → biblio_utilisateur_après = MAJ_BIBLIO_UTILISATEUR (système_biblio_avant) and biblio_emprunts_après = MAJ_BIBLIO_EMPRUNTS (emprunt, système_biblio_avant) and biblio_après = MAJ_BIBLIO (emprunt, système_biblio_avant) EXPLICITATION OF PRECONDITION : EMPRUNT_OK (emprunt, système_biblio_avant) → IN (disponibles_avant, exemplaire) with disponibles_avant = DISPONIBLES (: biblio_avant) with biblio_avant = BIBLIO (: système_biblio_avant) and with exemplaire = EXEMPLAIRE_EMPRUNTE (: emprunt) and PEUT_ENCORE_EMPRUNTER (emprunteur, emprunts_actuels_avant) with emprunts_actuels_avant = EMPRUNTS_ACTUELS (: biblio_emprunts_avant) with biblio_emprunts_avant = BIBLIO_EMPRUNTS (: système_biblio_avant) and with emprunteur = EMPRUNTEUR (: emprunt) and not (DETIENT_EXEMPLAIRE (emprunt, emprunts_actuels_avant)) with emprunts_actuels_avant = EMPRUNTS_ACTUELS (: biblio_emprunts_avant) with biblio_emprunts_avant = BIBLIO_EMPRUNTS (: système_biblio_avant)

fig. 2.5. SPECIFICATION OF OPERATION : MAJ_BIBLIO

LEXICON	OUTLINE	FORMAL DESCRIPTION
OBJECTIVE : Soustraction de l'ensemble des exemplaires disponibles, d'un exemplaire correspondant à un emprunt donné OBJECTS : biblio_après : la bibliothèque (livres, exemplaires, exemplaires disponibles) après l'opération biblio_avant : la bibliothèque (livres, exemplaires, exemplaires disponibles) avant l'opération emprunt : un emprunt système_biblio : le système automatisé de gestion de bibliothèque disponibles_après : les exemplaires disponibles après l'opération disponibles_avant : les exemplaires disponibles avant l'opération exemplaires_après : les exemplaires de la bibliothèque répertoire_après : les livres de la bibliothèque OPERATIONS : PAS_EXEMPLAIRE_DONNE : prédicat vrai si un exemplaire disponible donné est différent d'un exemplaire donné, faux sinon PRECONDITIONS : /	TYPES : biblio_après, biblio_avant : BIBLIO, emprunt : EMPRUNT, système_biblio : SYSTEME_BIBLIO, disponibles_après, disponibles_avant : DISPONIBLES, disponible_avant : DISPONIBLE, exemplaires_après : EXEMPLAIRES, exemplaire : EXEMPLAIRE_EMPRUNTE répertoire_après : REPERTOIRE OPERATIONS : MAJ_BIBLIO : EMPRUNT X SYSTEME_BIBLIO →BIBLIO PAS_EXEMPLAIRE_DONNE : DISPONIBLE X EXEMPLAIRE → BOOL CONSTUPLE : in toolbox REMOVE : in toolbox (:) : in toolbox	RESULT : biblio_après ARGUMENTS : emprunt, système_biblio EXPLICITATION OF INPUT- OUTPUT ASSERTION : biblio_après = MAJ_BIBLIO (emprunt, système_biblio) →biblio_après = CONSTUPLE (disponibles_après, exemplaires_après, répertoire_après) with disponibles_après = FILTER (disponibles_avant, PAS_EXEMPLAIRE_DONNE (disponible_avant, exemplaire)) with disponibles_avant = DISPONIBLES (: biblio_avant) with biblio_avant = BIBLIO (: système_biblio) and with exemplaire = EXEMPLAIRE_EMPRUNTE (: emprunt) and with exemplaires_après = EXEMPLAIRES (: biblio_avant) with biblio_avant = BIBLIO (: système_biblio) and with répertoire_après = REPERTOIRE (: biblio_avant) with biblio_avant = BIBLIO (: système_biblio)

Enfin, notre approche peut être qualifiée de descendante et de régressive. Elle est descendante car nous sommes partis de types et opérations généraux pour produire des types et opérations décomposés. Elle est régressive car c'est à partir de la structure du résultat à produire que nous avons essayé de décomposer l'opération à spécifier. Par exemple, "système_biblio_après", le résultat de l'opération "EMPRUNT_DEF" (voir fig. 2.4.) , est une occurrence d'un produit cartésien des types "BIBLIO_UTILISATEUR", "BIBLIO_EMPRUNTS" et "BIBLIO" (voir fig. 2.2.). L'opération "EMPRUNT_DEF" est décomposée en trois opérations:"MAJ_BIBLIO_UTILISATEUR", "MAJ_BIBLIO_EMPRUNTS" et "MAJ_BIBLIO" portant chacune respectivement sur un des types du produit cartésien.

Cette approche descendante et régressive convient particulièrement au type de problème posé. Une approche descendante favorise l'obtention d'une spécification complète mais a l'inconvénient de forcer le spécifieur à structurer le système dès les premiers niveaux de décomposition, c'est-à-dire à un moment où il n'en a pas encore une idée précise. Cela peut l'obliger par la suite à apporter de nombreuses et importantes modifications à sa spécification. Dans notre cas, la taille réduite de l'exemple, le nombre peu important de niveaux de décomposition limitait cet inconvénient. L'approche régressive s'est imposée d'elle-même car nous savions les résultats que nous voulions obtenir; nous connaissions par contre moins bien les arguments dont il fallait disposer. Notons encore qu'une stratégie régressive à l'avantage de garantir la production de tous les résultats nécessaires.

Une fois les opérations d'explicitation déterminées, nous mettions à jour la liste des arguments de l'opération spécifiée. Souvent, nous avons préféré garder en argument un objet d'un type plus général (exemple : SYSTEME_BIBLIO) dont la seule partie qui nous intéressait était alors exprimée par une succession de données intermédiaires figurant dans des équations de sélection introduites par le connecteur "with". L'avantage de ce choix est de minimiser le nombre d'arguments des opérations. L'inconvénient est qu'une décomposition n'est pas exprimée seulement en fonction de la composition immédiatement supérieure. Il est parfois nécessaire de connaître plusieurs niveaux de décomposition de types ce qui se traduit dans la spécification par de nombreux "with" en cascade (voir fig. 2.5.). L'assertion d'explicitation est par conséquent plus longue.

2.5.2. Un fragment de spécification.

Nous présentons ici le bloc de spécification correspondant au type "BIBLIO" (fig. 2.3.), le bloc de spécification d'une opération de définition d'un emprunt (fig. 2.4.) et celui correspondant à une opération introduite en explicitant cette dernière : "MAJ_BIBLIO" (fig. 2.5.).

L'enregistrement d'un emprunt est une fonction du système à spécifier. L'arbre des opérations y correspondant est partiellement représenté à la fig. 2.6. où " " représente cette fonction appelée EMPRUNT. Pour être complet, l'arbre devrait contenir la décomposition de la précondition "EMPRUNT_OK". Les opérations "MAJ_BIBLIO_UTILISATEUR", "MAJ_BIBLIO_EMPRUNTS" et "MAJ_BIBLIO" s'expriment en termes d'opérations de la bibliothèque et leur explicitation n'est donc pas davantage représentée graphiquement. L'arbre représente donc uniquement deux pas d'explicitation par une conjonction d'opérations. On a ici une illustration d'une idée déjà évoquée, à savoir que le problème étant relativement simple, le nombre de niveaux dans un arbre des opérations est réduit.

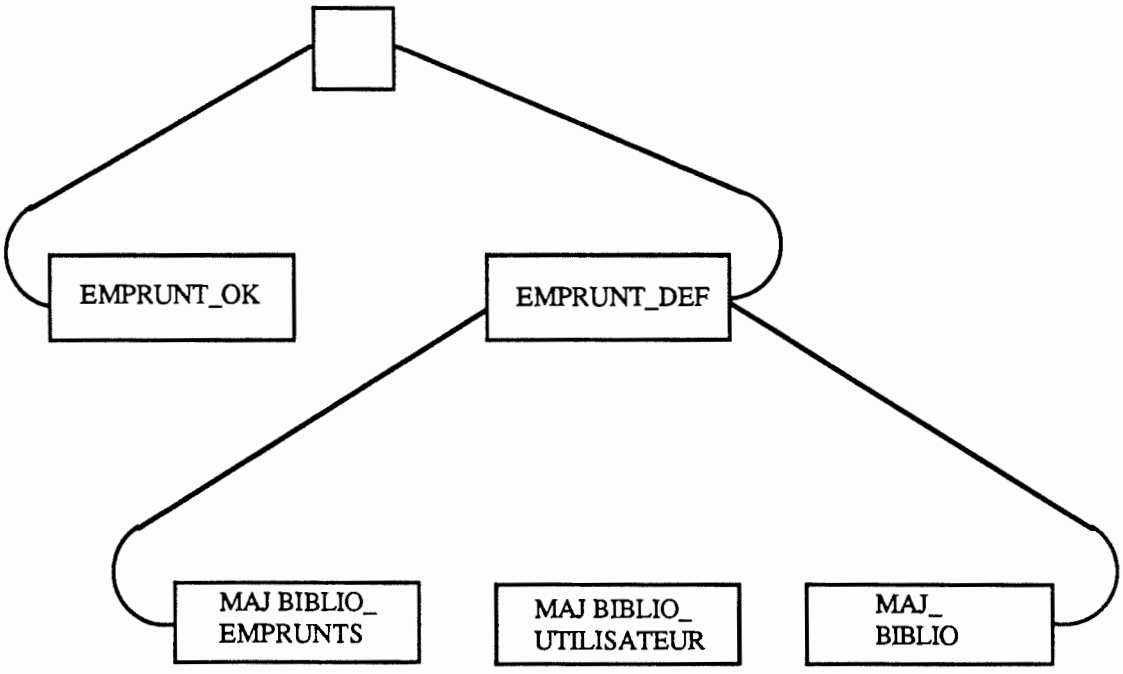


fig. 2.6. arbre correspondant à la fonction "EMPRUNT"

On notera que cette spécification décrit le comportement du système lorque la précondition est vérifiée. Ce que le système doit faire dans le cas contraire n'a pas été développé ici. L'énoncé informel ne précise pas le comportement attendu du système en cas d'erreur [WING 88]. Le spécifieur doit pallier cette lacune dans la spécification formelle. Il peut le faire de diverses manières :

- en renforçant la précondition de sorte que l'erreur ne puisse se produire, ou
- en renforçant la postcondition en spécifiant explicitement le comportement en cas d'erreurs, ou
- en combinant les deux moyens cités ci-dessus.

Des variantes peuvent aussi s'observer dans la spécification du comportement en cas d'erreurs :

- une réaction générale peut être spécifiée, valable dans tous les cas d'erreurs, ou
- le traitement des erreurs peut être adapté à chaque type d'erreur.

Notre philosophie de spécification pourrait se résumer par : "Cela va sans dire, mais cela va encore mieux en le disant". Le résultat est assez verbeux.

Seule la relation entre les résultats des opérations simples utilisées dans l'explicitation et le résultat de l'opération principale spécifiée a été laissée implicite. Par exemple, dans le cas d' "EMPRUNT_DEF", la relation "système_biblio_après = CONSTUPLE (biblio_utilisateur_après, biblio_emprunts_après, biblio_après) est implicite. Tout le reste a été explicité. Notons que la structure de RSL est telle qu'il est possible, sous certaines hypothèses de laisser beaucoup plus de relations implicites. Ce point sera développé au chapitre 4.

Lorsqu'un même argument intermédiaire apparaît plusieurs fois, nous l'avons chaque fois explicité. Par exemple, dans la spécification de l'opération "MAJ_BIBLIO", l'explicitation "with biblio_avant = BIBLIO (: système_biblio)" apparaît trois fois. Rappelons que cela est lié en partie au choix de garder des arguments de types généraux et ajoutons que cette redondance est inutile. Il suffit d'expliciter la donnée intermédiaire lors de sa première apparition; elle est supposée connue dans les déclarations qui suivent. Disons déjà que le principe de fonctionnement de l'outil de prototypage réalisé est tel qu'il faut expliciter les arguments intermédiaires lors de leur introduction; s'ils figurent encore dans la même explicitation, il n'est plus nécessaire alors de répéter leur explicitation.

Dans la spécification de l'opération "MAJ_BIBLIO", on peut remarquer que les parties d'assertion portant sur "exemplaires_après" et "répertoire_après" ne font rien d'autre qu'exprimer que l'opération "MAJ_BIBLIO" laisse les objets de types "EXEMPLAIRES" et "REPERTOIRE" inchangés. Il serait souhaitable de disposer d'une construction pour exprimer qu'une partie des arguments sont inchangés. On peut imaginer une opération de mise-à-jour paramétrée sur les types qu'on insérerait dans la bibliothèque, par exemple UPDATE $_c$ (x, v) où dans l'occurrence x d'un type composé "CP", la valeur du champ c explicitement connu serait remplacée par v. Cette solution est assez restrictive : une telle opération ne peut porter que sur le type "produit cartésien" et on ne peut remplacer plus d'un élément à la fois. Une solution qui ne présente pas ces inconvénients et qui est par ailleurs moins procédurale consiste en l'introduction d'une égalité avec exception pour certains composants. Dans ce dernier cas, la spécification est exprimée de manière très concise. L'assertion de notre exemple deviendrait :

```
biblio_après = MAJ_BIBLIO ( emprunt, système_biblio )
  →
biblio_après = biblio_avant
      with biblio-avant = BIBLIO ( : système_biblio )
  except disponibles_après = FILTER ( disponibles_avant,
      PAS_EXEMPLAIRE_DONNE (disponible_avant,exemplaire) )
            with disponibles_avant  = DISPONIBLES ( : biblio_avant )
      and   with exemplaire = EXEMPLAIRE_EMPRUNTE ( : emprunt ).
```

Si dans les références données au début de la section 2.2., on fait l'inventaire des types et opérations pouvant figurer dans la bibliothèque de spécification, on s'aperçoit qu'un sous-ensemble très restreint suffit à la spécification du système de bibliothèque. Nous avons abondamment utilisé l'opération "FILTER (s, P) sélectionnant dans un objet de type "suite" s tous les éléments vérifiant une propriété P portant sur les éléments de s. Par contre, assez ironiquement, le connecteur "forall" introduisant l'explicitation d'une opération par une autre opération qui s'applique à chacun des éléments d'une suite donnée n'a pas été utilisé une seule fois dans cet exemple, alors qu'il en sera abondamment question dans la suite (chapitres portant sur les transformations). L'explication est sans doute simple: bien qu'il suffise pour que tous les processus décrits en 2.3. y trouvent une illustration, l'exemple est de portée trop réduite pour nécessiter l'exploitation de toutes les ressources de la bibliothèque de spécification.

Cependant, paradoxalement, alors que nous n'utilisions qu'une partie restreinte des opérations de la bibliothèque, nous avons ressenti le besoin d'une opération qui ne s'y trouvait pas (du moins à notre connaissance). Cette opération portant sur le type "SEQ" a pour objectif d'ôter toutes les occurrences d'un élément donné e de type T d'une suite donnée s d'éléments de type T. On peut en donner la définition inductive suivante.

Soit sres de type SEQ [T].

```
sres = REMOVE ( s, e )
  →
si s = SEMPTY
alors  sres = SEMPTY
sinon        si s = CONS ( e, s' )
             alors  sres = REMOVE ( s', e )
             sinon  si s = CONS ( e', s' )
                    alors  sres = CONS ( e' , REMOVE ( s', e ) ).
```

Il va de soi qu'on peut se passer de cette dernière opération. Si e' est de type T et est l'élément représentatif de s, la construction FILTER (s, P (e', e)) où P (e', e) est vrai si e' est différent de e, faux sinon, est équivalente à REMOVE (s, e). Elle est cependant plus longue à exprimer.

On s'aperçoit très vite que gérer une spécification RSL sans outil automatisé est extrêmement fastidieux. Même sur un exemple aussi limité que celui de la bibliothèque (huit fonctions seulement), le nombre de blocs à manipuler est assez important. De plus, dans chaque bloc, il convient de gérer les trois colonnes en conservant la consistance de toute la spécification. C'est pourquoi il est particulièrement intéressant de disposer pour le langage d'un éditeur syntaxique, éventuellement couplé à différents vérificateurs de cohérence. Notons que ces assistances existent déjà et qu'un outil automatisé d'aide à la spécification, paramétré sur les processus et stratégies de construction de spécifications est actuellement en cours de développement [LEVY 87].

2.6. Syntaxe concrète de RSL.

Définition : La syntaxe concrète décrit les règles de formation des constructions permises dans le langage, indépendamment de la sémantique de ces constructions.

Elle est ici présentée sous forme d'un ensemble de règles de grammaire exprimées dans le formalisme BNF (voir le chapitre 3 section 3.2. pour une définition d'une grammaire context-free). Les symboles terminaux sont imprimés en gras. Les symboles non-terminaux apparaissent entre "<" et ">". Le symbole "::=" est le symbole de réécriture. Le symbole "|" sépare les alternatives. (<cons>)* signifie que la construction "cons" apparaît de 0 à N fois.

< specif > :: = < bloc-oper >* < bloc-type-obj >*

< bloc-oper >
:: = **SPECIFICATION OF OPERATION** < nom-maj >
FORMAL DESCRIPTION : < dec-form-op >
OUTLINE : < profil-op >
LEXICON : < lexicon-op >
END < nom-maj > .

< dec-form-op >
:: = **RESULT :** < nom-obj >
ARGUMENTS : < nom-obj > (, < nom-obj >)*
EXPLICITATION OF INPUT-OUTPUT ASSERTION :
< expr-op > → < explic-op >

| **RESULT :** < nom-obj >
ARGUMENTS : < nom-obj > (, < nom-obj >)*
EXPLICITATION OF INPUT-OUTPUT ASSERTION :
< expr-op > → < explic-op >
EXPLICITATION OF PRECONDITION :
< expr-pre-op > → < pre-explic-op >

< expr-op >

:: = < nom-obj > = < nom-maj > (< nom-obj > (, < nom-obj >)*)

< expr-pre-op > :: = < expr-op >

| < nom-maj > (< nom-obj > (, < nom-obj >)*)

< explic-op >

:: = < pre-explic-op >

| (< explic-op >)

| **forall** < nom-obj > :

IN (< nom-obj > , < nom-obj >) : < explic-op >

| < pré-et-explic >

< pre-explic-op >

:: = < expr-pre-op >

| **not** (< pre-explic-op >)

| (< pre-explic-op >)

| < expr-terminale > = < expr-terminale >

| < explic-op > **or** < explic-op >

| < explic-op > **and** < explic-op >

| < pre-explic-op > < expr-interm > (**and** < expr-interm >)*

< expr-terminale > :: = < constante >

| < nom-obj >

| < biblio-oper >

< expr-interm >

:: = **with** < expr-op > (**and** < expr-interm >)*

| **with** < expr-op > < expr-interm > (**and** < expr-interm >)*

| **with** < nom-obj > = < expr-terminale >

(**and** < expr-interm >)*

| **with** < nom-obj > = < biblio-oper > < expr-interm >

(**and** < expr-interm >)*

| **with** (< expr-op > (**and** < expr-interm >)*)

| **with** (< expr-op > < expr-interm >

(**and** < expr-interm >)*)

| **with** (< nom-obj > = < expr-terminale >
(**and** < expr-interm >)*)
| **with** (< nom-obj > = < biblio-oper > < expr-interm >
(**and** < expr-interm >)*)

< pré-et-explic >
:: = < pre-explic-op > **AND** < explic-op >
| < explic-op > **if** < expr-pre-op >
(, < explic-op > **if** < expr-pre-op >)*
| < explic-op > **if** < expr-pre-op >
(, < explic-op > **if** < expr-pre-op >)* ,
< explic-op > **otherwise**

< prédicat > :: = < atome >
| ¬ < prédicat >
| < prédicat > ∧ < prédicat >
| < prédicat > ∨ < prédicat >
| < prédicat > ⇒ < prédicat >
| < prédicat > ⇔ < prédicat >
| (< prédicat >)
| (∀ < nom-obj >) < prédicat >
| (∃ < nom-obj >) < prédicat >

< atome > :: = < nom-maj > (< terme > (, < terme >)*)
| < terme > < oper-relat > < terme >

< terme > :: = < constante >
| < nom-obj >
| (< terme >)
| < nom-maj > (< terme > (, < terme >)*)
| < terme > < oper-arithm > < terme >

< oper-arith > :: = + | - | * | / | **mod** | **div**

< oper-relat > :: = = | < | > | <= | >= | ><

< profil-op > :: = **TYPES** : < prof-obj > (, < prof-obj >)*
OPERATIONS : < prof-op >*

< prof-obj > :: = < nom-obj > (, < nom-obj >)* : < nom-maj >

< prof-op > :: = < nom-maj > (, < nom-maj >)* : < def-op >

< def-op > :: = < nom-maj > (**x** < nom-maj >)* → < nom-maj >
| < biblio-def > (, < biblio-def >)* : **in toolbox**

< lexicon-op > :: = **OBJECTIVE** : < ligne-texte >*
OBJECTS : < obj-lex >*
OPERATIONS : < maj-lex >*
PRECONDITIONS : < maj-lex >*

< ligne-texte > :: = [tout caractère sauf ↵]* ↵

< obj-lex > :: = < nom-obj > : < ligne-texte >*

< maj-lex > :: = < nom-maj > : < ligne-texte >*

< bloc-type-obj >
:: = **SPECIFICATION OF OBJECT TYPE** < nom-maj >
FORMAL DESCRIPTION : < dec-form-obj >
OUTLINE : < profil-obj >
LEXICON : < lexicon-obj >
END < nom-maj > .

< dec-form-obj >

:: = **ASSOCIATED OPERATIONS :**
< nom-maj > (, < nom-maj >)*
EXPLICITATION OF TYPE STRUCTURE :
< nom-maj > → < explic-type >

| **ASSOCIATED OPERATIONS :**
< nom-maj > (, < nom-maj >)*
EXPLICITATION OF TYPE STRUCTURE :
< nom-maj > → < explic-type >
EXPLICITATION OF INVARIANT : < prédicat >

< explic-type >

:: = < nom-maj >
| < type-base >
| **UNION** [< explic-type > (, < explic-type >)*]
| **CP** [< explic-type > (, < explic-type >)*]
| **SEQ** [< explic-type >]
| **SET** [< explic-type >]
| **TABLE** [< explic-type > → < explic-type >]
| **IS-A** [< explic-type >]
| **IS-A** [< explic-type >]
without (< nom-obj > (, < nom-obj >)*

< type-base > :: = **INT | NAT | CHAR | CHARST | BOOL | DATE**

< profil-obj > :: = **OPERATIONS :** < prof-op >*
| **TYPES :** < prof-obj > (, < prof-obj >)*

< lexicon-obj > :: = **OBJECTIVE :** < ligne-texte >*
OPERATIONS : < maj-lex >*
TYPES : < maj-lex >*
INVARIANTS : < ligne-texte >*

< biblio-def > :: = **CONSTUPLE**
| **(:)**
| **SELECT**
| **AS**

| **INJ**
| **IS**
| **FILTER**
| **LENGTH**
| **#**
| **ITH**
| **IN**
| **SEMPTY**
| **CONS**

< biblio-oper >
:: = **CONSTUPLE** (< nom-obj > (, < nom-obj >)*)
| **CONSTUPLE** (< nom-champ > : < nom-obj >
(, < nom-champ > : < nom-obj >)*)
| < nom-maj > (: < nom-obj >)
| **SELECT** (< nom-maj > , < nom-obj >)
| **AS** (< nom-maj > , < nom-obj >)
| **INJ** (< nom-maj > , < nom-obj >)
| **IS** (< nom-maj > , < nom-maj >)
| **FILTER** (< nom-obj > ,
< nom-maj > (< nom-obj > (, < nom-obj >)*))
| **LENGTH** (< nom-obj >)
| # (< nom-obj >)
| **ITH** (< nom-obj > , < nom-obj >)
| **IN** (< nom-obj > , < nom-obj >)
| **SEMPTY**
| **CONS** (< nom-obj > , < nom-obj >)

< nom-champ > :: = [a - z] [a - z , 0 - 9 , _]*

< nom-obj > :: = [a - z] [a - z , 0 - 9 , _]*

< nom-maj > :: = [A - Z] [A - Z , 0 - 9 , _]*

< constante > :: = [0 - 9] [0 - 9]* | [true, false]

Cette syntaxe s'inspire de [VAN LAMSWEERDE 86]. Elle est cependant plus restrictive sur certains aspects. Par exemple, dans notre syntaxe, le connecteur "**with**" n'introduit pas n'importe quelle explicitation d'opération : seules quelques formes d'explicitations intermédiaires sont permises.

Il est intéressant de constater que figurent dans cette syntaxe trois expressions différentes de conjonctions :

- le symbole "∧" désigne la conjonction logique et est utilisé dans les prédicats exprimant les invariants pouvant porter sur les types d'objets;
- le symbole "**and**" désigne le conjonction de deux opérations figurant dans une même explicitation d'opération; contrairement au "∧", il n'unit pas toujours deux objets à valeurs booléennes;
- le symbole "**AND**" désigne la conjonction d'une précondition et d'une opération; il unit une partie d'explicitation de précondition (à valeur booléenne) et une partie d'explicitation d'opération.

On pourrait s'étonner de ne pas trouver dans l'explicitation d'une précondition la même structure que dans celle d'une opération. La distinction introduite entre <explic-op> et <pre-explic-op> vise à faciliter un contrôle sémantique qu'on pourrait insérer entre l'analyse syntaxique et la production du prototype (voir le chapitre 6 pour les extensions possibles) : une <pre-explic-op> doit toujours être de type booléen.

Deux syntaxes totalement équivalentes sont permises pour l'opération de sélection d'un élément dans un produit cartésien : **SELECT** et "(:)", et pour l'opération donnant pour résultat la longueur d'une suite argument : **LENGTH** et "#".

Deux syntaxes sont permises pour l'opération de construction d'une occurrence de produit cartésien.

La première : **CONSTUPLE** (< nom-obj > (, < nom-obj >)*) suppose que les types de tous les objets composants sont distincts.

La seconde: **CONSTUPLE** (< nom-champ > : < nom-obj > (, < nom-champ > : < nom-obj >)* lève cette hypothèse : les objets composants peuvent être de même type.

Leur rôle dans le produit cartésien est identifié par un nom de champ. Par exemple,

CONSTUPLE (champ$_1$: v_1 , champ$_2$: v_2)

où v_1 et v_2 sont de type T, est une occurrence du produit cartésien CP(T,T).

On peut remarquer que la syntaxe actuelle ne couvre pas toutes les structures du langage. Par exemple, la construction "where" (voir section 2.3.) ne fait pas partie de cette syntaxe.

Chapitre 3

Le Cornell Synthesizer Generator

3.1. Introduction.

Notre but est d'arriver à produire un prototype exécutable en PROLOG à partir d'une spécification formelle écrite en RSL. L'objectif de ce chapitre est de présenter l'outil utilisé pour atteindre ce but. Il s'agit du Cornell Synthesizer Generator (CSG), un outil conçu pour automatiser la construction d'éditeurs guidés par la syntaxe. Le CSG permet de générer des environnements de programmation pour effectuer de manière incrémentale l'analyse syntaxique, les vérifications de types, et la compilation d'un programme en cours d'édition. Il peut également générer des éditeurs pour des langages formels de spécification ou des langages de vérification. Les descriptions à fournir en entrée du CSG pour générer un environnement instancié de manipulation de textes formels doivent être exprimées dans un langage qui lui est particulier : le Synthesizer Specification Language (SSL). Ce langage est basé sur les concepts d'une algèbre de termes et d'une grammaire attribuée. La section 3.2. donne une définition des concepts utilisés. La section 3.3. expose brièvement quelques notions d'édition syntaxique. Pour permettre la compréhension des chapitres 5 et 6 consacrés aux transformations, des rudiments du langage SSL figurent en section 3.4. Cette section énumère ce qu'il est nécessaire de spécifier au CSG pour qu'il génère un éditeur syntaxique instancié à un formalisme : une syntaxe abstraite et une syntaxe concrète du formalisme considéré, des schémas de décompilation décrivant les modes de visualisation de l'arbre abstrait manipulé par l'éditeur, des propriétés qu'on veut voir attachées aux nœuds de l'arbre abstrait, décrites sous forme d'attributs et d'équations sémantiques, et éventuellement des transformations portant sur les objets édités. La section 3.5. expose le fonctionnement du CSG. Enfin, la section 3.6. résume pourquoi nous avons choisi d'utiliser le Cornell Synthesizer Generator.

3.2. Concepts de base.

3.2.1. Syntaxes d'un formalisme.

Les propriétés syntaxiques d'un langage formel caractérisent la structure des constructions bien formées de ce langage.

3.2.1.1.Définition d'une grammaire context-free.

La syntaxe d'un langage formel peut être décrite par une grammaire context-free.

Une grammaire context-free est définie par

- un vocabulaire V, c'est-à-dire l'ensemble (fini) des symboles qui y apparaissent,
- un ensemble de symboles non terminaux N inclut dans ou égal à V; un symbole est non terminal s'il apparaît à gauche d'une règle de production; les autres symboles sont dits terminaux;
- un symbole S appartenant à N dit l'axiome de la grammaire (start symbol) : un symbole qui n'apparaît jamais dans la partie droite d'une règle de production et
- un ensemble P de règles de production définissant les combinaisons permises de symboles. Une règle se présente sous forme d'une partie gauche séparée d'une partie droite par un symbole conventionnel (nous utiliserons "→", ":", ":: =").

En conséquence, une grammaire context-free peut se représenter sous forme d'un quadruplet G = (V, N, S, P) [KNUTH 68].

Chaque production décrit une classe de constructions du langage. Les symboles terminaux désignent des objets dont la définition est supposée connue a priori : les objets de base du langage.

Il est possible de représenter les objets produits par application des règles de la grammaire sous forme d'arbres de dérivation. Le symbole non terminal apparaissant à gauche de chaque production est représenté par un nœud dont les fils sont les symboles terminaux ou non terminaux figurant à droite. La racine de l'arbre de dérivation de la grammaire complète correspond à l'axiome (start symbol); les feuilles correspondent aux symboles terminaux.

3.2.1.2. Syntaxe concrète.

La syntaxe d'entrée concrète est décrite par une grammaire context-free, représentable sous forme d'un arbre dit arbre concret. Chaque règle de cette grammaire définit une concaténation de chaînes de caractères, permise dans le langage considéré.

Une grammaire concrète peut être ambigüe : il est possible qu'en présence d'une chaîne de caractères, on ignore quelles concaténations ont été appliquées pour la produire. Dans ce cas, à une même chaîne correspondent plusieurs arbres concrets possibles. Les ambiguïtés peuvent être levées par la déclaration de priorités.

3.2.1.3. Syntaxe abstraite.

Par opposition à la syntaxe concrète, la syntaxe abstraite décrit la structure d'un langage de manière non ambigüe et sans introduire de détails arbitraires (mots-clefs, ponctuation, ...).

Pour définir une syntaxe abstraite, on peut se donner des domaines syntaxiques, supposés connus a priori, et des opérations de construction à partir de domaines existants de nouveaux domaines syntaxiques. Chaque construction permise est décrite dans une règle de production par un opérateur de construction.

Un domaine syntaxique représente une classe de constructions du langage. On peut le voir comme un ensemble [MEYER 87]. Chaque production est associée à un domaine syntaxique, et définit la structure des éléments de ce domaine.

Les objets produits par application des règles de la grammaire sont représentables sous forme d'arbres de dérivation, aussi appelés arbres syntaxiques ou arbres abstraits.

3.2.1.4. Définition d'une expression régulière.

Si on se donne un ensemble fini ou dénombrable de symboles quelconques, soit A appelé l'alphabet, il est possible de concevoir l'ensemble A* formé de toutes les chaînes finies que l'on peut écrire avec les éléments de A et de λ la chaîne vide. Une expression régulière désigne un sous-ensemble de A* dit ensemble régulier obtenu par répétition un nombre fixé ou non de fois d'une construction d'éléments de A. Ainsi,

- λ est l'expression régulière représentant l'ensemble formé de la chaîne vide,
- $\forall$ a $\in$ A, a désigne l'expresssion régulière représentant l'ensemble formé de la chaîne a.

Si x et y sont deux expressions régulières représentant les ensembles formés respectivement des chaînes x et y,

- x | y est une expression régulière désignant l'ensemble formé de la chaîne x et de la chaîne y,
- xy est une expression régulière désignant l'ensemble formé de la chaîne xy formée en concaténant y à x,
- x* est une expression régulière représentant l'ensemble formé par les chaînes λ, x et toutes leur concaténations.
- (x) est l'expression régulière x.

Par exemple, si A est l'ensemble des caractères ASCII, une expression régulière désigne un ensemble de chaînes de caractères. Elle est représentée par des caractères désignant des symboles de l'alphabet A et par des caractères spéciaux spécifiant des répétitions, des choix, etc. En plus des constructions déjà définies, on convient de noter

- x^+ pour x x* (une ou plusieurs répétitions)
- x? pour λ | x (zéro ou une répétition)
- [abc] pour a | b | c
- [a-z] pour a | b | c | ... | x | y | z.

3.2.2. Sémantique d'un formalisme : définition par grammaires attribuées.

La section 3.2.1 expose comment exprimer la syntaxe abstraite d'un formalisme. Il est aussi utile d'en préciser la sémantique. Les grammaires attribuées [KNUTH 68] [LIVERCY 78] permettent de définir toute propriété d'un langage déterminable de manière statique. Elles sont généralement utilisées pour exprimer la sémantique de langages de programmation en ajoutant à la structure donnée par la syntaxe de ces langages des propriétés sémantiques exprimées sous forme de valeurs d'attributs. Elles peuvent également être utilisées pour d'autres manipulations sur les langages formels, par exemple pour exprimer des transformations.

3.2.2.1. Définition d'une grammaire attribuée.

Un traitement plus complet de ce qui suit figure dans [KNUTH 68] et [LIVERCY 78].

La grammaire décrivant la syntaxe abstraite d'un formalisme introduit un ensemble de domaines syntaxiques contenant les constructions bien formées de ce langage formel. Il est possible d'attribuer des valeurs à chacune de ces constructions. Une grammaire attribuée est une grammaire context-free étendue en attachant des attributs à ses symboles.

Pour chaque symbole X de la grammaire, on définit un ensemble d'attributs A(X). Les attributs sont définis par des fonctions associées aux productions de la grammaire. Une équation sémantique associe à chaque attribut $\alpha(X)$ un ensemble de valeurs V_α de l'attribut. Pour un symbole non terminal X donné, on distingue les attributs synthétisés de X dont les valeurs ne dépendent que de valeurs d'autres attributs de X et/ou de valeurs d'attributs des descendants de X, des attributs hérités de X définis en termes de valeurs d'attributs des ancêtres de X.

A(X) est une partition des ensembles

- $A_S(X)$: ensemble des attributs synthétisés de X et
- $A_h(X)$: ensemble des attributs hérités de X.

Soient les productions de la grammaire de la forme:

$$X : Y1, \ldots, Yn ;$$

$A_S(X)$ est défini par l'ensemble des équations sémantiques de la forme

$$\alpha(X) = f (\beta(X), \beta(Y1), ..., \beta(Yn))$$

où chaque équation signifie que les valeurs de l'attribut synthétisé $\alpha(X)$ sont fonction des valeurs d'un ou plusieurs autres attributs de X: $\beta(X)$ et/ou des valeurs d'un ou plusieurs attributs appartenant aux symboles du membre de droite de la production (les fils de X): $\beta(Y1), ..., \beta(Yn)$.

Soient toutes les productions de la grammaire où X figure dans la partie droite, que nous noterons: $Y : X, Y1, ..., Yn.$

$A_h(X)$ est défini par l'ensemble des équations sémantiques de la forme

$$\alpha(X) = f (\beta(Y), \beta(Y1), ..., \beta(Yn))$$

où chaque équation signifie que les valeurs de l'attribut hérité $\alpha(X)$ sont fonction des valeurs d'un ou plusieurs attributs $\beta(Y)$ du membre de gauche de la production (le père de X) et éventuellement d'un ou plusieurs attributs $\beta(Yi)$ ($i = 1, ...,n$) des symboles du membre de droite de la production.

On appellera les attributs β les arguments de α et on dira que α dépend des β; f est appelée la fonction sémantique de α.

Les valeurs des attributs synthétisés de X ne dépendent que des valeurs d'autres attributs de X et/ou des valeurs des attributs des symboles de droite de la production. Autrement dit, dans un arbre syntaxique, la valeur d'un attribut synthétisé du noeud correspondant à X dépend uniquement de la valeur des attributs de X et des fils de ce noeud. Par conséquent, les attributs synthétisés sont évalués de bas en haut. Par opposition, les attributs hérités sont évalués en allant du haut vers le bas. Les valeurs des attributs hérités d'un noeud X ne dépendent que de ses ancêtres.

Remarque 1 : Si X est l'axiome de la grammaire ou autrement dit la racine de l'arbre syntaxique, $A_h(X)$ est vide. Si X est un symbole terminal de la grammaire, $A_S(X)$ est vide.

Remarque 2 : S'il y a des circuits dans les définitions sémantiques, il est possible de ne pas parvenir à évaluer tous les attributs. Dans ce cas, la grammaire est dite circulaire: le graphe de dépendance représentant les dépendances fonctionnelles entre les instances d'attributs contient au moins un cycle.

Remarque 3 : Si l'ordre dans lequel les équations sémantiques sont déclarées n'a aucune importance, les calculs de valeurs d'attributs doivent par contre être effectués dans un certain ordre [MEYER 87]. Pour évaluer un attribut α, il faut disposer des valeurs de ses arguments.

3.2.2.2. Avantages des grammaires attribuées.

- L'utilisation d'attributs hérités permet de donner à des noeuds de l'arbre syntaxique des valeurs dépendant du contexte, de l'environnement de ces noeuds.

- Les équations sémantiques sont définies au niveau d'une règle syntaxique. L'impact d'une modification d'équation sémantique est par conséquent facilement localisable.

- L'introduction d'attributs peut se faire au fur et à mesure des besoins. La méthode supporte un développement incrémental.

3.3. Edition syntaxique.

Le Cornell Synthesizer Generator (CSG) est un outil permettant la génération d'un éditeur "bimodal" (textuel - structurel) paramétré sur un langage spécifié par l'utilisateur. La structure d'une telle spécification fait l'objet de la section 3.4. Le but de la présente section est de rappeler ce qu'est un éditeur syntaxique. Nous nous sommes inspirés pour sa rédaction de [CRISMER 88].

Un éditeur syntaxique est un outil qui permet l'édition d'un texte formel syntaxiquement correct : il vérifie en cours d'édition la syntaxe du programme introduit, voire aussi sa sémantique statique c'est-à-dire ce qu'il est possible de vérifier sans avoir à effectuer une exécution et signale les erreurs; parfois même, il les corrige automatiquement. Pour ce faire, il manipule une représentation arborescente du texte formel édité (un arbre syntaxique) et non plus une représentation sous forme de chaînes de caractères. L'arbre manipulé par l'éditeur est l'arbre de dérivation de l'objet édité par rapport à la grammaire context-free décrivant la syntaxe abstraite du formalisme auquel l'éditeur est taillé.

3.3.1. Edition textuelle.

Un éditeur syntaxique en mode textuel traite un programme comme un texte : ligne par ligne, caractère par caractère. Ses commandes ont ces objets pour arguments. Il est capable d'insérer des mots-clefs là où ils manquent pour rendre le programme syntaxiquement correct. De plus, il signale les erreurs de sémantique statique.

3.3.2. Edition structurelle à entrée structurée.

Un éditeur syntaxique travaillant en mode structurel considère un programme comme un arbre de dérivation. Les commandes d'édition ont pour arguments des sous-arbres.

L'édition d'un texte formel se fait progressivement par remplissage de gabarits syntaxiques (templates) affichant les mots-clefs, la ponctuation et des "emplacements" (placeholders) à remplir. Le processus de remplissage est guidé par le système grâce à sa connaissance de la syntaxe abstraite du formalisme. Remplir un emplacement correspondant à un symbole non terminal de la grammaire du langage se fait par insertion de nouveaux gabarits jusqu'à ce que les symboles terminaux soient atteints. Ceux-ci sont enfin introduits textuellement. Le remplissage de certains emplacements peut être facultatif.

Il est possible de se déplacer dans l'arbre représentant le texte formel, d'en sélectionner certaines parties. Modifier un texte formel revient à restructurer son arbre de dérivation en y supprimant, ajoutant, remplaçant des sous-arbres.

Parmi les outils suivant ce paradigme d'édition, citons MENTOR, un système conçu pour manipuler des données structurées, recevant une description abstraite de formalisme écrite en METAL, et possédant un langage spécialisé de manipulation d'arbres : MENTOL [DONZEAU-GOUGE 84].

Contrainte liée à l'édition structurelle.

Un texte formel partiellement développé doit pouvoir être considéré comme complet du point de vue du système. Celui-ci doit pouvoir reconnaître un symbole non terminal non développé. Si on considère le CSG, dans une spécification SSL d'éditeur syntaxique structurel, il est nécessaire d'ajouter, pour chaque symbole non terminal, une production définissant un terme utilisé pour désigner ce symbole non développé : on appellera cette production la "production complétive"; elle définit le "completing term". Dans le cas de MENTOR, en MENTOL, on parlera à cette fin de nœuds terminaux spéciaux qu'on appellera "métavariables".

3.3.3. Edition "bimodale" textuelle - structurelle.

Un éditeur "bimodal" est un éditeur permettant de faire de l'édition textuelle ou structurelle. Un nœud non terminal peut être développé en mode structurel par insertion de gabarits ou en mode texte par l'introduction d'un fragment textuel qu'un analyseur syntaxique traduit a posteriori en sous-arbre qui se greffe dans la représentation interne arborescente de l'objet édité.

C'est ce dernier type d'éditeur que génère le CSG.

Notons que le CSG respecte une stricte division entre les constructions introduites en mode structurel et celles introduites en mode textuel : un objet introduit par remplissage d'un gabarit ne peut être manipulé que sous la forme d'une unité structurelle, un objet édité en mode texte ne peut être modifié qu'en mode texte. Cette caractéristique distingue le CSG d'autres systèmes : dans MENTOR par exemple, toute entrée textuelle est analysée pour créer une structure correspondant au texte; par la suite, seules des modifications structurelles sont permises [REPS 89].

3.4. Structure d'une spécification SSL.

Le Synthesizer Specification Language (SSL) est le langage fonctionnel qui permet de spécifier les caractéristiques du formalisme auquel l'éditeur généré par le CSG doit être instancié : syntaxe abstraite du langage à éditer, attributs et équations sémantiques, syntaxe concrète, transformations. Chacune de ces caractéristiques est développée aux points suivants. Pour les détails, on peut consulter [REPS 87] dont ce qui suit s'inspire; les formules encadrées sont extraites de cette référence.

3.4.1. Spécification de la syntaxe abstraite.

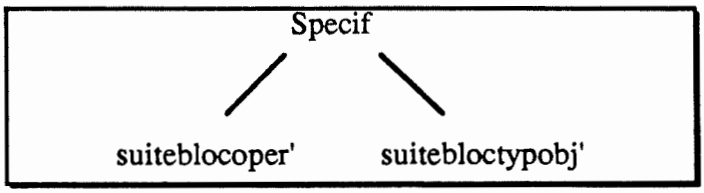
3.4.1.1. Définition d'une syntaxe abstraite.

Rappelons qu'une syntaxe abstraite s'exprime par les concepts d'une algèbre de termes: "phylum", "opérateur" et "terme" qui se définissent mutuellement. Les domaines syntaxiques sont appelés "phyla" : ils correspondent à des types de données. Un phylum est un ensemble de termes. Un opérateur n-aire est une fonction de construction d'un terme à partir de n termes. Les opérateurs étant typés, le résultat et chaque position d'argument correspondent à un phylum. Un terme est un n-uplet s'obtenant par l'application d'un opérateur n-aire à n termes ($0 \leq n$) appartenant aux phyla qui conviennent. C'est un élément d'un domaine syntaxique, représentable sous forme d'un arbre de dérivation.

Exemple

specification : Specif (suiteblocoper suitebloctypobj) ;

specification, suiteblocoper et suitebloctypobj sont des noms de phylum. Specif est un opérateur binaire. Un terme specification' du phylum "specification" est un couple (suiteblocoper', suitebloctypobj') où suiteblocoper' et suitebloctypobj' sont des termes respectivement des phyla "suiteblocoper" et "suitebloctypobj". La partie de l'arbre de dérivation de spécification' correspondant à cette production est représentée ci-dessous.



3.4.1.2. Phyla primitifs.

Le CSG reconnaît quelques phyla primitifs pré-définis - par exemple "BOOL": les valeurs de vérité, "INT" : les entiers, etc. - sur lesquels sont pré-définies quelques opérations telles "&&" : la conjonction logique, "*" : la multiplication, etc. Les autres phyla doivent être définis par l'utilisateur.

3.4.1.3. Déclarations de phyla.

Les nouveaux phyla et opérateurs sont déclarés en termes d'autres phyla ou récursivement en termes d'eux-mêmes.

Il existe deux sortes de déclarations de phylum : les productions et les déclarations de lexèmes. Les productions définissent les phyla correspondant aux symboles non terminaux de la grammaire. Par opposition, les lexèmes correspondent aux symboles terminaux.

a. Productions.

Une production définit un nouvel opérateur et rassemble dans un phylum tous les termes qu'il est possible de construire avec cet opérateur. La forme générale d'une production est :

$$\text{phylum}_0 : \text{operator} (\text{phylum}_1 \ldots \text{phylum}_n) ;$$

"phylum_0" comprend tous les termes construits par application de l'opérateur n-aire "operator" aux termes arguments appartenant aux phyla "phylum_i", $i = 1,\ldots,n$.

Notons qu'il est possible de factoriser différentes productions portant sur un même phylum et de regrouper les opérateurs portant sur les mêmes phyla arguments.

Exemple 1

specification : Specif (suiteblocoper suitebloctypobj) ;

Toute spécification RSL (voir chapitre 2) est un terme du phylum "specification", terme désigné par "Specif" et construit par application de "Specif" à des termes de "suiteblocoper" et "suitebloctypobj" où "suiteblocoper" est un phylum représentant les suites de blocs de description d'opérations, et "suiteblocotypobj", un phylum désignant les suites de blocs de description de types d'objets.

Exemple 2

```
suiteblocoper      : VideBlocOper ( )
                   | ConsBlocOper ( blocoper suiteblocoper ) ;
```

La déclaration a été factorisée; "|" désigne les alternatives. Le phylum "suiteblocoper" est défini en termes de lui-même. Il représente les suites de descriptions de blocs d'opérations. Les temes qui le composent sont :

- une suite vide désignée par VideBlocOper ( ) et
- désignées par "ConsBlocOper (blocoper suiteblocoper)", les suites composées
  - d'un terme du phylum "blocoper" représentant les descriptions de blocs d'opération et
  - récursivement d'un terme du phylum "suiteblocoper".

b. Déclarations de lexèmes.

Un lexème est un phylum inclus dans le phylum primitif "STR" comprenant l'ensemble de toutes les chaînes de caractères. De manière générale, une déclaration de lexème se présente comme suit :

```
phylum : lexeme-name < regular-expression >;
```

"phylum" désigne toutes les chaînes de caractères qu'il est possible de générer à partir de l'expression régulière "regular-expression".

Les expressions régulières reconnues par le CSG désignent des ensembles de chaînes de caractères, et sont, à quelques exceptions près, celles admises par LEX, un outil disponible sous UNIX et utilisé par le CSG pour générer un analyseur lexical (voir [REPS 87] pp 13 - 14).

Notons qu'un phylum peut être associé à plusieurs déclarations de lexèmes.

Remarque : L'ordre de déclaration des productions n'est pas significatif. S'il est par conséquent possible d'ajouter des productions sans tenir compte de l'ordre de déclaration de celles-ci, les déclarations de lexèmes doivent par contre être ordonnées. En effet, lorsqu'à une chaîne de caractères qu'il reçoit en entrée, l'analyseur lexical peut faire correspondre plusieurs expressions régulières, il choisit la première spécifiée. Il est par conséquent nécessaire, si l'on veut que des mots-clefs soient reconnus, de les déclarer avant les classes d'identifiants.

Exemple 1.

[a - z] [a - z0 - 9_] *

est une expression régulière désignant l'ensemble des chaînes de caractères alphanumériques minuscules et " _ ", chaînes commençant par un caractère alphabétique minuscule.

Exemple 2.

IDMIM : IdMinLex < [a - z] [a-z0 - 9_] * > ;

Le phylum IDMIN est défini par le lexème dont les termes sont les chaînes de caractères désignées par l'expression régulière [a - z] [a - z0 - 9_] *.

Exemple 3.

```
LENGTH    : LengthLex < "LENGTH" >
          | DieseLex < "#" >
          ;
```

Le phylum LENGTH est associé à deux déclarations de lexèmes; ses termes sont les chaînes de caractères "LENGTH" et "#".

c. Déclaration d'un symbole "distingué" (start symbol).

Un phylum doit être déclaré comme axiome de la grammaire, ce qui se fait selon le schéma suivant :

```
root phylum ;
```

Le phylum "phylum" est la racine de l'arbre syntaxique.

Pour compléter cette brève description d'une spécification de syntaxe abstraite en SSL, mentionnons que chaque phylum contient deux termes remarquables liés à la spécification d'un éditeur syntaxique: le terme "complétif" (completing term) et le terme "emplacement" (placeholder term). Ces termes jouent des rôles particuliers dans le processus d'édition. Le premier représente le sous-arbre non développé du phylum auquel il appartient; le second dénote les endroits où une insertion d'édition peut être effectuée* .

3.4.2. Attributs et équations sémantiques.

Un texte SSL est une spécification formelle de langage. Cette section a pour objectif d'introduire au fonctionnement du mécanisme des attributs en SSL. Remarquons que nous n'avons pas utilisé ce mécanisme pour définir la sémantique de RSL mais essentiellement pour générer et transmettre les informations nécessaires aux transformations qui font l'objet des chapitres 4 et 5. Le but étant ici d'aider à comprendre ces chapitres, la description qui suit n'est pas exhaustive.

3.4.2.1. Déclaration des attributs.

La grammaire décrivant la syntaxe abstraite introduit un ensemble de domaines syntaxiques, appelés phyla. A chaque domaine syntaxique, on peut associer des attributs dont les valeurs décrivent des propriétés des objets - termes - de ces phyla. Le domaine de valeurs de chaque attribut est également un phylum.

* Lors d'une édition, un terme inséré remplace le "placeholder term"; inversément, si on efface un terme, celui-ci est remplacé par le "placeholder term". "Completing terms" et "placeholder terms" sont généralement les mêmes. On les distingue lorsqu'une insertion est facultative. Dans ce cas, le "completing term" n'a pas de représentation visuelle, tandis que le "placeholder term" apparaît à l'écran comme un emplacement à remplir. Lorsque la sélection se trouve sur un phylum facultatif non développé, le "placeholder term" apparaît. S'il n'est pas remplacé par un terme inséré, lorque la sélection se déplace, il est remplacé par le "completing term".

Les attributs doivent être déclarés.

Exemple : suiteblocoper { syn fform mafo;
inh fterfter typexpdec ; } ;

Mafo est un attribut synthétisé de type "fform" du phylum "suiteblocoper"; typexpdec est un attribut hérité de type "fterfter" du phylum "suiteblocoper"; "fform" et "fterfter" sont des phyla déclarés dans la syntaxe abstraite. La signification de ces attributs et le pourquoi de ces phyla sont donnés dans le chapitre 4.

SSL ne permet pas de déclarer d'attribut hérité pour le phylum déclaré comme racine. Ceci est une conséquence de la remarque 1 du point 3.2.2.1.

3.4.2.2. Equations sémantiques.

A chaque règle de la grammaire, on associe les équations sémantiques définissant les valeurs des attributs déclarés. En SSL, seules les productions sont susceptibles d'être enrichies d'équations sémantiques. On n'associe pas d'attributs aux lexèmes. Cette restriction est plus générale que celle de la remarque 1 du point 3.2.2.1. : l'ensemble de tous les attributs des symboles terminaux est vide, et non pas seulement leur ensemble d'attributs synthétisés.

Une équation sémantique a la forme générale suivante :

phylum.a = expression ;

où "phylum est le nom d'un phylum et "a" celui d'un attribut déclaré pour ce phylum. "phylum.a" désigne une valeur d'attribut. Les valeurs d'attributs étant des termes de phylum, "expression" doit définir un terme du même phylum.

Nous avons utilisé des expressions de la forme

- phylum.a
- operator ( )
- operator ($expression_1$, ..., $expression_n$)
- function ($expression_1$, ..., $expression_n$)
- with-expression.

"operator" est un opérateur apparaissant dans la syntaxe abstraite et désigne un terme du phylum pour lequel cet opérateur est déclaré.

"function ($expression_1$, ..., $expression_n$)" est un appel de fonction. Les "$expression_i$" (i = 1, ..., n) sont les paramètres actuels devant correspondre aux paramètres formels apparaissant dans la déclaration de fonction. Toute fonction doit être déclarée.

Une déclaration de fonction a la forme

$phylum_0$ $identifier_0$ ($phylum_1$ $identifier_1$, ..., $phylum_n$ $identifier_n$) { expression } ;

et signifie que "$identifier_0$" est une fonction n-aire dont le résultat est un terme du phylum "$phylum_0$" et dont les n paramètres formels de types "$phylum_i$" sont nommés "$identifier_i$", (i = 1, ..., n).

Une "with-expression" est une expression conditionnelle à choix multiples. Sa syntaxe est la suivante :

with ($expression_0$) ($pattern_1$: $expression_1$, $pattern_2$: $expression_2$, ... $pattern_n$: $expression_n$)

La valeur de l'expression "with" est la valeur de l' "$expression_i$" correspondant au premier "$pattern_i$" tel que "$expression_0$" soit une instance de "$pattern_i$"; dans ce cas, il existe une substitution θ telle que "$pattern_i\ \theta$ = $expression_0$" ("call-by-pattern"). Des variables libres peuvent apparaître dans les "$pattern_i$". Si le "pattern-matching" réussit, les variables libres du "pattern" sont liées aux variables de l'expression correspondante.

Exemple 1.

- une déclaration de fonction dont le corps est une "with-expression" :

```
BOOL dssfterme ( fsterme a , fterme b )
{ with ( a )
    ( VideFTerme ( ) : false,
      ConsFTerme ( x, y ) :
            ( x == b )      ? true
                            : dssfterme ( y, b )
    )
} ;
```

Les arguments de cette fonction sont une suite de termes a et un terme b. Le résultat est vrai si le terme b figure dans la suite de termes a, est faux sinon.

Remarquons la syntaxe de l'expression conditionnelle à deux branches : ce qui en Pascal se noterait "if x = b then true else dssfterme (y, b) ;" se note "(x == b) ? true : dssfterme (y, b)".

Exemple 2.

- équations sémantiques jointes aux productions déclarant un phylum :

```
suiteblocoper      : VideBlocOper ( )
    { $$.mafo = FFormVide ( ) ; }
                    | ConsBlocOper ( blocoper suiteblocoper )
    { $$.mafo = FFormEt ( blocoper.mafo, suiteblocoper$2.mafo ) ;
      blocoper.typexpdec = $$.typexpdec ;
      suiteblocoper$2.typexpdec = $$.typexpdec ; } ;
```

"FFormVide ( )" est une expression du type "operator ( )";
"FFormEt (blocoper.mafo, suiteblocoper$2.mafo)" est une expression du type "operator (expression$_1$, expression$_2$)";
"$$.typexpdec" est une expression du type "phylum.a".

La signification des attributs et l'objectif poursuivi par la déclaration de ces équations sont exposés au chapitre 4.

L'abbréviation "$$" peut être utilisée pour désigner le nom du phylum de gauche de la production. Si la production contient plus d'une occurrence d'un phylum, les différentes occurrences doivent être numérotées $1, $2, etc.

Lorsque plusieurs productions définissent les termes d'un même phylum, il est nécessaire d'attacher à chacune d'elles une équation sémantique pour chacun des attributs synthétisés. Dans l'exemple, une équation sémantique définissant l'attribut "mafo" est jointe à chacune des productions.

3.4.2.3. Evaluation des attributs.

La spécification SSL n'est acceptée que si la grammaire attribuée est bien formée ce qui signifie que pour chaque production, il y a exactement une équation sémantique par attribut déclaré pour ce phylum.

De plus, il n'est pas possible au CSG de construire un arbre de dérivation et d'évaluer les attributs si la grammaire attribuée est circulaire. Dans ce cas, le graphe de dépendance représentant les dépendances fonctionnelles entre les instances d'attributs de l'arbre considéré contient au moins un cycle, ce qui revient à exiger pour calculer une valeur d'attribut de disposer de cette valeur (cfr. remarque 2 du point 3.2.2.1.).

Deux schémas d'évaluation des attributs sont disponibles sur le CSG. Ils se distinguent notamment par l'ordre dans lequel les attributs sont évalués (cfr. remarque 3 du point 3.2.2.1.). Selon le schéma choisi, la non-circularité de la grammaire est également testée ou non. Pour les détails, on consultera [REPS 87 pp 28 et suivantes].

3.4.2.4. Grammaires attribuées et développement incrémental.

L'introduction d'attributs peut se faire au fur et à mesure des besoins. Le développement incrémental est concrétisé en SSL par les possibilités :

1) d'ajouter de nouveaux attributs aux phyla déjà déclarés;
2) d'ajouter de nouvelles équations sémantiques aux productions déjà déclarées;
3) d'ajouter des déclarations d'opérateurs et leurs équations sémantiques aux phyla déjà déclarés.

3.4.3. Spécification de la syntaxe concrète.

Nous distinguerons ici deux syntaxes concrètes :

1) la syntaxe d'entrée concrète qui permet de convertir un texte formel en un arbre abstrait lui correspondant, et

2) la syntaxe de visualisation de l'arbre abstrait; celle-ci s'exprime par des schémas de décompilation.

Les mécanismes traitant les représentations en entrée et celles en sortie sont totalement indépendants. En conséquence, ces deux syntaxes peuvent être les mêmes ou non.

3.4.3.1. Spécification d'une syntaxe concrète d'entrée.

Le CSG utilise pour le traitement de la syntaxe concrète d'entrée les services d'un analyseur lexical et d'un analyseur syntaxique générés respectivement par LEX et YACC (section 3.4.3.2.).

La traduction du texte en arbre de syntaxe abstraite est définie en SSL par des équations sémantiques associées aux productions de la syntaxe concrète d'entrée. Chaque texte bien formé par rapport à la syntaxe concrète détermine un arbre concret que le mécanisme des attributs permet de traduire alors en un fragment de l'arbre abstrait (section 3.4.3.3.).

3.4.3.2. LEX et YACC.

Si nous présentons ici une brève description de LEX et YACC [LESK 75] [JOHNSON 75] [JOHNSON 78], c'est parce qu'il n'est pas possible d'ignorer totalement ces outils lorsqu'on utilise le CSG. Ainsi, par exemple, l'ordre imposé aux déclarations de lexèmes est lié au fonctionnement de LEX, tandis que les diagnostics apparaissant lors du traitement de la syntaxe concrète sont directement ceux de YACC.

3.4.3.2.1. Description générale.

LEX et YACC sont deux générateurs de programmes disponibles dans un environnement de programmation UNIX. Un générateur de programme est un outil qui recevant la spécification d'une tâche produit un programme pour effectuer cette tâche. LEX est conçu pour construire des analyseurs lexicaux. YACC produit des analyseurs syntaxiques. LEX et YACC peuvent chacun être utilisés seuls; néanmoins, ils ont été conçus pour coopérer. Utilisés en tandem, comme dans le CSG, ils permettent de réaliser le passage d'un texte formel à son arbre de syntaxe concrète (voir fig.3.1.).

3.4.3.2.2. LEX.

LEX reçoit comme données des règles représentables sous forme d'une table dont chaque entrée associe à une expression régulière une action à effectuer exprimée sous forme d'un fragment de programme. LEX traduit ces règles en un programme capable de détecter dans les chaînes de caractères qu'il reçoit les occurrences des expressions régulières spécifiées et d'effectuer les actions qui y correspondent dans la table.

Un analyseur lexical est un tel programme. Il est capable d'assembler des caractères alphanumériques en noms, de reconnaître des classes de caractères; il traite les blancs, les retours à la ligne, les commentaires; il peut associer à une représentation de valeur la valeur représentée. Les symboles qu'il construit en traitant les caractères sont appelés "tokens" et peuvent être utilisés comme symboles terminaux d'une grammaire context-free.

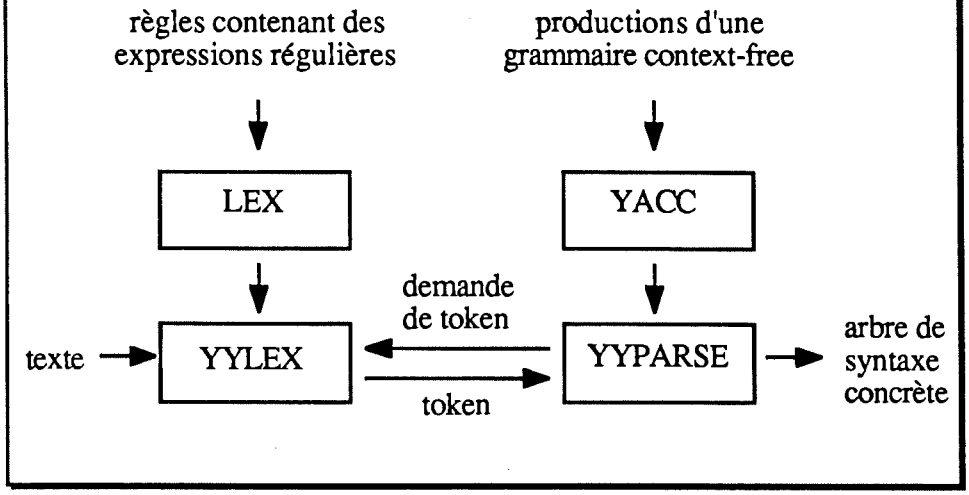
Un programme d'analyse lexicale produit par LEX accepte des spécifications ambigües. Quand plus d'une expression régulière peut correspondre aux caractères du texte en entrée, l'expression permettant de traiter le plus grand nombre de caractères est choisie. Parmi les expressions permettant de traiter un même nombre de caractères, c'est la première déclarée qui est choisie. On retrouve ici la règle conditionnant l'ordre des déclarations de lexèmes dans une spécification SSL (voir section 3.4.1.3.).

3.4.3.2.3. YACC.

YACC est conçu pour transformer une spécification de grammaire context-free en un analyseur syntaxique paramétré sur cette grammaire. L'analyseur syntaxique généré a recours aux services d'un analyseur lexical qui lui fournit des suites de "tokens". Il vérifie alors si la suite fournie appartient à la syntaxe du langage pour lequel il est taillé.

fig. 3.1.

Schéma de LEX et YACC fonctionnant en tandem pour produire une représentation arborescente d'un texte.



La compréhension du fonctionnement de l'analyseur facilite le traitement des ambiguïtés et des recouvrements d'erreurs.

L'analyseur produit par YACC est une machine à états (en nombre fini), munie d'une pile et capable de retenir le "token" suivant dans le texte en entrée la partie couramment traitée. L'état courant est celui au sommet de la pile. Initialement, la machine est dans l'état 0, la pile ne contient que l'état 0 et aucun token n'a été lu.

Selon son état courant, l'analyseur décide s'il a besoin de retenir un nouveau "token". Si oui, il appelle l'analyseur lexical qui lui en renvoie un (voir fig.3.1.).

Selon son état courant et éventuellement la valeur du "token" retenu, l'analyseur décide de procéder à l'une des quatre actions suivantes :

- glisser un nouvel état sur la pile;

- réduire une série d'états;

  l'analyseur a reconnu le membre de droite d'une règle; il enlève de la pile les états correspondant aux "tokens" identifiés pour ce membre de droite et glisse sur la pile un nouvel état correspondant au membre de gauche de la règle;

- accepter l'entrée;

  cela signifie que le texte en entrée a été traité et correspond à la syntaxe donnée;

- signaler une erreur;

  lorsque le texte en entrée n'est pas syntaxiquement correct, l'analyseur signale une erreur et entreprend éventuellement de récupérer la situation pour continuer le traitement.

La grammaire soumise à YACC peut être ambigüe. Il se peut qu'une même chaîne puisse être structurée de plus d'une façon.

Considérons par exemple la règle

< explicop > :: = < explicop > **and** < explicop >

et l'explicitation suivante soumise à l'analyseur :

explicop$_1$ **and** explicop$_2$ **and** explicop$_3$.

Lorsque l'analyseur a lu "explicop$_2$", il est confronté à deux choix possibles :

1) il réduit selon la règle "explicop$_1$ **and** explicop$_2$" en "explicop" puis traite "explicop$_3$"; le résultat final équivaudra à "(explicop$_1$ **and** explicop$_2$) **and** explicop$_3$" (associativité à gauche) ou

2) il glisse l'état correspondant à "explicop$_3$" sur la pile, et réduit alors "explicop$_2$ **and** explicop$_3$" en "explicop". Dans ce cas, le résultat équivaut à "explicop$_1$ **and** (explicop$_2$ **and** explicop$_3$)" (associativité à droite).

Cette situation est appelée un conflit "shift/reduce".

Il se peut aussi que l'analyseur soit confronté au cas où deux règles différentes permettent la réduction d'une même chaîne en entrée. Dans ce cas, on a un conflit "reduce/reduce".

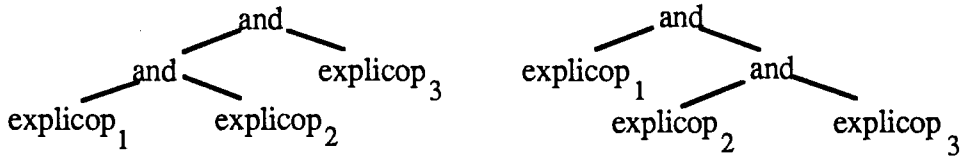
En cas de conflit, à une même chaîne donnée correspondent plusieurs arbres de dérivation possibles (voir fig. 3.2.). YACC détecte les conflits, les signale et génère un analyseur produisant un arbre unique, en tranchant les conflits selon les règles suivantes :

1) dans un conflit "shift/reduce", par défaut, l'analyseur glisse un nouvel état sur sa pile (associativité à droite);

2) dans un conflit "reduce/reduce", par défaut, l'analyseur réduit selon la première règle déclarée.

fig. 3.2.

Deux arbres de dérivation peuvent être construits à partir de l'explicitation "explicop$_1$ **and** explicop$_2$ **and** explicop$_3$ ".



associativité à gauche associativité à droite

Par défaut, l'analyseur construit par YACC génère l'arbre de droite.

Si l'on désire une associativité à droite, on peut parfaitement tolérer les conflits "shift/reduce". Réécrire les règles de la grammaire pour les éviter peut conduire à la génération d'analyseurs plus lents. Cependant, le comportement de l'analyseur est moins évident à contrôler en cas de conflit "reduce/reduce". Cette dernière situation est à éviter autant que possible.

Certains conflits peuvent être résolus par la déclaration de priorités (voir section 3.4.3.2.3).

3.4.3.3. Déclaration de la syntaxe concrète d'entrée.

3.4.3.3.1. Passage de la syntaxe concrète à la syntaxe abstraite.

Le passage de la syntaxe concrète à la syntaxe abstraite utilise le mécanisme des attributs. L'analyseur syntaxique construit à partir des règles de la syntaxe concrète et des équations sémantiques qu'on y associe un arbre de dérivation concret attribué dont un des attributs est l'arbre abstrait correspondant. Le lien entre l'arbre abstrait d'un terme et l'attribut correspondant de l'arbre concret est exprimé dans une "déclaration d'entrée".

Nous n'avons utilisé que des déclarations d'entrée de la forme

```
abstract-syntax-phylum ~ concrete-syntax-phylum.attribute ;
```

où "abstract-syntax-phylum" et "concrete-syntax-phylum" sont des phyla, et "attribute" est un attribut synthétisé du phylum "concrete-syntax-phylum".

Une telle déclaration joue un double rôle :

- en édition en mode textuel (voir section 3.3.1.), elle spécifie que la racine du sous-arbre abstrait couramment édité doit être analysée selon les déclarations d'analyse syntaxique du "concrete-syntax-phylum";

- la valeur de l'attribut "concrete-syntax-phylum.attribute" doit être insérée dans l'arbre abstrait à la place de la valeur courante du sous-arbre de "abstract-syntax-phylum".

Evidemment, il faut que la valeur de "concrete-syntax-phylum.attribute" soit un terme du phylum "abstract-syntax-phylum".

Notons l'intérêt dans notre cas de disposer d'un arbre abstrait et d'un arbre concret séparés. Nous n'avons besoin que de l'arbre concret correspondant à la syntaxe du langage RSL. Par contre, nous devons disposer d'un arbre abstrait plus touffu. En effet, nous calculons des attributs dont les valeurs appartiennent à des domaines de la syntaxe abstraite correspondant au FOPL et à PROLOG (voir chapitres 4 et 5).

3.4.3.3.2. Déclarations d'analyse syntaxique.

La grammaire décrivant la syntaxe concrète est définie comme suit :

- l'axiome est le symbole "YYentry";

- les symboles non terminaux sont "YYentry" et tous les phyla non terminaux déclarés dans une production de la syntaxe concrète;

- les symboles terminaux sont tous les caractères ASCII, tous les lexèmes déclarés par l'utilisateur (voir section 3.4.1.3.) et un symbole spécial "p_ENTRY" pour chaque déclaration d'entrée où p apparaît à gauche;

- les productions seront notées $A \rightarrow \alpha$ où A est un symbole non terminal de la grammaire concrète et α une suite de symboles terminaux et non terminaux de cette grammaire.

Une production "phylum → lexeme-name" est associée à chaque déclaration de lexème: "phylum : lexeme-name < regular-expression > ;".

Une production "YYentry → p_ENTRY.p'" est associée à chaque déclaration d'entrée
"p ~ p'.attribute ;".

Une production "$phylum_0$ → tokens $phylum_1$ tokens $phylum_2$ ... $phylum_n$ tokens" est associée à chaque production de la syntaxe concrète qui a la forme :

$phylum_0$ :: = operator (tokens $phylum_1$ tokens $phylum_2$ ... $phylum_n$ tokens) ;

Le symbole ":: =" dénote une déclaration d'analyse syntaxique, les "tokens" sont des séquences de constantes de type CHAR. Le nom d'opérateur "operator" est optionnel.

Notons que ces productions aussi peuvent être factorisées.

Exemple.

```
(1)  ysuiteblocoper        { syn suiteblocoper a ; } ;
     yblocoper      { syn blocoper a ; } ;
```

```
(2) ysuiteblocoper          :: = ( )
            { $$.a = VideBlocOper ( ) ; }
                  | ( yblocoper ysuiteblocoper )
            { $$.a = ConsBlocOper ( yblocoper.a, ysuiteblocoper$2.a ) ; }
                  ;

(3) suiteblocoper ~ ysuiteblocoper.a ;

(4) yblocoper       :: = ( SPECOP yidmaj FORMDEC ydecformop OUTLINE
                              yprofilop LEXIC ylexop END yidmaj POINT )
            { $$.a = BlocOper ( yidmaj$1.a, ydecformop.a, yprofilop.a,
                                          ylexop.a, yidmaj$2.a ; }
                  ;

(5) blocoper ~ yblocoper.a ;
```

Des attributs synthétisés "a" dont les valeurs appartiennent aux phyla de la syntaxe abstraite "suiteblocoper" et "blocoper" sont définis pour les phyla de la syntaxe concrète "ysuiteblocoper" et "yblocoper" (1).

Pour chaque production de la syntaxe concrète, une valeur est donnée à ces attributs (2) et (4). La production (2) exprime < bloc-oper >*. La production (4) correspond à la règle

< bloc-oper > :: =
SPECIFICATION OF OPERATION < nom-maj >
FORMAL DESCRIPTION : < dec-form-op >
OUTLINE : < profil-op >
LEXICON : < lexicon-op >
END < nom-maj > .

de la syntaxe concrète (voir chapitre 2, section 2.6).

La factorisation des déclarations est illustrée en (2), la présence de lexèmes (SPECOP, FORMDEC, OUTLINE, LEXIC, END, POINT) peut s'observer dans la production (4).

Les déclarations d'entrée (3) et (5) établissent la correspondance entre les attributs "a" de l'arbre concret et l'arbre abstrait.

3.4.3.3.3. Déclarations de priorités.

Nous avons vu que YACC détecte les ambiguïtés des productions de la syntaxe concrète et les signale comme conflits "reduce/reduce" ou "shift/reduce" (voir section 3.4.3.2.3.). Les règles d'application dans YACC pour trancher ces conflits restent valables pour le CSG.

Des priorités explicites peuvent être déclarées pour trancher certains conflits autrement qu'en acceptant l'option par défaut.

Les déclarations de priorité sont de trois types :

left token-or-phylum$_1$, ..., token-or-phylum$_n$;
rigth token-or-phylum$_1$, ..., token-or-phylum$_n$;
non assoc token-or-phylum$_1$, ..., token-or-phylum$_n$;

où "token-or-phylum" est soit une constante de type "CHAR", soit le nom d'un phylum auquel est associé au moins une déclaration de lexème.

"Left" définit une associativité à gauche, et correspond à forcer l'analyseur à réduire; "rigth" définit une associativité à droite et signale à l'analyseur qu'il doit empiler l'état suivant; "non assoc" conduit l'analyseur à signaler une erreur si l'on tente d'associer des occurrences de "token-or-phylum".

Tous les "token-or-phylum" d'une même déclaration ont la même priorité. A chaque nouvelle déclaration, la priorité augmente.

Il faut déclarer explicitement qu'on va donner une certaine priorité à une production de la syntaxe concrète d'entrée, ce qui se fait comme suit :

phylum() ::= operator (tokens-and-phylum-list **prec** token-or-phylum) ;

Le niveau de priorité de "token-or-phylum" est alors assigné à la production. L'énoncé complet des règles d'assignation de priorités figure dans [REPS 87 pp 68-69].

3.4.3.4. Syntaxe de visualisation de l'arbre abstrait.

La syntaxe concrète selon laquelle les termes d'un phylum apparaissent à l'écran est définie par des schémas de décompilation associés à chaque production.

Un schéma de décompilation ressemble à une production : le symbole ":" ou ":: =" sépare un membre de gauche d'un membre de droite.

La déclaration d'un schéma de décompilation se présente comme suit :

```
phylum  : operator  [ left-side : rigth-side ] ;
phylum :: = operator  [ left-side : rigth-side ] ;
```

Un gîte est un terme dont une représentation apparaît à l'écran et qu'il est possible de sélectionner.

Un schéma de décompilation contient des indicateurs de gîtes, notés "@", "^" ou "..", qui correspondent aux occurrences des phyla figurant dans la production dont le schéma définit la représentation visuelle.

Il y a autant d'indicateurs de gîtes dans le membre de droite "rigth-side" du schéma de décompilation d'une production que de phyla dans le membre de droite de cette production. Indicateurs de gîtes et phyla se correspondent dans l'ordre de leur déclaration.

Le membre de gauche "left-side" est soit le symbole "@", soit le symbole "^".

"@" dénote un terme à visualiser et un gîte.
".." indique que la visualisation du sous-terme correspondant est supprimée; aucun nœud de ce sous-terme n'est un gîte.
"^" indique que le sous-terme correspondant est à visualiser et qu'il ne sera un gîte que si le membre de droite du schéma est "@".

Remarquons que ces considérations sur les gîtes ne prennent toute leur importance que lorsqu'on réalise un éditeur syntaxique*.

Dans le membre de droite du schéma de décompilation peuvent apparaître des éléments de décompilation (unparsing items), par exemple des chaînes de caractères entre guillemets et des occurrences de phyla de la production à décompiler (voir [REPS 87] pp52 et suivantes pour une description complète).

Des commandes de formattage permettent enfin de présenter le texte sous une forme acceptable.

Exemple.

Les représentations visuelles associées aux productions définissant le phylum "blocoper" sont définies par les schémas de décompilation suivants :

```
blocoper      : BlocOperVide
                    [ @ :: = "" ]
              | BlocOper
                    [ @ :: = "SPECIFICATION OF OPERATION" @
                             "%n FORMAL DESCRIPTION : %t %n" @
                             "%b %n OUTLINE : %t %n" @
                             "%b %n LEXICON : %t %n" @
                             "%b %n END" @
                             ". %n"          ]
```

* En fait, les schémas de décompilation définissent plus qu'une représentation visuelle. Ils déterminent également les composants d'un terme que l'on peut sélectionner (les gîtes). Ils définissent enfin le mode d'édition par défaut de ces composants.

Il n'est possible d'éditer en mode texte un terme du phylum correspondant au membre de gauche d'un schéma de décompilation que si une "déclaration d'entrée" (voir section 3.4.3.2.1.) porte sur ce phylum. Dans le cas où cette condition est satisfaite, si le symbole ":: =" est utilisé dans la déclaration du schéma de décompilation, l'édition textuelle est toujours directement permise; si par contre, le symbole ":" est utilisé, cela signifie que la production est habituellement traitée comme une unité structurelle qu'il n'est possible de considérer textuellement qu'après une commande explicite ("text-capture").

La représentation visuelle d'un bloc vide de description d'opération est une chaîne de caractères vide. La représentation visuelle d'un bloc non vide de description d'opération est conforme à la syntaxe RSL définie au chapitre 2 section 2.6. par la règle

< bloc-oper > :: = **SPECIFICATION OF OPERATION** < nom-maj >

FORMAL DESCRIPTION : < dec-form-op >
OUTLINE : < profil-op >
LEXICON : < lexicon-op >
END < nom-maj > .

"%t" et "%b" sont des commandes de tabulation; "%n" marque le passage à la ligne.

3.4.4. Transformations d'édition structurelle.

Chaque éditeur généré obéit à des commandes d'édition pré-définies. Il est en plus possible de déclarer des transformations pour spécifier d'autres commandes d'édition permettant de restructurer un objet sélectionné. L'application d'une transformation a pour effet de remplacer un terme par un terme du même phylum. Par exemple, des transformations sont utilisées pour définir l'insertion de nouveaux gabarits. Selon la structure de l'objet sélectionné, seules certaines transformations sont permises. Ces transformations (à ne pas confondre avec les transformations décrites aux chapitres 4 et 5) n'interviennent que dans la réalisation d'un éditeur syntaxique structurel. Nous mentionnons leur existence pour compléter notre description. Nous ne les détaillerons pas car nous ne les avons pas utilisées.

3.4.5. Conclusion.

Le Cornell Synthesizer Generator est un outil qui permet de générer un éditeur syntaxique textuel et/ou structurel à partir d'une spécification décrivant le langage à éditer et de manipuler des arbres syntaxiques attribués. Une spécification de langage comprend la syntaxe abstraite exprimée dans des productions et des déclarations de lexèmes et la sémantique exprimée par des attributs et des équations sémantiques. La spécification du langage à éditer doit aussi décrire dans des règles de grammaire la syntaxe concrète d'entrée et exprimer sous forme de schémas de décompilation la syntaxe concrète de sortie. Elle doit décrire enfin, sous forme de transformations, les commandes d'édition permises en mode structurel.

Si l'on spécifie un éditeur uniquement structurel, on peut se passer de définir une syntaxe concrète d'entrée et ses déclarations associées. Si l'on ne désire qu'un éditeur fonctionnant en mode texte, il n'est pas nécessaire de spécifier de transformations.

3.5. Fonctionnement du CSG.

Le CSG est un outil fonctionnant sous UNIX, qui a recours aux services de certains outils standards disponibles dans cet environment, par exemple LEX et YACC (voir section 3.4.3.2.).

Il est commode de se représenter le CSG sous forme de deux composants [REPS 89] :

1) le générateur proprement dit, qui génère les éditeurs instanciés à partir des spécifications d'éditeurs écrites en SSL, et
2) un noyau d'édition couplé à un évaluateur d'attributs.

Le générateur est composé de deux modules :

- un compilateur acceptant une spécification SSL comme entrée, et produisant :
  - différentes tables contenant notamment les propriétés des phyla, des occurrences de phyla, des opérateurs, des occurrences d'opérateurs, des attributs, et des occurrences d'attributs;
  - un fichier contenant le code des expressions SSL à évaluer;
  - différents fichiers contenant notamment les codes sources à donner à LEX et YACC pour la production respectivement de l'analyseur lexical et de l'analyseur syntaxique;
- un programme shell, nommé sgen, qui coordonne l'activité du compilateur avec celles des autres outils UNIX utilisés dans la génération d'un éditeur.

Le compilateur SSL effectue également différents tests et certaines optimisations de code. Par exemple, il vérifie la cohérence des déclarations de types; il génère un code optimisant l'évaluation des expressions booléennes.

Le noyau d'édition et l'évaluateur d'attributs sont constitués de quatre modules:

- un module de gestion d'arbres attribués contenant entre autres l'implémentation de l'algorithme permettant la mise à jour des valeurs d'attributs de sorte que celles-ci soient toujours cohérentes;

- un interprêteur d'expressions SSL qui calcule les valeurs d'expressions lors de l'application de transformations d'édition et lors de la mise à jour des valeurs d'attributs;

- un module d'édition acceptant les commandes introduites au clavier ou éventuellement avec une souris, et gérant en conséquence les tampons contenant les objets manipulés par l'éditeur;

- un module d'affichage qui gère la présentation sur écran du texte édité, de la liste des commandes permises, de l'écho des commandes introduites, éventuellement des messages d'erreurs.

3.6. Choix du CSG.

Avouons de suite les raisons inavouables pour lesquelles nous avons choisi le Cornell Synthesizer Generator :

- disponibilité immédiate de l'outil et une certaine familiarisation acquise lors d'un travail pratique alors que l'alternative (une implémentation en CAML [QUERE 88]) n'offrait pas ces avantages, et

- l'espoir (en partie déçu) de ne pas avoir à passer de temps à maîtriser LEX et YACC.

En dehors de ces considérations fort peu scientifiques, il s'est avéré que le CSG présente de réels intérêts.

Avec un formalisme déclaratif, le CSG gère une structure d'arbre abstrait et calcule automatiquement les valeurs des attributs déclarés pour lesquels une équation sémantique a été spécifiée. Ces caractéristiques font que pour un minimum d'investissement intellectuel (maîtrise des concepts), il est possible de programmer à un niveau élevé d'abstraction. Cette puissance de l'outil permet de soulager le programmeur de maints détails techniques, ce qui lui laisse davantage l'occasion de se concentrer sur son problème. Celui-ci se reformule en d'autres termes : comment bien définir la syntaxe abstraite, trouver les bons attributs, etc.

Le CSG permet de disposer de la structure de listes tout en en ignorant la représentation. Son langage fonctionnel est également très intéressant : avec un minimum de formalisme, il est possible de faire beaucoup, notamment parce qu'il permet la récursivité. Grâce à ce mécanisme, les procédures sont plus courtes et plus facilement vérifiables.

Outre qu'il permet de résoudre le problème consistant à transformer une spécification RSL en formule équivalente exprimée en langage du premier ordre dans un premier temps, et dans un second en programme PROLOG (problème initial), le CSG offre en plus, et pour un faible coût additionnel - son usage premier est, ne l'oublions pas, la génération d'éditeurs syntaxiques - la possibilité de générer un éditeur syntaxique pour RSL.

Ces avantages nous paraissent compenser largement l'effort de l'apprentissage nécessaire pour maîtriser l'outil.

Chapitre 4

Première étape de la transformation : passage de RSL à la logique des prédicats du premier ordre

4.1. Introduction.

Nous allons construire un outil de prototypage qui consiste à transformer des spécifications formelles écrites selon le formalisme RSL en procédures exécutables écrites en Prolog. Cette transformation est réalisée de façon automatique ou semi-automatique (automatique pour un sous-ensemble intéressant du langage) par l'application de règles de transformation qui tenteront de conserver la structure initiale de la spécification pour favoriser la traçabilité qui est nécessaire pour faciliter les éventuelles mises au point de la spécification.

La transformation est effectuée en deux phases.

La première phase réécrit la spécification RSL sous forme de relations définies en logique des prédicats du premier ordre. L'avantage de cette étape intermédiaire est de faire apparaître explicitement certaines équations qui peuvent être déduites sans ambiguïté du contexte et qui sont laissées implicites dans la spécification formelle en RSL. Cette étape permet en outre de faciliter la compréhension et la justification de la seconde phase de la transformation. L'étude de cette première phase fait l'objet de ce chapitre.

La deuxième phase transforme les relations définies en logique des prédicats du premier ordre en procédures Prolog, en tenant compte du mécanisme d'exécution de ce langage et du mode d'utilisation souhaité du prototype. Cette phase sera abordée dans le chapitre suivant.

Ce chapitre 4 explique comment, à partir de spécifications écrites en RSL, nous pouvons transformer ces dernières en des formules de la théorie des prédicats du premier ordre. En réalité, seulement une partie des spécifications RSL sera utilisée pour réaliser cette transformation. Toute la partie théorique (les trois premières sections) qui constitue le début de ce chapitre est fortement inspirée de [HABRA88] et [HABRA89].

Le processus, qui en fait n'est qu'une reformulation préservant toute la sémantique de la spécification, revient à transformer une par une les opérations décrites par la "spécification de l'opération" en utilisant essentiellement les parties "description formelle" , "type des objets" et "spécification du type des objets" données dans la spécification RSL. Signalons qu'une déclaration de précondition sera considérée et traitée exactement comme une déclaration d'opération. La partie "explicitation de la structure" dans la "spécification du type" servira à compléter cette transformation. Remarquons que la partie "invariant" liée à certains types ne sera pas prise en considération dans le mécanisme de transformation, du moins dans l'état actuel de ce dernier.

La deuxième section donnera un aperçu des problèmes liés au processus de transformation, et en particulier explicitera le problème attaché aux équations implicites.

La troisième section décrira de façon intuitive le mécanisme de transformation en en soulignant les idées principales.

La quatrième section exposera comment la transformation a été réalisée par l'utilisation des grammaires attribuées et du Cornell Synthesizer Generator.

Un exemple illustrera cette transformation dans la dernière section.

4.2. Problèmes liés à la transformation.

La transformation de l'explicitation d'une opération écrite dans le formalisme RSL vers une formule du langage des prédicats du premier ordre, dorénavant noté FOPL pour First Order Predicate Language, nécessite non seulement la conversion des connecteurs utilisés en RSL (and, or, forall, ...) en leur correspondant FOPL

($\wedge$, $\vee$, $\forall$, ...), mais aussi la mise en évidence des équations implicites et leur intégration dans la formule FOPL générée.

La recherche des équations implicites constitue la difficulté principale de la transformation, et nous verrons qu'elle est la seule à nécessiter des informations sur le type des objets.

4.2.1. Principes de base de la transformation.

Mise à part la considération des équations implicites qui sera traitée au point suivant, on peut dire que la transformation d'une explicitation d'une opération écrite en RSL vers une formule FOPL ne pose pratiquement aucune difficulté.

Rappelons qu'une assertion qui explicite une opération OP dans le langage de spécifications RSL[1] est une équivalence de la forme

expression → explicitation

dans laquelle

- le membre de gauche est une équation de la forme
  res = OP (arg1, ... , argn)
  où OP est l'opération à définir [2] ,

- le membre de droite est un ensemble d'équations portant sur d'autres opérations plus fines OP_i , et liées entre elles par les connecteurs d'opérations RSL and , or , forall et with.

De telles assertions sont à transformer en formules FOPL de la forme

F(expression) <=> F(explicitation)

dans les quelles

[1] Le chapitre 2 donne plus de détails à ce sujet.

[2] Nous utilisons les caractères majuscules pour noter les opérations et les types, les caractères minuscules pour noter les occurences de types.

F(expression) et *F(explicitation)* dénotent respectivement la transformation de *expression* et de *explicitation* en formule logique du premier ordre:

- *F(expression)* est une formule FOPL de la forme Res = op (...),
- *F(explicitation)* est une formule FOPL dans laquelle les **seules** variables libres sont les variables res, arg1, ..., argn apparaissant dans le terme *expression* de gauche.

Le problème est d'écrire l'explicitation et l'expression fournies par RSL en une formule FOPL syntaxiquement correcte. Cette transformation peut pratiquement se réaliser moyennant des règles de réécriture entre les objets de RSL et ceux de FOPL. Explicitons ces règles de transformation syntaxique :

- l'égalité " = " devient le symbole de prédicat binaire " = "

- le symbole d'explicitation " $\rightarrow$ " devient le symbole de prédicat binaire " <=> "

- un symbole d'objet " obj " devient un symbole de variable " Obj "
  (conversion minuscules / majuscules de la première lettre du nom désignant l'objet)

- un symbole de fonction " OP_i " ou de prédicat " PREi " devient un symbole de fonction " op_i " ou de prédicat " préi "
  (conversion majuscules / minuscules du nom des opérations et des prédicats)

- le connecteur " and " devient le connecteur logique " $\wedge$ "

- le connecteur " or " devient le connecteur logique " $\vee$ "

- le connecteur " for all " devient le connecteur logique " $\forall$ "

- le connecteur " with " devient le connecteur logique " $\wedge$ "

- une expression de prédicat " PRE (arg1 , ... , argn) " devient une équation logique " pre (Arg1 , ... , Argn) = true "

- une expression de prédicat " not (PRE (arg1 , ... , argn)) " devient une équation logique " pre (Arg1 , ... , Argn) = false "

Illustrons par un exemple ce procédé systématique de transformation.

Soit l'explicitation de l'opération suivante:

$$res = OP(a, b, c, d, e) \rightarrow res1 = OP1(a, b, d) \text{ and } res2 = OP2(b, c) \text{ or } res3 = OP3(d, b, a)$$

dont la transformation en FOPL donne

$$Res = op(A, B, C, D, E) <=> Res1 = op1(A, B, D) \wedge Res2 = op2(B, C) \vee Res3 = op3(D, B, A)$$

Cette transformation n'est pas complète: parmi les variables apparaissant à droite du symbole " <=> ", il y a des variables libres qui sont autres que celles apparaissant dans le terme de gauche, à savoir Res1, Res2, Res3, Res4. Il faut donc compléter cette transformation en vue de lier ces variables à des variables libres appartenant au membre de gauche. C'est là le problème des équations implicites qui est détaillé au paragraphe suivant.

4.2.2. Les équations implicites.

A partir d'une assertion qui a la forme "expression → explicitation", nous voulons obtenir pour "explicitation" une formule logique du premier ordre dans

laquelle les seules variables libres sont celles apparaissant dans "expression" et uniquement celles-là. Nous appellerons cette contrainte "**contrainte de liberté des variables**".

Cet objectif peut-être atteint de deux façons:

- ou bien on oblige le spécifieur à expliciter l'opération jusqu'à ce que la contrainte de liberté des variables soit satisfaite (ce qui est fort contraignant, reconnaissons-le),

- ou bien on permet au spécifieur de passer sous silence certaines parties de l'explicitation à condition que celles-ci puissent être déduites sans ambiguité du contexte afin de satisfaire la contrainte de liberté des variables. La notion de "non-ambiguité" sera explicitée au paragraphe 4.2.3.2 .

La deuxième alternative semble être pourvue d'un intérêt certain, pour autant qu'elle soit réalisable. En fait, il faut préciser dans quelles conditions une explicitation peut être déduite sans ambiguité du contexte.

L'idée de base sous-jacente est que, dans le langage de spécification RSL, la structure des données est parallèle à la stucture des opérations (ceci est lié au processus de la spécification; voir chapitre 2).

Ainsi, à partir d'objets simples (par exemple, résultats ou arguments d'opérations), il est possible de construire des objets composés (par exemple, résultats ou arguments d'une opération explicitée), tout comme il est possible de construire des objets (par exemple, résultats ou arguments d'opérations) à partir d'objets plus fins (par exemple, résultats ou arguments des opérations plus fines).

Illustrons ceci par les deux exemples suivant:

exemple 1:

explicitation de l'assertion:

res = OP (arg1,.....,argn) →

res1 = OP1 (arg1', ... ,argi')

and

res2 = OP2 (arg1" , ... , argj")

profil de l'opération:

OP : CP [TARG1' , TARG1"] X ... → CP [TRES1 , TRES2] [3]

exemple 2:

explicitation de l'assertion:

res = OP (seq , arg1 , ... , argn) →

forall elem : IN (elem , seq) :

resa = OPA (elem , arg1', ... , argi')

profil de l'opération:

OP : SEQ (TELEM) X ... → SEQ (TRESA)

Telles qu'elles sont données dans les exemples ci-dessus, les spécifications ne satisfont pas la contrainte de liberté des variables.

Voyons comment il y a moyen de satisfaire la contrainte de liberté en exprimant les liens qui existent de façon implicite entre les arguments (respectivement résultats) du membre de droite de l'explicitation et les arguments (respectivement le résultat) du membre de gauche.

Entre autres, les résultats " res1", " res2 " et " resa ", et les arguments " arg1' " et " arg1" " ne sont pas des résultats intermédiaires puisqu'ils n'ont pas été introduits par le connecteur RSL with.

Dans l'exemple 1 ci-dessus, on peut représenter sous forme d'arbre et selon les conventions RSL,

[3] Nous prenons comme convention d'écriture que TX désigne le type associé à la variable x.

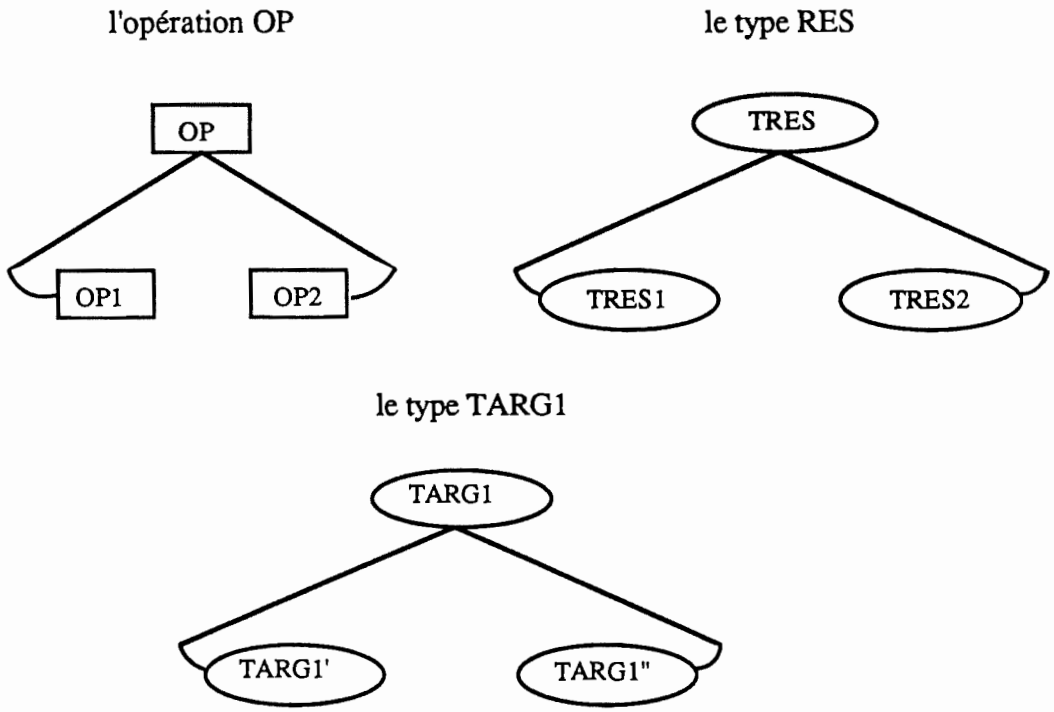


fig. 4.1. Exemple 1.

les deux derniers arbres étant déduits du profil de l'opération.

Cette représentation montre bien le parallélisme qui existe entre la structure de l'opération et celle du type des arguments ou du résultat. De plus on peut en déduire que les résultats " res1" et " res2 " de type respectif TRES1 et TRES2 sont en relation avec le résultat " res " de type TRES dans le membre de gauche par une équation dite implicite, et donc que l'objet " res " est un tuple composé par les objets " res1" et " res2 ". En d'autres mots, la relation implicite peut-être donnée par une équation dite de composition qui s'exprime en fonction de l'opération de construction liée au produit cartésien:

res = CONSTUPLE [TRES1 : res1, TRES2 : res2]

ou par les équations de sélection équivalentes:

res1 = TRES1 (: res) [4]
res2 = TRES2 (: res)

Par un raisonnement analogue, on peut écrire que les objets arg1' et arg1" sont en relation avec l'objet arg1 via une équation de composition implicite de la forme

arg1 = CONSTUPLE [TARG1' : arg1' , TARG1" : arg1"]

ou via les équations de sélection équivalentes

arg1' = TARG1' (: arg1)
arg1" = TARG1" (: arg1)

L'explicitation de l'assertion de l'exemple 1 peut être transformée en une formule FOPL respectant la contrainte de liberté des variables en y intégrant les équations implicites déduites, soit de sélection, soit de composition, ce qui donne les formules équivalentes suivantes:

- <u>en terme de composition</u>:

$$
\begin{aligned}
&Res = op\ (\ Arg1, \ldots, Argn\ ) \\
&\Leftrightarrow\ (\ \exists\ Res1\ )\ (\ \exists\ Res2\ )(\ \exists\ Arg1'\ )(\ \exists\ Arg1"\ ) \\
&\qquad Res1 = op1\ (\ Arg1', \ldots, Argi'\ ) \\
&\quad \wedge\ \ Res2 = op2\ (\ Arg1", \ldots, Argj"\ ) \\
&\quad \wedge\ \ Arg1 = constuple\ [\ Targ1' : Arg1', Targ1" : Arg1"\ ] \\
&\quad \wedge\ \ Res = constuple\ [\ Tres1 : Res1, Tres2 : Res2\ ]
\end{aligned}
$$

[4] l'opération T(:x) extrait du produit cartésien x l'élément ayant le type T

- en terme de sélection:

$$
\begin{aligned}
&Res = op\ (\ Arg1, \ldots, Argn\ ) \\
&\Leftrightarrow\ (\ \exists\ Res1\ )\ (\ \exists\ Res2\ )(\ \exists\ Arg1'\ )(\ \exists\ Arg1''\ ) \\
&\qquad Res1 = op1\ (\ Arg1', \ldots, Argi'\ ) \\
&\wedge\ \ Res2 = op2\ (\ Arg1'', \ldots, Argj''\ ) \\
&\wedge\ \ Arg1' = Targ1'\ (\ :Arg1\ ) \\
&\wedge\ \ Arg1'' = Targ1''\ (\ :Arg1\ ) \\
&\wedge\ \ Res\ 1 = Tres1\ (\ :Res\ ) \\
&\wedge\ \ Res\ 2 = Tres2\ (\ :Res\ )
\end{aligned}
$$

Appliquons le même raisonnement à l'exemple 2.

Représentons sous forme d'arbre et selon les conventions RSL,

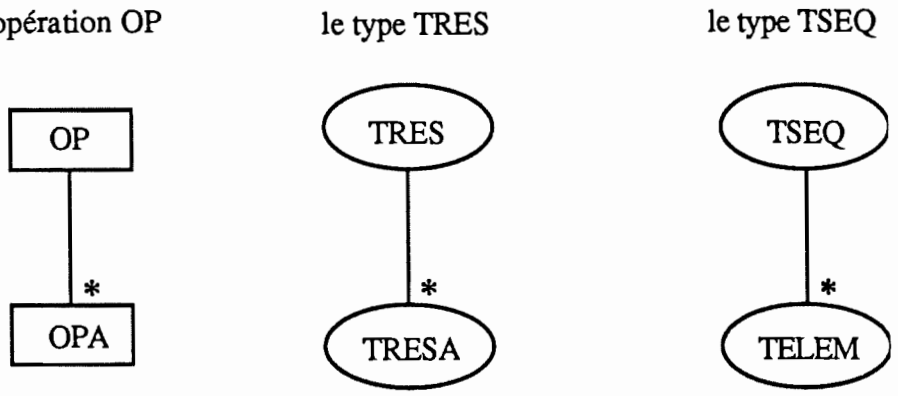


fig. 4.2. Exemple 2.

les deux derniers arbres étant déduits du profil de l'opération.

Le parallélisme existant entre la structure de l'opération et celle du type des arguments ou du résultat étant mis en évidence, on peut observer que " resa " est un élément de la séquence " res ".

De plus, puisque l'explicitation de l'opération est du type forall , la décomposition est réalisée selon la séquence " seq " dont l'élément représentatif est " elem ". En outre, puisque la séquence " res " est décomposée dans la même opération, elle le sera parallèlement à la séquence " seq ". Par conséquent,

l'équation implicite qui lie " resa " à " res " doit exprimer le fait que la position de "resa " dans " res " est la même que celle de " elem " dans " seq ". En d'autres mots, la relation implicite peut être exprimée par les équations de décomposition liées à la séquence:

resa = ITH (k , res)

et elem = ITH (k , seq)

où k indique la position de l'élément " resa " (respectivement " elem ") dans la séquence " res " (respectivement " seq ").

Remarquons en passant qu'il n'est pas possible d'exprimer ces relations implicites en utilisant des équations dites de composition développées en fonction de l'opération de construction dans le cas de la séquence. Pour ce faire, il faut disposer en effet du concept de construction récursive d'une séquence, qui n'est pas un concept admis dans FOPL.

L'explicitation de l'assertion de l'exemple 2 peut être transformée en une formule FOPL respectant la contrainte de liberté des variables en y intégrant les équations implicites déduites:

$$
\begin{aligned}
&Res = op\ (\ Seq\ ,\ Arg1,\ \ldots\ ,\ Argn\ ) \\
&<=>\ (\ \forall Elem\ ) \\
&\quad (\ in\ (\ Elem\ ,\ Seq\ ) = true \\
&\qquad => \ (\ \exists Resa\ )(\ \exists K\ ) \\
&\qquad\quad Resa = opa\ (\ Elem\ ,\ Arg1',\ \ldots\ ,\ Argi'\ ) \\
&\qquad \wedge\ Resa = ith\ (\ K\ \ ,\ Res\ ) \\
&\qquad \wedge\ Elem = ith\ (\ K\ ,\ Seq\ )\ )
\end{aligned}
$$

Cependant , cette formulation ne garantit pas la correspondance "un-à-un" entre la séquence " seq " et la séquence " res ": plusieurs valeurs du résultat " res " peuvent correspondre à la même valeur des arguments " seq ", " arg1' ", ..., ce qui ne donne pas une description complète de la fonction OP.

C'est pourquoi, en vue de compléter la description de la fonction OP, sera ajoutée une relation implicite supplémentaire portant sur la longueur des séquences concernées par l'explicitation forall. Ceci est exprimé par l'équation implicite

LENGTH (seq) = LENGTH (res)

Finalement, à l'assertion de l'exemple 2 correspond la formule FOPL suivante:

$$
\begin{aligned}
& Res = op\ (\ Seq\ ,\ Arg1,\ \ldots\ ,\ Argn\ ) \\
& <=>\ (\ \forall Elem\ ) \\
& \qquad (\ in\ (\ Elem\ ,\ Seq\ ) = true \\
& \qquad\quad =>\ (\ \exists Resa)\ (\ \exists K) \\
& \qquad\qquad Resa = opa\ (\ Elem\ ,\ Arg1'\ ,\ \ldots\ ,\ Argi'\ ) \\
& \qquad\quad \wedge\ Resa = ith\ (\ K, Res\ ) \\
& \qquad\quad \wedge\ Elem = ith\ (\ K, Seq\ ) \\
& \qquad) \\
& \wedge\ \ length\ (\ Seq\ ) = length\ (\ Res\ )
\end{aligned}
$$

Cette façon de compléter une assertion d'explicitation du type forall peut sembler altérer la traçabilité désirée entre les formules générées et la spécification initiale. Rassurons le lecteur attentif en signalant que ces équations implicites, nécessaires pour une formulation FOPL complète, vont disparaître dans la version finale du prototype, c'est-à-dire dans la deuxième phase de la transformation, comme nous le verrons dans le chapitre suivant.

En résumé, deux types d'équations implicites ont été envisagés:

- les équations ajoutées en vue de lier les objets à leur composants. Ces équations sont toujours exprimées en terme d'équations de décomposition ou d'équations de sélection.

- les équations de longueur ajoutées pour compléter les assertions forall en vue de garantir la propriété de fonction à l'opération concernée.

4.2.3. Quelques définitions et conventions d'écriture.

Le but de cette section est de définir exactement les concepts qui seront utilisés dans la suite, entre autres la notion de non-ambiguité, et ce après avoir pris quelques conventions d'écriture qui nous permettront d'alléger les notations.

4.2.3.1. Conventions de notation.

Dans la suite, nous utiliserons les notations qui suivent:

- Dans une assertion d'explicitation d'opération écrite en RSL, l'équation de gauche sera notée " g-eq " et celles de droite " d-eqs " puisqu'elle est en fait la composition de plusieurs équations.
  En RSL, nous avons donc **g-eq → d-eqs.**

- Lors du processus de transformation, nous considérerons " d-eqs " comme un arbre syntaxique abstrait. Nous noterons les différents sous-arbres de " d-eqs " par " d-eqs-a ", " d-eqs-b ", ...

- Les feuilles de l'arbre " d-eqs " sont les sous-arbres qui sont des expressions terminales de la forme " res = OP (arg1, ...) " et nous les noterons " d-eq-a ", " d-eq-b ", ...

- Les formules FOPL correspondant aux sous-arbres RSL " d-eqs ", " d-eqs-a ", ... seront notées respectivement " F (d-eqs) ", " F (d-eqs-a) ", ...

– Dans l'assertion RSL " g-eq → d-eqs ",
  " vg " désigne les variables du membre de gauche " g-eq ";
  " vd" désigne les variables du membre de droite " d-eqs ";
  " vda ", " vdb ", ... désignent les variables des sous-arbres de droite " d-eqs-a ", " d-eqs-b ", ... respectivement.

- Dans une formule FOPL, on représentera la répétition multiple de formes similaires par [.....]* , ainsi

[∃ Vd]* représente une quantification multiple de variables du membre de droite, c'est-à-dire quelque chose de la forme

" ∃ Vr1 ∃ Vr2 ∃ Vr3 ... "

[d-eq]* représente la conjonction de plusieurs équations simples du membre de droite, c'est-à-dire quelque chose comme

" d-eq1 ∧ d-eq2 ∧ d-eq3 ∧ ... "

[{ t / Vd }]* représente la substitution multiple de termes ayant la même forme, c'est-à-dire quelque chose comme

" { t1 / Vd1, t2 / Vd2, t3 / Vd3, ...} " [5]

- Un opérateur " SEL " est un nom générique utilisé pour désigner une quelconque opération de sélection comme

  sélectionner un élément ayant le type T d'un tuple x " $T\ (:x)$ "

  projeter un objet x sur le type T " $AS_T\ (x)$ "

  sélectionner le ième élément d'une séquence s " $ITH\ (i, s)$ "

- L'opérateur " CSEL " est un nom générique utilisé pour désigner des opérations de sélection réalisées à partir d'une composition quelconque d'opérations de sélection simple du type " SEL ".

4.2.3.2. Définitions.

Définition 4.1.

Nous dirons que Ti est une **sous-structure** du type T si et seulement si on a une des relations suivantes:

i - La structure du type T est un produit cartésien incluant la structure du type Ti parmi ses composantes; cette relation est notée

is-cp-of (T , Ti) .

ii - La structure du type T est une union disjointe incluant la structure du type Ti parmi ses composantes; cette relation est notée

is-union-of (T , Ti) .

[5] La notation " t1/vd1 " signifie que l'objet vd1 est remplacé par l'objet t1.

iii - La structure du type T est une séquence de la structure du type Ti; cette relation est notée

is-seq-of (T , Ti) .

iv - La structure du type T est la même que la structure du type Ti; cette relation est notée

is-same-st (T , Ti) .

Définition 4.2.

A chaque connecteur d'opérations en RSL and, or et forall correspond une relation entre les types de la façon suivante:

" is-cp-of "	est la relation pendante du connecteur	and
" is-union-of "	est la relation pendante du connecteur	or
" is-seq-of "	est la relation pendante du connecteur	forall

Définition 4.3.

L'assertion d'explicitation " g-eq → d-eqs " est une **assertion non-ambigüe** si et seulement si nous avons que chaque argument ou résultat (que nous noterons vd de type TPD) apparaissant dans le membre de droite " d-eqs " vérifie une des propriétés suivantes

a- La variable vd est un objet intermédiaire introduit par un connecteur with, et donc " d-eqs " a la forme

d-eq-a
with vd = OPINTER (...).

b- La variable vd est un élément représentatif introduit par un connecteur forall, et donc " d-eqs " a la forme

forall vd : IN (vd , seq) : d-eq-a.

c- La variable vd n'est ni un élément représentatif ni un objet intermédiaire, et une des alternatives suivantes est de vigueur:

c1- la variable vd est une des variables de l'équation du membre gauche " g-eq ", autrement dit vd sera une des variables libres de la formule du premier ordre F (d-eqs),

c2- il y a une et une seule variable de l'équation du membre de gauche " g-eq ", disons vg de type TPG, telle que

*) TPD est une sous-structure de TPG (au sens de la définition 4.1.)

*) la relation de structuration liant les types TPG et TPD correspond au connecteur d'opérations utilisé (selon la définition 4.2.).

Dans le cas (c2), les variables vd et vg seront liées par une équation implicite.

Les définitions et les notations ayant été précisées, les problèmes liés à la transformation ayant été soulevés et décrits, le processus de transformation d'une assertion non-ambigue peut être abordé.

4.3. Description intuitive du mécanisme de transformation.

Le mécanisme de transformation se compose essentiellement de deux étapes: la déduction des équations implicites et la transcription dans le langage FOPL.

La détection des équations implicites, reliant des objets appartenant à l'équation du membre de gauche dans l'explicitation d'une opération à leur correspondant dans les équations du membre de droite, découle de la correspondance entre la structure des opérations et la structure des types concernés.

Si la déduction des équations implicites est relativement simple dans le cas où au plus un opérateur de connexion est toléré dans le membre de droite de l'explicitation (nous parlerons alors d'assertion simple), celle-ci devient nettement plus complexe dans le cas où un nombre quelconque de connecteurs et de

combinaisons de connecteurs sont imbriqués (nous parlerons alors d'assertion composée).

Une fois les équations implicites déduites, l'adaptation dans le langage FOPL ne pose pratiquement plus aucun problème.

Dans cette section, nous allons d'abord présenter l'idée intuitive de la transformation (déduction des équations implicites et adaptation en syntaxe FOPL) pour le cas d'une assertion simple, avant d'expliquer comment procéder dans le cas composé. La transformation dans ce dernier cas sera vue comme GENERALISATION de la transformation du cas des assertions simples.

4.3.1. Transformation d'assertions simples.

Le but de cette section est de décrire la transformation d'une assertion de la forme " g-eq → d-eqs " en une formule logique du premier ordre " F (g-eq) <=> F (d-eqs) ", dans le cas où le terme " d-eqs " a au plus un connecteur entre les opérations.

Nous appellerons **assertion simple** une assertion dans laquelle l'explicitation est soit une équation terminale, soit une composition à un seul niveau d'équations terminales liées entre elles par un seul des connecteurs and, or ou forall, c'est-à-dire quelque chose de la forme

" d-eq-a "
ou " d-eq-a and d-eq-b "
ou " d-eq-a or d-eq-b "
ou " forall x : IN (x , seq) : d-eq-a "

Les autres formes n'impliquent pas d'équations implicites.

Voyons dans ce cas comment déduire les équations implicites et transcrire la spécification complétée en langage des prédicats du premier ordre.

4.3.1.1. Déduction des équations implicites.

Supposons qu'une variable vd[6] de type TPD qui apparaît dans le membre de droite " d-eqs " de l'assertion d'explicitation ne soit ni un objet intermédiaire introduit par un connecteur with ni un élément représentatif introduit par un connecteur forall, alors deux cas peuvent encore se présenter (cas (c) de la définition 4.3.)

ou bien " vd " est une des variables du membre gauche " g-eq " de l'équation,

ou bien il y a une et une seule variable dans " g-eq ", disons vg de type TPG, telle que les variables vd et vg soient liées par une équation impicite. Cette équation est donnée par la règle suivante:

Règle R1[7]

(a) si les stuctures des types sont liées par " is-cp-of (TPG , TPD) " et si l'opération est affinée par l'utilisation du connecteur and alors l'équation implicite est

$$vd = TPD (: vg) .$$

(b) si les stuctures des types sont liées par " is-union-of (TPG , TPD) " et si l'opération est affinée par l'utilisation du connecteur or, alors l'équation implicite est

$$vd = AS_{TPD} (: vg) .$$

(c) si les stuctures des types sont liées par " is-seq-of (TPG , TPD) " et si l'opération est affinée par l'utilisation du connecteur forall, alors l'équation implicite est

$$vd = ITH (k , vg) ,$$

où k est la position de l'élément représentatif dans la séquence. Nous avons aussi l'équation implicite

$$elem = ITH (k , seq) ,$$

en supposant que forall ait la forme forall elem : IN (elem , seq) : ...

---

6 Par variable, nous parlons indistinctement d'un argument ou d'un résultat.

7 Déduite de la définition 4.1. et de la définition 4.2.

4.3.1.2. Adaptation au langage FOPL.

Une fois réalisée la déduction des équations implicites correspondant à chaque variable vérifiant le cas (c2) de la définition 4.3., les règles suivantes sont utilisées pour transformer l'expression " d-eqs " en la formule correspondante " F (d-eqs)" écrite dans la syntaxe FOPL[8].

Règle R2.

(a) Si " d-eqs " est une équation terminale de la forme " res = OP (arg1 , ... , argn) " alors la formule F (d-eqs) correspondante est donnée par l'équation atomique FOPL

$$Res = op\ (\ Arg1, \ldots, Argn\ )\ .$$

(b) Si " d-eqs " est une opération terminale de la forme " PRE (arg1, ... , argn) ", alors la formule F (d-eqs) correspondante est donnée par

$$pre\ (\ Arg1, \ldots, Argn\ ) = true\ .$$

(c) Si " d-eqs " est une opération terminale de la forme " not (PRE (arg1, ..., argn)", alors la formule F (d-eqs) correspondante est donnée par

$$pre\ (\ Arg1, \ldots, Argn\ ) = false\ .$$

(d) Si " d-eqs " a la forme " d-eq-a with inter = OPINTER (arg1, ... , argn)", alors la formule F (d-eqs) correspondante est donnée par

$$(\ \exists Inter\ )\ F\ (\ d\text{-}eq\text{-}a\ ) \wedge Inter = opinter\ (\ Arg1, \ldots, Argn\ )\ .$$

(e) Si " d-eqs " a la forme " d-eq-a and d-eq-b ", alors la formule F (d-eqs) correspondante est donnée par

$$[\ \exists\ Vda\ ]^*\ \ [\ Vda = TPDA\ (\ :Vg\ )\ ]^*\ \wedge\ F\ (\ d\text{-}eq\text{-}a\ )$$
$$\wedge\ [\ \exists\ Vdb\ ]^*\ \ [\ Vdb = TPDB\ (\ :Vg\ )\ ]^*\ \wedge\ F\ (\ d\text{-}eq\text{-}b\ )$$

où Vda (respectivement Vdb) représente les variables de " d-eq-a " (respectivement de " d-eq-b ") qui ont été liées aux variables Vg correspondantes dans " g-eq " en utilisant la règle R1.

---

[8] Voir les principes de base de la transformation au point 4.2.1. de ce chapitre.

(f) Si " d-eqs " a la forme " d-eq-a or d-eq-b ", alors la formule F (d-eqs) correspondante est donnée par

$$[\exists\ Vda\ ]^*\ \ [\ Vda = as_{TPDA}\ (\ Vg\ )\ ]^*\ \wedge\ F\ (\ d\text{-}eq\text{-}a\ )$$
$$\vee\ \ [\ \exists\ Vdb\ ]^*\ \ [\ Vdb = as_{TPDB}\ (\ Vg\ )\ ]^*\ \wedge\ F\ (\ d\text{-}eq\text{-}b\ )$$

où Vda (respectivement Vdb) représente les variables de " d-eq-a " (respectivement de " d-eq-b ") qui ont été liées aux variables Vg correspondantes dans " g-eq " en utilisant la règle R1.

(g) Si " d-eqs " a la forme " forall elem : IN (elem , seq) : d-eq-a ", alors la formule F (d-eqs) correspondante est donnée par

$$\forall\ Elem\ (\ in\ (\ Elem\ ,\ Seq\ ) = true\ \Rightarrow\ \exists\ K\ [\ \exists Vda\ ]^*$$
$$Elem = ith\ (\ K\ ,\ Seq\ )$$
$$\wedge\ \ [\ \ Vda = ith\ (\ K\ ,\ Vg\ )\ ]^*$$
$$\wedge\ F\ (\ d\text{-}eq\text{-}a\ )\ )$$
$$\wedge\ [\ length\ (\ Seq\ ) = length\ (\ Vg\ )\ ]^*$$

où Vda représente les variables de " d-eq-a " qui ont été liées aux variables Vg correspondantes dans " g-eq " en utilisant la règle R1.

(h) Si " d-eqs " a la forme " d-eq-a if PRED (arg1 , ... , argn) ", alors la formule F (d-eqs) correspondante est donnée par

$$pred\ (\ Arg1\ ,\ \ldots\ ,\ Argn\ ) = true\ \wedge\ F\ (\ d\text{-}eq\text{-}a\ )$$

(i) Si " d-eqs " a la forme " d-eq-a if PRED (arg1 , ... , argn) d-eq-b otherwise ", alors la formule F (d-eqs) correspondante est donnée par

$$pred\ (\ Arg1\ ,\ \ldots\ ,\ Argn\ ) = true\ \wedge\ F\ (\ d\text{-}eq\text{-}a\ )$$
$$\vee\ pred\ (\ Arg1\ ,\ \ldots\ ,\ Argn\ ) = false\ \wedge\ F\ (\ d\text{-}eq\text{-}b\ )$$

4.3.2. Transformations des assertions composées.

4.3.2.1. Les assertions composées.

Dans le cas des assertions simples, nous nous étions limités aux assertions d'explicitation d'opérations qui contiennent au plus un opérateur de connexion dans la partie explicitation de l'opération.

La correspondance entre les étapes de décomposition des types d'objets et les étapes qui affinent une opération n'est pas toujours une correspondance pas à pas.

Une assertion d'explicitation d'opération peut englober plus d'une étape de raffinement. L'exemple ci-dessous montre une combinaison de connecteurs and et or dans l'explicitation d'une même opération.

assertion : *res = OP (arga , argb , ...)*

→

res1 = OP1 (arga1 , argb1 , ...)

and

res2 = OP2 (arga2 , argb1 , ...)

or

res3 = OP3 (arga1 , argb2 , ...)

and

res4 = OP4 (arga2 , argb2 , ...)

profil : *CP [TARGA1 , TARGA2] X UNION [TARGB1 , TARGB2] X ...*

→ *UNION [CP [TRES1 , TRES2] , CP [TRES3 , TRES4]]*

Dans ce cas également, quelques objets apparaissant dans l'équation du membre de droite sont liés à des objets correspondants apparaissant dans l'équation du membre de gauche. Malheureusement, la décomposition des types d'objets concernés correspond à certaines étapes de raffinement de l'opération, mais pas nécessairement à toutes.

Ainsi, dans l'exemple ci-dessus, la décomposition de TARGA correspond à une étape and lors du raffinement de l'opération OP, la décomposition de TARGB correspond à une étape or, et la décomposition de TPRES correspond aux deux étapes simultanément. Illustrons ceci en représentant la décomposition de l'opération OP et les types des objets par des arbres selon les conventions RSL:

l'opération OP

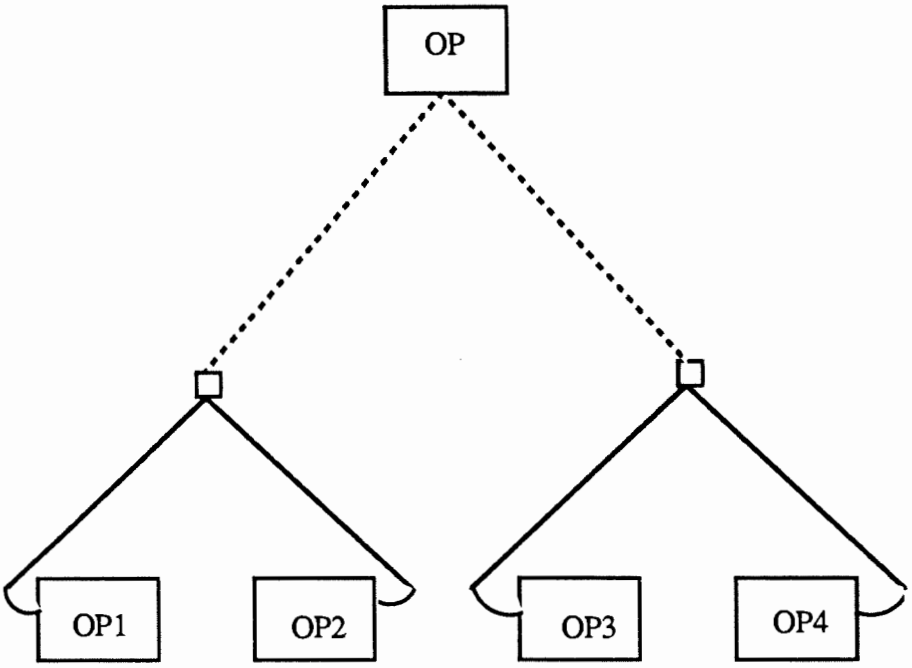


fig. 4.3. Représentation de l'opération " OP ".

le type TRES

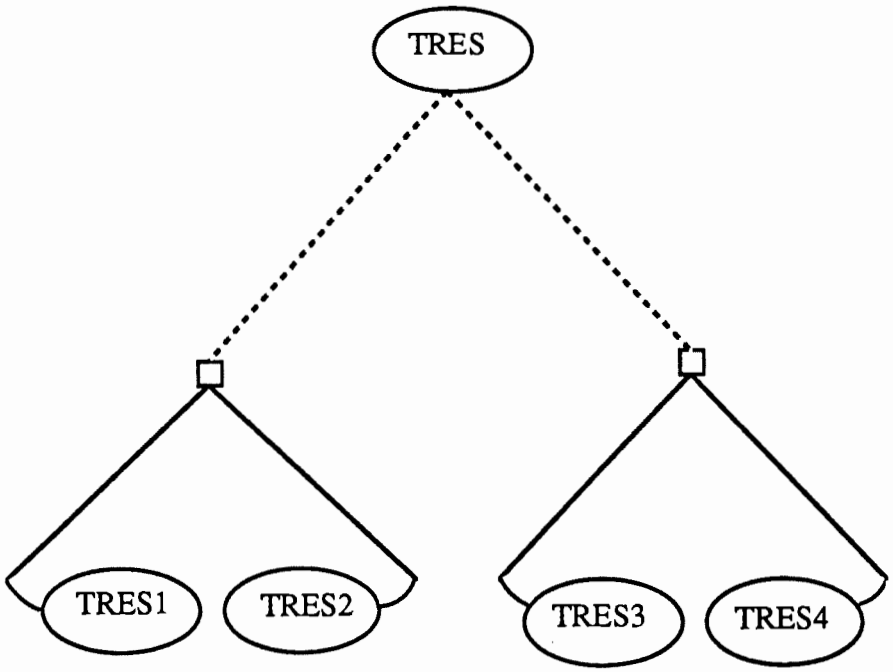


fig. 4.4. Représentation du type " résultat " .

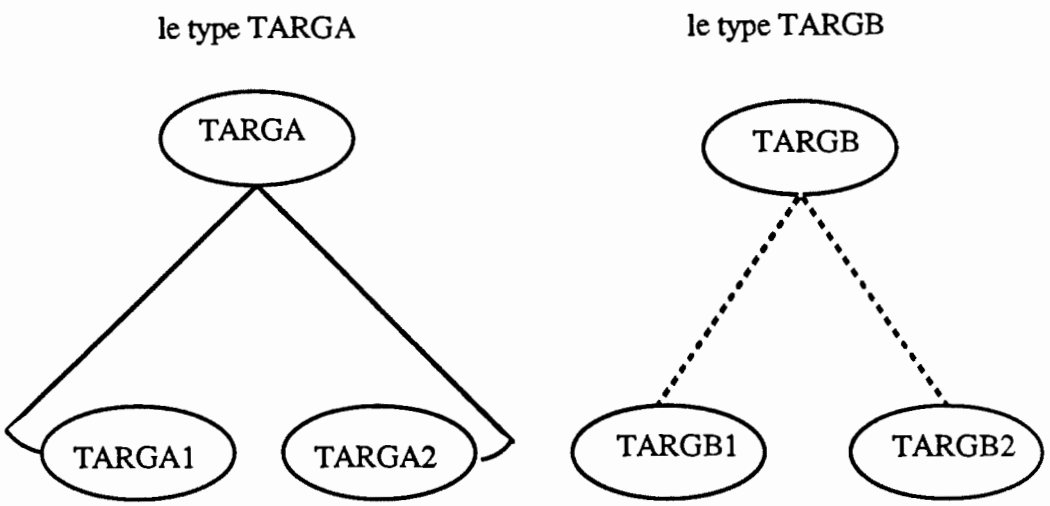


fig. 4.5. Représentation des types " arguments ".

Ainsi, l'arbre de décomposition d'un type peut correspondre à un sous-arbre de l'arbre de la décomposition de l'opération.

A partir de l'assertion d'explicitation de l'opération OP et en combinant avec les structures des types d'objets concernés, on peut déduire les équations implicites suivantes:

$arga1 = TARGA1\ (:arga)$
$arga2 = TARGA2\ (:arga)$
$argb1 = AS_{TARGB1}\ (:argb)$
$argb2 = AS_{TARGB2}\ (:argb)$
$res1 = TPRES1\ (:AS_{CP[TRES1,TRES2]}\ (res))$
$res2 = TPRES2\ (:AS_{CP[TRES1,TRES2]}\ (res))$
$res3 = TPRES3\ (:AS_{CP[TRES3,TRES4]}\ (res))$
$res4 = TPRES4\ (:AS_{CP[TRES3,TRES4]}\ (res))$

Essayons de voir ce qu'il faut faire pour systématiser la génération de telles équations implicites dans le cas d'assertions composées.

4.3.2.2. Nouvelles définitions.

Dans le cas général, la partie " d-eqs " apparaissant dans le membre de droite d'une assertion d'explicitation " g-eq → d-eqs " est formée de la composition d'équations plus fines par un nombre quelconque de connecteurs d'opérations RSL. On peut donc dire que cette partie a une structure multi-niveaux. Si l'on considère une assertion comme étant un arbre de syntaxe abstraite, on peut dire que la partie " d-eqs " est formée d'une structure d'arbres emboîtés dont les feuilles sont des équations terminales de la forme " res = OP (arg1 , ... , argn) " ou de la forme " PRE (arg1 , ... , argn) " ou encore de la forme " not (PRE (arg1 , ... , argn)) " , et dont les noeuds non-terminaux représentent les connecteurs d'opérations RSL and , or , forall , ... Si nous représentons de cette façon l'exemple suivant déjà traité ci-dessus

assertion : *res = OP (arga , argb , ...)*
→
res1 = OP1 (arga1 , argb1 , ...)
and
res2 = OP2 (arga2 , argb1 , ...)
or
res3 = OP3 (arga1 , argb2 , ...)
and
res4 = OP4 (arga2 , argb2 , ...)

profil : *CP [TARGA1 , TARGA2] X UNION [TARGB1 , TARGB2] X ...*
→ UNION [CP [TRES1 , TRES2] , CP [TRES3 , TRES4]]

nous obtenons l'arbre suivant qui correspond à la partie " explicitation " dans le membre de droite de l'assertion :

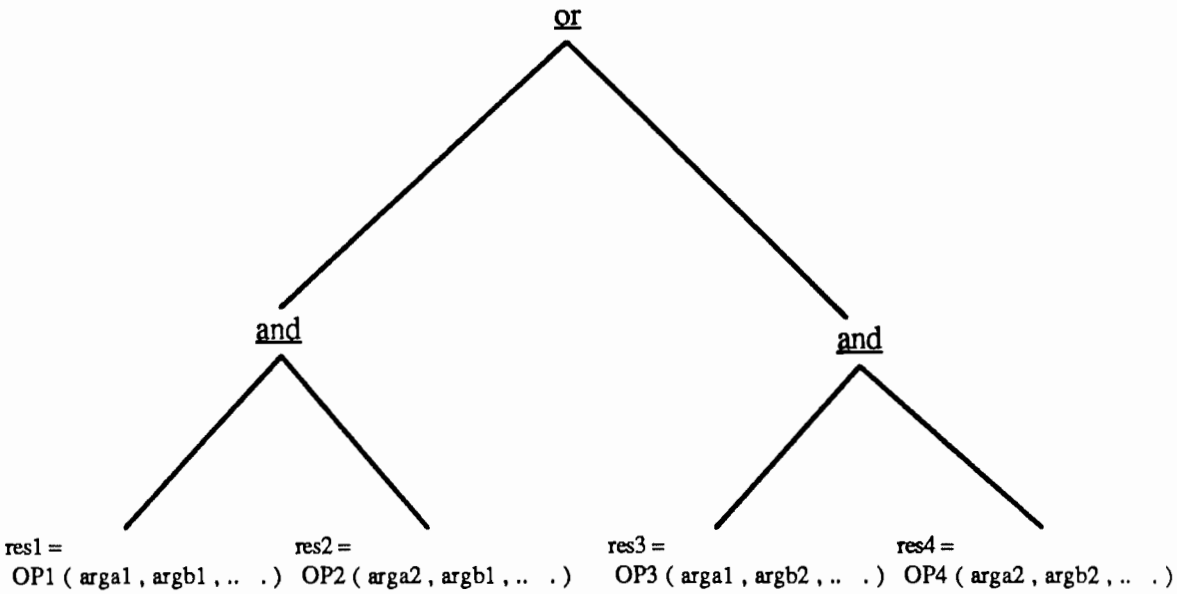


fig. 4.6. " explicitation " mise sous forme d'abre.

Cette représentation illustre bien qu'en fait une explicitation dans le cas composé est constituée de parties de plus en plus fines, qui en réalité correspondent chaque fois à des sous-arbres emboîtés. Chaque sous-arbre de " d-eqs " , disons " d-eqs-x " , conduira à une formule logique du premier ordre, disons " F (d-eqs-x)", dans laquelle certaines variables, disons x1 , x2 , ... , seront libres. Nous appellerons de telles variables xi des **variables composées** du sous-arbre " d-eqs-x ".

Pour transformer les assertions complexes en formules de la logique des prédicats du premier ordre en y incluant les équations implicites, on généralise la transformation décrite dans le cas des assertions simples. Cette généralisation nécessite de généraliser également la définition de non-ambiguité.

Définition 4.3.bis.

Une assertion d'explicitation " g-eq <-> d-eqs " dans le cas général est une **assertion non-ambigüe** si et seulement si nous avons :

I Chaque variable, disons vd de type TPD, apparaissant dans un des sous-arbres de " d-eqs ", disons dans le sous-arbre " d-eqs-x ", vérifie l'une des propriétés suivantes:

a'- La variable vd est un objet intermédiaire introduit par un connecteur with, et le sous-arbre " d-eqs-x " considéré a la forme

d-eqs-a

with vd = OPINTER (...).

b'- La variable vd est un élément représentatif introduit par un connecteur forall, et le sous-arbre " d-eqs-x " considéré a la forme

forall vd : IN (vd , seq) : d-eqs-a.

c'- La variable vd n'est ni un élément représentatif ni un objet intermédiaire; dès lors une des alternatives suivantes est vérifiée:

c1'- la variable vd est une des variables composées du sous-arbre considéré d-eq-x, autrement dit vd sera une des variables libres de la formule du premier ordre F (d-eqs-x),

c2'- la variable vd est une composition de variables d'un sous-arbre plus fin et d'une et une seule variable composée du sous-arbre considéré d-eqs-x, disons vdx de type TDX, telle que

*) TPD est une sous-structure de TDX (au sens de la définition 4.1.), et

*) la relation de structuration liant les types TDX et TPD correspond au connecteur d'opérations utilisé (selon la définition 4.2.).

Dans le cas (c2'-), les variables vd et vdx sont liées par une équation implicite.

III La détermination de l'alternative adéquate (c1' ou c2') est réalisable de façon non-ambigüe en utilisant le fait que les variables composées de l'arbre tout entier " d-eqs" matchent exactement les variables de l'équation " g-eq ".

Voyons maintenant comment on peut, à partir de cette définition de la non-ambiguité, trouver les équations implicites dans le cas des assertions composées.

4.3.2.3. Déduction des équations implicites.

Supposons que " d-eqs-x " soit un sous-arbre intermédiaire de l'arbre " d-eqs " et que vd soit une variable apparaissant dans d-eqs-x telle que cette variable vd ne soit ni un élément représentatif introduit par un forall ni un objet intermédiaire introduit par un with (nous sommes dans le cas c'-).

Nous pouvons générer pour vd une équation implicite en appliquant la règle R1 vue auparavant. Mais, à ce niveau, nous ne sommes pas capables de déterminer directement à laquelle des deux alternatives c1'- ou c2'- nous avons affaires. C'est pourquoi, dans un premier temps, nous allons considérer les deux alternatives, la détermination de l'alternative adéquate ne sera possible que lorsque l'arbre " d-eqs " tout entier aura été considéré.

A partir de c1'- et de c2'- nous savons que F (d-eqs-x) a une variable libre, disons vdx, telle que l'on ait soit vd = vdx, soit vd = SEL (vdx), où SEL représente l'opérateur de sélection correspondant à une application possible de la règle R1. De plus, si " d-eqs-x " est lui-même un sous arbre de " d-eqs-y ", alors vdx est à attacher à une certaine variable composée de F (d-eqs-y), disons vdy, telle que l'on ait soit " vdx = vdy ", soit " vdx = SEL (vdy) ", et ainsi de suite.

Finalement, ceci conduit à une série d'équations de sélection dans laquelle la première équation est attachée à vd et la dernière à une des variables composées de F (d-eqs), disons x. Cette série aura la forme

vd = SEL1 (vdx), vdx = SEL2 (vdy) , ... , ... = SELn (x) ,

ce qui est équivalent à l'équation unique

vd = SEL1 (SEL2 (... (SELn (x)) ...)) **(1)**

dans laquelle chaque SELi est soit l'opérateur de sélection correspondant, soit l'opérateur d'identité.

La conséquence du non-déterminisme de c1'- et c2'- est que plusieurs équations de composition du type (1) seront candidates.

De la propriété III donnée dans la définition d'assertions composées non-ambigues, nous savons que l'équation implicite composée adéquate est celle déterminée par le fait que la formule F (d-eqs) correspondant à l'arbre tout entier " d-eqs " dans l'assertion " g-eq → d-eqs " a exactement les mêmes variables libres que l'équation " g-eq". A partir de celà, on peut conclure qu'il y aura une et une seule variable de " g-eq ", disons vg, telle que ce " vg " peut être mis à la place de la variable " x " dans une et une seule des équations composées candidates de la forme (1). Nous avons ainsi déterminé l'équation implicite liée à la variable vd. Il est bien entendu que ce procédé peut être appliqué à toutes les variables vérifiant la condition (c'-), ce qui conduit à l'explicitation de toutes les équations implicites liées à une assertion d'explicitation composée.

Après la déduction de l'équation implicite correspondant à chaque variable composée vérifiant la condition (c'-), on peut transformer l'assertion d'explicitation en la formule FOPL correspondant. Pour ce faire, il suffit de réutiliser la règle R2 énoncée dans le cas des assertions simples où il est bien entendu qu'une expression du type "d-eqs" ou "d-eq-a" ou "d-eq-b" est à voir comme désignant un sous-arbre, et F (d-eqs) ou F (d-eq-a) ou F (d-eq-b) comme représentant la formule FOPL correspondant respectivement à ces sous-arbres.

Nous sommes ainsi parvenu à transformer une assertion d'explicitation d'opérations écrite en RSL, et de forme tout à fait générale, en une formule FOPL équivalente, en conservant les objets initiaux (à une réécriture près) et en y ajoutant une série d'équations implicites (à condition que l'assertion initiale vérifie la contrainte de non-ambiguité). Nous supposerons cette précondition vérifiée pour pouvoir utiliser notre transformateur. Ceci termine la première phase de la transformation.

Une description de la validation complète de ce processus de transformation et des propriétés qui lui sont attachées dépasse le cadre de ce travail. Le lecteur intéressé trouvera plus de détails à ce sujet dans [HABRA 89].

La section suivante va expliquer comment toute la théorie décrite ci-dessus a été mise en oeuvre pour produire un outil de transformation automatique de spécifications RSL en formules FOPL de la logique du premier ordre.

4.4. Réalisation de la transformation par les grammaires attribuées.

Cette section a pour but d'expliquer comment, à partir de la description théorique de la transformation de spécifications RSL en formules FOPL, nous avons réalisé la mise en oeuvre de cette transformation au moyen des grammaires attribuées en utilisant le Cornell Synthésizer Generator (CSG).

Au point 4.4.1., nous expliquerons la méthodologie suivie pour construire progressivement la solution, et les hypothèses qui ont été imposées dès le départ pour assurer une base correcte au processus de construction.

Après avoir élagué l'arbre abstrait RSL au point 4.4.2., nous verrons en 4.4.3. comment disposer là où il le faut du type complet de tous les objets utilisés.

Le point 4.4.4. mettra en évidence la façon dont on a créé les équations implicites selon le noeud (and, or, forall) d'abord dans le cas des assertions simples, tout en restant, grâce aux attributs, très proche de la théorie.

Le cas des assertions composées sera traité au point 4.4.5.

La construction de la formule FOPL correspondant à la spécification initiale sera décrite en 4.4.6.

Un exemple complet illustrant ce processus de transformation sera inséré dans la section suivante.

4.4.1. Eléments de méthodologie.

4.4.1.1. Quelques rappels.

Comme nous l'avons vu au chapitre 3, l'ensemble des attributs attachés à une production de la grammaire abstraite se divise en deux sous-ensembles disjoints: celui des attributs hérités et celui des attributs synthétisés.

Chaque attribut possède une équation sémantique qui en définit la valeur.

Toutes les équations sémantiques seront décrites dans le formalisme SSL (Synthesizer Specification Language).

Une valeur d'attribut synthétisé d'un objet est calculée à partir des valeurs d'attributs des composants de cet objet.

Une valeur d'attribut hérité est calculée à partir des valeurs d'attributs d'objets dont il est le composant, c'est-à-dire de son contexte.

Donnons-nous également quelques conventions d'écriture:

- un nom de phylum est toujours en lettres minuscules; par exemple: *blocoper*.

- les noms d'opérateurs sont la concaténation de mots qui commencent par une lettre majuscule et continuent en lettres minuscules; par exemple: *BlocOperVide*.

4.4.1.2. Démarche adoptée.

Trouver de "bons" attributs, choisir leur mode de propagation (synthèse ou héritage), leur définir une équation sémantique est un premier problème. "Programmer" tout cela en SSL en constitue un second: la principale difficulté réside dans le changement des "habitudes" procédurales. Tout raisonnement est d'office basé sur la notion de récursivité. Une des façons de mener à bien cette tâche est de se poser, à chaque règle de grammaire, des questions du genre

- " pour définir les valeurs d'attributs synthétisés du noeud courant, quelle contribution apportent les attributs hérités du noeud père et les attributs synthétisés des noeuds fils? "

- " pour définir les valeurs d'attributs hérités par un noeud fils, quelle contribution apporte le noeud courant aux valeurs d'attributs reçues en héritage de son noeud père? "

A tout moment, lors de la construction progressive des attributs et de leur équation sémantique, notre démarche fut basée essentiellement sur cette façon de procéder. Il faut bien reconnaître que cela ne fut pas sans difficultés au départ et

que bien souvent de nombreuses heures de travail furent anéanties en quelques secondes[1] : CSG venait de détecter une circularité dans nos attributs et tout était à refaire!

Il est donc nécessaire de se rendre compte que si la construction progressive de la solution telle qu'elle est décrite ci-après semble couler de source, celle-ci a nécessité pas mal d'heures de réflexion et de travail non reflétées dans le résultat final.

Au début du travail, la question qui s'est posée est celle de savoir quel sous-ensemble de la syntaxe abstraite de RSL[2] sera nécessaire, et surtout quel type d'information pourra en être extrait pour répondre à notre problème de transformation de spécifications.

Dans un deuxième temps, connaissant, de par la théorie exposée au début de ce chapitre, les noeuds qui posent problème, nous nous sommes efforcés d'imaginer une structure des données qui conduirait à résoudre le problème dans le cas élémentaire d'une assertion simple.

Les bases étant ainsi posées, il ne reste plus qu'à se pencher sur le problème des assertions composées et voir comment généraliser les attributs issus de la réflexion sur les assertions simples pour qu'ils produisent un résultat correct dans le cas général des assertions composées.

Finalement, nous expliquerons comment nous pouvons produire l'arbre FOPL au moyen de tous les attributs produits, c'est-à-dire synthétisés ou hérités, lors des étapes précédentes.

4.4.1.3. Hypothèse de départ.

Avant de commencer notre réflexion sur les règles de grammaire nécessaires et sur les attributs à leur associer, nous avons supposé disposer d'un arbre abstrait RSL **correct sur le plan de la sémantique statique**: toute variable utilisée

[1] Nous avions déjà rencontré ce problème au moment de la construction de la grammaire concrète de RSL: la construction d'une grammaire non-ambigüe n'est pas chose évidente, loin s'en faut!

[2] Celle-ci se trouve intégralement donnée en annexe.

est déclarée, tout type est complétement explicité, les seuls éléments manquant vérifient la contrainte de non-ambiguité, ...

La vérification sémantique qui, à partir de spécifications écrites en RSL, produit un arbre abstrait RSL sémantiquement correct, peut être envisagée. L'implémentation de cette dernière fait partie des extensions possibles à notre travail.

Nous disposons donc de l'arbre abstrait RSL (dont la grammaire qui est reprise complètement en annexe correspond à la grammaire concrète détaillée au chapitre 2). Voyons comment la compléter par des attributs en vue de produire son correspondant FOPL.

4.4.2. Elagage de l'arbre abstrait.

En réalité, le langage de spécifications RSL possède des parties utiles pour la génération de prototpes, et des aspects redondants ayant d'autres utilités. Les premières seront détaillées ci-dessous, les secondes découlent soit d'une certaine redondance obligée par le langage de spécifications (comme la réexpression de la signature des opérations dans l'explicitation des types), soit d'explications écrites en langue naturelle (le lexicon en particulier), soit encore d'aspects non-couverts par notre outil (comme la vérification des invariants liés à un type, la vérification de l'exactitude de la signature d'une opération ...).

L'arbre abstrait RSL contient deux branches principales duales:

- la première contient les blocs permettant de définir les opérations (elle correspond à " l'énoncé du problème "). Dans notre syntaxe abstraite, elle est désignée par *suiteblocoper*.

- la seconde contient les blocs permettant de décrire le type des objets (elle correspond à " l'univers du problème "). Dans notre syntaxe abstraite, elle est désignée par *suitebloctypobj*.

Il est clair, au vu de la théorie, que dans ces deux branches certaines parties ne seront pas à considérer pour la transformation.

Essayons donc de voir de quels sous-arbres nous aurons exactement besoin.

Pour une opération donnée (*blocoper* dans notre syntaxe abstraite), il est nécessaire de disposer

*) de l'assertion d'explicitation de l'opération, des arguments, du résultat et des préconditions éventuelles (*decformop* dans notre syntaxe abstraite), puisque c'est la base même de la transformation (rappelons qu'une précondition (*expre* dans notre syntaxe abstraite) sera traitée comme une opération particulière).

*) de la partie du profil de l'opération (*profilop* dans notre syntaxe abstraite) dans laquelle apparaît le type de tout objet utilisé dans la description formelle (*sprofobj* dans notre syntaxe abstraite), puisqu'il s'agira, lors de la création des équations implicites, de considérer les objets dont la structure est parallèle à la structure d'explicitation de l'opération.

*) de l'assertion permettant d'expliciter un type d'objet en fonctions d'autres types, puisque les structures d'objets sont des structures emboîtées. Il faudra donc nécessairement les déplier à un moment donné. Cette information se trouve dans la partie " univers du problème ". Dans le cas de notre syntaxe abstraite, il s'agit du sous-arbre *expltyp* du sous-arbre *decformobj* de l'arbre *bloctypobj*.

A première vue, il s'agit là de toutes les parties de l'arbre abstrait qui seront nécessaires. Nous avons donc pu réaliser un premier élagage qui a enlevé en outre les branches, inutiles dans notre cas, concernant les descriptions en langage naturel des types et des opérations (respectivement les sous-arbres *lexobj* et *lexop* dans notre syntaxe) et celles concernant les invariants éventuels et le profil dans la description des types d'objets (respectivement les sous-arbres *predicat* et *profilobj* dans notre syntaxe abstraite).

Ayant conservé les parties utiles de l'arbre RSL, voyons maintenant comment se servir de ces parties pour construire la transformation en formules du langage des prédicats du premier ordre.

4.4.3. Comment disposer dans tout l'arbre RSL de l'explicitation des types d'objets?

Nous avons vu dans la partie théorique, au début de ce chapitre, que disposer à tout instant du type de tous les objets était crucial. Tout le raisonnement est en effet basé sur cette connaissance des types. Un de nos premiers objectifs fut donc d'avoir à notre disposition, et à quelqu'endroit que ce soit dans l'arbre abstrait RSL, le type de tout objet, et même plus, le type " déplié " de tout objet, c'est-à-dire sa définition complète en termes des types de base. Puisqu'un type a lui-même une structure d'arbre et de sous-arbres emboîtés, il semble nécessaire de devoir le développer jusqu'aux types de base qui constituent en fait les feuilles d'un tel arbre.

En fait, nous savons que les informations dont nous avons besoin se trouvent à deux endroits dans l'arbre abstrait:

- la déclaration de tous les objets utitlisés et de leur type correspondant se trouve dans la partie "profil " de la déclaration de l'opération, soit dans le sous-arbre *profobj* de l'arbre *profilop* dans notre syntaxe abstraite.

- l'explicitation de tous les types se trouve quant à elle dans la partie "déclaration des types ", soit dans le sous-arbre *suitebloctypobj* de notre syntaxe.

La totalité du raisonnement théorique étant basée sur la connaissance du type de tous les objets (voir "is-cp-of", "is-seq-of", "is-union-of"), il s'agit donc d'extraire cette information des sous-arbres concernés, de la synthétiser puis de la passer en héritage à chaque noeud de la partie réellement utile de l'arbre abstrait. La réalisation de cet objectif est atteinte par l'utilisation d'attributs et se découpe en quatres phases:

1. Nous synthétisons, dans le sous-arbre correspondant aux explicitations de types d'objets, l'information concernant tous les types et leur explicitation. Ce renseignement est conservé dans un attribut de nom *typexp* qui accumule toutes les explicitations des types. La structure de cet attribut est celle d'une liste de couples, un couple étant constitué de deux éléments (un type et son

explicitation), tous deux de structure *fterme* [3].Une telle liste est désignée par *fterfter*. La construction en soi de cet attribut ne pose aucun problème, si ce n'est celui de veiller à ce que chaque composante ait bien la structure *fterme*.

En termes de notre syntaxe abstraite, cet attribut est synthétisé (construit) dans le sous-arbre *suitebloctypobj* pour remonter au niveau le plus haut de la spécification, c'est-à-dire jusqu'au phylum *spécification*. A partir de cet endroit, cet attribut pourrait être passé en héritage à tout le sous-arbre contenant la description de toutes les opérations, soit à *suiteblocoper*. En effet, la portée de ce type est tout l'arbre contenant la description des opérations. Mais rappelons-nous qu'en réalité un type d'objet a lui-même une structure d'arbres emboîtés. Il nous a donc semblé que le noeud *spécification* était le bon endroit pour déplier tous les types qui nous sont fournis dans *typexp*.

Le dépliage consiste à construire un attribut, que nous avons appelé *typexpdec*, qui contient tous les types et leur dépliage. Cet attribut a la même structure que l'attribut *typexp*, mais dans lequel chaque type est explicité complétement, c'est-à-dire uniquement en fonction des types de base. Cette transformation est réalisée à l'aide de la fonction *detypexpdontypexpdec*. Le fait de devoir déplier jusqu'aux types de base est obligatoire pour garantir la terminaison de la fonction récursive *detypexpdontypexpdec*, dans son état actuel du moins. Il est clair qu'une extension possible serait de modifier cette fonction pour que le dépliage ne se fasse pas au-delà des types strictement nécessaires et pas forcément d'imposer ce dépliage jusqu'aux types de base. Ceci revient essentiellement à considérer que la portée d'un type (lors de l'explicitation d'un type) n'est pas l'arbre tout entier, mais que cette portée se limite au bloc contenant la spécification d'une opération par exemple.

Le lecteur intéressé par les détails techniques trouvera cette fonction, ainsi que le texte complet du programme, en annexes. Pour le commun des mortels, nous nous contenterons ici d'illustrer le principe du dépliage par un petit exemple.

---

[3] La structure *fterme* est en fait un domaine syntaxique de la syntaxe abstraite du langage des prédicats du premier ordre FOPL. Cette syntaxe se trouve en annexe.

Soit la suite d'explicitations de types contenue dans *suitebloctypobj* :

A <-> CP [A1 , A2 , **INT**] [4]
B <-> SEQ [B2]
A1 <-> **INT** (exemple 1)
A2 <-> CP [**CHAR** , B]
B2 <-> UNION [**DATE** , **BOOL**]

L'attribut *typexp* sera la liste suivante :

{ (A , CP [A1 , A2 , **INT**]) , (B , SEQ [B2]) , (A1 , **INT**),
(A2 , CP [**CHAR** , B]) , (B2 , UNION [**DATE** , **BOOL**])
} [5]

Tandis que l'attribut *typexpdec* sera la liste :

{ (A, CP[**INT**,CP[**CHAR**, SEQ[UNION[**DATE**, **BOOL**]]], **INT**]),
(B , SEQ [UNION [**DATE** , **BOOL**]]) ,
(A1 , **INT**) ,
(A2 , CP [**CHAR** , SEQ [UNION [**DATE** , **BOOL**]]]) ,
(B2 , UNION [**DATE** , **BOOL**])
}

En résumé, au niveau de *spécification*, nous recevons un attribut synthétisé *typexp* que nous modifions légèrement en un attribut *typexpdec* qui sera passé en héritage à l'arbre *suiteblocoper* , fils de *spécification*.

2. Nous synthétisons dans le sous-arbre correspondant aux déclarations de toutes les variables utilisées dans l'explicitation d'une opération (le raisonnement se fait opération par opération étant bien entendu que la portée d'une variable se limite au bloc de spécification de l'opération dans lequel elle est déclarée), l'information concernant toutes les variables et leur type associé dans la déclaration. Ce renseignement est conservé dans un attribut

[4] Les types de base sont écrits en caractères gras.
[5] Ceci n'est qu'une représentation visuelle de la liste, puisqu'en réalité cette liste est constituée de *fterme* et donc est plutôt à voir comme un arbre abstrait.

de nom *vtyp* , dont la structure est celle d'une liste de couples, un couple étant constitué de deux éléments (une variable et son type), tous deux de structure *id* , élément de la syntaxe abstraite RSL. Une telle liste est désignée par *idid*.

En termes de notre syntaxe, cet attribut est synthétisé dans le sous-arbre *profilop* pour remonter au noeud *blocoper* sur lequel il sera traité. La construction de cet attribut au sein de *profilop* ne pose aucun problème et nous ne nous y attarderons pas.

Prenons un exemple et imaginons que dans *profilop* se trouve la suite de déclarations suivante :

a : A
b : B2
c : A1 (exemple 2)
d : B

L'attribut *vtyp* aura pour valeur la liste suivante :

{ (a , A) , (b , B2) , (c , A1) , (d , B) }

3. Le noeud *blocoper* a hérité de son père l'attribut *typexpdec* (qui contient tous les types et leur dépliage) et vient de recevoir d'un de ses fils l'attribut *vtyp* qui contient les variables de l'opération et leur type associé. Le type attaché à une variable dans *vtyp*. n'est pas forcément un type complétement déplié. Il faut donc à partir de ces deux attributs en produire un seul qui fera correspondre à toute variable son type déplié jusqu'aux types de base. La fonction *devarsdonexpls* réalise cette opération en construisant l'attribut *vextyp* que le noeud *blocoper* pourra passer en héritage à son fils *decformop*. La structure de l'attribut *vextyp* est celle d'une liste de couples, constituée de deux éléments, le premier désignant la variable est de structure *id* et le second désignant le type déplié est de structure *fterme*. Une telle liste est désignée par *idfterme*.

Si l'on reprend l'exemple 1 et l'exemple 2 ci-dessus, l'attribut *vextyp* à passer en héritage aura la forme

{ (a , CP[**INT**,CP[**CHAR**,SEQ[UNION[**DATE**,**BOOL**]]] , **INT**]) ,
(d , SEQ [UNION [**DATE** , **BOOL**]]) ,
(c , **INT**) ,
(b , UNION [**DATE** , **BOOL**])
}

4. *decformop* a reçu en héritage l'attribut *vextyp* contenant les variables de l'opération et leur type déplié. Ce dernier pourra être passé en héritage et sans plus jamais être modifié, de noeud en noeud à travers tout le sous-arbre correspondant à *decformop*, et cela jusqu'aux feuilles de ce sous-arbre.

Nous disposons à présent grâce à cet attribut *vextyp*., dans tout l'arbre concernant l'explicitation d'une opération, de l'ensemble des objets déclarés et de leur type déplié jusqu'aux types de base. Voyons comment utiliser cette information, sur laquelle repose tout le développement théorique, pour créer les équations implicites dans le cas d'assertions simples d'abord, dans celui des assertions composées ensuite.

4.4.4. Le cas des assertions simples.

4.4.4.1. Quelles variables sont concernées?

Si l'on se réfère à la théorie et plus particulièrement au concept de non-ambiguité (définition 4.3.), les seules variables du membre de droite de l'assertion d'explicitation pour lesquelles nous n'aurons pas à produire des équations implicites sont les variables qui

- apparaissent telles quelles comme arguments ou résultat dans le membre de gauche de l'assertion d'explicitation (puisqu'alors elles sont libres) (cas c1 de la définition 4.3.),

- sont des objets intermédiaires introduits par un connecteur with (cas a de la définition 4.3.),

- sont des éléments représentatifs introduits par un connecteur forall (cas b de la définition 4.3.).

Il serait donc utile de disposer à tout instant dans l'arbre de la liste des variables qui vérifient une de ces conditions et pour lesquelles aucun traitement particulier ne sera nécessaire. A cette fin, nous avons introduit un attribut, noté *vgfw* , dont la valeur sera précisément la liste de ces variables. La structure de cet attribut découle de sa fonctionnalité: ce sera une liste d'éléments dont la structure est *id* (le nom seul de la variable nous importe). Cette liste est désignée par *sid*.

Il reste le problème de savoir où sera synthétisé cet attribut. L'endroit tout indiqué nous semble être la déclaration formelle de l'opération, soit *decformop* dans notre syntaxe. En effet, à cet endroit, on dispose, dans la déclaration du résultat et dans la déclaration de la liste des arguments, de la totalité des variables libres (celles qui sont dans le membre de gauche de l'assertion d'explicitation).

L'attribut *vgfw* sera donc synthétisé au niveau de *decformop* puis il sera passé en héritage à tous les fils de ce noeud et ce jusqu'aux feuilles de ce sous-arbre. Cependant, ne perdons pas de vue que la valeur de cet attribut devra être ajustée pendant le parcours de l'arbre. Là où nous aurons affaires à un noeud forall ou à un noeud with, il faudra compléter la liste des variables héritées par ce noeud via l'attribut *vgfw* en y ajoutant l'élément représentatif dans le cas du forall et le(s) objet(s) intermédiaire(s) introduits(s) par un with (andwith). Ces objets intermédiaires sont contenus dans un attribut *vinter* de même type que *vgfw* et synthétisé dans l'explicitation de l'opération intermédiaire (*sexprinterm* dans le cas de notre syntaxe). Ainsi, cet attribut contient à chaque noeud les variables qui sont libres dans les sous-arbres fils de ce noeud (pour lesquelles aucune équation implicite n'est à découvrir).

Une petite remarque doit être faite également au sujet des éventuelles préconditions.

Qu'il y ait ou non une précondition n'influence pas les attributs créés jusqu'ici (*typexp* , *typexpdec* , *vextyp* , *vtyp*), puisque ces attributs sont synthétisés sur des noeuds ancêtres de celui contenant l'expression de la précondition. Malheureusement, il n'en est pas de même pour l'attribut *vgfw* qui

est précisément synthétisé sur ce noeud. De plus, l'ensemble des variables libres associées à l'opération n'est pas forcément le même que celui des variables libres associées à une précondition. C'est pourquoi nous synthétisons dans l'explicitation *expre* de la précondition l'attribut pendant de *vgfw*, soit *vgfwpre*, qui a exactement la même structure et joue exactement le même rôle, mais dans l'arbre associé à la précondition *expre* . N'oublions pas que nous traitons une précondition comme un cas particulier d'une opération.

Nous pouvons à présent dire, à n'importe quel endroit dans l'arbre, si une variable est concernée ou non par la production d'une équation implicite: elle ne le sera pas si elle est dans l'attribut *vgfw*, elle le sera dans le cas contraire. De plus, nous pouvons disposer à tout endroit de l'arbre du type des variables (dans l'attribut *vextyp*). Ces deux informations combinées vont nous permettre de générer les équations implicites, dans le cas des assertions simples d'abord, dans celui des assertions composées ensuite.

4.4.4.2. Produire une équation implicite.

Rappelons qu'une équation implicite dans le cas d'assertions simples a toujours la forme[6]

vd = SEL (vg)

où

- vd est une variable du membre de droite de l'assertion d'explicitation, qui n'est ni liée, ni un objet intermédiaire, ni un élément représentatif (cette variable vérifie la condition c2 de la définition 4.3.),
- vg est la variable du membre de gauche de l'assertion d'explicitation dont le type a pour sous-structure le type de vd,
- SEL est un des opérateurs de sélection dépendant de la relation entre les types (au sens de la définition 4.2.).

Pour construire une telle équation implicite, nous nous sommes dit que le plus simple était de partir des feuilles, soit au niveau *exprop* dans notre syntaxe, et de remonter dans l'arbre abstrait.

---

[6] Revoir la définition 1 et la définition 2 au point 4.2.3.1.

Rappelons qu'au niveau des feuilles, comme partout ailleurs, nous disposons de *vgfw* qui permettra de sélectionner les variables concernées, et de *vextyp* qui permettra de connaître le type exact de chaque variable concernée. Le but avoué est de trouver finalement la variable du membre de gauche à faire correspondre à une variable donnée du membre de droite.

Sachant où nous commencerons à construire nos équations implicites (aux feuilles), voyons comment représenter ces dernières. En fait, si on la regarde bien, une équation implicite de la forme $\alpha = SEL(\beta)$ peut se représenter par un triplet

$$(\alpha, SEL(\beta), \delta)$$

où α est la variable qui nécessite une équation implicite,
β est la variable du membre de gauche à trouver,
δ est le type de cette variable β, type qui a le type de α comme sous-structure (définition 4.1.).

Si nous représentons de la sorte l'équation implicite qu'il faut générer pour une variable concernée (c'est une variable qui n'appartient pas à *vgfw*), la représentation des équations implicites de toutes les autres variables également concernées consistera en une liste de tels triplets. Il suffit dès lors de considérer un attribut dont la valeur sera cette liste. Nous avons appelé cet attribut *vrimp*. Il est synthétisé des feuilles jusqu'à la racine de l'arbre concerné par la recherche des équations implicites (la variable β sera connue à coup sûr au niveau de la racine).

Décrivons la façon de synthétiser cet attribut *vrimp* en utilisant les attributs *vgfw* et *typexpdec* . Selon l'endroit où nous nous trouvons dans l'arbre, la théorie nous indique la façon de procéder à la construction des équations implicites. Nous exposons ci-dessous comment nous avons réalisé cela et en quoi la démarche de construction reste proche de la théorie.

- Si nous nous trouvons au niveau des feuilles.
  Au niveau des feuilles, il faut retenir chaque variable qui va nécessiter une équation implicite: elle ne se trouve pas dans *vgfw*. Cette étape qui est sous-entendue au niveau de la théorie, impose simplement d'initialiser l'attribut *vrimp* avec les dites variables.

Règle-feuilles-1

Pour chaque variable vd de type TPD faisant partie des feuilles associées à l'arbre correspondant au membre de droite de l'assertion d'explicitation, c'est-à-dire pour chaque variable du noeud *exprop* de notre syntaxe,

si vd ∉ *vgfw* (vd est une variable nécessitant une équation implicite)

alors ajouter à *vrimp* **le triplet (vd , ω , TPD*)**

sinon aucun traitement n'est à effectuer

où TPD* désigne le type TPD déplié jusqu'aux types de base. Ce type est obtenu par l'attribut hérité *typexpdec*

ω désigne une valeur momentanément inconnue qui représentera le terme de sélection en cours de construction et qui, au niveau de la racine sera remplacée par la variable de gauche correspondant à vd.

- Si nous nous trouvons sur un noeud and.

Sur un noeud and, la théorie nous indique comment générer les équations de sélection pour une seule variable (règle R1(a)). Il suffit d'appliquer cette règle à toutes les variables concernées (ce sont les différents triplets de *vrimp*).

Règle-and-1

Pour chaque triplet T = (vd , ω , TPD*) de *vrimp* ,

si ∃ **dans l'attribut** *typexpdec* **CP [... , TPD* , ...]**
(i.e. is-cp-of (... , TPD*))

alors remplacer le triplet T par le triplet
(vd , TPD* (: ω) , CP [... , TPD* , ...])
(i.e. générer vd = TPD*(:ω))

sinon le triplet T est inchangé

- Si nous nous trouvons sur un noeud or.

Sur un noeud or, la théorie nous indique comment générer les équations de sélection pour une seule variable (règle R1(b)). Il suffit d'appliquer cette règle à toutes les variables concernées (ce sont les différents triplets de *vrimp*).

Règle-or-1

Pour chaque triplet T = (vd , ω , TPD*) de *vrimp* ,

si ∃ **dans l'attribut** ***typexpdec*** **UNION [... , TPD* , ...]**
(i.e. is-union-of(... , TPD*))

alors remplacer le triplet T par le triplet
(vd , AS_{TPD*} (ω) , UNION [... , TPD* , ...])
(i.e. générer vd = AS_{TPD*}(ω))

sinon le triplet T est inchangé

- Si nous nous trouvons sur un noeud forall.

Sur un noeud forall, la théorie nous indique comment générer les équations de sélection pour une seule variable (règle R1(c)). Il suffit d'appliquer cette règle à toutes les variables concernées (ce sont les différents triplets de *vrimp*).

Règle-forall-1

Pour chaque triplet T = (vd , ω , TPD*) de *vrimp* ,

si ∃ **dans l'attribut** ***typexpdec*** **SEQ [TPD*]**
(i.e. is-seq-of(TPD*))

alors remplacer le triplet T par le triplet
(vd , ITH (K , ω) , SEQ [TPD*])
(i.e. générer vd = ITH(K, ω))

sinon le triplet T est inchangé

- Si nous nous trouvons sur un noeud autre que ceux cités ci-dessus.
  Tout triplet T = (vd , ω , TPD*) de *vrimp* reste inchangé. L'attribut *vrimp* ne subit aucune modification sur de tels noeuds: ce ne sont pas des noeuds pouvant générer des équations implicites au sens des définitions 4.1.,4.2. et 4.3.

- Si nous nous trouvons au niveau de la racine.
  Nous venons de voir comment évolue l'attribut *vrimp* contenant les équations implicites en respectant la règle R1, ce qui en garantit l'exactitude. Au niveau de la racine, il suffit de compléter chacune de ces équations implicites par la variable de gauche correspondant à une variable de droite. Mais qu'appelle-t-on exactement "racine"?

  La racine de l'arbre dans lequel s'effectue la recherche des équations implicites est l'endroit où l'assertion est explicitée, soit au noeud *decformop* dans notre syntaxe.

  A ce noeud racine, nous obtenons de nos fils l'attribut *vrimp* dans lequel il faut compléter le deuxième élément de chaque triplet par le nom d'une et une seule variable vg du membre de gauche (l'unicité est garantie par le point c2 de la définition 4.3. et par le respect de la règle R1) et nous produirons un autre attribut qui désigne l'équation implicite ainsi complétée.

  Règle-racine-1

  Pour chaque triplet T = (vd , SEL (ω) , TPω [7]) de *vrimp* ,

  ∃ ! vg dans l'attribut *typexpdec* tel que TPG ≡ TPω
  et il faut produire (vd , SEL (vg))

  où (vd, SEL(vg)) est une représentation de l'équation implicite vd = SEL(vg).

  Ce résultat sera conservé dans un attribut *eqimp* dont la structure est celle d'une liste de couples; un couple est constitué de deux éléments, le premier de structure *id* désigne la variable et le second de structure *fterme* désigne

[7] Lire " le type de ω ".

l'opération de sélection qui est associée à cette variable. Cette liste est désignée par *idfterme*.

Comme nous l'avons vu et justifié ci-dessus, cette représentation conduit bien aux équations implicites recherchées.

Illustrons, sur un petit exemple qui va servir de jeu de test, l'évolution de la valeur de l'attribut *vrimp* vers la valeur de l'attribut *eqimp*. contenant une représentation des équations implicites.

explicitation de l'assertion:

res = OP (arg1, arg2) →
res1 = OP1 (arg1' , arg2)
and
res2 = OP2 (arg1" , arg2)

explicitation des types:

RES : CP [RES1 , RES2]
ARG1 : CP [ARG1' , ARG1"]
*ARG2 : **INT***
*RES1 : **CHAR***
*RES2 : **BOOL***
*ARG1' : **INT***
*ARG1" : **CHARST***

déclaration des variables:

res : RES
res1 : RES1
res2 : RES2
arg2 : ARG2
arg1 : ARG1
arg1' : ARG1'
arg1" : ARG1"

1) De par leur définition, nous pouvons donner la valeur des attributs (voir 4.4.3.)

vgfw { res , arg1 , arg2 }

vextyp { (res,CP[**CHAR**,**BOOL**]), (res1,**CHAR**) , (res2, **BOOL**) ,
(arg1 , CP [**INT** , **CHARST**) , (arg1' , **INT**) ,
(arg1" , **CHARST**) , (arg2 , **INT**) }

2) Nous disposons de ces attributs aux feuilles. A cet endroit on a également l'ensemble des variables de droite { res1 , res2 , arg1' , arg1" , arg2 } et nous allons devoir construire *vrimp* comme expliqué ci-dessus :

arg2 ∈ *vgfw* nous n'avons rien à faire
res1 ∉ *vgfw* nous générons (res1 , ω , **CHAR**) (Règle-feuilles-1)
res2 ∉ *vgfw* nous générons (res2 , ω , **BOOL**) (Règle-feuilles-1)
arg1' ∉ *vgfw* nous générons (arg1' , ω , **INT**) (Règle-feuilles-1)
arg1" ∉ *vgfw* nous générons (arg1" , ω , **CHARST**) (Règle-feuilles-1)

la valeur de l'attribut *vrimp* est donc
{ (res1 , ω , **CHAR**) , (res2 , ω , **BOOL**) , (arg1' , ω , **INT**) ,
(arg1" , ω , **CHARST**) }

Selon l'explicitation de l'assertion, nous sommes sur un noeud and. Nous appliquons l'algorithme adéquat:

Pour (res1 , ω , **CHAR**) (Règle-and-1)
∃ **CP [CHAR , BOOL]**
donc *vrimp* = { (res1 , **CHAR** (: ω) , **CP [CHAR , BOOL]**) ,
(res2 , ω , **BOOL**) , (arg1' , ω , **INT**) ,
(arg1" , ω , **CHARST**) }

Pour (res2 , ω , **BOOL**) (Règle-and-1)
∃ **CP [CHAR , BOOL]**
donc *vrimp* = { (res1 , **CHAR** (: ω) , **CP [CHAR , BOOL]**) ,
(res2 , **BOOL** (: ω) , **CP [CHAR , BOOL]**) ,
(arg1' , ω , **INT**) ,
(arg1" , ω , **CHARST**) }

Pour (arg1' , ω , **INT**) (Règle-and-1)

∃ **CP [INT , CHARST]**

donc *vrimp* = { (res1 , **CHAR** (: ω) , **CP [CHAR , BOOL]**) ,
(res2 , **BOOL** (: ω) , **CP [CHAR , BOOL]**) ,
(arg1' , **INT** (: ω) , **CP [INT , CHARST]**) ,
(arg1" , ω , **CHARST**) }

Pour (arg1" , ω , **CHARST**) (Règle-and-1)

∃ **CP [INT , CHARST]**

donc *vrimp* = { (res1 , **CHAR** (: ω) , **CP [CHAR , BOOL]**) ,
(res2 , **BOOL** (: ω) , **CP [CHAR , BOOL]**) ,
(arg1' , **INT** (: ω) , **CP [INT , CHARST]**) ,
(arg1", **CHARST** (: ω) , **CP[INT,CHARST]**) }

3) A la racine, nous recherchons dans *vextyp* la variable dont le type coïncide avec le type du triplet considéré: ainsi pour

(res1 , **CHAR** (: ω) , **CP [CHAR , BOOL]**) on trouve res (Règle-racine-1)

(res2 , **BOOL** (: ω) , **CP [CHAR , BOOL]**) on touve res (Règle-racine-1)

(arg1' , **INT** (: ω) , **CP [INT , CHARST]**) on trouve arg1 (Règle-racine-1)

(arg1" , **CHARST** (: ω) , **CP [INT , CHARST]**) on trouve arg1 (Règle-racine-1)

Ceci permet de générer l'attribut *eqimp* ((Règle-racine-1)) dont la valeur est

{ (res1 , **CHAR** (: res)) , (res2 , **BOOL** (: res)) ,
(arg1' , **INT** (: arg1)) , (arg1" , **CHARST** (: arg1)) }

et qui est bien une représentation des équations implicites attendues

res1 = **CHAR** (: res)
res2 = **BOOL** (: res)
arg1' = **INT** (: arg1)
arg1" = **CHARST** (: arg1)

4.4.5. Le cas des assertions composées.

4.4.5.1. Mise au point.

Le raisonnement dans le cas des assertions simples nous a conduits à la découverte de la structure d'un attribut dont la valeur finale représente les équations implicites recherchées et la valeur intermédiaire représente ces équations "en cours de construction". Expliquons comment cet attribut reste de mise pour le cas des assertions composées, moyennant de légères modifications.

Nous savons d'après la théorie (point c2 de la définition 4.3.) que seulement trois noeuds posent problème: les noeuds and , or et forall. Nous examinerons le cas de chacun d'eux en particulier.

Le raisonnement récursif adopté est le suivant: étant sur un de ces noeuds-là, nous recevons de notre noeud père des attributs hérités et de nos noeuds fils des attributs synthétisés. Que faut-il que ce noeud apporte comme modifications aux attributs hérités (respectivement synthétisés) pour les passer correctement à ses noeuds fils (respectivement à son noeud père)?

En réalité, la réponse à cette question nous est en partie fournie par la théorie, et plus particulièrement par la définition 4.3.bis de la non ambiguïté dans le cas des assertions composées.

Si nous analysons cette définition en supposant être sur un noeud problématique, nous nous rendons compte qu'un objet que nous recevons d'un de nos noeuds fils est

soit (cas c1' de la définition 4.3.bis) une variable simple localement libre dans ce sous-arbre (c'est-à-dire qui n'est encore liée à aucun noeud du sous-arbre du noeud courant) : son équation implicite n'est pas encore générée. Deux cas peuvent dès lors typiquement se produire:

i- ou bien il faudra lier cette variable au noeud courant et produire son équation implicite (on a une relation du type "is-cp-of", "is-union-of" ou "is-seq-of"), **(cas c1'-a)**

ii- ou bien il faudra transmettre au noeud père cette variable en laissant son statut inchangé (on a une relation du type "is-same-str").

(cas c1'-b)

soit (cas c2' de la définition 4.3.bis) un terme composé pour lequel il faudra éventuellement générer une opération de sélection. Deux situations peuvent en effet se produire dans ce cas:

i- ou bien ce terme est concerné par le noeud courant et il faudra compléter son équation implicite (on a une relation du type "is-cp-of", "is-union-of" ou "is-seq-of", **(cas c2'-a)**

ii- ou bien ce terme n'est pas concerné par le noeud courant et il faudra le transmettre au noeud père en le laissant tel qu'on l'a reçu (on a une relation du type "is-same-str"). **(cas c2'-b)**

Il nous faudra donc tenir trace d'un certain historique lié à chaque objet, cet historique dépendant de la possibilité ou non de générer une opération de sélection pour cet objet. L'historique étant lié à l'opération de sélection, nous avons décidé de compléter l'attribut *vrimp* en transformant le triplet initial (α , β , δ) en un quadruplet (α , β , δ , ε) où $\varepsilon = -1$ si α est une variable localement libre (ce qui correspond au cas c1'-b), et $\varepsilon = 1$ si α est une variable déjà liée à un noeud.

La valeur de l'attribut *vrimp* est donc une liste de quadruplets de la forme

$$(\alpha , \textbf{CSEL} (\beta) , \delta , \varepsilon)$$

où

- α est la variable qui nécessite une équation implicite,
- CSEL est une composition quelconque, éventuellement vide, d'opérateurs de sélection,
- β est la variable du membre de gauche à trouver,
- δ est le type de cette variable β, type qui a le type de α comme sous-structure,
- $\varepsilon = -1$ si α est une variable localement libre,
- $\varepsilon = 1$ si α est une variable déjà liée à un noeud fils.

Dans notre syntaxe, cette liste sera désignée par *iffi*, rappelant que la structure de α est *id*, celle de β et de δ*fterme*, celle de ε **INT**.

Nous constatons donc que l'attribut représentant les équations implicites ne subit que très peu de changements par rapport à sa forme décrite dans le cas des assertions simples. Il faut donc s'attendre à ce que son évolution le long des noeuds de l'arbre dans le cas des assertions composées que nous traitons actuellement soit un reflet de celle décrite dans le cas des assertions simples.

Au niveau des feuilles, cette liste *vrimp* sera initialisée comme nous l'avons vu pour les assertions simples à partir des variables nécessitant une équation implicite.

Règle-feuilles-2

Pour chaque variable vd de type TPD faisant partie des feuilles associées à l'arbre correspondant au membre de droite de l'assertion d'explicitation, c'est-à-dire pour chaque variable du noeud *exprop* de notre syntaxe,

si vd ∉ *vgfw* (vd est une variable nécessitant une équation implicite)

alors ajouter à *vrimp* **le quadruplet (vd , ω , TPD* , -1)**

sinon aucun traitement n'est à effectuer

où

TPD* désigne le type TPD déplié jusqu'aux types de base. Ce type est obtenu par l'attribut hérité *typexpdec*

ω désigne une valeur momentanément inconnue qui représentera le terme de sélection en cours de construction et qui, au niveau de la racine, sera remplacée par la variable de gauche correspondant à vd.

-1 puisqu'au niveau des feuilles une variable n'est encore liée à aucun noeud.

Disposant de l'attribut *vrimp* initialisé au niveau des feuilles, comment le mettre à jour pour le passer correctement au noeud père d'un noeud problématique? C'est ce que nous allons essayer d'expliquer en passant en revue chacun des trois noeuds problématiques.

4.4.5.2. Le cas d'un noeud and.

Un noeud and a la particularité de posséder deux et seulement deux fils que nous appellerons selon notre syntaxe (et selon les notations SSL) *explicop$1* et *explicop$2*. D'après nos conventions, le père de ce noeud portera le nom *prexplicop*. Sous forme de graphe, nous avons donc la structure

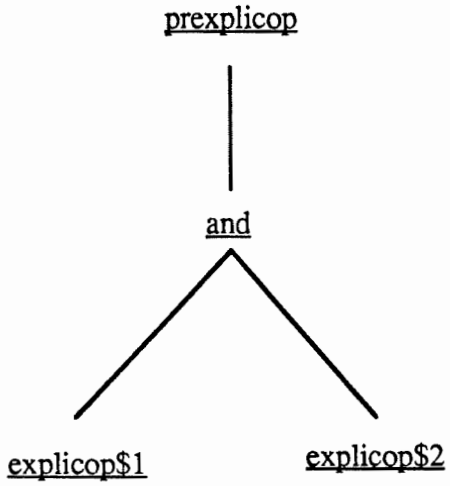


fig. 4.7. Représentation d'un noeud and.

Rappelons que du noeud père nous recevons en héritage les attributs *vgfw* (contenant toutes les variables ne nécessitant aucune équation implicite) et *vextyp* (contenant les variables et leur type déplié) et que de chacun des noeuds fils nous recevons un attribut *vrimp* (contenant les équations implicites en cours de construction). Comment allons-nous passer ces attributs du père aux fils, et des fils au père?

De par leur nature et leur fonctionnalité, les attributs reçus en héritage passeront tels quels du père aux fils, ce qui s'écrit en SSL

```
explicop$1.vgfw = $$.vgfw[8]
explicop$2.vgfw = $$.vgfw
explicop$1.vextyp = $$.vextyp
explicop$2.vextyp = $$.vextyp
```

[8] En SSL, $$ est une notation tolérée pour représenter le noeud père qui est unique. La notation $$.vgfw signifie l'attribut venant du père.

Il reste le cas de l'attribut synthétisé *vrimp*, dont la valeur est une liste de quadruplets de la forme (vd , CSEL (ω) , TPω , ε). Le problème est de mettre à jour cet attribut correctement, c'est-à-dire en respectant les règles fournies par la théorie. Entre autres, d'après cette théorie, quatre cas seront à envisager: les cas c1'-a , c1'-b , c2'-a et c2'-b énoncés ci-dessus.

En fait, nous recevons une telle liste *vrimp* du fils explicop$1 et une autre du fils explicop$2. Rien ne nous permet d'affirmer que ces listes soient disjointes. En effet, une variable vd qui serait libre localement pour le sous-arbre déterminé par le noeud courant, se retrouverait typiquement comme premier élément d'un quadruplet dans chacune de ces deux listes.

Considérons donc un attribut intermédiaire *vrimpinter* dont la valeur est la liste intersection des listes correspondant aux attributs explicop$1.vrimp et explicop$2.vrimp. Dans cette liste intersection, le premier élément de chaque quadruplet désigne une variable simple qui est libre dans chacun des deux sous-arbres (c'est-à-dire ε = 1 dans les quadruplets concernés dans chaque sous-arbre). Une telle variable ne peut être que libre aussi sur le noeud and courant (elle sera liée à un noeud plus haut dans l'arbre) et sera donc passée telle quelle dans l'attribut *vrimp* du noeud père (cas c1'-b). Ainsi, une partie des variables non concernées par ce noeud est trivialement traitée. Considérons les quadruplets qui appartiennent à la liste donnée par l'attribut du premier sous-arbre explicop$1.vrimp et pas à la liste intersection *vrimpinter*: Ces quadruplets forment une liste que nous appellerons *vrimpg*. De manière symétrique, nous avons une liste *vrimpd* qui a les mêmes caractéristiques. Dans chacune de ces deux listes disjointes, le premier élément de chaque quadruplet désigne une variable simple qui est susceptible d'engendrer une opération de sélection correspondant au noeud and. Parmi ces variables simples, certaines seront éventuellement liées à ce noeud and bien précis (cas c1'-a), les autres correspondent à des termes pour lesquels il faut compléter l'équation implicite (cas c2'-a) ou bien des termes à transmettre tels quels (cas c2'-b et c1'-b).

Le but est donc de déterminer quelles variables simples seront liées au noeud courant et comment l'attribut *vrimp* du noeud père sera mis-à-jour sur ce noeud. Pour ce faire, nous procéderons en trois étapes:

- nous pointons les termes candidats à la production d'une équation implicite,
- nous déduisons les variables à lier au noeud courant,
- nous mettons à jour la construction des équations implicites sur le noeud and.

Voici la description détaillée de ces trois étapes.

1. Nous allons détecter les termes[9] candidats à la production d'une équation implicite sur ce noeud and bien précis (cas c1'-a et c2'-a).
   C'est ainsi qu'à partir des attributs *vrimp* des fils et utilisant l' attribut *typexpdec*, nous construirons deux attributs intermédiaires dont la valeur est une liste de triplets de la forme (v , T , n)
   où v représente une variable simple provenant du sous-arbre concerné et susceptible d'être liée à ce noeud,
   T représente un type d'objet, le type de cette variable,
   n vaut 1 ou -1, selon que la variable est une variable déjà liée plus bas dans l'arbre ou pas encore.
   Une telle liste est désignée par *idfterint*.
   Ces attributs, que nous appellerons *vtypentg* et *vtypentd*, sont construits en appliquant à chacune des deux listes disjointes ci-dessus l'algorithme vu dans le cas particulier des assertions simples (Règle-and-1) et en l'adaptant légèrement (ce cas-ci n'est jamais qu'une généralisation du cas des assertions simples):

   Règle-and-2-bis'

   Pour chaque quadruplet Q = (vd , CSEL (ω) , TPω , ε) de *vrimpg*,

   si ∃ **dans l'attribut** *typexpdec* **CP [... , TPω, ...]**
   (i.e. "is-cp-of")

   alors ajouter à *vtypentg* **le triplet**
   (vd , CP [... , TPω , ...] , ε)

   sinon aucun traitement n'est à effectuer

9 Une variable est en fait un cas particulier de terme.

Règle-amd-2-bis''

Pour chaque quadruplet Q = (vd , CSEL (ω) , TPω, ε) de *vrimpd* ,

si ∃ **dans l'attribut** ***typexpdec*** **CP [... , TPω, ...]**
(i.e. "is-cp-of")

alors ajouter à ***vtypentd*** **le triplet**
(vd , CP [... , TPω , ...] , ε)

sinon aucun traitement n'est à effectuer

L'application de ces deux règles conduit bien à la génération des termes susceptibles de produire ou de développer une équation implicite.

2. Maitenant, en utilisant ces deux listes de triplets, nous détecterons les variables à lier au noeud courant (cas c1'-a). Une variable de ces triplets (vd , T , ε) est à lier
si elle n'est pas encore attachée à un noeud (elle est donc pointée à "-1") et
si le type T et le type du terme pendant dans l'autre branche, s'il existe, sont des sous-types du même type du noeud père.

Nous conserverons ces variables dans un attribut appelé *varbonend*, dont la valeur est la liste de ces variables. Cette liste sera désignée par *sid*. La construction de cette liste se déroule de la façon suivante:

Règle-amd-2-ter'

Pour chaque triplet Tg = (vdg , TPg , εg) de *vtypentg*

si ∃ **un triplet Td = (vdd , TPd , εd) de** ***vtypentd***
tel que TPg ≡ TPd (les types des variables vdg et vdd sont sous-types du même type)
et εg = -1 (et la variable vdg était libre localement jusqu'ici)

alors ajouter vdg à ***varbonend*** (on est dans le cas c1'-a)

sinon aucun traitement n'est à effectuer

et symétriquement

Règle-and-2-ter''

Pour chaque triplet Td = (vdd , TPd , εd) de *vtypentd*

si ∃ **un triplet Tg = (vdg , TPg , εg) de** ***vtypentg***
tel que TPd ≡ TPg (les types des variables vdd et vdg sont sous-types du même type)
et εd = -1 (et la variable vdd était libre localement jusqu'ici)

alors ajouter vdd à ***varbonend***. (on est dans le cas c1'-a)

sinon aucun traitement n'est à effectuer

3. Nous mettrons à jour l'attribut *vrimp* à passer au père du noeud courant en respectant les cas c1'-a , c1'-b , c2'-a et c2'-b vus ci-avant. Nous disposerons à ce moment de
   - la liste des variables à lier au noeud courant : *varbonend* construite à l'étape précédente,
   - la liste des quadruplets Q = (vdq , CSEL (ωq) , Tωq , εq) appartenant soit à *vrimpg* soit à *vrimpd*,
   - la liste des triplets T = (vdt , CP[... , Tωt , ...] , εt) de *vtypentg* et de *vtypentd*.

Chaque quadruplet Q doit se retrouver, modifié ou non, dans l'attribut *vrimp* du noeud père.Quand et comment ce quadruplet sera-t-il modifié? La réponse est donnée par l'algorithme suivant:

Règle-and-2

Pour chaque quadruplet Q de *vrimpg* et de *vrimpd*, les attributs des noeuds fils respectifs,

Si (∃ vdt dans T de *vtypent* avec vdt = vdq)
et (vdq ∈ *varbonend*)
et εq = -1
alors ajouter à $$.vrimp le quadruplet
(vdq,CSEL(Tωq(: ωq)),CP[... , Tωt , ...],1)[10]
(la variable vdq était libre et devient liée à ce noeud)
(cas c1'-a)

Si (∃ vdt dans T de *vtypent* avec vdt = vdq)
et (vdq ∉ *varbonend*)
et εq = 1
alors ajouter à $$.vrimp le quadruplet
(vdq,CSEL(Tωq(: ωq)) ,CP[... , Tωt , ...] , εq)
(la variable vdq est liée et son équation se construit petit à petit)
(cas c2'-a)

Si (∃ vdt dans T de *vtypent* avec vdt = vdq)
et (vdq ∉ *varbonend*)
et εq = -1
alors ajouter à $$.vrimp le quadruplet Q
(la variable vdq n'est pas concernée par ce noeud)
(cas c1'-b)

Si (∃ pas vdt dans T de *vtypent* avec vdt = vdq)
alors ajouter à $$.vrimp le quadruplet Q
(la variable vdq n'est pas concernée du tout par ce noeud)
(cas c2'-b)
sinon aucun traitement n'est à effectuer

[10] Dans le cas d'un CP, l'opérateur de sélection T(:x) sera noté SEL(T,x) lors de l'implémentation. Nous avons gardé dans ces notes la notation T(:x) afin de ne pas confondre l'opérateur général de sélection noté également SEL(...) avec la notation particulière dans le cas du CP.

L'implémentation de cet algorithme en fonctions récursives SSL est relativement complexe. Le lecteur intéressé par les détails techniques peut se référer aux annexes; il y trouvera la fonction *majdevrimpet* qui réalise la fonctionnalité de cet algorithme.

La validation de l'implémentation sur le noeud and découle du fait que les différents cas prévus par la théorie sont couverts, et que la manière de les traiter est celle décrite théoriquement.

Le cas du noeud and étant couvert et justifié, il reste à voir les cas des noeuds or et forall, avant de passer à la construction de la formule FOPL correspondante.

4.4.5.3. Le cas d'un noeud or.

La façon de traiter un noeud or est tout à fait symétrique à celle de traiter un noeud and. Tout ce qui a été dit au point précédent reste d'application dans ce cas, à condition de substituer le **CP** par **UNION** et de considérer le sélecteur **AS_** au lieu de _(:). Nous ne nous étendrons pas sur ce cas.

4.4.5.4. Le cas d'un noeud forall.

Le cas du forall se traite de façon équivalente à celle des cas and et or , à ceci près que le connecteur forall n'a qu'un seul fils. Nous ne recevrons donc qu'un seul attribut de ce fils; nous le noterons *explicop.vrimp*. Néanmoins, nous traiterons également le noeud forall en trois passes:

- la première donnera les termes candidats à la production d'une équation implicite,
- la seconde détermine les variables à lier à ce noeud forall,
- la troisième met à jour la construction des équations implicites sur ce noeud.

Voici comment s'effectuent ces étapes.

1. Règle-forall-2-bis

Pour chaque quadruplet Q = (vd , CSEL (ω) , TPω, ε) de *explicop.vrimp*,

si ∃ **dans l'attribut** ***typexpdec*** **SEQ [TPω]**
(i.e. on a la relation "is-seq-of")

alors ajouter à ***vtypent*** **le triplet**
(vd , SEQ [TPω] , ε)

sinon aucun traitement n'est à effectuer

2. Nous conserverons les variables à lier au noeud courant dans l'attribut *varbonend* de la façon suivante:

Règle-forall-2-ter

Pour chaque triplet T = (vd , TP , ε) de *vtypent*

si ε = **-1** (la variable vd était libre localement jusqu'ici)
(cas c1'-a)

alors ajouter vd à ***varbonend***

sinon aucun traitement n'est à effectuer

3. Nous mettrons à jour l'attribut *vrimp* à passer au noeud père du noeud courant forall de la façon expliquée ci-après et en respectant les cas c1'-a , c1'-b , c2'-a et c2'-b décrits ci-dessus.

Nous disposerons à ce moment de
- la liste des variables à lier au noeud courant : *varbonend*, construite à l'étape précédente,
- la liste des quadruplets Q = (vdq , CSEL (ωq) , Tωq , εq),
- la liste des triplets T = (vdt , SEQ [Tωt] , εt) dans *vtypent*.

Règle-forall-2

Pour chaque quadruplet Q de l'attribut *vrimp* du noeud fils,

Si (∃ vdt dans T de *vtypent* avec vdt = vdq)
et (vdq ∈ *varbonend*)
et εq = -1
alors ajouter à $$.vrimp le quadruplet
(vdq , CSEL (ITH (K[11] , ωq)) , SEQ [Tωt] , 1)

(la variable vdq qui était libre est liée à ce noeud)
(cas c1'-a)

Si (∃ vdt dans T de *vtypent* avec vdt = vdq)
et (vdq ∉ *varbonend*)
alors ajouter à $$.vrimp le quadruplet
(vdq , CSEL (ITH (K , ωq)) , SEQ [Tωt] , 1)
(la variable vdq est liée et son équation se construit petit à petit)
(cas c2'-a)

Si (∃ pas vdt dans T de *vtypent* avec vdt = vdq)
alors ajouter à $$.vrimp le quadruplet Q
(la variable vdq n'est pas concernée par ce noeud)
(cas c1'-b et c2'-b)

Pour chaque variable de *vtypent* , générer
LENGTH (seqguide) = LENGTH (ωq)
(on génère obligatoirement l'équation de longueur pour toutes les variables concernées)

Ces trois étapes garantissent le passage correct de l'attribut contenant les équations implicites au travers du noeud forall, tous les cas prévus par la théorie étant couverts de la façon décrite dans la dite théorie.

En résumé, nous avons vu comment on initialise, au niveau des feuilles, l'attribut *vrimp* qui va servir à la construction progressive des équations implicites. Nous savons également comment cet attribut passe et est mis-à-jour correctement

[11] K indique la position de l'élément représentatif dans la séquence guide. Son implémentation, puisque c'est une variable introduite artificiellement, se fait en concaténant au nom de l'élément représentatif le caractère #, ce qui garantit l'unicité de cette variable (l'élément représentatif étant unique).

(c'est-à-dire selon la théorie) sur un des noeuds problématiques (and , or , forall). Il reste à signaler que sur tous les autres noeuds, cet attribut *vrimp* passe sans aucune modification. Finalement cet attribut synthétisé arrive à la racine de l'arbre concerné par l'explicitation de l'opération (*decformop* dans notre syntaxe). Nous allons voir comment on y génère la représentation des équations implicites dans l'attribut *eqimp* et pour terminer comment sera générée la formule FOPL complète correspondant à une spécification.

4.4.6. Construction de la formule FOPL.

4.4.6.1. Comment sont générées les équations implicites.

Pour compléter les équations implicites correspondant au cas des assertions composées, nous procéderons comme dans le cas des assertions simples.

Au noeud *decformop* de notre syntaxe, nous obtenons de nos fils l'attribut *vrimp* dans lequel se trouvent entre autres la représentation des diverses équations implicites pour lesquelles il faut compléter le deuxième élément de chaque triplet par le nom d'une et une seule variable vg du membre de gauche (l'unicité est garantie par la théorie au point II de la définition 4.3.bis). En fait,

Règle-racine-2

Pour chaque quadruplet $Q = (vd , CSEL (\omega) , TP\omega , \varepsilon)$ de *vrimp* ,

$\exists$! vg dans l'attribut *typexpdec* tel que TPG $\equiv$ TPω
et il faut produire (vd , CSEL (vg))

où $(vd, CSEL(vg))$ est une représentation de l'équation implicite $vd = CSEL(vg)$.

Ce résultat, l'ensemble des équations implicites complétées, sera conservé dans un attribut *eqimp* dont la structure est celle d'une liste de couples; un couple est constitué de deux éléments, le premier de type *id* désigne la variable et le second de type *fterme* désigne l'opération de sélection qui est associée à cette variable. Cette liste est désignée par *idfterme*.

Comme nous l'avons vu et justifié ci-dessus, cette représentation conduit bien aux équations implicites recherchées: leur construction et leur finition sont réalisées en conformité avec la théorie.

Une fois synthétisé au noeud *decformop* , cet attribut *eqimp* sera passé en héritage à travers l'entièreté de l'arbre qui a pour racine le noeud *decformop*. Disposant partout dans cet arbre des équations implicites, il suffira de les attacher en conjonction au bon noeud pour produire la formule FOPL correspondant. Nous expliquerons comment cela est réalisé dans la section suivante.

4.4.6.2. Comment est progressivement construite la formule FOPL.

En vue de produire la formule FOPL correspondant à la spécification, nous avons défini un attribut de nom *mafo* dont la structure est précisément celle d'une formule du langage des prédicats du premier ordre, soit *fform* en termes de notre syntaxe. Il suffit de synthétiser, c'est-à-dire de construire progressivement cet attribut des feuilles à la racine, en le complétant uniquement par des éléments de la syntaxe FOPL, et selon le noeud sur lequel on se trouve. En fait, le principe de la transformation est décrit en long et en large au point 4.3.1.2. et la réécriture en SSL se trouve en annexe. Cette réécriture ne pose aucun problème si ce n'est celui de la retranscription (liée à la syntaxe concrète uniquement). Il est peu intéressant de donner tous les détails de celle-ci. La lecture du code du programme donné en annexe suffit pour satisfaire la curiosité du lecteur intéressé.

Il faut cependant faire remarquer que l'écriture des équations implicites au bon endroit dans l'arbre se fait sans difficultés. Nous avons, dans l'étape précédente, précisément conservé sur chaque noeud problématique l'ensemble des variables liées à ce noeud (dans l'attribut *varbonend*). Ceci détermine donc le noeud exact sur lequel il faut greffer l'équation implicite vd = CSEL (vl). Il suffit dès lors de récupérer les équations implicites liées à ces variables dans l'attribut *eqimp*, les ajouter à l'attribut *mafo* en produisant les " [∃ Vd]* nécessaires. Cette opération est réalisée par la fonction *eqimplenfform* que l'on trouvera également en annexe.

Ceci termine l'implémentation de la transformation de RSL en FOPL. Avant de passer à la transformation en Prolog, donnons les quelques petits exemples qui ont servi de jeu de test au moment de la mise en oeuvre de notre outil .

4.5. Exemples.

Dans la spécification de l'énoncé des exemples qui suivent, seule les parties qui nous intéressent pour la transformation seront mentionnées. Que le lecteur ne s'étonne donc pas si cet énoncé ne colle pas à cent pour cent avec la syntaxe des spécifications RSL telles qu'elle est décrite au chapitre 2.

Il faut de plus faire remarquer que la technique étant ce qu'elle est, il a été impossible de générer la décompilation réelle de certains symboles propres à FOPL, ces symboles n'existant pas dans l'outil d'implémentation. C'est ainsi que par convention pûrement personnelle, nous avons représenté

le symbole ∃ par " il existe " ,
le symbole ∀ par " pour tout " ,
le symbole ∨ par " &&" ,
le symbole ∧ par " || ".

4.5.1. Un seul and.

1) Formulation RSL.

```
SPECIFICATION OF OPERATION EXEMPLE_AND_1
FORMAL DESCRIPTION:
   RESULT: res
   ARGUMENTS : arga , argb , argc
   EXPLICITATION OF INPUT-OUTPUT ASSERTION:
         res = OP ( arga , argb , argc )
         →
         (
            res1 = OP1 ( argal , argb , argc1 , inter )
                 with
                      inter = OP3 ( argb )
            and
            res2 = OP2 ( arga2 , argc2 )
            )
```

OUTLINE:

TYPES: res : RES ,
res1 : RES1 ,
res2 : RES2 ,
arga1 : ARGA1 ,
argc1 : ARGC1 ,
inter : INTER ,
argc2 : ARGC2 ,
arga2 : ARGA2 ,
arga : ARGA ,
argb : ARGB ,
argc : ARGC

SPECIFICATION OF OBJECTS TYPE

EXPLICITATION OF TYPE STRUCTURE :

RES -> CP [RES1 , RES2]
ARGA -> CP [ARGA1 , ARGA2]
ARGB -> ARGB1
ARGC -> CP [ARGC1 , ARGC2]
ARGA2 -> INT
ARGC2 -> DATE
ARGC1 -> NAT
ARGB1 -> INT
ARGA1 -> CHAR
RES2 -> BOOL
RES1 -> CHARST
INTER -> BOOL

2) Représentation graphique de l'opération.

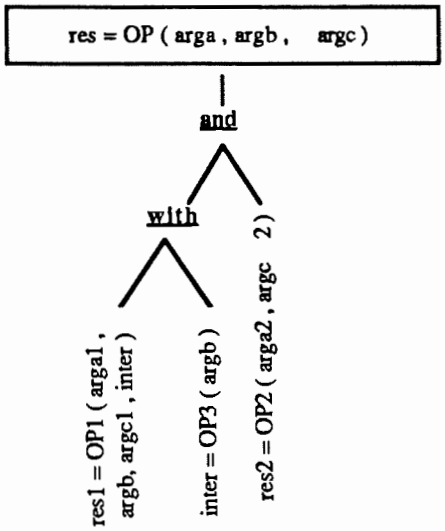


fig. 4.8. Exemple d'un seul and.

3) Formules FOPL générées.

Res = op (Arga , Argb , Argc)

<=>

(il existe Res1)
(il existe Arga1)
(il existe Argc1)
(il existe Res2)
(il existe Arga2)
(il existe Argc2)
Res1 = op1 (Arga1 , Argb , Argc1 , Inter)
&&
  (il existe Inter)
  Inter = op3 (Argb)
&&
Res2 = op2 (Arga2 , Argc2)
&&

Argc2 = sel (date , Argc)
&&
Arga2 = sel (int , Arga)
&&
Res2 = sel (bool , Res)
&&
Argc1 = sel (nat , Argc)
&&
Arga1 = sel (char , Arga)
&&
Res1 = sel (charst , Res)

4.5.2. Un seul or.

1) Formulation RSL.

SPECIFICATION OF OPERATION EXEMPLE_OR_1
FORMAL DESCRIPTION:
RESULT: res
ARGUMENTS : arga , argb , argc
EXPLICITATION OF INPUT-OUTPUT ASSERTION:
res = OP (arga , argb , argc)
→
(
res1 = OP1 (arga1 , argb , argc1 , inter)
with
inter = OP3 (argb)
or
res2 = OP2 (arga2 , argc2)
)

OUTLINE:
TYPES: res : RES ,
res1 : RES1 ,
res2 : RES2 ,
arga1 : ARGA1 ,
argc1 : ARGC1 ,

inter : INTER ,
argc2 : ARGC2 ,
arga2 : ARGA2 ,
arga : ARGA ,
argb : ARGB ,
argc : ARGC

SPECIFICATION OF OBJECTS TYPE
EXPLICITATION OF TYPE STRUCTURE :
RES -> UNION [RES1 , RES2]
ARGA -> UNION [ARGA1 , ARGA2]
ARGB -> ARGB1
ARGC -> UNION [ARGC1 , ARGC2]
ARGA2 -> INT
ARGC2 -> DATE
ARGC1 -> NAT
ARGB1 -> INT
ARGA1 -> CHAR
RES2 -> BOOL
RES1 -> CHARST
INTER -> BOOL

2) Représentation graphique de l'opération.

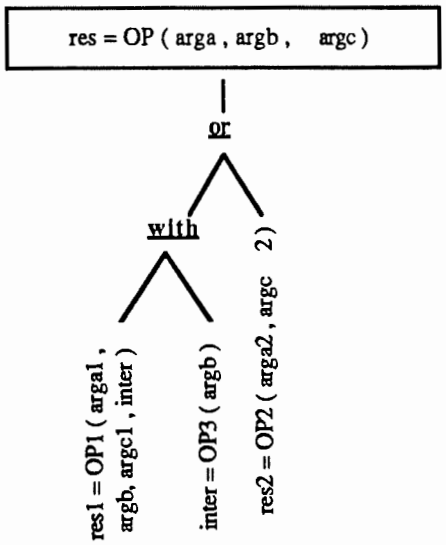


fig. 4.9. Exemple d'un seul or.

3) Formules FOPL générées.

Res = op (Arga , Argb , Argc)

<=>

(il existe Res1)
(il existe Arga1)
(il existe Argc1)
(il existe Res2)
(il existe Arga2)
(il existe Argc2)
Res1 = op1 (Arga1 , Argb , Argc1 , Inter)
&&
  (il existe Inter)
  Inter = op3 (Argb)
||
Res2 = op2 (Arga2 , Argc2)
&&
Argc2 = as (date , Argc)
&&
Arga2 = as (int , Arga)
&&
Res2 = as (bool , Res)
&&
Argc1 = as (nat , Argc)
&&
Arga1 = as (char , Arga)
&&
Res1 = as (charst , Res)

4.5.3. Un seul forall.

1) Formulation RSL.

```
SPECIFICATION OF OPERATION EXEMPLE_FORALL_1
FORMAL DESCRIPTION:
   RESULT: res
   ARGUMENTS : arga , argb , argc
   EXPLICITATION OF INPUT-OUTPUT ASSERTION:
         res = OP ( arga , argb , argc )
         →
         (
            forall arge : IN ( arge , arga ) : res1 = OP1 ( arge, argb,argc )
         )
OUTLINE:
   TYPES: res   : RES ,
          res1 : RES1 ,
          res2  : RES2 ,
          arga1  : ARGA1 ,
          argc1  : ARGC1 ,
          inter  : INTER ,
          argc2  : ARGC2 ,
          arga2  : ARGA2 ,
          arga  : ARGA  ,
          argb  : ARGB ,
          argc  : ARGC

SPECIFICATION OF OBJECTS TYPE
   EXPLICITATION OF TYPE STRUCTURE :
          RES  -> SEQ [ RES1 ]
          ARGA -> SEQ [ ARGA1 ]
           ARGB -> ARGB1
           ARGC -> SEQ [ ARGC1 ]
           ARGC1 -> NAT
           ARGB1 -> INT
           ARGA1 -> CHAR
```

RES1 -> CHARST
ARGE -> BOOL

2) Représentation graphique de l'opération.

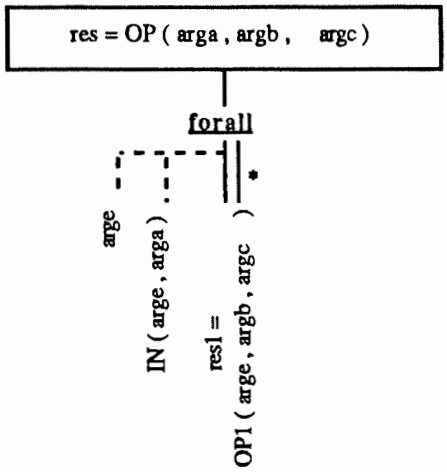


fig. 4.10. Exemple d'un seul forall.

3) Formules FOPL générées.

Res = op (Arga , Argb , Argc)

<=>

length (Arga) = length (Res)
&&
(pour tout Arge)
in (Arge , Arga) = true
=>
(il existe Arge#)
Arge = ith (Arge# , Arga)
&&
(il existe Res1)
Res1 = op1 (Arge , Argb , Argc)
&&
Res1 = ith (Arge# , Res)

4.5.4. Un exemple plus complexe.

1) Formulation RSL.

```
SPECIFICATION OF OPERATION EXEMPLE_DEMO
FORMAL DESCRIPTION:
   RESULT: res
   ARGUMENTS : arga , argb , argc, argd , arge , argf
   EXPLICITATION OF INPUT-OUTPUT ASSERTION:
         res = OP ( arga , argb , argc , argd , arge , argf , seg1 , seq2 )
         →
         (
           ( res1 = OP1 ( arga1 , argb1 , argc1 , inter , argf1 )
            and
           res2 = OP2 ( argb1 , argc2 , inter , argf2 )
           with
             inter = OP3 ( argb1 ) )
          or
           res4 = ( arga1 , argb2 , arge , argf3 )
         )
         and
          forall argc3 : in ( argc3 , seq1 ) :
                (
                 ( forall argd1 : in (argd1 , seq2 ) :
                              res5 = OP5 ( arga2 , argc3 , argd1 ) )
                  and
                     res6 = OP6 ( argc3 , arge )
                )

OUTLINE:
     TYPES: res : RES ,
            res1 : RES1 ,
             res2 : RES2 ,
             res4 : RES4 ,
             res5 : RES5 ,
             res6 : RES6 ,
             arga1 : ARGA1 ,
```

argb1 : ARGB1 ,
argc1 : ARGC1 ,
inter : INTER ,
argf1 : ARGF1
argc2 : ARGC2 ,
argf2 : ARGF2 ,
argb2 : ARGB2 ,
argf3 : ARGF3 ,
argc3 : ARGC3 ,
seq1 : SEQ1 ,
argd1 : ARGD1 ,
seq2 : SEQ2 ,
arga2 : ARGA2 ,
arga : ARGA ,
argb : ARGB ,
argc : ARGC ,
argd : ARGD ,
arge : ARGE ,
argf : ARGF ,

SPECIFICATION OF OBJECTS TYPE

EXPLICITATION OF TYPE STRUCTURE :

RES -> CP [UNION [CP [RES1 , RES2] , RES4] ,
SEQ [CP [SEQ [RES5] , RES6]]]

ARGA -> CP [ARGA1 , ARGA2]

ARGB -> UNION [ARGB1 , ARGB2]

ARGC -> CP [CP [ARGC1 , ARGC2] , SEQ [ARGC3]]

ARGD -> SEQ [SEQ [ARGD1]]

ARGF -> UNION [CP [ARGF1 , ARGF2] , ARGF3]

ARGA2 -> SEQ [INT]

SEQ2 -> SEQ [DATE]

ARGD1 -> CHAR

SEQ1 -> SEQ [DATE]

ARGC3 -> SEQ [CHARST]

ARGF3 -> SET [INT]

ARGB2 -> SEQ [NAT]

ARGF2 -> SET [CHAR]

ARGC2 -> SET [SET [DATE]]
ARGF1 -> SET [DATE]
ARGC1 -> SET [NAT]
ARGB1 -> SET [SET [INT]]
ARGA1 -> SET [SET [CHAR]]
RES6 -> NAT
RES5 -> SET [CHARST]
RES4 -> CHAR
RES2 -> DATE
RES1 -> CHARST
ARGE -> BOOL
INTER -> SET [BOOL]

2) Représentation graphique de l'opération.

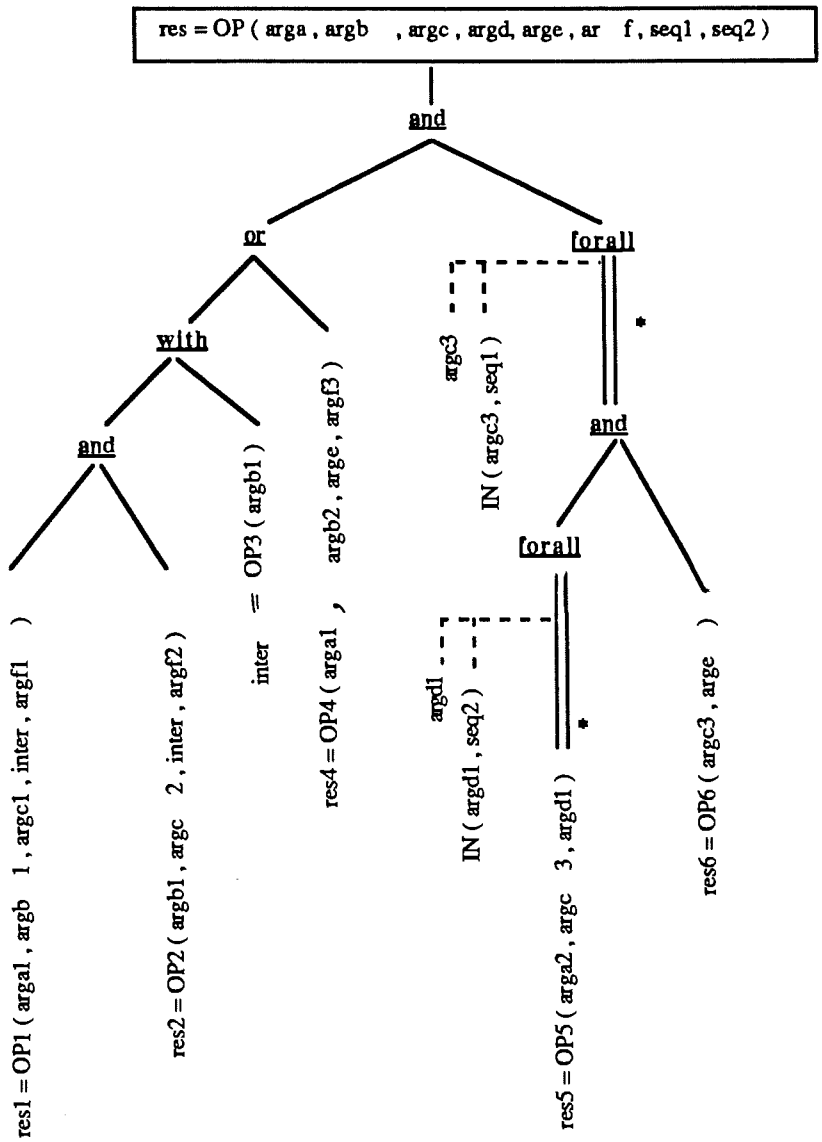


fig. 4.11. Exemple complexe.

3) Formules FOPL générées.

Res = op (Arga , Argb , Argc , Argd , Arge , Argf , Seq1 , Seq2)

<=>

(il existe Arga1)
(il existe Arga2)

```
( il existe Argb1 )
( il existe Res4 )
( il existe Argb2 )
( il existe Argf3 )

  ( il existe Res1 )
  ( il existe Argc1 )
  ( il existe Argf1 )
  ( il existe Res2 )
  ( il existe Argc2 )
  ( il existe Argf2 )
  Res1 = op1 ( Arga1 , Argb1 , Argc1 , Inter , Argf1 )
&&
  Res2 = op2 ( Argb1 , Argc2 , Inter , Argf2 )
&&
  ( il existe Inter )
      Inter = op3 ( Argb1 )
&&
   Argf2 = sel ( set ( char ) , as ( cp ( set ( date ) , set ( char ) ) , Argf ) )
&&
   Argc2 = sel(set (set (date ) ) , sel(cp (set (nat ),set (set date ) ) ) , Argc ) )
&&
   Res2=sel(date,as(cp(charst,date),sel(union(cp(charst, date), char),Res)))
&&
   Argf1 = sel ( set ( date ) , set ( char ) ) , Argf  ) )
&&
   Argc1 = sel ( set ( nat) , sel ( cp ( set ( nat ) , set ( set ( date ) ) ) , Argc))
&&
  Res1=sel(charst,as(cp(charst,date),sel(union(cp(charst,date),char),Res)))
||
   Res4 = op4 ( Arga1 , Argb2 , Arge , Argf3 )
&&
   Argf3 = as ( set ( int ) , Argf )
&&
   Argb2 = as ( seq ( nat ) , Argb )
&&
```

```
    Res4 = as ( char, sel ( union ( cp ( charst,date ) , char ) ,Res ) )
  &&
    Argb1 = as ( set ( set ( int ) ) , Argb )
&&
  length ( seq1 ) = length ( sel ( seq ( cp ( seq ( set ( charst ) ) , nat ) ), Res ) )
&&
  ( pour tout Argc3 )
  in ( Argc3 , Seq1 ) = true
  =>
    ( il existe Argc3# )
    Argc3 = ith ( Argc3# , Seq1 )
  &&
    ( il existe res6 )
    length ( seq2 ) = length(sel (seq(set (charst) ) ith ( Argc3# , sel ( seq ( cp (
                                        seq ( set ( charst ) ), nat ) ), Res ) ) ) )
  &&
    ( pour tout Argd1 )
    in ( Argd1 , Seq2 ) = true
    =>
     ( il existe Argd1# )
     Argd1 = ith ( Argd1# , Seq2 )
  &&
      ( il existe Res5 )
      Res5 = ith ( Argd1#, sel ( seq ( set ( charst ) ), ith (Argc3#,sel ( seq(cp (
                                        seq ( set ( charst ) ) , nat ) ),Res ) ) ) )
  &&
       Res6 = op6 ( Argc3 , Arge )
  &&
       Res6=sel(nat,ith(Argc3#,sel(seq(cp(seq(set( charst ) ), nat ) ), Res ) ) )
  &&
        Arga2 = sel ( seq ( int ) , Arga )
  &&
        Arga1 = sel ( set ( set ( char ) ) , Arga )
```

Chapitre 5

Deuxième étape de la transformation : de la théorie FOPL au code Prolog

---

5.1. Introduction.

Dans ce chapitre, nous allons voir comment construire des procédures Prolog à partir de la transformation générée dans le langage des prédicats du premier ordre FOPL et explicitée dans la première phase de la construction. La construction de ces procédures Prolog tiendra compte du mécanisme d'exécution de ce langage d'une part et du mode d'utilisation souhaitée du prototype ainsi généré d'autre part.

La section 5.2. présentera les idées théoriques sous-jacentes, tandis que la section 5.3. donnera une description intuitive du principe de transformation. La section 5.4. parlera de façon assez brève de la validation de la transformation; la section 5.5 quant à elle exposera comment nous avons réalisé l'outil de transformation et montrera en quoi la théorie nous a servi de fil conducteur.

Signalons dès à présent que la partie théorique de ce chapitre est fortement inspirée de [HABRA 89] .

5.2. Idées théoriques sous-jacentes.

5.2.1. Idées de base.

La première phase de transformation a produit un ensemble d'équivalences de la forme " Res = op (Arg1, ... , Argn) <=> FF (d-eqs) ".

Le but de cette étape-ci est de transformer chacune de ces équivalences FOPL en clauses Prolog. L'ensemble de clauses déduites par le procédé que nous allons décrire constituera le prototype Prolog. Les procédures Prolog qui traduisent les

opérations de la bibliothèque seront pré-programmées une fois pour toutes à partir de leur spécification algébrique donnée dans la bibliothèque. Ceci fait l'objet d'un mémoire parallèle.

Rappelons que lors de la transformation, une attention particulière aura été portée au fait de conserver la traçabilité du programme produit par rapport aux spécifications de départ. Le prototype généré doit en effet servir à valider les spécifications initiales et permettre la mise au point de celles-ci. Une fois le prototype soumis à l'utilisateur, ce dernier, dès qu'une anomalie semble apparaître, doit pouvoir retrouver l'opération et/ou les types d'objet à redéfinir. Il est donc impératif de garder dans le prototype généré une trace des noms d'opérations et des noms d'objets fournis initialement. A cette fin, nous éviterons également et au maximum d'introduire de nouveaux noms d'opérations et/ou de types.

5.2.2. Choix d'une représentation Prolog adéquate.

Avant d'expliquer comment se déroulera la phase de transformation en Prolog, nous allons préciser quelle représentation a été choisie.

Le prototype que nous allons générer sera formé d'un ensemble de clauses Prolog qui se conforment à la syntaxe concrète de la version choisie de Prolog, celle du Prolog d'Edinburg. Cet ensemble de clauses formera un programme Prolog exécutable. Par rapport à la fin de la première étape de la transformation, la seule adaptation syntaxique qui devra être envisagée lors du processus de transformation est celle d'enlever les quantificateurs universels et existentiels et de bien représenter les prédicats et les foncteurs.

Nous décrirons explicitement cette adaptation dans le point suivant.

Signalons dès à présent les conventions de représentation Prolog qui seront adoptées:

- Les variables apparaissant dans l'assertion associée à une explicitation deviennent des variables Prolog et gardent leur nom initial : les noms des variables commencent par une lettre majuscule (par exemple Vx, Vd, Vdx, ...) et se terminent par des lettres minuscules.

- Les noms d'opérations et les noms de prédicats seront complètement écrits en lettres minuscules (par exemple op, pre, constuple, ...).

- Les opérations apparaissant dans l'assertion associée à une explicitation deviennent des foncteurs Prolog. Ces opérations conservent leur nom initial et leur arité .

- Le prédicat " = " est vu comme une relation entre deux termes et est utilisé dans une notation infixée: " t1 = t2 ". A ce prédicat, on associe la sémantique supplémentaire " Un terme, disons t1, est vu comme l'évaluation fonctionnelle de l'autre, disons t2 ". [HABRA89] prouve que cette sémantique supplémentaire sera respectée. Dans les conditions d'utilisation des prototypes, dans la version Prolog générée, le prédicat " t1 = t2 " est devenu le litéral " eval (t1, t2) ". Eval sera le seul symbole de prédicat introduit dans le prototype Prolog déduit.

5.2.3. Notations et conventions.

Tout comme dans le chapitre précédent, nous allons introduire quelques conventions de notations utilisées pour expliquer le processus de transformation.

- Les opérateurs de sélection issus de l'étape précédente sont notés de façon générique par " sel (...) ". Une composition quelconque de ceux-ci sera notée " csel (...) ".

  De plus, dans ce chapitre, nous dénoterons les opérateurs de composition par " comp (...) " et une composition quelconque de ceux-ci par " ccomp (...) ".

  Parmi les opérateurs de composition utilisés, signalons

  le constructeur de séquence:
  " cons (tête, queue) " de type SEQ (...),

  le constructeur de tuple:
  " constuple (TYPE1: var1 , TYPE2: var2 , ...) "
  de type PC (... , TYPE1, ... , TYPE2, ...),

  le constructeur d' UNION
  " inj_{TYPE} (var) " de type UNION (..., TYPE, ...).

- Selon les notations conventionnelles du chapitre précédent, Tr, Trx, ... dénotent des termes de la forme " Csel (..., Vd, ...) " ou "Csel (..., Vg, ...) " et sont appelés des termes de sélection.

De la même façon, dans ce chapitre, (Ctr, Ctrx, ...) dénotent des termes de la forme " Ccomp (..., Vd, ...) " et sont appelés des termes de composition.

- En cours de construction du principe de transformation, quelques nouvelles variables qui ne sont pas des variables de l'assertion initiale " g-eq <=> FF (d-eqs)" seront introduites. Nous noterons ces variables par Nv, Nvx, ... Ces nouvelles variables apparaîtront essentiellement dans les termes de composition.

- La notation " d-eq " représente une équation simple telle que "Res = op (Arg1, ..., Argn) ", " Inter = opinter (Arg1, ..., Argn) " ou une opération simple telle que " pred (Arg1, ..., Argn) = true ". Cette notation n'inclut pas les équations implicites générées lors de la première phase de transformation.

Ces quelques notations étant prises, voyons les idées de base sous-jacentes au principe de la transformation.

5.3. Description intuitive du procédé de transformation.

Le schéma de transformation qui est exposé çi-après est extrait de [HABRA89] où il est présenté en détails. Nous nous contenterons dans ce chapitre d'en présenter les grandes lignes, le but de notre travail étant de produire un prototype Prolog à partir de la théorie développée.

5.3.1. Les trois étapes de la transformation.

La transformation en procédures Prolog se fera en trois étapes.

Etape 1. Le but de cette étape est de présenter l'équivalence FOPL sous forme clausale. Pour y parvenir, deux types de transformation sont exécutées:

1. Chaque équivalence " g-eq <=> FF (d-eqs) " est transformée pour remplacer les équations de sélection par des équations de composition. Ainsi une équation de sélection de la forme " Vd = Sel (... Vg) ", lie une variable quelconque du membre de droite " Vd " à la variable " Vg " qui lui correspond dans le membre de gauche de l'assertion d'explicitation. L'idée de cette étape est de remplacer de telles équations équivalentes de la forme "Vg = Comp (... Vd ...) ", où " comp " est un opérateur de composition (constuple, inj, cons, ...). Ceci permettra de simplifier ces équations par réécriture de termes dans l'étape 3.

2. Les quantificateurs " ∀ " dans la formule FF (d-eqs) générée par la première transformation sont éliminés par l'introduction de définitions récursives. Rappelons qu'en fait ces quantificateurs sont issus du connecteur forall de la spécification écrite en RSL. Ce connecteur exprime comment une séquence résultat est construite à partir de l'application d'une opération donnée sur des arguments qui sont les éléments d'une (des) séquence(s) . Intuitivement donc, à ce connecteur correspond également un opérateur de composition qui est une forme généralisée de l'opérateur du second ordre "map" connu en langage fonctionnel. Celui-ci ne pouvant être exprimé par un opérateur du premier ordre, une formulation récursive est utilisée.

Etape 2. Le but de cette étape est de mettre la transformation sous forme de clauses de Horn.

En fait, la première étape a fourni un ensemble d'équivalences, chacune correspondant à une des formules FF (d-eqs), et qui incluent les conjonctions et les disjonctions FOPL. La deuxième étape va " plier " ces équivalences et éliminer les disjonctions. Les équivalences " <=> " sont transformées en implications " <= ". Deux techniques de pliage sont utilisées, à savoir:

- Remplacer des implications comme

wffo <= C0 ∧ wff1
wff1 <= C1 ∧ wff2
.....
wffn <= Cn ∧ wff

par une implication unique

wff0 <= C0 ∧ C1 ∧ ... ∧ Cn ∧ wff

- Remplacer des conjonctions comme

Vg = comp1 (V1) ∧ V1 = comp2 (V2) ∧ ... ∧ Vn = comp (V)

par une équation unique

Vg = comp1 (comp2 (... comp (V) ...))

Etape 3: Le but de la troisième étape est d'adapter les clauses issues de la deuxième étape aux particularités du langage Prolog.

Ceci comprend

- l'adaptation des clauses à la syntaxe concrète Prolog,

- l'adaptation des clauses à la sémantique opérationnelle Prolog en vue d'assurer la terminaison du programme final. Ce but est atteint entre autres en réarrangeant les littéraux à l'intérieur des clauses et en réarrangeant les clauses elles mêmes.

Nous venons de voir les grandes lignes du principe de transformation. La section suivante va exposer pour chaque noeud la transformation théorique qui lui sera appliquée. De nouveau cette section s'inspire fortement de [HABRA89] dans lequel le lecteur intéressé pourra trouver tous les détails nécessaires.

5.3.2. Le cas des assertions simples.

Cette section donne dans une description intuitive les grandes idées du processus de transformation en Prolog. Son but essentiel dans le cadre de ce travail est de mettre en évidence la partie théorique qui servira de support aux diverses décisions qui ont été prises lors de l'implémentation.

Après avoir rappelé les diverses formes issues de la transformations de RSL en FOPL, nous donnerons pour chacune d'elles la forme clausale normale, ensuite, le passage aux clauses de Horn et enfin l'adaptation au langage Prolog.

5.3.2.1. Rappels.

Le point de départ de la transformation en Prolog est l'équivalence FOPL

"g-eq <=> FF (d-eqs) "

issue de l'assertion RSL

" g-eq -> d-eqs "

et qui peut prendre une des formes suivantes.

(1) d-eq

(2) (∃ Inter) d-eq-a ∧ Inter = opinter (Arg1, ..., Argn)

(3) (∃ Vda)* (Vda = tpda (: Vg))* ∧ d-eq-a
∧
(∃ Vdb)* (Vdb = tpdb (: Vg))* ∧ d-eq b

(4) (∃ Vda)* (Vda = as_{tpda} (Vg))* ∧ d-eq-a
∨
(∃ Vdb)* (Vdb = as_{tpdb} (Vg))* ∧ d-eq-b

(5) ∀ Elem (in (Elem, Seq) = true
=> ∃ K (∃ Vda)* Elem = ith (K, Seq)
∧ (Vda = ith (K, Vg))*
∧ d-eq-a)
∧ (length (Seq) = length (Vg))*

(6) pred (Arg1, ..., Argn) = true ∧ d-eq-a

(7) (pred (Arg1, ..., Argn) = true ∧ d-eq-a)
∨
(pred (Arg1, ..., Argn) = false ∧ d-eq-b)

Dans la suite, chacune de ces formes sera reprise par son numéro et ne sera plus explicitée.

5.3.2.2. Réecriture des équations de sélection en équations de composition.

Dans cette partie, nous visons deux objectifs:

- Transformer les équations de sélection de la forme " Vd = Csel(Vg) " en des équations de composition équivalentes de la forme " Vg = Ccomp (... Vd ....) ",

- Supprimer les connecteurs ∀ en introduisant une forme récursive.

Partant de l'équivalence " g-eq <=> FF (d-eqs) " avec FF (d-eqs) ayant une des formes ((1) à (7)) ci-dessus, nous voulons arriver à une équivalence de la

forme g-eq <=> CN1 ν CN2 ν .... ν CNn appelée équivalence normale, où chaque CNi est une conjonction normale, c'est-à-dire de la forme

$$(\exists\ Vd)^* \quad (\exists\ Nv)^* \quad (Vg = comp (..., Vd, Nv, ...))^*$$
$$\wedge\ (d\text{-}eq)^* \ \wedge\ g\text{-}eq\ \{ (Nv / vg)^* \}^1$$

Décrivons ce qu'il en est pour les formes de (1) à (7).

1. Cas de l'assertion simple.

Nous avons directement l'équivalence normale

" g-eq <=> d-eq "

2. Cas de l'explicitation with.

Nous avons également directement l'équivalence normale

" g-eq <=> (∃ Inter) d-eq-a ∧ Inter = opinter (Arg1, ..., Argn) "

3.Cas de l'explicitation and.

L'équivalence normale associée est donnée par

" g-eq <=> (∃ Vda)* (∃ Vdb)* (∃ Nv)*
(Vg = constuple (..., Vda, Vdb, Nv, ...))*
∧ d-eq-a ∧ d-eq-b "

4. Cas de l'explicitation or.

L'équivalence normale associée est donnée par

" g-eq <=> (∃ Vda)* (Vg = inj (Vda))* ∧ d-eq-a
ν (∃ Vdb)* (Vg = inj (Vdb))* ∧ d-eq-b "

[1] Rappelons que la notation { Nv / Vg } représente la substitution de la variable Vg par la variable Nv.

5. Cas de l'explicitation forall.

L'équivalence normale associée est donnée par

" g-eq <=>
(Seq = empty ∧ (Vg = empty)*)
∨
(∃ Elem ∃ Queueseq (∃ Vda)* (∃ Queue)*
Seq = cons (Elem, Queueseq) ∧ (Vg = cons (Vda, Queue))*
∧ d-eq-a
∧ g-eq {Queueseq / Seq } { (Queue / Vg)* } "

6. Cas de l'explicitation if.

Nous avons directement l'équivalence normale

" g-eq <=> pred (Arg1, ..., Argn) = true ∧ d-eq-a "

7. Cas de l'explicitation if-otherwise.

Nous avons directement l'équivalence normale

" g-eq <=> pred (Arg1, ..., Argn) = true ∧ d-eq-a
∨ pred (Arg1, ..., Argn) = false ∧ d-eq-b "

Nous avons donc une équivalence normale pour chacune des formes (1) à (7).

5.3.2.3. Mise sous forme de clauses de Horn.

L'équivalence " <=> " n'étant pas traduisible en Prolog, le sens " <= " de l'explicitation a été essentiellement retenu puisque l'objectif poursuivi est de réduire un problème en des sous-problèmes plus simples, ce qui correspond au mode de travail du moteur d'inférence de Prolog. L'objectif à établir est réduit à des sous-objectifs, et cela récursivement jusqu'aux faits.

Dans le cas qui nous occupe, cette mise sous forme de clauses de Horn est directe. On utilise donc seulement la partie " si " de l'équivalence " g-eq <=> CN1 ν CN2 ν .... ν CNn ", c'est-à-dire l'implication " g-eq <= CN1 ν CN2 ν .... ν CNn ", et on peut directement déduire un ensemble de clauses

" g-eq <= CN1 "
" g-eq <= CN2 "
....
" g-eq <= CNn "

ayant la forme requise, à savoir

" g-eq <= (∃ Vd)* (∃ Nv)* (Vg = comp (... , Vd , Nv, ...))*
∧ (d-eq)* ∧ g-eq { (Nv / Vg)* } "

Cet ensemble de clauses de Horn va être adapté à la syntaxe concrète Prolog et quelques transformations classiques lui seront appliquées en vue de garantir la terminaison.

5.3.2.4. Adaptation au langage Prolog.

Nous appliquerons d'abord quelques substitutions d'égalité sur les clauses, nous adapterons ensuite ces clauses à la syntaxe concrète Prolog, et finalement nous arrangerons les clauses à l'intérieur du programme et les littéraux à l'intérieur des clauses.

A. Substitution d'égalité.

Dans une clause, l'équation du membre de gauche g-eq a la forme " Res = op (Arg1, ..., Argn) où les variables Res et Arglist sont celles notées Vg. Ainsi, les équations de composition (Vg = comp (... , Vd , Nv, ...))* peuvent être enlevées en appliquant les substitutions correspondantes dans l'équation g-eq. Finalement, la clause normale

" g-eq <= (∃ Vd)* (∃ Nv)* (Vg = comp (... , Vd , Nv, ...))*
∧ (d-eq)* ∧ g-eq { (Nv / Vg)* } "

peut être écrite

" (∀ Vd)* (∀ Nv)* (g-eq { (comp (... , Nv, Vr , ...) / Vg)* }
<= (d-eq)* ∧ g-eq { (Nv / Vg)* } "

B. Adaptation à la syntaxe concrète.

Les transformations syntaxiques sont les suivantes:

- Les noms d'opérations prédéfinies de la librairie sont transformés en leur nom de foncteur correspondant.
  Ainsi,

type (: Objet)	deviendra	**sel (type, Objet)**
inj_{type} (Objet)	deviendra	**inj (type, Objet)**
as_{type} (Objet)	deviendra	**as (type, Objet)**

  Les autres opérations de la librairie restent inchangées, entre autres constuple (type1: objet1, type2: objet2, ...), le constructeur de listes [ ] .

- Les noms d'opérations fournis par le spécifieur sont transformés en foncteur Prolog ayant le même nom.

- Les quantificateurs deviennent implicites.

- L'équation atomique apparaissant dans les expressions terminales"Res = op (Arg1, ... , Argn) " donne le littéral Prolog " eval (Res , op (Arg1, ... , Argn))".

- L'équation atomique introduite par un connecteur with de la forme " Inter = opinter (Arg1, ..., Argn) " donne le littéral Prolog " eval (Inter, opinter (Arg1, ... , Argn)) ".

- L'équation atomique exprimant des préconditions " pre (Arg1, ... , Argn) = true " ou " pre (Arg1, ... , Argn) = false " donne le littéral Prolog " eval (true , pre (Arg1, ..., Argn)) " ou " eval (false , pre (Arg1, ..., Argn)) ".

- Un nom de variable généré par le processus de transformations lui-même, Nv, et apparaissant seulement une fois dans une clause est remplacé par la variable anonyme " _ " .

C. Réarrangement des clauses dans le programme.

Le réarrangement des clauses Prolog se passera comme suit:

- les clauses correspondant à la même opération initiale RSL sont groupées.

- Parmi les clauses correspondant à la même opération initiale RSL, la clause ayant une forme non-récursive est placée avant toutes les clauses ayant une forme récursive, s'il y en a.

D. Réarrangement des littéraux dans les clauses.

Le réarrangement des littéraux dans les clauses se déroulera comme suit:

- Dans une clause récursive issue d'une assertion forall, le littéral exprimant l'appel récursif est placé à la fin de la clause. Le moteur d'inférence Prolog travaillant de gauche à droite, ceci garantit l'évaluation de tous les littéraux à chaque appel récursif.

- Dans une clause issue d'une assertion with, le littéral introduisant l'objet intermédiaire est placé avant tous les autres littéraux. Le moteur d'inférence Prolog travaillant de gauche à droite, ceci garantit l'évaluation du littéral introduisant l'objet intermédiaire avant l'évaluation des autres littéraux de la clause.

- Dans une clause issue d'une assertion conditionnelle, le littéral exprimant les préconditions est placé avant tous les autres littéraux. Le moteur d'inférence Prolog travaillant de gauche à droite, ceci garantit l'évaluation du littéral

correspondant à la précondition avant de passer à l'évaluation de tous les autres littéraux.

5.3.3. Le cas des assertions composées.

Le cas des assertions composées sera exposé de façon relativement succinte. Le lecteur intéressé trouvera le complément d'informations nécessaires dans [HABRA89] .

L'idée sous-jacente est que la transformation décrite dans le cas des assertions simples peut aussi être appliquée comme transformation dans le cas des assertions composées, moyennant un jeu de substitution judicieux.

En effet, la généralisation du principe de la transformation au cas des assertions composées ne pose pratiquement aucun problème, si ce n'est dans le cas du forall: il faut dans ce cas tenir compte des appels récursifs. C'est ainsi qu'à chaque itération, les formules récursives venant des noeuds fils seront réécrites comme formules récursives du noeud représentant le sous-arbre considéré.

Le processus de transformation est décrit par un ensemble de règles qui s'appliquent à des sous-arbres dont les fils ont déjà été substitués par des clauses normales et qui produisent comme résultat une ou plusieurs clauses normales également. En fait, les clauses en cours de construction sont représentées par quatre séquences :

1. une séquence de N termes de sélection " [Trx_1 , ... , Trx_N] ". Ces termes sont en fait les termes déduits de la première phase de transformation (RSL vers FOPL),

2. une séquence de N termes de composition " [$Ctrx_1$, ... , $Ctrx_N$] ",

3. une séquence représentant chaque équation atomique " d-eq ",

4. une séquence représentant une substitution multiple { θ_1 , ... , θ_N }, où θ_i est une substitution concernant le terme Trxi correspondant et qui aura la forme {NCtri/Trxi} ou la forme {Trxi/Trxi}[2] (cette notation signifie que le

[2] Cette substitution est utilisée pour standardiser les substitutions multiples en une correspondance "un-à-un" entre les substitutions θ_i et la séquence des termes de sélection Trxi.

terme Trxi n'est pas concerné par une substitution). L'idée de cette substitution est de remplacer chaque terme de sélection Trxi par un terme de composition Ctrxi.

En réalité, ces quatre séquences représentent une clause de la forme:

Trx1 = op (Trx2 , ... , Trxn) **(seq.1)**
<=
Trx1 = Ctrx1 ∧ Trx2 = Ctrx2 ∧ ... Trxn = Ctrxn **(seq.2)**

∧ d-eq-a ∧ d-eq-b ∧ ... **(seq.3)**

∧ Trx1 = op (Trx2 , ... , Trxn) {θ1 , θ2 , ... , θn } ∧ ... **(seq.4)**

dans lesquelles op est le nom de l'opération correspondant au sous-arbre en question.

Il reste à voir comment ces éléments évoluent sur les différents noeuds d'un arbre. Pour ce faire, nous allons passer en revue ces noeuds et exposer brièvement la façon de tenir à jour ces éléments constituant les clauses.

(a) **cas-feuilles**

a1- la séquence [Trx1 , ... , Trxn] est celle associée aux feuilles lors de la transformation FOPL. Dans ce cas, les termes sont des variables simples.

a2- la séquence [Ctrtx1 , ... , Ctrxn] est une copie de la séquence [Trx1 , ... , Trxn].

a3- l'équation atomique "d-eq" est la seule et reste telle quelle dans la nouvelle clause.

a4- la séquence de substitution est vide.

(b) cas-with

b1- la séquence [Trx1 , ... , Trxn] est celle associée au noeud with lors de la transformation FOPL.

b2- un terme de composition Ctrxi est une simple copie du terme de composition correspondant venant de l'explicitation du with (il n'y a pas d'équartion de décomposition sur un noeud with).

b3- l'équation atomique correspondant à la formule atomique introduisant l'objet interrmédiaire se place devant les équations atomiques venant du sous-arbre associé au noeud with dans la nouvelle clause.

b4- la séquence de substitution est modifiée comme suit: chaque substitution du fils {θaj ... θm} produit une substitution du père {θx1 ... θxn} telle que, pour i de 1 à n,

*) si Trxi est un terme composé du sous-arbre associé au noeud with, alors θxi est la substitution identité {Trxi/Trxi},

*) si Trxi est un terme composé de la formule atomique introduisant l'objet intermédiaire, disons Traj, alors θi est la substitution {Traj{θaj}|Trxi}.

(c) cas-and

c1- la séquence [Trx1 , ... , Trxn] vient directement de la séquence des termes composés associée au noeud and lors de la transformation FOPL.

c2- chaque terme de composition Ctrxi est déduit de l'équation de composition de son terme de sélection Trxi correspondant: ainsi si l'équation de composition est "Trxi = comp (Traj , Trbk , Nv , ...)", alors le terme de composition correspondant Ctrxi sera " comp (Traj , Trbk , Nv , ...){Ctraj/Traj,Ctrbk/Trbk}", où "comp" est soit "constuple", soit "=".

c3- les équations atomiques "d-eq-a" et "d-eq-b" deviennent l'équation atomique "d-eq-a ∧ d-eq-b" dans la nouvelle clause.

c4- chaque substitution θxi du sous-arbre de gauche (respectivement de droite) du noeud and produit une nouvelle substitution sur le noeud and, déduite de l'équation de composition du terme Trxi et de la valeur initiale de cette substitution. Ainsi, si l'équation de composition est

"Trxi = comp (Traj , Trbk , Nv , ...)", alors θxi est la substitution {comp(Traj,Trbk,Nv,...){θaj})/Trxi}.

(d) **cas-or**

Ce cas implique la création de deux clauses qui seront construites selon les règles ci-dessous:

d1- la séquence [Trx1 , ... , Trxn] vient directement de la séquence des termes composés associée au noeud or lors de la transformation FOPL.

d2- chaque terme de composition Ctrxi est déduit de l'équation de composition de son terme de sélection Trxi correspondant: ainsi si l'équation de composition est "Trxi = comp (Traj)", alors le terme de composition correspondant Ctrxi sera "comp (Traj){Ctraj/Traj}", où "comp" est soit "inj", soit "=".

d3- les équations atomiques "d-eq-a" et "d-eq-b" restent deux équations atomiques "d-eq-a" et "d-eq-b" dans la nouvelle clause.

d4- chaque substitution θxi du sous-arbre de gauche (respectivement de droite) du noeud or produit une nouvelle substitution sur le noeud or, déduite de l'équation de composition du terme Trxi et de la valeur initiale de cette substitution. Ainsi, si l'équation de composition est "Trxi = comp (Traj)", alors θxi est la substitution {comp(Traj){θaj})/Trxi}.

(e) **cas-forall**

La mise sous forme récursive du forall implique la création de deux clauses qui seront construites selon les règles ci-dessous.

Pour la première clause:

e11- la séquence [Trx1 , ... , Trxn] vient directement de la séquence des termes composés associée au noeud forall lors de la transformation FOPL.

e12- pour chaque terme de sélection Trxi, nous trouvons d'abord son équation de composition déduite de l'équation de sélection:

*) si l'équation de sélection est "Traj = ith (index , Trxi)", alors l'équation de composition est "Trxi = empty".

*) si l'équation de sélection est "Traj = Trxi", alors l'équation de composition est "Trxi = Traj".

Dans les deux cas, cette équation de sélection peut être représentée de façon générale par "Trxi = comp (Traj)".

e13- chaque terme de composition Ctrxi est déduit de l'équation de composition de son terme de sélection Trxi correspondant: ainsi si l'équation de composition est "Trxi = comp (Traj)", alors le terme de composition correspondant Ctrxi sera "comp (Traj){Ctraj/Traj}".

Pour la deuxième clause:

e21- la séquence [Trx1 , ... , Trxn] vient directement de la séquence des termes composés associée au noeud forall lors de la transformation FOPL.

e22- pour chaque terme de sélection Trxi, nous trouvons d'abord son équation de composition déduite de l'équation de sélection:

*) si l'équation de sélection est "Traj = ith (index , Trxi)", alors l'équation de composition est "Trxi = cons (Traj , Nvj)".

*) si l'équation de sélection est "Traj = Trxi", alors l'équation de composition est "Trxi = Traj".

Dans les deux cas, cette équation de sélection peut être représentée de façon générale par "Trxi = comp (Traj)".

e23- chaque terme de composition Ctrxi est déduit de l'équation de composition de son terme de sélection Trxi correspondant: ainsi si l'équation de composition est "Trxi = comp (Traj)", alors le terme de composition correspondant Ctrxi sera "comp (Traj) {Ctraj/Traj}".

e24- l'équation atomique "d-eq-a" devient l'équation atomique "d-eq-a" dans la nouvelle clause.

e25- chaque substitution θxi produit une nouvelle substitution sur le noeud forall, déduite de l'équation de composition du terme Trxi et de la valeur initiale de cette substitution. Ainsi, si l'équation de composition est "Trxi = comp (Traj)", alors θxi est la substitution {comp(Traj){θaj})/Trxi}.

e26- une nouvelle séquence de substitution est ajoutée à la fin de la nouvelle clause: chaque nouvelle substitution θxi est déduite de

l'équation de composition (voir e22-) correspondant au terme Trxi de la façon suivante:

*) si l'équation de composition est "Trxi = cons (Traj , Nvi)", alors la substitution θxi est "Nvi/Trxi".

*) si l'équation de composition est "Trxi = Traj", alors la substitution θxi est "Trxi/Trxi".

(f) cas-if_then

f1- la séquence [Trx1 , ... , Trxn] vient directement de la séquence des termes composés associée au noeud if_then lors de la transformation FOPL.

f2- chaque terme de sélection Trxi apparaît directement car il n'y a pas d'équations de décomposition sur un noeud if_then. Le terme de composition Ctrxi sera une simple copie du terme de composition correspondant dans les sous-arbres associés à ce noeud.

f3- l'équation atomique "d-eq-a", tout comme l'équation "d-eq-b" correspondant à la précondition, devient une équation atomique dans la nouvelle clause générée. L'équation atomique "d-eq-b" se place devant l'équation atomique "d-eq-a" dans cette clause.

f4- chaque substitution θxi produit une nouvelle substitution sur le noeud if_then, déduite du terme Trxi et de la valeur initiale de cette substitution de la façon suivante:

*) si Trxi est un terme composé de "d-eq-b", alors θxi est la substitution identité {Trxi/Trxi}.

*) si Trxi est un terme composé de "d-eq-a", alors θxi est la substitution {Traj{θaj}/Trxi}.

(g) cas-if_then_otherwise

Ce cas implique la création de deux clauses qui seront construites selon les règles ci-dessous:

g1- la séquence [Trx1 , ... , Trxn] vient directement de la séquence des termes composés associée au noeud if_then_otherwise lors de la transformation FOPL.

g2- chaque terme de sélection Trxi apparaît directement car il n'y a pas d'équations de décomposition sur un noeud if then otherwise. Le terme de composition Ctrxi sera une simple copie du terme de composition correspondant dans les sous-arbres associés à ce noeud.

g3- l'équation atomique "d-eq-a", tout comme l'équation "d-eq-b" correspondant à la précondition, devient une équation atomique dans la nouvelle clause générée. L'équation atomique "d-eq-b" se place devant l'équation atomique "d-eq-a" dans cette clause.

g4- chaque substitution θxi produit une nouvelle substitution sur le noeud if then otherwise, déduite du terme Trxi et de la valeur initiale de cette substitution de la façon suivante:

*) si Trxi est un terme composé de "d-eq-b", alors θxi est la substitution identité {Trxi/Trxi}.

*) si Trxi est un terme composé de "d-eq-a", alors θxi est la substitution {Traj{θaj}/Trxi}.

Les règles de création de la seconde clause sont exactement les mêmes que celles utilisées pour la création de la première clause, en utilisant les sous-arbres correspondant à la partie otherwise du noeud if then otherwise.

(h) cas de l'arbre tout entier

A chaque clause finale obtenue par les transformations décrites ci-dessus, nous pouvons appliquer les substitutions d'égalité nécessaires contenues dans la séquence [θ1 , ... , θn], ce qui nous fournit une clause à adapter à la syntaxe concrète Prolog selon les conventions notationnelles décrites dans le cas des assertions simples au paragraphe 5.3.2.4.

Le résultat est une procédure Prolog "eval" composée d'un ensemble de sous-procédures, c'est-à-dire une séquence de clauses ayant toutes le même symbole d'opération "op" en tête, de la forme "eval (... , op (...)) :- ...".

Le fondement théorique étant établi, nous allons donner les bases de sa validation avant d'expliquer comment tout cela a été mis en oeuvre au moyen du CSG et des grammaires attribuées pour produire un prototype exécutable en Prolog.

5.4. Ebauche de validation de la théorie.

Nous savons maintenant comment sera construit théoriquement le prototype. Encore faut-il voir si celui-ci possède les propriétés requises, à savoir

- d'une part si le programme disons P généré est correct par rapport à la théorie produite par la première phase de la transformation. Cette théorie est un ensemble d'équivalences de la forme " Res = op (Arg1 , ... , Argn) <=> FF (d-eqs) ".

- d'autre part si les objets et les opérations de la dite théorie ont gardé leur traçabilité dans le programme P généré.

5.4.1. Correction.

La propriété formelle de correction est basée sur la définition de [HOGGER84] qui s'exprime en termes de correction partielle et de complétude.

Le but premier du prototype est d'appliquer une opération donnée à une liste donnée d'arguments. Ainsi, si nous supposons donnée une opération "op" et une liste d'arguments "Arg1 , Arg2 , ... ", notre but est de trouver la valeur de "op (Arg1 , Arg2 , ...)". Nous devons donc trouver la valeur d'une variable, disons "Res", telle que l'équation "Res = op (Arg1 , Arg2 , ...)" soit un théorème de la théorie. Du point de vue logique, on peut montrer que les sous-procédures Prolog P produites sont des conséquences logiques de la spécification S, c'est-à-dire S |=P [HABRA88]. Ceci est un critère suffisant pour que le programme soit partiellement correct par rapport à sa spécification [HOGGER84].

[HABRA89] donne la preuve de la correction de la transformation envisagée dans ce chapitre. Nous n'insisterons pas sur ce fait qui sort du cadre et des objectifs de ce mémoire.

5.4.2. Traçabilité.

Puisque le prototype produit sera utilisé pour valider les spécifications initiales et supporter leurs éventuelles modifications, il est très utile que ce prototype garde la trace des composants, opérations et objets, introduits dans les spécifications initiales.

Nous sommes parvenus à atteindre cet objectif en imposant au processus de transformation de conserver la structure initiale, de conserver les noms initiaux des objets et des opérations, et d'éviter autant que faire se peut d'introduire de nouveaux noms.

Au niveau de la structure de la spécification.

La structure de la spécification est conservée: le prototype Prolog produit une procédure unique "eval" structurée en une série de sous-procédures. Chacune de ces sous-procédures correspond à l'assertion d'explicitation d'une opération RSL. La seule perte de traçabilité au niveau de la structure de la spécification vient du fait que les noeuds RSL or, forall, if then otherwise ont produit en FOPL des noeuds or qui se sont scindés en deux clauses de Horn.

Au niveau des noms d'objets.

Le processus de transformation utilise exclusivement les noms des variables de l'assertion d'explicitation initiale. Les seuls nouveaux noms de variables introduits pendant cette étape de la transformation l'ont été:

*) soit dans le cas d'un produit cartésien n-aire: sur un noeud and, seulement deux des composants de ce produit cartésien sont disponibles pour construire le constuple. On met de nouvelles variables pour compléter ce constuple. Ces nouvelles variables disparaissent dans le programme Prolog car elles y sont remplacées par la variable anonyme "_".

*) soit dans le cas d'une séquence: sur un noeud forall, le constructeur de séquence, introduit par la notion de récursivité, décompose cette séquence en l'élément de tête (qui a un nom en RSL, le nom de variable utilisé dans l'explicitation) et en la queue qui n'a pas de nom en RSL. Seuls les noms de variables introduits pour désigner ces queues de liste resteront comme nouveaux noms de variables dans le code Prolog généré.

Au niveau des noms d'opérations.

Le processus de transformation n'introduit aucun nouveau nom d'opération.

Ce choix nous a obligé à laisser tomber l'efficacité du prototype généré. Mais la traçabilité est préférable à l'efficacité dans le cas de la génération d'un prototype. Selon ce choix, pour chaque connecteur forall d'un niveau

intermédiaire, chaque appel récursif portant sur une opération sans nom explicite a été transformé en un appel récursif à l'opération principale "op" qui est nommée explicitement dans le membre de gauche de l'assertion d'explicitation.

En conclusion des trois niveaux ci-dessus, on peut dire que la lisibilité du code Prolog finalement généré diminue proportionnellement à la lisibilité de la spécification RSL initiale. En outre, ce phénomène s'accroît dès qu'il y a plusieurs niveaux de forall imbriqués ou dès qu'un usage intempestif des équations implicites est adopté. En tout, il faut savoir rester raisonnable et abuser modérément des privilèges qui sont offerts.

5.4.3. Eval.

Nous avons déjà mentionné le fait que le "=" est appelé "eval" dans le programme prolog final. Nous en donnerons ici la justification .

En fait, l'assertion "res = OP (arg1, arg2, ...)" est vue comme une relation "=" entre deux termes. Cette relation lie deux termes qui sont équivalents du point de vue sémantique. Ceci est fondamentalement différent de l'égalité syntaxique des termes prédéfinie en Prolog. Ainsi, nous transformerons une explicitation d'opération "res = OP (arg1, arg2, ...)" en une procédure Prolog "eval (Res, op (Arg1, Arg2, ...))" qui peut instancier Res à partir d'une instanciation des Argi. Cette façon de procéder correspond bien au mode d'utilisation normale d'un prototype.

"Evaluator does evaluate": Un appel de procédure "eval (Res, expression)" se termine toujours et produit une instanciation de "Res" qui est correcte, c'est-à-dire qui est égale à "op (Arg1 , ... , Argn)" dans le modèle de la théorie. De plus nous pouvons prouver que cette instantiation de "Res" n'est pas n'importe quel terme équivalent à "op (Arg1, Arg2, ... , Argn)" mais bien le terme normal, c'est-à-dire l'évaluation au sens fonctionnel de l'expression "op (Arg1,Arg2,...) ". Cette instanciation constitue l'évaluation cherchée.

Après cette brève mise au point quant à la validation de la transformation, nous pouvons passer à la façon dont cette dernière a été réalisée au moyen de l'outil CSG et des grammaires attribuées.

5.5. Réalisation de la transformation à l'aide du CSG.

Cette section a pour but de décrire comment, en partant de la théorie décrite ci-dessus, nous avons réalisé l'implémentation de notre outil de transformation à l'aide d'une part des grammaires attribuées et d'autre part du CSG et de son langage associé SSL.

Après avoir rappelé les éléments qui sont à notre disposition à la sortie de la première étape de la transformation et dont nous aurons essentiellement besoin dans cette étape, nous décrirons les attributs nouveaux introduits afin de mener à bien la seconde phase de la transformation. Dans un troisième temps, nous verrons comment ces attributs sont mis à jour sur chaque noeud de l'arbre abstrait RSL en respectant les conventions Prolog décrites auparavant. L'étape suivante expliquera comment à la racine de l'arbre nous produirons le code Prolog exécutable à partir des attributs construits. La validation sur le jeu de test présenté à la fin du chapitre 4 et les résultats obtenus seront donnés en fin de ce chapitre.

5.5.1. Attributs fournis par la première phase de transformation.

La théorie propose, comme nous l'avons vu, de partir des équations de sélection générées lors de la première phase de transformation et de construire progressivement les équations de composition correspondantes, tout en conservant la séquence des substitutions attachée à ces équations de composition en vue de produire la clause Prolog équivalente à l'explicitation de l'opération sur la racine de l'arbre concerné.

Notre outil d'implémentation de la première phase nous fournit dans un attribut dont le nom est, rappelons-le, *eqimp* et dont la structure est une liste de couples de la forme (id , fterme), la liste complète des équations de sélection implicites liant une variable du membre de droite de l'explicitation de l'assertion à la variable correspondante dans le membre de gauche. Un couple de *eqimp* a la forme **(Vd , sel (sel (... , sel (Vg) ...)))** et représente l'équation **Vd = sel (sel (... , sel (Vg) ...))**.

Il faudra donc transformer toutes les équations de ce type en des équations de la forme **Vg = comp (comp (... , comp (Vd) ...))** et cela bien sûr en fonction du noeud courant et de l'opérateur de sélection éventuel qui lui est associé.

Rappelons que l'attribut *eqimp* est un attribut hérité qui est transmis de la racine de l'arbre RSL aux feuilles de cet arbre et qui est par conséquent disponible en tout noeud de cet arbre.

De plus, signalons que le type générique qui servira à construire l'opérateur de composition, essentiellement dans le cas de l'opérateur "constuple (... , TYPE : var , ...)" correspondant à un noeud and et dans le cas de l'opérateur "inj (TYPE, var)" correspondant à un noeud or, pourra être aisément trouvé dans l'attribut *vextyp* qui contient toutes les variables et leur explicitation de type. Cet atribut hérité est disponible à tout endroit de l'arbre RSL.

C'est là l'essentiel des attributs dont nous aurons besoin lors de cette phase de transformation de RSL à Prolog via FOPL.

Il est bien entendu que pour réaliser cette phase de transformation, de nouveaux attributs seront nécessaires et devront être introduits. C'est ce que présente la section suivante.

5.5.2. Attributs propres à la transformation en Prolog.

La méthodologie adoptée pour construire les différents attributs nécessaires à la réalisation de la seconde phase de la transformation est tout à fait semblable à celle adoptée lors de la première phase.

Sachant ce qu'il faut produire, de quelles structures de données aurons-nous besoin?

Ayant ces structures de données et sachant que nous sommes sur tel ou tel noeud de l'arbre abstrait RSL, comment devons-nous modifier les attributs hérités de notre noeud père et ceux synthétisés par nos noeuds fils en vue de conserver la cohérence et de fournir des informations correctes selon la théorie respectivement à notre noeud père et à nos noeuds fils. Ainsi, de fil en aiguille, arriverons nous à la racine du sous-arbre correspondant à l'explicitation (rappelons qu'il s'agit de

decformop dans notre syntaxe) où il suffira d'apporter la touche finale au transformateur.

Après avoir décrit les attributs nécessaires au transformateur, nous verrons comment les mettre à jour sur les différents noeuds de l'arbre abstrait RSL en respectant les règles théoriques de transformation, et comment produire le programme Prolog final sur la racine *decformop* .

Dans un premier temps, essayons de construire les structures d'attributs dont nous aurons besoin tout au long de cette étape de transformation.

Constatation 1.

Selon la théorie décrite au point 5.3.2.2., nous devons construire progressivement les opérateurs de composition à partir des opérateurs de sélection. Le principe de cette construction est simple:

- à partir d'une équation de sélection de la forme

  Obj = $as_{TYPEOBJ}$ (csel (Var))

  on va générer un opérateur de composition de la forme

  inj (TYPEOBJ,Obj)

  tel que

  inj (TYPEOBJ , Obj) = csel (Var),

- à partir d'une équation de sélection de la forme

  Obj = TYPEOBJ (: csel (Var))

  on va générer un opérateur de composition de la forme

  constuple (... , TYPEOBJ : Obj , ...)

  tel que

  constuple (... , TYPEOBJ : Obj , ...) = csel (Var).

Nous constatons donc qu'en réalité deux éléments vont être d'une importance capitale:

- l'opérateur de composition en cours de construction, par exemple

  constuple (... , TYPEOBJ : Obj , ...)

  inj (TYPEOBJ , Obj)

- l'opérateur de sélection en cours de dépliage: à chaque fois qu'un opérateur de composition est généré, l'opérateur de sélection lui correspondant a été amputé de ses deux éléments de tête. Par exemple,

l'opérateur de sélection as$_{TYPEOBJ}$ (csel (Var)) est devenu l'opérateur de sélection csel (Var) dès que l'opérateur de composition "inj" associé a été généré,

l'opérateur de sélection TYPEOBJ (: csel (Var)) est devenu l'opérateur de sélection csel (Var) dès que l'opérateur de composition "constuple" associé a été généré.

Dans le cas bien particulier des équations de sélection associées à un produit cartésien, il est clair que plusieurs équations de sélection conduiront au même opérateur de composition. Ce problème vient du fait qu'un produit cartésien n-aire est représenté par n - 1 connecteurs binaires and: par exemple, la structure CP [A , B , C] sera vue sous forme d'arbre par

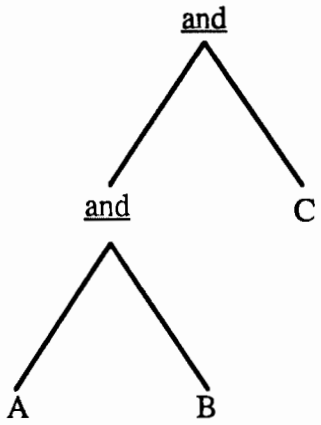


fig 5.1. Représentation d'un produit cartésien ternaire.

Lors de la construction automatisée de l'opérateur de composition associé à cet arbre, le risque de produire "constuple [A : a , B : b , C : Nv1]" suivi de la production de "constuple [A : Nv2 , B : Nv3 , C : c]" est grand. En réalité, ces deux constuples n'en forment qu'un seul, et il n'est pas nécessaire de recourir à l'introduction de nouvelles variables "Nvi" pour construire ce constuple.

Afin d'éviter les doublons que cela produirait parmi les opérateurs de composition construits et parmi les opérateurs de sélection dépliés, nous avons pensé envisager deux alternatives dans le cas particulier du noeud and: connaissant la structure d'un produit cartésien affecté par ce noeud and, peut-on recevoir d'un des fils (une des deux branches du noeud and) une construction de composition "constuple (...)" portant sur les types intervenant dans le dit produit cartésien? Si c'est le cas, il suffit de compléter ce constuple, sinon il faut le générer. Cette façon de procéder évite bien la construction inutile de constuples identiques et l'introduction non nécessaire de nouvelles variables.

Toutes ces constatations font apparaître clairement qu'il est nécessaire de disposer d'une structure de données dont les éléments représenteront respectivement le terme de composition en cours de construction et le terme de sélection associé en cours de dépliage, et cela pour une variable quelconque du membre de droite de l'explicitation de l'opération. Nous avons choisi comme structure la structure de couple d'éléments qui conserve effectivement le lien étroit existant entre le terme de composition et celui de sélection.

Il est bien entendu qu'à toutes les variables du membre de droite va correspondre un tel couple: nous aurons donc besoin en fait d'une liste de tels couples.

Nous noterons cette liste par

(ctr , tr)*

où

ctr est le terme de composition en cours de construction. Il a la forme ccomp (... Vd ...).
tr est le terme de sélection en cours de dépliage. Il a la forme csel (... Vg ...).
avec ccomp (... Vd ...) = csel (... Vg ...).

Cette liste deviendra à la racine la partie (Vg = comp (Vd))* des formes numérotées de (1) à (7) au point 5.3.2.2.

Constatation 2.

Toujours en analysant la théorie décrite au point 5.3.2.2., il est assez évident qu'à cette liste il va falloir associer les équations atomiques de la forme "d-eq-a", "d-eq-b", ...

Nous avons donc pensé augmenter la liste des couples ci-dessus de la façon suivante:

(ctr , tr)* , eqs

où eqs représente les équations atomiques correspondant à la liste des couples (ctr , tr)*.

Cet élément est en fait la partie (Vg = comp (Vd))* $\wedge$ d-eq-a des formes numérotées de (1) à (7) au point 5.3.2.2.

Nous verrons comment évolue cette composante "eqs" sur les divers noeuds de l'arbre RSL. Signalons dès à présent qu'en réalité "eqs" sera mis directement sous la forme Prolog "eval (... , op (...))" puisque "eqs" vient en réalité des équations atomiques obtenues sur les feuilles de l'arbre.

Toujours selon la théorie, certains noeuds vont produire plusieurs éléments de la forme "(ctr , tr)*, eqs", entre autres sur les noeuds or, forall et if then otherwise qui dédoublent les lignes de la forme rappelée ci-dessus, dans le sens où ces transformations produisent de telles lignes jointes par le connecteur logique "v".

Il est clair en conséquence qu'il est judicieux de considérer une liste d'éléments de cette forme; nous noterons cette liste par

((ctr , tr)* , eqs)*.

Constatation 3.

Comme nous l'avons exposé dans la théorie, le noeud forall est particulier dans le sens où il faut y générer une récursivité, en produisant entre autres une ligne de la forme "(Vg = empty[3])* ∧ (Seq = empty)", qui correspond dans notre représentation actuelle à l'élément "(empty , csel (Vg))*, _", où csel est une composition d'opérateurs de sélection ou l'égalité.

De plus, et toujours selon la théorie, dans le cas du forall, une formule récursive sera générée et ajoutée, au noeud racine, tout à la fin de toutes les clauses concernées, moyennant un jeu de substitutions adéquat. En fait, seule cette séquence de substitutions est à mémoriser. Nous compléterons en conséquence notre liste en y ajoutant une liste contenant cette séquence de substitutions, séquence qui sera elle-même une liste de substitutions d'égalité. Nous noterons cette séquence par subseq. "Physiquement", dans notre cas, cette séquence aura la forme ((ctr , tr)*)* pour garder trace du lien existant également entre ctr et tr, à savoir ctr = tr.

En résumé, la seule structure de données qui va nous permettre de réaliser la transformation de RSL et FOPL vers Prolog est celle donnée par

((ctr , tr)* , eqs , subseq)*

[3] "empty" désigne n'importe quelle liste vide.

où

ctr est un terme représentant l'opération de composition sur un noeud donné,

tr est un terme représentant l'opération de sélection sur un noeud donné.
On a ctr (...) = csel (...), propriété (seq.2) en 5.3.3.

eqs est une conjonction d'antécédents au sens Prolog, obtenue à partir des équations atomiques, propriété (seq.3) en 5.3.3.

subseq est une séquence éventuellement vide représentant une substitution multiple, propriété (seq.4) en 5.3.3.

Il est à remarquer qu'en réalité la liste (ctr , tr)* contient la séquence des égalités. Cette séquence sera nécessaire pour la transformation finale sur la racine de l'arbre RSL en vue de produire le code Prolog, propriété (seq.1) en 5.3.3..

Pour la petite histoire, signalons enfin que cette liste dans notre implémentation est la valeur d'un attribut appelé *compsel*.

Ayant la structure des données, il "suffit" de la faire évoluer le long de l'arbre RSL et conformément aux règles données par la théorie pour produire un code Prolog exécutable. C'est le but de la section suivante.

5.5.3. Construction du programme Prolog.

Dans toute la partie description de l'implémentation du processus de transformation de RSL combiné à FOPL vers Prolog exposée dans cette section, nous parlerons en termes de la liste introduite dans la section précédente, soit

$$\textit{compsel} = ((\text{ctr} , \text{tr})^* , \text{eqs} , \text{subseq})^*$$

Nous allons parcourir les noeuds de l'arbre abstrait RSL un à un et voir comment y évolue cette liste.

De plus, toutes les références à la théorie citées dans cette section se rapportent au point 5.3.3. du présent chapitre.

5.5.3.1. Au niveau des feuilles.

A cet endroit, il suffit d'initialiser la liste *compsel* de la façon suivante, sachant qu'une feuille a forcément la forme Resi = OPi (Argi1 , ... , Argin) :

(ctr , tr)* = (ctri , tri)*

où ctri prend pour valeur les différentes variables apparaissant dans l'équation atomique, soit Resi , Argi1 , ... , Argin.
tri est soit csel (...) s'il existe une équation implicite pour la variable ctri (ceci est donné par l'attribut *eqimp*),
soit ctri sinon.

Ceci découle des cas a1- et a2- de la théorie.

eqs = eval (Resi , opi (Argi1 , ... , Argin))

en se référant au cas a3- de la théorie.

subseq = empty

car aucune forme récursive n'est à envisager dans ce cas,
en se référant au cas a4- de la théorie.

5.5.3.2. Sur un noeud with.

Sur un tel noeud, nous obtenons en fait dans la partie explicitation du with une liste que nous noterons ((ctrw , trw)* , eqsw , subseqw)*. Aucune opération de décomposition n'étant à effectuer dans ce cas, seule la partie introduisant les objets intermédiaires est à ajouter devant l'élément "eqsw" de cette liste. Cette partie introduisant les objets intermédiaires nous est donnée directement par le noeud du sous-arbre correspondant à l'explicitation de ces objets.

Ainsi, nous mettons à jour la liste *compsel* de la façon suivante:

(ctr , tr)* = (ctrw , trw)*

Ceci découle des cas b1- et b2- de la théorie.

eqs = eqsint , eqsw[4]

où eqsint correspond aux objets intermédiaires introduits par le with.

Ceci découle du cas b3- de la théorie.

subseq = subseqw

car rien n'a changé au niveau des formes récursives dans ce cas.

Ceci découle du cas b4- de la théorie.

5.5.3.3. Sur un noeud and.

Sur un tel noeud, nous obtenons en fait une liste que nous noterons compselg = ((ctrg , trg)* , eqsg , subseqg)* venant du fils de gauche et une liste que nous noterons compseld = ((ctrd , trd)* , eqsd , subseqd)* venant du fils de droite.

Il s'agit dès lors de mettre à jour la liste *compsel* qu'il faudra passer au noeud père en fonction de ces deux listes et du fait que nous sommes sur un noeud and.

Pour ce faire, voici comment nous procéderons.

1. Nous allons d'abord fusionner les listes compselg et compseld en une seule liste compsel :

   Pour chaque élément de compselg, produire pour chaque élément de compseld l'élément dont les composantes sont créées comme suit

   (ctr , tr)* = (ctrg , trg)* @ (ctrd , trd)*

   Où @ est le symbole de concaténation de deux listes (symbole SSL).

   eqs = eqsg , eqsd

   subseq = subseqg @ subseqd

[4] La virgule utilisée ici dénote la conjonction de littéraux au sens Prolog: "eqs" a déjà la forme eval (...).

Ceci découle des cas c3- et c4- de la théorie.

2. Pour chaque élément de la liste compsel ainsi créée, nous éliminerons ensuite les doublons éventuels apparaissant dans les listes (ctr , tr)* et subseq et qui correspondent au même terme remontant de la partie gauche et de la partie droite du noeud and.

3. Pour chaque élément de la liste compsel ainsi créée, nous mettrons à jour chaque couple de la liste (ctr , tr)* en vertu des cas c2- et c1- de la théorie :

ou bien ce couple a la forme (ctra , sel (typea , termea)) et il y a nécessité d'une mise à jour d'un ctr. Selon la remarque faite sur l'arité du produit cartésien au point 5.4.2., nous devons envisager deux sous-cas:

ou bien dans la liste (ctr , tr)* il y a déjà un couple de la forme (constuple (... , typea : _ , ...) , tr) qu'il suffit de compléter en (constuple (... , typea : ctra , ...) , termea),

ou bien dans la liste (ctr , tr)* il n'y a aucun couple de la forme (constuple (... , typea : _ , ...) , tr). Il faut dans ce cas produire (constuple (... , typea : _ , ...) , tr) et l'ajouter à la liste (ctr , tr)*. Ce constuple peut facilement être construit à partir du produit cartésien obtenu dans la liste des types explicités contenus dans l'attribut vextyp. Ce constuple sera ensuite mis à jour comme dans le cas précédent.

Dans les deux sous-cas, le couple (ctra , sel (...)) est enlevé de la liste.

Toute l'information dont il était porteur se retrouve dans le constuple.

ou bien tr n'a pas la forme sel (type , terme) et la liste (ctr , tr)* reste inchangée.

4. Mise à jour des substitutions multiples éventuelles apparaissant dans la partie "subseq" de la liste et résultant de la création d'appels récursifs (cas c4- de la théorie) de la même façon qu'au point 3. ci-dessus.

5.5.3.4. Sur un noeud or.

Sur un tel noeud, nous obtenons en fait une liste que nous noterons compselg = ((ctrg , trg)* , eqsg , subseqg)* venant du fils de gauche et une liste que nous noterons compseld = ((ctrd , trd)* , eqsd , subseqd)* venant du fils de droite.

Il s'agit dès lors de mettre à jour la liste *compsel* qu'il faudra passer au noeud père en fonction de ces deux listes et du fait que nous sommes sur un noeud or.

Pour ce faire, voici comment nous procéderons.

1. Nous allons d'abord concaténer les listes compselg et compseld en une seule liste compsel (cas d3- de la théorie): le noeud or indiquant une disjonction Prolog qui se représente par deux clauses.

((ctrg, trg)*, eqsg,subseqg)* @ ((ctrd, trd)*, eqsd,subseqd)*

Où @ est le symbole de concaténation de deux listes (symbole SSL).

2. Pour chaque élément de la liste compsel ainsi créée, nous mettrons à jour chaque couple de la liste (ctr , tr)* en vertu des cas c2- et c1- de la théorie :

ou bien ce couple a la forme (ctra , as (typea , termea)) et il y a nécessité d'une mise à jour d'un ctr. Dans ce cas il suffit de remplacer cet élément (ctra , as (typea , termea)) par l'élément (inj (UNION (... , typea , ...) , ctra) , termea) .

ou bien tr n'a pas la forme as (type , terme) et la liste (ctr , tr)* reste inchangée.

3. Mise à jour des substitutions multiples éventuelles apparaissant dans la partie "subseq" de la liste et résultant de la création d'appels récursifs (cas d4- de la théorie) et de la même façon qu'en **2.** ci-dessus.

5.5.3.5. Sur un noeud forall.

Sur un tel noeud, nous obtenons en fait une liste que nous noterons compself = ((ctrf , trf)* , eqsf , subseqf)* .

Il s'agit dès lors de mettre à jour cette liste *compsel* qu'il faudra passer au noeud père en fonction du fait que nous sommes sur un noeud forall.

Pour ce faire, voici comment nous procéderons.

1. Pour chaque élément de la liste compsel, nous ajouterons à la liste (ctr , tr)* le couple (elem , ith (indice , seqguide)) qui ne s'y trouve pas encore car ces composantes sont données explicitement dans l'écriture même du forall : (forall elem : IN (elem , seqguide) : ...).

2. Pour chaque élément de la liste compsel, il faut produire, en vertu des cas e11- et e12- de la théorie :

(ctr , tr)* = (ctri , tri)*

où ctri = [] et tri = seqf si trf = ith (indice , seqf).
ctri = ctrf et tri = trf sinon.

eqs = /

aucune équation atomique n'est associée à ce cas.

subseq = empty

car ce n'est pas la partie récursive qui est à envisagée dans ce cas.

3. Pour chaque élément de la liste compsel, il faut produire

(ctr , tr)* = (ctri , tri)*

où ctri = cons (ctrf , Nvf) et tri = seqf si trf = ith (indice , seqf).
ctri = ctrf et tri = trf sinon.

où Nvf est une nouvelle variable introduite pour indiquer la queue de la séquence. La génération du nom de cette variable se passe de la façon suivante: nous concaténons le string "_queue" au string représentant la variable de gauche concernée dans l'équation de sélection (nous gardons ainsi une relative traçabilité) et au string représentant le nombre i,diquant la profondeur de cette variable de gauche dans le terme de sélection qui lui est associé.

Ce sont les cas e21- à e23 de la théorie.

eqs = eqsf

C'est le cas e24- de la théorie.

subseq = subseqf ::[5] (ctri , tri)*

où ctri = Nvf et tri = seqf si trf = ith (indice , seqf).
ctri = ctrf et tri = trf sinon.

car c'est la partie récursive qui est à envisagée dans ce cas.

Ce sont les cas e25- et e26- de la théorie.

4. Il suffit d'enlever de la liste compsel les éléments qui seraient des doublons éventuels.

5.5.3.6. Sur un noeud if_then.

Sur un tel noeud, nous obtenons en fait une liste que nous noterons compselt = ((ctrt , trt)* , eqst , subseqt)* . Aucune opération de décomposition n'étant à effectuer dans ce cas, seule la partie précisant la précondition est à ajouter devant l'élément "eqst" de cette liste. Cette partie précisant la précondition nous est donnée

[5] :: indique que l'on ajoute un élément d'une liste.

directement par le noeud du sous-arbre correspondant à l'explicitation de cette précondition.

Ainsi, nous mettons à jour la liste *compsel* de la façon suivante:

(ctr , tr)* = (ctrt , trt)*

Ceci découle des cas f1- et f2- de la théorie.

eqs = eqpré , eqst

où eqpré correspond à la formulation de la précondtion introduite par le if_then.

Ceci découle du cas f3- de la théorie.

subseq = subseqt

car rien n'a changé au niveau des formes récursives dans ce cas.

C'est le cas f4- de la théorie.

5.5.3.7. Sur un noeud if_then_otherwise.

Sur un tel noeud, nous obtenons en fait deux listes que nous noterons compselov = ((ctrov, trov)* , eqsov , subseqov)* pour celle correspondant au cas "vrai" du noeud if_then_otherwise, et compselof = ((ctrof, trof)* , eqsof , subseqof)* pour celle correspondant au cas "faux" du noeud if_then_otherwise. Aucune opération de décomposition n'étant à effectuer dans ce cas, seule la partie précisant la précondition est à ajouter devant l'élément "eqsov" de la liste compselov et la négation de la partie précisant la précondition est à ajouter devant l'élément "eqsof" de la liste compselof. Ces deux listes devront ensuite être concaténées. La partie précisant la précondition nous est donnée directement par le noeud du sous-arbre correspondant à l'explicitation de cette précondition.

1. Nous mettons à jour la liste *compselov* de la façon suivante:

   (ctr , tr)* = (ctrov , trov)*

Ceci découle des cas g1- et g2- de la théorie.

eqs = eqpré , eqsov

où eqpré correspond à la formulation de la précondtion introduite par le if then otherwise.

Ceci découle du cas g3- de la théorie.

subseq = subseqov

car rien n'a changé au niveau des formes récursives dans ce cas.

Ceci découle du cas g4- de la théorie.

2. Nous mettons à jour la liste *compselof* de la façon suivante:

(ctr , tr)* = (ctrof , trof)*

Ceci découle des cas g1- et g2- de la théorie.

eqs = not (eqpré) , eqsof

où not (eqpré) correspond à la formulation de la négation de la précondtion introduite par le if then otherwise.

Ceci découle du cas g3- de la théorie.

subseq = subseqof

car rien n'a changé au niveau des formes récursives dans ce cas.

Ceci découle du cas g4- de la théorie.

3. Nous concaténons ces deux listes mises à jour pour produire la liste compsel.

5.5.3.8. Sur la racine de l'arbre.

Sur la racine, nous disposons

d'une part du littéral exprimant l'opération décrite dans le membre de gauche de l'explicitation RSL, soit eval (Res , op (Arg1 , ... , Argn)),

d'autre part de la liste compsel = ((ctr , tr)* , eqs , subseq) dans laquelle tous les "tr" ont été complétement dépliés et représentent tous une variable de gauche, aussi bien dans (ctr , tr)* que dans subseq.

Il s'agit donc simplement

1. de substituer dans le littéral toutes les variables de gauche représentées par un "tr" par le terme lui correspondant dans le couple (ctr , tr),

2. d'appliquer au littéral les substitutions d'égalité contenues dans la séquence "subseq" puis d'ajouter ce littéral en conjonction à la fin de la partie "eqs",

3. de produire la clause Prolog "conseq :- eqs",

4. d'ordonner ces clauses en plaçant d'abord celles pour lesquelles "subseq = empty" (elles ne contiennent aucune forme de récursivité) et ensuite toutes les autres clauses.

Nous venons donc de décrire comment est construit le programme Prolog à partir des spécifications RSL et des résultats obtenus lors de la transformation en FOPL. La validation de cette étape de transformation est due au fait que toutes les règles de transformation découlent de la théorie qui elle est démontrée être correcte. Pour être complet, nous allons donner quelques exemples de réalisation basés sur le jeu de test présenté en fin du chapitre 4. C'est ce que fait la dernière section.

5.6. Exemples.

Nous donnons ci-après les codes Prolog générés par notre transformateur et correspondant aux quatre exemples illustrant le chapitre 4.

5.6.1. Un seul and.

```
eval ( constuple ( [ charst : Res1 , bool : Res2 ] ) , op ( constuple ( [ char : Arga1 ,
        int : Arga2 ] ) , Argb , constuple ( [ nat : Argc1, date : Argc2 ] ) ) )

:- eval ( Res2 , op2 ( Arga2 , Argc2 ) ) ,
   eval ( Inter , op3 ( Argb ) ) ,
   eval ( Res1 , op1 ( Arga1 , Argb , Argc1 , Inter ) ) .
```

5.6.2. Un seul or.

```
eval ( inj ( union ( charst , bool ) , Res2 ) , op ( inj ( union ( char , int ) , Arga2 ) ,
        Argb , inj ( union ( nat , date ) , Argc2 ) ) )

:- eval ( Res2 , op2 ( Arga2 , Argc2 ) ) .


eval ( inj ( union ( charst , bool ) , Res1 ) , op ( inj ( union ( char , int ) , Arga1 ) ,
        Argb , inj ( union ( nat , date ) , Argc1 ) ) )

:- eval ( Inter , op3 ( Argb ) ) ,
   eval ( Res1 , op1 ( Arga1 , Argb , Argc1 , Inter ) ) .
```

5.6.3. Un seul forall.

```
eval ( [ ] , op ( [ ] , Argb , Argc ) ) .


eval ( [ Res1 | _queueRes1 ] , op ( [ Arge | _queueArga1 ] , Argb , Argc ) )

:- eval ( Res1 , op1 ( Arge , Argb , Argc ) ) ,
   eval ( _queueRes1 , op ( _queueArga1 , Argb , Argc ) ) .
```

5.6.4. Un exemple plus complexe.

```
eval(constuple([union(cp(charst,date),char):inj(union(cp(charst,date),
        char),constuple([charst:Res1,date:Res2])),seq(cp(seq(set(charst))
        ,nat)):[ ]]),op(constuple([set(set(char)):_,seq(int):Arga2]),
        inj(union(set(set(int)),seq(nat)),Argb1),constuple([cp(set(nat),
        set(set(date))):constuple([set(nat):Argc1,set(set(date)):Argc2]),
        seq(seq(charst)):_]),Argd,Arge,inj(union(cp(set(date),set(char)),
        set(int)),constuple([set(date):Argf1,set(char):Argf2])),[ ],[ ]))

 :-

eval(Inter,op3(Argb1)),eval(Res2,op2(Argb1,Argc2,Inter,Argf2)),
eval(Res1,op1(Arga1,Argb1,Argc1,Inter,Argf1)).


eval(constuple([union(cp(charst,date),char):inj(union(cp(charst,date),
        char),constuple([charst:Res1,date:Res2])),seq(cp(seq(set(charst))
        ,nat)):[ ]]),op(constuple([set(set(char)):_,seq(int):Arga2]),
        inj(union(set(set(int)),seq(nat)),Argb1),constuple([cp(set(nat),
        set(set(date))):constúple([set(nat):Argc1,set(set(date)):Argc2]),
        seq(seq(charst)):_]),Argd,Arge,inj(union(cp(set(date),set(char)),
        set(int)),constuple([set(date):Argf1,set(char):Argf2])),[ ],
        [Argd1|_queueSeq21]))

 :-

eval(Inter,op3(Argb1)),eval(Res2,op2(Argb1,Argc2,Inter,Argf2)),
eval(Res1,op1(Arga1,Argb1,Argc1,Inter,Argf1)).


eval(constuple([union(cp(charst,date),char):inj(union(cp(charst,date),
        char),Res4),seq(cp(seq(set(charst)),nat)):[ ]]),op(constuple([set
        (set(char)):Arga1,seq(int):Arga2]),inj(union(set(set(int)),seq(
        nat)),Argb2),Argc,Argd,Arge,inj(union(cp(set(date),set(char)),
        set(int)),Argf3),[ ],[ ]))

 :-

eval(Res4,op4(Arga1,Argb2,Arge,Argf3)).


eval(constuple([union(cp(charst,date),char):inj(union(cp(charst,date),
        char),Res4),seq(cp(seq(set(charst)),nat)):[ ]]),op(constuple(
        [set(set(char)):Arga1,seq(int):Arga2]),inj(union(set(set(int)),
        seq(nat)),Argb2),Argc,Argd,Arge,inj(union(cp(set(date),set(char))
        ,set(int)),Argf3),[ ],[Argd1|_queueSeq21]))

 :-

eval(Res4,op4(Arga1,Argb2,Arge,Argf3)).
```

```
eval(constuple([union(cp(charst,date),char):inj(union(cp(charst,date),
        char),Res4),seq(cp(seq(set(charst)),nat)):[constuple([seq(set(
        charst)):[Res5|_queueRes4],nat:Res6])|_queueRes2]]),op(constuple(
        [set(set(char)):Arga1,seq(int):Arga2]),inj(union(set(set(int)),
        seq(nat)),Argb2),Argc,Argd,Arge,inj(union(cp(set(date),set(char))
        ,set(int)),Argf3),[Argc3|_queueSeq11],[Argd1|_queueSeq21]))

 :-

eval(Res6,op6(Argc3,Arge)),
eval(Res5,op5(Arga2,Argc3)),
eval(Res4,op4(Arga1,Argb2,Arge,Argf3)),
eval(Res,op(Arga,Argb,Argc,Argd,Arge,Argf,Seq1,_queueSeq21)),
eval(constuple([union(cp(charst,date),char):_,seq(cp(seq(set(charst)),
        nat)):_queueRes2]),op(Arga,Argb,Argc,Argd,Arge,Argf,_queueSeq11,
        Seq2)).


eval(constuple([union(cp(charst,date),char):inj(union(cp(charst,date),
        char),Res4),seq(cp(seq(set(charst)),nat)):[constuple([seq(set(
        charst)):[ ],nat:Res6])|_queueRes2]]),op(constuple([set(set(
        char)):Arga1,seq(int):Arga2]),inj(union(set(set(int)),seq(nat)),
        Argb2),Argc,Argd,Arge,inj(union(cp(set(date),set(char)),set(int))
        ,Argf3),[Argc3|_queueSeq11],[ ]))

 :-

eval(Res6,op6(Argc3,Arge)),
eval(Res4,op4(Arga1,Argb2,Arge,Argf3)),
eval(constuple([union(cp(charst,date),char):_,seq(cp(seq(set(charst)),
        nat)):_queueRes2]),op(Arga,Argb,Argc,Argd,Arge,Argf,_queueSeq11,
        Seq2)).


eval(constuple([union(cp(charst,date),char):inj(union(cp(charst,date),
        char),constuple([charst:Res1,date:Res2])),seq(cp(seq(set(charst))
        ,nat)):[constuple([seq(set(charst)):[Res5|_queueRes4],nat:Res6])
        |_queueRes2]]),op(constuple([set(set(char)):_,seq(int):Arga2]),
        inj(union(set(set(int)),seq(nat)),Argb1),constuple([cp(set(nat),
        set(set(date))):constuple([set(nat):Argc1,set(set(date)):Argc2]),
        seq(seq(charst)):set(int)),constuple([set(date):Argf1,set(char):
        Argf2])),[Argc3|_queueSeq11],[Argd1|_queueSeq21]))

 :-

eval(Res6,op6(Argc3,Arge)),
eval(Res5,op5(Arga2,Argc3)),
eval(Inter,op3(Argb1)),
eval(Res2,op2(Argb1,Argc2,Inter,Argf2)),
eval(Res1,op1(Arga1,Argb1,Argc1,Inter,Argf1)),
eval(Res,op(Arga,Argb,Argc,Argd,Arge,Argf,Seq1,_queueSeq21)),
eval(constuple([union(cp(charst,date),char):_,seq(cp(seq(set(charst)),
        nat)):_queueRes2]),op(Arga,Argb,Argc,Argd,Arge,Argf,_queueSeq11,
        Seq2)).
```

```
eval(constuple([union(cp(charst,date),char):inj(union(cp(charst,date),
        char),constuple([charst:Res1,date:Res2])),seq(cp(seq(set(charst))
        ,nat)):[constuple([seq(set(charst)):[ ],nat:Res6])|_queueRes2]]),
        op(constuple(set(set(char)):_,seq(int):Arga2,:_),inj(union(set(
        set(int)),seq(nat)),Argb1),constuple([cp(set(nat),set(set(date)))
        :constuple([set(nat):Argc1,set(set(date)):Argc2]),seq(seq(charst)
        ):_]),Argd,Arge,inj(union(cp(set(date),set(char)),set(int)),
        constuple([set(date):Argf1,set(char):Argf2])),[Argc3|_queueSeq11]
        ,[ ]))

 :-

eval(Res6,op6(Argc3,Arge)),
eval(Inter,op3(Argb1)),
eval(Res2,op2(Argb1,Argc2,Inter,Argf2)),
eval(Res1,op1(Arga1,Argb1,Argc1,Inter,Argf1)),
eval(constuple([union(cp(charst,date),char):_,seq(cp(seq(set(charst)),
        nat)):_queueRes2]),op(Arga,Argb,Argc,Argd,Arge,Argf,_queueSeq11,
        Seq2)).
```

Chapitre 6

Extensions possibles

6.1. Limites actuelles.

Le fonctionnement actuel de l'outil demande la création, par un éditeur de texte quelconque, d'un fichier contenant la spécification RSL d'une ou plusieurs opérations et de tous les types d'objets qui y correspondent. L'exécution du programme de prototypage avec ce fichier comme donnée produit une visualisation de l'arbre abstrait correspondant à la spécification introduite et le calcul des attributs. En se plaçant à la racine de l'arbre et en demandant au CSG d'écrire les valeurs des attributs ad hoc, il est possible d'obtenir deux fichiers contenant l'un, la formule FOPL et l'autre le programme PROLOG correspondant aux opérations de la spécification donnée.

Il est possible de traiter plusieurs opérations simultanément. Ce traitement collectif à l'avantage d'éviter l'introduction répétée à chaque opération de certains blocs de description de types d'objets utilisés dans chacune des opérations.

La spécification initiale est supposée correcte par rapport à la syntaxe concrète présentée au chapitre 2 (section 2.6.). Remarquons que la syntaxe actuelle exclut certains processus d'élaboration de spécification. Par exemple, il n'est pas possible d'instancier une opération à partir d'une opération générique car la construction "where" ne fait pas partie de la syntaxe actuelle. Si les types paramétrés "SET" et "TABLE" sont syntaxiquement corrects, les opérations de base leur correspondant n'apparaissent pas encore dans la syntaxe ce qui peut limiter l'emploi de ces types.

L'implémentation actuelle de l'outil de prototypage impose des restrictions qui n'apparaissent pas dans la syntaxe. Bien que syntaxiquement correctes, les constructions "IS-A" et "IS-A ...without" ne sont pas traitées dans la production du prototype. Il vaut donc mieux éviter de recourir au processus de spécialisation.

Rappelons que deux syntaxes non équivalentes avaient été définies pour l'opération de construction d'une occurrence du type "produit cartésien" (voir chapitre 2 section 2.6.). La déduction des équations implicites de décomposition de n-uplets est basée sur l'hypothèse que les composants sont de types différents. Le mécanisme levant l'ambiguité lorsqu'une occurrence de produit cartésien contient au moins deux composants de même type n'ayant pas encore été implémenté, les deux syntaxes du constructeur d'occurrences de produit cartésien "CONSTUPLE" ont la même puissance d'expression du point de vue de l'utilisation des équations implicites.

La spécification initiale est supposée avoir certaines propriétés de complétude et de consistance : on suppose par exemple que tous les objets utilisés sont déclarés avec leur type, qu'aucun objet n'est déclaré deux fois et de types différents dans le même bloc de spécification d'opération, etc.

La construction du prototype suppose la connaissance de la spécification complète des types; tout type doit être complètement défini par des types de base appartenant à la bibliothèque. Pour obtenir un prototype à partir d'un bloc de spécification d'une opération, il est donc nécessaire de connaître tous les blocs de spécification des types la concernant directement et indirectement (voir chapitre 4 pour les détails).

Des équations ne peuvent être laissées implicites que si elles portent sur les types "CP", "UNION" et "SEQ". Dans ce cas, la spécification doit être non ambigüe.

En fait, l'outil actuel ne peut traiter que les types définis par l'utilisateur, les types de base et les types paramétrés "CP", "UNION" et "SEQ"; parmi les opérations associées, il ne traite que les opérations définies par l'utilisateur, les opérations de construction "CONSTUPLE", "INJ", "SEMPTY" et "CONS", les opérations de sélection "SELECT", "AS", "ITH" et enfin, portant sur le type "SEQ", les opérations "LENGTH", "IN" et introduite par le connecteur "forall", l'application d'une même opération à tous les éléments d'une suite. Le mécanisme d'assertions gardées sous-jacent aux constructions représentées par <pré-et-explic> (voir chapitre 2 section 2.6.) est également traité. La traduction des autres opérations de base -losqu'elles seront disponibles- sera immédiate.

Remarquons qu'il est possible de remédier, moyennant un effort raisonnable à beaucoup des faiblesses de cette première version du système.

6.2. Recul des limites. Extensions possibles.

6.2.1. Création d'un éditeur syntaxique.

Il serait intéressant de disposer d'un éditeur syntaxique. L'introduction de la spécification RSL serait plus rapide si seuls des gabarits devaient être remplis des informations utiles. De plus, l'éditeur syntaxique effectuant les vérifications syntaxiques de manière interactive, les erreurs de syntaxe pourraient être directement corrigées.

Le CSG est un outil de génération d'éditeurs syntaxiques. Il offre donc toutes les facilités pour cette création. Le programme actuel contient déjà des définitions de "completing terms" pour tous les phyla. Restent à compléter les schémas de décompilation et à définir les transformations nécessaires.

6.2.2. Ecriture systématique des résultats.

Le CSG n'affiche à l'écran ou n'écrit dans un fichier les valeurs d'attributs qu'en réponse à une commande explicite. Dans un souci d'automatisation totale, on peut désirer se passer de ces commandes. Il est possible d'interfacer un éditeur généré avec du code C. on peut imaginer tirer parti de cette facilité pour produire directement les fichiers contenant les valeurs d'attributs désirées. La question reste ouverte quant à savoir si une telle réalisation est facile ou non.

6.2.3. Extension incrémentale de la syntaxe concrète.

Toutes les opérations disponibles dans la bibliothèque de spécification doivent apparaître dans la syntaxe abstraite et être traitées par l'outil. Le formalisme SSL étant déclaratif, l'ajout de nouvelles règles pour introduire ces opérations ne pose pas de problème de fond.

Dans la syntaxe concrète d'entrée, des problèmes d'ambiguïtés pourraient éventuellement se produire, que YACC signalerait par des conflits "reduce / reduce". Si l'éditeur syntaxique structurel est utilisé, ces problèmes potentiels disparaissent.

6.2.4. Traitement des structures syntaxiquement permises.

Le traitement des équations implicites dans le cas d'un produit cartésien composé de plusieurs occurrences d'un même type devrait pouvoir se faire sans trop de problèmes, l'idée étant d'appliquer aux noms de champs le traitement actuellement appliqué aux types, moyennant quelques adaptations d'écriture. L'information supplémentaire contenue dans les noms de champs est déjà enregistrée dans la version actuelle du programme, bien qu'elle ne soit pas exploitée. Le traitement des équations implicites de composition prévoit déjà la possibilité de générer des "CONSTUPLES" dont chaque composant est un couple (< nom-champ > , < nom-obj >).

6.2.5. Nouvelles relations implicites.

Le traitement d'équations implicites dans le cas de structures autres que celles déjà traitées est un problème beaucoup plus compliqué.

Citons quelques exemples de structures pour lesquelles un tel traitement pourrait être introduit.

Il arrive parfois qu'un objet arg_i dont le type est le i-ème type composant du type d'un argument se retrouve tel quel comme objet res_i composant du résultat. Cela signifie qu'à cette étape de décomposition, aucune opération ne porte sur cette partie de l'argument. Dans ce cas, il serait intéressant de ne pas avoir à écrire explicitement $res_i = arg_i$.

L'utilisation des types "SET" et "TABLE" doit révéler aussi des constructions qu'il serait souhaitable de laisser implicites.

Evidemment, résoudre de tels problèmes d'équations implicites nécessite une approche théorique (étude des hypothèses, vérification, ...), la découverte de nouvelles astuces d'implémentation, bref beaucoup de cogitations en perspective.

6.2.6. Vérification de la sémantique statique.

Certaines erreurs sont détectables à partir de la structure de la spécification. Par vérification de la sémantique statique, nous entendons la détection des erreurs qu'il est possible de déceler sans avoir à exécuter le prototype construit. On peut imaginer vérifier que tous les objets utilisés sont déclarés et typés sans contradiction, que les opérations explicitant une précondition ont un résultat booléen, que l'utilisation des opérations implicites ne laisse aucune ambiguïté, etc.

Parmi les attributs déjà introduits, certains peuvent être réutilisés à des fins de vérification : par exemple, l'attribut "vtyp" contenant l'information relative aux noms d'objets et à leur type associé peut servir à vérifier que tout objet utilisé est déclaré et typé. Lors du traitement des déclarations, le même attribut peut être testé pour vérifier qu'une nouvelle déclaration n'est pas redondante, voire contradictoire par rapport aux déclarations précédentes.

Notons qu'une grande partie de la spécification RSL n'intervient pas dans notre outil de prototypage. La description informelle ("LEXICON") est purement et simplement ignorée. Dans l'état actuel d'avancement de notre travail, les profils des opérations ne sont pas utilisés. On peut imaginer employer ces derniers pour tester que résultat et arguments d'une opération sont bien du type nécessaire à cette opération.

Ce genre de vérification peut s'opérer par la définition de nouveaux attributs. Leur ajout ne modifie en rien ce qui a été fait. La seule question restant posée est de savoir si l'accroissement de confiance acquis par le test vaut le temps de sa mise au point et de son exécution.

6.2.7. Prototypage par niveau d'abstraction.

Une amélioration possible serait de relâcher l'hypothèse concernant la disponibilité des explicitations de types jusqu'aux types de base et d'admettre des types non encore définis comme terminaux. Relaxer cette hypothèse demande pratiquement de changer les conditions de terminaison d'une fonction récursive. Le problème réside vraisemblablement moins dans ce changement que dans l'analyse des conséquences de cette modification sur l'implémentation du traitement des équations implicites.

6.2.8. Intégration de l'outil.

On peut enfin imaginer le développement d'un outil intégré permettant d'abord l'édition des spécifications, puis leur transformation en prototype exécutable, permettant ensuite l'exécution du prototype pour valider les spécifications, permettant enfin la modification éventuelle de celles-ci, le cycle "édition - prototypage - modification" pouvant être parcouru autant de fois qu'on le juge nécessaire.

Conclusion

Le développement de prototypes est une technique courante en ingénierie. En génie logiciel, on produit des prototypes pour affiner et valider des spécifications, pour tester plus tôt certaines caractéristiques d'un système à produire.

La production d'un prototype doit être rapide et relativement bon marché. Il est donc particulièrement intéressant de disposer d'aides logicielles pour supporter cette phase.

L'outil de prototypage que nous avons implémenté vise à produire automatiquement un prototype PROLOG à partir d'une spécification écrite en RSL. Le principe d'obtention du prototype consiste en l'application à la spécification initiale de transformations qui, tout en préservant la sémantique, conduisent finalement à un programme exécutable.

La production du programme exécutable s'opère en deux temps.

1) Une première transformation produit, à partir de l'expression en RSL de la relation décrivant une opération et de son explicitation en termes d'autres relations, une formule équivalente exprimée dans le langage des prédicats du premier ordre (FOPL).

L'assertion RSL "expression $\rightarrow$ explicitation" est transformée en formule "F(expression) $\Leftrightarrow$ F(explicitation)".

Cette première phase fait apparaître explicitement certaines équations qui avaient éventuellement été laissées implicites dans la spécification RSL.

Elle est indépendante du langage de programmation dans lequel sera finalement exprimé le prototype et est par conséquent réutilisable.

2) Une seconde transformation assure enfin le passage en PROLOG. Les équivalences issues de la première transformation sont transformées en clauses de

Horn. Tenant compte de la stratégie de résolution du moteur d'inférence de l'interprêteur PROLOG d'une part, et de l'utilisation souhaitée du programme (la directionnalité) d'autre part, les clauses et les littéraux dans les clauses sont ordonnés pour assurer la terminaison du programme.

Le prototype produit préserve la traçabilité par rapport aux spécifications dont il est issu. Dans chacune des transformations, les noms d'objets et d'opérations présents dans la spécification initiale sont conservés. Autant que possible, l'introduction de nouvelles opérations et de nouveaux objets est évitée. La traçabilité est une qualité importante si on veut utiliser le prototype à des fins de validation de la spécification.

Les relations qui, sous l'hypothèse qu'on peut les déduire sans ambiguïté du contexte, ont été laissées implicites dans la spécification initiale sont traitées au cours des deux transformations de sorte que le prototype en tient compte. Le traitement de ces équations implicites est d'ailleurs ce qui a posé le plus de problèmes lors de la réalisation de l'outil. En effet, il convenait, pour bénéficier de la justification théorique de calquer au maximum l'implémentation sur la description théorique. Cela a demandé une compréhension de la théorie et une certaine maîtrise de l'outil d'implémentation qui ne s'acquièrent que progressivement, nous incitant à distinguer de plus en plus prototypage rapide de production rapide d'outils de prototypage.

Nous disposions de la description théorique des transformations [HABRA 89]. La partie originale de notre travail est l'implémentation d'un outil basé sur ces transformations. Le choix de l'outil utilisé pour l'implémentation nous est également imputable.

Pour l'implémentation, nous avons pu disposer d'un outil puissant : le Cornell Synthesizer Generator, un générateur d'éditeurs syntaxiques, manipulant des arbres abstraits attribués.

Le fonctionnement de l'outil de prototypage actuellement réalisé peut se résumer comme suit :

- La spécification RSL d'une opération et des types d'objets y intervenant est traduite en arbre abstrait par l'analyseur syntaxique du CSG.

- Le CSG calcule les valeurs d'attributs dont l'arbre abstrait est décoré. Les attributs prennent leurs valeurs dans les domaines syntaxiques (phyla) déclarés. Ces valeurs d'attributs peuvent être représentées sous forme d'arbres de dérivation.

- Parmi les attributs que nous avons déclarés, deux sont remarquables. L'un a pour valeur la formule FOPL équivalant à la spécification d'opération donnée; l'autre a pour valeur une procédure PROLOG. L'un résulte de l'implémentation de la première transformation "RSL → FOPL", l'autre de la seconde "RSL + FOPL → PROLOG".

- Des schémas de décompilation spécifient la représentation visuelle des éléments des phyla. Le prototype PROLOG s'obtient par la décompilation de la valeur d'attribut idoine.

On remarquera la facilité qu'offre un tel système pour être étendu à la production de prototypes écrits en d'autres langages. Les attributs réalisant la première transformation peuvent être conservés tels quels. Il suffit de déclarer de nouveaux attributs implémentant la deuxième transformation pour un autre langage cible et de calculer leurs valeurs.

PROLOG est un langage qui, tout en étant concis, offre une grande puissance d'expression et une manipulation uniforme et aisée des structures de données. L'aspect relationnel de PROLOG permet de garder une bonne traçabilité par rapport aux spécifications RSL où les opérations sont représentées par des relations. Néanmoins, on peut trouver que l'utilisation du prototype, dans la mesure où elle impose une direction précise, n'exploite pas toutes les qualités du langage logique. Une exécution du prototype se réduit essentiellement à une évaluation de fonction, ce qui laisse à penser qu'un langage fonctionnel conviendrait également pour exprimer le prototype.

Diverses extensions de ce travail sont envisageables. La plus ambitieuse consiste peut-être à imaginer l'intégration de l'outil préalablement enrichi d'autres constructions syntaxiques et de divers mécanismes de validation sémantique, dans un environnement complet, offrant une aide à l'édition d'une spécification RSL correcte, et une présentation agréable du prototype, incitant à parcourir le cycle "édition-prototypage-modifications" autant de fois qu'il est nécessaire à l'obtention d'une spécification rencontrant les exigences du client.

Bibliographie

[BLUM 82] Blum B. and Houghton R.C.Jr., *Rapid Prototyping Of Information Management Systems,* ACM. SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, pp 35 - 38

[CHOPPY 87] Choppy C., *Formal Specifications, Prototyping And Integration Tests - Spécifications Formelles, Prototypage Et Tests D'Intégration,* ESEC'87, Première Conférence Européenne De Génie Logiciel, AFCET, Paris, 1987

[COHEN 82] Cohen D., Swartout W. and Balzer R., *Using Symbolic Execution To Characterize Behavior,* ACM.SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, pp 25 - 32

[CRISMER 88] Crismer P.-G. et de Wergifosse J., *Etude De La Conception Et De La Réalisation D'Un Environnement De Programmation Interactif COBOL A L'Aide Du Cornell Synthesizer Generator,* Mémoire présenté en vue de l'obtention du titre de Licencié et Maître en Informatique, FUNDP, Namur, Année Académique 1987-1988

[DAVIS 87] Davis N., *Problem Set,* in Proceedings of the 4th International Workshop on Software Specification and Design, IEEE, Monterey (California),1987

[DE MARCO 79] De Marco T., *Structured Analysis And System Specification,* A Yourdon Book, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1979

[DODD 82] Dodd W.P., Ramsay P., Axford T.H. and Parkyn D.G., *A Prototyping Language For Text Processing Applications,* ACM. SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, p 50

[DONZEAU - GOUGE 84] Donzeau-Gouge V., Huet G., Kahn G. and Lang B., *Programming Environments Based On Structured Editors : The MENTOR Experience,* In "Interactive Programming Environments", David R. Barstow, Howard E. Shrobe, Erik Sandewall (editors), Mc Graw - Hill Book Company, 1984

[DUBOIS 84] Dubois E., *Cadre Et Méthode De Spécification De Systèmes D'Information Fondés Sur Les Types De Données,* Thèse de Docteur-Ingénieur en Informatique, Institut National Polytechnique de Lorraine, Nancy, 1984

[DUBOIS 85] Dubois E., Finance J.-P., Van Lamsweerde A., *A Constructive Approach To Requirements Specification,* Report M101 Philips Research Laboratory, Brussels, 1985

[DUBOIS 87] Dubois E. and Van Lamsweerde A., *Making Specification Processes Explicit*, in Proceedings of the 4th International Workshop on Software Specification and Design, IEEE, Monterey (California), April 1987

[FEATHER 82] Feather M. S., *Mappings For Rapid Prototyping*, ACM. SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, pp 17 - 24

[FEATHER 87] Feather M. S., *Language Support For The Specification And Development Of Composite System*, ACM.Transactions On Programming Languages Ans Systems, Vol 9 N° 2, April 1987, pp 198 - 234

[GOGUEN 78] Goguen J.A., Thatcher J.W., Wagner E.G., *An Initial Algebra Approach To The Specification, Correctness And Implementation Of Abstract Data Type*, in Current Trends In Programming Methodology, Vol 4, R. Yeh (editor), Prentice Hall, 1978

[HABRA 88] Habra N. et Van Lamsweerde A., *Génération De Prototypes Prolog A Partir De Spécifications Formelles De Besoins*, to appear in CGL4 "4 ème colloque de Génie Logiciel", 19-21 octobre 1988 Paris-France.

[HABRA 89] Habra N., *Rapport RP 11/89*, Facultés Universitaires Notre-Dame de la Paix Namur, 1989

[HAINAUT 86] Hainaut J.-L., *Conception Assistée Des Applications Informatiques 2. Conception De La Base De Données*, Masson, Presses Universitaires de Namur, 1986

[HEITMEYER 82] Heitmeyer C., Landwehr C. and Cornwell M., *The Use Of Quick Prototypes In The Secure Military Message Systems Project*, ACM. SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, pp 85 - 87

[HOGGER 84] Hogger C.J., *Introduction To Logic Programming*, Academic Press, 1984

[JOHNSON 75] Johnson S.C., *Yacc - Yet Another Compiler-Compiler*, Comp. Sci. Tech. Rep. N° 32, Bell Laboratories, July 1975

[JOHNSON 78] Johnson S.C. and Lesk M.E., *UNIX Time-Sharing System : Language Development Tools*, The Bell System Technical Journal, Vol 57 N°6, July - August 1978, pp 2155 - 2175

[KLAUSNER 82] Klausner A. and Konchan T.E., *Rapid Prototyping And Requirements Specification Using PDS*, ACM. SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, pp 96 - 99

[KNUTH 68] Knuth D. E., *Semantics Of Context-Free Languages*, in Mathematical Systems Theory, Vol 2, pp 127 - 145, 1968 Correction in Vol 5 pp 95 - 96, 1971

[LANGELEZ 86] Langelez F. et Paris C., *Etude Et Evaluation D'Un Outil De Prototypage*, Mémoire présenté en vue de l'obtention du titre de Licencié et Maître en Informatique, FUNDP, Namur, Année Académique 1985-1986

[LARCH] *The Larch Project*, MIT Laboratory For Computer Science and DEC's systems Research Center

[LESK 75] Lesk M.E., *Lex- A Lexical Analyzer Generator*, Comp. Sci. Tech. Rep. N° 39, Bell Laboratories, Oct 1975

[LEVY 87] Levy N., Piganiol A. and Souquières J., *Specifying With SACSO*, in Proceedings of the 4th International Workshop on Software Specification and Design, IEEE, Monterey (California), April 1987

[LIVERCY 78] Livercy C., *Théorie Des Programmes*, Dunod, Paris, pp 285 - 293, 1978

[MAC EWEN 82] Mac Ewen G. H., *Specification Prototyping*, ACM. SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, pp 112 - 116

[MEYER 87] Meyer B., *Introduction To The Theory Of Programming Languages*, Prentice Hall International, 1987

[QUERE 88] Quéré A., *Présentation Du Langage ML*, Rapport CRIN 88-R-084, octobre 88

[REPS 87] Reps T. and Teitelbaum T., *The Synthesizer Generator Reference Manual*, Department of Computer Science Cornell University, 1987

[REPS 89] Reps T.W. and Teitelbaum T., *The Synthesizer Generator - A Systen For Constructing Language -Based Editors;* Springer-Verlag New York, 1989

[SCHOMAN 77] Schoman K. and Ross D.T., *Structured Analysis For Requirements Definition*, IEEE Trans. On Software Engineering, SE 3 (1), pp 41 - 48 , 1987

[SHNEIDERMAN 87] Shneiderman B., *Designing The User Interface, Strategies For Effective Human-Computer Interaction*, Addison-Wesley Publishing Company, Inc, 1987

[SOMMERVILLE 85] Sommerville I., *Software Engineering*, Second Edition, Addison - Wesley, 1982 - 1985

[SPIVEY 87] Spivey J.M., *An Introduction To Z And Formal Specification*, Oxford University Computing Laboratory, August 1987

[STAVELY 82] Stavely A.M., *Models As Executable Designs*, ACM. SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, p 167

[TAYLOR 82] Taylor T. and Standish T. A., *Initial Thoughts On Rapid Prototyping Techniques*, ACM. SIGSOFT SOFTWARE

Engineering Notes, Vol 7 N° 5, Dec 1982, pp 160 - 166

[TEICHROW 77] Teichrow D. and Hershey E.A., *PSL/PSA : A Computer Aided Technique For Structured Documentation And Analysis Of Information Processing Systems,* IEEE Trans. On Software Engineering, SE 3 (1), pp 41 - 48 , 1987

[TEITELBAUM 84] Teitelbaum T. and Reps T., *The Cornell Program Syntheziser : A Syntax - Directed Programming Environment,* In "Interactive Programming Environments", David R. Barstow, Howard E. Shrobe, Erik Sandewall (editors), Mc Graw - Hill Book Company, 1984

[VAN LAMSWEERDE 82] Van Lamsweerde A., *Les Outils D'Aide Au Développement De Logiciels - Un Aperçu Des Tendances Actuelles,* JIIA, Paris, 1982

[VAN LAMSWEERDE 86] Van Lamsweerde A., *Syntaxe concrète de RSL,* Mars 1986

[WASSERMAN 85] Wasserman A.I. and Shewmake D.T., *The Role Of Prototypes In The User Software Engineering (USE) Methodology,* In Hartson, Rex (Editor), Advances in Human-Computer Interaction 1, Ablex Publishing Corporation, Norwood, NJ , 1985, pp 191 - 210

[WEISER 82] Weiser M., *Scale Models And Rapid Prototyping,* ACM. SIGSOFT SOFTWARE Engineering Notes, Vol 7 N° 5, Dec 1982, pp 181 - 185

[WING 88] Wing J.M., *A Study Of 12 Specifications Of The Library Problem,* IEEE Software, July 1988

ANNEXE 1

Les syntaxes abstraites de RSL , FOPL , Prolog

Remarque:

Nous n'avons pas la prétention d'avoir donné les syntaxes abstraites complètes pour FOPL et Prolog. En fait seulement les parties qui nous sont nécessaires ont été utilisées pour ces deux syntaxes.

```
/*************************/
/*                       */
/*  Syntaxes Abstraites  */
/*                       */
/*************************/


/* 1.1. Definition des domaines syntaxiques des grammaires abstraites */
/*      ------------------------------------------------------------- */


/* 1.1.1. Definition des domaines syntaxiques de la syntaxe abstraite   */
/*        de RSL                                                        */
/*        ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''' */


root racine ;

racine          : Racine ( arbrersl arbrefopl arbreprolog )
                ;

arbrersl        : ArbreRSLVide ( )
                | ArbreRSL ( specification )
                ;

specification   : SpecifVide ( )
                | Specif ( suiteblocoper suitebloctypobj )
                ;

list suiteblocoper ;

suiteblocoper   : VideBlocOper ( )
                | ConsBlocOper ( blocoper suiteblocoper )
                ;

blocoper        : BlocOperVide ( )
                | BlocOper ( id decformop profilop lexop id )
                ;

decformop       : DecOperVide ( )
                | DecOper ( id sid exprop explicop expre )
                ;

expre           : ExPreVide ( )
                | ExPre( expreop prexplicop )
                ;

exprop          : ExprOpVide ( )
                | ExprOp ( id id sid )
                ;

expreop         : ExPreOpVide ( )
                | ExPreOp ( id sid )
                | ExPreOpExprOp (exprop )
                ;

prexplicop      : PrExplicOpVide ( )
                | NotPrExplicOp ( prexplicop )
                | BinOpExplicOp ( explicop binop explicop )
                | WithExplicOp ( prexplicop sexprinterm )
                | Egalite ( exprterminale exprterminale )
```

```
                | PrExplicOp ( expreop )
                ;

explicop        : ExplicOpVide ( )
                | ExplicOpPre ( prexplicop )
                | ForAll ( id id id explicop )
                | PreEtExplicOp ( pretexplicop )
                ;

exprterminale   : ExprTerminaleVide ( )
                | ExprTermElm ( termelm )
                | BibliOper ( biblioper )
                ;

termelm         : TermElmVide ( )
                | Constante ( cst )
                | NomObj ( id )
                ;

exprinterm      : ExprIntermVide ( )
                | ExprInterm ( exprop sexprinterm )
                | TerminInterm ( id exprterminale sexprinterm )
                ;

list sexprinterm ;

sexprinterm     : VideExprInterm ( )
                | ConsExprInterm ( exprinterm sexprinterm )
                ;

pretexplicop    : PrEtExplicOpVide ( )
                | PreEt ( prexplicop explicop )
                | PreSi ( sexsipre )
                | PreSinon ( sexsipre explicop )
                ;

exsipre         : ExSiPreVide ( )
                | ExSiPre ( explicop expreop )
                ;


list sexsipre ;

sexsipre        : VideExSiPre ( )
                | ConsExSiPre ( exsipre sexsipre )
                ;

predicat        : PredVide ( )
                | Atome ( atome )
                | UnPred ( UNPRED predicat )
                | BinPred ( predicat binpred predicat )
                | Quantif ( quantif id predicat )
                ;

atome           : AtomeVide ( )
                | AtFonc ( id sterme )
                | AtRel ( terme oprel terme )
                ;

list sterme ;

sterme          : VideTerme ( )
                | ConsTerme ( terme sterme )
                ;
```

```
terme           : TermeVide ( )
                | TermElmCste ( cst )
                | TermElmNomObj ( id )
                | TermFonc ( id sterme )
                | OpArithm ( terme OPARITH terme )
                ;

profilop        : ProfilOpVide ( )
                | ProfilOp ( sprofobj sprofop )
                ;

profobj         : ProfObjVide ( )
                | ProfObj ( sid id )
                ;

list sprofobj ;

sprofobj        : VideProfObj ( )
                | ConsProfObj ( profobj  sprofobj )
                ;

list sprofop ;

sprofop         : VideProfOp ( )
                | ConsProfOp ( profop sprofop )
                ;

profop          : ProfOpVide ( )
                | ProfOp ( sid defop )
                ;

defop           : DefOpVide ( )
                | DefOp ( sid id )
                | BiblioDef ( sid )
                ;

lexop           : LexOpVide ( )
                | LexOp ( slgntex sdestyp sdestyp sdestyp )
                ;

list slgntex ;

slgntex         : VideLgnText ( )
                | ConsLgnText ( lgntex slgntex )
                ;

lgntex          : LgnTexVide ( )
                | LgnTex ( STR )
                ;

list sdestyp ;

sdestyp         : VideDesTyp ( )
                | ConsDesTyp ( destyp sdestyp)
                ;

destyp          : DescrTypVide ( )
                | DescrTyp ( id slgntex )
                ;

list suitebloctypobj ;
```

```
suitebloctypobj : VideBlocObj ( )
                | ConsBlocObj ( bloctypobj suitebloctypobj )
                ;

bloctypobj      : BlocTypeVide ( )
                | BlocType ( id decformobj profilobj lexobj id )
                ;

decformobj      : DecFormObjVide ( )
                | DecFormObj ( sid id expltyp predicat )
                ;

expltyp         : ExplTypVide ( )
                | IdExplTyp ( id )
                | TypeBase ( STR )
                | NrExplTyp ( STR sexpltyp )
                | NrRoleExplTyp ( STR svarexpltyp )
                | UnExplTyp ( unexpltyp expltyp )
                | BinExplTyp ( STR expltyp expltyp )
                | ISAWithout ( unexpltyp expltyp sid )
                ;


list sexpltyp ;

sexpltyp        : VideExplTyp ( )
                | ConsExplTyp ( expltyp sexpltyp )
                ;

list svarexpltyp ;

svarexpltyp     : VideVarExplTyp ( )
                | ConsVarExplTyp ( varexpltyp svarexpltyp )
                ;

varexpltyp      : VarExplTypVide ( )
                | VarExplTyp ( id expltyp )
                ;

profilobj       : ProfilObjVide ( )
                | ProfilObj ( sprofop sprofobj )
                ;

lexobj          : LexObjVide ( )
                | LexObj ( slgntex sdestyp sdestyp slgntex )
                ;

id              : IdVide ( )
                | MinId ( STR )
                | MajId ( STR )
                ;

list sid ;

sid             : VideId ( )
                | ConsId ( id sid )
                ;

oprel           : OpRelVide ( )
                | Equal ( EGAL )
                | AutreRel ( OPREL )
                ;

binop           : BinOpVide ( )
```

```
                | Et ( AND )
                | Ou ( OR )
                ;

binpred         : BinPredVide ( )
                | BinPredOu ( OU )
                | BinPredEt ( ET )
                | BinPredImplique ( IMPLIQUE )
                | BinPredEquiv ( EQUIV )
                ;

quantif         : QuantifVide ( )
                | QuantifPourTout ( POURTOUT )
                | QuantifIlExiste ( ILEXISTE )
                ;

unexpltyp       : UnExplTypVide ( )
                | SeType ( STR )
                | Isa ( STR )
                ;

biblioper       : BibliOperVide ( )
                | Constuple ( sid )
                | ConstupleType ( idid )
                | Selection ( id id )
                | As ( id id )
                | Inj ( id id )
                | Is ( id id )
                | Filter ( id id sid )
                | Length (id )
                | Nieme ( id id )
                | Sempty ( )
                | ConsSuite ( id id )
                | In ( id id )
                ;

cst             : CsteVide ( )
                | CsteBool ( BOOL )
                | CsteEnt ( INT )
                ;

list idid       ;

idid            : VideIdId ( )
                | ConsIdId ( coupleidid idid )
                ;

coupleidid      : CoupleIdIdVide ( )
                | CoupleIdId ( id id )
                ;


/* 1.1.2. Definition des domaines syntaxiques de la syntaxe abstraite   */
/*        du langage des predicats du premier ordre                     */
/*        ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''' */


arbrefopl       : ArbreFOPLVide ( )
                | ArbreFOPL ( fform )
                ;

fform           : FFormVide ( )
```

```
                | FAtome ( fatome )
                | FFormNon ( fform )
                | FFormEt ( fform fform )
                | FFormOu ( fform fform )
                | FFormImplique ( fform fform )
                | FFormEquiv ( fform fform )
                | FFormQuantifExist ( id fform )
                | FFormQuantifPrTt ( id fform )
                ;

fatome          : FAtomeVide ( )
                | FPred ( id fsterme )
                | FEgal ( fterme fterme )
                ;

fterme          : FTermeVide ( )
                | FUnderScore ( )
                | FVariable ( id )
                | FFoncteur ( ffoncteur fsterme )
                | FConstuple ( fsterme )
                | FSel ( fterme fterme )
                | FAs ( fterme fterme )
                | FInj ( fterme fterme )
                | FIs ( fterme fterme )
                | FFilter ( fterme fform )
                | FLength ( fterme )
                | FIn ( id id )
                | FNieme ( id fterme )
                | FChampObj ( fterme fterme )
                | FSEmpty ( )
                | FCons ( fterme fterme )
                | FTypeBase ( STR )
                ;

list fsterme ;

fsterme         : VideFTerme ( )
                | ConsFTerme ( fterme fsterme )
                ;

ffoncteur       : FFoncteurVide ( )
                | FId ( id )
                | FBool ( BOOL )
                | FEnt ( INT )
                | FNrExplTyp ( STR )
                | FUnExplTyp ( unexpltyp )
                | FBinExplTyp ( STR )
                ;


/* 1.1.3. Definition des domaines syntaxiques de la syntaxe abstraite   */
/*        de Prolog                                                     */
/*        ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''' */


arbreprolog     : ArbrePROLOGVide ( )
                | ArbrePROLOG ( pprogramme )
                ;

pprogramme      : PProgrammeVide ( )
                | PProgramme ( suiteclauses )
                ;
```

```
list suiteclauses;

suiteclauses     : VideSuiteClauses ( )
                 | ConsSuiteClauses ( clause suiteclauses )
                 ;

clause           : ClauseVide ( )
                 | Clause ( pconseq psantec )
                 ;

pconseq          : PConseqVide ( )
                 | PConseq ( patome )
                 ;

patome           : PAtomeVide ( )
                 | PAtome ( fterme fterme )
                 ;

list psantec ;

psantec          : VideAntec ( )
                 | ConsAntec ( pantec psantec )
                 ;

pantec           : PAntecVide ( )
                 | PAntecSEmpty ( )
                 | PAntecEt ( pantec pantec )
                 | PAntec ( patome )
                 ;


/* 1.1.4. Definition des domaines syntaxiques auxiliaires */
/*        '''''''''''''''''''''''''''''''''''''''''''''''' */


list iffi ;

iffi             : VideIFFI ( )
                 | ConsIFFI ( quadriffi iffi )
                 ;

quadriffi        : QuadrIFFIVide ( )
                 | QuadrIFFI ( id fterme fterme INT )
                 ;

list fterfter ;

fterfter         : VideFTerFTer ( )
                 | ConsFTerFTer ( couplefterfter fterfter )
                 ;

couplefterfter   : CoupleFTerFTerVide ( )
                 | CoupleFTerFTer ( fterme fterme )
                 ;

list idfterme ;

idfterme         : VideIdFTerme ( )
                 | ConsIdFTerme ( coupleidfterme idfterme )
                 ;
```

```
coupleidfterme  : CoupleIdFTermeVide ( )
                | CoupleIdFTerme ( id fterme )
                ;

list idfterint ;

idfterint       : VideIdFTerInt ( )
                | ConsIdFTerInt ( triplidfterint idfterint )
                ;

triplidfterint  : TriplIdFTerIntVide ( )
                | TriplIdFTerInt ( id fterme INT )
                ;

list ifterfterpsantec ;

ifterfterpsantec        : VideIFTerFTerPSAntec ( )
                        | ConsIFTerFTerPSAntec
                                ( triplifterfterpsantec ifterfterpsantec )
                        ;

triplifterfterpsantec   : TriplIFTerFTerPSAntecVide ( )
                        | TriplIFTerFTerPSAntec(fterfter psantec listefterfter)
                        ;

list listefterfter ;

listefterfter           : VideListeFTerFTer ( )
                        | ConsListeFTerFTer ( fterfter listefterfter )
                        ;


/* 1.2. Definition des symboles terminaux */
/*      ---------------------------------- */


OR              : Orlex < "or" >
                ;

AND             : Andlex < "and" >
                ;

ANDPRE          : AndPrelex < "AND" >
                ;

NOT             : Notlex < "not" >
                ;

UNPRED          : Nonlex < "!" >
                ;

OU              : OuLex < "||" >
                ;

ET              : EtLex < "&&" >
                ;

IMPLIQUE        : ImpliqueLex < "=>" >
                ;

EQUIV           : EquivLex < "<=>" >
                ;
```

```
POURTOUT        : PourToutLex < "for_all" >
                ;

ILEXISTE        : IlExisteLex < "there_is" >
                ;

EGAL            : EgalLex < "=" >
                ;

OPREL           : PlusPetitLex < "<" >
                | PlusGrandLex < ">" >
                | DifferentLex < "><" >
                | PlusPetitEgalLex < "<=" >
                | PlusGrandEgalLex < ">=" >
                ;

OPARITH         : PlusLex < "+" >
                | MoinsLex < "-" >
                | FoisLex < "*" >
                | DivLex < "/" >
                | ModLex < "mod" >
                | DivEntLex < "div" >
                ;

TYPEBASE        : IntLex < "INT" >
                | NatLex < "NAT" >
                | CharLex < "CHAR" >
                | CharstLex < "CHARST" >
                | BoolLex < "BOOL" >
                | DateLex < "DATE" >
                ;

NREXPLTYP       : UnionLex < "UNION" >
                | CPLex < "CP" >
                ;

BINEXPLTYP      : TabLex < "TABLE" >
                ;

SETYP           : SeqLex < "SEQ" >
                | SetLex < "SET" >
                ;

ISA             : ISALex < "IS-A" >
                ;

SPECOP          : SpecOpLex < "SPECIFICATION\ OF\ OPERATION" >
                ;

FORMDEC         : FormDecLex < "FORMAL\ DESCRIPTION:" >
                ;

OUTLINE         : OutlineLex < "OUTLINE:" >
                ;

END             : EndLex < "END" >
                ;

RESULT          : ResultLex < "RESULT:" >
                ;

ARG             : ArgLex < "ARGUMENTS:" >
                ;
```

```
EXPLIO          : ExplIOLex < "EXPLICITATION\ OF\ INPUT-OUTPUT\ ASSERTION:" >
                ;

EXPLPRE         : ExplPreLex < "EXPLICITATION\ OF\ PRECONDITION:" >
                ;

IN              : InLex < "IN" >
                ;

IF              : IfLex < "if" >
                ;

WITH            : WithLex < "with" >
                ;

WITHOUT         : WithoutLex < "without" >
                ;

OTHER           : OtherwiseLex < "otherwise" >
                ;

FORALL          : ForallLex < "forall" >
                ;

TYPE            : TypeLex < "TYPES:" >
                ;

OPER            : OperLex < "OPERATIONS:" >
                ;

CROSS           : CrossLex < "X" >
                ;

INTOOL          : InToolLex < "IN\ TOOLBOX" >
                ;

OBJECTIVE       : ObjectiveLex < "OBJECTIVE:" >
                ;

OBJECTS         : ObjectsLex < "OBJECTS:" >
                ;

PREC            : PrecLex < "PRECONDITIONS:" >
                ;

SPECOBJ         : SpecObjLex < "SPECIFICATION\ OF\ OBJECT\ TYPE" >
                ;

LEXIC           : LexicLex < "LEXICON:" >
                ;

ASSOP           : AssOpLex < "ASSOCIATED\ OPERATIONS:" >
                ;

EXPTYPSTRUCT    : ExpTypStructLex < "EXPLICITATION\ OF\ TYPE\ STRUCTURE:" >
                ;

EXPLINV         : ExplInvLex < "EXPLICITATION\ OF\ INVARIANT:" >
                ;

INV             : InvLex < "INVARIANTS:" >
                ;
```

```
CONSTUPLE       : ConstupleLex < "CONSTUPLE" >
                ;

SELECT          : SelectLex < "SELECT" >
                ;

INJ             : InjLex < "INJ" >
                ;

IS              : IsLex < "IS" >
                ;

AS              : AsLex < "AS" >
                ;

FILTER          : FilterLex < "FILTER" >
                ;

LENGTH          : LenghtLex < "LENGTH" >
                | DieseLex < "#" >
                ;

ITH             : IthLex < "ITH" >
                ;

SEMPTY          : SemptyLex < "SEMPTY" >
                ;

CONS            : ConsLex < "CONS" >
                ;

POINT           : PointLex < "." >
                ;

VIR             : VirLex < "," >
                ;

FLECHE          : FlecheLex < "->" >
                ;

DEUXPTS         : DeuxPtsLex < ":" >
                ;

PAROUV          : ParOuvLex < "(" >
                ;

PARFER          : ParFerLex < ")" >
                ;

CROUV           : CrOuvLex < "[" >
                ;

CRFER           : CrFerLex < "]" >
                ;

BOOLEEN         : VraiLex < "true" >
                | FauxLex < "false" >
                ;

IDMIN           : IdMinLex < [a-z][a-z0-9_]* >
                ;

IDMAJ           : IdMajLex < [A-Z][A-Z0-9_]* >
                ;
```

```
ENTIER          : EntierLex < [0-9]+ >
                ;

WHITESPACE      : WtspcLex < [\ \t\n]* >
                ;

STRING          : StringLex < [.] >
                ;

let format_strings = true ;
```

ANNEXE 2

Passage de la syntaxe concrète de RSL à la syntaxe abstraite de RSL

```
/********************/
/*                  */
/* Syntaxe concrete */
/*                  */
/********************/



/* Priorites */
/* --------- */



left EQUIV      ;
left IMPLIQUE   ;
left OR , OU    ;
left AND , ET   ;
left IF         ;



/* Definition des attributs de la syntaxe concrete */
/* ----------------------------------------------- */



yracine          { syn racine a ; } ;
yarbrersl        { syn arbrersl a ; } ;
yspecification   { syn specification a ; } ;
ysuiteblocoper   { syn suiteblocoper a ; } ;
yblocoper        { syn blocoper a ; } ;
ydecformop       { syn decformop a ; } ;
yexprop          { syn exprop a ; } ;
yexpreop         { syn expreop a ; } ;
yprexplicop      { syn prexplicop a ; } ;
yexplicop        { syn explicop a ; } ;
yexprterminale   { syn exprterminale a ; } ;
yexprinterm      { syn exprinterm a ; } ;
ysexprinterm     { syn sexprinterm a ; } ;
ypretexplicop    { syn pretexplicop a ; } ;
ysexsipre        { syn sexsipre a ; } ;
ypredicat        { syn predicat a ; } ;
yatome           { syn atome a ; } ;
ysterme          { syn sterme a ; } ;
yterme           { syn terme  a ; } ;
yprofilop        { syn profilop a ; } ;
yprofobj         { syn profobj a ; } ;
ysprofobj        { syn sprofobj a ; } ;
ysprofop         { syn sprofop a ; } ;
yprofop          { syn profop a ; } ;
ydefop           { syn defop a ; } ;
ylexop           { syn lexop a ; } ;
yslgntex         { syn slgntex a ; } ;
ysobjlex         { syn sdestyp a ; } ;
yobjlex          { syn destyp a ; } ;
ysmajlex         { syn sdestyp a ; } ;
ymajlex          { syn destyp a ; } ;
ysuitebloctypobj { syn suitebloctypobj a ; } ;
ybloctypobj      { syn bloctypobj a ; } ;
ydecobj          { syn decformobj a ; } ;
yexpltyp         { syn expltyp a ; } ;
ysexpltyp        { syn sexpltyp a ; } ;
ysvarexpltyp     { syn svarexpltyp a ; } ;
yprofilobj       { syn profilobj a ; } ;
```

```
ylexobj          { syn lexobj a ; } ;
ybibliodef       { syn id a ; } ;
ysbibliodef      { syn sid a ; } ;
ycrossid         { syn sid a ; } ;
yidmaj           { syn id a ; } ;
yidmin           { syn id a ; } ;
ysidmaj          { syn sid a ; } ;
ysidmin          { syn sid a ; } ;
yidid            { syn idid a ; } ;
yunexpltyp       { syn unexpltyp a ; } ;
ybiblioper       { syn biblioper a ; } ;
yoprel           { syn oprel a ; } ;
ycst             { syn cst a ; } ;


/* Fonction auxiliaire pour la syntaxe concrete */
/* -------------------------------------------- */


BOOL STRtoBOOL ( BOOLEEN a )
                 { ( a  == "true" ) ? true : false
                 } ;


/* Regles de grammaire de la syntaxe concrete */
/* ------------------------------------------ */


yracine          ::= ( yarbrersl )
        { $$.a = Racine ( yarbrersl.a , ArbreFOPLVide ( ) ,
                          ArbrePROLOGVide ( ) ) ; }
                 ;

racine ~ yracine.a ;

yarbrersl        ::= ( yspecification )
        { $$.a = ArbreRSL ( yspecification.a ) ; }
                 ;

arbrersl ~ yarbrersl.a ;

yspecification   ::= ( ysuiteblocoper ysuitebloctypobj )
        { $$.a = Specif ( ysuiteblocoper.a , ysuitebloctypobj.a ) ; }
                 ;

specification ~ yspecification.a ;

ysuiteblocoper ::= ( )
        { $$.a = VideBlocOper ( ) ; }
                 | ( yblocoper ysuiteblocoper )
        { $$.a = ConsBlocOper ( yblocoper.a , ysuiteblocoper$2.a ) ; }
                 ;

suiteblocoper ~ ysuiteblocoper.a ;

yblocoper        ::= ( SPECOP yidmaj FORMDEC ydecformop OUTLINE yprofilop
                           LEXIC ylexop END yidmaj POINT )
        { $$.a = BlocOper ( yidmaj$1.a , ydecformop.a , yprofilop.a ,
                           ylexop.a , yidmaj$2.a ) ; }
                 ;
```

```
blocoper ~ yblocoper.a ;

ydecformop      ::= ( RESULT yidmin ARG yidmin ysidmin EXPLIO yexprop FLECHE
                        yexplicop )
        { $$.a = DecOper ( yidmin$1.a , ConsId ( yidmin$2.a , ysidmin.a ) ,
                        yexprop.a , yexplicop.a , ExPreVide ( ) ) ; }
                | ( RESULT yidmin ARG yidmin ysidmin EXPLIO yexprop FLECHE
                        yexplicop EXPLPRE yexpreop FLECHE yprexplicop )
        { $$.a = DecOper ( yidmin$1.a , ConsId ( yidmin$2.a , ysidmin.a ) ,
                        yexprop.a , yexplicop.a ,
                        ExPre ( yexpreop.a , yprexplicop.a ) ) ; }
                ;

decformop ~ ydecformop.a ;

yexprop         ::= ( yidmin EGAL yidmaj PAROUV yidmin ysidmin PARFER )
        { $$.a = ExprOp ( yidmin$1.a, yidmaj.a ,
                        ConsId ( yidmin$2.a , ysidmin.a ) ) ; }
                ;

exprop ~ yexprop.a ;

yexpreop        ::= ( yexprop )
        { $$.a = ExPreOpExprOp ( yexprop.a ) ; }
                | ( yidmaj PAROUV yidmin ysidmin PARFER )
        { $$.a = ExPreOp ( yidmaj.a , ConsId ( yidmin.a , ysidmin.a ) ) ; }
                ;

expreop ~ yexpreop.a ;

yexplicop       ::= ( yprexplicop )
        { $$.a = ExplicOpPre ( yprexplicop.a ) ; }
                | ( PAROUV yexplicop PARFER )
        { $$.a = yexplicop$2.a ; }
                | ( FORALL yidmin DEUXPTS IN PAROUV yidmin VIR yidmin PARFER
                                DEUXPTS yexplicop )
        { $$.a = ForAll ( yidmin$1.a , yidmin$2.a , yidmin$3.a ,
                        yexplicop$2.a ) ; }
                | ( ypretexplicop )
        { $$.a = PreEtExplicOp ( ypretexplicop.a ) ; }
                ;

explicop ~ yexplicop.a ;

yprexplicop     ::= ( yexpreop )
        { $$.a = PrExplicOp ( yexpreop.a ) ; }
                | ( NOT PAROUV yprexplicop PARFER )
        { $$.a = NotPrExplicOp ( yprexplicop$2.a ) ; }
                | ( PAROUV yprexplicop PARFER )
        { $$.a = yprexplicop$2.a ; }
                | ( yexprterminale EGAL yexprterminale )
        { $$.a = Egalite ( yexprterminale$1.a , yexprterminale$2.a ) ; }
                | ( yexplicop AND yexplicop prec AND )
        { $$.a = BinOpExplicOp ( yexplicop$1.a , Et (AND) , yexplicop$2.a ) ; }
                | ( yexplicop OR yexplicop prec OR )
        { $$.a = BinOpExplicOp ( yexplicop$1.a , Ou (OR) , yexplicop$2.a ) ; }
                | ( yprexplicop yexprinterm ysexprinterm )
        { $$.a = WithExplicOp ( yprexplicop$2.a ,
                        ConsExprInterm ( yexprinterm.a , ysexprinterm.a ) ) ; }
                ;

prexplicop ~ yprexplicop.a ;
```

```
ysexprinterm    ::= ( )
        { $$.a = VideExprInterm ( ) ; }
                | ( yexprinterm AND ysexprinterm prec AND )
        { $$.a = ConsExprInterm ( yexprinterm.a , ysexprinterm$2.a ) ; }
                ;

sexprinterm ~ ysexprinterm.a ;

yexprterminale  ::= ( ycst )
        { $$.a = ExprTermElm ( Constante ( ycst.a ) ) ; }
                | ( yidmin )
        { $$.a = ExprTermElm ( NomObj ( yidmin.a ) ) ; }
                | ( ybiblioper )
        { $$.a = BibliOper ( ybiblioper.a ) ; }
                ;

exprterminale ~ yexprterminale.a ;

yexprinterm     ::= ( WITH yexprop ysexprinterm )
        { $$.a = ExprInterm ( yexprop.a , ysexprinterm.a ) ; }
                | ( WITH yexprop yexprinterm ysexprinterm )
        { $$.a = ExprInterm ( yexprop.a ,
                        ConsExprInterm ( yexprinterm$2.a , ysexprinterm.a )); }
                | ( WITH yidmin EGAL ybiblioper yexprinterm ysexprinterm )
        { $$.a = TerminInterm ( yidmin.a , BibliOper ( ybiblioper.a ) ,
                        ConsExprInterm ( yexprinterm$2.a , ysexprinterm.a )); }
                | ( WITH yidmin EGAL yexprterminale ysexprinterm )
        { $$.a = TerminInterm ( yidmin.a , yexprterminale.a ,
                        ysexprinterm.a ) ; }
                | ( WITH PAROUV yexprop ysexprinterm PARFER )
        { $$.a = ExprInterm ( yexprop.a , ysexprinterm.a ) ; }
                | ( WITH PAROUV yexprop yexprinterm ysexprinterm PARFER )
        { $$.a = ExprInterm ( yexprop.a ,
                        ConsExprInterm ( yexprinterm$2.a , ysexprinterm.a )); }
                | ( WITH PAROUV yidmin EGAL ybiblioper yexprinterm
                        ysexprinterm PARFER )
        { $$.a = TerminInterm ( yidmin.a , BibliOper ( ybiblioper.a ) ,
                        ConsExprInterm ( yexprinterm$2.a , ysexprinterm.a )); }
                | ( WITH PAROUV yidmin EGAL yexprterminale ysexprinterm
                        PARFER )
        { $$.a = TerminInterm ( yidmin.a , yexprterminale.a ,
                        ysexprinterm.a ) ; }
                ;

exprinterm ~ yexprinterm.a ;

ysexsipre       ::= ( yexplicop IF yexpreop )
        { $$.a = ConsExSiPre ( ExSiPre ( yexplicop.a , yexpreop.a ) ,
                        VideExSiPre ( ) ) ; }
                | ( yexplicop IF yexpreop VIR ysexsipre prec IF )
        { $$.a = ConsExSiPre ( ExSiPre ( yexplicop.a , yexpreop.a ) ,
                        ysexsipre$2.a ) ; }
                ;

sexsipre ~ ysexsipre.a ;

ypretexplicop   ::= ( yprexplicop ANDPRE yexplicop )
        { $$.a = PreEt ( yprexplicop.a , yexplicop.a ) ; }
                | ( ysexsipre )
        { $$.a = PreSi ( ysexsipre.a ) ; }
                | ( ysexsipre VIR yexplicop OTHER )
        { $$.a = PreSinon ( ysexsipre.a , yexplicop.a ) ; }
                ;
```

```
pretexplicop ~ ypretexplicop.a ;

ypredicat       ::= ( yatome )
        { $$.a = Atome ( yatome.a ) ; }
                | ( PAROUV ypredicat PARFER )
        { $$.a = ypredicat$2.a ; }
                | ( UNPRED ypredicat )
        { $$.a = ypredicat$2.a ; }
                | ( ypredicat ET ypredicat prec ET )
        { $$.a = BinPred ( ypredicat$2.a , BinPredEt ( ET ) ,
                        ypredicat$3.a ) ; }
                | ( ypredicat OU ypredicat prec OU )
        { $$.a = BinPred ( ypredicat$2.a , BinPredOu ( OU ) ,
                        ypredicat$3.a ) ; }
                | ( ypredicat IMPLIQUE ypredicat prec IMPLIQUE )
        { $$.a = BinPred ( ypredicat$2.a , BinPredImplique ( IMPLIQUE ) ,
                        ypredicat$3.a ) ; }
                | ( ypredicat EQUIV ypredicat prec EQUIV )
        { $$.a = BinPred ( ypredicat$2.a , BinPredEquiv ( EQUIV ) ,
                        ypredicat$3.a ) ; }
                | ( PAROUV POURTOUT yidmin PARFER ypredicat )
        { $$.a = Quantif ( QuantifPourTout ( POURTOUT ) , yidmin.a ,
                        ypredicat$2.a ) ; }
                | ( PAROUV ILEXISTE yidmin PARFER ypredicat )
        { $$.a = Quantif ( QuantifIlExiste ( ILEXISTE ) , yidmin.a ,
                        ypredicat$2.a ) ; }
                ;

predicat ~ ypredicat.a ;

yatome          ::= ( yidmaj PAROUV yterme ysterme PARFER )
        { $$.a = AtFonc ( yidmaj.a, ConsTerme ( yterme.a , ysterme.a ) ) ; }
                | ( yterme yoprel yterme )
        { $$.a = AtRel ( yterme$1.a , yoprel.a , yterme$2.a ) ; }
                ;

atome ~ yatome.a ;

ysterme         ::= ( )
        { $$.a = VideTerme ( ) ; }
                | ( VIR yterme ysterme )
        { $$.a = ConsTerme ( yterme.a , ysterme$2.a ) ; }
                ;

sterme ~ ysterme.a ;

yterme          ::= ( ycst )
        { $$.a = TermElmCste ( ycst.a ) ; }
                | ( yidmin )
        { $$.a = TermElmNomObj ( yidmin.a ) ; }
                | ( PAROUV yterme PARFER )
        { $$.a = yterme$2.a ; }
                | ( yidmaj PAROUV yterme ysterme PARFER )
        { $$.a = TermFonc ( yidmaj.a ,
                        ConsTerme ( yterme$2.a , ysterme.a ) ) ; }
                | ( yterme OPARITH yterme )
        { $$.a = OpArithm ( yterme$2.a , OPARITH , yterme$3.a ) ; }
                ;

terme ~ yterme.a ;

yprofilop       ::= ( TYPE yprofobj ysprofobj OPER ysprofop )
        { $$.a = ProfilOp ( ConsProfObj ( yprofobj.a , ysprofobj.a ) ,
                        ysprofop.a ) ; }
```

```
                ;

profilop ~ yprofilop.a ;

ysprofobj       ::= ( )
        { $$.a = VideProfObj ( ) ; }
                |  ( VIR yprofobj ysprofobj )
        { $$.a = ConsProfObj ( yprofobj.a , ysprofobj$2.a ) ; }
                ;

sprofobj ~ ysprofobj.a ;

ysprofop        ::= ( )
        { $$.a = VideProfOp ( ) ; }
                | ( yprofop ysprofop )
        { $$.a = ConsProfOp ( yprofop.a , ysprofop$2.a ) ; }
                ;

sprofop ~ ysprofop.a ;

yprofobj        ::= ( yidmin ysidmin DEUXPTS yidmaj )
        { $$.a = ProfObj ( ConsId ( yidmin.a , ysidmin.a ) , yidmaj.a ) ; }
                ;

profobj ~ yprofobj.a ;

yprofop         ::= ( ysidmaj DEUXPTS ydefop )
        { $$.a = ProfOp ( ysidmaj.a , ydefop.a ) ; }
                ;

profop ~ yprofop.a ;

ydefop          ::= ( ycrossid FLECHE yidmaj )
        { $$.a = DefOp ( ycrossid.a , yidmaj.a ) ; }
                | ( ysbibliodef DEUXPTS INTOOL )
        { $$.a = BiblioDef ( ysbibliodef.a ) ; }
                ;

defop ~ ydefop.a ;

ybibliodef      ::= ( CONSTUPLE )
        { $$.a = IdVide ( ) ; }
                | ( PAROUV DEUXPTS PARFER )
        { $$.a = IdVide ( ) ; }
                | ( SELECT )
        { $$.a = IdVide ( ) ; }
                | ( AS PAROUV PARFER )
        { $$.a = IdVide ( ) ; }
                | ( INJ PAROUV PARFER )
        { $$.a = IdVide ( ) ; }
                | ( IS PAROUV PARFER )
        { $$.a = IdVide ( ) ; }
                | ( FILTER )
        { $$.a = IdVide ( ) ; }
                | ( LENGTH )
        { $$.a = IdVide  ( ) ; }
                | ( ITH )
        { $$.a = IdVide ( ) ; }
                | ( SEMPTY )
        { $$.a = IdVide ( ) ; }
                | ( CONS )
        { $$.a = IdVide ( ) ; }
                | ( IN )
        { $$.a = IdVide ( ) ; }
```

```
                ;

id ~ ybibliodef.a ;

ysbibliodef     ::= ( ybibliodef )
        { $$.a = VideId ( ) ; }
                | ( ybibliodef VIR ysbibliodef )
        { $$.a = ConsId ( ybibliodef.a , ysbibliodef$2.a ) ; }
                ;

sid ~ ysbibliodef.a ;

ylexop          ::= ( OBJECTIVE yslgntex OBJECTS OPER ysmajlex
                        PREC ysmajlex )
        { $$.a = LexOp ( yslgntex.a , VideDesTyp ( ) , ysmajlex$1.a ,
                        ysmajlex$2.a) ; }
                | ( OBJECTIVE yslgntex OBJECTS ysobjlex OPER ysmajlex
                        PREC ysmajlex )
        { $$.a = LexOp ( yslgntex.a , ysobjlex.a , ysmajlex$1.a ,
                        ysmajlex$2.a ) ; }
                ;

lexop ~ ylexop.a ;

yslgntex        ::= ( )
        { $$.a = VideLgnText ( ) ; }
                | (STRING yslgntex )
        { $$.a = ConsLgnText ( LgnTex ( STRING ) , yslgntex$2.a ) ; }
                ;

slgntex ~ yslgntex.a ;

ysobjlex        ::= ( yobjlex )
        { $$.a = ConsDesTyp ( yobjlex.a , VideDesTyp ( ) ) ; }
                | ( yobjlex ysobjlex )
        { $$.a = ConsDesTyp ( yobjlex.a , ysobjlex$2.a ) ; }
                ;

sdestyp ~ ysobjlex.a ;

yobjlex         ::= ( yidmin DEUXPTS STRING yslgntex )
        { $$.a = DescrTyp ( yidmin.a ,
                        ConsLgnText ( LgnTex ( STRING ) , yslgntex.a ) ) ; }
                ;

destyp ~ yobjlex.a ;

ysmajlex        ::= ( )
        { $$.a = VideDesTyp ( ) ; }
                | ( ymajlex ysmajlex )
        { $$.a = ConsDesTyp ( ymajlex.a , ysmajlex$2.a ) ; }
                ;

sdestyp ~ ysmajlex.a ;

ymajlex         ::= ( yidmaj DEUXPTS STRING yslgntex )
        { $$.a = DescrTyp ( yidmaj.a , ConsLgnText ( LgnTex ( STRING ) ,
                        yslgntex.a ) ) ; }
                ;

destyp ~ ymajlex.a ;

ysuitebloctypobj ::= ( )
        { $$.a = VideBlocObj ( ) ; }
```

```
                | ( ybloctypobj ysuitebloctypobj )
        { $$.a = ConsBlocObj ( ybloctypobj.a , ysuitebloctypobj$2.a ) ; }
                ;

suitebloctypobj ~ ysuitebloctypobj.a ;

ybloctypobj     ::= ( SPECOBJ yidmaj FORMDEC ydecobj OUTLINE yprofilobj
                        LEXIC ylexobj END yidmaj POINT )
        { $$.a = BlocType ( yidmaj$1.a , ydecobj.a , yprofilobj.a ,
                        ylexobj.a , yidmaj$2.a ) ; }
                ;

ydecobj         ::= ( ASSOP ysidmaj EXPTYPSTRUCT yidmaj FLECHE yexpltyp )
        { $$.a = DecFormObj ( ysidmaj.a , yidmaj.a ,
                        yexpltyp.a , PredVide ( ) ) ; }
                | ( ASSOP ysidmaj EXPTYPSTRUCT yidmaj FLECHE yexpltyp
                        EXPLINV ypredicat )
        { $$.a = DecFormObj ( ysidmaj.a , yidmaj.a ,
                        yexpltyp.a , ypredicat.a ) ; }
                ;

decformobj ~ydecobj.a ;

yexpltyp        ::=  ( yidmaj )
        { $$.a = IdExplTyp ( yidmaj.a ) ; }
                | ( TYPEBASE )
        { $$.a = TypeBase ( TYPEBASE ) ; }
                | ( NREXPLTYP CROUV ysexpltyp CRFER )
        { $$.a = NrExplTyp ( NREXPLTYP , ysexpltyp.a ) ; }
                | ( NREXPLTYP CROUV ysvarexpltyp CRFER )
        { $$.a = NrRoleExplTyp ( NREXPLTYP , ysvarexpltyp.a ) ; }
                | ( BINEXPLTYP CROUV yexpltyp FLECHE yexpltyp CRFER )
        { $$.a = BinExplTyp ( BINEXPLTYP , yexpltyp$2.a , yexpltyp$3.a ) ; }
                | ( yunexpltyp CROUV yexpltyp CRFER )
        { $$.a = UnExplTyp ( yunexpltyp.a , yexpltyp$2.a ) ; }
                | ( ISA CROUV yexpltyp CRFER WITHOUT yidmin ysidmin )
        { $$.a = ISAWithout ( Isa ( ISA ) , yexpltyp$2.a ,
                        ConsId ( yidmin.a , ysidmin.a ) ) ; }
                ;

expltyp ~ yexpltyp.a ;

ysexpltyp       ::= ( yexpltyp )
        { $$.a = ConsExplTyp ( yexpltyp.a , VideExplTyp ( ) ) ; }
                | ( yexpltyp VIR ysexpltyp )
        { $$.a = ConsExplTyp ( yexpltyp.a , ysexpltyp$2.a ) ; }
                ;

sexpltyp ~ ysexpltyp.a ;

ysvarexpltyp    ::= ( yidmin DEUXPTS yexpltyp )
        { $$.a = ConsVarExplTyp ( VarExplTyp ( yidmin.a , yexpltyp.a ) ,
                        VideVarExplTyp ( ) ) ; }
                | ( yidmin DEUXPTS yexpltyp VIR ysvarexpltyp )
        { $$.a = ConsVarExplTyp ( VarExplTyp ( yidmin.a , yexpltyp.a ) ,
                        ysvarexpltyp$2.a ) ; }
                ;

svarexpltyp ~ ysvarexpltyp.a ;

yprofilobj      ::= ( OPER ysprofop TYPE yprofobj ysprofobj )
        { $$.a = ProfilObj ( ysprofop.a , ConsProfObj ( yprofobj.a ,
                        ysprofobj.a ) ) ; }
                ;
```

```
profilobj ~ yprofilobj.a ;

ylexobj          ::= ( OBJECTIVE yslgntex OPER ysmajlex TYPE ysmajlex
                                    INV yslgntex )
        { $$.a = LexObj ( yslgntex$1.a , ysmajlex$1.a , ysmajlex$2.a ,
                                    yslgntex$2.a ) ; }
                ;

lexobj ~ ylexobj.a ;

yunexpltyp       ::= ( SETYP )
        { $$.a = SeType ( SETYP ) ; }
                | ( ISA )
        { $$.a = Isa ( ISA ) ; }
                ;

unexpltyp ~ yunexpltyp.a ;

ycrossid         ::= ( yidmaj )
        { $$.a = ConsId ( yidmaj.a , VideId ( ) ) ; }
                | ( yidmaj CROSS ycrossid )
        { $$.a = ConsId ( yidmaj.a , ycrossid$2.a ) ; }
                ;

ysidmaj          ::=  ( yidmaj )
        { $$.a = ConsId ( yidmaj.a , VideId ( ) ) ; }
                | ( yidmaj VIR ysidmaj )
        { $$.a = ConsId ( yidmaj.a , ysidmaj$2.a ) ; }
                ;

sid ~ ysidmaj.a ;

ysidmin          ::=  ( )
        { $$.a = VideId ( ) ; }
                | ( VIR yidmin ysidmin )
        { $$.a = ConsId ( yidmin.a , ysidmin$2.a ) ; }
                ;

sid ~ ysidmin.a ;

yidmin           ::= ( IDMIN )
        { $$.a = MinId ( IDMIN ) ; }
                ;

id ~ yidmin.a ;

yidmaj           ::= ( IDMAJ )
        { $$.a = MajId ( IDMAJ ) ; }
                ;

id ~ yidmaj.a ;

yoprel           ::=  ( EGAL )
        { $$.a = Equal ( EGAL ) ; }
                | ( OPREL )
        { $$.a = AutreRel ( OPREL ) ; }
                ;

oprel ~ yoprel.a ;

ybiblioper       ::= ( CONSTUPLE PAROUV yidmin ysidmin PARFER )
        { $$.a = Constuple ( ConsId ( yidmin.a , ysidmin.a ) ) ; }
                | ( CONSTUPLE PAROUV yidid PARFER )
```

```
        { $$.a = ConstupleType ( yidid.a ) ; }
                | ( yidmaj PAROUV DEUXPTS yidmin PARFER )
        { $$.a = Selection ( yidmaj.a , yidmin.a ) ; }
                | ( SELECT PAROUV yidmaj VIR yidmin PARFER )
        { $$.a = Selection ( yidmaj.a , yidmin.a ) ; }
                | ( AS PAROUV yidmaj VIR yidmin PARFER )
        { $$.a = As ( yidmaj.a , yidmin.a ) ; }
                | ( INJ PAROUV yidmaj VIR yidmin PARFER )
        { $$.a = Inj ( yidmaj.a , yidmin.a ) ; }
                | ( IS PAROUV yidmaj VIR yidmaj PARFER )
        { $$.a = Is ( yidmaj$1.a , yidmaj$2.a ) ; }
                | ( FILTER PAROUV yidmin VIR yidmaj PAROUV yidmin ysidmin
                        PARFER PARFER )
        { $$.a = Filter ( yidmin$1.a , yidmaj.a , ConsId ( yidmin$2.a ,
                        ysidmin.a ) ) ; }
                | ( LENGTH PAROUV yidmin PARFER )
        { $$.a = Length ( yidmin.a ) ; }
                | ( ITH PAROUV yidmin VIR yidmin PARFER )
        { $$.a = Nieme ( yidmin$1.a , yidmin$2.a ) ; }
                | ( SEMPTY )
        { $$.a = Sempty ( ) ; }
                | ( CONS PAROUV yidmin VIR yidmin PARFER )
        { $$.a = ConsSuite ( yidmin$1.a , yidmin$2.a ) ; }
                | ( IN PAROUV yidmin VIR yidmin PARFER )
        { $$.a = In ( yidmin$1.a , yidmin$2.a ) ; }
                ;

biblioper ~ ybiblioper.a ;

ycst            ::= ( ENTIER )
        { $$.a = CsteEnt ( STRtoINT ( ENTIER ) ) ; }
                | ( BOOLEEN )
        { $$.a = CsteBool ( STRtoBOOL ( BOOLEEN ) ) ; }
                ;

cst ~ ycst.a ;

yidid           ::= ( yidmaj DEUXPTS yidmin )
        { $$.a = ConsIdId ( CoupleIdId ( yidmaj.a , yidmin.a ) ,
                        VideIdId ( ) ) ; }
                | ( yidmaj DEUXPTS yidmin VIR yidid )
        { $$.a = ConsIdId ( CoupleIdId ( yidmaj.a , yidmin.a ) ,
                        yidid$2.a ) ; }
                ;

idid ~ yidid.a ;
```

ANNEXE 3

Les schémas de décompilation des parties utiles des syntaxes abstraites de RSL , FOPL et Prolog

```
/****************************/
/*                          */
/* Schemas de decompilation */
/*                          */
/****************************/



/* RSL */
/* --- */



racine          : Racine
                        [ @ ::= @ "%n%n%n%n" @ "%n%n%n%n" @ ]
                ;

arbrersl        : ArbreRSLVide
                        [ @ ::= "" ]
                | ArbreRSL
                        [ @ ::= @ ]
                ;

specification   : SpecifVide
                        [ @ ::= "" ]
                | Specif
                        [ @ ::= "SPECIFICATIONS%n%n%n%n%n%n" @
                                "%n%n%n%n%n%n" @ "%n" ]
                ;

suiteblocoper   : VideBlocOper
                        [ @ ::= "" ]
                | ConsBlocOper
                        [ @ ::= "%n%n" @ @ "%n" ]
                ;

blocoper        : BlocOperVide
                        [ @ ::= "" ]
                | BlocOper
                        [ @ ::= "SPECIFICATION OF OPERATION " @
                                "%nFORMAL DESCRIPTION : %t%n" @
                                "%b%nOUTLINE : %t%n" @ "%b%nLEXICON : %t%n" @
                                "%b%nEND " @ " .%n" ]
                ;

decformop       : DecOperVide
                        [ @ ::= "" ]
                | DecOper
                        [ @ ::= "RESULT :" @ "%nARGUMENTS : %t%t%n" @
                          "%b%b%nEXPLICITATION OF INPUT-OUTPUT ASSERTION :%t%n"
                           @ "%t%t%n->%t%t%n" @ "%b%b%b%b%b%n" @ "%n" ]
                ;

expre           : ExPreVide
                        [ @ ::= "%n" ]
                | ExPre
                        [ @ ::= "EXPLICITATION OF PRECONDITION :%t%n" @
                                "%t%t%n->%t%t%n" @ "%b%b%b%b%b%n" ]
                ;

exprop          : ExprOpVide
                        [ @ ::= "" ]
                | ExprOp
                        [ @ ::= @ " = " @ " ( " @ " )%n" ]
```

```
                ;

expreop         : ExPreOpVide
                      [ @ ::= "" ]
                | ExPreOpExprOp
                      [ @ ::= @ ]
                | ExPreOp
                      [ @ ::= @ " ( " @ " )%n" ]
                ;

prexplicop      : PrExplicOpVide
                      [ @ ::= "" ]
                | PrExplicOp
                      [ @ ::= @ ]
                | NotPrExplicOp
                      [ @ ::= "Not%t%t%n( " @ "%n)%b%n" ]
                | BinOpExplicOp
                      [ @ ::= @ " " @ " " @ "%n" ]
                | WithExplicOp
                      [ @ ::= @ @ ]
                | Egalite
                      [ @ ::= @ " = " @ ]
                ;

explicop        : ExplicOpVide
                      [ @ ::= "" ]
                | ExplicOpPre
                      [ @ ::= @ ]
                | ForAll
                      [ @ ::= "forall " @ " : IN ( " @ " , " @ " ) :%t%t%n"
                        @ "%b%b%n" ]
                | PreEtExplicOp
                      [ @ ::= @ ]
                ;

exprterminale   : ExprTerminaleVide
                      [ @ ::= "" ]
                | ExprTermElm
                      [ @ ::= @ ]
                | BibliOper
                      [ @ ::= @ ]
                ;

termelm         : TermElmVide
                      [ @ ::= "" ]
                | Constante
                      [ @ ::= @ ]
                | NomObj
                      [ @ ::= @ ]
                ;

exprinterm      : ExprIntermVide
                      [ @ ::= "" ]
                | ExprInterm
                      [ @ ::= "%t%nwith%t%n" @ "%b%n" @ "%b%n" ]
                | TerminInterm
                      [ @ ::= "%t%nwith%t%n" @ " = " @ "%b%n" @ "%b%n" ]
                ;

sexprinterm     : VideExprInterm
                      [ @ ::= "" ]
                | ConsExprInterm
                      [ @ ::= "%b%n" @ ["%t%nand"] @ "%n" ]
                ;
```

```
pretexplicop    : PrEtExplicOpVide
                      [ @ ::= "" ]
                | PreEt
                      [ @ ::= @ " and " @  ]
                | PreSi
                      [ @ ::= @ "%n" ]
                | PreSinon
                      [ @ ::= @ @ " Otherwise%n" ]
                ;

exsipre         : ExSiPreVide
                      [ @ ::= "" ]
                | ExSiPre
                      [ @ ::= @ " if " @ ]
                ;

sexsipre        : VideExSiPre
                      [ @ ::= "" ]
                | ConsExSiPre
                      [ @ ::=  @ [" ,%n"] @  ]
                ;

predicat        : PredVide
                      [ @ ::= "" ]
                | Atome
                      [ @ ::= @ ]
                | UnPred
                      [ @ ::= @ " " @ ]
                | BinPred
                      [ @ ::= @ " " @ " " @ ]
                | Quantif
                      [ @ ::= "( " @ " " @ " ) " @ ]
                ;

atome           : AtomeVide
                      [ @ ::= "" ]
                | AtFonc
                      [ @ ::= @ " ( " @ " )" ]
                | AtRel
                      [ @ ::= @ " " @ " " @ ]
                ;

sterme          : VideTerme
                      [ @ ::= "" ]
                | ConsTerme
                      [ @ ::= @ " " @ ]
                ;

terme           : TermeVide
                      [ @ ::= "" ]
                | TermElmCste
                      [ @ ::= @ ]
                | TermElmNomObj
                      [ @ ::= @ ]
                | TermFonc
                      [ @ ::= @ " ( " @ " )" ]
                | OpArithm
                      [ @ ::= @ " " @ " " @ ]
                ;

profilop        : ProfilOpVide
                      [ @ ::= "" ]
                | ProfilOp
```

```
                        [ @ ::= "TYPES :%t%n" @ "%b%nOPERATIONS :%t%n" @
                                "%b%n" ]
                ;

profobj         : ProfObjVide
                        [ @ ::= "" ]
                | ProfObj
                        [ @ ::= @ " : " @ ]
                ;

sprofobj        : VideProfObj
                        [ @ ::= "" ]
                | ConsProfObj
                        [ @ ::= @ "%n" @ ]
                ;

sprofop         : VideProfOp
                        [ @ ::= "" ]
                | ConsProfOp
                        [ @ ::= @ "%n" @ ]
                ;

profop          : ProfOpVide
                        [ @ ::= "" ]
                | ProfOp
                        [ @ ::= @ " : " @ ]
                ;

defop           : DefOpVide
                        [ @ ::= "" ]
                | DefOp
                        [ @ ::= "PC [ " @ " } -> " @ ]
                | BiblioDef
                        [ @ ::= @ " : IN TOOLBOX%n" ]
                ;

lexop           : LexOpVide
                        [ @ ::= "" ]
                | LexOp
                        [ @ ::= "OBJECTIVE :%t%n" @ "%b%nOBJECTS :%t%n" @
                                "%b%nOPERATIONS :%t%n" @ "%b%nPRECONDITIONS :"
                                "%t%n" @ "%b%n" ]
                ;

slgntex         : VideLgnText
                        [ @ ::= "" ]
                | ConsLgnText
                        [ @ ::= @ "%n" @ ]
                ;

sdestyp         : VideDesTyp
                        [ @ ::= "" ]
                | ConsDesTyp
                        [ @ ::= @ "%n" @ ]
                ;

destyp          : DescrTypVide
                        [ @ ::= "" ]
                | DescrTyp
                        [ @ ::= @ " : " @ ]
                ;

suitebloctypobj : VideBlocObj
                        [ @ ::= "" ]
```

```
                | ConsBlocObj
                        [ @ ::= @ "%n" @ ]
                ;

bloctypobj      : BlocTypeVide
                        [ @ ::= "" ]
                | BlocType
                        [ @ ::= "SPECIFICATION OF OBJECT TYPE : " @
                                "%nFORMAL DESCRIPTION :%t%n" @
                                "%b%nOUTLINE :%t%n" @ "%b%nLEXICON :%t%n" @
                                "%n%nEND " @ ".%n" ]
                ;

decformobj      : DecFormObjVide
                        [ @ ::= "" ]
                | DecFormObj
                        [ @ ::= "ASSOCIATED OPERATIONS :%t%n" @
                                "%b%nEXPLICATION OF TYPE STRUCTURE :%t%n" @
                                "%t%t%n->%t%t%n" @ "%b%b%b%b%n"
                                "%b%nEXPLICATION OF INVARIANT :%t%n"@"%b%n" ]
                ;

expltyp         : ExplTypVide
                        [ @ ::= "" ]
                | IdExplTyp
                        [ @ ::= @ ]
                | TypeBase
                        [ @ ::= @ ]
                | NrExplTyp
                        [ @ ::= @ " [ " @ " ]" ]
                | NrRoleExplTyp
                        [ @ ::= @ " [ " @ " ]" ]
                | UnExplTyp
                        [ @ ::= @ " [ " @ " ]" ]
                | BinExplTyp
                        [ @ ::= @ " [ " @ " -> " @ " ]" ]
                | ISAWithout
                        [ @ ::= @ " [ " @ " ] without " @ ]
                ;

sexpltyp        : VideExplTyp
                        [ @ ::= "" ]
                | ConsExplTyp
                        [ @ ::= @ [ " , " ] @ ]
                ;

svarexpltyp     : VideVarExplTyp
                        [ @ ::= "" ]
                | ConsVarExplTyp
                        [ @ ::= @ [ " , " ] @ ]
                ;

varexpltyp      : VarExplTypVide
                        [ @ ::= "" ]
                | VarExplTyp
                        [ @ ::= @ " : " @ ]
                ;

profilobj       : ProfilObjVide
                        [ @ ::= "" ]
                | ProfilObj
                        [ @ ::= "OPERATIONS :%t%n" @ "%b%nTYPES :%t%n" @
                                "%b%n" ]
                ;
```

```
lexobj          : LexObjVide
                        [ @ ::= "" ]
                | LexObj
                        [ @ ::= "OBJECTIVE :%t%n" @ "%b%nOPERATIONS :%t%n" @
                                "%b%nTYPES :%t%n" @ "%b%nINVARIANTS :"
                                "%t%n" @ "%b%n" ]
                ;

id              : IdVide
                        [ @ ::= "" ]
                | MinId
                        [ @ ::= @ ]
                | MajId
                        [ @ ::= @ ]
                ;

sid             : VideId
                        [ @ ::= "" ]
                | ConsId
                        [ @ ::= @ [ " , " ] @ ]
                ;

oprel           : OpRelVide
                        [ @ ::= "" ]
                | Equal
                        [ @ ::= @ ]
                | AutreRel
                        [ @ ::= @ ]
                ;

binop           : BinOpVide
                        [ @ ::= "" ]
                | Et
                        [ @ ::= @ ]
                | Ou
                        [ @ ::= @ ]
                ;

binpred         : BinPredVide
                        [ @ ::= "" ]
                | BinPredEt
                        [ @ ::= @ ]
                | BinPredOu
                        [ @ ::= @ ]
                | BinPredImplique
                        [ @ ::= @ ]
                | BinPredEquiv
                        [ @ ::= @ ]
                ;

quantif         : QuantifVide
                        [ @ ::= "" ]
                | QuantifPourTout
                        [ @ ::= @ ]
                | QuantifIlExiste
                        [ @ ::= @ ]
                ;

unexpltyp       : UnExplTypVide
                        [ @ ::= "" ]
                | SeType
                        [ @ ::= @ ]
                | Isa
```

```
                        [ @ ::= @ ]
                ;

lgntex          : LgnTexVide
                        [ @ ::= "" ]
                | LgnTex
                        [ @ ::= @ ]
                ;

cst             : CsteVide
                        [ @ ::= "" ]
                | CsteBool
                        [ @ ::= @ ]
                | CsteEnt
                        [ @ ::= @ ]
                ;

biblioper       : BibliOperVide
                        [ @ ::= "" ]
                | Constuple
                        [ @ ::= "CONSTUPLE ( " @ " ) " ]
                | ConstupleType
                        [ @ ::= "CONSTUPLE ( " @ " ) " ]
                | Selection
                        [ @ ::= @ " ( : " @ " ) " ]
                | As
                        [ @ ::= "AS ( " @ " , " @ " ) " ]
                | Inj
                        [ @ ::= "INJ ( " @ " , " @ " ) " ]
                | Is
                        [ @ ::= "IS ( " @ " , " @ " ) " ]
                | Filter
                        [ @ ::= "FILTER ( " @ " , " @ " ( " @ " ) ) " ]
                | Length
                        [ @ ::= "LENGTH ( " @ " ) " ]
                | Nieme
                        [ @ ::= "ITH ( " @ " , " @ " ) " ]
                | Sempty
                        [ @ ::= "SEMPTY" ]
                | ConsSuite
                        [ @ ::= "CONS ( " @ " , " @ " ) " ]
                | In
                        [ @ ::= "In ( " @ " , " @ " ) " ]
                ;

idid            : VideIdId
                        [ @ ::= "" ]
                | ConsIdId
                        [ @ ::= @ [" , "] @ ]
                ;

coupleidid      : CoupleIdIdVide
                        [ @ ::= "" ]
                | CoupleIdId
                        [ @ ::= "( " @ " , " @ " )" ]
                ;



/* FOPL */
/* ---- */
```

```
arbrefopl       : ArbreFOPLVide
                        [ @ ::= "" ]
                | ArbreFOPL
                        [ @ ::= @ ]
                ;

fform           : FFormVide
                        [ @ ::= "" ]
                | FAtome
                        [ @ ::= @ ]
                | FFormNon
                        { local STR x ;
                          x = ( fform$2 == FFormVide ()) ? "" : " = false%n " ;
                        }
                        [ @ ::= @ x ]
                | FFormEt
                        { local STR x ;
                          x = ( fform$3 == FFormVide ())
                                ? ""
                                : ( fform$2 == FFormVide ( ) )
                                        ? ""
                                        : "%n&&%n" ;
                        }
                        [ @ ::= @ x @ ]
                | FFormOu
                        { local STR x ;
                          x = ( fform$3 == FFormVide ())
                                ? ""
                                : ( fform$2 == FFormVide ( ) )
                                        ? ""
                                        : "%n||%n" ;
                        }
                        [ @ ::= @ x @ ]
                | FFormImplique
                        { local STR x ;
                          x = ( fform$3 == FFormVide ())
                                ? ""
                                : ( fform$2 == FFormVide ( ) )
                                        ? ""
                                        : "%n=>%n" ;
                        }
                        [ @ ::= @ x @ ]
                | FFormEquiv
                        { local STR x ;
                          x = ( fform$3 == FFormVide ())
                                ? ""
                                : ( fform$2 == FFormVide ( ) )
                                        ? ""
                                        : "%n<=>%n" ;
                        }
                        [ @ ::= @ x @ ]
                | FFormQuantifExist
                        { local STR x ;
                          local STR y ;
                          x = ( id == IdVide ( ) ) ? "" : "( il existe " ;
                          y = ( id == IdVide ( ) ) ? "" : " )%n " ;
                        }
                        [ @ ::= x @ y @ ]
                | FFormQuantifPrTt
                        { local STR x ;
                          local STR y ;
                          x = ( id == IdVide ( ) ) ? "" : "( pour tout " ;
                          y = ( id == IdVide ( ) ) ? "" : " )%n " ;
                        }
```

```
                    [ @ ::= x @ y @ ]
                ;

fatome          : FAtomeVide
                    [ @ ::= "" ]
                | FPred
                    [ @ ::= @ @ ]
                | FEgal
                    { local STR x ;
                      x = ( fterme$2 == FTermeVide ( ) )
                          ? ""
                          : ( fterme$1 == FTermeVide ( ) )
                                ? ""
                                : " = " ;
                    }
                    [ @ ::= @ x @ ]
                ;

fterme          : FTermeVide
                    [ @ ::= "" ]

                | FUnderScore
                    [ @ ::= "_" ]
                | FVariable
                    [ @ ::= @ ]
                | FFoncteur
                    [ @ ::= @ "(" @ ")" ]
                | FConstuple
                    [ @ ::= "constuple([" @ "])" ]
                | FSel
                    [ @ ::= "sel(" @ "," @ ")" ]
                | FAs
                    [ @ ::= "as(" @ "," @ ")" ]
                | FInj
                    [ @ ::= "inj(" @ "," @ ")" ]
                | FIs
                    [ @ ::= "is(" @ "," @ ")" ]
                | FFilter
                    [ @ ::= "filter(" @ "," @ ")" ]
                | FLength
                    [ @ ::= "length(" @ ")" ]
                | FIn
                    [ @ ::= "in(" @ "," @ ")" ]
                | FNieme
                    [ @ ::= "ith(" @ "," @ ")" ]
                | FChampObj
                    [ @ ::= @ ":" @ ]
                | FSEmpty
                    [ @ ::= "[ ]" ]
                | FCons
                    [ @ ::= "[" @ "|" @ "]" ]
                | FTypeBase
                    [ @ ::= @ ]
                ;

fsterme         : VideFTerme
                    [ @ ::= "" ]
                | ConsFTerme
                    { local STR x ;
                      x = ( fterme == FTermeVide ())
                          ? ""
                          : with ( fsterme$2 )
                                ( VideFTerme ( ) : "" ,
                                  ConsFTerme ( z , y ) :
```

```
                                                    ( z == FTermeVide ( ) )
                                                          ? ""
                                                          : ","
                                        ) ;
                        }
                        [ @ ::= @ [x] @ ]
                ;

ffoncteur       : FFoncteurVide
                        [ @ ::= "" ]
                | FId
                        [ @ ::= @ ]
                | FEnt
                        [ @ ::= @ ]
                | FBool
                        [ @ ::= @ ]
                | FNrExplTyp
                        [ @ ::= @ ]
                | FUnExplTyp
                        [ @ ::= @ ]
                | FBinExplTyp
                        [ @ ::= @ ]
                ;


/* PROLOG */
/* ------ */


arbreprolog     : ArbrePROLOGVide
                        [ @ ::= "" ]
                | ArbrePROLOG
                        [ @ ::= @ ]
                ;

pprogramme      : PProgrammeVide
                        [ @ ::= "" ]
                | PProgramme
                        [ @ ::= "%n" @ "%n" ]
                ;

suiteclauses    : VideSuiteClauses
                        [ @ ::= "" ]
                | ConsSuiteClauses
                        [ @ ::= @ ["%n%n"] @ "%n%n" ]
                ;

clause          : ClauseVide
                        [ @ ::= "" ]
                | Clause
                        { local STR x ;
                          local STR y ;
                          x = ( psantec == VideAntec ( ) ) ? "" : "%n:-%t%t%n";
                          y = ( psantec == VideAntec ( ) ) ? "." : ".%b%b";
                        }
                        [ @ ::= @ x @ y ]
                ;

pconseq         : PConseqVide
                        [ @ ::= "" ]
                | PConseq
                        [ @ ::= @ ]
```

```
                ;

patome          : PAtomeVide
                        [ @ ::= "" ]
                | PAtome
                        [ @ ::= "eval(" @ "," @ ")" ]
                ;

psantec         : VideAntec
                        [ @ ::= "" ]
                | ConsAntec
                        [ @ ::= @ [",%n"] @ ]
                ;

pantec          : PAntecVide
                        [ @ ::= "" ]
                | PAntecSEmpty
                        [ @ ::= "" ]
                | PAntecEt
                        [ @ ::= @ ",%n" @ ]
                | PAntec
                        [ @ ::= @ ]
                ;
```

ANNEXE 4

Déclaration des attributs nécessaires à la réalisation des deux transformations :

de RSL vers FOPL

et de RSL + FOPL vers Prolog

```
/*****************************************************************************/
/*                                                                           */
/* Declaration des attributs necessaires a la transformation de RSL en FOPL  */
/*                                                                           */
/*****************************************************************************/


racine          { syn fform mafo ;
                } ;

arbrersl        { syn fform mafo ;
                } ;

specification   { syn fterfter typexp    ;
                  syn fterfter typexpdec ;
                  syn fform    mafo      ;
                } ;

suiteblocoper   { syn fform    mafo      ;
                  inh fterfter typexpdec ;

                } ;

blocoper        { syn idid     vtyp      ;
                  syn fform    mafo      ;
                  inh fterfter typexpdec ;
                  } ;

decformop       { syn sid      vgfw      ;
                  syn idfterme vextyp    ;
                  syn iffi     vrimp     ;
                  syn idfterme eqimp     ;
                  syn fform    mafo      ;
                  inh fterfter typexpdec ;
                  inh idid     vtyp      ;
                } ;

expre           { syn sid      vgfwpre   ;
                  syn idfterme vextyppre ;
                  syn iffi     vrimp     ;
                  syn fform    mafo      ;
                  syn idfterme eqimp     ;
                  inh fterfter typexpdec ;
                  inh idid     vtyp      ;
                } ;

exprop          { syn sid      vgfwpre ;
                  syn sid      vinter  ;
                  syn iffi     vrimp   ;
                  syn fform    mafo    ;
                  inh sid      vgfw    ;
                  inh idfterme eqimp   ;
                  inh idfterme vextyp  ;
                } ;

expreop         { syn sid      vgfwpre ;
                  syn iffi     vrimp   ;
                  syn fform    mafo    ;
                  inh sid      vgfw    ;
                  inh idfterme eqimp   ;
                  inh idfterme vextyp  ;
                } ;
```

```
prexplicop      { syn iffi      vrimpg     ;
                  syn iffi      vrimpd     ;
                  syn iffi      vrimpinter ;
                  syn sid       varbonend  ;
                  syn idfterint vtypentg   ;
                  syn idfterint vtypentd   ;
                  syn iffi      vrimp      ;
                  syn fform     mafo       ;
                  inh sid       vgfw       ;
                  inh idfterme  vextyp     ;
                  inh idfterme  eqimp      ;
                } ;

explicop        { syn idfterint vtypent   ;
                  syn sid       varbonend ;
                  syn iffi      vrimp     ;
                  syn fform     mafo      ;
                  inh sid       vgfw      ;
                  inh idfterme  vextyp    ;
                  inh idfterme  eqimp     ;
                } ;

exprterminale   { syn fterme   mafter ;
                  inh sid      vgfw   ;
                  inh idfterme vextyp ;
                } ;

termelm         { syn fterme   mafter ;
                  inh sid      vgfw   ;
                  inh idfterme vextyp ;
                } ;

exprinterm      { syn sid      vinter ;
                  syn fform    mafo   ;
                  inh sid      vgfw   ;
                  inh idfterme vextyp ;
                  inh idfterme eqimp  ;
                } ;

sexprinterm     { syn sid      vinter ;
                  syn fform    mafo   ;
                  inh sid      vgfw   ;
                  inh idfterme vextyp ;
                  inh idfterme eqimp  ;
                } ;

pretexplicop    { syn iffi     vrimp  ;
                  syn fform    mafo   ;
                  inh sid      vgfw   ;
                  inh idfterme vextyp ;
                  inh idfterme eqimp  ;
                } ;

exsipre         { syn iffi     vrimp  ;
                  syn fform    mafo   ;
                  inh sid      vgfw   ;
                  inh idfterme vextyp ;
                  inh idfterme eqimp  ;
                } ;

sexsipre        { syn iffi     vrimp  ;
                  syn fform    mafo   ;
                  inh sid      vgfw   ;
                  inh idfterme vextyp ;
```

```
                  inh idfterme eqimp  ;
                  } ;

profilop          { syn idid vtyp ;
                  } ;

profobj           { syn idid vtyp ;
                  } ;

sprofobj          { syn idid vtyp ;
                  } ;

suitebloctypobj   { syn fterfter typexp ;
                  } ;

bloctypobj        { syn fterfter typexp ;
                  } ;

decformobj        { syn fterfter typexp ;
                  } ;

expltyp           { syn fterme fopter ;
                  } ;

sexpltyp          { syn fsterme fsopter;
                  } ;

svarexpltyp       { syn fsterme fsopter;
                  } ;

varexpltyp        { syn fterme fopter ;
                  } ;

unexpltyp         { syn unexpltyp chgt ;
                  } ;

biblioper         { syn fterme   mafter ;
                    inh sid      vgfw   ;
                    inh idfterme vextyp ;
                  } ;



/*************************************************************************/
/*                                                                       */
/* Declaration des attributs necessaires a la transformation de RSL + FOPL   */
/* en PROLOG                                                             */
/*                                                                       */
/*************************************************************************/


racine            { syn pprogramme rfapro ;
                  } ;

arbrersl          { syn pprogramme rfapro ;
                  } ;

specification     { syn pprogramme rfapro ;
                  } ;

suiteblocoper     { syn suiteclauses suitclaus ;
```

```
                } ;

blocoper        { syn suiteclauses suitclaus ;
                  } ;

decformop       { syn suiteclauses suitclaus ;
                } ;

expre           { syn suiteclauses suitclaus ;
                } ;

exprop          { syn patome           evalop  ;
                  syn ifterfterpsantec compsel ;
                } ;

expreop         { syn patome           evalop  ;
                  syn ifterfterpsantec compsel ;
                } ;

prexplicop      { syn ifterfterpsantec compsel ;
                } ;

explicop        { syn ifterfterpsantec compsel ;
                } ;

exprinterm      { syn psantec psantecwith  ;
                } ;

sexprinterm     { syn psantec psantecwith ;
                } ;

pretexplicop    { syn ifterfterpsantec compsel ;
                } ;

exsipre         { syn psantec          listevalop  ;
                  syn ifterfterpsantec compsel     ;
                } ;

sexsipre        { syn psantec          listevalop  ;
                  syn ifterfterpsantec compsel     ;
                  } ;
```

ANNEXE 5

Implémentation des attributs réalisant les deux transformations :

de RSL vers FOPL

et de RSL + FOPL vers Prolog

```
/*****************************************************************/
/*                                                               */
/* Equations des attributs pour la transformation de RSL en FOPL */
/*                                                               */
/*****************************************************************/


racine          : Racine
        { $$.mafo = arbrersl.mafo ;
        } ;

arbrersl        : ArbreRSLVide
        { $$.mafo = FFormVide ( ) ;
        }
                | ArbreRSL
        { $$.mafo = specification.mafo ;
        } ;

specification   : SpecifVide
        { $$.typexp = ConsFTerFTer ( CoupleFTerFTerVide ( ) ,
                                     VideFTerFTer ( ) ) ;
          $$.typexpdec = ConsFTerFTer ( CoupleFTerFTerVide ( ) ,
                                     VideFTerFTer ( ) ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | Specif
        { $$.typexp = suitebloctypobj.typexp ;
          $$.typexpdec = detypexpdontypexpdec ( $$.typexp , $$.typexp ) ;
          $$.mafo = suiteblocoper.mafo ;
          suiteblocoper.typexpdec = $$.typexpdec ;
        } ;

suiteblocoper   : VideBlocOper
        { $$.mafo = FFormVide ( ) ;
        }
                | ConsBlocOper
        { $$.mafo = FFormEt ( blocoper.mafo , suiteblocoper$2.mafo ) ;
          blocoper.typexpdec = $$.typexpdec ;
          suiteblocoper$2.typexpdec = $$.typexpdec ;
        } ;

blocoper        : BlocOperVide
        { $$.vtyp = VideIdId ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | BlocOper
        { $$.vtyp = profilop.vtyp ;
          $$.mafo = decformop.mafo ;
          decformop.typexpdec = $$.typexpdec ;
          decformop.vtyp = $$.vtyp ;
        } ;

decformop       : DecOperVide
        { $$.vgfw = VideId ( ) ;
          $$.vextyp = VideIdFTerme ( ) ;
          $$.vrimp = VideIFFI ( ) ;
          $$.eqimp = VideIdFTerme ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | DecOper
        { $$.vgfw = ConsId ( envar ( id ) , ensvar ( sid ) ) ;
          $$.vextyp = devarsdonexpls ( deididdonvar ( $$.vtyp ) , $$.vtyp ,
                                                          $$.typexpdec ) ;
```

```
          $$.vrimp = explicop.vrimp ;
          $$.eqimp = selecteqimp ( $$.vrimp , devarsdonexpls ( $$.vgfw ,
                                  $$.vtyp , $$.typexpdec ) ) ;
          $$.mafo = with ( expre )
                        ( ExPreVide ( ) : FFormEquiv ( exprop.mafo,
                                                       explicop.mafo ) ,
                          ExPre ( a,b ) : FFormEt ( FFormEquiv
                                ( exprop.mafo , explicop.mafo ) ,
                                                       expre.mafo ) ) ;
          exprop.vgfw = $$.vgfw ;
          exprop.eqimp = $$.eqimp ;
          exprop.vextyp = $$.vextyp ;
          explicop.vgfw = $$.vgfw ;
          explicop.vextyp = $$.vextyp ;
          explicop.eqimp = $$.eqimp ;
          expre.typexpdec = $$.typexpdec ;
          expre.vtyp = $$.vtyp ;
        } ;

expre           : ExPreVide
        { $$.vgfwpre = VideId ( ) ;
          $$.vextyppre = VideIdFTerme ( ) ;
          $$.vrimp = VideIFFI ( ) ;
          $$.eqimp = VideIdFTerme ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | ExPre
        { $$.vgfwpre = expreop.vgfwpre ;
          $$.vextyppre = devarsdonexpls ( deididdonvar ( $$.vtyp ) , $$.vtyp ,
                                                       $$.typexpdec ) ;
          $$.vrimp = prexplicop.vrimp ;
          $$.eqimp = selecteqimp ( $$.vrimp , devarsdonexpls ( $$.vgfwpre ,
                                  $$.vtyp , $$.typexpdec ) ) ;
          $$.mafo = FFormEquiv ( expreop.mafo , prexplicop.mafo ) ;
          expreop.vgfw = $$.vgfwpre ;
          expreop.eqimp = $$.eqimp ;
          expreop.vextyp = $$.vextyppre ;
          prexplicop.vgfw = $$.vgfwpre ;
          prexplicop.vextyp = $$.vextyppre ;
          prexplicop.eqimp = $$.eqimp ;
        } ;

exprop          : ExprOpVide
        { $$.vgfwpre = VideId ( ) ;
          $$.vinter = VideId ( ) ;
          $$.vrimp = VideIFFI ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | ExprOp
        { $$.vgfwpre = ConsId ( envar ( id$1 ) , ensvar ( sid ) ) ;
          $$.vinter = ConsId ( envar ( id$1 ) , VideId ( ) ) ;
          $$.vrimp = desiddonvrimp ( ConsId ( envar ( id$1 ) , ensvar ( sid )),
                          $$.vgfw , $$.vextyp ) ;
          $$.mafo = FAtome ( FEgal ( FVariable ( envar ( id$1 ) ) , FFoncteur
                          ( FId ( enmin ( id$2 ) ) , desiddonfsterme
                          ( ensvar ( sid ) ) ) ) ) ;
        } ;

expreop         : ExPreOpVide
        { $$.vgfwpre = VideId ( ) ;
          $$.vrimp = VideIFFI ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | ExPreOp
```

```
        { $$.vgfwpre = ensvar ( sid ) ;
          $$.vrimp = desiddonvrimp ( ensvar ( sid ) , $$.vgfw , $$.vextyp ) ;
          $$.mafo = FAtome ( FEgal ( FFoncteur ( FId ( enmin ( id ) ) ) ,
                        desiddonfsterme ( ensvar ( sid ) ) ) ,
                        FFoncteur ( FBool ( true ) , VideFTerme ( ) ) ) ) ;
        }
                | ExPreOpExprOp
        { $$.vgfwpre = exprop.vgfwpre ;
          $$.vrimp = exprop.vrimp ;
          $$.mafo = exprop.mafo ;
          exprop.vgfw = $$.vgfw ;
          exprop.eqimp = $$.eqimp ;
          exprop.vextyp = $$.vextyp ;
        } ;

prexplicop      : PrExplicOpVide
        { $$.vtypentg = VideIdFTerInt ( ) ;
          $$.vtypentd = VideIdFTerInt ( ) ;
          $$.varbonend = VideId ( ) ;
          $$.vrimpinter = VideIFFI ( ) ;
          $$.vrimpd = VideIFFI ( ) ;
          $$.vrimpg = VideIFFI ( ) ;
          $$.vrimp = VideIFFI ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | PrExplicOp
        { $$.vtypentg = VideIdFTerInt ( ) ;
          $$.vtypentd = VideIdFTerInt ( ) ;
          $$.varbonend = VideId ( ) ;
          $$.vrimpinter = VideIFFI ( ) ;
          $$.vrimpd = VideIFFI ( ) ;
          $$.vrimpg = VideIFFI ( ) ;
          $$.vrimp = expreop.vrimp ;
          $$.mafo = expreop.mafo ;
          expreop.vgfw = $$.vgfw ;
          expreop.eqimp = $$.eqimp ;
          expreop.vextyp = $$.vextyp ;
        }
                | NotPrExplicOp
        { $$.vtypentg = VideIdFTerInt ( ) ;
          $$.vtypentd = VideIdFTerInt ( ) ;
          $$.varbonend = VideId ( ) ;
          $$.vrimpinter = VideIFFI ( ) ;
          $$.vrimpd = VideIFFI ( ) ;
          $$.vrimpg = VideIFFI ( ) ;
          $$.vrimp = prexplicop$2.vrimp ;
          $$.mafo = FFormNon ( prexplicop$2.mafo ) ;
          prexplicop$2.vgfw = $$.vgfw ;
          prexplicop$2.vextyp = $$.vextyp ;
          prexplicop$2.eqimp = $$.eqimp ;
        }
                | BinOpExplicOp
        { $$.vtypentg = with ( binop )
                      ( BinOpVide ( ) : VideIdFTerInt ( ) ,
                        Et ( a ) : varPCint ( $$.vrimpg , $$.vextyp ) ,
                        Ou ( a ) : varUNIONint ( $$.vrimpg , $$.vextyp )
                      ) ;
          $$.vtypentd = with ( binop )
                      ( BinOpVide ( ) : VideIdFTerInt ( ) ,
                        Et ( a ) : varPCint ( $$.vrimpd , $$.vextyp ) ,
                        Ou ( a ) : varUNIONint ( $$.vrimpd , $$.vextyp )
                      ) ;
          $$.varbonend = donlistvbonend ( $$.vtypentg , $$.vtypentd )
                        @ donlistvbonend ( $$.vtypentd , $$.vtypentg ) ;
```

```
  $$.vrimpinter = interlistiffi ( explicop$1.vrimp , explicop$2.vrimp);
  $$.vrimpg = difflistesiffi ( explicop$1.vrimp , $$.vrimpinter ) ;
  $$.vrimpd = difflistesiffi ( explicop$2.vrimp , $$.vrimpinter ) ;
  $$.vrimp = with ( binop )
             ( BinOpVide ( ) : VideIFFI ( ) ,
               Et ( e ) :
                 $$.vrimpinter
                 @ majdevrimpet ( $$.vrimpg , $$.vtypentg ,
                                                  $$.varbonend )
                 @ majdevrimpet ( $$.vrimpd , $$.vtypentd ,
                                                $$.varbonend ) ,
               Ou ( o ) :
                 $$.vrimpinter
                 @ majdevrimpou ( $$.vrimpg , $$.vtypentg ,
                                                  $$.varbonend )
                 @ majdevrimpou ( $$.vrimpd , $$.vtypentd ,
                                                $$.varbonend ) ) ;
  $$.mafo = with ( binop )
            ( BinOpVide ( ) : FFormVide ( ) ,
              Et ( e ) :
                 eqimplenfform ( eqattach ( $$.eqimp ,
                 $$.varbonend ) , FFormEt ( explicop$1.mafo ,
                 explicop$2.mafo ) ) ,
              Ou ( o ) :
                 eqimplenfform ( eqattach ( $$.eqimp ,
                 $$.varbonend ) , FFormOu ( explicop$1.mafo ,
                 explicop$2.mafo ) )
            ) ;
  explicop$1.vgfw = $$.vgfw ;
  explicop$1.vextyp = $$.vextyp ;
  explicop$1.eqimp = $$.eqimp ;
  explicop$2.vgfw = $$.vgfw ;
  explicop$2.vextyp = $$.vextyp ;
  explicop$2.eqimp = $$.eqimp ;
  }

        | WithExplicOp
{ $$.vtypentg = VideIdFTerInt ( ) ;
  $$.vtypentd = VideIdFTerInt ( ) ;
  $$.varbonend = VideId ( ) ;
  $$.vrimpinter = VideIFFI ( ) ;
  $$.vrimpd = VideIFFI ( ) ;
  $$.vrimpg = VideIFFI ( ) ;
  $$.vrimp = prexplicop$2.vrimp ;
  $$.mafo = FFormEt ( prexplicop$2.mafo , sidenfform
          ( sexprinterm.vinter , sexprinterm.mafo ) ) ;
  prexplicop$2.vgfw = $$.vgfw @ sexprinterm.vinter ;
  prexplicop$2.vextyp = $$.vextyp ;
  prexplicop$2.eqimp = $$.eqimp ;
  sexprinterm.vgfw = $$.vgfw ;
  sexprinterm.vextyp = $$.vextyp ;
  sexprinterm.eqimp = $$.eqimp ;
}
        | Egalite
{ $$.vtypentg = VideIdFTerInt ( ) ;
  $$.vtypentd = VideIdFTerInt ( ) ;
  $$.varbonend = VideId ( ) ;
  $$.vrimpinter = VideIFFI ( ) ;
  $$.vrimpd = VideIFFI ( ) ;
  $$.vrimpg = VideIFFI ( ) ;
  $$.vrimp = VideIFFI ( ) ;
  $$.mafo = FAtome ( FEgal ( exprterminale$1.mafter ,
                exprterminale$2.mafter ) ) ;
  exprterminale$1.vgfw = $$.vgfw ;
```

```
          exprterminale$1.vextyp = $$.vextyp ;
          exprterminale$2.vgfw = $$.vgfw ;
          exprterminale$2.vextyp = $$.vextyp ;
        } ;

explicop        : ExplicOpVide
        { $$.vtypent = VideIdFTerInt ( ) ;
          $$.varbonend = VideId ( );
          $$.vrimp = VideIFFI ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
              | ExplicOpPre
        { $$.vtypent = VideIdFTerInt ( ) ;
          $$.varbonend = VideId ( );
          $$.vrimp = prexplicop.vrimp ;
          $$.mafo = prexplicop.mafo ;
          prexplicop.vgfw = $$.vgfw ;
          prexplicop.vextyp = $$.vextyp ;
          prexplicop.eqimp = $$.eqimp ;
        }
              | ForAll
        { $$.vtypent = varSEQint ( explicop$2.vrimp , $$.vextyp ) ;
          $$.varbonend = donvbonendforall ( $$.vtypent ) ;
          $$.vrimp = majdevrimpseq ( explicop$2.vrimp , $$.vtypent ,
                                                    envar ( id$2 ) ) ;
          $$.mafo = FFormEt ( eqlg ( donvardevtypent ( $$.vtypent ) ,
                          $$.eqimp , envar ( id$3 ) , envar ( id$2 ) ) ,
                          FFormQuantifPrTt ( envar ( id$2 ) ,
                        FFormImplique ( FAtome ( FEgal ( FIn ( envar ( id$2 ) ,
                                                    envar ( id$3 ) ) ) ,
                                FFoncteur ( FBool ( true ) , VideFTerme ( ) )
                                                    ) ) ,
                        FFormQuantifExist ( nvlid ( envar ( id$2 ) ) ) ,
                        FFormEt ( FAtome ( FEgal ( FVariable ( envar ( id$2 ))
                                , FNieme ( nvlid ( envar ( id$2 ) ) ) ,
                                  FVariable ( envar ( id$3 ) ) )
                                                    ) ) ,
                        eqimplenfform ( eqattach ( $$.eqimp , $$.varbonend ) ,
                                                  explicop$2.mafo ) ) ) ) ) ) ;
          explicop$2.vgfw = envar ( id$1 ) :: $$.vgfw ;
          explicop$2.vextyp = $$.vextyp ;
          explicop$2.eqimp = $$.eqimp ;
        }
              | PreEtExplicOp
        { $$.vtypent = VideIdFTerInt ( ) ;
          $$.varbonend = VideId ( );
          $$.vrimp = pretexplicop.vrimp ;
          $$.mafo = pretexplicop.mafo ;
          pretexplicop.vgfw = $$.vgfw ;
          pretexplicop.vextyp = $$.vextyp ;
          pretexplicop.eqimp = $$.eqimp ;
        } ;

exprterminale   : ExprTerminaleVide
        { $$.mafter = FTermeVide ( ) ;
        }
              | ExprTermElm
        { $$.mafter = termelm.mafter ;
          termelm.vgfw = $$.vgfw ;
          termelm.vextyp = $$.vextyp ;
        }
              | BibliOper
        { $$.mafter = biblioper.mafter ;
          biblioper.vgfw = $$.vgfw ;
```

```
          biblioper.vextyp = $$.vextyp ;
        } ;

termelm         : TermElmVide
        { $$.mafter = FTermeVide ( ) ;
        }
                | Constante
        { $$.mafter =
          with ( cst )
               ( CsteVide ( ) : FTermeVide ( ) ,
                 CsteBool ( a ) : FFoncteur ( FBool ( a ) , VideFTerme ( ) ) ,
                 CsteEnt ( a ) : FFoncteur ( FEnt ( a ) , VideFTerme ( ) )
               ) ;
        }
                | NomObj
        { $$.mafter = FVariable ( envar ( id ) ) ;
        } ;

exprinterm      : ExprIntermVide
        { $$.vinter = VideId ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | ExprInterm
        { $$.vinter = diffdeuxlistessid ( exprop.vinter , sexprinterm.vinter )
                      @ sexprinterm.vinter ;
          $$.mafo =
          with ( sexprinterm )
               ( VideExprInterm ( ) : exprop.mafo ,
                 ConsExprInterm ( a , b ) :
                       FFormEt ( exprop.mafo , sexprinterm.mafo ) ) ;
          exprop.vgfw = $$.vgfw @ sexprinterm.vinter ;
          exprop.vextyp = $$.vextyp ;
          exprop.eqimp = $$.eqimp ;
          sexprinterm.vgfw = $$.vgfw @ exprop.vinter ;
          sexprinterm.vextyp = $$.vextyp ;
          sexprinterm.eqimp = $$.eqimp ;
        }
                | TerminInterm
        { $$.vinter = diffdeuxlistessid ( ConsId ( envar ( id ) , VideId ( ) ),
                      sexprinterm.vinter ) @ sexprinterm.vinter ;
          $$.mafo = with ( sexprinterm )
                       ( VideExprInterm ( ) : FAtome ( FEgal ( FVariable
                              ( envar ( id ) ) , exprterminale.mafter ) ) ,
                         ConsExprInterm ( a,b ) : FFormEt ( FAtome ( FEgal
                              ( FVariable ( envar ( id ) ) ,
                                     exprterminale.mafter ) ),
                              sexprinterm.mafo ) ) ;
          exprterminale.vgfw = $$.vgfw @ sexprinterm.vinter ;
          exprterminale.vextyp = $$.vextyp ;
          sexprinterm.vgfw = $$.vgfw ;
          sexprinterm.vextyp = $$.vextyp ;
          sexprinterm.eqimp = $$.eqimp ;
        } ;

sexprinterm     : VideExprInterm
        { $$.vinter = VideId ( ) ;
          $$.mafo = FFormVide ( ) ;
        }
                | ConsExprInterm
        { $$.vinter = diffdeuxlistessid ( exprinterm.vinter ,
                      sexprinterm$2.vinter ) @ sexprinterm$2.vinter ;
          $$.mafo = FFormEt ( exprinterm.mafo , sexprinterm$2.mafo ) ;
          exprinterm.vgfw = $$.vgfw @ sexprinterm$2.vinter ;
          exprinterm.vextyp = $$.vextyp ;
```

```
            exprinterm.eqimp = $$.eqimp ;
            sexprinterm$2.vgfw = $$.vgfw @ exprinterm.vinter ;
            sexprinterm$2.vextyp = $$.vextyp ;
            sexprinterm$2.eqimp = $$.eqimp ;
          } ;

pretexplicop    : PrEtExplicOpVide
          { $$.vrimp = VideIFFI ( ) ;
            $$.mafo = FFormVide ( ) ;
          }
                | PreEt
          { $$.vrimp = difflistesiffi ( prexplicop.vrimp , explicop.vrimp )
                       @ explicop.vrimp ;
            $$.mafo = FFormEt ( prexplicop.mafo , explicop.mafo ) ;
            prexplicop.vgfw = $$.vgfw ;
            prexplicop.vextyp = $$.vextyp ;
            prexplicop.eqimp = $$.eqimp ;
            explicop.vgfw = $$.vgfw ;
            explicop.vextyp = $$.vextyp ;
            explicop.eqimp = $$.eqimp ;
          }
                | PreSi
          { $$.vrimp = sexsipre.vrimp ;
            $$.mafo =sexsipre.mafo ;
            sexsipre.vgfw = $$.vgfw ;
            sexsipre.vextyp = $$.vextyp ;
            sexsipre.eqimp = $$.eqimp ;
          }
                | PreSinon
          { $$.vrimp = difflistesiffi ( sexsipre.vrimp , explicop.vrimp )
                       @ explicop.vrimp ;
            $$.mafo = FFormOu ( sexsipre.mafo , explicop.mafo ) ;
            sexsipre.vgfw = $$.vgfw ;
            sexsipre.vextyp = $$.vextyp ;
            sexsipre.eqimp = $$.eqimp ;
            explicop.vgfw = $$.vgfw ;
            explicop.vextyp = $$.vextyp ;
            explicop.eqimp = $$.eqimp ;
          } ;

exsipre         : ExSiPreVide
          { $$.vrimp = VideIFFI ( ) ;
            $$.mafo = FFormVide ( ) ;
          }
                | ExSiPre
          { $$.vrimp = difflistesiffi ( expreop.vrimp , explicop.vrimp )
                       @ explicop.vrimp ;
            $$.mafo = FFormEt ( expreop.mafo , explicop.mafo ) ;
            explicop.vgfw = $$.vgfw ;
            explicop.vextyp = $$.vextyp ;
            explicop.eqimp = $$.eqimp ;
            expreop.vgfw = $$.vgfw ;
            expreop.eqimp = $$.eqimp ;
            expreop.vextyp = $$.vextyp ;
          } ;

sexsipre        : VideExSiPre
          { $$.vrimp = VideIFFI ( ) ;
            $$.mafo = FFormVide ( ) ;
          }
                | ConsExSiPre
          { $$.vrimp = difflistesiffi ( exsipre.vrimp, sexsipre$2.vrimp )
                       @ sexsipre$2.vrimp ;
            $$.mafo = FFormOu ( exsipre.mafo , sexsipre$2.mafo ) ;
```

```
            exsipre.vgfw = $$.vgfw ;
            exsipre.vextyp = $$.vextyp ;
            exsipre.eqimp = $$.eqimp ;
            sexsipre$2.vgfw = $$.vgfw ;
            sexsipre$2.vextyp = $$.vextyp ;
            sexsipre$2.eqimp = $$.eqimp ;
          } ;

profilop        : ProfilOpVide
          { $$.vtyp = VideIdId ( ) ;
          }
                | ProfilOp
          { $$.vtyp = sprofobj.vtyp ;
          } ;

profobj         : ProfObjVide
          { $$.vtyp = VideIdId ( ) ;
          }
                | ProfObj
          { $$.vtyp = listeconsvtyp ( ensvar ( sid ) , enmin ( id ) ) ;
          } ;

sprofobj        : VideProfObj
          { $$.vtyp = VideIdId ( ) ;
          }
                | ConsProfObj
          { $$.vtyp = profobj.vtyp @ sprofobj$2.vtyp ;
          } ;

suitebloctypobj : VideBlocObj
          { $$.typexp = VideFTerFTer ( ) ;
          }
                | ConsBlocObj
          { $$.typexp = bloctypobj.typexp @ suitebloctypobj$2.typexp ;
          } ;

bloctypobj      : BlocTypeVide
          { $$.typexp = VideFTerFTer ( ) ;
          }
                | BlocType
          { $$.typexp = decformobj.typexp ;
          } ;

decformobj      : DecFormObjVide
          { $$.typexp = VideFTerFTer ( ) ;
          }
                | DecFormObj
          { $$.typexp = ConsFTerFTer ( CoupleFTerFTer ( FVariable
                        ( enmin ( id ) ) , expltyp.fopter ) , VideFTerFTer ( ) ) ;
          } ;

expltyp         : ExplTypVide
          { $$.fopter = FTermeVide ( ) ;
          }
                | IdExplTyp
          { $$.fopter = FVariable ( enmin ( id ) ) ;
          }
                | TypeBase
          { $$.fopter = FTypeBase ( STRtolower ( STR ) ) ;
          }
                | NrExplTyp
          { $$.fopter = FFoncteur ( FNrExplTyp ( STRtolower ( STR ) ) ,
                                                  sexpltyp.fsopter ) ;
          }
```

```
                | NrRoleExplTyp
        { $$.fopter = FFoncteur ( FNrExplTyp ( STRtolower ( STR ) ) ,
                                                svarexpltyp.fsopter ) ;
        }
                | UnExplTyp
        { $$.fopter = FFoncteur ( FUnExplTyp ( unexpltyp.chgt ) , ConsFTerme
                                ( expltyp$2.fopter , VideFTerme ( ) ) ) ;
        }
                | BinExplTyp
        { $$.fopter = FFoncteur ( FBinExplTyp ( STRtolower ( STR ) ) ,
                        ConsFTerme ( expltyp$2.fopter ,
                        ConsFTerme ( expltyp$3.fopter , VideFTerme ( ) ) ) ) ;
        }
                | ISAWithout
        { $$.fopter = FFoncteur ( FUnExplTyp ( unexpltyp.chgt ) , ConsFTerme
                                ( expltyp$2.fopter , VideFTerme ( ) ) ) ;
        } ;

sexpltyp        : VideExplTyp
        { $$.fsopter = ConsFTerme ( FTermeVide ( ) , VideFTerme ( ) ) ;
        }
                | ConsExplTyp
        { $$.fsopter = ConsFTerme ( expltyp.fopter , sexpltyp$2.fsopter ) ;
        } ;

svarexpltyp     : VideVarExplTyp
        { $$.fsopter = ConsFTerme ( FTermeVide ( ) , VideFTerme ( ) ) ;
        }
                | ConsVarExplTyp
        { $$.fsopter = ConsFTerme ( varexpltyp.fopter ,
                                        svarexpltyp$2.fsopter ) ;
        } ;

varexpltyp      : VarExplTypVide
        { $$.fopter = FTermeVide ( ) ;
        }
                | VarExplTyp
        { $$.fopter = expltyp.fopter ;
        } ;

biblioper       : BibliOperVide
        { $$.mafter = FTermeVide ( ) ;
        }
                | Constuple
        { $$.mafter = FConstuple ( desiddonfsterme ( ensvar ( sid ) ) ) ;
        }
                | ConstupleType
        { $$.mafter = FConstuple ( desiddonfsterme
                                        ( ensvar ( deididdonsid ( idid ) ) ) );
        }
                | Selection
        { $$.mafter = FSel ( FVariable ( enmin ( id$1 ) ) ,
                                                FVariable ( envar ( id$2 ) )
                        ) ;
        }
                | As
        { $$.mafter = FAs ( FVariable ( enmin ( id$1 ) ) ,
                                                 FVariable ( envar ( id$2 ) )
                        ) ;
        }
                | Inj
        { $$.mafter = FInj ( FVariable ( enmin ( id$1 ) ) ,
                                                FVariable ( envar ( id$2 ) )
                        ) ;
```

```
        }
                | Is
        { $$.mafter = FIs ( FVariable ( enmin ( id$1 ) ) ,
                            FVariable ( enmin ( id$2 ) )
                          ) ;
        }
                | Filter
        { $$.mafter = FFilter ( FVariable ( envar ( id$1 ) ) ,
                        FAtome ( FPred ( enmin ( id$2 ) ,
                        desiddonfsterme ( ensvar ( sid ) ) ) ) ) ;
        }
                | Length
        { $$.mafter = FLength ( FVariable ( envar ( id ) ) ) ;
        }
                | Nieme
        { $$.mafter = FNieme ( envar ( id$1 ) , FVariable ( envar ( id$2 ) )
                             ) ;
        }
                | Sempty
        { $$.mafter = FTermeVide ( ) ;
        }
                | ConsSuite
        { $$.mafter = FIs ( FVariable ( envar ( id$1 ) ) ,
                            FVariable ( envar ( id$2 ) )
                          ) ;
        }
                | In
        { $$.mafter = FIn ( envar ( id$1 ) , envar ( id$2 ) ) ;
        }
                ;

unexpltyp       : UnExplTypVide
        { $$.chgt = UnExplTypVide ( ) ;
        }
                | SeType
        { $$.chgt = SeType ( STRtolower ( STR ) ) ;
        }
                | Isa
        { $$.chgt = Isa ( STRtolower ( STR ) ) ;
        }
                ;
```

```
/*****************************************************************************/
/*                                                                           */
/* Equations des attributs pour la transformation de RSL + FOPL en PROLOG    */
/*                                                                           */
/*****************************************************************************/
```

```
racine          : Racine
        { $$.rfapro = arbrersl.rfapro ;
        } ;

arbrersl        : ArbreRSLVide
        { $$.rfapro = PProgrammeVide ( ) ;
        }
                | ArbreRSL
        { $$.rfapro = specification.rfapro ;
        } ;
```

```
specification    : SpecifVide
        { $$.rfapro = PProgrammeVide ( ) ;
        }
                | Specif
        { $$.rfapro = PProgramme ( suiteblocoper.suitclaus ) ;
        } ;

suiteblocoper    : VideBlocOper
        { $$.suitclaus = VideSuiteClauses ( ) ;
        }
                | ConsBlocOper
        { $$.suitclaus = blocoper.suitclaus @ suiteblocoper$2.suitclaus ;
        } ;

blocoper         : BlocOperVide
        { $$.suitclaus = VideSuiteClauses ( ) ;
        }
                | BlocOper
        { $$.suitclaus = decformop.suitclaus ;
        } ;

decformop        : DecOperVide
        { $$.suitclaus = VideSuiteClauses ( ) ;
        }
                | DecOper
        { $$.suitclaus = expre.suitclaus @ genereprogprolog ( explicop.compsel
                                                    , exprop.evalop ) ;
        } ;

expre            : ExPreVide
        { $$.suitclaus = VideSuiteClauses ( ) ;
        }
                | ExPre
        { $$.suitclaus = genereprogprolog ( prexplicop.compsel ,
                                                    expreop.evalop ) ;
        } ;

exprop           : ExprOpVide
        { $$.evalop = PAtomeVide ( ) ;
          $$.compsel = VideIFTerFTerPSAntec ( ) ;
        }
                | ExprOp
        { $$.evalop = PAtome ( FVariable ( envar ( id$1 ) ) ,
                               FFoncteur ( FId ( enmin ( id$2 ) ) ,
                                     desiddonfsterme ( ensvar ( sid ) ) ) ) ;
          $$.compsel = deidsiddonifterfterpsantec ( envar ( id$1 ) ,
                         enmin ( id$2 ) , ensvar ( sid ) , exprop.eqimp ) ;
        } ;

expreop          : ExPreOpVide
        { $$.evalop = PAtomeVide ( ) ;
          $$.compsel = VideIFTerFTerPSAntec ( ) ;
        }
                | ExPreOp
        { $$.evalop = PAtome ( FFoncteur ( FBool ( true ) , VideFTerme ( ) ) ,
                               FFoncteur ( FId ( enmin ( id ) ) ,
                                     desiddonfsterme ( ensvar ( sid ) ) ) ) ;
          $$.compsel = deidsiddonifterfterpsantec ( IdVide ( ) ,
                         enmin ( id ) , ensvar ( sid ) , expreop.eqimp ) ;
        }
                | ExPreOpExprOp
        { $$.evalop = exprop.evalop ;
          $$.compsel = exprop.compsel ;
```

```
        } ;

prexplicop      : PrExplicOpVide
        { $$.compsel = VideIFTerFTerPSAntec ( ) ;
        }
                | PrExplicOp
        { $$.compsel = expreop.compsel ;
        }
                | NotPrExplicOp
        { $$.compsel = VideIFTerFTerPSAntec ( ) ;
        }
                | BinOpExplicOp
        { $$.compsel = with ( binop )
                       ( BinOpVide ( ) : VideIFTerFTerPSAntec ( ) ,
                         Et ( e ) : majtripletsPC ( elimindoublons2 ( concatPC
                                  ( explicop$1.compsel , explicop$2.compsel ) ) ,
                                                    $$.vextyp ) ,
                         Ou ( o ) : majtripletsUNION ( explicop$1.compsel @
                                                       explicop$2.compsel , $$.vextyp )
                       ) ;
          }

                | WithExplicOp
        { $$.compsel = ajoutpsantecdvtpsantec ( prexplicop$2.compsel ,
                                                sexprinterm.psantecwith ) ;
        }
                | Egalite
        { $$.compsel = ConsIFTerFTerPSAntec ( TriplIFTerFTerPSAntec (
                         VideFTerFTer ( ) , ConsAntec ( PAntec ( PAtome (
                         exprterminale$1.mafter , exprterminale$2.mafter ) ) ,
                         VideAntec ( ) ) , VideListeFTerFTer ( ) ) ,
                         VideIFTerFTerPSAntec ( ) ) ;
        } ;

explicop        : ExplicOpVide
        { $$.compsel = VideIFTerFTerPSAntec ( ) ;
        }
                | ExplicOpPre
        { $$.compsel = prexplicop.compsel ;
        }
                | ForAll
        { $$.compsel = elimindoublons1 ( majtripletsSEQ ( ajoutelemforall (
                                         explicop$2.compsel , envar ( id$2 ) ,
                                                              envar ( id$3 ) ) ) ) ;
        }
                | PreEtExplicOp
        { $$.compsel = pretexplicop.compsel ;
        } ;

exprinterm      : ExprIntermVide
        { $$.psantecwith = VideAntec ( ) ;
        }
                | ExprInterm
        { $$.psantecwith = ConsAntec ( PAntec ( exprop.evalop ) ,
                                                sexprinterm.psantecwith ) ;
        }
                | TerminInterm
        { $$.psantecwith = ConsAntec ( PAntec ( PAtome ( FVariable
                                 ( envar ( id ) ) , exprterminale.mafter ) ) ,
                                                   sexprinterm.psantecwith ) ;
        } ;

sexprinterm     : VideExprInterm
        { $$.psantecwith = VideAntec ( ) ;
```

```
        }
                | ConsExprInterm
        { $$.psantecwith = exprinterm.psantecwith @ sexprinterm$2.psantecwith ;
        } ;

pretexplicop    : PrEtExplicOpVide
        { $$.compsel = VideIFTerFTerPSAntec ( ) ;
        }
                | PreEt
        { $$.compsel = majtripletsPC ( elimindoublons2 ( concatPC (
                        prexplicop.compsel , explicop.compsel ) ) , $$.vextyp ) ;
        }
                | PreSi
        { $$.compsel = sexsipre.compsel ;
        }
                | PreSinon
        { $$.compsel = sexsipre.compsel @ ajoutpsantecdvtpsantec (
                        explicop.compsel , negpsantec ( sexsipre.listevalop ) ) ;
        } ;

exsipre         : ExSiPreVide
        { $$.compsel = VideIFTerFTerPSAntec ( ) ;
          $$.listevalop = VideAntec ( ) ;
        }
                | ExSiPre
        { $$.compsel = ajoutpsantecdvtpsantec ( explicop.compsel ,
                    ConsAntec ( PAntec ( expreop.evalop ) , VideAntec ( ) ) ) ;
          $$.listevalop = ConsAntec ( PAntec ( expreop.evalop ), VideAntec ());
        } ;

sexsipre        : VideExSiPre
        { $$.compsel = VideIFTerFTerPSAntec ( ) ;
          $$.listevalop = VideAntec ( ) ;
        }
                | ConsExSiPre
        { $$.compsel = exsipre.compsel @ sexsipre$2.compsel ;
          $$.listevalop = exsipre.listevalop @ sexsipre$2.listevalop ;
        } ;
```

ANNEXE 6

Les diverses fonctions auxiliaires introduites lors de l'implémentation des attributs réalisant les deux transformations :

de RSL vers FOPL

et de RSL + FOPL vers Prolog

```
/*************************************************/
/*                                               */
/*      Definitions des fonctions utilisees      */
/*                                               */
/*************************************************/


/* Premiere transformation : RSL -> FOPL                                  */
/* ======================================                                 */



/* Vrai si l'identificateur b figure dans la suite d'identificateurs a,
   faux sinon                                                             */

BOOL dssidid ( sid a , id b )
   { with ( a )
        ( VideId ( ) : false ,
          ConsId ( x,y ) :
           ( x == b )    ? true
                         : dssidid ( y,b )
        )
   };



/* Vrai si le terme b figure dans la suite de termes a,
   faux sinon                                                             */

BOOL dssfterme ( fsterme a , fterme b )
   { with ( a )
        ( VideFTerme ( ) : false ,
          ConsFTerme ( x,y ) :
           ( x == b )    ? true
                         : dssfterme ( y,b )
        )
   };



/* Vrai si l'element b de type ( id , fterme , fterme , INT ) figure
   dans la suite de quadriffi a , faux sinon                              */

BOOL dsiffi ( iffi a , quadriffi b )
   { with ( a )
        ( VideIFFI ( ) : false ,
          ConsIFFI ( x,y ) :
           ( x == b )    ? true
                         : dsiffi ( y,b )
        )
   };



/* A partir d'une liste sid et d'un identificateur b donnes,
   construit une liste de ( id , b )                                      */

idid listeconsvtyp ( sid a , id b )
   { with ( a )
        ( VideId ( ) : VideIdId ( ) ,
          ConsId ( x,y ) : ConsIdId ( CoupleIdId ( x,b ) ,
                                      listeconsvtyp ( y,b ) )
        )
   };
```

```
/* A partir d'une liste de ( x,y ) ou x et y sont des identificateurs
   renvoie la liste des identificateurs y de la liste initiale          */

sid deididdonsid ( idid a )
   { with ( a )
        ( VideIdId ( ) : VideId ,
          ConsIdId ( w,z ) :
           with ( w )
                ( CoupleIdIdVide ( ) : deididdonsid ( z ) ,
                  CoupleIdId ( x,y ) :
                                ConsId ( y , deididdonsid ( z ) )
                )
        )
   };



/* A partir d'une liste de ( x,y ) ou x et y sont des identificateurs
   renvoie la liste des identificateurs x de la liste initiale          */

sid deididdonvar ( idid a )
   { with ( a )
        ( VideIdId ( ) : VideId ,
          ConsIdId ( w,z ) :
           with ( w )
                ( CoupleIdIdVide ( ) : deididdonvar ( z ) ,
                  CoupleIdId ( x,y ) :
                                ConsId ( x , deididdonvar ( z ) )
                )
        )
   };



/* A partir d'une liste a de ( id , fterme ) et d'un fterme b donnes,
   produit le id de l'element de a dont le fterme egale b si un tel
   element existe , produit le fterme vide sinon                        */

id deidftermedonid ( idfterme a , fterme b )
   { with ( a )
        ( VideIdFTerme ( ) : IdVide ( ) ,
          ConsIdFTerme ( w,z ) :
           with ( w )
                ( CoupleIdFTermeVide ( ) : deidftermedonid ( z,b ) ,
                  CoupleIdFTerme ( x,y ) :
           ( y == b )    ? x
                         : deidftermedonid ( z,b )
                )
        )
   };



/* A partir d'un identificateur donne a, construit un identificateur a# */

id nvlid ( id a )
   { with ( a )
        ( IdVide ( ) : IdVide ( ) ,
          MinId ( a ) : MinId ( a # "#" ) ,
          MajId ( a ) : MajId ( a # "#" )
        )
```

```
};


/* recoit une liste a de ( ida , ftermeb , INTc ) et ub id b,
   renvoie la liste des elements de a tels que ida = b,
           la liste vide s'il n'existe pas de tel element                */

triplidfterint dsvtypent ( idfterint a , id b )
   { with ( a )
        ( VideIdFTerInt ( ) : TriplIdFTerIntVide ( ) ,
          ConsIdFTerInt ( x,y ) :
           with ( x )
                ( TriplIdFTerIntVide ( ) : dsvtypent ( y,b ) ,
                  TriplIdFTerInt ( r , s , t ) :
                   ( r == b )    ? x
                                 : dsvtypent ( y,b )
                )
        )
   };


/* A partir d'une liste de ( id , type de id ) et d'un id donne,
   retrouve dans la liste le type du id donne                             */

id devardontyp ( idid a , id b )
   { with ( a )
        ( VideIdId ( ) : IdVide ( ) ,
          ConsIdId ( w,z ) :
           with ( w )
                ( CoupleIdIdVide ( ) : devardontyp ( z,b ) ,
                  CoupleIdId ( x,y ) :
           ( x == b )    ? y
                         : devardontyp ( z,b )
                )
        )
   };


/* A partir d'une liste a de ( id , fterme ) et d'un id b donnes,
   produit le fterme de l'element de a dont le id egale b si un tel
   element existe , produit le fterme vide sinon                          */

fterme deidftermedonfter ( idfterme a , id b )
   { with ( a )
        ( VideIdFTerme ( ) : FTermeVide ( ) ,
          ConsIdFTerme ( w,z ) :
           with ( w )
                ( CoupleIdFTermeVide ( ) : deidftermedonfter ( z,b ) ,
                  CoupleIdFTerme ( x,y ) :
           ( x == b )    ? y
                         : deidftermedonfter ( z,b )
                )
        )
   };


/* A partir d'une liste de ( type , expltyp ) et d'un type donne,
   retrouver dans la liste l'expltyp du type donne                        */

fterme detypdonexpl ( fterfter a , id b )
```

```
   { with ( a )
        ( VideFTerFTer ( ) : FTermeVide ( ) ,
          ConsFTerFTer ( w,z ) :
           with ( w )
                ( CoupleFTerFTerVide ( ) : detypdonexpl ( z,b ) ,
                  CoupleFTerFTer ( x,y ) :
             ( x == FVariable ( b ) )
                        ? y
                        : detypdonexpl ( z,b )
                )
        )
   };


/* disposant d'un fterme a et d'un fterme b,
   si le fterme a est le fterme vide, genere b,
   si le fterme a est de la forme CSEL ( x , vide ) genere le fterme
   CSEL ( x , b ),
   pour les autres ftermes, genere le fterme vide.                    */

fterme consfterme ( fterme a , fterme b )
   { with ( a )
        ( FTermeVide ( ) : b ,
          FSel ( x , y ) : FSel ( x , consfterme ( y , b ) ) ,
          FAs ( x , y ) : FAs ( x , consfterme ( y , b ) ) ,
          FNieme ( x , y ) : FNieme ( x , consfterme ( y , b ) ) ,
          default : FTermeVide ( )
        )
   } ;


/* Disposant de deux listes, l'une de ( id , type de id ), l'autre de
   ( type , expltyp ) , retrouver pour tout id appartenant a une liste de
   id donnee l'expltyp de ce id                                       */

idfterme devarsdonexpls ( sid a , idid b , fterfter c )
   { with ( a )
        ( VideId ( ) : VideIdFTerme ( ) ,
          ConsId ( x,y ) : ConsIdFTerme ( CoupleIdFTerme ( x ,
                detypdonexpl ( c , devardontyp ( b,x ) ) ) ,
                devarsdonexpls ( y , b , c ) )
        )
   };


/* A partir d'une suite sid renvoie la suite de ftermes correspondant aux
   elements de la suite initiale                                      */

fsterme desiddonfsterme ( sid a )
   { with ( a )
        ( VideId ( ) : VideFTerme ( ) ,
          ConsId ( x,y ) : ConsFTerme ( FVariable ( x ) ,
                           desiddonfsterme ( y ) )
        )
   };


/* A partir d'une liste de ( id , fsterme )  produit la liste des fstermes
   de la liste initiale                                               */
```

```
fsterme deidftermedonfster ( idfterme a )
   { with ( a )
        ( VideIdFTerme ( ) : VideFTerme ( ) ,
          ConsIdFTerme ( w,z ) :
           with ( w )
                ( CoupleIdFTermeVide ( ) : deidftermedonfster ( z ) ,
                  CoupleIdFTerme ( x,y ) :
                ConsFTerme ( y , deidftermedonfster ( z ) )
                )
        )
   };



/* Etant donnes une liste a de ftermes,
                un identificateur b et
                un entier c,
   construit pour chaque terme t de a un triplet ( b , t , c )          */

idfterint defstermedonidfterint ( fsterme a , id b , INT c )
   { with ( a )
        ( VideFTerme ( ) : VideIdFTerInt ( ) ,
          ConsFTerme ( t,v ) : ConsIdFTerInt ( TriplIdFTerInt ( b , t , c ) ,
                               defstermedonidfterint ( v , b , c ) )
        )
   };



/* A partir d'une suite sid et d'une fform b , genere pour tout idi de sid :
   ( il existe id1 ) ... ( il existe idn ) : b                          */

fform sidenfform ( sid a , fform b )
   { with ( a )
        ( VideId ( ) : b ,
          ConsId ( x,y ) : FFormQuantifExist ( x , sidenfform ( y , b ) )
        )
   };



/* A partir d'une liste a de ftermes et d'un fterme b, renvoie
   une liste vide si la liste a donnee ne contient pas de PC contenant b,
   renvoie la liste de tous les ftermes du type PC(..., b, ...) sinon   */

fsterme dsPC ( fsterme a , fterme b )
   { with ( a )
        ( VideFTerme ( ) : VideFTerme ( ) ,
          ConsFTerme ( y,z ) :
           with ( y )
                ( FFoncteur ( f,d ) :
                   with ( f )
                        ( FNrExplTyp ( e ) :
                           ( e == "cp" ) && dssfterme ( d,b )
                                ? ConsFTerme ( y , dsPC ( d @ z , b ) )
                                : dsPC ( d @ z , b ) ,
                          default : dsPC ( d @ z , b )
                        ) ,
                  default : dsPC ( z,b )
                )
        )
   };
```

```
/* A partir d'une liste a de ftermes et d'un fterme b, renvoie une liste
   vide si la liste a donnee ne contient pas d'union disjointe contenant b,
   renvoie la liste de tous les ftermes du type UNION (..., b, ...) sinon */

fsterme dsUNION ( fsterme a , fterme b )
   { with ( a )
        ( VideFTerme ( ) : VideFTerme ( ) ,
          ConsFTerme ( y,z ) :
           with ( y )
                ( FFoncteur ( f,d ) :
                   with ( f )
                        ( FNrExplTyp ( e ) :
                           ( e == "union" ) && dssfterme ( d,b )
                                ? ConsFTerme ( y , dsUNION ( d @ z , b ) )
                                : dsUNION ( d @ z , b ) ,
                          default : dsUNION ( d @ z , b )
                        ) ,
                  default : dsUNION ( z,b )
                )
        )
   };



/* A partir d'une liste a de ftermes et d'un fterme b, renvoie
   une liste vide si la liste a donnee ne contient pas une sequence
   d'elements de type b, renvoie la liste de tous les ftermes contenant
   SEQ ( b ) sinon
                                                                        */

fsterme dsSEQ ( fsterme a , fterme b )
   { with ( a )
        ( VideFTerme ( ) : VideFTerme ( ) ,
          ConsFTerme ( y,z ) :
           with ( y )
                ( FFoncteur ( f,d ) :
                   with ( f )
                        ( FUnExplTyp ( e ) :
                           ( e == SeType ( "seq" ) ) && dssfterme ( d,b )
                                ? ConsFTerme ( y , dsSEQ ( d @ z , b ) )
                                : dsSEQ ( d @ z , b ) ,
                          default : dsSEQ ( d @ z , b )
                        ) ,
                  default : dsSEQ ( z,b )
                )
        )
   };

/* Intersection de deux listes de type iffi                             */

iffi interlistiffi ( iffi a , iffi b )
   { with ( a )
        ( VideIFFI ( ) : VideIFFI ( ) ,
          ConsIFFI ( x,y ) :
           dsiffi ( b,x )
                ? ConsIFFI ( x , interlistiffi ( y,b ) )
                : interlistiffi ( y,b )
        )
   };
```

```
/* Difference d'une liste a par rapport a une liste b ou a et b sont de type
   iffi                                                                    */

iffi difflistesiffi ( iffi a , iffi b )
   { with ( a )
        ( VideIFFI ( ) : VideIFFI ( ) ,
          ConsIFFI ( x,y ) :
           dsiffi ( b,x )
                ? difflistesiffi ( y,b )
                : ConsIFFI ( x , difflistesiffi ( y,b ) )
        )
   };



/* Difference d'une liste a par rapport a une liste b ou a et b sont de type
   sid                                                                     */

sid diffdeuxlistessid ( sid a , sid b )
   { with ( a )
        ( VideId ( ) : VideId ( ) ,
          ConsId ( x,y ) :
           dssidid ( b,x )
                ? diffdeuxlistessid ( y,b )
                : ConsId ( x , diffdeuxlistessid ( y,b ) )
        )
   };



/* Construction de l'attribut vrimp pour une suite a donnee d'identificateurs
   de variables pour lesquelles il faut une equation implicite i.e. variables
   ne figurant pas dans la suite b                                         */

iffi desiddonvrimp ( sid a , sid b , idfterme c )
   { with ( a )
        ( VideId ( ) : VideIFFI ( ) ,
          ConsId ( x,y ) :
           ( ! dssidid ( b , x ) )
                ? ConsIFFI ( QuadrIFFI ( x , FTermeVide ( ) ,
                   deidftermedonfter ( c,x ) , -1 ) ,
                   desiddonvrimp ( y , b , c ) )
                : desiddonvrimp ( y , b , c )
        )
   };



/* Mise a jour de l'attribut vrimp sur un AND ;
   recoit une liste a de ( idix , ftermeex , ftermetx , INTcx )
          une liste b de ( idm , ftermentx , INTnc )
          une liste c de ( id )
   produit une liste vide si a est vide , sinon
   produit une liste de ( idix' , ftermeex' , ftermetx' , INTcx' )
   ou idix' = idix,
      ftermeex' = sel ( ftermetx , ftermeex ) si (1) ou (2)
                = ftermeex si (3) ou (4),
      ftermetx' = ftermentx si (1) ou (2)
                = ftermetx si (3) ou (4),
      INT cx' = 1 si (1)
              = INTcx si (2), (3) ou (4)
avec pour le ( idix , ftermeex , ftermetx , INTcx ) de a considere
(1) INTcx < 0 et
```

```
    idix appartient a c et
    il existe dans b un element tel que idm = idix,
(2) INTcx > 0 et
    idix n'appartient pas a c et
    il existe dans b un element tel que idm = idix,
(3) pour tout INTcx
    idix n'appartient pas a c et
    il n'existe pas dans b d'element tel que idm = idix,
(4) INTcx < 0 et
    idix n'appartient pas a c et
    il existe dans b un element tel que idm = idix,
les autres combinaisons etant theoriquement impossibles.                */

iffi majdevrimpet ( iffi a , idfterint b , sid c )
   { with ( a )
        ( VideIFFI ( ) : VideIFFI ( ) ,
          ConsIFFI ( x,y ) :
           with( x )
                ( QuadrIFFIVide ( ) : majdevrimpet ( y , b , c ) ,
                  QuadrIFFI ( ix , ex , tx , cx ) :
                   with ( dsvtypent ( b , ix ) )
                        ( TriplIdFTerIntVide ( ) :      /* cas (3) */
                           ConsIFFI ( x , majdevrimpet ( y , b , c ) ) ,
                          TriplIdFTerInt ( m , ntx , nc ) : /* cas (1)(2)(4)*/
                                ( dssidid ( c , ix ) )
        ? /* cas (1) */ ConsIFFI ( QuadrIFFI ( ix , consfterme ( ex ,
                                    FSel ( tx , FTermeVide ( ) ) ) , ntx , 1 )
                                  , majdevrimpet ( y , b , c ) )
        : ( cx > 0 )
            ? /* cas (2) */ ConsIFFI ( QuadrIFFI ( ix , consfterme ( ex ,
                                FSel ( tx , FTermeVide ( ) ) ) , ntx , cx ) ,
                                majdevrimpet ( y , b , c ) )
            : /* cas (3) */ ConsIFFI ( x , majdevrimpet ( y , b , c ) )
                        )
              )
        )
   };



/* Mise a jour de l'attribut vrimp sur un OR
   cfr. mise a jour de vrimp sur un AND mais sel devient as              */

iffi majdevrimpou ( iffi a , idfterint b , sid c )
   { with ( a )
        ( VideIFFI ( ) : VideIFFI ( ) ,
          ConsIFFI ( x,y ) :
           with( x )
                ( QuadrIFFIVide ( ) : majdevrimpou ( y , b , c ) ,
                  QuadrIFFI ( ix , ex , tx , cx ) :
                   with ( dsvtypent ( b , ix ) )
                        ( TriplIdFTerIntVide ( ) :      /* cas (3) */
                           ConsIFFI ( x , majdevrimpou ( y , b , c ) ) ,
                          TriplIdFTerInt ( m , ntx , nc ) : /* cas (1)(2)(4)*/
                                ( dssidid ( c , ix ) )
        ? /* cas (1) */ ConsIFFI ( QuadrIFFI ( ix, consfterme ( ex ,
                                    FAs ( tx , FTermeVide ( ) ) ) , ntx , 1 )
                                  , majdevrimpou ( y , b , c ) )
        : ( cx > 0 )
            ? /* cas (2) */ ConsIFFI ( QuadrIFFI ( ix, consfterme ( ex ,
                                FAs ( tx , FTermeVide ( ) ) ) , ntx , cx ) ,
                                majdevrimpou ( y , b , c ) )
              : /* cas (3) */ ConsIFFI ( x , majdevrimpou ( y , b , c ) )
                        )
```

```
                )
        )
    };


/* Mise a jour de l'attribut vrimp sur un forall ;
   Pour toutes les variables dont le type contient une sequence correspondant
   au forall, on genere une equation de selection ITH et on assure que leur
   compteur est positif.
   Pour les autres variables, on "genere meme structure" et on laisse leur
   compteur inchange                                                    */

iffi majdevrimpseq ( iffi a , idfterint b , id c )
   { with ( a )
        ( VideIFFI ( ) : VideIFFI ( ) ,
          ConsIFFI ( x,y ) :
           with( x )
                ( QuadrIFFIVide ( ) : majdevrimpseq ( y , b , c ) ,
                  QuadrIFFI ( ix , ex , tx , cx ) :
                   with ( dsvtypent ( b , ix ) )
                        ( TriplIdFTerIntVide ( ) :
                           ConsIFFI ( x , majdevrimpseq ( y , b , c ) ) ,
                          TriplIdFTerInt ( m , ntx , nc ) :
                           ConsIFFI ( QuadrIFFI ( ix , consfterme ( ex ,
                                      FNieme ( nvlid ( c ) , FTermeVide ( ) ) )
                                      , ntx , 1 ) ,
                                majdevrimpseq ( y , b , c ) )
                        )
                )
        )
   };


/* recoit une liste a de ( ida , ftermeb , ftermec , INTd )
   renvoie une liste vide si a est vide sinon
   pour tout ida appartenant a un element de a
   donne un element ( ida' , fterme , INTd' )
   ou ida' = ida,
      fterme = le PC dans lequel figure le type de ida si un tel PC existe,
      INTd' = INTd,
   ne donne rien sinon                                                  */

idfterint varPCint ( iffi a , idfterme b )
   { with ( a )
        ( VideIFFI ( ) : VideIdFTerInt (ₙ) ,
          ConsIFFI ( y,z ) :
           with (y )
                ( QuadrIFFIVide ( ) : VideIdFTerInt ( ) ,
                  QuadrIFFI ( m , n , o , p ) : defstermedonidfterint
                                ( dsPC ( deidftermedonfster ( b ) , o ) ,
                                m , p ) @ varPCint ( z,b )
                )
        )
   };


/* recoit une liste a de ( ida , ftermeb , ftermec , INTd )
   renvoie une liste vide si a est vide sinon
   pour tout ida appartenant a un element de a
   donne un element ( ida' , fterme , INTd' )
   ou ida' = ida,
```

```
      fterme = l'union disjointe dans laquelle figure le type de ida si
               une telle union disjointe existe,
      INTd' = INTd,
   ne donne rien sinon                                                    */

idfterint varUNIONint ( iffi a , idfterme b )
   { with ( a )
        ( VideIFFI ( ) : VideIdFTerInt ( ) ,
          ConsIFFI ( y,z ) :
           with ( y )
                ( QuadrIFFIVide ( ) : VideIdFTerInt ( ) ,
                  QuadrIFFI ( m , n , o , p ) : defstermedonidfterint
                        ( dsUNION ( deidftermedonfster ( b ) , o ) ,
                        m , p ) @ varUNIONint ( z,b )
                )
        )
   };



/* recoit une liste a de ( ida , ftermeb , ftermec , INTd )
   renvoie une liste vide si a est vide sinon
   pour tout ida appartenant a un element de a
   donne un element ( ida' , fterme , INTd' )
   ou ida' = ida,
      fterme = la sequence dans laquelle figure le type de ida si
               une telle sequence existe,
      INTd' = INTd,
   ne donne rien sinon                                                    */

idfterint varSEQint ( iffi a , idfterme b )
   { with ( a )
        ( VideIFFI ( ) : VideIdFTerInt ( ) ,
          ConsIFFI ( y,z ) :
           with ( y )
                ( QuadrIFFIVide ( ) : VideIdFTerInt ( ) ,
                  QuadrIFFI ( m , n , o , p ) : defstermedonidfterint
                        ( dsSEQ ( deidftermedonfster ( b ) , o ) ,
                        m , p ) @ varSEQint ( z,b )
                )
        )
   };



/* Comparaison d'un element ( ida , ftermea, INTa ) donne avec chacun
   des elements d'une liste de ( idb , ftermeb , INTb ) donnee.
   S'il existe un ( idb , ftermeb , INTb ) tel que ftermea = ftermeb
   et INTa < 0 alors renvoie ida, sinon renvoie le vide                   */

id traiterelement ( triplidfterint a , idfterint b )
   { with ( a )
        ( TriplIdFTerIntVide ( ) : IdVide ( ) ,
          TriplIdFTerInt ( x , y , z ) :
           with ( b )
                ( VideIdFTerInt ( ) : IdVide ( ) ,
                  ConsIdFTerInt ( c , d ) :
                   with ( c )
                        ( TriplIdFTerIntVide ( ) : IdVide ( ) ,
                          TriplIdFTerInt ( r , s , t ) :
                            ( ( z < 0 ) && ( y == s ) )
                                 ? x
                                 : traiterelement ( a , d )
                        )
```

```
                )
        )
    };


/* Production de la liste des variables de a pour lesquelles on est au bon
   endroit pour attacher l'equation implicite de type Sel ou As i.e. variables
   pour lesquelles on a une meme composition de type venant des sous-arbres
   gauche et droit et dont le compteur est negatif                          */

sid donlistvbonend ( idfterint a , idfterint b )
   { with ( a )
        ( VideIdFTerInt ( ) : VideId ( ) ,
          ConsIdFTerInt ( x,y ) :
           with ( traiterelement ( x,b ) )
                ( IdVide ( ) : donlistvbonend ( y,b ) ,
                  default : ConsId ( traiterelement ( x,b ) ,
                                     donlistvbonend ( y,b ) )
                )
        )
   };


/* Production de la liste des variables pour lesquelles on est au bon
   endroit pour attacher l'equation implicite FORALL i.e. variables de
   la liste a dont le compteur est negatif                                  */

sid donvbonendforall ( idfterint a )
   { with ( a )
        ( VideIdFTerInt ( ) : VideId ( ) ,
          ConsIdFTerInt ( v,z ) :
           with ( v )
                ( TriplIdFTerIntVide ( ) : donvbonendforall ( z ) ,
                  TriplIdFTerInt ( w , x , y ) :
           ( y < 0 )    ? ConsId ( w , donvbonendforall ( z ) )
                        : donvbonendforall ( z )
                )
        )
   };


/* Production, a partir de la liste a de type idfterint, de la liste des
   variables pour lesquelles on a su generer une equation implicite sur le
   FORALL                                                                   */

sid donvardevtypent ( idfterint a )
   { with ( a )
        ( VideIdFTerInt ( ) : VideId ( ) ,
          ConsIdFTerInt ( v,z ) :
           with ( v )
                ( TriplIdFTerIntVide ( ) : donvardevtypent ( z ) ,
                  TriplIdFTerInt ( w , x , y ) :
                                 ConsId ( w , donvardevtypent ( z ) )
                )
        )
   };


/* recoit une liste b d'identificateurs de variables pour lesquelles
   il faut une equation implicite et une liste a de ftermes representant
```

```
   des equations implicites,
   selectionne dans a les ftermes correspondant aux id de b                */

idfterme eqattach ( idfterme a , sid b )
   { with ( a )
        ( VideIdFTerme ( ) : VideIdFTerme ( ) ,
          ConsIdFTerme ( w,z ) :
           with ( w )
                ( CoupleIdFTermeVide ( ) : eqattach ( z,b ) ,
                  CoupleIdFTerme ( x,y ) :
           ( dssidid ( b,x ) )
                ? ConsIdFTerme ( CoupleIdFTerme ( x,y ) , eqattach ( z,b ) )
                : eqattach ( z,b )
                )
        )
   };


/* Generation d'une fform qui est une conjonction de fforms correspondant
   chacune a une equation implicite de longueur                            */

fform eqlg ( sid a , idfterme b , id c , id d )
   { with ( a )
        ( VideId ( ) : FFormVide ( ) ,
          ConsId ( x,y ) : FFormEt ( FAtome ( FEgal ( FLength
                ( FVariable ( c ) ) , FLength ( donseqimp ( b , x , d ) ) ) ),
                eqlg ( y , b , c , d ) )
        )
   };


/* A partir d'un nom de variable b donne , d'un indice de sequence c donne
   et de la liste a des equations implicites, trouve dans a la sequence
   apparaissant dans le type de b et correspondant a l'indice c            */


fterme donseqimp ( idfterme a , id b , id c )
   { with ( a )
        ( VideIdFTerme ( ) : FTermeVide ( ) ,
          ConsIdFTerme ( w,z ) :
           with ( w )
                ( CoupleIdFTermeVide ( ) : donseqimp ( z , b , c ) ,
                  CoupleIdFTerme ( x,y ) :
           ( b == x ) ? with ( y )
                        ( FNieme ( m,n ) :
                           ( m == nvlid ( c ) )
                                ? n
                                : donseqimp ( ConsIdFTerme ( CoupleIdFTerme
                                   ( b , n ) , VideIdFTerme ( ) ) , b , c ),
                          FSel ( m,n ) : donseqimp ( ConsIdFTerme
                                ( CoupleIdFTerme ( b , n ) ,
                                VideIdFTerme ( ) ) , b , c ),
                          FAs ( m,n ) : donseqimp ( ConsIdFTerme
                                ( CoupleIdFTerme ( b , n ) ,
                                VideIdFTerme ( ) ) , b , c ),
                          default : FTermeVide ( )
                        )
                      : donseqimp ( z , b , c )
                )
        )
   };
```

```
/* Etant donnees une fform correspondant a une explicitation d'operation
   et une liste d'equations implicites y correspondant, renvoie la fform
   correspondant au tout                                                  */

fform eqimplenfform ( idfterme a , fform b )
   { with ( a )
        ( VideIdFTerme ( ) : b ,
          ConsIdFTerme ( w,z ) :
           with ( w )
                ( CoupleIdFTermeVide ( ) : eqimplenfform ( z,b ) ,
                  CoupleIdFTerme ( x,y ) :
           FFormQuantifExist ( x , FFormEt ( eqimplenfform ( z,b ) ,
           FAtome ( FEgal ( FVariable ( x ) , y ) ) ) )
                )
        )
   };



/* But : trouver pour chaque variable l'unique equation implicite qui
   correspond a son type.
   Etant donnees 1) une liste a de ( ida , ftermeb , ftermec, INTd )
                    ou ftermeb represente une equation implicite,
                    et ftermec represente un type de variable et
                 2) une liste b de ( ida' , ftermec' )
                    ou ftermec' represente un type de variable,
   produit pour chaque element de a un element ( ida" , ftermeb" )
   tel que ida" = ida et
           ftermeb" = ftermeb ou le terme vide a ete remplace par ida'
                      si il existe dans b un fterme c' = ftermec,
   ne produit rien sinon                                                  */

idfterme selecteqimp ( iffi a , idfterme b )
   { with ( a )
        ( VideIFFI ( ) : VideIdFTerme ( ) ,
          ConsIFFI ( u,z ) :
           with ( u )
                ( QuadrIFFIVide ( ) : selecteqimp ( z,b ) ,
                  QuadrIFFI ( v , w , x , y ) :
           with ( deidftermedonid ( b , x ) )
                ( IdVide ( ) : selecteqimp ( z,b ) ,
                  MinId ( m ) : ConsIdFTerme ( CoupleIdFTerme ( v ,
                   ajoutalafin ( w , FVariable ( MinId ( m ) ) ) ) ,
                   selecteqimp ( z,b ) ) ,
                  MajId ( m ) : selecteqimp ( z,b )
                )
                )
        )
   };



/* renvoie un fterme egal a un fterme a donne ou on a substitue dans a
   un fterme b donne au fterme vide si a = le fterme vide ou est compose
   de FSel, FAs ou FNieme dans lesquels figure le fterme vide,
   renvoie le fterme vide sinon                                           */

fterme ajoutalafin ( fterme a , fterme b )
   { with ( a )
        ( FTermeVide ( ) : b ,
          FSel ( x,y ) : FSel ( x , ajoutalafin ( y,b ) ) ,
          FAs ( x,y ) : FAs ( x , ajoutalafin ( y,b ) ) ,
```

```
          FNieme ( x,y ) : FNieme ( x , ajoutalafin ( y,b ) ) ,
          default : FTermeVide ( )
        )
   };


/* recoit deux fois une liste de ( ftermea , ftermeb )
   envoie une liste de ( ftermea , ftermeb' ) ou les ftermeb' sont les
   ftermeb exprimes en fonction des types de base                          */

fterfter detypexpdontypexpdec ( fterfter a , fterfter b )
   { with ( a )
        ( VideFTerFTer ( ) : VideFTerFTer ( ) ,
          ConsFTerFTer ( w,z ) :
           with ( w )
                ( CoupleFTerFTerVide ( ) : detypexpdontypexpdec ( z , b ) ,
                  CoupleFTerFTer ( x , y ) :
           ConsFTerFTer ( CoupleFTerFTer ( x , decompfterme ( a , b , y ) ) ,
                          detypexpdontypexpdec ( z , b ) )
                )
        )
   };


/* recoit deux fois une liste de ( ftermea , ftermeb ) et un fterme c
   decompose en types de base le fterme c en fonction de la liste donnee ) */

fterme decompfterme ( fterfter a , fterfter b , fterme c )
   { with ( a )
        ( VideFTerFTer ( ) : FTermeVide ( ) ,
          ConsFTerFTer ( w,z ) :
           with ( w )
                ( CoupleFTerFTerVide ( ) : FTermeVide ( ) ,
                  CoupleFTerFTer ( x , y ) :
           with ( c )
                ( FFoncteur ( f , ConsFTerme ( g , h ) ) :
                   FFoncteur ( f , ConsFTerme ( decompfterme ( b , b , g ) ,
                                                decompfsterme ( b , b , h ) ) ) ,
                  FVariable ( v ) :
                        ( x == c )      ? decompfterme ( b , b , y )
                                        : decompfterme ( z , b , c ) ,
                  FTypeBase ( k ) : c ,
                  default : FTermeVide ( )
                )
                )
        )
   };


/* en fonction d'une liste de ( fterme , fterme ) donnee deux fois ,
   decompose en fonctions des types de base chaque fterme d' une liste
   de fsterme c donnee                                                     */

fsterme decompfsterme ( fterfter a , fterfter b , fsterme c )
   { with ( c )
        ( VideFTerme ( ) : VideFTerme ( ) ,
          ConsFTerme ( x , y ) :
              ConsFTerme ( decompfterme ( a , b , x ) ,
                           decompfsterme ( a , b , y ) )
        )
   };
```

```
/* transforme un identificateur en lui-meme ecrit avec un premier
   caractere en majuscule, les autres en minuscules                    */

id envar ( id a )
    { with ( a )
        ( IdVide ( ) : IdVide ( ) ,
          MinId ( x ) : MinId ( repeatCHAR ( ( STRtoupper ( x ) )[1] , 1 )
                                  # STRtolower ( x[2:] ) ) ,
          MajId ( y ) : MajId ( repeatCHAR ( ( STRtoupper ( y ) )[1] , 1 )
                                  # STRtolower ( y[2:] ) )
        )
    };


/* transforme un identificateur en lui-meme ecrit en minuscules        */

id enmin ( id a )
    { with ( a )
        ( IdVide ( ) : IdVide ( ) ,
          MinId ( x ) : a ,
          MajId ( y ) : MajId ( STRtolower ( y ) )
        )
    };


/* transforme une suite d'objets tout en minuscules en une suite d'objets tels
   que leur premier caractere soit en majuscule et les autres en minuscules  */

sid ensvar ( sid a )
    { with ( a )
        ( VideId ( ) : VideId ( ) ,
          ConsId ( x , y ) : ConsId ( envar ( x ) , ensvar ( y ) )
        )
    };


/* Deuxieme transformation : RSL + FOPL -> PROLOG */
/* ============================================== */


/* recoit une liste a de triplets de type ( fterfter, psantec, listefterfter )
       et une liste b de triplets de type ( fterfter, psantec, listefterfter ),
   produit la liste de triplets de type ( fterfter, psantec, listefterfter )
       egale a la "concatenation" des listes a et b ou la "concatenation" est
       definie comme suit :
   pour tout triplet ( ( compa , sela )* , psanteca , listefterftera ) de a,
       on genere
   pour tout triplet ( ( compb , selb )* , psantecb , listefterfterb ) de b,
       un triplet
       ( ( compa , sela )* @ ( compb , selb )* , "psanteca,psantecb" ,
                                  listefterftera @ listefterfterb )   */

ifterfterpsantec concatPC ( ifterfterpsantec a , ifterfterpsantec b )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ),
          ConsIFTerFTerPSAntec ( x,y ) :
                              concatuneligne PC ( b,x ) @ concatPC ( y,b )
```

```
        )
    };


/* recoit un triplet b de type ( fterfter , psantec , listefterfter ) de la
        forme ( ( compa , sela )* , psanteca , listefterftera ),
        et une liste a de triplets de type
                                        ( fterfter , psantec , listefterfter ),
   produit la liste de triplets de type ( fterfter , psantec , listefterfter )
         obtenue par la "concatenation" de b a chacun des elements de a      */

ifterfterpsantec concatunelignePC (ifterfterpsantec a,triplifterfterpsantec b)
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ),
          ConsIFTerFTerPSAntec ( x,y ) : ConsIFTerFTerPSAntec
                ( concatdeuxtriplets ( x,b ) , concatunelignePC ( y,b ) )
        )
    };


/* recoit deux triplets de type ( fterfter , psantec , listefterfter )
        soient  (( compa , sela )* , psanteca , listefterftera ) et
                (( compb , selb )* , psantecb , listefterfterb ) ,
   produit le triplet de type ( fterfter , psantec , listefterfter )
        ( ( compa , sela )* @ ( compb , selb )* , "psanteca,psantecb" ,
                                        listefterftera @ listefterfterb )
                                                                          */

triplifterfterpsantec concatdeuxtriplets
                        ( triplifterfterpsantec a , triplifterfterpsantec b )
    { with ( a )
        ( TriplIFTerFTerPSAntecVide ( ) : b,
          TriplIFTerFTerPSAntec ( m , n , l ) :
            with ( b )
                ( TriplIFTerFTerPSAntecVide ( ) : a,
                  TriplIFTerFTerPSAntec ( q , r , p ) :
                    TriplIFTerFTerPSAntec ( m @ q , n @ r , l @ p )
                )
        )
    };


/* recoit une liste a de type ifterfterpsantec
   produit la liste de type ifterfterpsantec egale a a ou chaque element
        ne figure qu'en une et une seule occurrence                       */

ifterfterpsantec elimindoublons1 ( ifterfterpsantec a )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ),
          ConsIFTerFTerPSAntec ( x,y ) :
            dsifterfterpsantec ( y,x )
                ? elimindoublons1 ( y )
                : ConsIFTerFTerPSAntec ( x , elimindoublons1 ( y ) )
        )
    };


/* recoit une liste a de type ifterfterpsantec
        et un element b de type triplifterfterpsantec
   renvoie vrai si b appartient a a, faux sinon                           */
```

```
BOOL dsifterfterpsantec ( ifterfterpsantec a , triplifterfterpsantec b )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : false,
          ConsIFTerFTerPSAntec ( x,y ) :
            with ( x )
                ( TriplIFTerFTerPSAntecVide ( ) : ( x == b ) ? true
                        : dsifterfterpsantec ( y,b ),
                  TriplIFTerFTerPSAntec ( m , n , p ) : ( x == b ) ? true
                        : dsifterfterpsantec ( y,b )
                )
        )
    };



/* recoit une liste a de type ifterfterpsantec
                        de forme ( fterftera , psanteca , listefterftera )*,
   produit la liste de type ifterfterpsantec
                       de forme ( fterftera' , psanteca , listefterftera' )* ou
        fterftera' est la liste ferftera de laquelle on a elimine les doublons
        c-a-d que si un element figure en plusieurs occurrences dans ftertera,
        il ne figure plus qu'en une seule dans fterftera'
        listefterftera' est la liste listeferftera de laquelle on a elimine les
        doublons c-a-d que si un element figure en plusieurs occurrences dans
        un element fterftera de la liste listefterftera, il ne figure plus
        qu'en une seule dans l'element fterftera de listefterftera'          */

ifterfterpsantec elimindoublons2 ( ifterfterpsantec a )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ),
          ConsIFTerFTerPSAntec ( x,y ) :
            with ( x )
                ( TriplIFTerFTerPSAntecVide ( ) : elimindoublons2 ( y ),
                  TriplIFTerFTerPSAntec ( n , p , m ) :
                     ConsIFTerFTerPSAntec ( TriplIFTerFTerPSAntec (
                        elimindoublonsfterfter ( n ) , p ,
                        elimindoublonslistefterfter ( m ) ) ,
                        elimindoublons2 ( y ) )
                )
        )
    };


/* recoit une liste a de type fterfter,
   produit une liste de type fterfter contenant une et une seule occurrence
        de chaque element de a                                               */

fterfter elimindoublonsfterfter ( fterfter a )
    { with ( a )
        ( VideFTerFTer ( ) : VideFTerFTer ( ),
          ConsFTerFTer ( x,y ) : ( dsfterfter ( y,x ) )
                ? elimindoublonsfterfter ( y )
                : ConsFTerFTer ( x , elimindoublonsfterfter ( y ) )
        )
    };



/* recoit une liste a de type listefterfter
                        de forme ( fterftera )*,
   produit la liste de type listefterfter
                       de forme ( fterftera' )* ou
```

```
        fterftera' est la liste ferftera de laquelle on a elimine les doublons
        c-a-d que si un element figure en plusieurs occurrences dans ftertera,
        il ne figure plus qu'en une seule dans fterftera'                    */

listefterfter elimindoublonslistefterfter ( listefterfter a )
    { with ( a )
        ( VideListeFTerFTer ( ) : VideListeFTerFTer ( ) ,
          ConsListeFTerFTer ( x , y ) :
                ConsListeFTerFTer ( elimindoublonsfterfter ( x ) ,
                                    elimindoublonslistefterfter ( y ) )
        )
    } ;



/* recoit une liste a de type fterfter dont les elements sont des couples de la
        forme ( fterme , fterme ), et
          un couple b de la forme ( fterme , fterme ),
   renvoie vrai si b est un element de la liste a, faux sinon            */

BOOL dsfterfter ( fterfter a , couplefterfter b )
    { with ( a )
        ( VideFTerFTer ( ) : false,
          ConsFTerFTer ( x,y ) : ( b == x ) ? true
                                            : dsfterfter ( y,b )
        )
    };



/* recoit une liste a de type ifterfterpsantec et
          une liste b de type idfterme
   produit la liste de type ifterfterpsantec egale a a mise a jour pour un
        passage sur un noeud AND i.e.
        pour tout element (( comp , csel )* , psantec , listefterfter ) de a
        on traite ( comp , csel )* comme suit :
      - pour tout element de ( comp , csel )* de la forme (cons , sel ( j,k ))
        - ou il existe dans ( comp , csel )* un element
                ( constuple ( ... , a : _ , ... ) , csel ) et csel = k
          et on remplace cet element par
                ( constuple ( ... , a : cons , ... ) , k )
        - ou il n'existe pas dans ( comp , csel )* d'element de la forme
                ( constuple ( ... , a : _ , ... ) , csel ) et csel = k
          et - on genere en utilisant b un element
                ( constuple ( ... , a : _ , ... ) , k ) qu'on ajoute a
                ( comp , csel )* ( generation de constuple ) puis
             _ on remplace cet element par
                ( constuple ( ... , a : cons , ... ) , k ) ( garnissage )
      - pour tout element de ( comp , csel )* different de (cons , sel ( j,k ))
        on garde cet element inchange;
        psantec reste le meme;
        on traite listefterfter comme suit :
        tout element fterfter de la liste listefterfter est modifie comme
        explique pour ( comp , csel )*                                    */

ifterfterpsantec majtripletsPC ( ifterfterpsantec a , idfterme b )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ),
          ConsIFTerFTerPSAntec ( x,y ) :
            with ( x )
                ( TriplIFTerFTerPSAntecVide ( ) : majtripletsPC ( y,b ),
                  TriplIFTerFTerPSAntec ( n , p , m ) :
                    ConsIFTerFTerPSAntec ( TriplIFTerFTerPSAntec (
                    majfterfterPC ( n , n , b ) , p , majsubseqPC ( m , b ) ),
```

```
                majtripletsPC ( y,b ) )
            )
        )
    };


/* recoit deux occurrences a et b d'une liste de type fterfter
       de la forme ( comp , csel )* et une liste c de type idfterme
   produit une liste de type fterfter egale a a modifiee comme suit
       pour tout element de a de la forme
       - ou il existe dans ( comp , csel )* un element
              ( constuple ( ... , a : _ , ... ) , csel ) et csel = k
         et on remplace cet element par
              ( constuple ( ... , a : cons , ... ) , k )
       - ou il n'existe pas dans ( comp , csel )* d'element de la forme
              ( constuple ( ... , a : _ , ... ) , csel ) et csel = k
         et - on genere en utilisant b un element
              ( constuple ( ... , a : _ , ... ) , k ) qu'on ajoute a
              ( comp , csel )* ( generation de constuple ) puis
           _ on remplace cet element par
              ( constuple ( ... , a : cons , ... ) , k ) ( garnissage )
     - pour tout element de ( comp , csel )* different de (cons , sel ( j,k ))
       on garde cet element inchange                                    */

fterfter majfterfterPC ( fterfter a , fterfter b , idfterme c )
    { with ( a )
        ( VideFTerFTer ( ) : b,
          ConsFTerFTer ( x,y ) :
            with ( x )
              ( CoupleFTerFTerVide ( ) : majfterfterPC ( y , b , c ),
                CoupleFTerFTer ( m,n ) :
                  with ( n )
                    ( FSel ( p,q ) : majfterfterPC ( y ,
                                majunseldsfterfter ( b , x , m , p , q , c ) ,
                                c ) ,
                      default : majfterfterPC ( y , b , c )
                    )
              )
        )
    };


/* recoit une liste a de type fterfter d'elements de la forme (fterme,fterme),
          un couple b appartenant a a de la forme ( fterme1 , fterme2 ),
          un fterme c ou c = fterme1 de b,
          un fterme d et un fterme e tels que sel ( d,e ) = fterme2 de b,
          une liste f de type idfterme,
   produit une liste de type fterfter d'elements de la forme (fterme,fterme)
       egale a a dont l'element b a ete enleve et remplace par
       ( fterme1' , fterme2' ) ou fterme1' = constuple ( ... , d : c , ... )
       et fterme2' = e                                                  */

fterfter majunseldsfterfter ( fterfter a , couplefterfter b , fterme c ,
                              fterme d , fterme e , idfterme f )
    { constuplesliste ( a,d )
        ? retirerelement ( completerconstuple ( a , b , c , d , e ) , b )
        : retirerelement ( completerconstuple ( creerconstuple
              ( a , d , e , f ) , b , c , d , e ) , b )
    };
```

```
/* recoit une liste a de type fterfter d'elements de la forme (fterme1,fterme2)
        et un element b de type fterme
   renvoie vrai si il existe un element de a tel que fterme1 soit de la forme
        constuple ( ... , b : ... , ... ), faux sinon                    */

BOOL constuplesliste ( fterfter a , fterme b )
    { with ( a )
        ( VideFTerFTer ( ) : false,
          ConsFTerFTer ( x,y ) :
            with ( x )
                ( CoupleFTerFTerVide ( ) : constuplesliste ( y,b ),
                  CoupleFTerFTer ( m,n ) :
                    with ( m )
                        ( FConstuple ( z ) : dsconstuple ( z,b )
                                             ? true
                                             : constuplesliste ( y,b ),
                          default : constuplesliste ( y,b )
                        )
                )
        )
    };



/* recoit une suite a de ftermes et un fterme b
   renvoie vrai si les elements de a correspondent a des couples
        ( fterme1 , fterme2 ) ou fterme1 designe un nom de champ et
        il existe un fterme1 = b, renvoie faux sinon                     */

BOOL dsconstuple ( fsterme a , fterme b )
    { with ( a )
        ( VideFTerme ( ) : false,
          ConsFTerme ( x,y ) :
            with ( x )
                ( FChampObj ( m,n ) : ( m == b ) ? true
                                                 : dsconstuple ( y,b ),
                  default : false
                )
        )
    };



/* recoit une liste a de type fterfter d'elements de la forme
        ( ftermea , ftermeb ) = ( comp , csel ),
          un couple b appartenant a a de la forme ( fterme1 , fterme2 ),
          un fterme c ou c = fterme1 de b,
          un fterme d et un fterme e tels que sel ( d,e ) = fterme2 de b,
   produit une liste de type fterfter d'elements de la forme ( comp , csel )
        egale a a dont l'element b a ete remplace par ( fterme1' , fterme2' )
        ou fterme1' = constuple ( ... , d : c , ... ) et fterme2' = e    */

fterfter completerconstuple ( fterfter a , couplefterfter b , fterme c,
                              fterme d , fterme e )
    { with ( a )
        ( VideFTerFTer ( ) : VideFTerFTer ( ),
          ConsFTerFTer ( x,y ) :
            with ( x )
                ( CoupleFTerFTerVide ( ) : ConsFTerFTer ( x ,
                                completerconstuple ( y , b , c , d , e ) ),
                  CoupleFTerFTer ( m,n ) :
                    with ( m )
                        ( FConstuple ( z ) :
                            dsconstuple ( z,d )
```

```
                ? ConsFTerFTer ( CoupleFTerFTer ( FConstuple (
                                garnirelmtconstuple ( z , d , c ) ) , e ) , y )
                : ConsFTerFTer (x , completerconstuple ( y , b , c , d , e )),
                        default : ConsFTerFTer ( x ,
                                completerconstuple ( y , b , c , d , e ) )
                    )
                )
        )
    };


/* recoit une liste a de type fterfter d'elements de la forme (fterme,fterme),
         un fterme b correspondant a un nom de champ,
         un fterme c correspondant a une occurrence d'un type PC,
         une liste d de type idfterme contenant les explicitations de types,
   produit une liste de type fterfter d'elements de la forme (fterme,fterme)
        egale a a plus un element de la forme
        ( constuple ( ... , b : _ , ... ) , c )                              */

fterfter creerconstuple ( fterfter a , fterme b , fterme c , idfterme d )
    { ConsFTerFTer ( CoupleFTerFTer ( depcdonconstuple ( dsPC
        ( deidftermedonfster ( d ) , b ) ) , c ) , a )
    };


/* recoit une liste a dont les elements sont de type couplefterfter et
         un element b de type couplefterfter
   produit la liste a dont l'element b a ete ote si b appartient a a ,
          la liste a sinon                                                    */

fterfter retirerelement ( fterfter a , couplefterfter b )
    { with ( a )
        ( VideFTerFTer ( ) : VideFTerFTer ( ),
          ConsFTerFTer ( x,y ) : ( x == b )
                                ? retirerelement ( y,b )
                                : ConsFTerFTer ( x , retirerelement ( y,b ) )
        )
    };


/* recoit une liste a de ftermes representant des couples
         ( nom de champ, nom d'objet ),
         un fterme b correspondant a un nom de champ,
         un fterme c correspondant a un nom d'objet,
   produit une liste de ftermes egale a á ou le couple ( b , nom d'objet )
        s'il existe a ete remplace par ( b,c ),
   restitue a inchangee sinon                                                 */

fsterme garnirelmtconstuple ( fsterme a , fterme b , fterme c )
    { with ( a )
        ( VideFTerme ( ) : VideFTerme ( ),
          ConsFTerme ( x,y ) :
            with ( x )
                ( FChampObj ( m,n ) : ( m == b )
                        ? ConsFTerme ( FChampObj ( m,c ) , y )
                        : ConsFTerme ( x , garnirelmtconstuple ( y , b , c ) ),
                  default : a
                )
        )
    };
```

```
/* recoit une liste de ftermes contenant uniquement un fterme vide ou un
        fterme representant un PC,
   produit un fterme = constuple ( < nom-de-champ : _ >* ) ou les noms de
        champs sont dans l'ordre les types figurant dans le PC,
   produit un fterme vide sinon                                          */

fterme depcdonconstuple ( fsterme a )
    { with ( a )
        ( VideFTerme ( ) : FTermeVide ( ),
          ConsFTerme ( x,y ) :
            with ( x )
                ( FTermeVide ( ) : FTermeVide ( ),
                  FFoncteur ( m,n ) :
                        ( m == FNrExplTyp ( "cp" ) )
                            ? FConstuple ( createconstuple ( n ) )
                            : FTermeVide ( ),
                  default : FTermeVide ( )
                )
        )
    };


/* recoit une suite a de ftermes representant des noms de champ,
   produit une suite de ftermes dont chaque terme est un couple
        ( nom de champ , _ ) ou les noms de champs correspondent a la
        liste initiale                                                   */

fsterme createconstuple ( fsterme a )
   { with ( a )
        ( VideFTerme ( ) : VideFTerme ( ),
          ConsFTerme ( x,y ) :
                ( x == FTermeVide ( ) )
                        ? createconstuple ( y )
                        : ConsFTerme ( FChampObj ( x , FUnderScore ( ) ),
                                createconstuple ( y ) )
        )
   };


/* recoit une liste a de type listefterfter et une liste b de type idfterme,
   produit une liste de type listefterfter egale a a ou tous les elements de
        type fterfter sont modifies par majfterfterPC pour tenir compte du
        passage sur un AND                                              */

listefterfter majsubseqPC ( listefterfter a , idfterme b )
    { with ( a )
        ( VideListeFTerFTer ( ) : VideListeFTerFTer ( ),
          ConsListeFTerFTer ( x,y ) :
                ConsListeFTerFTer ( majfterfterPC ( x , x , b ) ,
                                    majsubseqPC ( y,b ) ) ,
        )
    };


/* recoit une liste a de type ifterfterpsantec d'elements
                de la forme ( ( comp , csel )* , psantec , listefterfter ) et
          une liste b de type idfterme
   produit la liste de type ifterfterpsantec egale a a mise a jour pour un
        passage sur un noeud OR i.e.
```

```
        pour tout element (( comp , csel )* , psantec , listefterfter ) de a
        genere pour tout element de ( comp , csel )* de la forme ( comp ,csel )
        tel que csel = as ( fterme1 , fterme2 ) ou fterme1 represente un type,
        le constructeur inj ( ... , ... ),
        tout element fterfter de listefterfter est modifie comme explique pour
        ( comp , csel )*;
        psantec reste inchange                                              */

ifterfterpsantec majtripletsUNION ( ifterfterpsantec a , idfterme b )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ),
          ConsIFTerFTerPSAntec ( x,y ) :
            with ( x )
                ( TriplIFTerFTerPSAntecVide ( ) : majtripletsUNION ( y,b ),
                  TriplIFTerFTerPSAntec ( n , p , m ) :
                    ConsIFTerFTerPSAntec'( TriplIFTerFTerPSAntec (
                    majfterfterUNION ( n , b ) , p , majsubseqUNION ( m , b )),
                    majtripletsUNION ( y,b ) )
                )
        )
    };



/* recoit  une liste a de type fterfter de la forme ( comp , csel )*
        et une liste b de type idfterme
   produit une liste de type fterfter egale a a modifiee comme suit :
        tout element de a tel que ( comp , csel ) = ( comp , as ( fterme1,
        fterme2 ) ) ou fterme1 represente un type, est remplace par un element
        de la forme ( inj ( UNION ( ... , fterme1 , ... ) , comp ) , fterme2 ),
        tous les autres elements de a restent inchanges
   utilise l'hypothese que "dsUNION" renvoie une liste d'un et un seul element
                                                                        */

fterfter majfterfterUNION ( fterfter a , idfterme b )
    { with ( a )
        ( VideFTerFTer ( ) : VideFTerFTer ( ),
          ConsFTerFTer ( x,y ) :
            with ( x )
                ( CoupleFTerFTerVide ( ) : majfterfterUNION ( y,b ),
                  CoupleFTerFTer ( m,n ) :
                    with ( n )
                        ( FAs ( p,q ) : ConsFTerFTer ( CoupleFTerFTer
                                ( FInj ( donfterme ( dsUNION
                                ( deidftermedonfster ( b ) , p ) ) , m ) , q )
                                , majfterfterUNION ( y,b ) ),
                          default : ConsFTerFTer ( x , majfterfterUNION (y,b) )
                        )
                )
        )
    };



/* recoit une liste a de fstermes
   renvoie le premier element de cette liste si elle est non vide,
            le terme vide sinon                                         */

fterme donfterme ( fsterme a )
    { with ( a )
        ( VideFTerme ( ) : FTermeVide ( ),
          ConsFTerme ( x,y ) : x
        )
    };
```

```
/* recoit une liste a de type listefterfter
   produit une liste de type listefterfter egale a a ou les elements de type
        fterfter sont modifies par "majfterfterUNION" pour tenir compte du
        passage sur un OR                                                    */

listefterfter majsubseqUNION ( listefterfter a , idfterme b )
    { with ( a )
        ( VideListeFTerFTer ( ) : VideListeFTerFTer ( ),
          ConsListeFTerFTer ( x,y ) :
                ConsListeFTerFTer ( majfterfterUNION ( x , b ) ,
                                    majsubseqUNION ( y , b ) )
        )
    };



/* recoit une liste a de type fterfter de couples de la forme ( comp , csel ),
   produit une liste de type fterfter egale a a dans laquelle les couples de
        la forme ( comp , ith ( x,y ) ) ont ete remplaces par des couples
        ( [ ] , y )                                                          */

fterfter majfterftervideforall ( fterfter a )
    { with ( a )
        ( VideFTerFTer ( ) : VideFTerFTer ( ),
          ConsFTerFTer ( x,y ) :
            with ( x )
                ( CoupleFTerFTerVide ( ) : majfterftervideforall ( y ),
                  CoupleFTerFTer ( m,n ) :
                    with ( n )
                        ( FNieme ( r,s ) : ConsFTerFTer ( CoupleFTerFTer
                                ( FSEmpty ( ) , s ) ,
                                majfterftervideforall ( y ) ),
                          default : ConsFTerFTer ( x ,
                                majfterftervideforall ( y ) )
                        )
                )
        )
    };



/* recoit une liste a de type fterfter de couples de la forme ( comp , csel ),
   produit une liste de type fterfter egale a a dans laquelle les couples de
        la forme ( comp , ith ( x,y ) ) ont ete remplaces par des couples
        ( [ comp | queue ] , y )                                             */

fterfter majfterfterconsforall ( fterfter a )
    { with ( a )
        ( VideFTerFTer ( ) : VideFTerFTer ( ),
          ConsFTerFTer ( x,y ) :
            with ( x )
                ( CoupleFTerFTerVide ( ) : majfterfterconsforall ( y ),
                  CoupleFTerFTer ( m,n ) :
                    with ( n )
                        ( FNieme ( r,s ) : ConsFTerFTer ( CoupleFTerFTer
                                ( FCons ( m , generqueue ( n ) ) , s ) ,
                                majfterfterconsforall ( y ) ),
                          default : ConsFTerFTer ( x ,
                                majfterfterconsforall ( y ) )
                        )
                )
```

```
        )
    };


/* recoit un fterme a de la forme csel ( vg )
   produit un fterme "_queuevgi" ou vg est tel que a = csel ( vg )
        et i est la "profondeur" de vg dans a                          */

fterme generqueue ( fterme a )
    { FVariable ( MajId ( "_queue" # donvargauchedefterme ( a )
        # INTtoSTR ( donprofondeurdsfterme ( a ) ) ) )
    };


/* recoit un fterme a de la forme csel ( vg )
   produit "vg" sous forme de string                                   */

STR donvargauchedefterme ( fterme a )
    { with ( a )
        ( FVariable ( x ) :
            with ( x )
                ( IdVide ( ) : "",
                  MinId ( m ) : m,
                  MajId ( m ) : m
                ),
          FSel ( x,y ) : donvargauchedefterme ( y ),
          FAs ( x,y ) : donvargauchedefterme ( y ),
          FNieme ( x,y ) : donvargauchedefterme ( y ),
          default : ""
        )
    };


/* recoit un fterme a de la forme csel ( vg )
   produit un entier egal au nombre de selections necessaires pour obtenir
        a a partir de vg                                               */

INT donprofondeurdsfterme ( fterme a )
    { with ( a )
        ( FVariable ( x ) : 0,
          FSel ( x,y ) : 1 + donprofondeurdsfterme ( y ),
          FAs ( x,y ) : 1 + donprofondeurdsfterme ( y ),
          FNieme ( x,y ) : 1 + donprofondeurdsfterme ( y ),
          default : 0
        )
    };


/* recoit une liste a de type fterfter de couples de la forme ( comp , csel ),
   produit une liste de type fterfter egale a a dans laquelle les couples de la
        forme ( comp , ith ( x,y ) ont ete remplaces par des couples
        ( queue , y )                                                  */

fterfter crefterfterprmark ( fterfter a )
    { with ( a )
        ( VideFTerFTer ( ) : VideFTerFTer ( ),
          ConsFTerFTer ( x,y ) :
            with ( x )
                ( CoupleFTerFTerVide ( ) : crefterfterprmark ( y ),
                  CoupleFTerFTer ( m,n ) :
```

```
                    with ( n )
                        ( FNieme ( r,s ) : ConsFTerFTer ( CoupleFTerFTer
                                ( generqueue ( n ) , s ) ,
                                crefterfterprmark ( y ) ),
                          default : crefterfterprmark ( y )
                        )
                )
        )
    };


/* recoit une liste a de type ifterfterpsantec
   produit la liste de type ifterfterpsantec egale a a mise a jour pour un
        passage sur un noeud FORALL i.e.
        pour tout element (( comp , csel )* , psantec , listefterfter ) de a
        - on genere un element correspondant a une suite vide:
                                listefterfter = VideListeFTerFTer ( ),
          tout ( comp , csel ) de ( comp , csel )* tel que csel = ith ( x,y )
          est remplace par ( [ ] , y ) , psantec = VideAntec ( ), et
        - on genere un element correspondant a une suite non vide :
          tout ( comp , csel ) de ( comp , csel )* tel que csel = ith ( x,y )
          est remplace par ( [ comp | queue ] , y ) , psantec = psantec,
          ajout a listefterfter du jeu de substitutions multiples adequat   */

ifterfterpsantec majtripletsSEQ ( ifterfterpsantec a )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ),
          ConsIFTerFTerPSAntec ( x,y ) :
            with ( x )
                ( TriplIFTerFTerPSAntecVide ( ) : majtripletsSEQ ( y ),
                  TriplIFTerFTerPSAntec ( n , p , m ) :
                        ConsIFTerFTerPSAntec ( TriplIFTerFTerPSAntec (
                        majfterftervideforall ( n ) , VideAntec ( ),
                                                VideListeFTerFTer ( ) ) ,
                        ConsIFTerFTerPSAntec ( TriplIFTerFTerPSAntec (
                        majfterfterconsforall ( n ) , p , ajoutasubseq ( m ,
                        crefterfterprmark ( n ) ) ) , majtripletsSEQ ( y ) ) )
                )
        )
    };


/* recoit une liste a de type listefterfter et une liste b de type fterfter
   produit une liste de type listefterfter egale a a, a laquelle on a ajoute a
   la fin l'element de type fterfter ( b )                                  */

listefterfter ajoutasubseq ( listefterfter a , fterfter b )
    { with ( a )
        ( VideListeFTerFTer ( ) :
                        ConsListeFTerFTer ( b , VideListeFTerFTer ( ) ) ,
          ConsListeFTerFTer ( x,y ) :
                        ConsListeFTerFTer ( x , ajoutasubseq ( y , b ) )
        )
    };


/* recoit une liste a de type ifterfterpsantec et un patome b,
   produit une liste de clauses ou chaque clause a pour consequent b dans
        lequel on a effectue les substitutions ad hoc et comme suite
        d'antecedants le psantec d'un element de a                          */
```

```
suiteclauses genereprogprolog ( ifterfterpsantec a , patome b )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideSuiteClauses ( ),
          ConsIFTerFTerPSAntec ( x,y ) :
            with ( x )
                ( TriplIFTerFTerPSAntecVide ( ) : genereprogprolog ( y,b ),
                  TriplIFTerFTerPSAntec ( n , p , m ) :
                    with ( m )
                      ( VideListeFTerFTer ( ) :
                            ajouterclauseaudebut ( genereprogprolog ( y,b ) ,
                                                   n , p , b ) ,
                        ConsListeFTerFTer ( i , j ) :
                            ajouterclausealafin ( genereprogprolog ( y,b ) ,
                                                  n , p , b , m )
                      )
                )
        )
    };



/* recoit une suite a de clauses,
         une liste b de type fterfter d'elements de la forme ( comp , csel ),
         une liste c d'antecedants psantec,
         un patome d,
   produit une suite de clauses egale a a precedee d'une clause dont les
        antecedants sont formes par c et le consequent est patome dans lequel
        on a effectue des substitutions en utilisant b                    */

suiteclauses ajouterclauseaudebut ( suiteclauses a , fterfter b , psantec c ,
                                    patome d )
    { ConsSuiteClauses ( Clause ( PConseq ( substvardegauche ( d , b ) ) , c ),
                         a )
    };



/* recoit une suite a de clauses,
         une liste b de type fterfter d'elements de la forme ( comp , csel ),
         une liste c d'antecedants psantec,
         un patome d,
         une liste de substitutions multiples e,
   produit une suite de clauses egale a a suivie d'une clause dont les
        antecedants sont formes par c et le consequent est patome dans lequel
        on a effectue des substitutions en utilisant b                    */

suiteclauses ajouterclausealafin ( suiteclauses a , fterfter b , psantec c ,
                                   patome d , listefterfter e )
    { a @ ConsSuiteClauses ( Clause ( PConseq ( substvardegauche ( d,b ) ) ,
      c @ substvardspsantec ( e , d ) ) , VideSuiteClauses ( ) )
    };



/* recoit un patome a et une liste b de type fterfter d'elements de la forme
        ( fterme1 , fterme2 ),
   produit un patome egal a a dont les ftermes tels qu'ils contiennent un
        fterme = fterme2 sont remplaces par le fterme1 correspondant      */

patome substvardegauche ( patome a , fterfter b )
    { with ( a )
        ( PAtomeVide ( ) : PAtomeVide ( ),
          PAtome ( x,y ) : PAtome ( remplacervar ( b,x ) ,
                                    remplacervars ( b,y ) )
```

```
        )
    };


/* recoit une liste a de type fterfter d'elements de la forme (fterme1,fterme2)
        et un fterme b,
   produit si il existe un fterme2 d'un element de a tel que fterme2 = b
        le fterme1 correspondant, produit le fterme b sinon                */

fterme remplacervar ( fterfter a , fterme b )
    { with ( a )
        ( VideFTerFTer ( ) : b,
          ConsFTerFTer ( x,y ) :
            with ( x )
                ( CoupleFTerFTerVide ( ) : remplacervar ( y,b ),
                  CoupleFTerFTer ( m,n ) : ( n == b ) ? m
                                                      : remplacervar ( y,b )
                )
        )
    };


/* recoit une liste a de type fterfter d'elements de la forme (fterme1,fterme2)
        et un fterme b de la forme FFoncteur ( ffoncteur , fsterme )
   produit un fterme egal a b ou pour tout fterme appartenant a fsterme,
        - si il existe un fterme2 d'un element de a tel que fterme2 = fterme
        alors fterme est remplace par le fterme1 correspondant,
        - sinon fterme reste inchange                                      */

fterme remplacervars ( fterfter a , fterme b )
    { with ( b )
        ( FFoncteur ( x,y ) : FFoncteur ( x , remplacersvars ( a,y ) ),
          default : b
        )
    };


/* recoit une liste a de type fterfter d'elements de la forme (fterme1,fterme2)
        et une suite b de ftermes,
   produit une suite de ftermes egale a b ou pour tout fterme s'il existe
        un fterme2 d'un element de a tel que fterme2 = fterme alors fterme
        est remplace par le fterme1 correspondant, fterme reste inchange sinon
                                                                           */

fsterme remplacersvars ( fterfter a , fsterme b )
    { with ( b )
        ( VideFTerme ( ) : VideFTerme ( ),
          ConsFTerme ( x,y ) : ConsFTerme ( remplacervar ( a,x ),
                  remplacersvars ( a,y ) )
        )
    };


/* recoit une suite a de type listefterfter de substitutions multiples ,
         un patome b,
   produit une suite de type psantec ou les substitutions multiples ont ete
        remplacees par un PAntec constitue du patome b donne dans lequel
        les substitutions necessaires ( trouvees dans les listes fterfter
        constituant la listefterfter a ) ont ete effectuees
```

```
psantec substvardspsantec ( listefterfter a , patome b )
    { with ( a )
        ( VideListeFTerFTer ( ) : VideAntec ( ),
          ConsListeFTerFTer ( x,y ) :
                ConsAntec ( PAntec ( substvardegauche ( b , x ) ) ,
                            substvardspsantec ( y,b ) )
        )
    };


/* recoit une liste a de type ifterfterpsantec dont chaque element a la forme
                        ( ( comp, csel )* , psantec , listefterfter )
        et deux identificateurs b et c,
   produit la liste a de type ifterfterpsantec ou dans chaque element, on a
        ajoute a la liste ( comp , csel )* l'element ( b , ith ( b# , c ) ).
   Le but de cette fonction est, lorsque l'on passe sur un FORALL, d'ajouter
        a la liste ( comp , csel )* un element correspondant a l'element
        representatif b et a la sequence guide c                          */

ifterfterpsantec ajoutelemforall ( ifterfterpsantec a , id b , id c )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ),
          ConsIFTerFTerPSAntec ( x,y ) :
            with ( x )
                ( TriplIFTerFTerPSAntecVide ( ) : ajoutelemforall ( y,b,c ),
                  TriplIFTerFTerPSAntec ( n , p , m ) :
                        ConsIFTerFTerPSAntec ( TriplIFTerFTerPSAntec (
                        ConsFTerFTer ( CoupleFTerFTer ( FVariable ( b ) ,
                        FNieme ( nvlid ( b ), FVariable ( c ) ) ), n ), p, m ),
                        ajoutelemforall ( y , b , c ) )
                )
        )
    };


/* recoit une liste a de type ifterfterpsantec dont chaque element a la forme
                        ( ( comp , csel )* , psantec , listefterfter )
        et une suite b de type psantec,
   produit une liste de type ifterfterpsantec egale a a si a est vide, egale a
        a ou devant le psantec de chaque element on a concatene b si a est non
        vide i.e. dans ce cas chaque element a la forme ( ( comp , csel )*,
        b @ psantec , listefterfter)                                      */

ifterfterpsantec ajoutpsantecdvtpsantec ( ifterfterpsantec a , psantec b )
    { with ( a )
        ( VideIFTerFTerPSAntec ( ) : VideIFTerFTerPSAntec ( ) ,
          ConsIFTerFTerPSAntec ( x,y ) :
            with ( x )
                ( TriplIFTerFTerPSAntecVide ( ) : ajoutpsantecdvtpsantec (y,b),
                  TriplIFTerFTerPSAntec ( n , p , m ) :
                        ConsIFTerFTerPSAntec ( TriplIFTerFTerPSAntec ( n ,
                        b @ p , m ) , ajoutpsantecdvtpsantec ( y,b ) )
                )
        )
    };


/* recoit une liste a d'identificateurs
        et une liste b de type idfterme d'elements de type ( idb , ftermeb ),
   produit une liste de type fterfter contenant pour chaque element ida de a
        un element de type ( fterme1 , fterme2 ) ou fterme1 = ida et fterme2 =
```

```
        le ftermeb pour le couple tel que ida = idb,
   produit une liste vide sinon                                          */

fterfter desiddonfterfter ( sid a , idfterme b )
    { with ( a )
        ( VideId ( ) : VideFTerFTer ( ),
          ConsId ( x,y ) :
            with ( deidftermedonfter ( b,x ) )
                ( FTermeVide ( ) : ConsFTerFTer ( CoupleFTerFTer (
                        FVariable ( x ) , FVariable ( x ) ) ,
                        desiddonfterfter ( y,b ) ),
                  default : ConsFTerFTer ( CoupleFTerFTer (
                        FVariable ( x ) , deidftermedonfter ( b,x ) ) ,
                        desiddonfterfter ( y,b ) )
                )
        )
    };



/* recoit deux identificateurs a et b ( res et op ),
          une suite d'identificateurs c ( args ),
          une liste d de type idfterme,
   produit une liste de type ifterfterpsantec d'elements de type
        ( ( comp , csel )* , psantec , listefterfter ) ou
                ( comp , csel )* = desiddonfterfter ( x,d )
                        ou x est la concatenation de a et c,
                psantec est la liste ne contenant qu'un seul antecedant du type
                PAtome ( y,z ) ou y est le fterme obtenu de a et z est le
                fterme obtenu de b et c,
                listefterfter = empty                                    */

ifterfterpsantec deidsiddonifterfterpsantec (id a , id b , sid c , idfterme d)
    { ConsIFTerFTerPSAntec ( TriplIFTerFTerPSAntec ( desiddonfterfter
        ( a :: c , d ) , ConsAntec ( PAntec ( PAtome ( FVariable ( a ) ,
        FFoncteur ( FId ( b ) , desiddonfsterme ( c ) ) ) ) ) ,
        VideAntec ( ) ) , VideListeFTerFTer ( ) ) , VideIFTerFTerPSAntec ( ) )
    };



/* recoit une suite a de type psantec
   produit une suite de type psantec ou tous les pantec de type PAtome tels
        que le premier fterme de PAtome soit une valeur booleenne aient cette
        valeur inversee ; laisse les autres pantec inchanges             */

psantec negpsantec ( psantec a )
    { with ( a )
        ( VideAntec ( ) : VideAntec ( ),
          ConsAntec ( x,y ) :
            with ( x )
                ( PAntec ( m ) :
                    with ( m )
                        ( PAtomeVide ( ) : ConsAntec ( x , negpsantec ( y ) ),
                          PAtome ( r,s ) :
                            with ( r )
                                ( FFoncteur ( u,v ) :
                                    with ( u )
                                        ( FBool ( w ) : ( w == true )
                                            ? ConsAntec ( PAntec ( PAtome
                                                ( FFoncteur ( FBool ( false ),
                                                VideFTerme ( ) ) , s ) ) ,
                                                negpsantec ( y ) )
                                            : ConsAntec ( PAntec ( PAtome
```

```
                                        ( FFoncteur ( FBool ( true ),
                                        VideFTerme ( ) ) , s ) ) ,
                                        negpsantec ( y ) ),
                                    default : ConsAntec
                                        ( x , negpsantec ( y ) )
                                ),
                            default : ConsAntec ( x , negpsantec ( y ) )
                        )
                    ),
                default : ConsAntec ( x , negpsantec ( y ) )
            )
    )
};
```